Cloudy Transactions: Cooperative XML Authoring on Amazon S3

Francis Gropengießer Stephan Baumann Kai-Uwe Sattler
Ilmenau University of Technology, Germany
first.last@tu-ilmenau.de

Abstract: Over the last few years cloud computing has received great attention in the research community. Customers are entitled to rent infrastructure, storage, and even software in form of services. This way they just have to pay for the actual use of these components or services. Cloud computing also comes with great opportunities for distributed design applications, which often require multiple users to work cooperatively on shared data. In order to enable cooperation, strict consistency is necessary. However, cloud storage services often provide only eventual consistency. In this paper, we propose a system that allows for strict consistent and cooperative XML authoring in distributed environments based on Amazon S3. Our solution makes use of local and distributed transactions, which are synchronized in an optimistic fashion, in order to ensure correctness. An important contribution of this paper is the evaluation of our system in a real deployment scenario upon Amazon S3. We show the strong impact of write operations to S3 on the transaction throughput. Furthermore, we show that fragmenting data increases write performance and reduces storage costs.

1 Introduction

Driven by the big players of the internet who want to better utilize their large data center infrastructures, cloud computing has often been viewed as the next paradigm shift or even big revolution in IT over the last few years. Normally the term cloud computing is understood as a mechanism to provide shared resources (computing capacity, software, data) on demand to multiple computers over the internet. Though the basic idea of centralization and hosting is not new, in fact it goes back to the mainframe era, cloud computing offers some new benefits and opportunities. First of all, availability and reachability guarantees given by the cloud provider by relying on highly redundant infrastructure simplifies building business-critical services. Second, the elasticity of resources allows to provide the illusion of infinite resources for customers to rent on demand. This is strongly related to the scalability: a customer can start with only a few machines and extend the number with a growing demand of resources. Together with a pay-per-use pricing model this helps to reduce the TCO dramatically and, therefore, is particularly interesting for small projects and companies that cannot afford large investments in infrastructure.

In cloud computing resources are provided at different levels, ranging from Infrastructure as a Service (IaaS) like Amazon Elastic Compute Cloud (short: EC2), where fundamental computing resources are rented, over Platform as a Service (PaaS) like Amazon Simple Storage Service (short: S3) or Google AppEngine, where customer-created applications can be deployed to the cloud, to Software as a Service (SaaS) like Salesforce or Google

Docs, where a provider's application already running in the cloud can be used over the Internet.

Apart from computing-intensive jobs, web and application hosting or scalable data analytics using MapReduce [DG04], storage and database services are an application class well-suited for running in the cloud. Examples like Dropbox, Ubuntu One as cloud file storage and Slideshare for sharing presentation slides already exist. These systems are all based on Amazon's S3, but also the deployment of full-fledged SQL databases using Amazon RDS or Microsoft SQL Azure is possible as well as using data management services as part of higher level SaaS solutions. Such database services are particularly promising for cooperative applications in which different users can share and edit a common set of documents and data. Google Docs or Microsoft Office Web Apps are well-known examples of these cooperative applications. Engineering design and media production processes could benefit from cloud-based database services in a similar way. In this work we consider media production using spatial sound systems based on the wave field synthesis [Ber88] as an example application. This technique enables the creation of more realistic surround sound for movies or concerts. During the design process the task of a sound designer is to animate static objects and to define their locations and movements as well as the characteristics of their surroundings. This way a listener can for example be given the impression to be in a cave or in a concert hall. The result of this design process is a scene description stored in an XML-based scene graph. However, apart from relying on a graph-based model and the usage of specialized authoring tools the approach we are going to present here is not limited to this application domain but rather usable for any kind of cooperative XML authoring.

In this context, cooperativeness means to allow multiple users to work at the same time on shared XML data and to exchange information in arbitrary directions without restrictions. This way it is possible for each user to adjust his own work to the current state of the project and the work of others. Furthermore, a cooperative authoring environment should support transactional semantics in order to ensure recoverability in case of transaction and system failures as well as strict consistency. The latter is needed to *(i)* guarantee that every user has the current state of the project and to *(ii)* prevent wrong design decisions of a user due to incorrect or outdated data.

Thus, the goal of our work is to build a cloud-based data management system for cooperative authoring of XML scene graphs which is as easily usable as a cloud storage system like Dropbox but supports transactional semantics. Building such a system on top of existing cloud platforms raises several questions. First of all, an appropriate storage abstraction and system have to be chosen, this can either be a low-level BLOB store like S3 or a full-fledged SQL database. Based on this decision a system architecture which maps a cooperative transaction model to the abstractions and operations of the underlying storage technology has to be designed. Amazon S3, for example, only supports eventual and read-after-write consistency (depending on the region where the service is provided) and atomic updates are restricted to single keys (tuples).

In this paper, we present such a system. Based on our previous work [GHS09], where we have developed a transaction model and an appropriate synchronization strategy for closely-coupled client/server-based workgroup environments, we discuss the design and implementation of a cooperative transactional authoring system for XML data on top of Amazon S3 as an example. The contribution of our work is twofold:

- We discuss the implementation of distributed optimistic synchronization of XML updates using S3 as storage layer.
- We present results of our experimental evaluation using a real deployment which shows that fragmenting data reduces storage costs and increases write performance. Furthermore, we show that S3 write performance has a strong impact on the overall transaction throughput.

The remainder of this paper is structured as follows. Section 2 summarizes related work. In Section 3, we briefly introduce our data and fragmentation model, the operations on tree structured data and Amazon S3. Furthermore, we sketch our optimistic concurrency control protocol developed in our previous work. In Section 4 we describe our proposed system model which enables cooperative processing of XML data using Amazon S3. In Sections 5 and 6 we discuss the applied transaction model as well as synchronizing and committing transactions in more detail. Section 7 reveals how strict consistency upon an eventually consistent storage layer can be achieved. In Section 8 we consider some aspects concerning transaction and system recovery. Section 9 presents the results of our evaluation followed by a conclusion in Section 10.

2 Related Work

The CAP theorem [FGC+97] states that only two of the three properties – consistency, availability and network partitioning tolerance – can be fulfilled in a distributed environment. Most of the current solutions show a lack of consistency guarantees in favor of a higher availability. Examples are Amazon S3, Amazon SimpleDB, Dynamo [DHJ+07], Yahoo PNUTS [CRS+08] and Google Bigtable [CDG+08]. Hence, without further extensions, these systems are not suitable for cooperative environments.

Not only in cooperative environments, but also in fields like business or e-commerce a strong need for strict consistency exists. With Amazon RDS, Microsoft SQL Azure, and Google AppEngine three companies tried to fulfill the consistency requirements of their customers. These systems provide strict consistency and transaction support. In [KKL10] an evaluation regarding performance, scalability, and costs of these systems (amongst others) can be found. However, in these systems transaction processing is either restricted to a certain entity group (Google AppEngine) or it is only supported on a single database instance (Amazon RDS, SQL Azure). Since we assume distributed data, support for distributed transaction processing is essential. Due to this, these systems are not applicable to our use case without further extensions.

The endeavor of building databases upon cloud storage systems is not new. Our work is mainly inspired by [BFG+08], [DAA10a] and [DAA10b]. In [BFG+08] the design of a database system on S3 is described. The authors address in detail, how atomicity, consistency and durability of transactions can be fulfilled. Concerning isolation they argue that protocols, like the BOCC (backward-oriented concurrency control [KR79]) protocol, can only be partly implemented, as they need a global transaction counter, which might become a bottleneck. For this reason we use timestamps for validation purposes, which are assigned by every transaction manager separately. This is possible, because the transaction managers run on virtual machines within an IaaS layer, where synchronized clocks can be assumed. We discuss this in more detail in Section 6.

The authors of [DAA10a] propose a scalable and elastic layered system approach, called ElasTraS, for transaction processing upon S3. They assume partitioned data as we do. In order to process transactions across different partitions they use minitransactions. Minitransactions were first used in Sinfonia [AMS$^+$09]. They provide only very restricted transactional semantics. Precisely spoken, every data object accessed by a minitransaction must be specified before the minitransaction is started. This is almost impossible in design environments we consider. However, minitransactions can be used in order to implement optimistic concurrency control [AGS08].

In [DAA10b] the authors propose a transactional system approach for collaborative purposes, e.g., collaborative editing. They support transactions on so-called key groups, which can be established dynamically. Amongst others, the authors describe an implementation of their system on top of existing key-value stores, e.g., Bigtable. However, resolving the problem of how to ensure strict consistency on top of a weak consistent key-value store is left as future work.

With our endeavor of synchronizing transactions in distributed environments, we also intersect with current research projects in the field of distributed transactional memories. In these systems, a preferred solution for distributed transaction processing is to run transactions on a single site and move the accessed objects between different sites [HS07, ZR09]. Although, this is an interesting approach for future work, frequently moving fragments between different buckets might decrease the overall system performance.

Besides the approach of building a cooperative system upon an existing cloud storage layer, the development and hosting of a tailored cloud storage system is another possibility. Thereby, systems like Scalaris [SSR08] and Chubby [Bur06], which use the Paxos commit protocol [Lam02] to guarantee strict consistency, could be used as an entry point. Furthermore, separating transaction processing and data access, like proposed in [LFWZ09], is an interesting research area. It should be considered for future work.

3 Preliminaries

In order to understand the approaches proposed in this paper, we briefly sketch some basic concepts needed. We start with a characterization of the data model before going into details on the tree operations and concurrency control. A detailed description of all concepts can be found in [GHS09].

3.1 Data Model and Fragmentation Model

Basically, we assume XML data as a tree structure following the tailored DOM (short: taDOM) specification [HH03]. There, a tree consists of *nodes* with unique *node ids*, *node labels* and *node values*. Nodes are connected via *directed edges* denoting parent-child relationships. Figure 1 shows an example.

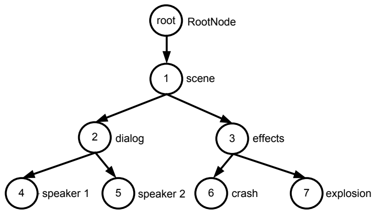


Figure 1: Example XML Tree

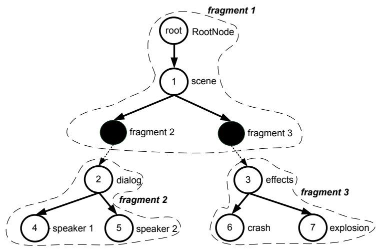


Figure 2: Fragmented XML Tree

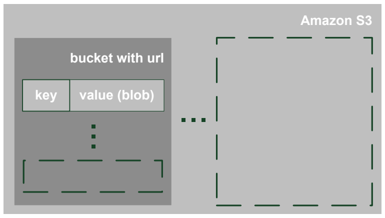


Figure 3: Amazon S3

Amazon S3 (see Figure 3), on the other hand, just provides a simple key-value store. Data is stored as blobs under unique keys. Furthermore, key-value pairs are organized in buckets, which are containers with a unique url. The task is to map our tree model to Amazon's storage model. Therefore, we serialize tree data as simple strings and store these under unique names in S3 buckets. Serializing a whole tree results in a very large string. This leads to less read/write performance from/to S3 and hampers transaction throughput as we show later. Another possibility is to store every single node as a string using a unique key. However, this would increase costs, as multiple *get* operations have to be performed in order to retrieve partial tree data. Hence, a fragmentation model – as a trade off between costs and performance – is necessary. Figure 2 shows an example: The XML tree is shredded into three fragments with unique ids *fragment 1*, *fragment 2* and *fragment 3*. In order to reconstruct the whole tree, we introduce special nodes *fragment 2* and *fragment 3*, which reference the corresponding tree fragments. The reconstruction is performed in a top-down fashion. This approach is sufficient for our use case.

Fragmentation of a tree can be performed with respect to different aspects. Regarding our use case, sound production with spatial sound systems, a whole scene graph could be fragmented based on cooperation on data units. Assuming that in a sound studio different teams exist, e.g., one for effects and one for speech, then a possible fragmentation could be dialogs and effects as most in cases each team works on their own data set. However, there are other possibilities, for example fragmentation with respect to certain efficiency criteria. Summarizing, it depends on the application which fragmentation strategy should be used.

3.2 Operations

Our overall goal is to enable cooperative processing of tree data. In order to achieve this, we specified semantic tree operations. These allow for fine-grained conflict specification and hence support cooperative synchronization protocols.

In order to read a fragment and the contained tree, we defined a simple *read(fragment id)*. The manipulation of XML data is possible with the help of some update operations. An *edit(fragment id, n, new node value)* (E) is used to assign a new *node value* to a node *n*. The operation *insert(fragment id, n, sub tree)* (I) adds a new *sub tree* to node *n*. Thereby, a new edge between *n* and the root node of the *sub tree* is inserted. The operation *move(fragment ids, n, m)* (M) assigns node *m* as new *parent* of node *n* so that the existing edge *(p, n)* between *n*'s old parent *p* and *n* is removed and the new edge *(m,n)* is inserted. A *delete(fragment ids, n)* (D) removes node *n*, the edge *(p, n)* between *n* and its parent *p* and *n*'s children from the tree. Note that move and delete can affect several fragments.

Amazon S3 just offers really simple operations in order to read and update data, i.e., *get* and *put* for an object (blob).

The mapping of semantic tree operations to simple get and put operations is performed as follows. A fragment to be read is retrieved from S3 with the help of a get operation. Updates on the tree are performed externally with the help of semantic tree operations. In the end the changed fragment is written back to S3 using a put operation.

3.3 Synchronization Model

Simply using S3 operations in order to perform updates on data would have negative consequences regarding transaction synchronization, as a fine-grained conflict specification is impossible. Consider, for example, two authors who are working on the example scene graph in Figure 1. One author changes a dialog element and the other one changes an effects element. Both operations do not influence each other, because they are executed on different parts of the scene. However, both operations result in one put operation of the whole scene graph each and, hence, have to be considered as conflicting. Using semantic tree operations allows fine-grained conflict specifications. Furthermore, we consider functional dependencies between read and update operations. This leads to the following conflict definition:

Definition 3.1. *Two operations o_i and p_j, which belong to transactions t_i and t_j respectively, are conflicting, iff they are incompatible according to the compatibility matrix shown in Table 1. If both are $read(fragment\ id)$ operations, they are not conflicting. Without loss of generality, we assume that o_i is a $read(fragment\ id)$ operation and p_j is an update operation. Then both operations are conflicting, only iff o_i is followed by an update operation u_i which itself depends on o_i. Otherwise, they are not conflicting.*

Table 1 shows the compatibility of the update operations with respect to the nodes or edges our operations consider. Thereby, $\surd$ states that the operations are fully compatible, $-$ that they are not compatible at all, and $+$ that they are only compatible if the tree they are performed on is considered unordered. Compatible operations are not in conflict and can be executed in parallel. Read operations are not further considered. In our use case it is sufficient to investigate update operations in order to detect conflicts between different transactions.

	E	I	M	D
E	$-$	$\surd$	$\surd$	$-$
I	$\surd$	$+$	$+$	$-$
M	$\surd$	$+$	$+$	$-$
D	$-$	$-$	$-$	$-$

Table 1: Compatibility of Update Operations

In order to synchronize transactions, several approaches exist, each depending on the considered scenario. Pessimistic protocols, e.g. locking, are useful for working environments with high conflict rates. Optimistic protocols are applied in situations where a lower conflict rate is assumed. In our scenario we assume that in most cases authors are working on different parts of the same scene. Furthermore, the semantic tree operations lead to a lower conflict probability, as shown in Table 1. Hence, we apply an optimistic synchronization protocol, which is based on the well-known BOCC protocol. Here, we only present a short summary, more detailed information can be found in [GS10]. Like the traditional BOCC, we divide a transaction into three phases – read, validate and persist. Within the read phase all operations are performed on local copies of the data. In the validation phase the transaction is checked against all successfully committed transactions that interleaved the considered transaction. After successful validation the changes are stored persistently. Like the inventors of the BOCC protocol we assume validate and persist phase as an indivisible unit.

The interesting part of the protocol is the validate-persist phase. First, we summarize the validation criterion, as this is the main point where our protocol differs from the traditional BOCC protocol. For this purpose we introduce the notion of $UpdateSets$. An $UpdateSet_{T_i}$ contains all update operations a transaction T_i has performed. Update operations may or may not conflict, as it is shown in Table 1. Let T_j be a transaction with transaction number (or timestamp) $tn(T_j)$. T_j is successfully validated, iff $\forall T_i, tn(T_i) < tn(T_j)$ one of the following conditions holds:

1. T_i has completed its validate-persist phase before T_j starts its read phase.
2. $UpdateSet_{T_i}$ is not in conflict with $UpdateSet_{T_j}$ according to Table 1.

The first case follows the traditional BOCC approach. Hence, if T_i and T_j are executed serially, T_j is validated successfully. The second case implies that validation is not successful if the transactions conflict in their $UpdateSets$. Otherwise, if the transactions used different data items, serializability is preserved and T_j is validated successfully. The validation criterion works, because we assume less functional dependencies between read and update operations. In [GS10] we show that this validation criterion leads to fewer transaction aborts, even in case of environments with high conflict rates.

Finally, we briefly show correctness of our protocol.

Theorem 3.1. *The proposed validation criterion produces serializable schedules. Hence, every local transaction manager following this protocol produces locally serializable schedules.*

Proof. We construct a conflict graph G where nodes denote transactions and directed edges denote conflicts between these transactions [WV01]. A conflict is defined as in Definition 3.1. Iff G is acyclic, the schedule is serializable. Otherwise, it is not serializable. Now assume G is an acyclic graph of the form $(... \rightarrow\ t_i \rightarrow ... \rightarrow t_k\ \rightarrow ...)$. This implicitly means that all transactions were successfully validated and committed (Aborted transactions are not contained). By inserting a running transaction t_j this graph becomes cyclic. This means there must be at least one conflict directed from t_j to t_i and one directed from t_k to t_j, where $k = i$ is possible. However, if there are conflicts detected during the validation phase where t_j is involved in, then t_j is aborted. Hence, the graph stays acyclic. Note that validation is performed indivisibly. This means, only one transaction is validated at a certain time. Hence, if t_j is conflicting with another running transaction t_z, these conflicts are detected during validation of t_z (in case t_j has successfully been committed). □

4 System Architecture

In Section 3 we discovered discrepancies between our requirements and what is provided by S3. Namely, these are the different data models and the different operation sets. Furthermore, we want to enable strict consistent and cooperative transaction processing upon S3 which by itself provides at most read-after-write consistency for put operations of new data. Hence, we need to define a system model which maps our data model and operations to S3 and provides strict consistency for the application.

Our proposed system architecture is shown in Figure 4. We chose a layered approach on top of Amazon S3.

Amazon S3 is organized in buckets, as described in the last section. In the context of media production with spatial sound systems we assume that all scene data (fragment sets) is partitioned over a set of buckets. This way every fragment belongs to exactly one bucket. The partitioning is performed in a way that a bucket contains the amount of scene data which is shared within a certain group of authors (clients). This is a natural approach with respect to huge movie projects, where we can find several teams for, e.g., sound effects or music.

Clients possess fragment caches in which they store local copies of the fragments they want to work on in form of trees. Every update operation is first executed on these copies. Only after the successful validation of the corresponding transactions the updates are stored persistently in Amazon S3. The advantage is that in case of a transaction abort no compensation (or version restoring) has to be performed in order to remove inconsistencies from S3. Clients are only allowed to read and update fragments via transaction managers within a transactional context.

The **VM layer** is situated between S3 and the clients. This layer consists of a set of virtual machines, which are started when needed and closed if they get dispensable. The virtual

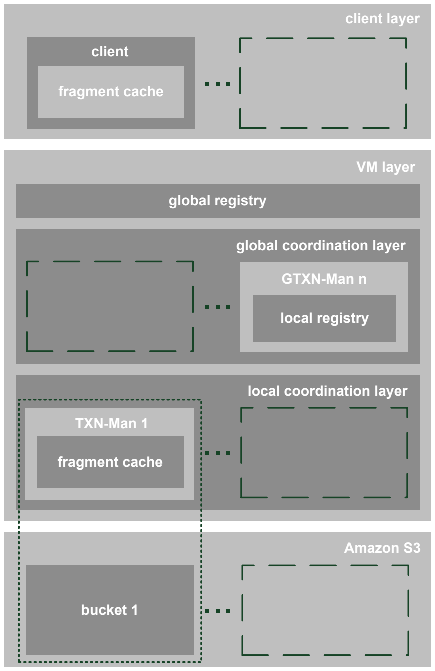


Figure 4: System Architecture

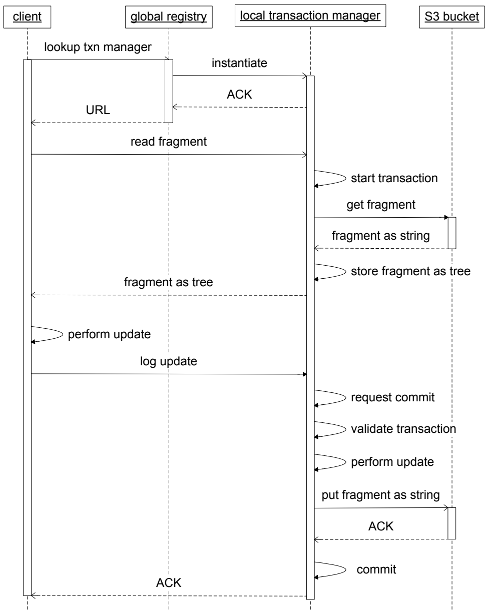


Figure 5: Sequence Diagram: Client Request

machines execute three kinds of services - local transaction managers (TXN-Man), organized in the local coordination layer, global transaction managers (GTXN-Man), organized in the global coordination layer and a single global registry. (Note that several services can run on a single virtual machine.) Every local transaction manager is exactly assigned to one single bucket and vice versa. It retrieves (get operation) copies of the scene data from the bucket it is assigned to, parses the XML string and stores this data in form of a tree in its fragment cache. The copies are written back to the bucket (in form of an XML string using the put operation) after manipulation. A local transaction manager is started/executed if at least one client intends to work on a certain bucket, otherwise it is stopped. Occasionally it is required that several groups of authors synchronize their work. This may result in concurrent access to different buckets. In order to handle this, global transaction managers are needed. Their task is to route client operations or requests to the responsible local transaction managers. However, they are not allowed to access S3 buckets. Hence, they have a local registry where they store which local transaction manager is responsible for which fragment of the tree. A global transaction executed by a global transaction manager is decomposed into several sub transactions. These are then executed by the responsible local transaction managers. Note that the sets of local transaction managers accessed by different global transaction managers do not necessarily have to be disjoint in order to allow for correct transaction synchronization. We show this in Section 6. Similar to local transaction managers, global transaction managers are only instantiated when needed. In addition to the transaction managers a single global registry is maintained. It is responsible for starting, stopping, and monitoring transaction managers and their corresponding virtual machines. The registry knows which fragments are stored in which bucket. This is required in order to route client requests to the right transaction manager.

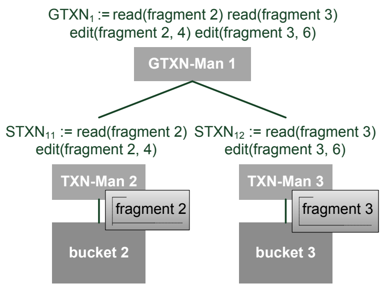
Next we give a brief example of how client requests are treated (Figure 5). Assume an author knows a fragment he wants to work on. It could have been assigned to him by the supervisor in his team. The client sends a lookup request to the global registry in order to get the address of the responsible transaction manager. In the case that an appropriate transaction manager is not yet running, it is instantiated. After obtaining the transaction manager's address, the author is able to start his work. Every atomic unit of work is encapsulated into a transaction. The client retrieves the fragment copy the author wants to work on with the help of a read operation. Thereby, a new transaction is started implicitly by the transaction manager. The transaction manager retrieves a copy of the fragment from S3 and stores it locally if it is not yet in the cache. Thereafter, the client (author) performs an update on the local copy which is also logged by the transaction manager. After a certain number of update operations were performed, the transaction manager tries to commit the transaction. Therefore, the transaction enters the indivisible validate-persist phase. If validation is successful, the logged updates are applied to the local fragment copy and the updated fragment is written back to the S3 bucket. The validation and committing of transactions is described in more detail in Section 6.

We briefly discuss two possible APIs which enable application developers to connect to the proposed system. Using the first approach, the client application directly communicates with the VM layer (see Figure 4). However, this way application programmers have to care about fragment caching and have to use the tree operations proposed in Section 3.2. Following the second approach, application programmers use a low-level client provided by us. This client software cares about fragment caching. Additionally, it maps application operations onto the tree operations used by our system. Hence, it would be possible to use, for example, XPath or XUpdate for XML processing on the application side. These operations are then transparently mapped onto semantic tree operations.

5 Transaction Model

In [GHS09] we defined a transaction model for cooperative XML processing. However, it has to be simplified for this use case, because of the following reasons. First, managing root transactions in such a distributed environment causes too much overhead. They are simply running too long. During this time a client could change the transaction manager and the transactional context has to be exchanged between the participating transaction managers. Second, splitting up operations on sub trees (like deleting a node with its corresponding children) into sub transactions (containing operations on nodes and edges) for recovery purposes is unnecessary, because S3 does not provide versioning of nodes and edges but only of fragments. Hence, a fine-grained recovery is impossible at all.

The resulting transaction model looks as follows. A transaction starts with a set of *read(fragment id)* operations followed by a set of update operations. Thereby, every data item that is affected by these update operations has to be read. This means we do not allow so-called "blind writes". Hence, we can assure that every author knows the current state of the project before he starts to make changes. This is necessary to allow for coopera-



GTXN1 := read(fragment 2) read(fragment 3) edit(fragment 2, 4) edit(fragment 3, 6)
GTXN-Man 1
STXN11 := read(fragment 2) edit(fragment 2, 4)
STXN12 := read(fragment 3) edit(fragment 3, 6)
TXN-Man 2
fragment 2
bucket 2
TXN-Man 3
fragment 3
bucket 3

Figure 6: Example Transaction Execution

tive working. We illustrate our transaction model with a short example. Assume an author wants to balance the volumes of a crash and a speaker in the fragmented scene in Figure 2. Therefore, he reads fragment 2 and fragment 3 and performs two edit operations, one on node 4 and another on node 6. Figure 6 shows the resulting transactions in case that both fragments are stored in different buckets. In order to ensure atomicity across two buckets, a global transaction manager is used. All operations performed by the client are encapsulated in the global transaction $GTXN_1$. The global transaction manager knows the responsible local transaction managers and, hence, performs a decomposition of the global transaction into two sub transactions ($STXN_{11}$ and $STXN_{12}$). In case both fragments are stored in the same bucket, no global transaction manager is needed. Then, the client communicates directly with the responsible local transaction manager and the example transaction is not decomposed at all.

Finally, the transaction boundaries (begin and commit) have to be determined. On one possibility they are explicitly specified by the designers or the application programmers. However, specifying transactions in design environments is not an easy task. In contrast to, for example, banking applications, where predefined transactions exist, e.g., for withdrawal, in design environments, the transactions are constructed during the interaction of a designer with the system. Usually, designers are not interested in specifying transactions. Hence, this task must be performed transparently by the system. In [GS10] we described an appropriate method, which automatically determines transaction boundaries based on a user defined degree of cooperation and the importance of the executed update operations.

6 Transaction Validation and Commit

In the last section, we described how it is determined, when a transaction is started and when it should be committed. Now, we clarify how transactions in this distributed environment are committed in order to guarantee atomicity and correctness.

In case an author works with XML data that belongs to a single bucket, there is no need for further investigation, because only a local transaction is executed at a single site. Here, validation is performed, and in case of success, the transaction is committed and the changes are stored persistently in S3. Correctness is guaranteed, because of our validation scheme described in 3.3.

However, in case an author works on scene data that is distributed across several buckets, a global transaction manager and several local transaction managers are involved. The task is, *i)* to synchronize the global and local transactions against other global and local transactions in order to assure correctness and *ii)* to commit a global and all its local transactions entirely in order to guarantee atomicity. A common approach for these problems is the adaption of the two phase commit protocol [WV01] to optimistic concurrency control protocols. If a distributed transaction shall be committed, the global transaction manager becomes the *coordinator* and sends PREPARE messages to all affected local transaction managers. This message is a request for validation. Every local transaction is validated at its corresponding site. If validation is successful, a READY for commit message is sent to the coordinator. Otherwise, the local transaction manager answers with an ABORT message. If all local transactions voted for commit, the coordinator sends COMMIT messages to all participating local transaction managers and the whole distributed transaction is committed and the changes are stored persistently. If at least one transaction fails, all local transactions are aborted by the coordinator with the help of ABORT messages, followed by an abort of the global transaction itself.

Global transaction managers support parallel validation of several global transactions. However, local transaction managers treat the validate-persist phase including commit as indivisible unit, leading to serial validation.

The two phase commit protocol described above can lead to deadlocks between global transactions, because of cyclic wait-for graphs between sub transactions waiting for the entrance into the validate-persist phase. Two alternatives are known to deal with deadlock situations. First, avoid deadlocks - all transactions are validated simultaneously. If at least one sub transaction is not able to enter validation phase, the global transaction is aborted and, hence, all sub transactions are aborted, too. Second, deal with deadlocks - in the literature, e.g. [WV01], several mechanisms to detect and handle deadlocks are described.

In order to determine the serialization order of global and local transactions global transaction counters or timestamps are necessary. Since using a global transaction counter can become a bottleneck in highly distributed systems (as mentioned in Section 2), we chose timestamps for our solution. These can be assigned in a distributed manner. However, the precondition is, that clocks are synchronized at all transaction managers. In commonly known distributed systems this is hard to achieve. However, we assume our system (VM-layer) to be running in a data center on top of virtualization software like Eucalyptus, which also needs synchronized clocks. Current versions of the network time protocol provide time accuracy in the range of less than 10 milliseconds for local networks. We believe this is sufficient for our use case. However, for the unlikely case, that accuracy is too low, the approach described in [AGLM95] could be adapted and applied. However, this is beyond the scope of this paper.

Finally, we show that the proposed protocol is correct.

Theorem 6.1. *If every global transaction T_i follows the proposed commit protocol, the resulting schedule is globally serializable.*

Proof. Due to our proposed validation method, all locally executed transactions are serializable. Now, assume a schedule of committed global transactions T_k, $k \in \mathbb{N}$. We construct a global conflict graph S. A conflict is defined according to Definition 3.1. Iff S is acyclic,

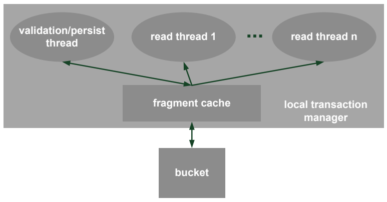


Figure 7: Local Transaction Manager Threads

the schedule is globally serializable. Assume, without loss of generality, that S contains a cycle of the form $(T_i \rightarrow ... \rightarrow T_k \ \rightarrow ... \rightarrow T_i)$. Hence, there must be a (direct or indirect) conflict directed from t_{ir} to t_{ks} and one directed from t_{ku} to t_{iv}, where t_{mn} is the n-th sub transaction of T_m executed at the n-th local transaction manager. Since all sub transactions were successfully validated and committed at the corresponding local transaction managers, T_i and T_k must have been serialized in opposite directions at different local transaction managers. This is not possible - at a local transaction manager site the validation phase is indivisible. Hence, only one transaction can be performed at a certain point in time. If a sub transaction entered the validation phase, it cannot leave it until its corresponding global transaction commits. (And a global transaction cannot commit if at least one sub transaction is not validated successfully.) This means, validation (serialization) order can only be unidirectional. □

Note, that we did not assume, that the validation phase at the global transaction manager site must be indivisible. Hence, validation can be performed in parallel. This also means, it does not matter, if two global transactions are validated in parallel on the same or different global transaction managers.

7 Ensuring Consistency and Enhancing Read Performance

In order to understand how transaction throughput can be increased, we give some insights into the implementation of a local transaction manager. Figure 7 shows a local transaction manager with its corresponding S3 bucket. We implemented the transaction manager multi-threaded. We start exactly one validation/persist thread in order to serialize update operations and an arbitrary amount of reading threads, which are only allowed to read from the fragment cache. (In order to allow for real parallel reading, the number of read threads should follow the number of assigned CPU cores.) This implementation allows to serve a lot of client read requests in parallel.

The fragment cache is limited. Hence, it is possible that a cache resident fragment has to be replaced by another fragment that is immediately needed. In the literature, several page replacement strategies usable for our case are described. Thus, we do not consider fragment replacement any further. However, one problem remains with respect to fragment replacement. Assume a worst-case scenario where a fragment is updated and written to the corresponding S3 bucket. Right after this, it is replaced in cache by a different fragment and then immediately requested again and, thus, read from S3. As already said, Amazon S3 provides no strict consistency. So, if a fragment is written and read immediately thereafter, it is possible that an older version is retrieved. In order to prevent this, we make use of the entire versioning feature S3 provides. An Amazon S3 put operation returns the version ID of the written object. We just store the returned version ID in the local transaction manager. This ID is updated every time a put of the considered fragment is executed. When the fragment is read, this version ID is passed to the Amazon S3 get operation. If the requested version is already available, it is returned. Otherwise, the transaction manager waits for some milliseconds and tries retrieving the fragment again.

8 Recovery

Concerning recovery we have to distinguish between transaction and system recovery.

Transaction recovery is necessary to guarantee atomicity in case of failures. Aborted transactions have to be rolled back to not have any effect on the persistent storage. In our approach transaction recovery and synchronization are closely coupled. In case of local transaction processing the use of our extended BOCC protocol ensures that only committed changes are stored persistently. Dirty updates are only performed on the client side during the read phase. Furthermore, cascading aborts are avoided as a transaction may only depend on previously committed transactions. (A transaction T depends on a transaction S if there is a directed conflict (Definition 3.1) from S to T.) For global transactions the variation of the two phase commit protocol ensures atomicity and durability. In cases where local or global transactions are aborted during read or validation phase, the affected transactions can just be aborted, as no changes have been stored persistently. Only the cases, where local transactions are aborted during the persist phase are a bit more intricate. However, Amazon S3 provides versioning feature, which is helpful in these situations. Even if a transaction is aborted after the put operation has been successfully performed, the previous version can easily be restored. Every put operation produces a new version of the fragment with a unique version ID. This version can simply be deleted by using the version ID. Since global transactions are not allowed to access S3 there is no need for further investigations.

A main problem of the original two phase commit protocol is that a deadlock situation occurs if the coordinator (global transaction manager) fails. Then, the local transaction managers are waiting for notifications infinitely. A possible solution is the introduction of time outs. In our system a global registry is running, which monitors all transaction managers. This way the registry can inform all affected local transaction managers if the global transaction manager fails and all local transactions are aborted. If a local transaction manager fails, the responsible global transaction manager is informed, which then reacts accordingly.

The base for ensuring system recovery is the global registry, which, as already mentioned, instantiates, monitors, and restarts virtual machines and transaction managers. In order to guarantee that the global registry is always available and also for load balancing purposes we use backup instances of this service. If a virtual machine or a transaction manager fails, it is detected by the global registry and the affected machines and managers are restarted. There is only a need for simple recovery steps after a local or global transaction manager was restarted. All running transactions have been aborted and all changes of committed transactions are stored in the corresponding S3 bucket. However, there could be uncommitted changes in the bucket, as previously described. They can easily be removed by using the version IDs. After a local transaction was committed, the produced version ID is communicated to the global registry (during monitoring). If a local transaction manager is restarted, the last stored version ID is communicated to the local transaction manager, which simply retrieves a list of version IDs from the concerned bucket. If the last known version ID is not the current one in the list, the current version is simply deleted by using its version ID. Recovery for global transaction managers is much easier. A global transaction manager retrieves the lists of fragment IDs from its local transaction managers and can this way assure a proper routing of new client requests.

The versioning feature can also be used to allow undo operations. If a client wants to restore an old version of a fragment, it just retrieves all versions with a listing command via the global and/or local transaction manager. Then, it chooses one version and deletes all other versions. However, the undo feature has to be refined in order to enable for example access control. We consider this in future work.

9 Evaluation

In this section we evaluate the proposed system with respect to performance, applicability, scalability, and costs. We measure S3 performance and costs with respect to different fragment sizes in order to show that fragmentation reduces costs and enhances performance. Furthermore, we determine the local and global transaction throughput in order to show that the proposed system is applicable to cooperative media production scenarios. Concluding, we give a simple cost formula to helps calculate the overall S3 costs and show that the proposed system is scalable within Amazon EC2.

In order to determine S3 costs we use prices for the region EU/Ireland from the Amazon S3 web page:

- $StorageCosts$ = \$0.15 for storing 1 GB of data
- $GetCosts$ = \$0.01 for 10000 get operations
- $PutCosts$ = \$0.01 for 1000 put operations
- $DataInCosts$ and $DataOutCosts$ = \$0.15 for transferring 1 GB of data

The versioning feature is charged via storage costs. Every update operation leads to a full copy of the data item and hence increases storage costs.

Costs and Performance

As mentioned in Section 3.1, there are two possible extreme solutions for storing the tree data as blob objects – one blob per node or one blob for the whole tree. Here we outline how S3 costs and performance evolve between these two extrema and we show that fragmenting data reduces S3 costs and enhances data processing performance.

In order to measure the results we used an S3 bucket in the region EU/Ireland with the versioning feature enabled. Versioning is necessary for transaction recovery as described in Section 8. The overall storage costs for a fragment of size $FragSize$ in MB, which is updated n times in an EU/Ireland bucket with enabled versioning feature, are calculated as follows: $TotalStorageCosts = FragSize * (StorageCosts/1000) * (n + 1)$. Furthermore, the whole get costs of retrieving all fragments of a tree are: $TotalGetCosts = GetCosts/10000 * NumberOfFragments$.

For our first measurement we consider a tree with a size of 500 KB. We fragment the tree step by step and consider get performance (the time needed for retrieving the whole data tree) and put performance (the time needed for putting a single fragment into the bucket). The tests were performed on an EC2 instance in region EU/Ireland as well as an instance in our institute. The results are shown in Figure 8. As expected, put performance increases with decreasing fragment size as decreasing fragment size avoids transferring of unchanged parts of the whole tree. The get performance decreases with an increasing degree of fragmentation. This can be explained by the higher number of get operations needed to retrieve the whole data tree and the associated higher number of expensive http requests sent to S3. Overall, results measured in EC2 are better than those measured externally. The reason is better infrastructure with less latencies and higher bandwidth.

Next, we measure $TotalStorageCosts$ and $TotalGetCosts$. Therefore, we, again, consider a tree with a size of 500 KB, which is stepwise fragmented. We assume that always the whole tree is retrieved from S3 and a single fragment of considered size is updated and written back to S3 ten times. The results are shown in Figure 9. As expected, the get costs increase and storage costs decrease with decreasing fragment size as decreasing fragment size avoids versioning of unchanged parts of the whole tree.

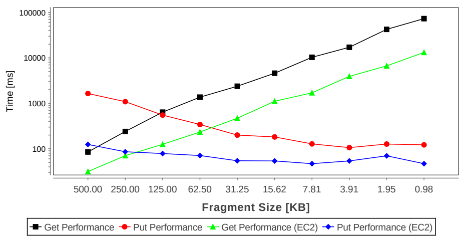


Figure 8: Put and Get Performance

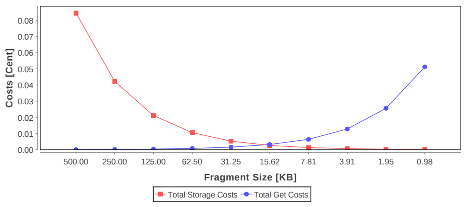


Figure 9: Total Storage Costs and Total Get Costs

Transaction Throughput

The goal of this part of the evaluation is to show that using the proposed system architecture and synchronization protocol results in sufficient transaction throughput. Furthermore, we show that fragmentation has some impact on transaction throughput.

Before presenting experimental results we first introduce the notion of *conflict rate*. The conflict rate determines the number of transactions which are conflicting with each other in the whole transaction set (for details see [GS10]). The set of conflicting transactions is determined based on the validation criterion of our new approach (conflicting $UpdateSets$). The tree the transactions are executed on is considered to be unordered.

We measure the number of successful committed transactions per minute ($TransactionThroughput = NumOfCommittedTxns/duration$, where $duration$ is the measured time in minutes for executing the whole transaction set) at the local/global transaction manager site. Thereby, we analyze the following scenarios:

1. Local transaction throughput depending on fragment size with fixed conflict rate of 25%.

2. Global transaction throughput depending on fragment size with fixed conflict rate of 25%.

For our test scenarios we used a simulation where a randomly generated set of transactions is executed on a randomly generated tree that follows our data model specification of Section 3.1. The tree is again fragmented step by step.

In test scenarios with local transactions the setup consisted of a local transaction manager upon an S3 bucket in the region EU/Ireland. In order to measure global transaction throughput the setup consisted of two local transaction managers upon two S3 buckets in the same region and one global transaction manager. All transaction managers are running on the same virtual machine. Local transactions consist of a set of read fragment operations and exactly one update operation. Global transactions consist of exactly two local transactions. It should be noted that in the local transaction scenario no global transactions are executed. Furthermore, in the global transaction scenario no local transactions that do not belong to global transactions are executed. Since duration of the read phase of a transaction is unpredictable (because it depends on the working time of an author), we assume a read phase duration between one and two seconds. Figures 10 and 11 show the measured results in case of transaction manager deployments in EC2 (region EU/Ireland) as well as in our institute. In both cases we measured transaction throughput when writing updates to

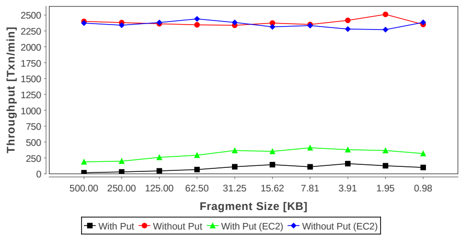


Figure 10: Local Transaction Throughput

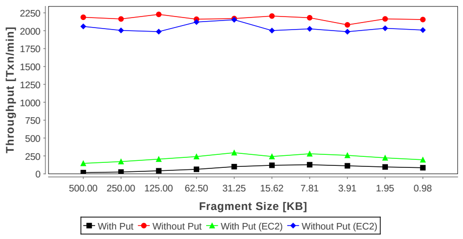


Figure 11: Global Transaction Throughput

S3 and in a second experiment only writing to the fragment cache. Basically, the measured transaction throughput is sufficient for typical cooperative design environments. However, write performance of S3, obviously, has a big impact on transaction throughput. In future work we have to think about sophisticated update caching mechanisms in order to save put operations. Furthermore, although writing a smaller fragment is faster than writing a large fragment to S3, transaction throughput slightly decreases with increasing fragmentation degree. We get this effect as with decreasing fragment size update operations affect

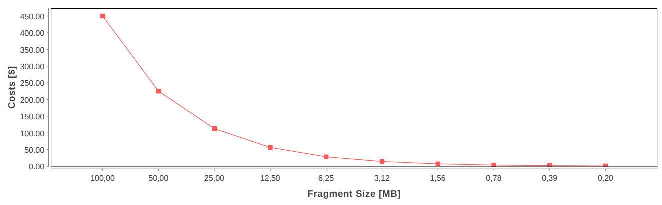


Figure 12: Total S3 Costs per Month

more fragments. Hence, more fragments have to be written back to S3 and this operation is heavily influenced by expensive http put requests.

Total S3 Costs

Here, we show that fragmentation reduces total S3 costs. We use the following cost formula in order to calculate costs resulting from external access to S3 (not from EC2):

$$TotalS3Costs = F * ((n+1) * (FragSize * \frac{StorageCosts}{1000} + \frac{PutCosts}{1000} + FragSize * \frac{DataInCosts}{1000}) + n * (\frac{GetCosts}{10000} + FragSize * \frac{DataOutCosts}{1000})) \quad (1)$$

F is the number of fragments, n is the number of updates, and $FragSize$ is the size of a fragment in MB. Note that we assume that every update of a fragment leads to a new version of this fragment. Even if only a small part of the tree data within a fragment is changed, the whole fragment has to be read and written as one unit, because S3 is used as a block storage device.

These costs are worst-case costs, because we assume that get operations are always performed on S3. In real scenarios we assume most of the get operations on the cached fragments (local transaction manager). Hence, costs should be much lower.

Figure 12 shows the total S3 costs for one month, assuming a fixed project size of 100 MB and 10000 updates of the project file. The project file is fragmented step by step and updates are assumed to be distributed equally upon the fragments. This means, if we, for example, split up a single document into two fragments, $n/2$ updates are performed on each fragment. Again, we used a single S3 bucket in the region EU/Ireland with the versioning feature enabled. As expected, costs for the whole project decrease with an increasing degree of fragmentation. Hence, fragmentation should always be considered in big projects.

Scalability

Previously, we have shown that global and local transaction throughput is sufficient for typical design scenarios. For our measurements we used a relatively small test setup. Here, we want to show that our system model scales with an increasing workload. Therefore, we assume that the necessary transaction managers are hosted in EC2.

In EC2 it is possible to rent different instances with different memory and cpu equipment. For example, one can rent a machine with 7 GB RAM and 20 EC2 compute units. One EC2 compute unit compares to a single 2007 Opteron or Xeon cpu, which is at least a dual core cpu. This means, we can execute at least 40 threads in parallel on one instance. As already mentioned in Section 4, it is possible to execute several transaction managers on a single virtual machine. Assuming a local transaction manager consists of one validate/persist thread and three read threads, we can run 10 local transaction managers (with corresponding S3 buckets) on a single instance. By assuming a local transaction throughput of 100 committed transactions per minute and local transaction manager, we get a total local transaction throughput of 1000 committed transactions per minute and instance.

If a bucket becomes a "hot spot", we can split up the data across other buckets in order to distribute the load. If this is also not sufficient, fragments could be further decomposed and distributed across several buckets.

10 Conclusion and Future Work

In this paper, we proposed a system model, which enables strict consistent and cooperative processing of XML data in a distributed environment upon an existing cloud storage service, namely Amazon S3. We described applicable transaction models for local and distributed transaction processing, followed by a discussion of appropriate mechanisms for optimistic transaction synchronization and transaction commit protocols. We proved that these proposed protocols ensure correctness. Furthermore, we described how strict consistency can be achieved upon a storage layer that at most provides read-after-write consistency for put operations of new data. We also considered recovery with respect to transaction aborts and system failures. A critical part of this paper is the evaluation. We investigated the impact of S3 performance on our implementation within a real deployment on S3. We have shown that fragmentation is necessary in order to reduce storage costs and increase write performance. Furthermore, the evaluation reveals the strong impact of writing to S3 on the transaction throughput. For future work we propose to look for sophisticated approaches in order to avoid this bottleneck. A possible solution is caching of write operations. However, appropriate recovery mechanisms are necessary in order to guarantee durability.

References

[AGLM95] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *In ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1995.

[AGS08] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *PVLDB*, 1(1):598–609, 2008.

[AMS$^+$09] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3), 2009.

[Ber88] A. J. Berkhout. A Holographic Approach to Acoustic Control. In *Journal Audio Eng. Soc.*, volume 36, pages 977–995. 1988.

[BFG$^+$08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.

[Bur06] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

[CDG$^+$08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[CRS$^+$08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[DAA10a] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. *CoRR*, abs/1008.3751, 2010.

[DAA10b] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10. USENIX Association, 2004.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[FGC$^+$97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *SOSP*, pages 78–91, 1997.

[GHS09] F. Gropengießer, K. Hose, and K. Sattler. An Extended Transaction Model for Cooperative Authoring of XML Data. *Computer Science - Research and Development*, 2009.

[GS10] Francis Gropengießer and Kai-Uwe Sattler. Optimistic Synchronization of Cooperative XML Authoring Using Tunable Transaction Boundaries. In *DBKDA*, pages 35–40, 2010.

[HH03] Michael Peter Haustein and Theo Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In *ADBIS*, pages 88–102, 2003.

[HS07] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[KKL10] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD Conference*, pages 579–590, 2010.

[KR79] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. In *VLDB*, pages 351–351, 1979.

[Lam02] Leslie Lamport. Paxos Made Simple, Fast, and Byzantine. In *OPODIS*, pages 7–9, 2002.

[LFWZ09] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling Transaction Services in the Cloud. In *CIDR*, 2009.

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Erlang Workshop*, pages 41–48, 2008.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.

[ZR09] Bo Zhang and Binoy Ravindran. Brief Announcement: Relay: A Cache-Coherence Protocol for Distributed Transactional Memory. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, volume 5923 of *Lecture Notes in Computer Science*, pages 48–53. Springer Berlin / Heidelberg, 2009.