On-Demand Server Synchronization Algorithms for Session Guarantees

Łukasz Piątkowski Cezary Sobaniec Grzegorz Sobański

Institute of Computing Science
Poznań University of Technology, Poland
{Lukasz.Piatkowski,Cezary.Sobaniec,Grzegorz.Sobanski}@cs.put.poznan.pl

Abstract

Session guarantees define required properties of the system regarding consistency of replicas in a distributed system from a single, migrating client's point of view. This paper presents propositions of two new server synchronization algorithms. These algorithms solve two of the main problems of some earlier protocols: lack of server's history pruning and synchronous history update sending.

1. Introduction

Replication is a key concept in providing high performance and availability of data and services in distributed systems. However, replication introduces the problem of data consistency that arises when replicas are modified. There are numerous consistency models developed for *Distributed Shared Memory* systems. These models, called *data-centric* consistency models [7], assume that servers replicating data are also accessing the data for processing purposes. In a mobile environment, however, clients accessing the data are not bound to particular servers, they can switch from one server to another. *Session guarantees* [8], called also *client-centric* consistency models [7], have been proposed to define required properties of the system regarding consistency from the migrating client's point of view. Intuitively: the client wants to continue processing after a switch to another server so that new operations will remain consistent with previously issued operations within a *session*. To allow a client to switch to another server, the servers need to synchronize state of objects the client is using. This synchronization is performed by exchanging operations between servers. Because of this exchange, protocols implementing session guarantees must efficiently represent sets of operations performed in the system. Version vectors based on vector clocks [4, 1] may be used for this purpose. The original protocol presented in [8] used server-based version vectors, where each position of a version vector represents the number of writes performed by a given server. However, other approaches are also possible: protocols using client-based or object-based version vectors [3].

One of the key elements of a system with session guarantees is the synchronization protocol run by servers. One of the first papers about synchronization protocols for servers running session guarantee replication is [5]. This work however presents a solution, which is quite complex and designed for old network architectures. In newer papers the problem of server synchronization was considered orthogonal to the problem of providing session guarantees to clients. For a performance evaluation, theirs authors used a very simple periodic update algorithm. In this paper two new server synchronization protocols are proposed. They address two main problems of the periodic approach: an on demand server synchronization and a server's operation history pruning. The protocols introduced in this work are *ODSAP*, which is applicable for all protocols using a server or client version vectors, and *ODSAP-O*, which is suitable for slightly different approach with an object based version vectors. Correctness of the protocols have been formally proved but the proofs are not included in this paper due to limited space.

2. System Model and Session Guarantees

The system consists of N_S servers holding a full copy of a set of data items. Clients access the data after selecting a single server and sending a direct request to the server in a form of remote procedure calls. The requests are divided into two classes: non-modifying operations (*reads*), and modifying operations (*writes*). Clients are separated from servers, and they can run on a different computer than the server. Clients are mobile, i.e. they can switch from one server to another. The new server selected by a client may hold replicas of objects that are inconsistent with the replicas held by the previous server. Session guarantees are expected to take care of data consistency observed by a single, migrating client.

A client can instruct the servers to check specific consistency criteria expressed using session guarantees. There are four session guarantee types: *Read Your Writes* (RYW), *Monotonic Writes* (MW), *Writes Follow Reads* (WFR) and *Monotonic Reads* (MR). A client requesting RYW expects, that every read operation will reflect all previous writes issued by him, regardless of switching to another server during the session. The MW session guarantee orders globally writes of a given client. A client requesting MR knows, that his read operation will always return a state newer or equal to the one observed by a previous read operation. WFR session guarantee keeps track of ca-

sual dependencies resulting from clients operations. Formal definitions of all guarantees can be found in [6].

Version vectors are used to effectively represent sets of writes resulting from definitions of session guarantees. Each client C_i maintains two version vectors W_{C_i} and R_{C_i} representing the set of writes it has requested, and the set of writes observed by reads issued by the client. Each server S_j maintains a version vector V_{S_j} updated on every write. Version vector representations of sets of writes provide sufficient conditions for assuring session guarantees. Every client along with every request sends a version vector V_{C_i} representing its requirements. The version vector represents the set of writes that are expected to be performed by the destination server before proceeding to the current operation. The version vector is calculated based on W_{C_i}, R_{C_i} and the set of required session guarantees. Before performing a new operation a server S_j checks whether its version vector V_{S_j} dominates the client's version vector V_{C_i} which is denoted by $V_{S_j} \geq V_{C_i}$ and fulfilled when $\forall k\ V_{S_j}[k] \geq V_{C_i}[k]$. Such domination means that the required session guarantees are preserved. Otherwise the server must be updated before proceeding to the current operation.

Every server S_j records write operations it has performed in an ordered history H_{S_j}. Servers exchange information about writes performed in the past in order to synchronize the states of replicas. This synchronization procedure eventually causes total propagation of all writes directly submitted by clients.

It is assumed, that there are available communication primitives like sending or broadcasting a message, that no server will fail and that every message sent by the network is delivered to the destination. These are strong assumptions, but there is already some research done targeted at removing them, e.g. a rollback recovery protocol for session guarantees [2].

3. The Synchronization Protocol

We tried to evaluate a real performance of system running session guarantees replication. The first algorithm of server synchronization used in our experiment was the one described in [6]. The algorithm periodically sends a full copy of the history of server processing to all other servers. As was predicted in [6], the algorithm proved to be completely unacceptable for two reasons. The first one is the size of update messages containing full histories, which generates a lot of unnecessary network traffic. The second one is the fact, that every server keeps all the history in it's memory and never removes any operation from it. Moreover, sending the whole history every time an update is sent, forces the receiving server to process data, which was already received many times. The server needs to find in the received history only those operations, which were not performed yet. This approach causes the system to degenerate in time and finally renders it unusable. To overcome this problem new synchronization algorithms were proposed. They are event-driven and allow servers to prune their histories.

Upon receiving request $\langle op, W\rangle$ **from client** C_i
at server S_j
1: **if** $\left(V_{S_j} \not\geq W\right)$ **then**
2: send $\left\langle syncreq_{S_j}, V_{S_j}\right\rangle$ to all other servers
3: **end if**
4: **while** $\left(V_{S_j} \not\geq W\right)$ **do**
5: `wait()`
6: **end while**
7: perform op and store results in res
8: **if** `iswrite`(op) **then**
9: $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
10: $M_{S_j}[j] \leftarrow V_{S_j}$
11: timestamp op with V_{S_j}
12: $H_{S_j} \leftarrow H_{S_j} \cup \{op\}$
13: **end if**

Upon receiving request $\langle syncreq_{S_i}, V_{S_i}\rangle$
from server S_i **at server** S_j
14: $M_{S_j}[i] \leftarrow V_{S_i}$
15: $V_{\min} \leftarrow V_{S_j}$
16: **foreach** $S_k \neq S_j$ **do**
17: $V_{\min} \leftarrow \min\left(V_{\min}, M_{S_j}[k]\right)$
18: **end for**
19: **foreach** $op_j \in H_{S_j} : V_{\min} \geq T(op_j)$ **do**
20: $H_{S_j} \leftarrow H_{S_j} \setminus \left\{op_j\right\}$
21: **end for**
22: $H_{\text{diff}} \leftarrow \{op_j \in H_{S_j} : V_{S_i} \not\geq T(op_j)\}$
23: **if** $H_{\text{diff}} \neq \emptyset$ **then**
24: send $\left\langle S_j, H_{\text{diff}}\right\rangle$ to S_i
25: **end if**

On receipt of update message $\langle S_i, H_{\text{diff}}\rangle$ **at server** S_j
26: **foreach** $w_i \in H_{\text{diff}}$ **do**
27: **if** $V_{S_j} \not\geq T(w_i)$ **then**
28: perform w_i
29: $V_{S_j} \leftarrow \max\left(V_{S_j}, T(w_i)\right)$
30: $M_{S_j}[j] \leftarrow V_{S_j}$
31: $H_{S_j} \leftarrow H_{S_j} \cup \{w_i\}$
32: **end if**
33: **end for**
34: `signal()`

Alg. 1: On-Demand Synchronization Algorithm with Pruning (ODSAP)

ODSAP Algorithm 1 presents *On-Demand Synchronization Algorithm with Pruning* (ODSAP). In this new approach no periodic updates are sent. Instead, for every client request $\langle op, W\rangle$ coming to a server S_j, version vectors are compared and if the server has not performed all operations required by the client, it sends a synchronization request message to all other servers (line 2 of Alg. 1). This message includes the server's internal version vector V_{S_j} to indicate which operations the server has already performed and which ones are missing. Next, the request is processed as in the original version of the synchronization algorithm (lines 4–13).

Each server holds an array M of size N_S, which repre-

sents the server S_j knowledge about other server's version vector states. This array allows to find operations that have already been performed by all servers in the system and can be pruned from the local history. When a server S_j receives a synchronization request $\langle syncreq_{S_i}, V_{S_i}\rangle$ from a server S_i, it uses included version vector of the server S_i to update its local knowledge about the requesting server, as shown in line 14. During the processing of each synchronization request, the server computes the $V_{\min}$ vector (line 17), which represents the set of operations, that are known to all servers. After that, the server can prune from its history all operations, that are dominated by $V_{\min}$, as no server will ever need them again. This pruning process is described in lines 19–21 of Alg. 1.

After the history is pruned, the server S_j computes a set H_{diff}, which includes all write operations, that are not dominated by V_{S_i} (line 22). This way only operations, that were not known to S_i at the moment of request sending are included in the H_{diff} set. If the resulting set is not empty, it is sent to the requesting server S_i as shown in lines 23–25.

When a server S_j receives an update message $\langle S_i, H_{\text{diff}}\rangle$, it selects from the history H_{diff} only those operations, that it has not performed yet. After performing every operation, the server updates its local version vector V_{S_j} and adds the operation to its local history. After performing all operations, it signals all awaiting requests. This procedure is described in Alg. 1 in lines 26–34.

ODSAP-O ODSAP algorithm cannot be applied when the system is running a protocol using object-based version vectors [3]. Object-based version vectors consist of N_O positions, i.e. the total number of objects in the system. An i-th position of such a version vector represents the number of writes performed on an object identified by i ($oid = i$). Protocols using object-based version vectors must globally order writes on individual objects, and this causes problems.

Figure 1 shows, that the ODSAP algorithm combined with a protocol using object-based version vectors can cause a deadlock. In this figure a client C_1 is requesting a write operation on object with $oid = 4$ (a message *Req_*1). This operation must be ordered and it gets a sequence number 2. The receiving server S_1 cannot perform this request, because it is not up to date enough. It tries to update it's state, so it sends a *SyncReq* message to the server S_2. Meanwhile another client C_2 comes with another write request (*Req_*2) concerning the same object, and gets next free sequence number for the object ($seq = 3$). Now, S_2 also cannot perform the new request, because it has invalid sequence number (the next is 2 not 3). S_2 also sends a synchronization request, which is received by S_1. If S_1 answers S_2 before completing the request from C_1, its update will not contain the operation from *Req_*1 (actually, in this situation S_1 will not send any answer, because the H_{diff} set is empty). Every synchronization request is answered exactly once. If we omit the operation from *Req_*1 in the answer for S_2 the server will wait infinitely for missing operation with a sequence number 2. Any other request coming now to the object with $oid = 4$ will never proceed. Similarly, every operation requested at S_2, for which the client

Upon receiving request $\langle op, W\rangle$ **from client** C_i
at server S_j
1: $seq \leftarrow 0$
2: **if** `iswrite`(op) **then**
3: $\quad seq \leftarrow$ `getSeqNumber(id`(op)`)`
4: **end if**
5: **if** $\left(V_{S_j} \not\geq W \vee \left(seq > V_{S_j}[\texttt{id}(op)] + 1\right)\right)$ **then**
6: $\quad$ send $\left\langle syncreq_{S_j}, V_{S_j}, seq, \texttt{id}(op)\right\rangle$ to all servers
7: **end if**
8: **while** $\left(V_{S_j} \not\geq W \vee \left(seq > V_{S_j}[\texttt{id}(op)] + 1\right)\right)$ **do**
9: $\quad$ `wait()`
10: **end while**
11: perform op and store results in *res*
12: **if** `iswrite`(op) **then**
13: $\quad V_{S_j}[\texttt{id}(op)] \leftarrow V_{S_j}[\texttt{id}(op)] + 1$
14: $\quad M_{S_j}[j] \leftarrow V_{S_j}$
15: $\quad$ timestamp op with V_{S_j}
16: $\quad H_{S_j} \leftarrow H_{S_j} \cup \{op\}$
17: $\quad$ `signal()`
18: **end if**

Upon receiving request $\langle syncreq_{S_i}, V_{S_i}, seq_{S_i}, oid_{S_i}\rangle$
from server S_i **at server** S_j
19: $M_{S_j}[i] \leftarrow V_{S_i}$
20: $V_{\min} \leftarrow V_{S_j}$
21: **foreach** $S_k \neq S_j$ **do**
22: $\quad V_{\min} \leftarrow \min(V_{\min}, \mathrm{M}[k])$
23: **end for**
24: **foreach** $op_j \in H_{S_j} : V_{\min} \geq T(op_j)$ **do**
25: $\quad H_{S_j} \leftarrow H_{S_j} \setminus \{op_j\}$
26: **end for**
27: **while** $seq_{S_i} \neq 0 \wedge \exists op$ `waiting`$(op) \wedge$ `id`$(op) = oid_{S_i} \wedge$ `seq`$(op) < seq_{S_i}$ **do**
28: $\quad$ `wait()`
29: **end while**
30: $H_{\text{diff}} \leftarrow \{op_j \in H_{S_j} : V_{S_i} \not\geq T(op_j)\}$
31: **if** $H_{\text{diff}} \neq \emptyset$ **then**
32: $\quad$ send $\langle S_j, H_{\text{diff}}\rangle$ to S_i
33: **end if**

On receipt of update message $\langle S_k, H_{\text{diff}}\rangle$ **at server** S_j
34: **foreach** $w_i \in H_{\text{diff}}$ **do**
35: $\quad$ **if** $V_{S_j} \not\geq T(w_i)$ **then**
36: $\quad\quad$ perform w_i
37: $\quad\quad V_{S_j} \leftarrow \max\left(V_{S_j}, T(w_i)\right)$
38: $\quad\quad M_{S_j}[j] \leftarrow V_{S_j}$
39: $\quad\quad H_{S_j} \leftarrow H_{S_j} \cup \{w_i\}$
40: $\quad$ **end if**
41: **end for**
42: `signal()`

Alg. 2: On-Demand synchronization Algorithm with Pruning for Object Vectors (ODSAP-O)

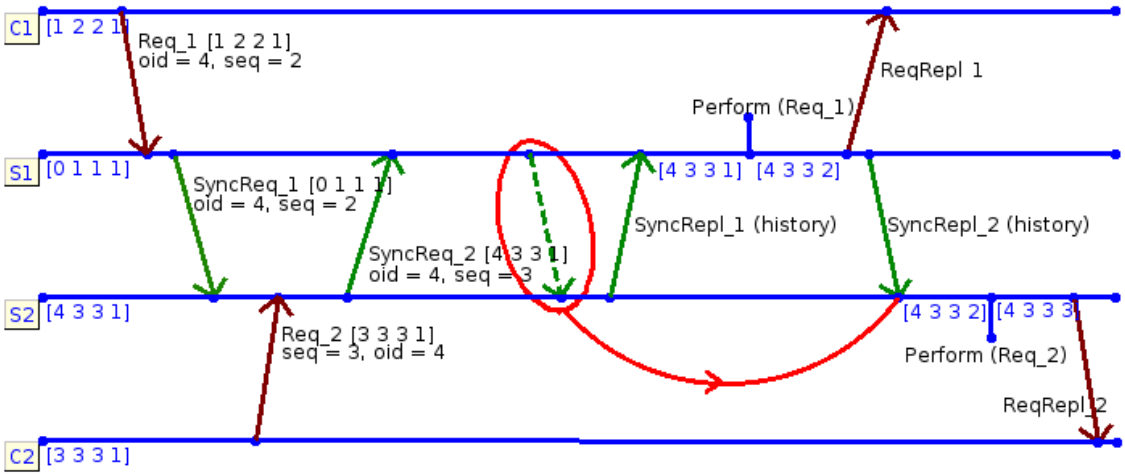

Figure 1: Deadlock in ODSAP algorithm used with a protocol using object-based version vectors

version vector was not dominated by V_{S_2}. This way the deadlock can spread to all other objects and servers. It's worth noting, that even if another client will send another request to S_2, which will cause synchronization with S_1, it will not resolve the deadlock, because $V_{S_2} \geq V_{S_1}$. ODSAP tries to minimize a number of messages sent and does not send a synchronization reply in this case.

The proposed solution of the problem described above is to suspend sending synchronization replies if there is a request from a client being processed. This client request has to have a lower sequence number than the request, that caused synchronization, and of course it must be related to the same object as the synchronization request. Our solution of this problem is an extended ODSAP-O algorithm, which is presented in Alg. 2.

Lines 5–10 of Alg. 2 contain a new condition for sending synchronization requests. Line 17 contains an additional `signal()` operation, because adding an operation to the history may cause a suspended synchronization request to be resumed. In ODSAP-O it is possible to hold back an incoming synchronization request, if it can cause a deadlock. The condition for holding back such requests is presented in lines 27–29. This condition selects all clients requests, that called `wait()`, accessed the same object as stated in the synchronization request, and have lower sequence number than the one included in the synchronization request. If any such request exists at a given server, the incoming synchronization request must be hold back. There is a chance that it will be resumed after adding a new operation to the history and calling `signal()`, which is possible in lines 17 and 42.

4. Conclusions and Future Work

In this paper two new synchronization protocols for systems using session guarantees were proposed. This protocols allow servers to perform synchronization only when it is really needed. They also enable servers to prune their history of operations, which solves the problem of system degeneration. Both of these problems were present in the synchronization protocols described in the earlier work. Future work will be concentrated on performance evaluation of the proposed algorithms and further optimizations of server synchronization protocols.

References

[1] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[2] A. Kobusinska. *Rollback-Recovery Protocols for Distributed Mobile Systems Providing Session Guarantees*. PhD thesis, Institute of Computing Science, Poznan University of Technology, September 2006.

[3] A. Kobusińska, M. Libuda, C. Sobaniec, and D. Wawrzyniak. Version vector protocols implementing session guarantees. In *Proc. of Int. Symp. on Cluster Computing and the Grid (CCGrid 2005)*, pages 929–936, Cardiff, UK, May 2005.

[4] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. of the Int. Conf. on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., October 1988.

[5] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, pages 288–301, Saint Malo, France, October 1997.

[6] Cezary Sobaniec. *Consistency Protocols of Session Guarantees in Distributed Mobile Systems*. PhD thesis, Poznań University of Technology, Poznań, September 2005.

[7] A. S. Tanenbaum and M. van Steen. *Distributed Systems — Principles and Paradigms*. Prentice Hall, New Jersey, 2002.

[8] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of the Third Int. Conf. on Parallel and Distributed Information Systems (PDIS 94)*, pages 140–149, Austin, USA, September 1994. IEEE Computer Society.