Correctness Proofs of On-Demand Server Synchronization Protocols of Session Guarantees*

Łukasz Piątkowski, Cezary Sobaniec, and 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 a distributed system regarding consistency from the point of view of a single mobile client. Consistency protocols of session guarantees are composed of two elements: the first is aimed at providing safety (the *guarantees*), the second is aimed at providing liveness (data synchronization). This paper presents correctness proofs of two new data server synchronization protocols, which solve main problems of earlier proposals.

1 Introduction

Nowadays, distributed, mobile and ubiquitous computer systems are becoming more and more popular. It is expected, that these systems are able to provide high performance and availability of data and services. A key concept in providing these features is replication. However, it 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 [1], assume that servers replicating data are also accessing the data for processing purposes. All consistency requirements are defined from the point of view of a data object.

Our work is aimed toward a mobile environment, where clients are accessing data and services located on different servers. This approach is very different from DSM replication and is called *session guarantees* [2] or *client-centric* consistency models [1]. Clients using session guarantees are not bound to a particular server, but they can switch from one server to another to achieve better performance or keep data available. Data consistency requirements are specified only from the point of view of a single migrating client. Intuitively: the client wants to continue processing after a switch to another server, but it requires new operations to remain consistent with previously issued operations within *a session*. This model is expected to be better suited for mobile systems. Of course DSM models are also possible to use in a mobile environment. Unfortunately, weak DSM models which are applicable in such environments, are hard to use by the user, while strong models are easy to use, but very hard, or even impossible, to achieve in a highly mobile and asynchronous system [3]. Moreover, DSM consistency protocols maintain consistency from the point of view of a data object, independently of a number of clients performing operations and their locations.

* The research presented in this paper has been partially supported by the European Union within the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

To allow a client to switch to another server, the servers need to synchronize operations which were performed on objects being accessed by the client. This synchronization is done by exchanging histories of writes between servers. Constant size version vectors based on vector clocks [4, 5] may be used for efficient representation of write sets. It is proved in [6] that the version vector representation of a set of writes is always a superset of the exact set of operations resulting from definitions of session guarantees. Each position of the version vector represents the number of writes performed by a given server. [7] shows that other approaches are also possible: protocols using client-based or object-based version vectors.

One of the key elements of a system with session guarantees is the synchronization protocol used by servers to exchange write history sets. [8] presents a solution, which is quite complex and designed for old and slow network architectures. Newer papers [7, 6] focus on the problem of providing formal description of session guarantees, but they consider the problem of server synchronization to be orthogonal and use a simple periodic history multicasting protocol. To overcome serious limitations of this basic approach, two new server synchronization protocols were proposed in [9]. They introduced an on-demand server synchronization protocol with pruning of histories.

The purpose of this paper is to prove that protocols proposed in [9] are safe and provide liveness. This paper is organized as follows. First, a system model is introduced and session guarantees are defined in Section 2. In Section 3 the synchronization protocols are described. Section 4 contains proofs of safety and liveness. Finally, conclusions and future work is outlined in Section 5.

2 System Model and Session Guarantees

The system consists of a set $\mathcal{S}$ of servers ($|\mathcal{S}| = N_S$) holding a full copy of a set of data objects. There are N_C clients accessing the data. Each of them selects a single server and sends direct requests divided into two classes: non-modifying operations (*reads*), and modifying operations (*writes*). Clients are separated from servers, and they can run on a different computers than servers. Clients are mobile, i.e. they can switch from one server and object to another. 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 guarantees [2]: *Read Your Writes* (RYW), *Monotonic Writes* (MW), *Monotonic Reads* (MR) and *Writes Follow Reads* (WFR). Formal definitions of session guarantees can be found in [6].

A client requesting RYW session guarantee expects, that every read operation will reflect all previous writes issued by him, regardless of switching to another server during the session. This guarantee can be exemplified by a user writing a TODO list to a file. When he is traveling between different locations and servers, he wants to recall the most urgent tasks on the TODO list. Without RYW session guarantee the read may miss some recent updates of the list.

The MW session guarantee globally orders writes of a given client. Let us consider a counter object with two methods for updating its state: *increment()*, and *set()*. A user of the counter issues the *set()* function at first, and then updates the counter by calling

increment() function. Without MW session guarantee the final result would be unpredictable, because it depends on the order of the execution of these two functions.

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. Usage of the MR guarantee can be illustrated by a mailbox of a traveling user. The user opens the mailbox at one location, and reads a few emails. Afterward he opens the same mailbox at different location; he expects to see at least all the messages he has seen previously. The new state may not reflect the most recent one, but must be at least as new as the previously observed state.

WFR session guarantee keeps track of causal dependencies resulting of the client's operations. As an example let us consider a discussion forum, where a user has read some posts. Some time later the user wants to post a reply to a message he has read. The new message must be submitted to a server that knows the post to which the user is going to reply. WFR session guarantee may solve the problem by tracking causal dependency between the read of the original message and the posting (write) of the reply.

It is important to note, that differences between data centric and session guarantees approaches are essential and clearly seen when we try to compare them. The dissimilarity is caused by a very different way of client cooperation model and consistency properties. This can be seen when one tries to achieve session guarantees using DSM protocols and vice versa. In [10] it is proved, that only if we enable all possible session guarantees, we can get a causal consistency known from DSM systems, and in [11] that enabling three of four guarantees is required to get a PRAM consistency. Thus trying to implement DSM consistency using session guarantees is inefficient. Also trying to get session guarantee consistency using DSM algorithms is very ineffective. Let us consider a client with just MW guarantee enabled issuing two write operations w_1 and w_2 at two different servers S_1 and S_2, respectively. If a strong consistency of Sequential Order is enabled on the server side, it is still possible, that during the synchronization of servers, the final order of the writes will become w_2, w_1. This still satisfies sequential consistency, but violates MW guarantee. Therefore, to achieve MW guarantee, no new client's operation in the whole system may be processed until a previously issued operation is finished and replicated to other servers — and this requires Atomic Consistency. Moreover, the client cooperation model in session guarantees differs from the one in DSM systems. In session guarantees, if a set of clients is requesting operations only from a subset of all servers, there is no need to synchronize operations among all servers, but only among the subset. This is not true in DSM, i.e. when using Atomic Consistency, many protocols require all replicas to be updated before client's operation is finished. Some of those problems are solved in voting base protocols, but still they require a synchronous update of all replicas in a write quorum [12].

To achieve consistency of replicated objects, an exchange protocol is required [7]. As mentioned in Section 1, such protocols use version vectors to efficiently 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 performed at this server. Every client along with

every request sends a version vector V_{C_i} representing its consistency requirements. This 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 version vectors 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 version vector V_{C_i}. The domination is denoted by $V_{S_j} \geq V_{C_i}$ and is 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 performing the requested operation.

Every server S_j records all 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 the system supports reliable communication primitives like sending unicast and broadcast messages. Another assumption is that the servers do not fail. These assumptions are quite strong, but there is already some research targeted at alleviating them, e.g. consistency protocols of session guarantees using rollback recovery proposed in [13].

3 The Synchronization Protocol

The simple synchronization protocol used in [6] was proposed only to allow evaluation of the quality of version vector representations of sets of writes. In that approach a full copy of the server's history of processing is sent periodically to all other servers. Due to growing size of the set of writes, the algorithm is obviously unacceptable, as it causes the system to degenerate in short time. To overcome this problem, new synchronization protocols called *ODSAP* and *ODSAP-O* were proposed. They are event-driven and allow servers to prune their histories. Understanding those protocols is important for providing correctness proofs, so they are briefly discussed here. More detailed description is available in [9].

In order to present ODSAP synchronization protocols, it is necessary to introduce some basic operations. *Send* is understood as a network communication primitive allowing to send a message to a remote node. *Wait* causes a calling thread to suspend until the *signal* operation is executed. The function *T(op)* returns a version vector timestamp assigned earlier to the operation *op*. The type of operation may be determined by the *iswrite(op)* function. *Deliver* means that a result of a remote call can be delivered to the application.

The client side part of the proposed protocols is the same in both cases and is presented on Alg. 1.1. Lines 1–8 of this algorithm show how a client computes a version vector associated with every request sent to a server. The client's version vectors R_{C_i} and W_{C_i} are updated after receiving a response from a server, lines 9–14. A detailed explanation of this part of the protocol can be found in [6].

Algorithm 1.2 presents server side of the *On-Demand Synchronization Algorithm with Pruning* (ODSAP). In this approach no periodic updates are sent, but the server

On send of $\langle Req \rangle\, op, SG$ **from** C_i **to** S_j
1: $V_{C_i} \leftarrow \mathbf{0}$
2: **if** (`iswrite`(op) **and** MW $\in SG$) **or** (**not** `iswrite`(op) **and** RYW $\in SG$) **then**
3: $\quad V_{C_i} \leftarrow \max(V_{C_i}, W_{C_i})$
4: **end if**
5: **if** (`iswrite`(op) **and** WFR $\in SG$) **or** (**not** `iswrite`(op) **and** MR $\in SG$) **then**
6: $\quad V_{C_i} \leftarrow \max(V_{C_i}, R_{C_i})$
7: **end if**
8: send $\langle Req \rangle\, op, V_{C_i}$ to S_j

Upon receiving $\langle Repl \rangle\, op, res, V_{S_j}$ **from server** S_j **at client** C_i
9: **if** `iswrite`(op) **then**
10: $\quad W_{C_i} \leftarrow \max(W_{C_i}, V_{S_j})$
11: **else**
12: $\quad R_{C_i} \leftarrow \max(R_{C_i}, V_{S_j})$
13: **end if**
14: deliver res

Alg. 1.1: Client side of ODSAP and ODSAP-O protocols

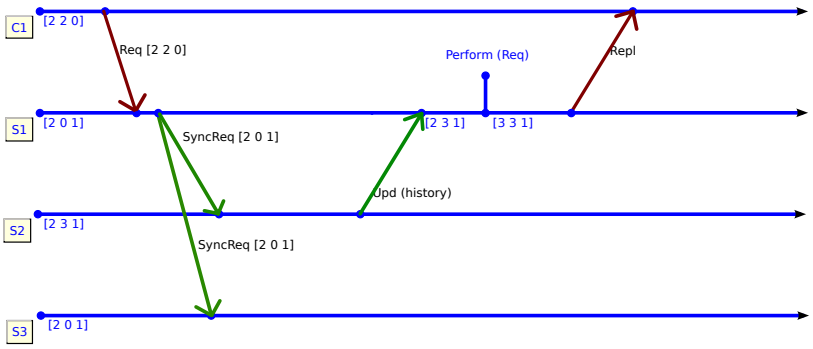

Fig. 1: Synchronization protocol

sends an on-demand synchronization request (line 2 of Alg. 1.2) when consistency criteria are not met. When a server S_j receives a synchronization request $\langle SyncReq, V_{S_i} \rangle$ from a server S_i, it computes a set H_{diff}, which includes all write operations that are missing (not dominated) by S_i (line 16). After receiving an update message $\langle Upd, S_i, H_{\text{diff}} \rangle$ by S_j, it executes from the H_{diff} set only those operations, that it has not yet performed. After that, it signals all awaiting requests. This procedure is described in Alg. 1.2 in lines 20–27. Both proposed protocols include also the same history pruning solution, which was fully described in [9] and is presented in lines 28–32 of Alg. 1.2.

Figure 1 shows a simple case of the synchronization protocol execution. In this case client C_1 is sending a request to server S_1, which does not dominate client's request version vector. Hence the S_1 sends a synchronization request to all other servers. The S_3 server does not have any operations dominating S_1's version vector, so it does not send any reply. On the contrary, the server S_2 has such operations, so it sends a reply. After receiving the reply, server S_1 can perform the client's operation.

Upon receiving $\langle Req \rangle\, op, V_{C_i}$ **from client** C_i **at server** S_j
1: **if** $\left(V_{S_j} \not\geq V_{C_i}\right)$ **then**
2: send $\langle SyncReq \rangle\, V_{S_j}$ to all other servers
3: **end if**
4: **while** $\left(V_{S_j} \not\geq V_{C_i}\right)$ **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**
14: send $\langle Repl \rangle\, op, res, V_{S_j}$ to C_i

Upon receiving $\langle SyncReq \rangle\, V_{S_i}$ **from server** S_i **at server** S_j
15: $M_{S_j}[i] \leftarrow V_{S_i}$
16: $H_{\text{diff}} \leftarrow \{op \in H_{S_j} : V_{S_i} \not\geq T(op)\}$
17: **if** $H_{\text{diff}} \neq \emptyset$ **then**
18: send $\langle Upd \rangle\, S_j, H_{\text{diff}}$ to S_i
19: **end if**

Upon receiving $\langle Upd \rangle\, S_i, H_{\text{diff}}$ **at server** S_j
20: **foreach** $w_i \in H_{\text{diff}}$ **do**
21: **if** $V_{S_j} \not\geq T(w_i)$ **then**
22: perform w_i
23: $V_{S_j} \leftarrow \max\left(V_{S_j}, T(w_i)\right)$
24: $H_{S_j} \leftarrow H_{S_j} \cup \{w_i\}$
25: **end if**
26: **end for**
27: `signal`

On idle event at server S_j
28: $V_{\min_j} \leftarrow V_{S_j}$
29: **for** $k = 1 \ldots N_S, k \neq j$ **do**
30: $V_{\min_j} \leftarrow \min\left(V_{\min_j}, M_{S_j}[k]\right)$
31: **end for**
32: $H_{S_j} \leftarrow H_{S_j} \setminus \{op \in H_{S_j} : V_{\min_j} \geq T(op)\}$

Alg. 1.2: ODSAP — server side

Unfortunately, the proposed ODSAP protocol cannot be applied directly when the system is using object-based version vectors, which require a distributed sequence counter of operations on every object[6]. In this case, every request of a client is associated with a sequence number of an operation on an object with a given id. Servers need to keep track of both version vectors and sequence numbers. Figure 2 shows, that the ODSAP protocol combined with an object-based version vectors can cause a dead-

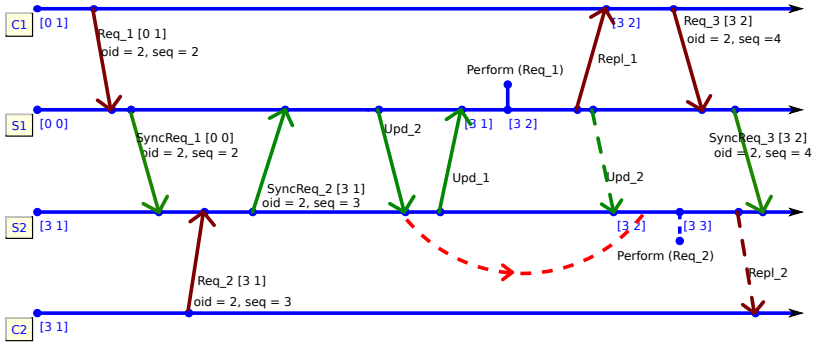

Fig. 2: Deadlock in ODSAP protocol used with object-based version vectors

lock. The whole problem and its solution are discussed in [9]. The protocol resolving this problem is an extended ODSAP-O protocol, which is presented on Alg. 1.3.

Algorithm 1.3 introduces new system operations: *id(op)*, which returns *id* of an object, on which an operation *op* is requested; *getNextSeqNumber(oid)*, which returns the next value of the distributed sequence number generator for all writes on the object *oid*; *waiting(op)* indicates whether the operation *op* is waiting for completion of the server synchronization; *seq(op)* returns the value of the distributed counter assigned earlier by a call to *getNextSeqNumber(oid)*.

Lines 5–10 of Alg. 1.3 contain a new condition for sending synchronization requests, which now requires both vector dominance and correct sequence number. 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 request is presented in lines 20–22.

4 Safety and Liveness of Proposed Protocols

Both presented protocols include the same history pruning solution, which is based on the $V_{\min}$ infimum. Safety of this protocol means that it cannot remove from history any single operation that may be ever needed by another server. Liveness means that any operation which may be removed from a server's history will be eventually removed.

Lemma 1. *If the operation op is performed by all servers, than the* $V_{\min}$ *vector of every server dominates* $T(op)$.

Proof. By contradiction: Let us consider an operation *op* which is already performed by all servers and its $T(op)$ is not dominated by $V_{\min_j}$ of server S_j. A server can perform an operation only as a result of a direct client request or an update from another server. The V_S version vector of every server S is monotonous and always dominates timestamps of all operations performed by this server. This is shown in lines 9, 11 and 23 of Alg. 1.2 and in lines 13, 15 and 31 of Alg. 1.3. Because S_j already performed *op*, it means that also $V_{S_j} \geq T(op)$. The $V_{\min}$ vector of every server is defined as the infimum of it's local V_S vector and vectors representing view of other servers, thus on every server $V_{\min} \geq V_S$. As a result $V_{\min_j} \geq V_{S_j} \geq T(op)$ — a contradiction. □

Upon receiving $\langle Req \rangle\, op, V_{C_i}$ **from client** C_i **at server** S_j
1: $seq \leftarrow 0$
2: **if** `iswrite`(op) **then**
3: $\quad seq \leftarrow$ `getNextSeqNumber`(`id`(op))
4: **end if**
5: **if** $\big(V_{S_j} \not\geq V_{C_i} \vee \big(seq > V_{S_j}[\texttt{id}(op)] + 1\big)\big)$ **then**
6: $\quad$ send $\langle SyncReq \rangle\, V_{S_j}, seq,$ `id`(op) to all servers
7: **end if**
8: **while** $\big(V_{S_j} \not\geq V_{C_i} \vee \big(seq > V_{S_j}[\texttt{id}(op)] + 1\big)\big)$ **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**
19: send $\langle Repl \rangle\, op, res, V_{S_j}$ to C_i

Upon receiving $\langle SyncReq \rangle\, V_{S_i}, seq_{\,S_i}, oid_{S_i}$ **from server** S_i **at server** S_j
20: **while** $seq_{\,S_i} \neq 0 \wedge \exists\, op\ \texttt{waiting}(op) \wedge \texttt{id}(op) = oid_{S_i} \wedge \texttt{seq}(op) < seq_{\,S_i}$ **do**
21: $\quad$ `wait`
22: **end while**
23: $H_{\text{diff}} \leftarrow \big\{op_j \in H_{S_j} : V_{S_i} \not\geq T(op_j)\big\}$
24: **if** $H_{\text{diff}} \neq \emptyset$ **then**
25: $\quad$ send $\langle Upd \rangle\, S_j, H_{\text{diff}}$ to S_i
26: **end if**
27: $M_{S_j}[i] \leftarrow \max\big(V_{S_i}, V_{S_j}\big)$

Upon receiving $\langle Upd \rangle\, S_i, H_{\text{diff}}$ **at server** S_j
28: **foreach** $w_i \in H_{\text{diff}}$ **do**
29: $\quad$ **if** $V_{S_j} \not\geq T(w_i)$ **then**
30: $\quad\quad$ perform w_i
31: $\quad\quad V_{S_j} \leftarrow \max\big(V_{S_j}, T(w_i)\big)$
32: $\quad\quad H_{S_j} \leftarrow H_{S_j} \cup \{w_i\}$
33: $\quad$ **end if**
34: **end for**
35: `signal`

Alg. 1.3: ODSAP-O — server side without pruning

Theorem 1. *(Liveness of pruning) All operations which are performed by all servers are eventually removed from histories.*

Proof. By contradiction: Let us consider an operation op which is already performed by all servers, but will never be removed from server's S_j history. According to Lem. 1 the following holds: $\forall_{S_j \in \mathcal{S}} : V_{\min_j} \geq T(op)$. When the server S_j becomes idle and

executes the pruning algorithm, it will prune all operations op_i from its history such that $V_{\min_j} \geq T(op_i)$. It means that also op will be removed — a contradiction. □

Lemma 2. *If the $V_{\min}$ vector of every server dominates $T(op)$, than the operation op was performed by all servers.*

Proof. By contradiction: Let us consider an operation op which is not performed by all servers, but its $T(op)$ is dominated by $V_{\min}$ of all servers. The $V_{\min_j}$ vector is calculated based on the array of vectors M_{S_j} by taking minimal values of appropriate positions of all but the j-th elements of the array. As a result we get: $\forall_{k \neq j} M_{S_j}[k] \geq V_{\min_j}$. The values stored in the array M_{S_j} are copies of appropriate version vectors maintained by servers (line 10 of Alg. 1.2 and line 14 of Alg. 1.3). Version vectors maintained by servers are monotonically increasing, therefore the real version vector of a server V_{S_k} always dominates its (possibly outdated) copy $M_{S_j}[k]$ at another server S_j. If we additionally take into account that $V_{S_j} \geq V_{\min_j}$, we get: $\forall_{S_k} V_{S_k} \geq V_{\min_j}$, and thus $\forall_{S_k} V_{S_k} \geq T(op)$. It means that version vectors of all servers dominate the timestamp of the operation op. The server S_i which has accepted the operation op as a direct request from a client has assigned a unique version vector to this operation by increasing its version vector V_{S_i} at position i (lines 9 and 11). Server version vectors get updated along with execution of appropriate ordered sequences of operations (lines 22 and 23). For a given server S_k, its version vector V_{S_k} represents all operations op_i already performed by the server, therefore $V_{S_k} \geq T(op_i)$. In our case $\forall_{S_k} V_{S_k} \geq T(op)$, which means that all servers have performed operation op — a contradiction. □

Theorem 2. *(Safety of pruning) The ODSAP's pruning algorithm removes from histories only those operations that have already been performed by all servers.*

Proof. By contradiction: Let us consider a server S_j removing an operation op which is not performed by all servers. As can be seen in line 32 of Alg. 1.2, the pruning algorithm on every server removes only such operations op, that $V_{\min} \geq T(op)$. According to Lem. 2 if $V_{\min} \geq T(op)$, then the operation op was performed by all servers — a contradiction. □

It is worth noting, that according to Lem. 1 and Lem. 2, the operation op is performed by all servers iff the $V_{\min}$ vector of every server dominates $T(op)$.

The two proposed server synchronization protocols do not change session guarantees definitions, so there is no need to prove they have the property of safety. These proofs can be found in [6]. Nevertheless, a proof of liveness is needed. In this case liveness means that every client request will eventually end.

Definition 1. *For a given server $S_i \in \mathcal{S}$ with a version vector V_{S_i}, a set of nodes $S_S(S_i) \subseteq \mathcal{S}$ is called a synchronization set of server S_i and is defined as follows:*

$$S_S(S_i) = \{S_j \in \mathcal{S} : \underset{op \in H_{S_j}}{\exists} V_{S_i} \not\geq T(op)\}$$

It implies that $V_{S_i} \not\geq V_{S_k}$ where $S_k \in S_S(S_i)$ and that $V_{S_i} \geq V_{S_l}$ where $S_l \notin S_S(S_i)$.

Theorem 3. *Every operation requested by a client to a server running ODSAP synchronization protocol eventually ends.*

Proof. Every client request of operation op_{C_k} sent by a client C_k to a server $S_i \in \mathcal{S}$ can be hold back only in line 5 of Alg. 1.2, otherwise client's request is performed by S_i and a reply message is sent to C_k. If the C_k's request was suspended by a call to `wait`(), also a synchronization request has been sent by S_i to $S \in \mathcal{S} \setminus \{S_i\}$, where $S_S(S_i) \subseteq \mathcal{S}$ (line 2). Client's version vectors can be updated only as a result of receiving a request reply from some server S_j. A server version vector is monotonous and always dominates all operations performed by that server, therefore if a client's request with a version vector W, where $V_{S_i} \not\geq W$, was hold back at server S_i, there must exist at least one server S_j such that $V_{S_j} \geq T(op_{C_k})$. It means that $S_S(S_i)$ is not empty and $S_j \in S_S(S_i)$.

Because the network and the servers never fail, every message sent, including the S_i's synchronization request, will be eventually delivered. The procedure of replying to the synchronization request is non-blocking (lines 15–19 of Alg. 1.2), thus S_i will eventually get update messages from servers from its synchronization set $S_s(S_i)$. Each of the servers $S_j \in S_S(S_i)$ sends to S_i all operations op from a S_j's history such that $op \in H_{S_j} \wedge V_{S_i} \not\geq T(op)$ (line 16). All server version vectors are monotonous. Because $S_S(S_i)$ contains all and only such servers that $V_{S_i} \not\geq V_{S_j},\ S_j \in S_S(S_i)$, every server which accepted operations unknown to S_i, must also belong to $S_S(S_i)$. The version vector of client C_k represents all writes which consequences the client needs to see. All these operations are unknown to S_i and therefore were performed by some servers from $S_s(S_i)$. All servers from this set respond in a reliable way to S_i's synchronization request. This means that eventually all operations blocking S_i from performing a held back client's request will be received and performed (lines 20–27). Thus, eventually $V_{S_i} \geq W$ and the held back operation op_{C_k} will be performed. □

Protocols that use object version vectors must assure an ordering of writes on respective objects. That order is achieved using a distributed counter, which assigns unique and consistent sequence numbers. The problem of generating such numbers in a distributed environment is considered orthogonal to the problem of server synchronization protocols.

Definition 2. *For every object we define a sequence number value called* next*, which is the sequence number of the first requested operation, that was not yet performed by any server:*

$$next(oid) = max\,\{\underset{S_i \in \mathcal{S}}{\forall} V_{S_i}[oid]\} + 1$$

Theorem 4. *Every operation requested by a client to a server running ODSAP-O synchronization protocol eventually ends.*

Proof. The generated sequence number for a given client request can be assigned to one of four cases of sequence values:

1. $seq = 0$ — this is true for read operations,
2. $seq > 0 \ \wedge \ seq < next$ — no current client request can be assigned a sequence number from this interval, as these are the numbers of requests already completed,
3. $seq = next$ — this is the next expected operation sequence number; this operation must be performed before all other active clients' requests on the same object,

4. $seq > next$ — these are the sequence numbers of operations, that must wait until all operations with lower sequence numbers will be completed and information about them synchronized with the server serving the request.

Considering the case 2., operations with such sequence number are already completed or there are no such operations (just after a system start, when no operation has been performed yet).

Let's now assume, that the incoming client request is assigned a sequence number like in the case 1. or 3. In these cases the conditions in lines 5 and 8 of Alg. 1.3 transform directly to corresponding conditions in Alg. 1.2. Similarly, the *while* condition in line 20 of Alg. 1.3 is never true, because no other client request can have lower sequence number than *next*. This way the whole procedure in lines 20–26 of Alg. 1.3 becomes identical to the analogous procedure of Alg. 1.2. As can be seen, for cases 1. and 3., the whole Alg. 1.3 is equivalent to Alg. 1.2, so the same proof of liveness as for Thm. 3 holds.

Now, in the 4. case, the request of a client C_k with a sequence number seq_{C_k}, directed to a server S_i, can be hold back because of out-of-order sequence number or a vector dominance. In both cases a synchronization request is sent by S_i to all $S_k \in \mathcal{S} \setminus \{S_i\}$. If this synchronization request is not held back by a receiving server S_k, the situation is again analogous to Alg. 1.2. Otherwise, S_k suspends sending the synchronization reply, if it has an already held back request from a client C_l with sequence number lower than the value seq_{S_i} included in a S_i's synchronization request. S_k cannot send synchronization reply to S_i, because reply is sent only once, so it must include the operation from C_l's request. Otherwise, S_i will not receive an operation, which is needed to satisfy condition in line 8 of Alg. 1.3. Now S_k's situation is the same as S_i's — it has sent a synchronization request with a sequence number $seq_{S_k} < seq_{S_i}$and is awaiting a reply. This chain of sequence number dependencies ends on a server S_m, where $seq_{S_m} = next$. As was stated earlier, in this case a client request to S_m eventually ends. After that S_m answers all synchronization requests it has held back, including the one coming from server S_l with sequence number $seq_{S_l} = seq_{S_m} + 1$. Next, S_l receives all synchronization replies, thus it will know all operations with previous sequence numbers and also all operations, that updated it's $V_{S_l}$to the state that dominates a version vector in a held back client request. The client request can now be performed, so does the *while* loop in line 8. Applying this recursively, S_k and S_i eventually get all required operations and every client request will be eventually performed.

Concluding all above cases: for all possible values of sequence number included in client's request, the request will eventually end. □

5 Conclusions and Future Work

Session guarantees are a poorly explored approach to the problem of objects and services consistency in mobile environments. This solution is particularly well suited for mobile clients, where data-centric consistency is hard and expensive to achieve. Nowadays, when Service Oriented Architecture and mobile computing is becoming more and more popular, an efficient way of services and data replication for mobile networks is becoming one of the major problems to solve.

In this paper proofs of safety and correctness of two new server synchronization protocols were presented. These protocols allow servers to perform synchronization only when it is really needed and enable servers to prune their operation histories, which solves the problem of system degeneration. Both of these problems were present in the synchronization protocols described in earlier work.

Of course, many additional problems remain open. The most important, in our opinion, research tasks are now a performance evaluation of proposed protocols and further work on methods of efficient server synchronization. Performance evaluation was already conducted, but length limitations of this paper does not allow us to present it here. Nevertheless, these results were described in a separate paper and submitted for review to the same conference.

References

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

[2] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proc. of the Third Int. Conf. on Parallel and Distributed Information Systems (PDIS 94)*, (Austin, USA), pp. 140–149, IEEE Computer Society, Sept. 1994.

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[4] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. of the Int. Conf. on Parallel and Distributed Algorithms* (Cosnard, Quinton, Raynal, and Robert, eds.), pp. 215–226, Elsevier Science Publishers B. V., Oct. 1988.

[5] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, pp. 28–33, Aug. 1991.

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

[7] 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)*, (Cardiff, UK), pp. 929–936, May 2005.

[8] 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)*, (Saint Malo, France), pp. 288–301, Oct. 1997.

[9] L. Piatkowski, C. Sobaniec, and G. Sobanski, "On-demand server synchronization algorithms for session guarantees," in *Proc. of the 23rd International Symposium on Computer and Information Sciences (ISCIS 2008)*, (Istanbul, Turkey), pp. 1–4, Oct. 2008.

[10] J. Brzeziński, C. Sobaniec, and D. Wawrzyniak, "From session causality to causal consistency," in *Proc. of the 12th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (PDP2004)*, (A Coruña, Spain), pp. 152–158, Feb. 2004.

[11] J. Brzeziński, C. Sobaniec, and D. Wawrzyniak, "Session guarantees to achieve PRAM consistency of replicated shared objects," in *Proc. of the Fifth Int. Conf. on Parallel Processing and Applied Mathematics (PPAM'2003), LNCS 3019*, (Częstochowa, Poland), pp. 1–8, Sept. 2003.

[12] D. Gifford, "Weighted voting for replicated data," in *Proc. of the 7th ACM Symp. on Operating Systems Principles (SOSP)*, (Pacific Grove, USA), pp. 150–162, Dec. 1979.

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