Checkpointing and Rollback-Recovery Protocol for Mobile Systems with MW Session Guarantee*

Jerzy Brzeziński, Anna Kobusińska, and Michał Szychowiak

Poznań University of Technology
Institute of Computing Science
ul. Piotrowo 2,60-965 Poznań, Poland
{Jerzy.Brzezinski,Anna.Kobusinska,Michal.Szychowiak}@cs.put.poznan.pl

Abstract

In the mobile environment, weak consistency replication of shared data is the key to obtaining high data availability, good access performance, and good scalability. Therefore new class of consistency models, called session guarantees, recommended for mobile environment, has been introduced. Session guarantees, called also client-centric consistency models, have been proposed to define required properties of the system regarding consistency from the client's point of view. Unfortunately, none of proposed consistency protocols providing session guarantees is resistant to server failures. Therefore, in this paper checkpointing and rollback-recovery protocol rVsMW, which preserves Monotonic Writes session guarantee is presented. The recovery protocol is integrated with the underlying consistency protocol by integrating operations of taking checkpoints with coherence operations of VsSG protocol.

1. Introduction

In the recent years, there has been a lot of work dealing with mobile computing, mobility management, disconnected operations and distributed algorithms for mobile hosts. Also, the research addressing information acccess in mobile environment has proliferated. Following their line of investigation, researcheres are unanimous, that mobility gives the opportunity to provide new services and allows the supplementary access to data and services. A key concept in providing high performance and availability in such access is replication. Replication introduces, however, the problem of data consistency that arises when replicated objects are modified. This problem is directly related to the question, how results of operations executed concurrently on different replicas of the same object (data, service) can be perceived.

The properties of distributed system concerning consistency depend in general on application and are formally specified by consistency models. Numerous consistency models have been proposed for Distributed Shared Memory systems. These models, called data-centric consistency models [9], 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. This switching adds a new dimension of complexity to the problem of consistency. Session guarantees [10], called also client-centric consistency models[9], have been proposed to define required properties of the system regarding consistency from the client's point of view. They are recommended for mobile environment, where weak consistency replication of shared data is the key to obtaining high data availability, good access performance, and good scalability. Four session guarantees have been defined: *Read Your Writes* (RYW), *Monotonic Writes* (MW), *Monotonic Reads* (MR) and *Writes Follow Reads* (WFR).

The dependability requirements of mobile applications are continually growing, thus, the existing consistency protocols which provide session guarantees [6, 10, 8, 9] should be provided with the fault–tolerant

*This work was supported in part by the State Committee for Scientific Research (KBN), Poland, under grant KBN 3 T11C 073 28

1-4244-0054-6/06/$20.00

techniques. The popular fault–tolerant techniques are checkpointing and rollback–recovery, which will allow servers to provide required session-guarantees despite their failures. However, because of mobility of users, the traditional fault-tolerance schemes cannot be directly applied to mobile systems. Moreover, because of client orientation of most mobile systems, run-time faults must be corrected with any intervention from the user. The fault-tolerance capability must be therefore, self-contained.

According to our knowledge, none of existing consistency protocols for mobile environment which preserve session guarantees [2, 10], is fault-tolerant. The lack of consistency protocols optimized in terms of rollback-recovery, makes the construction of effective solutions adjusted to real applications requirements more difficult.

For this reason, in this paper a checkpointing and rollback-recovery protocol rVsMW, that preservs Monotonic Writes session guarantee is presented. The proposed protocol takes advantage of the underlying VsSG consistency protocol proposed in [6, 8]. The checkpointing and rollback- recovery protocol integrates operations of taking checkpoints with coherence operations of VsSG protocol. As a result, the rVsSG protocol offers the ability to overcome the servers' failures, at the same time preserving MW session guarantee.

2. Basic definitions and problem formulation

2.1. System model

Throughout this paper, a replicated distributed storage system is considered. The system consists of a number of unreliable *servers* holding a full copy of a set of data items and *clients* running applications that access these data items. We assume the *crash-recovery* model of failures, i.e. servers may crash and recover after crashing a finite number of times [5]. Servers can fail at arbitrary moments and we require any such failure to be eventually detected, for example by failure detectors [7]. Clients are mobile, i.e. they can switch from one server to another. Since clients are separated from one another, a crash of one client does not influence the processing of other clients. For that reason, in this paper we consider only failures of servers.

Clients are separated from servers, i.e. a client's application may run on a separate computer than the server. To access shared data, clients select a single server and send a direct request to this server. Operations are issued by clients synchronously, i.e. a new operation may be issued after the results of the previous one have been obtained.

The storage replicated by servers does not imply the particular data model or organization. It is a set of data items, which may be simple variables, files, objects of object-oriented programming language, etc. Later in the paper, the data items are referred to as *objects*. Operations performed on shared objects are divided into *reads* and *writes*. Reads do not change the state of objects, whereas writes may create a new object, delete the existing one or cause an update of the object state. Some clients can concurrently submit conflicting writes on different servers, e.g. writes that modify the overlapping parts of data storage. From now on, writes, whose order of execution need not be preserved by all servers, will be called *commutative* and those, whose execution order is required to be the same on all servers, will be called *non-commutative.*

Servers occasionally synchronize states of their replicas by exchanging information about writes performed in the past. As a result, all writes submitted by clients are eventually propagated and executed by every server.

2.2. Notation and basic definitions

Operations on shared objects issued by client C_i are ordered by relation $\stackrel{C_i}{\rightarrow}$ called *client issue order*. As client processes are sequential, relation $\stackrel{C_i}{\rightarrow}$ determines total order on the set of operations issued by C_i. Operations performed by server S_j are ordered by relation $\stackrel{S_j}{\rightarrowtail}$, called *server execution order*. As the operations are performed one at a time, the server execution order is also totally ordered. Operations on objects are denoted by w, r or o, depending on an operation type (write, read or these whose type is irrelevant). The operation performed by server S_j will be denoted by $o|_{S_j}$, the operation performed on object x will be denoted by $o|_x$. Every server maintains the set CW_{S_j} of indexes of clients from which it has directly received write requests.

In the paper, it is assumed that clients perceive the data from the replicated storage according to Monotonic Writes session guarantee. MW session guarantee orders writes issued by a single client. A server, before accepting a new write from a client, must perform all previous writes requested by this client. Formally, MW session guarantee is defined as follows [8]:

Definition 1 *Monotonic Writes (MW) session guarantee is a property meaning that:*

$$\forall C_i \forall S_j \left[w_1 \overset{C_i}{\rightarrow} w_2|_{S_j} \implies w_1 \overset{S_j}{\rightarrowtail} w_2 \right]$$

$\mathcal{O}_{S_j}$ is a set of all writes performed by the server in the past. The writes that belong to $\mathcal{O}_{S_j}$ come from direct requests received by S_j from clients or are incorporated from other servers during the synchronization procedure. The sequence of past writes is called *history*. A formal definition of history is given below:

Definition 2 *A history H_{S_j} at time moment t, is a linearly ordered set $\left(\mathcal{O}_{S_j}, \overset{S_j}{\rightarrowtail}\right)$ where $\mathcal{O}_{S_j}$ is a set of writes performed by server S_j, till the time t and relation $\overset{S_j}{\rightarrowtail}$ represents an execution order of writes.*

During synchronization of servers, their histories are *concatenated*. Informally, a concatenation of two histories is a sum of writes from the first history and new writes from the second history. Formally:

Definition 3 *Concatenation of two histories $\left(\mathcal{O}_{S_j}, \overset{S_j}{\rightarrowtail}\right)$ and $\left(\mathcal{O}_{S_k}, \overset{S_k}{\rightarrowtail}\right)$, denoted by $\left(\mathcal{O}_{S_j}, \overset{S_j}{\rightarrowtail}\right) \oplus \left(\mathcal{O}_{S_k}, \overset{S_k}{\rightarrowtail}\right)$, is a history $\left(\mathcal{O}_{S_j} \cup \mathcal{O}_{S_j}, \overset{S'_j}{\rightarrowtail}\right)$, where relation $\overset{S'_j}{\rightarrowtail}$ has the following properties:*

1. $\forall w_1, w_2 \in \mathcal{O}_{S_j} : w_1 \overset{S_j}{\rightarrowtail} w_2 \Rightarrow w_1 \overset{S'_j}{\rightarrowtail} w_2$
2. $\forall w_1, w_2 \in \mathcal{O}_{S_k} \setminus \mathcal{O}_{S_j} : w_1 \overset{S_k}{\rightarrowtail} w_2 \Rightarrow w_1 \overset{S'_j}{\rightarrowtail} w_2$
3. $\forall w_1 \in \mathcal{O}_{S_j} \forall w_2 \in \mathcal{O}_{S_k} \setminus \mathcal{O}_{S_j} : w_1 \overset{S'_j}{\rightarrowtail} w_2$

2.3. General concept of the underlying VsSG coherency protocol

Data consistency in the paper is managed by the VsSG *consistency protocol* [2, 6, 8]. The VsSG protocol interacts with *requests* sent from clients to servers and with *replies* sent from servers to clients.

The underlying consistency protocol uses a concept of server-based version vectors for efficient representation of sets of writes required by clients and necessary to check on the server side. Server-based version vectors have the following form: $V = \left[\begin{array}{cccc} v_1 & v_2 & \ldots & v_{N_S} \end{array} \right]$, where N_S is a total number of servers in the system and single position v_i is the number of writes performed by server S_j.

Every write in the VsSG protocol is labeled with a *vector timestamp*, set to the current value of the vector clock V_{S_j} of server S_j performing the write for the first time. The vector timestamp of a write w is returned by a function $T : \mathcal{O} \mapsto V$. A single i-th position of the version vector timestamp associated with write w is denoted by $T(w)[i]$. During writes performed by server S_j, its version vector V_{S_j} is incremented at position j and a timestamped operation is recorded in history H_{S_j}.

As opposed to [10], the VsSG consistency protocol does not assume total ordering of non-commutative writes, but only the eventual total propagation of all writes to all servers.

The request sent from a client C_i to a server S_j carries the operation that is to be performed and vector W. Vector W is calculated according to the operation type (read or write). W is set either to vector W_{C_i}— representing writes issued by the client C_i, in case of write operation, or to 0 when read operation is required.

On receipt of write request sent by a client, the server S_j checks whether it has performed all previous writes requested by the client, which is expected to be sufficient for providing MW. If the state of the server is not sufficiently up to date, the request is postponed and will be resumed after the synchronization with another server.

The server which first obtains the write from a client is responsible for assigning it a globally unique identifier. The current value of the server vector clock is returned to the client and causes the update of the client's vector W_{C_i}.

Every server periodically sends an update message with its own history to all the other servers. On receipt of such an update message, a server performes writes stored in the obtained history and missing from its local one. An update message also updates the vector clock of server S_j by calculating the maximum of the vector clock sent with the update message and V_{S_j} of S_j. The maximum of two vector clocks V_1 and V_2 is vector clock V, for which $\forall i : V[i] = \max(V_1[i], V_2[i])$ holds.

2.4. Checkpoint and log definitions

The VsSG coherency protocol assumes that servers are reliable, i.e. they do not crash. Such assumption might be consider not plausible and too strong for certain mobile distributed systems. Therefore, in this paper we introduce the checkpointing and rollback–recovery protocol integrated with VsSG protocol, called rVsMW. The rVsMW protocol ensures that, after the server failure and its recovery, the MW

session guarantee is preserved. Below, we propose formal definitions of mechanisms used by the checkpointing and rollback-recovery protocol:

Definition 4 *A log Log_{S_j} is a set of triples:*

$$\{\ \langle i_1, o_1, T(o_1)\rangle \quad \langle i_2, o_2, T(o_2)\rangle \quad ... \quad \langle i_n, o_n, T(o_n)\rangle\ \},$$

where i_n represents the index of the client that issued a write operation o_n, $i_n \in 1..N_C$ and N_C is a number of clients in the system. The operation $o_n \in \mathcal{O}_{S_j}$ and $T(o_n)$ is its timestamp.

During a rollback-recovery procedure, operations from the log are executed according to their timestamps, from the earliest to the latest one.

Definition 5 *A checkpoint $Ckpt_{S_j}$ is a couple $\langle V_{S_j}, H_{S_j}\rangle$, of a version vector V_{S_j} and a history H_{S_j} maintained by server S_j at the time t, where t is a moment of taking a checkpoint.*

Checkpoints taken by a server are numbered. The n-th checkpoint of server S_j is denoted by $Ckpt^n_{S_j}$.

By $V_{Ckpt^n_{S_j}}$ and $H_{Ckpt^n_{S_j}}$ we denote the version vector V_{S_j} and the history H_{S_j} kept in a checkpoint $Ckpt^n_{S_j}$.

The log and the checkpoint are saved by the server in the stable storage, which is able to survive failures. The newly taken checkpoint replaces the old one, so just one checkpoint is kept for each server.

3. The rVsMW protocol

3.1. The general idea

If the client C_i requires MW session guarantee when executing write w, then results of all writes preceding w in a client issue order cannot be lost. However, when MW guarantee is required and write operation issued by a client C_i will not be followed by another one, then the results of the first write are not essential for preserving MW for C_i. Unfortunately, at the moment of performing the operation, the server does not possess the knowledge, whether in the future a client will issue another write request, or not.

So, to preserve the MW session guarantee, the rollback-recovery protocol must ensure that the results of all writes issued by the client are not lost after the server failure and its recovery. Taking a checkpoint on every write operation fulfills this requirement, but results in frequent saving of whole server state in the stable storage, which is time–consuming. The proposed logging procedure overcomes this disadvantage, by saving in the stable storage only th eoperation and its timestamp, and thus takes less time than checkpointing. On the other hand, the log size may grow infinitely and may turn out to be too large. To bound the length of a message log (and hence a recovery time), a server may periodically take a checkpoint of its state. Storing the write operations doubly: in the log and in the checkpoint, although seems to be excessive, leads in fact to protocol optimization. In such a way the stable storage space is saved because of checkpointing and less time is spent to store servers' states during logging.

Thus, in the proposed rVsMW protocol, the server which obtains the write request directly from a client, logs this request to stable storage. According to a definition (definition 4), such a log is a triple of write operation, unique operation timestamp and the index of client that issued the operation. Having logged the operation, the server performs the just logged write and continues its processing. Writes received from other servers during synchronization procedure only cause servers' state update and are not logged.

The moment of taking a checkpoint is determined by obtaining a second write request from the same client. The server which obtains write operation from a client, checks whether the write can be performed according to MW. If it is possible, the server checks whether, since the latest checkpoint, it has already already performed a write issued by this client. If such a write has been performed, the server checkpoints its state, performs write operation and sends a reply to a client. Otherwise, the new checkpoint need not be taken. After the checkpoint is taken, server logs are cleared.

After the failure occurrence, the failed server restarts from the latest checkpoint and replays operations from the log to restore the execution to a state before the failure.

Log-based recovery is widely studied in the context of process based systems with asynchronous message passing [4, 3], where three flavors of these protocols are considered: pessimistic, optimistic and causal logging [1]. The protocol presented in this paper has features in general similar to pessimistic message logging. However, in contrast to systems with message–passing, we consider the interaction between the client and the server, not between the servers. This is a novel feature, that follows directly from the session guarantees assumptions which are client–oriented. Moreover, in contrast to systems with message–passing, we also take into account the semantics of operations.

3.2. The rVsMW protocol implementation

The request sent from client C_i to server S_j carries the operation that is to be performed and a vector W that is calculated depending on the operation type which value is returned by the function `iswrite`(o) (line 2). W is set to $\mathbf{0}$ (line 1) or to W_{C_i}(line 3) for reads and writes respectively. Afterwards, the modified message $\langle o, i, W\rangle$ is send to a server (line 5).

Upon receiving a new write request from client C_i, server S_j checks whether it has performed all writes issued previously by client C_i, by comparing version vectors V_{S_j} and W (line 6)[8]. If the state of server S_j is not sufficiently up to date, the request is postponed because MW session guarantee is not fulfilled (line 7). The request will be resumed after synchronization with another server (line 45).

Moreover, the server checks if, since the latest checkpoint, it has already performed any write submitted by C_i (line 10), by checking the value of variable $secondWrite$. At the begining, the value of $secondWrite$ is FALSE.

If the obtained write is the first write request received by S_j from a given client C_i since the latest checkpoint, then the server S_j stores the identifier of client issuing a request by adding clients' index to the set CW_{S_j} (line 13). Otherwise, the server sets the value of the variable $secondWrite$ to TRUE (line 11), which indicates that S_j has already performed a write issued by C_i after the previous checkpoint was taken. In both cases, S_j updates its data structures: increases the value of its version vector V_{S_j} and timestamps the operation o to give o a unique identifier (lines 15-16). Before performing operation o, S_j logs data necessary to recover its state in case of the failure occurrence. Thus, client's index i, operation o and its timestamped $T(o)$ are logged to the stable storage (line 17). When the information necessary for rollback-recovery is logged, the server performs the clients' request (line 18) and adds it to its history of performed writes (line 19). Finally, if just performed write is the second write request received by S_j directly form C_i, the state of the server is checkpointed (line 21). By definition, the checkpoint contains the version vector and the history of S_j. Checkpointing the server state causes clearing log Log_{S_j} and set CW_{S_j} (lines 22-23), as well as setting the value of variable $secondWrite$ to FALSE (line 24).

When the read request from client C_i is received by server S_j (line 27), then the request is performed (line 28) and the results are send to client C_i (line 30).

The update message received from other servers changes the state of server S_j, only if the history H contains writes not performed by S_j yet (line 39). Ev-

Algorithm 1 rVsMW algorithm.

Upon sending a request $\langle o\rangle$ to server S_j at client C_i
1: $W \leftarrow \mathbf{0}$
2: **if** `iswrite`(o) **then**
3: $\quad W \leftarrow \max(W, W_{C_i})$
4: **end if**
5: send $\langle o, i, W\rangle$ to S_j

Upon receiving a request $\langle o, i, W\rangle$ from client C_i at server S_j
6: **while** $\left(V_{S_j} \not\geq W\right)$ **do**
7: $\quad$ `wait`()
8: **end while**
9: **if** `iswrite`(o) **then**
10: $\quad$ **if** $i \in CW_{S_j}$ **then**
11: $\quad\quad secondWrite \leftarrow TRUE$
12: $\quad$ **else**
13: $\quad\quad CW_{S_j} \leftarrow CW_{S_j} \cup i$
14: $\quad$ **end if**
15: $\quad V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
16: $\quad$ timestamp o with V_{S_j}
17: $\quad Log_{S_j} \leftarrow Log_{S_j} \cup \langle i, o, T(o)\rangle$
18: $\quad$ perform o and store results in res
19: $\quad H_{S_j} \leftarrow H_{S_j} \oplus \{o\}$
20: $\quad$ **if** $secondWrite$ **then**
21: $\quad\quad Ckpt_{S_j} \leftarrow \langle V_{S_j}, H_{S_j}\rangle$
22: $\quad\quad Log_{S_j} \leftarrow \emptyset$
23: $\quad\quad CW_{S_j} \leftarrow \emptyset$
24: $\quad\quad secondWrite \leftarrow FALSE$
25: $\quad$ **end if**
26: **end if**
27: **if not** `iswrite`(o) **then**
28: $\quad$ perform o and store results in res
29: **end if**
30: send $\left\langle o, res, V_{S_j}\right\rangle$ to C_i

Upon receiving a reply $\langle o, res, W\rangle$ from server S_j at client C_i
31: **if** `iswrite`(o) **then**
32: $\quad W_{C_i} \leftarrow \max(W_{C_i}, W)$
33: **end if**
34: deliver $\langle res\rangle$

Every Δt at server S_j
35: **foreach** $S_k \neq S_j$ **do**
36: $\quad$ send $\left\langle S_j, H_{S_j}\right\rangle$ to S_k
37: **end for**

Upon receiving an update $\langle S_k, H\rangle$ at server S_j
38: **foreach** $w_i \in H$ **do**
39: $\quad$ **if** $V_{S_j} \not\geq T(w_i)$ **then**
40: $\quad\quad$ perform w_i
41: $\quad\quad V_{S_j} \leftarrow \max\left(V_{S_j}, T(w_i)\right)$
42: $\quad\quad H_{S_j} \leftarrow H_{S_j} \oplus \{w_i\}$
43: $\quad$ **end if**
44: **end for**
45: `signal`()

On rollback-recovery
46: $\langle V_{S_j}, H_{S_j}\rangle \leftarrow Ckpt_{S_j}$
47: $Log^{'}_{S_j} \leftarrow Log_{S_j}$
48: $vrecover \leftarrow \mathbf{0}$
49: **foreach** $o^{'}_j \in Log^{'}_{S_j}$ **do**
50: **choose** $\langle i^{'}, o^{'}_i, T(o^{'}_i)\rangle$ **with minimal** $T(o^{'}_j)$ **from** $Log^{'}_{S_j}$ **where** $T(o^{'}_j) > V_{S_j}$
51: $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
52: perform $o^{'}_j$
53: $H_{S_j} \leftarrow H_{S_j} \oplus \{o^{'}_j\}$
54: $CW_{S_j} \leftarrow CW_{S_j} \cup i^{'}$
55: $vrecover \leftarrow T(o^{'}_i)$
56: **end for**
57: $secondWrite \leftarrow FALSE$

ery such write w_i, after being performed (line 40), is added to the history H_{S_j} (line 42) and its timestamp updates the version vector V_{S_j} (line 41).

After the failure, the server state is recovered according to the information remembered in the latest checkpoint $Ckpt_{S_j}$ of server S_j (line 46) and in log Log_{S_j}, which keeps the information about writes, that were performed after the latest checkpoint has been taken and before the failure occurred (lines 49-57). Operations from the log are reexecuted, according to their timestamps, from the earliest to the latest one. The timestamp of just recovered operation is stored in the vector R_{S_j} (line 48). The next operation chosen from the log to be recovered, has the minimal timestamp, which is greater than the one stored in R_{S_j} (line 49). According to recovered operation, the vector V_{S_j} is incremented at the position j by 1 (line 51) and CW_{S_j} (line 54) as well as history H_{S_j} (line 53) are recovered.

Operations received during synchronization procedure are lost. However, by the assumption, writes obtained during synchronization procedures are saved in the stable storage (in the log or in the checkpoint) of servers which received them directly from clients. Hence, we notice, that the loss of writes obtained during synchronization procedure from other servers, does not violate the MW session guarantee, because such writes will be obtained again in consecutive synchronizations.

3.3. Analysis of rollback-recovery mechanism

The recovery-based approach ensures the reliability of the system by restoring values of the data lost in such a way, that after the recovery the system remains in a consistent state, according to the assumed consistency model.

Section 3.1 presents the general idea of the recovery mechanism applied in the proposed rVsMW protocol. In this section, we analyze this mechanism in details by considering various possible moments of server failures and discussing the actions, that are taken when such failures occur.

First, let us assume, that server S_j obtains the write request w_1 from client C_i, and crashes before updating its data structures (between lines 6 and 13 of rVsMW protocol). In such a situation, the client will not get the results of its request and, after the predefined timeout, it issues write w_1 again. After the failure and rollback-recovery, server S_j does not possess any information on write w_1, as its state is recovered according to the data stored in the latest checkpoint and in the log. Thus, performing w_1 by S_j once again does not violate MW session guarantee, as w_1 is treated by the server as the request received from C_i for the first time.

The server failure that happens after: updating the value of variable $secondWrite$, storing the client identifier in set CW_{S_j}, increasing the number of performed writes or timestamping operation w_1 (i.e. respectively after lines 11, 13, 15 and 16 of the protocol), also forces the client to issue the request w_1 once again. In this situation, although servers' data structures were updated before the failure, after the rollback-recovery the carried out modifications are lost. Therfore, another execution of request w_1 does not violate the MW. The order of operations described in lines 11, 13, 15 and 16 of the protocol is not crucial to the recovery mechanism.

The operation w_1 that has been logged before the failure of S_j occurred, (the server failure took place after lines 17, 18 or 19 of the protocol) is performed again during the recovery and execution of operations saved in log Log_{S_j}. In this case, it is important that logging of write w_1 took place before performing this request. Such an order is crucial because, if the operation is performed but not logged, it could be lost in the case of a failure.

Let us now assume, that server S_j receives another write request w_2 from client C_i. If the server failure occurs after the request has been obtained, but before it has been checkpointed (before line 21), then the write w_2 is recovered from the log, similarly to write request w_1. On the contrary, if the new checkpoint is taken, then according to the protocol such a checkpoint contains the most recent values of vector V_{S_j} and history H_{S_j}. Thus, in case of failure, the checkpoint contains all data necessary to recover the server state according to MW. Essential here is the fact, that first the checkpoint is taken, and only afterwards the content of log Log_{S_j} is cleared.

The failure may occur just after the checkpoint is taken, but before the log is cleared (i.e. between lines 21 and 22). Such a failure causes that operations remembered in the checkpoint are also stored in the log. As a result, they might be performed twice: first time during rolling back to the checkpoint and secondly, during recovering operations from the log. Against this situation, the condition in line 49 of the algorithm is given, which allows to recover on the basis of the log only those operations, for which $T(o_j^{|}) > V_{S_j}$ holds.

Failures, which occur during the synchronization procedure, are not important from our point of view. Even, if results of such synchronization are lost, they can be obtained and applied again during next synchronization with servers which had checkpointed or logged these operations.

Finally, let us assume that the server fails during the rollback-recovery procedure. If the failure happens straight after the server state has been recovered (after line 46), then it has to be rolled back to this checkpoint again. If the failure occurs during data recovering from the log (anywhere between lines 49-56), then the results of operations executed according to the log, are lost. However, due to such a failure the recovery action is just prolonged, as the server must be rolled back again and operations from the log have to be executed from the beginning. The repeated recovery procedure works correctly, as the content of the checkpoint and the log are not changed (line 47).

3.4. The example of applying the protocol

Below, the example of a computation is presented, in which the server fails due to failures and is recovered on the basis of the proposed protocol .

The system is assumed to consist of two clients C_1, C_2 and two servers S_1, S_2 that maintain x and y object replicas. The initial consistent values of both x and y are 0. The values of version vectors V_{S_1} and V_{S_2} are 0. The checkpoints of both servers are $\langle \mathbf{0}, \emptyset \rangle$. The following scenario is considered: client C_1 issues a write request $w(x)2$. The request is received by server S_1, which logs the information about the write issued by C_1 and performs it. Therefore, the state of the server is the following: $V_{S_1} = [1,0]$, $H_{S_1} = \langle w(x)2 \rangle$, $Log_{S_1} = \langle 1, w(x)2, [1,0] \rangle$. After the write has been stored in the log and performed, S_1 sends the reply to client C_1 that sets client vector W_{C_1} to $[1,0]$. Then, a write $w(y)1$ is issued by C_2 and received again by server S_1. After logging and performing this request, the server state is as follows: $V_{S_1} = [2,0]$, $H_{S_1} = \langle w(x)2, w(y)1 \rangle$, $Log_{S_1} = \{\langle 1, w(x)2, [1,0] \rangle, \langle 2, w(y)1, [2,0] \rangle\}$ and the client vector $W_{C_2} = [2,0]$. Another write request issued by C_1 is received by server S_2. After logging and performing this request, the server state is $V_{S_2} = [0,1]$, $H_{S_2} = \langle w(x)1 \rangle$, $Log_{S_2} = \{\langle 1, w(x)1, [0,1] \rangle\}$ and $W_{C_1} = [1,1]$.

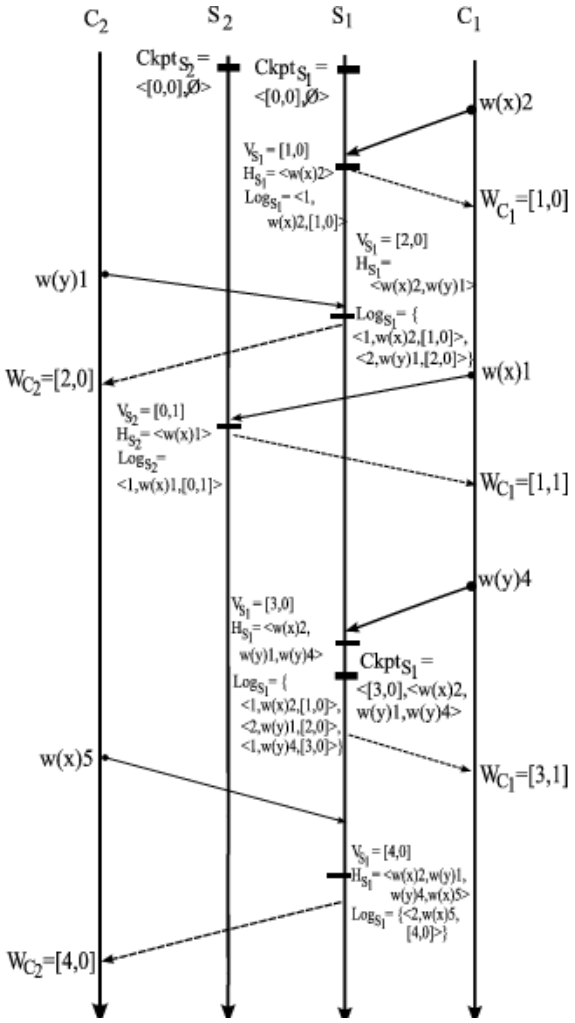

Figure 1. The checkpointing and rollback-recovery protocol.

In the meantime, another write $w(y)4$ issued by C_1 is obtained by S_1. The state of the server after performing the new write is the following: $V_{S_1} = [4,0]$, $H_{S_1} = \langle w(x)2, w(y)1, w(y)4, w(x)5 \rangle$, $Log_{S_1} = \{\langle 1, w(x)2, [1,0] \rangle, \langle 2, w(y)1, [2,0] \rangle, \langle 1, w(y)4, [3,0] \rangle\}$. Since $w(y)4$ is a second write issued by C_1 and obtained by S_1, S_1 takes a checkpoint $\langle [3,0], \{w(x)2, w(y)1, w(y)4\} \rangle$. After the checkpoint is taken, log of S_1 is cleared.

Finally, the write request send by C_2 is performed by the server S_1 and as a result $V_{S_1} = [4,0]$, $H_{S_1} = \langle w(x)2, w(y)1, w(x)5 \rangle$, $Log_{S_1} = \{\langle 2, w(x)5, [4,0] \rangle\}$.

4. rVsMW optimizations

Since in rVsMW protocol only write operations obtained directly from the client are logged, in the case of server failure, results of all writes performed by S_j during synchronization procedure are lost. However, there might happen a situation, when the results of lost write are required to fulfill the MW session guarantee.

For example, if client C_i requiring MW has issued a write w_1 which has been performed by server S_k, then another server S_j before performing next write w_2 issued by the same client, should have performed w_1 first.

If synchronization of servers has took place, but, because of S_j failure w_1 has been lost, then w_2 will be postponed (line 7) until the time of the next synchronization (line 45).

By computing the difference of the timestamp of the latest operation recovered from the log, saved in the R_{C_i} and the value of recovered vector V_{S_j}, the information on servers from which synchronization messages were received, can be obtained. Thus, to shorten the awaiting time for synchronization procedure, the synchronization on demand may be applied. In such a synchronization the request for update message is sent to servers, from which synchronization messages were obtained, but results of synchronization have been lost.

5. Conclusions

Applications in the mobile domain usually tend to be structured as client-server interactions. For such applications, the management of data consistency from the clients' perspective seems to be very attractive.

According to our knowledge, although several studies have examined the issues of checkpointing, logging and rollback-recovery in mobile systems, none of the existing solutions integrates these issues with the consistency protocol. Especially client-centric consistency models, have not been considered in the context of rollback-recovery.

Therefore, this paper addresses a problem of integrating the consistency management of the mobile system with the recovery mechanism. We introduce the rollback-recovery protocol rVsMW for distributed mobile systems, which preserve Monotonic Writes session guarantee. The proposed recovery protocol is integrated with the underlying VsSG consistency protocol.

The rVsMW protocol, in contrast to systems with message–passing, takes into account the semantics of operations during the rollback-recovery procedure. This results in checkpointing only results of every second write operation received from the same client. Moreover, the proposed rVsMW protocol combines logging and checkpointing in order to save space in the stable storage and spend less time while storing operations. As a result, rVsMW protocol offers the ability to overcome the servers' failures and preserves MW session guarantee, in the optimized way.

The proposed protocol preserves one of four existing session guarantees. Thus, our future work encompasses the development of rollback-recovery protocols that are integrated with VsSG consistency protocol, and preserve other session guarantees.

References

[1] L. Alvasi and K. Marzullo. Message logging: pessimistic, optimistic, causal and optimal. *IEEE Trans. Softw. Eng*, 24(2):149–159, 1998.

[2] J. Brzeziński, C. Sobaniec, and D. Wawrzyniak. Safety of a server-based version vector protocol implementing session guarantees. *Proc. of Int. Conf. on Computational Science (ICCS2005),LNCS 3516*, pages 423–430, May 2005.

[3] N. Elmootazbellah, Elnozahy, A. Lorenzo, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[4] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computer*, 41(5):526–531, May 1992.

[5] R. Guerraoui and L. Rodrigues. Introduction to distributed algorithms. 2004.

[6] A. Kobusińska, M. Libuda, C. Sobaniec, and D. Wawrzyniak. Version vector protocols implementing session guarantees. *Proc. of Int. Symp. on Cluster Computing and the Grid (CCGrid 2005)*, May 2005.

[7] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. (SSC/1999/019), May 1999.

[8] C. Sobaniec. Consistency protocols of session guarantees in distributed mobile systems. Sept. 2005.

[9] A. S. Tanenbaum and M. van Steen. Distributed systems — principles and paradigms. 2002.

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