Bounding Recovery Time in Rollback-Recovery Protocol for Mobile Systems Preserving Session Guarantees

Jerzy Brzeziński, Anna Kobusińska, and Jacek Kobusiński

Institute of Computing Science
Poznań University of Technology, Poland
{Jerzy.Brzezinski, Anna.Kobusinska, Jacek.Kobusinski}@cs.put.poznan.pl

Abstract. This paper addresses a problem of integrating the consistency management of session guarantees with recovery mechanisms in distributed mobile systems. To solve such a problem, rollback-recovery protocol rVsSG, preserving session guarantees is proposed. The protocol employs known rollback-recovery techniques, however, while applying them, the semantics of session guarantees is taken into account. Consequently, rVsSG protocol is optimized with respect to session guarantees requirements. The paper includes the proof of safety property of the presented protocol.

Keywords: fault tolerance, rollback-recovery, mobile systems, session guarantees

1 Introduction

Mobile computing brings about a new paradigm of distributed computing in which communication may be achieved through wireless or intermittently connected networks. In this paradigm, users can compute even if they relocate from one distributed resource to another, using different links at different locations. By enabling motion and location independence, mobility gives the opportunity to provide new services and allows supplementary information access that may occur any time and any place. Although being a relatively new area, mobile computing has attracted a lot of research efforts, motivated by both a great market potential and by many challenging research problems.

The impact of mobile computing on systems design goes beyond the networking level and directly effects data access and management. A key concept in providing high performance and availability in such an access is replication. Unfortunately, replication brings up a problem of replica consistency. Moreover, due to switching of clients, this problem gains a new dimension of complexity and thus, it should be tackled from new, client's perspective. For that reason, *session guarantees* [TDP+94], also called *client-centric* consistency models, have been proposed to define required properties of the system observed from client's point of view. Four session guarantees have been defined: *Read Your Writes* (RYW), *Monotonic Writes* (MW), *Monotonic Reads* (MR) and *Writes*

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 187–198, 2006.

Follow Reads (WFR) and protocols implementing them have been introduced [TDP+94, BSW05, Sob05]. Unfortunately these protocols assume that clients and servers are reliable and they do not crash. In practice, failures do happen, therefore, the existing consistency protocols should be provided with the fault–tolerant techniques, which allow servers to provide required session-guarantees despite their failures.

Thus, this paper addresses a problem of integrating the consistency management of session guarantees in systems with mobile clients, with recovery mechanisms. To solve such a problem, we propose rVsSG protocol, integrating logging and checkpointing techniques with coherence operations of VsSG consistency protocol [TDP+94, BSW05, Sob05]. Consequently, the proposed protocol offers the ability to overcome servers' failures transparently to the client, and preserves session guarantees at the same time. The rVsSG protocol is optimized in terms of checkpointing overhead, taking into account the specific requirements of required session guarantees. Moreover, correctness analysis of the protocol is carried out and its safety property is formally proved.

2 Session Guarantees

2.1 Basic Assumptions and Definitions

Throughout this paper, a replicated distributed storage system is considered. The system consists of a number of *servers* holding a full copy of *shared objects* and *clients* running applications that access these objects (see Fig. 1). Although all system components (mobile clients, servers, communication links) can be a subject of failures, in this paper we focus only on failures of servers. We assume *crash-recovery* model of failures, i.e. servers may crash and recover after crashing a finite number of times [GR04]. Servers can fail at arbitrary moments and we require any such failure to be eventually detected, for example by failure detectors [SDS99].

Clients are separated from servers, i.e. a client application may run on a separate computer than the server. A client may access a shared object after selecting a single server and sending a direct request to the server. Clients are mobile,

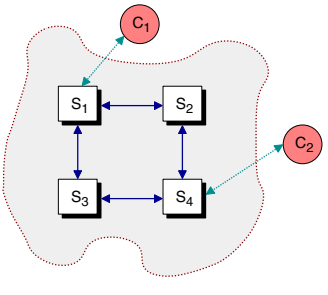

Fig. 1. Replication servers and client accessing them

i.e. they can switch from one server to another during application execution. Session guarantees are expected to take care of data consistency observed by a migrating client. The set of shared objects replicated by servers does not imply any particular data model or organization. Operations performed on shared objects are divided into *reads* and *writes*, denoted respectively by r and w. A read does not change states of the shared objects, while a write does. A write may cause an update of an object, it may create a new object, or delete an existing one. A write may also atomically update states of several objects.

Clients can concurrently submit conflicting writes at different servers, e.g. writes that modify the overlapping parts of data storage. Operations on shared objects issued by client C_i are ordered by a relation $\overset{C_i}{\rightarrow}$ called *client issue order*. Server S_j performs operations in an order represented by relation $\overset{S_j}{\rightarrowtail}$. An operation performed by a server S_j will be denoted by $w|_{S_j}$ or $r|_{S_j}$. Relevant writes $RW(r)$ of a read operation r is a set of writes that has influenced the current state of objects observed by the read r.

A set of basic consistency conditions for sessions of mobile clients has been introduced in Bayou project [TDP+94]. Informally, RYW expresses the user expectation not to miss his own modifications performed in the past, MW ensures that order of writes issued by a single client is preserved, MR ensures that the client's observations of the data storage are monotonic and WFR keeps the track of causal dependencies resulting from operations issued by a client. The following formal definitions, brought in [Sob05], are based on those descriptional concepts.

Definition 1. *Read Your Writes (RYW) session guarantee is defined as follows:*

$$\forall C_i \, \forall S_j \left[w \overset{C_i}{\rightarrow} r|_{S_j} \implies w \overset{S_j}{\rightarrowtail} r \right]$$

To illustrate RYW session guarantee, let us consider a user writing a TODO list to a file. After travelling to another location, the user wants to recall the most urgent tasks, and reads TODO list. Without RYW session guarantee the read may return any previous (possibly empty) version of the document.

Definition 2. *Monotonic Writes (MW) session guarantee is defined as follows:*

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

Let us consider a counter object with two methods for updating its state: `increment()`, and `set()`, which increment value of the counter, and set its new value, respectively. A user of the counter issues the `set()` function, 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.

Definition 3. *Monotonic Reads (MR) session guarantee is defined as follows:*

$$\forall C_i \, \forall S_j \left[r_1 \overset{C_i}{\rightarrow} r_2|_{S_j} \implies \forall w_k \in RW(r_1) : w_k \overset{S_j}{\rightarrowtail} r_2 \right]$$

In case of MR, let us consider a mailbox of a traveling user. The user opens the mailbox at one location, and reads emails. Afterwards he opens the same mailbox at different location, and expects to see at least all the messages he has read previously. The new state may not reflect the true current state, but must be at least as new as the previously observed version.

Definition 4. *Writes Follow Reads (WFR) session guarantee is defined as follows:*

$$\forall C_i \forall S_j \left[r \xrightarrow{C_i} w|_{S_j} \implies \forall w_k \in RW(r) : w_k \overset{S_j}{\rightarrowtail} w \right]$$

To present the usage of WFR, let us consider a user that reads a file with some information. Afterwards this user prepares some notes he wants to add to the document. Because, he was on journey, his computer switched in the meantime to another server. When the user finally wanted to perform the operation the new server was properly updated and user could append his note.

Besides Bayou, there are other systems that implement consistency conditions based on session guarantees: CASCADE — a caching service for distributed CORBA objects [CFV00], Pastis — a highly scable, multi-user, peer-to-peer file system [PBS05], or OceanStore — a global persistent data storage system [KBC+00].

2.2 The VsSG Consistency Protocol

Data consistency in rVsSG is managed by VsSG *consistency protocol* [BSW05], which uses a concept of server-based version vectors, having the following form: $V_{S_j} = \left[v_1 \; v_2 \; ... \; v_{N_S} \right]$, 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 VsSG protocol is labeled with a *vector timestamp*, set to the current value of vector clock V_{S_j} of server S_j, performing the write for the first time. The vector timestamp of write w is returned by function $T : \mathcal{O} \mapsto V$. All writes performed by the server in the past are kept in set $\mathcal{O}_{S_j}$. On the client's side, vectors W_{C_i} and R_{C_i} are maintained, representing writes issued by client C_i and writes relevant to reads issued by this client, respectively. The sequence of past writes is called *history*. A formal definition of history is given below:

Definition 5. *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.*

In order to satisfy the client's requirements concerning data consistency, the system intercepts client requests, and extends the standard communication. The request sent from a client C_i to a server S_j carries the operation that is to be performed and vector W, calculated according to the operation type and set SG of session guarantees required for the operation. W is set either to W_{C_i} or R_{C_i}.

On receipt of request sent by a client, server S_j checks whether for vectors V_{S_j} and W the following condition is fulfilled, $\forall i : V_{S_j}[i] \geq W[i]$, which is expected to be sufficient for providing appropriate session guarantee. If the state of the server is not sufficiently up to date, the request is postponed and will be resumed after synchronization with another server.

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}. 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}.

VsSG protocol eventually propagates all writes to all servers. During synchronization of servers, their histories are *concatenated*. The concatenation of histories H_{S_j} and H_{S_k}, denoted by $H_{S_j} \oplus H_{S_k}$, consists in adding new operations from H_{S_k} at the end of H_{S_j}, preserving at the same time the appropriate relations [BSW05].

2.3 Checkpoint and Log Definitions

Below, we propose formal definitions of fault-tolerance mechanisms used by rVsSG protocol:

Definition 6. *Log Log_{S_j} is a set of triples:*

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

where i_n represents the identifier of the client issuing a write operation $o_n \in \mathcal{O}_{S_j}$ and $T(o_n)$ is timestamp of o_n.

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

Definition 7. *Checkpoint $Ckpt_{S_j}$ is a couple $\left\langle V_{S_j}, H_{S_j} \right\rangle$, of version vector V_{S_j} and history H_{S_j} maintained by server S_j at time t, where t is a moment of taking a checkpoint.*

In this paper we assume, that log and checkpoint are saved by the server in the stable storage, able to survive all failures [EEL+02]. Additionally, we assume that the newly taken checkpoint replaces the previous one, so just one checkpoint for each server is kept in the stable storage.

3 The rVsSG Protocol

3.1 General Idea

To preserve required session guarantee, the rollback-recovery protocol must ensure that writes issued by client and essential to preserve this guarantee are not lost after the server failure and its recovery. Checkpointing every single write operation fulfills this requirement, but results in frequent saving of server state

in the stable storage, which is time–consuming. Logging procedure overcomes this disadvantage and takes less time than checkpointing. On the other hand, the log size may grow infinitely and may turn out to be too large. Thus, in the proposed protocol we use the known technique of combining logging and checkpointing. However, while applying these techniques, the semantics of operations, characteristic of session guarantees, is taken into account. Consequently, in rVsSG protocol, only operations essential to provide session guarantees are logged, so checkpoints are optimized with respect to required session guarantee requirements. Moreover, in rVsSG protocol, servers save only some of obtained operations, namely those received directly from clients. Operations obtained during synchronization procedure, even if required by session guarantee, are just performed by the server, because they have already been saved in the stable storage (in the log or in the checkpoint) of other servers. Hence, even if writes obtained in the result of synchronization procedure are lost, the required session guarantee is not violated. This steams from the fact, that such writes will be obtained again during consecutive synchronizations.

3.2 Protocol Implementation

Every request issued by client C_i indicates client's requirements for the issued operation that are calculated based on the type of operation (checked by `iswrite`(o) function), and set SG of session guarantees (lines 1, 3 and 6).

The server, which obtains the write request directly from client C_i, checks whether the request can be performed accordingly to required session guarantees (line 9). If the state of server S_j is not sufficiently up to date, the obtained request is postponed (line 10), otherwise server's S_j data structures are updated: the value of version vector V_{S_j} is increased and operation o is timestamped, to give o a unique identifier (lines 13-14). Afterwards, o is logged to stable storage (line 15). It is important that logging of write takes place before performing this request. Such an order is crucial, as the operation that is performed but not logged, may be lost in the case of subsequent failures. After the operation is performed (line 16), it is added to the history H_{S_j} of performed writes (line 17). With every logged operation the size of the log is increased, and thus a recovery takes more time. Therefore, in order to bound a recovery time after the server failure, the server state is occasionally checkpointed (line 19). The rVsSG protocol assumes, that a checkpoint is taken every K logged operations (line 18). K is constant and its value depends on the system characteristics. After the checkpoint is taken, log Log_{S_j} is cleared (line 20). Essential is the fact, that firstly the checkpoint is taken, and only afterwards the content of log Log_{S_j} is cleared.

The read request from client C_i received by server S_j is performed (line 24), if the condition from line (line 9) is fulfilled for this operation.

The update message received from other servers changes the state of server S_j, only if the history H contains writes that has not been performed by S_j yet (line 37). Such update operations are performed (line 38) and processed by S_j(lines 39-40).

Upon sending a request $\langle o \rangle$
to server S_j **at client** C_i

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

Upon receiving a request $\langle o, W, i \rangle$
from client C_i **at server** S_j

9: **while** $(V_{S_j} \not\geq W)$ **do**
10: `wait`()
11: **end while**
12: **if** `iswrite`(o) **then**
13: $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
14: timestamp o with V_{S_j}
15: $Logs_{S_j} \leftarrow Logs_{S_j} \cup \langle o, T(o) \rangle$
16: perform o and store results in res
17: $H_{S_j} \leftarrow H_{S_j} \oplus \{o\}$
18: **if** K operations is logged **then**
19: $Ckpts_{S_j} \leftarrow \langle V_{S_j}, H_{S_j} \rangle$
20: $Logs_{S_j} \leftarrow \emptyset$
21: **end if**
22: **end if**
23: **if** (not `iswrite`(o)) **then**
24: perform o and store results in res
25: **end if**
26: send $\langle o, res, V_{S_j} \rangle$ to C_i

Upon receiving a reply $\langle o, res, W \rangle$
from server S_j **at client** C_i

27: **if** `iswrite`(o) **then**
28: $W_{C_i} \leftarrow \max(W_{C_i}, W)$
29: **else**
30: $R_{C_i} \leftarrow \max(R_{C_i}, W)$
31: **end if**
32: deliver $\langle res \rangle$

Every Δt **at server** S_j

33: **foreach** $S_k \neq S_j$ **do**
34: send $\langle S_j, H_{S_j} \rangle$ to S_k
35: **end for**

Upon receiving an update $\langle S_k, H \rangle$
at server S_j

36: **foreach** $w_i \in H$ **do**
37: **if** $V_{S_j} \not\geq T(w_i)$ **then**
38: perform w_i
39: $V_{S_j} \leftarrow \max(V_{S_j}, T(w_i))$
40: $H_{S_j} \leftarrow H_{S_j} \oplus \{w_i\}$
41: **end if**
42: **end for**
43: `signal`()

On rollback-recovery
44: $\langle V_{S_j}, H_{S_j} \rangle \leftarrow Ckpt_{S_j}$
45: $Log'_{S_j} \leftarrow Logs_{S_j}$
46: $R_{S_j} \leftarrow \mathbf{0}$
47: **foreach** $o'_j \in Log'_{S_j}$ **do**
48: **choose** $\langle o'_i, T(o'_i) \rangle$ **with minimal** $T(o'_j)$ **from** Log'_{S_j} **where** $T(o'_j) > V_{S_j}$
49: $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
50: perform o'_j
51: $H_{S_j} \leftarrow H_{S_j} \oplus \{o'_j\}$
52: $R_{S_j} \leftarrow T(o'_i)$
53: **end for**

Fig. 2. Checkpointing and rollback-recovery rVsSG protocol

After the failure occurrence, the failed server restarts from the latest checkpoint (line 44) and replays operations from the log (lines 47-53) according to their timestamps, from the earliest to the latest one.

4 Safety of rVsSG Protocol

The safety property asserts that clients access object replicas maintained by servers according to required session guarantee, regardless of servers' failures.

Lemma 1. *Every write operation w issued by client C_i and performed by server S_j that received w directly from client C_i, is kept in checkpoint $Ckpt_{S_j}$ or in log Log_{S_j}.*

Proof. Let us consider a write operation w issued by client C_i and obtained by server S_j.

1. From the algorithm, server S_j before performing the request w, saves it in the stable storage by adding it to log Log_{S_j} (line 15). Because logging of w takes place before performing it (line 16), then even in the case of failure the operation w is not lost, but remains in the log.
2. Log Log_{S_j} is cleared (line 20) after taking by S_j the checkpoint (line 19). Therefore, the server failure that occurs after clearing the log does not affect safety of the algorithm, because writes from the log are already stored in the checkpoint.
3. After the checkpoint is taken, but before the log is cleared (between lines 19 and 20) writes issued by client C_i and performed by server S_j are stored in both the checkpoint $Ckpt_{S_j}$ and the log Log_{S_j}.

Lemma 2. *The rollback-recovery procedure recovers all write operations issued by clients, performed by server S_j and logged in log Log_{S_j} in the moment of failure occurrence.*

Proof. Let us assume that server S_j fails. After the failure, operations from the log are recovered (line 47), and cause the update of vector V_{S_j} (line 49). Afterwards they are performed by S_j (line 50) and added to history H_{S_j}(line 51). Assume now, that failures occur during recovery procedure. Due to such failures the results of operations that have already been recovered are lost again. However, since log Log_{S_j} is cleared (line 20) only after the checkpoint is taken (line 19) and it is not modified during the rollback-recovery procedure (line 45), log's content is not changed. Hence, the recovery procedure can be started from the beginning without loss of any operation issued by clients and performed by server S_j after the moment of taking checkpoint.

Lemma 3. *After the failure and recovery of server S_j, all write operations obtained during synchronization with other servers are performed by S_j again before applying new operations issued by a client and requiring results of lost operations to provide session guarantees.*

Proof. By contradiction, let us assume that server S_j has performed new operation o obtained from client C_i, before performing again operation w, received during a former synchronization with other servers and lost because of S_j failure. Due to underlying VsSG protocol [BSW05], the following condition must

be fulfilled (line 9) to perform operation o: $V_{S_j} \geq W$. More precisely, when o is a read operation required with RYW session guarantee, or a write operation requiring MW, above condition is equivalent to $\forall l : V_{S_j}[l] \geq W_{C_i}[l]$. In case of requiring by a client C_i MR guarantee, while issuing read operation or WFR, while issuing a write, the considered condition is $\forall l : V_{S_j}[l] \geq R_{C_i}[l]$.

Suppose that write operation w issued by client C_i has been performed by server S_k. After obtaining the reply from S_k, client C_i modifies its version vector W_{C_i} at least in position k: $W_{C_i} \leftarrow max(W_{C_i}, V_{S_k})$ (line 28). Server S_j, in the result of synchronization with S_k, performs w and updates its version vector V_{S_j}, modifying V_{S_j} at least in position k (line 39). Without loosing the generality, we assume that after performing operation w, server S_j has performed read operation r issued by C_i, which has read results of w. In the result, after obtaining results of r, client C_i has modified its version vector R_{C_i} at least in position k: $R_{C_i} \leftarrow max(R_{C_i}, V_{S_j})$ (line 30).

If the failure of S_j happens, the state of S_j is recovered accordingly to values stored in $Ckpt_{S_j}$ (line 44) and in Log_{S_j} (lines 47-53). During recovering operations from the log, vector V_{S_j} is updated only in position j. Thus, the recovered value of $V_{S_j}[k]$ does not reflect the information on w. Hence, until the next update message is obtained, $V_{S_j}[k] < W_{C_i}[k]$ and $V_{S_j}[k] < R_{C_i}[k]$, which contradicts the assumption.

Lemma 4. *The server performs new operation issued by a client C_i only after all writes issued by this client and performed before the failure are recovered.*

Proof. By contradiction, let us assume that server S_j has performed new operation o issued by client C_i, before recovering and performing again write operation w received directly from C_i and lost in the result of S_j failure. According to the underlying VsSG protocol, for server S_j performing new operation o the following condition must be fulfilled (lines 9-10): $V_{S_j} \geq W$, where W represents one of vectors: W_{C_i} or R_{C_i}, depending on the type of operation o and required session guarantee.

By assumption, after obtaining by server S_j write operation w, vector V_{S_j} is modified: $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$ and results of performed operation, together with vector V_{S_j} are sent to the client. At the client's side, after the reply is received, vector W_{C_i} is updated at least in position j : $W_{C_i} \leftarrow \max\left(W_{C_i}, V_{S_j}\right)$(line 28).

Without loosing the generality, let us assume that after performing operation w, server S_j has performed read operation r issued by C_i, which has read results of w. In the result, after obtaining results of r, client C_i has modified its version vector R_{C_i} at least in position j: $R_{C_i} \leftarrow max(R_{C_i}, V_{S_j})$ (line 30).

Thus, when operation w is not recovered after the server failure and its recovery, then either $V_{S_j}[j] < W_{C_i}[j]$ or $V_{S_j}[j] < R_{C_i}[j]$, which contradicts the assumption.

Theorem 1. *RYW, MW, WFR and MR session guarantees are preserved by rVsSG protocol for clients requesting them, even in the presence of server failures.*

Proof. According to Lemma 1, every write operation performed by server S_j is saved in the checkpoint or in the log. After the server failure, all operations from the checkpoint are recovered. Further, all operations performed before the failure occurred, but after the checkpoint was taken, are also recovered (following Lemma 2). As stated by Lemma 4, all recovered write operations are applied before new operation obtained from a client is performed. Hence, for any client C_i and any server S_j, required session guarantee is preserved by the rollback–recovery and checkpointing rVsSG protocol.

Full versions of presented theorems and proofs, and the proof of liveness property of proposed protocol can be found in [BKKS05].

5 Determining Desirable Moments of Taking Checkpoints

In general, checkpoints may be taken according to the following scenarios: periodically, every K operations issued by clients, or on the basis of semantics analysis.

In rVsSG protocol, described in section 3, server S_j takes a checkpoint after logging K operations. The value of constant K depends on system characteristics, among which are the frequency of requests issued by clients, or the complexity of issued operations. By adequate determining K, the checkpoint overhead, and thus total execution time of application may be minimized. The semantics analysis of session guarantees further minimizes the number of taken checkpoints, because it allows to avoid taking checkpoints that include operations not required by considered session guarantees.

To indicate sets of above operations, let us discuss the following four situations. When considering RYW, write operations issued by client C_i and not followed by read request issued by the same client, are not required to preserve this guarantee. Thus, when client C_i issues only writes and does not want to see their results, then checkpointing such writes is unnecessary. In case of MW, when write operation issued by client C_i is not followed by another write request, then results of first write are not essential to preserving MW from client's C_i point of view. Hence, the first write does not need not to be checkpointed. For MR session guarantee, set of writes, which results influenced read request issued by client C_i, does not need to be taken into account while taking a checkpoint, when such a read is not followed by a new one issued by the same client. Finally, when read request issued by a client C_i is not followed by write issued by the same client, then the results of writes that modified the state of server observed by read need not to be checkpointed if WFR is required by C_i.

Following above analysis, for each session guarantee, sets of operations essential to preserve this guarantee can be distinguished, and desirable moment of taking a checkpoint, denoted by DMTC, can be defined. DMTC indicates such a moment, before which there is no need to take a checkpoint, because the server has not performed any operation required by given session guarantee. For each session guarantee, we indicate operations that determine DMTC. In the case of RYW, it is a read request obtained from a client. For MR it is also a read

request, however the one that follows in the server execution order another read issued by the same client. With reference to MW session guarantee, DMTC is determined by obtaining a write request following in the server execution order all writes issued by the same client. Finally, for WFR session guarantee, it is a write request, following in the server execution order the read request issued by the same client.

Thus, the checkpointing and rollback-recovery protocol rVsSG, may be optimized, by taking checkpoints according to consecutive DMTC.

Below, we present an example of taking a checkpoint according to DMTC (Fig. **??**). In the considered example, a system consists of two servers and two clients. Client C_1, issues operations which should be performed according to MR session guarantee. Server S_1 takes a checkpoint according to idea of DMTC, i.e. when it obtains the second read request issued by C_i. Of course, depending on system characteristics, in general checkpoints can be taken every n-th DMTC. But, there is no need to take checkpoitns between two following DMTC.

6 Conclusions

This paper has dealt with a problem of integrating the consistency management of distributed systems with mobile clients with the recovery mechanisms. To solve such a problem, the rollback-recovery protocol rVsSG, preserving session guarantees has been proposed and its correctness in terms of safety has been formally proved.

The proposed protocol takes advantage of the known rollback-recovery techniques like logging and checkpointing, however, while applying these techniques, the semantics of operations is taken into account. Consequently, in rVsSG protocol, only the operations essential to provide required session guarantees are logged. Moreover, in the paper, we determine, how to take checkpoints in the most desirable moments for each session guarantee.

Our future work encompasses the development of rollback-recovery protocols, which are integrated with other consistency protocols. Moreover, appropriate simulation experiments to quantitatively evaluate overhead of rVsSG protocol are being carried out.

References

[BKKS05] J. Brzeziński, A. Kobusińska, J. Kobusiński, and M. Szychowiak. rvswfr recovery protocol for mobile systems. Technical Report RA-017/05, Institute of Computing Science, Poznań University of Technology, November 2005.

[BSW05] J. Brzeziński, C. Sobaniec, and D. Wawrzyniak. Safety of a server-based version vector protocol implementing session guarantees. In *Proc. of Int. Conf. on Computational Science (ICCS2005), LNCS 3516*, pages 423–430, Atlanta, USA, May 2005.

[CFV00] G. Chockler, R. Friedman, and R. Vitenberg. Consistency conditions for a CORBA caching service. In *Proc. of the 14th Int. Conf. on Distributed Computing (DISC'2000), LNCS 1914*, pages 374–388, October 2000.

[EEL+02] N. Elmootazbellah, Elnozahy, A. Lorenzo, Yi-Min Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[GR04] Rachid Guerraoui and Luis Rodrigues. *Introduction to distributed algorithms*. Springer-Verlag, 2004.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.

[PBS05] F. Picconi, J-M. Busca, and P. Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. *EuroPar 2005*, pages 1173–1182, 2005.

[SDS99] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, May 1999.

[Sob05] C. Sobaniec. *Consistency Protocols of Session Guarantees in Distributed Mobile Systems*. PhD thesis, Institute of Computing Science, Poznan University of Technology, September 2005.

[TDP+94] 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.