Concurrent Computing with Shared Replicated Memory

Egon Börger[1], Andreas Prinz[2], Klaus-Dieter Schewe[3]

[1] Università di Pisa, Dipartimento di Informatica, Pisa, Italy, boerger@di.unipi.it
[2] University of Agder, Department of ICT, Agder, Norway, andreas.prinz@uia.no
[3] Zhejiang University, UIUC Institute, China, kdschewe@acm.org

Abstract. Any concurrent system can be captured by a concurrent Abstract State Machine (cASM). This remains valid, if different agents can only interact via messages. It even permits a strict separation between memory managing agents and other agents that can only access the shared memory by sending query and update requests. This paper is dedicated to an investigation of replicated data that is maintained by a memory management subsystem, where the replication neither appears in the requests nor in the corresponding answers. We specify the behaviour of a concurrent system with such memory management using concurrent communicating ASMs (ccASMs), provide several refinements addressing different replication policies and internal messaging between data centres, and analyse their effects on the runs with respect to consistency. We show that on a concrete level even a weak form of consistency is only possible, if view serialisability can be guaranteed.

1 Introduction

Abstract State Machines (ASMs) have been used successfully to model sequential, parallel and concurrent systems [7]. They are mathematically grounded in behavioural theories of sequential algorithms [10], parallel algorithms [3, 8] and concurrent algorithms [4], by means of which it is proven that they capture the corresponding, axiomatically defined class of algorithms.

In particular, the behavioural theory of asynchronous concurrent systems shows that every concurrent system can be step-by-step simulated by a concurrent Abstract State Machine (cASM) [4]. The theory provides an important breakthrough in the theory of concurrent and distributed system, as it avoids mimicking concurrent behaviour by means of sequentiality using techniques such as interleaving and choice by an unspecified super-agent (see the semantics defined in [12]). This modernised theory of concurrency is so far only realised in cASMs. The proof of the concurrent ASM thesis was first only conducted for families of sequential algorithms,

but the generalisation to families of parallel algorithms does not cause serious difficulties as sketched in [16].

The theory can be applied to many different models of concurrent computation (see e.g. [1, 2, 9, 13, 18, 19]). It also remains valid, if different agents can only interact via messages [5]. This includes the case of a strict separation between memory managing agents and other agents that can only access the shared memory by sending query and update requests. Naturally, the expectation of an agent a sending a request to a memory management agent b is that the returned result is the same as if the agent a executed its request directly on the shared locations. This is challenged, if the memory management subsystem replicates the data as e.g. in the noSQL database system Cassandra [15].

In this paper we investigate the case of replicated data maintained by a memory management subsystem, where the replication should not appear in the requests nor in the corresponding answers, which is a standard requirement in distributed databases with replication [14, Chap.13]. Our objective is that the behaviour of the memory management subsystem must be understandable from the specification so that additional consistency assurance measures can be added if necessary.

For instance, consider four agents $a_1, \ldots, a_4$ having the following respective programs (where ; denotes sequential execution, and $Print(x)$ means to read the value x and to copy it to some output):

$$x := 1 \mid y := 1 \mid Print(x); Print(y) \mid Print(y); Print(x)$$

Then there is no concurrent run where (1) initially $x = y = 0$, (2) each agent makes once each possible move, (3) a_3 prints $x = 1, y = 0$, and (4) a_4 prints $x = 0, y = 1$. However, if x and y are replicated, say that there are always two copies, and an update by the programs a_1 or a_2 affects only a single copy, such an undesirable behaviour will indeed be enabled.

We assume a rather simple model, where shared data is logically organised in relations with primary keys, and data can only be accessed by means of the primary key values. We further assume relations to be horizontally fragmented according to values of a hash function on the primary key values, and these fragments are replicated. Replicas are assigned to different nodes, and several nodes together form a data centre, i.e. they are handled by one dedicated data management agent. In addition, values in replicas carry logical timestamps set by the data centres and stored in the records in the nodes of the memory management subsystem. This allows us to formulate and investigate policies that guarantee consistency.

For retrieval of a set of records a specified number of replicas has to be read, and for each record always the one with the latest timestamp will be returned. Depending on how many replicas are accessed the returned records may be (in the strict sense) outdated or not. Likewise, for the update of a set of records timestamps will be created, and a specified number of replicas of the records will be stored. Success of retrieval or update will be returned according to specified read- and write-policies.

In Section 2 we will first specify the behaviour of a concurrent system with shared data requiring that all agents interact with this subsystem for data retrieval and updates using appropriate `SEND` and `RECEIVE` actions. The memory management subsystem is specified by a separate collection of agents. In Section 3 we investigate a refinement concerning policies how many replicas are to be read or updated, respectively. We show that some combinations of replication policies enable *view compatibility*, which formalises the expectation above. In Section 4 we refine our specification taking the communication between data centres into account, and address the enforcement of the read and write policies. We obtain a complete, though not necessarily correct refinement, and as a consequence view compatibility cannot be guaranteed anymore. We even show that view compatibility implies view serialisability. Finally, we conclude with a brief summary and outlook.

This paper contains only a short version of our work on the subject, but a technical report with full details is available in [17]. In particular, proofs are only sketched here, but full-length proofs appear in this technical report.

2 Shared Memory Management with Replication

We assume some familiarity with Abstract State Machines (ASMs) (see [7, Sect.2.2/4]). The *signature* Σ of an ASM is a finite set of function symbols f, each associated with an arity ar_f. A *state* S is a set of functions f_S of arity $n = ar_f$ over some fixed base set B, given by interpretations of the corresponding function symbol f. Each pair $(f, (v_1, \ldots, v_n))$ comprising a function symbol and arguments $v_i \in B$ is called a *location*, and each pair (ℓ, v) of a location ℓ and a value $v \in B$ is called an *update*. A set of updates is called an *update set*. The evaluation of terms is defined as usual by $val_S(f(t_1, \ldots, t_n)) = f_S(val_S(t_1), \ldots, val_S(t_n))$. ASM *rules* r are composed using

assignments. $f(t_1, \ldots, t_n) := t_0$ (with terms t_i built over Σ),
branching. `IF` φ `THEN` r_+ `ELSE` r_-,

parallel composition. FORALL x WITH $\varphi(x)$ $r(x)$,
bounded parallel composition. $r_1 \dots r_n$,
choice. CHOOSE x WITH $\varphi(x)$ IN $r(x)$, and
let. LET $x = t$ IN $r(x)$.

Each rule yields an update set $\Delta(S)$ in state S. If this update set is consistent, i.e. it does not contain two updates $(\ell, v), (\ell, v')$ with the same location ℓ and different values $v \neq v'$, then applying this update set defines a successor state $S + \Delta(S)$.

2.1 Concurrent Communicating Abstract State Machines

A *concurrent ASM* (cASM) $\mathcal{CM}$ is defined as a family $\{(a, asm_a)\}_{a \in \mathcal{A}}$ of pairs consisting of an agent a and an ASM asm_a. Let Σ_a denote the signature of the ASM asm_a. Taking the union $\Sigma = \bigcup_{a \in \mathcal{A}} \Sigma_a$ we distinguish between $\mathcal{CM}$-states built over Σ and local states for agent a built over Σ_a; the latter ones are simply projections of the former ones on the subsignature.

A *concurrent run* of a concurrent ASM $\mathcal{CM} = \{(a, asm_a)\}_{a \in \mathcal{A}}$ is a sequence $S_0, S_1, S_2, \dots$ of $\mathcal{CM}$-states, such that for each $n \geq 0$ there is a finite set $A_n \subseteq \mathcal{A}$ of agents such that S_{n+1} results from simultaneously applying update sets $\Delta_a(S_{j(a)})$ for all agents $a \in A_n$ yielded by asm_a in some preceding state $S_{j(a)}$ ($j(a) \leq n$ depending on a), i.e. $S_{n+1} = S_n + \bigcup_{a \in A_n} \Delta_a(S_{j(a)})$ and $a \notin \bigcup_{i=j(a)}^{n-1} A_i$.

In order to isolate agents responsible for a memory management subsystem we exploit *communicating concurrent ASMs* (ccASM) [5]. In a ccASM the only shared function symbols take the form of mailboxes. Sending of a message m from a to b means to update the out-mailbox of a by inserting m into it. This mailbox is a set-valued shared location with the restriction that only the sender can insert messages into it and only the environment, i.e. the message processing system, can read and delete them. The message processing system will move the message m to the in-mailbox of the receiver b. Receiving a message m by b means in particular that b removes m from its in-mailbox and performs some local operation on m. Therefore, in ccASMs the language of ASM rules above is enriched by the following constructs (see [5] for further details):

Send. SEND(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩),
Receive. RECEIVE(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩),
Received. RECEIVED(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩), and
Consume. CONSUME(⟨message⟩, from:⟨sender⟩, to:⟨receiver⟩).

If all shared data is organised in relations with a unique primary key, this can be modelled by a set of function symbols $\Sigma_{\text{mem}} = \{p_1, \ldots, p_k\}$, where each p_i has a fixed arity a_i, and a fixed co-arity c_i, such that in each state S we obtain partial functions[4] $p_i^S : B^{a_i} \to B^{c_i}$.

A read access by an agent $a \in \mathcal{A}$ aims at receiving a subset of relation p_i containing those records with key values satisfying a condition φ, i.e. evaluate a term $p_i[\varphi] = \{(\boldsymbol{k}, \boldsymbol{v}) \mid \varphi(\boldsymbol{k}) \wedge \boldsymbol{v} \neq \mathit{undef} \wedge p_i(\boldsymbol{k}) = \boldsymbol{v}\}$. As p_i is not in the signature Σ_a, the agent a must send a read-request and wait for a response, i.e. it executes `SEND`(read(p_i, φ),from:a,to:$\mathit{home}(a)$) and waits until `RECEIVED`(answer($\mathit{ans}, p_i, \varphi$),from:$\mathit{home}(a)$,to:$a$) becomes true. Then it can execute `RECEIVE`(answer($\mathit{ans}, p_i, \varphi$),from:$\mathit{home}(a)$,to:$a$) to obtain the requested value.

We abstract from the details of the communication but assume the communication to be reliable. If there is no confusion, we omit the sender and receiver parameters in `SEND` and `RECEIVE`. The ans in the message must be a relation of arity $a_i + c_i$ satisfying the key property above. The agent can store such an answer using a non-shared function p_i^a or process it in any other way, e.g. aggregate the received values. This is part of the ASM rule in asm_a, which we do not consider any further.

In the `SEND` and `RECEIVE` rules we use a fixed agent $\mathit{home}(a)$ with which the agent a communicates. It will be unknown to the agent a, whether this agent $\mathit{home}(a)$ processes the read-request or whether it communicates with other agents to produce the answer.

Analogously, for bulk write access an agent a may want to execute the operation $p_i :\& p$ to update all records with a key defined in p to the new values given by p. As this corresponds to an ASM rule `FORALL` $(\boldsymbol{k}, \boldsymbol{v}) \in p$ $p_i(\boldsymbol{k}) := \boldsymbol{v}$, the agent a must send a write-request and wait for a response, i.e. it executes `SEND`(write(p_i, p),to:$\mathit{home}(a)$) and waits to receive an acknowledgement, i.e. to `RECEIVE`(acknowledge(p_i, p), from:$\mathit{home}(a)$).

We use the notation $\mathcal{CM}_0 = \{(a, \mathit{asm}_a^c)\}_{a \in \mathcal{A}} \cup \{(db, \mathit{asm}_{db})\}$ for the ccASM with a single memory agent db and $\mathit{home}(a) = db$ for all $a \in \mathcal{A}$. Thus, the rule of asm_{db} looks as follows:

```
IF RECEIVED(read(p_i, φ),from:a) THEN
   CONSUME(read(p_i, φ),from:a)
   LET ans = {(k, v) | φ(k) ∧ p_i(k) = v ∧ v ≠ undef} IN
       SEND(answer(ans, p_i, φ),to:a)
IF RECEIVED(write(p_i, p),from:a) THEN
```

[4] In ASMs partial functions are captured by total functions using a dedicated value undef.

`CONSUME`(write(p_i, p),from:a)
`FORALL` $(\boldsymbol{k}, \boldsymbol{v}) \in p$ $p_i(\boldsymbol{k}) := \boldsymbol{v}$
`SEND`(acknowledge(p_i, p),to:a)

2.2 Memory Organisation with Replication

For replication we use several *data centres*, and each data centre comprises several *nodes*. The nodes are used for data storage, and data centres correspond to physical machines maintaining several such storage locations. Let $\mathcal{D}$ denote the set of data centres. Then instead of a location $(p_i, \boldsymbol{k})$ there will always be several replicas, and at each replica we may have a different value.

Let us assume that each relation p_i is fragmented according to the values of a hash-key. That is, for each $i = 1, \ldots, k$ we can assume a static hash-function $h_i : B^{a_i} \to [m, M] \subseteq \mathbb{Z}$ assigning a hash-key to each key value. We further assume a partition $[m, M] = \bigcup_{j=1}^{q_i} \mathit{range}_j$ such that $\mathit{range}_{j_1} < \mathit{range}_{j_2}$ holds for all $j_1 < j_2$, so each range will again be an interval. These range intervals are used for the horizontal fragmentation into q_i fragments of the to-be-represented function p_i: $\mathit{Frag}_{j,i} = \{\boldsymbol{k} \in B^{a_i} \mid h_i(\boldsymbol{k}) \in \mathit{range}_j\}$.

All these fragments will be replicated and their elements associated with a value (where defined by the memory management system), using a fixed *replication factor* r_i. That is, each fragment $\mathit{Frag}_{j,i}$ will be replicated r_i-times for each data centre. A set of all pairs $(\boldsymbol{k}, \boldsymbol{v})$ with key $\boldsymbol{k} \in \mathit{Frag}_{j,i}$ and an associated value $\boldsymbol{v}$ in the memory management system is called a replica of $\mathit{Frag}_{j,i}$.

Assume that each data centre d consists of n_i nodes, identified by d and a number $j' \in \{1, \ldots, n_i\}$. Then we use a predicate $\mathit{copy}(i, j, d, j')$ to denote that the node with number j' in the data centre $d \in \mathcal{D}$ contains a replica of $\mathit{Frag}_{j,i}$. We also use $\mathcal{D}_i = \{d \in \mathcal{D} | \exists j, j'.\mathit{copy}(i, j, d, j')\}$. To denote the values in replicas we use dynamic functions $p_{i,j,d,j'}$ of arity a_i and co-arity $c_i + 1$ (functions we call again replicas). So we use function symbols $p_{i,j,d,j'}$ with $j \in \{1, \ldots, q_i\}$, $d \in \mathcal{D}$ and $j' \in \{1, \ldots, n_i\}$, and we request $h_i(\boldsymbol{k}) \in \mathit{range}_j$ for all $\boldsymbol{k} \in B^{a_i}$, whenever $\mathit{copy}(i, j, d, j')$ holds and $p_{i,j,d,j'}(\boldsymbol{k})$ is defined. For the associated values we have $p_{i,j,d,j'}(\boldsymbol{k}) = (\boldsymbol{v}, t)$, where t is an added timestamp value, and values $\boldsymbol{v}$ may differ from replica to replica.

Each data centre d maintains a logical clock clock_d that is assumed to advance (without this being further specified), and clock_d evaluates to the current time at data centre d. Timestamps must be totally ordered, differ if set by different data centres, and respect the inherent order of message

passing, i.e. when data with a timestamp t is created at data centre d and sent to data centre d', then at the time the message is received the clock at d' must show a time larger than t. This condition can be enforced by adjusting clocks according to Lamport's algorithm [11]. For this let us define adjust_clock$(d, t) = \mathit{clock}_d := t'$, where t' is the smallest possible timestamp at data centre d with $t \leq t'$.

2.3 Internal Request Handling for Replicated Memory

When dealing with replication the request messages sent by agents a remain the same, but the internal request handling by the memory management subsystem changes. This will define a refined ccASM $\mathcal{CM}_1 = \{(a, \mathit{asm}_a^c)\}_{a\in\mathcal{A}} \cup \{(d, \mathit{asm}_d)\}_{d\in\mathcal{D}}$. We will use the notions of *complete* and *correct* refinement as defined in [7, pp.111ff.].

Let M and M^* be cASMs (or ccASMs). We fix a correspondence relation $\sim$ between some states of M and some states of M^*. Then M^* is called a *correct refinement* of M iff for each run $S_0^*, S_1^*, \ldots$ of M^* there is a run $S_0, S_1, \ldots$ of M together with sequences $0 = i_0 < i_1 < \ldots$ and $0 = j_0 < j_1 < \ldots$ such that $S_{i_k} \sim S_{j_k}^*$ holds for all k, and if both runs are finite with final states S_f^* and S_f, respectively, then there exists an index ℓ with $S_{i_\ell} = S_f$ and $S_{i_\ell}^* = S_f^*$. We call M^* a *complete refinement* of M iff M is a correct refinement of M^*.

Consider a read request read(p_i, φ) received from agent a by data centre d. As data is horizontally fragmented, we need to evaluate several requests read$(p_{i,j}, \varphi)$ concerning keys $\boldsymbol{k}$ with $h_i(\boldsymbol{k}) \in \mathit{range}_j$, one request for each fragment index j, and then build the union so that $p_i[\varphi] = \bigcup_{j=1}^{q_i} p_{i,j}[\varphi]$. In order to evaluate $p_{i,j}[\varphi]$ several replicas of $\mathit{Frag}_{j,i}$ will have to be accessed. Here we will leave out any details on how these replicas will be selected and accessed, but the selection of replicas must comply with a *read-policy* that is left abstract for the moment.

When reading actual data, i.e. evaluating $p_{i,j,d,j'}(\boldsymbol{k})$ for selected key values $\boldsymbol{k}$, we obtain different time-stamped values $(\boldsymbol{v}, t)$, out of which a value $\boldsymbol{v}$ with the latest timestamp is selected and sent to a as the up-to-date value of $p_i(\boldsymbol{k})$. The requirement that timestamps set by different data centres differ implies that for given $\boldsymbol{k}$ the value $\boldsymbol{v}$ with the latest timestamp is unique. All records obtained this way will be returned as the result of the read request to the issuing agent a. Thus, we obtain the following ASM rule `AnswerReadReq`:

`AnswerReadReq` =
`IF RECEIVED`(read(p_i, φ), from:a) `THEN`

CONSUME(read(p_i, φ), from:a)
FORALL $j \in \{1, \ldots, q_i\}$ CHOOSE $G_{i,j}$ WITH *complies*($G_{i,j}$, read-policy)
$\quad \wedge\, G_{i,j} \subseteq \{(d', j') \mid copy(i, j, d', j') \wedge d' \in \mathcal{D}_i \wedge 1 \leq j' \leq n_i\}$
LET $t_{\max}(\boldsymbol{k}) = \max\{t \mid \exists \boldsymbol{v}', \bar{d}, \bar{j}.(\bar{d}, \bar{j}) \in G_{i,j} \wedge p_{i,j,\bar{d},\bar{j}}(\boldsymbol{k}) = (\boldsymbol{v}', t)\}$ IN
LET $ans_{i,j} = \{(\boldsymbol{k}, \boldsymbol{v}) \mid \varphi(\boldsymbol{k}) \wedge h_i(\boldsymbol{k}) \in range_j \wedge \boldsymbol{v} \neq undef \wedge$
$\qquad \exists d', j'.((d', j') \in G_{i,j} \wedge p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}, t_{\max}(\boldsymbol{k})))\}$ IN
LET $ans = \bigcup_{j=1}^{q_i} ans_{i,j}$ IN SEND(answer(ans, p_i, φ), to:a)

Note that the unique value $\boldsymbol{v}$ with $p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}, t_{\max}(\boldsymbol{k}))$ may be *undef* and that the returned ans may be the empty set.

For a write request write(p_i, p) sent by agent a to data centre d we proceed analogously. In all replicas of $Frag_{j,i}$ selected by a *write-policy* the records with a key value in p will be updated to the new value provided by p—this may be *undef* to capture deletion—and a timestamp given by the current time $clock_d$. However, the update will not be executed, if the timestamp of the existing record is already newer. In addition, clocks that "are too late" will be adjusted, i.e. if the new timestamp received from the managing data centre d is larger than the timestamp at data centre d', the clock at d' is set to the received timestamp. Thus, we obtain the following ASM rule PerformWriteReq to-be-executed by any data centre d upon receipt of an update request from an agent a:

PerformWriteReq =
IF RECEIVED(write(p_i, p), from:a) THEN
  CONSUME(write(p_i, p), from:a)
  FORALL $j \in \{1, \ldots, q_i\}$ CHOOSE $G_{i,j}$ WITH *complies*($G_{i,j}$, write-policy)
    $\wedge\, G_{i,j} \subseteq \{(d', j') \mid copy(i, j, d', j') \wedge d' \in \mathcal{D}_i \wedge 1 \leq j' \leq n_i\}$
  LET $t_{current} = clock_{\texttt{self}}$ IN
    FORALL $(d', j') \in G_{i,j}$
      FORALL $(\boldsymbol{k}, \boldsymbol{v}) \in p$ WITH $h_i(\boldsymbol{k}) \in range_j$
        IF $\exists \boldsymbol{v}', t.p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}', t) \wedge t < t_{current}$ THEN
          $p_{i,j,d',j'}(\boldsymbol{k}) := (\boldsymbol{v}, t_{current})$
      IF $clock_{d'} < t_{current}$ THEN adjust_clock($d', t_{current}$)
  SEND(acknowledge(p_i, p), to:a)

Proposition 1. *The ccASM* $\mathcal{CM}_1 = \{(a, asm_a^c)\}_{a \in \mathcal{A}} \cup \{(d, asm_d)\}_{d \in \mathcal{D}}$ *is a complete refinement of* $\mathcal{CM}_0 = \{(a, asm_a^c)\}_{a \in \mathcal{A}} \cup \{(db, asm_{db})\}$.

Proof (sketch, see also [17]). The only differences between the ccASMs $\mathcal{CM}_0$ and $\mathcal{CM}_1$ are that $home(a) \in \mathcal{D}$ differs in the refinement, but in both cases the handling of a request is done in a single step. For both cases the determination of the answer requires a more sophisticated rule in the refinement. □

3 Refinement Using Replication Policies

We define *view compatibility* formalising the intuitive expectation of the agents that answers to sent requests remain the same in case of replication as without, because replication is completely transparent. We show that for particular combinations of concrete read- and write-policies $\mathcal{CM}_1$ guarantees view compatibility, which further implies that the refinement of $\mathcal{CM}_0$ by $\mathcal{CM}_1$ is correct.

3.1 View Compatibility

Informally, view compatibility is to ensure that the system behaves in a way that whenever an agent sends a read- or write-request the result is the same as if the read or write had been executed in a state without replication. For a formal definition we need the notion of an *agent view* of a concurrent run $S_0, S_1, \ldots$ of the cASM $\{(a, asm_a^c)\}_{a\in\mathcal{A}}$ for an arbitrary agent $a \in \mathcal{A}$. Its view of the run is the subsequence of states $S_{a,0}, S_{a,1}, \ldots$ in which a makes a move (restricted to the signature of a). For any state $S_k = S_{a,n}$ its successor state in the a-view sequence depends on the move a performs in $S_{a,n}$.

If a in $S_{a,n}$ performs a send step, it contributes to the next state S_{k+1} by an update set which includes an update of its out-mailbox, which in turn yields an update of the mailbox of $home(a)$. But S_{k+1} is not yet the next a-view state, in which a will perform its next move. This move is determined by the *atomic request/reply assumption* for agent/db runs: If in a run an agent performs a send step, then its next step in the run is the corresponding receive step, which can be performed once the answer to the sent request has been received. By this assumption the next a-view state $S_{a,n+1} = S_l$ is determined by (what appears to a as) an environment action enabling the receive step by inserting the reply message into a's mailbox.

If a in $S_{a,n} = S_k$ performs a receive or an internal step, then besides the mailbox update to consume the received message it yields only updates to non-shared locations so that its next a-view state is the result of applying these updates together with updates of other agents to form $S_{k+1} = S_{a,n+1}$.

We further need the notion of a *flattening* which reduces the multiple values associated with replicas of a location ℓ to a single value: If $S_0, S_1, S_2, \ldots$ is a run of $\mathcal{CM}_1 = \{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(d, asm_d)\}_{d\in\mathcal{D}}$, then we obtain a *flattening* $S'_0, S'_1, S'_2, \ldots$ by replacing in each state all locations

$(p_{i,j,d',j'}, \boldsymbol{k})$ by a single location $(p_i, \boldsymbol{k})$ and letting the value associated with $(p_i, \boldsymbol{k})$ be one of the values in $\{\boldsymbol{v} \mid \exists j, d', j'.\exists t.p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}, t)\}$.

Obviously, a flattening is a sequence of states of the concurrent ASM $\{(a, asm_a^c)\}_{a\in\mathcal{A}}$, but in most cases it will not be a run. Therefore, take an arbitrary subsequence $S'_{j_0}, S'_{j_1}, \ldots$ of an arbitrary flattening $S'_0, S'_1, \ldots$ (restricted to the signature of the agent a) of $S_0, S_1, \ldots$. Then $S'_{j_0}, S'_{j_1}, \ldots$ is called a *flat view* of agent a of the run $S_0, S_1, \ldots$ if the following conditions hold: Whenever a performs a request in state S_k there is some S'_{j_i} such that $k = j_i$. If the corresponding reply is received in state S_m for some $m > k$, then $S'_{j_{i+1}} = S_m$. Furthermore, there exists some n with $k < n \leq m$ such that if the request is a write-request, then for each location ℓ with value v in this request $val_{S'_n}(\ell) = v$ holds, provided there exists an agent reading the value v, and if the request is a read-request, then for each location ℓ with value v in the answer $val_{S'_n}(\ell) = v$ holds. Whenever a performs a RECEIVE or an internal move in state S_k there is some j_i such that $S_k = S'_{j_i}$ and $S_{k+1} = S'_{j_{i+1}}$.

We say that $\{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(d, asm_d)\}_{d\in\mathcal{D}}$ is *view compatible* with the concurrent ASM $\{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(db, asm_{db})\}$ iff for each run $\mathcal{R} = S_0, S_1, S_2, \ldots$ of $\{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(d, asm_d)\}_{d\in\mathcal{D}}$ there exists a subsequence of a flattening $\mathcal{R}' = S'_0, S'_1, S'_2, \ldots$ that is a run of $\{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(db, asm_{db})\}$ such that for each agent $a \in \mathcal{A}$ the agent a-view of $\mathcal{R}'$ coincides with a flat view of $\mathcal{R}$ by a.

3.2 Specification of Replication Policies

In the specification of ASM rules handling read and write requests by a fixed data centre d we used sets $G_{i,j} \subseteq C_{i,j}$ with $C_{i,j} = \{(d', j') \mid copy(i, j, d', j')\}$ as well as an abstract predicate $complies(G_{i,j}, \text{policy})$. Let us now define the most important policies ALL, ONE, and QUORUM—more policies are handled in [17]. These policies differ in the number of replicas that are to be accessed. As the name indicates, the predicate $complies(G_{i,j}, \text{ALL})$ can be defined by $G_{i,j} = C_{i,j}$, i.e. all replicas are to be accessed. For ONE at least one replica is to be accessed, which defines $complies(G_{i,j}, \text{ONE})$ by $|G_{i,j}| \geq 1$. For QUORUM(q) we use a value q with $0 < q < 1$, and $complies(G_{i,j}, \text{QUORUM}(q))$ is defined by $q \cdot |C_{i,j}| < |G_{i,j}|$.

For consistency analysis we need *appropriate combinations* of read- and write-policies. If the write-policy is QUORUM(q) and the read-policy is QUORUM(q') with $q + q' \geq 1$, then the combination is appropriate. Furthermore, a combination of the write policy (or read-policy) ALL with any other policy is also appropriate.

Proposition 2. *If the combination of the read and write policies is appropriate, then $\mathcal{CM}_1$ is view compatible with the cASM $\{(a, asm_a^c)\}_{a\in\mathcal{A}}$ and a correct refinement of $\mathcal{CM}_0$.*

Proof (sketch, a full proof is available in [17]). If the write-policy is QUORUM(q), then for each location ℓ the multiplicity of replicas considered to determine the value with the largest timestamp is at least $\lceil\frac{m+1}{2}\rceil$ with m being the total number of replicas. Consequently, each read access with a policy QUORUM(q') (with $q + q' \geq 1$) reads at least once this value and returns it. That is, in every state only the value with the largest timestamp for each location uniquely determines the run, which defines the equivalent concurrent run. □

4 Refinement with Internal Communication

We will now address a refinement of the memory management subsystem taking into account that data centres refer to different physical machines, whereas in $\mathcal{CM}_1$ we abstracted from any internal communication. The gist of the refinement is therefore to treat the handling of a request as a combination of direct access to local nodes, remote access via messages to the other relevant data centres, and collecting and processing return messages until the requirements for the read- or write-policies are fulfilled. That is, the validation of the policy accompanies the preparation of a response message and is no longer under control of the *home* agent.

4.1 Request Handling with Communicating Data Centres

In Section 2 we specified how a data centre agent d handles a request received from an agent a. Now, we first specify an abstract rule which manages external requests, i.e. coming from an agent a and received by a data centre d, where *request* is one of these read or write requests. An external request is forwarded as internal request to all other data centres $d' \in \mathcal{D}_i$, where it is handled and answered locally (see the definition of `HandleLocally` below), whereas collecting (in $answer_{a'}$) and sending the overall answer to the external agent a is delegated to a new agent a'. `SELF` denotes the data centre agent d which executes the rule.

To `Initialize` a delegate a' it is equipped with a set $answer_{a'}$, where to collect the values arriving from the asked data centres and with counters $count_{a'}(j)$ (for the number of inspected replicas of the j-th fragment). The counters are used for checking compliance with the policies.

The *mediator* and *requestor* information serves to retrieve sender and receiver once the delegate completes the answer to the request.

```
DelegateExternalReq =
 IF RECEIVED(request, from:a) THEN
  CONSUME(request, from:a)
  LET t_current = clock_SELF, a' = new(Agent) IN
      Initialize(a')
      HandleLocally(request, a', t_current)
      ForwardToOthers(request, a', t_current)
WHERE
ForwardToOthers(request, a', t_current)=
      FORALL d' ∈ D_i WITH d' ≠ SELF SEND((request, a', t_current), to:d')
Initialize(a') =
      answer_a' := ∅
      FORALL 1 ≤ j ≤ q_i  count_a'(j) := 0
      IF request = read(p_i, φ)
      THEN asm_a' := CollectRespondToRead
      ELSE asm_a' := CollectRespondToWrite
      requestor_a' := a
      mediator_a' := SELF
      requestType_a' := request
```

In this way the request handling agent d simply forwards the request to all other data centre agents and in parallel handles the request locally for all nodes associated with d. The newly created agent (a 'delegate') will take care of collecting all response messages and preparing the response to the issuing agent a. Request handling by any other data centre d' is simply done locally using the following rule:

```
ManageInternalReq =
  IF RECEIVED((request, a', t), from:d) THEN
      HandleLocally(request, a', t)
      CONSUME((request, a', t), from:d)
```

Each data centre agent d is equipped with the following ASM rule, where the components `HandleLocally` and the two versions of `CollectRespond` are defined below.

$$asm_d = \texttt{DelegateExternalReq ManageInternalReq}$$

For local request handling policy checking is not performed by the data centre agent but by the delegate of the request; check the predicates

all_messages_received and *sufficient*(policy) below. We use a predicate *alive* to check, whether a node is accessible or not. For a read request we specify `HandleLocally`$(\text{read}(p_i, \varphi), a', t_{\text{current}})$ as follows:

`HandleLocally`$(\text{read}(p_i, \varphi), a', t)=$
  `LET` $d' =$ `SELF IN`
  `LET` $G_{i,j,d'} = \{j' \mid copy(i, j, d', j') \wedge alive(d', j')\}$ `IN`
  `LET` $t_{\max}(\boldsymbol{k}) = max(\{t \mid \exists \boldsymbol{v}', \bar{j}.\bar{j} \in G_{i,j,d'} \wedge p_{i,j,d',\bar{j}}(\boldsymbol{k}) = (\boldsymbol{v}', t)\})$ `IN`
  `LET` $ans_{i,j,d'} = \{(\boldsymbol{k}, \boldsymbol{v}, t_{\max}(\boldsymbol{k})) \mid \varphi(\boldsymbol{k}) \wedge h_i(\boldsymbol{k}) \in range_j \wedge$
    $\exists j' \in G_{i,j,d'}.p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}, t_{\max}(\boldsymbol{k}))\}$ `IN`
  `LET` $ans = \bigcup_{j=1}^{q_i} ans_{i,j,d'}$, $\boldsymbol{x} = (|G_{i,1,d'}|, \ldots, |G_{i,q_i,d'}|)$ `IN`
    `SEND`(answer$(ans, \boldsymbol{x})$, to:a')

Here we evaluate the request locally, but as the determined maximal timestamp may not be globally maximal, it is part of the returned relation. Also the number of replicas that contributed to the local result is returned, such that the delegate a' responsible for collection and final evaluation of the request can check the satisfaction of the read-policy. Therefore, the created partial result is not returned to the agent d that issued this local request, but instead to the delegate.

Rule `HandleLocally`$(\text{write}(p_i, p), a', t_{\text{current}})$ handles write requests:

`HandleLocally`$(\text{write}(p_i, p), a', t')=$
  `LET` $d' =$`SELF IN`
  `IF` $clock_{d'} < t'$ `THEN` adjust_clock(d', t')
  `LET` $G_{i,d'}(j) = \{j' \mid copy(i, j, d', j') \wedge alive(d', j')\}$ `IN`
  `FORALL` $j \in \{1, \ldots, q_i\}$ `FORALL` $j' \in G_{i,d'}(j)$
    `FORALL` $(\boldsymbol{k}, \boldsymbol{v}) \in p$ `WITH` $h_i(\boldsymbol{k}) \in range_j$
      `IF` $\exists \boldsymbol{v}', t.p_{i,j,d',j'}(\boldsymbol{k}) = (\boldsymbol{v}', t) \wedge t < t'$ `THEN` $p_{i,j,d',j'}(\boldsymbol{k}) := (\boldsymbol{v}, t')$
  `LET` $\boldsymbol{x} = (|G_{i,1,d'}|, \ldots, |G_{i,q_i,d'}|)$ `IN` `SEND`(ack_write$(p_i, p, \boldsymbol{x})$, to:a')

Again, the partial results acknowledging the updates at the nodes associated with data centre d' are sent to the collecting agent a' to verify the compliance with the write-policy. For the delegate a' that has been created by d to collect partial responses and to create the final response to the agent a issuing the request we need predicates *sufficient*(policy) for policy checking, in which case the response to a is prepared and sent:

$$\begin{aligned} \textit{sufficient}(\textsc{All}) &\equiv \forall j.(1 \le j \le q_i \Rightarrow count(j) = \gamma_{i,j}) \\ &\qquad \text{with } \gamma_{i,j} = |\{(d', j') \mid copy(i, j, d', j')\}| \\ \textit{sufficient}(\textsc{One}) &\equiv \forall j.(1 \le j \le q_i \Rightarrow count(j) \ge 1) \end{aligned}$$

$$\mathit{sufficient}(\textsc{Quorum}(q)) \;\equiv\; \forall j.(1 \leq j \leq q_i \Rightarrow \gamma_{i,j} \cdot q < \mathit{count}(j))$$

It remains to specify the delegate rules `CollectRespondToRead` and `CollectRespondToWrite`, i.e. the programs associated with the agent a' created upon receiving a *request* from an agent a. The `CollectRespond` action is performed until all required messages have been received and splits into two rules for read and write requests, respectively.

The delegate a' collects the messages it receives from the data centres d' to which the original *request* had been forwarded to let them `HandleLocally`$(\mathit{request}, a')$. If the set of collected answers suffices to respond, the delegate sends an answer to the original requester and kills itself.

Thus each of the rules `CollectRespondToRead` and `CollectRespondToWrite` has a subrule `Collect` and a subrule `Respond` with corresponding parameter for the type of expected messages.

```
CollectRespondToRead=
  IF RECEIVED(answer(ans, x), from:d') THEN
      CONSUME(answer(ans, x), from:d')
      Collect((ans, x), from:d')
  IF sufficient(read-policy) THEN
      Respond(requestType_SELF)
WHERE
Respond(read(p_i, φ))=
  LET d=mediator(SELF),  a=requestor(SELF) IN
  LET ans = {(k, v) | ∃t. (k, v, t) ∈ answer_SELF} IN
      SEND(answer(ans, p_i, φ), from:d, to:a)
  DELETE(SELF,Agent)
Collect((ans, x), from : d')=
  FORALL k WITH ∃v, t. (k, v, t) ∈ ans
      LET (k, v, t) ∈ ans IN
            IF ∃v', t'. (k, v', t') ∈ answer_SELF THEN
            LET (k, v', t') ∈ answer_SELF IN
                  IF t' < t THEN
                        DELETE((k, v', t'), answer_SELF
                        INSERT((k, v, t), answer_SELF
            ELSE  INSERT((k, v, t), answer_SELF)
  LET (x_1, ..., x_{q_i}) = x IN
      FORALL j ∈ {1, ..., q_i}
            count(j) := count(j) + x_j
```

The analogous collection of messages for write requests is simpler, as the final response is only an acknowledgement.

```
CollectRespondToWrite=
  IF RECEIVED(ack_write(p_i, p, x), from:d') THEN
      Collect(ack_write(p_i, x), from:d')
      CONSUME(ack_write(p_i, p, x), from:d')
  IF  sufficient(write-policy) THEN
      SEND(acknowledge(p_i, p), from:mediator(SELF), to:requestor(SELF))
      DELETE(SELF,Agent)
WHERE
Collect(ack_write(p_i, x), from:d') =
      LET (x_1, ..., x_{q_i}) = x IN
            FORALL j ∈ {1, ..., q_i}
                  count(j) := count(j) + x_j
```

Note that our specification does not yet deal with exception handling. We may tacitly assume that eventually all requested answers will be received by the collecting agent.

4.2 Analysis of the Refinement

Let $\mathcal{CM}_2$ denote the refined ccASM $\{(a, asm_a^c)\}_{a\in\mathcal{A}} \cup \{(d, asm'_d)\}_{d\in\mathcal{D}\cup\mathcal{E}xt}$ together with the dynamic set $\mathcal{E}xt$ of delegates. Note that the delegates are created on-the-fly by the agents $d \in \mathcal{D}$, needed for collecting partial responses for each request and preparing the final responses (a full proof is given in [17]).

Proposition 3. *$\mathcal{CM}_2$ is a complete refinement of $\mathcal{CM}_1$.*

Proof (sketch). Take a concurrent run of $\mathcal{CM}_1$ and first look at it from the perspective of a single agent $a \in \mathcal{A}$. Updates brought into the state S by a are read and write requests. If a continues in some state $S' = S_{i+x}$, then the transition from S to S' is achieved by means of a step of asm_d for $d = home(a)$. Therefore, there exist states $\bar{S}$, $\bar{S}'$ for $\mathcal{CM}_2$, in which the asm_a^c brings in the same read and write requests and receives the last response, respectively. In a concurrent run for $\mathcal{CM}_2$ the transition from $\bar{S}$ to $\bar{S}'$ results from several steps by the subsystem $\{(d, asm'_d)\}_{d\in\mathcal{D}\cup\mathcal{E}xt}$.

With respect to each of the requests received from a the agent $d = home(a)$ contributes to a state $\bar{S}_1$ with requests for each agent $d' \in \mathcal{D}$, $d' \neq d$, the creation and initialisation of a response collecting agent a', and the local handling of the request at nodes associated with data centre

d. Then each agent $d' \in D$ contributes to some state $\bar{S}_k$ ($k > 1$), in which the partial response to the request sent to d' is produced. Concurrently the collection agent a' on receipt of a partial response updates its own locations, and thus contributes to some state $\bar{S}_j$ ($j > 1$). Finally, a' will also produce and send the response to a. This response will be the same as the one in state S', if the refined run from $\bar{S}$ to $\bar{S}'$ uses the same selection of copies for each request referring to p_i and each fragment $Frag_{j,i}$.

This implies that $\mathcal{CM}_{2,a} = \{(a, asm_a^c)\} \cup \{(d, asm'_d)\}_{d \in \mathcal{D} \cup \mathcal{E}xt}$ is a complete refinement of $\mathcal{CM}_{1,a} = \{(a, asm_a^c)\} \cup \{(d, asm_d)\}_{d \in \mathcal{D}}$, from which the proposition follows immediately. □

Unfortunately, view compatibility cannot be preserved, unless additional conditions are enforced in the specification. Any such condition already implies serialisability (a full proof is given in [17]).

Proposition 4. *If $\mathcal{CM}_2$ is view compatible with the cASM $\mathcal{CM}_0$, then every run $\mathcal{R}$ of $\mathcal{CM}_2$ is view serialisable. If all runs of $\mathcal{CM}_2$ are view serialisable and an appropriate combination of a read and a write policy is used, then $\mathcal{CM}_2$ is also view compatible with the concurrent ASM $\mathcal{CM}_0$.*

Proof (sketch). For a run $\mathcal{R} = S_0, S_1, \ldots$ of $\mathcal{CM}_2$ view compatibility implies that there exists a subsequence of a flattening $\mathcal{R}' = S'_0, S'_1, \ldots$ that is a run of $\{(a, asm_a^c)\}_{a \in \mathcal{A}} \cup \{(db, asm_{db})\}$ such that for each agent $a \in \mathcal{A}$ the agent a-view of $\mathcal{R}'$ coincides with a flat view of $\mathcal{R}$ by a. Let Δ'_ℓ be the update set defined by $S'_\ell + \Delta'_\ell = S'_{\ell+1}$, and define $\Delta_\ell =$

$$\{((p_{i,j,d,j'}, \boldsymbol{k}), (\boldsymbol{v}, t_\ell)) \mid ((p_i, \boldsymbol{k}), \boldsymbol{v}) \in \Delta'_\ell \wedge copy(i, j, d, j') \wedge h_i(\boldsymbol{k}) \in range_j\}$$

using timestamps $t_0 < t_1 < t_2 < \ldots$. This defines a run $\bar{\mathcal{R}} = \bar{S}_0, \bar{S}_1, \ldots$ of $\mathcal{CM}_2$ with $\bar{S}_0 = S_0$ and $\bar{S}_{\ell+1} = \bar{S}_\ell + \Delta_\ell$, which implies that $\mathcal{R}'$ is serial.

Furthermore, the runs $\mathcal{R}$ and $\bar{\mathcal{R}}$ contain exactly the same requests and responses, for each agent a the sequence of its requests and responses is identical in both runs, and hence $\mathcal{R}$ and $\bar{\mathcal{R}}$ are view equivalent.

Conversely, for a run $\mathcal{R}'$ of $\mathcal{CM}_2$ and a view equivalent serial run $\mathcal{R}' = S_0, S_1, \ldots$ it suffices to show that that there exists a subsequence of a flattening $\mathcal{R}' = S'_0, S'_1, S'_2, \ldots$ that is a run of $\{(a, asm_a^c)\}_{a \in \mathcal{A}} \cup \{(db, asm_{db})\}$ such that for each agent $a \in \mathcal{A}$ the agent a-view of $\mathcal{R}'$ coincides with a flat view of $\mathcal{R}$ by a. For this we only have to consider states, in which a request or a response is issued to obtain the desired subsequence, and the flattening is defined by the answers to the write requests, which follows immediately from $\mathcal{R}$ being serial. □

5 Concluding Remarks

Concurrent ASMs (cASMs) have been introduced to show that truly concurrent algorithms can be captured by an extension of ASMs [4]. In particular, cASMs overcome limitations in the theory of concurrency associated with the presence of interleaving and unspecified selection agents that enable several agents to perform steps synchronously. This specification and refinement study in this paper shows that oncurrent ASMs are well suited to capture all requirements in distributed, concurrent systems.

We demonstrated the application of concurrent communicating ASMs (ccASMs) [5] for the specification, refinement and consistency analysis of concurrent systems in connection with shared replicated memory. We first specified a ground model, in which all access to replicas is handled synchronously in parallel by a single agent, then refined it addressing the internal communication in the memory management subsystem. This refinement significantly changed the way requests are handled, as replicas are not selected a priori in a way that complies with the read- or write-policies, but instead the acknowledgement and return of a response depends on these policies. These refinements could be taken further to capture also the means for handling inactive nodes and for recovery. Due to space limitations for this conference version some explanations remain terse and proofs are only sketched, but details are available in an extended technical report [17].

We further showed that consistency, formalised by the notion of view compatibility, cannot be preserved by the last refinement. We could show that even such a rather weak notion of consistency can only be obtained, if view serialisability is assured. Serialisability can be achieved by adopting transactions for at least single requests. For instance, one might integrate a transactional concurrent system [6] with the specification of a replicative storage system as done in this paper.

References

1. G. Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
2. E. Best. *Semantics of sequential and parallel programs.* Prentice Hall, 1996.
3. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Trans. Computational Logic*, 4(4):578–651, 2003.
4. E. Börger and K.-D. Schewe. Concurrent abstract state machines. *Acta Inf.*, 53(5):469–492, 2016.
5. E. Börger and K.-D. Schewe. Communication in Abstract State Machines. *J. Univ. Comp. Sci.*, 23(2):129–145, 2017.

6. E. Börger, K.-D. Schewe, and Q. Wang. Serialisable multi-level transaction control: A specification and verification. *Sci. Comput. Program.*, 131:42–58, 2016.
7. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
8. F. Ferrarotti, K.-D. Schewe, L. Tec, and Q. Wang. A new thesis concerning synchronised parallel computing – simplified parallel ASM thesis. *Theor. Comp. Sci.*, 649:25–53, 2016.
9. H. J. Genrich and K. Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
10. Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comp. Logic*, 1(1):77–111, 2000.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
12. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
13. A. Mazurkiewicz. Trace theory. volume 255 of *LNCS*, pages 279–324. Springer, 1987.
14. M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.
15. T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii. Solving big data challenges for enterprise application performance management. *PVLDB*, 5(12):1724–1735, 2012.
16. K.-D. Schewe, F. Ferrarotti, L. Tec, Q. Wang, and W. An. Evolving concurrent systems – behavioural theory and logic. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2017)*, pages 77:1–77:10. ACM, 2017.
17. K.-D. Schewe, A. Prinz, and E. Börger. Concurrent computing with shared replicated memory. *CoRR*, abs/1902.04789, 2019.
18. A. S. Tanenbaum and M. Van Steen. *Distributed systems*. Prentice-Hall, 2007.
19. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic and the Foundations of Computer Science: Semantic Modelling*, volume 4, pages 1–148. Oxford University Press, 1995.