Egon Börger (Pisa) & Alexander Raschke (Ulm)

Relaxed Shared Memory Management

User view ground model, replication policies and refined database view

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 6 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Goal of the lecture

Illustrate the use of *concurrent communicating ambient ASMs*

- with dynamically changing set of agents

to define *ground models* and their *stepwise refinement*

CaseStudy: noSQL database system Cassandra, exhibiting

- data replication and replica distribution (hidden to the user)
- various replication policies (choosable by the user)
- background propagation of updates (hidden to the user)

Three models will be defined:

- a user request/response pattern view (ground model ASM)
- a replication policy refinement
- a data center interaction view (ASM refinement)

Consistency issue when sharing distributed replicated data

- read/write consistency in ASM runs: in each step every agent atomically reads and/or writes in its locations l well-defined unique values
  - simultaneous read/write actions of a location l are consistent
    - reading l returns the value of l in the current state whereas writing l determines the value of l in the next state

    Management
  - conflicts due to simultaneous writes to a location l are forbidden by the consistency constraint on update sets
- how to guarantee consistency of read/write actions for locations l with (possibly distributed) replica $l', l'', \ldots$ which are all considered to be legitimate copies of l?
  - example: IRIW (Independent Read Independent Write)

IRIW (Independent Read Independent Write) example

Assume the following:

- four agents $a_1, \ldots, a_4$ equipped with respective program

  $x := 1, y := 1, read(x)$ **step** $read(y), read(y)$ **step** $read(x)$

- a database with two replicas x', x'' resp. y', y'' of x, y
  - a write to x resp. y triggers an immediate update of one of the replicas, say x' resp. y', and is followed later by an internal propagation action which performs an update also of the other replica
  - a read action of x, y returns the value of one of the replicas

Then in a concurrent run of a_i, started with value 0 for all replicas and in which each agent makes once its unique move, it may happen that a_3 reads $x = 1, y = 0$ and a_4 reads $x = 0, y = 1$

- depending on which replica is directly affected by a write/read request and on the propagation time

Users should know whether such db inconsistencies are to be expected.

Structure of replication-based shared data in Cassandra

- *data organized by relations* $P_i \subseteq K^{a_i} \times B^{c_i}$ $(1 \leq i \leq k)$
  - data *accessed only via primary key values* (in a static set K^{a_i})
  - pairs (p_i, k) with $k \in K^{a_i}$ represent for users a a 'shared' location
    - with value $eval_a^S(p_i(k))$ for states S of a and agent-dependent fct $p_i : K^{a_i} \rightarrow B^{c_i}$ (NB. db-agents execute an ambient ASM)
- *relations horizontally fragmented* via a hash fct on primary key values: $h_i : K^{a_i} \rightarrow [m, M] \subseteq \mathbb{Z}$ with $[m, M]$ partitioned into intervals
  - i.e. $[m, M] = \bigcup_{j=1}^{q_i}$ *range*$_j$ (*range*$_{j_1}$ $<$ *range*$_{j_2}$ for all $j_1 < j_2$)

  hash fct h_i yields fragments $Frag_{j,i} = \{k \in K^{a_i} \mid h_i(k) \in range_j\}$
- P_i *stored in fragments*: r_i replicas of $Frag_{j,i}$ in data centers $d \in \mathcal{D}_i$
  - a replica of $Frag_{j,i}$ is a set of all key/value pairs (k, v) with key in $Frag_{j,i}$ and (timestamped) value stored in memory (namely the data centers below) to 'serve as possible value of $p_i(k)$'

Structure of data centers in Cassandra

- a data center d consists of a set of nodes, say $j' \in \{1, \dots, n_i\}$, where *replicas of fragments $Frag_{j,i}$ are stored via* replica fcts $p_{i,j,d,j'}$
  - $p_{i,j,d,j'}$ records the timestamped values $p_{i,j,d,j'}(k) = (v, t)$ stored in d for a replica of $Frag_{j,i}$. Denote this by $HoldsReplica(j', d, j, i)$.
- for each relation P_i, $Frag_{j,i}$ and all data centers, all *replica functions for $Frag_{j,i}$* are assumed to be defined for the same keys $k \in Frag_{j,i}$ but *may differ in their values*:
  - if $HoldsReplica(j', d, j, i)$ and $HoldsReplica(j^*, d^*, j, i)$, then
    - $p_{i,j,d,j'}(k)$ is defined if and only if $p_{i,j,d^*,j^*}(k)$ is defined
    - $p_{i,j,d,j'}(k)$ and $p_{i,j,d^*,j^*}(k)$ may have different (relation or timestamp) value
- a memory system like Cassandra manages a set of data centers
  - writing replicas (with fresh timestamp)
  - reading replicas (to return the one with latest timestamp)

Logical timestamps (Lamport)

Timestamps are evaluated resp. updated by a data center when it evaluates a read request resp. receives a write request.

Thus each data center d has a logical $clock_d$ indicating the current time at d and assumed to advance (here without further specification).

1. Timestamps are totally ordered.
2. Timestamps set by different data centers are different from each other.
3. Timestamps respect the inherent order of message passing, i. e. when data with a timestamp t is created at d and sent to d', then when the message is received, the clock at d' must show a time larger than t.
   - for synchronization purposes a data center may adjust its clock:

     $\textsc{AdjustClock}(d, t) = (clock_d := t')$

     where $t' =$ the smallest possible timestamp at d with $t \leq t'$
4. When a timestamp is set (except when adjusted), it is increased.

User view of the memory system (request/response pattern)

- users a interact with memory sending requests to and receiving answers from a dedicated data center $home(a) \in \mathcal{D}_i$ to
  - $write(p_i, p)$ where p may have multiple key/value-pairs (bulk write)
  - $read(p_i, \varphi)$ the current value of locations (p_i, k) for the set of keys $k \in K^{a_i}$ which satisfy condition $\varphi(k)$ (bulk read)
- $home(a)$ answers to a by $\textsc{Send}$ing an $AckFor(write(p_i, p))$ resp. the current $ValFor(read(p_i, \varphi), \rho)$ of replicas in an appropriate set ρ.

User a does not see how $home(a)$

- selects which replicas at which data centers in the cluster to read resp. update replica values
- propagates value updates and adjusts data center clocks
  - only compliance with a configurable read/write policy is guaranteed

Therefore we keep this internal memory management and the read/write policies abstract in the ground model $\textsc{DataCenterUserView}_i$.

Definition of $\textsc{AnswerReadReq}_i$

if $\mathit{Received}(\mathit{read}(p_i, \varphi),$ **from** $a)$ **then**

forall $j \in \{1, \ldots, q_i\}$ -- for each fragment choose complying replicas

choose $C_{i,j} \subseteq \mathit{ReplicaNodes}_{i,j}$ **with** $\mathit{Complies}(C_{i,j}, \mathit{readPolicy})$

let $t_{max}(k) =$ -- compute most recent timestamp (per key)

$\max\{t \mid p_{i,j,d',j'}(k) = (v', t)$ **forsome** $v', (d', j') \in C_{i,j}\}$

-- collect most recent defined values in j-th fragment

let $\rho_{i,j} = \{(k, v) \mid k \in \mathit{Frag}_{j,i} \uparrow \varphi$ **and** $v \neq \mathit{null}$ **and**

$p_{i,j,d',j'}(k) = (v, t_{\max}(k))$ **forsome** $(d', j') \in C_{i,j}\}$

let $\rho = \bigcup_{j=1}^{q_i} \rho_{i,j}$ -- collect values from all fragments

$\textsc{Send}(\mathit{ValFor}(\mathit{read}(p_i, \varphi), \rho),$ **to** $a)$ -- current (p_i, φ)- values

$\textsc{Consume}(\mathit{read}(p_i, \varphi),$ **from** $a)$

where $\mathit{ReplicaNodes}_{i,j} =$ -- NB. Data centers have different nodes

$\{(d', j') \mid d' \in \mathcal{D}_i$ **and** $1 \leq j' \leq n_i$ **and** $\mathit{HoldsReplica}(j', d', j, i)\}$

Definition of $\textsc{PerformWriteReq}_i$

if $Received(write(p_i, p),$ **from** $a)$ **then**

let $t_{current} = clock_{\textbf{self}}$ -- retrieve current data center time

forall $j \in \{1, \ldots, q_i\}$ -- for each fragment choose complying replicas

choose $C_{i,j} \subseteq ReplicaNodes_{i,j}$ **with** $Complies(C_{i,j}, writePolicy)$

forall $(d', j') \in C_{i,j}$ -- for each chosen d' and replica

forall $(k, v) \in p$ **with** $h_i(k) \in range_j$ -- for each p-value

forall v', t **with** $p_{i,j,d',j'}(k) = (v', t)$ **and** $t < t_{current}$

$p_{i,j,d',j'}(k) := (v, t_{current})$ -- update older db value

$\textsc{Propagate}(i, j, k, v, t_{current}, C_{i,j})$

-- propagate p-value to non-chosen replicas

if $clock_{d'} < t_{current}$ **then** $\textsc{AdjustClock}(d', t_{current})$

$\textsc{Send}(AckFor(write(p_i, p)),$ **to** $a)$

$\textsc{Consume}(write(p_i, p),$ **from** $a)$

Definition of system internal $\textsc{Propagate}(i, j, k, v, t, C)$

$\textsc{Propagate}(i, j, k, v, t, C) =$

let $b =$ **new** $(Agent)$ -- b executes asynchronously

$pgm(b) :=$

forall $(d', j') \in ReplicaNodes_{i,j} \setminus C$ -- not yet considered replicas

forall v', t' **with** $p_{i,j,d',j'}(k) = (v', t')$ **and** $t' < t$

$p_{i,j,d',j'}(k) := (v, t)$ -- update older db value

if $clock_{d'} < t_{current}$ **then** $\textsc{AdjustClock}(d', t)$

$\textsc{Delete}(b, Agent)$

NB. In the ground model the propagation is formulated as happening in one step, in parallel for all involved replicas, whereafter the propagation agent is not needed any more.

User request/response pattern view of the memory system

$\textsc{DataCenterUserView}_i =$

$\textsc{AnswerReadReq}_i$

$\textsc{PerformWriteReq}_i$

Then $\textsc{ClusterUserView}_i$ is the concurrent ASM of all data center agents $d \in \mathcal{D}_i$ with $pgm(d) = \textsc{DataCenterUserView}_i$.

NB. In a concurrent run of $\textsc{Users}$ and $\textsc{ClusterUserView}_i$, not furthermore restricted simultaneous read/write requests by different users may yield conflicts and/or inconsistencies, depending on

- timing issues: for the communication, the reads/writes of memory locations, the propagation of writes
- the replication policies

Detailing replication policies

Cassandra uses replication policies to determine *tunable consistency*, i.e. the degree of consistency which can be guaranteed for read/write actions in concurrent runs.

Technically, $readPolicy$ and $writePolicy$ are used to determine how many and where replicas are to be taken into account to perform a user requested db read or write

- i.e. they are about the cardinality of the selected subsets $C_{i,j} \subseteq ReplicaNodes_{i,j}$ and about where—at which data centers—replicas are chosen, e.g.
  - cardinality conditions: $one, two, three, all$
  - locality contraint: pair $(num, At(d))$ of the number of replica nodes and the data center d where they have to be taken
  - a $quorum$ relation between $C_{i,j}$ and $ReplicaNodes_{i,j}$

Definition of $Complies$

$Complies(C, policy)$ **iff forsome** (i, j)

$$C \subseteq ReplicaNodes_{i,j} \textbf{ and }$$

$$\begin{cases} C = ReplicaNodes_{i,j} & \textbf{if } policy = all \\ |C| = num & \textbf{if } policy = num \\ |C| = num \textbf{ and } C = C \uparrow d & \textbf{if } policy = (num, At(d)) \\ q \cdot |ReplicaNodes_{i,j}| < |C| & \textbf{if } policy = quorum(q) \end{cases}$$

where

$num \in \{one, two, three\}$ **and** $0 < q < 1$

$C \uparrow d = \{(d', j') \in C \mid d' = d\}$ -- replicas taken only in d

For exl. $q = \frac{1}{2}$ expresses that the majority of replicas is considered.

2 extreme consistency cases for ClusterUserView_i

- If $readPolicy = writePolicy = all$, then by the atomicity of ground model actions one can prove:
  - for every read request the unique freshest replica value is returned, i.e. the one with latest timestamp in the db state in which the request is elaborated
  - if one (and simultaneously no other) data center in a cluster receives a write request, this request triggers an update of all replicas in the db to the requested new value (NB. no propagation happens)
- under the one policy different replicas may have different values so that inconsistency phenomena can occur, as in the above IRIW exl.

For a rigorous definition and detailed analysis of other forms of consistency, achievable in the Cassandra ground model with appropriately restricted read/write policies, see the paper in the references below.

Data center interaction to perform db read/write actions

Idea: refine each atomic ground model step by letting $home(a)$

- $\textsc{DelegateExternalReq}_i$ to each involved data center d' for the work to be done at its local replica nodes $(d', j') \in C_{i,j}$
  - $\textsc{Forward}$ing the received request from $home(a)$ to every $d \in \mathcal{D}_i$
  - letting each data center $\textsc{ManageInternalReq}_i$ asynchronously
- trigger (as part of the $\textsc{DelegateExternalReq}_i$ step) an instance of a $\textsc{ManageResponseCollection}_i$ process to asynchronously
  - collect the received local responses
  - send the final response from $home(a)$ to the requestor
    - once the collected local responses are $Sufficient$ for the underlying read/write policy

Formal definition of $\textsc{ClusterManagement}_i$

Therefore the refinement $\textsc{ClusterManagement}_i$ of $\textsc{ClusterUserView}_i$ is defined as concurrent ASM of

- the *data center agents* $d \in \mathcal{D}_i$, each equipped with program $\textsc{DataCenterManagement}_i$ defined by:

  $\textsc{DataCenterManagement}_i =$
  $\quad \textsc{DelegateExternalReq}_i$
  $\quad \textsc{ManageInternalReq}_i$

- the still alive *response collector agents* the data center agents create and equip with program $\textsc{ManageResponseCollection}_i$
  – which is defined as part of $\textsc{DelegateExternalReq}_i$
  upon receiving a request

The functionality of $\textsc{DelegateExternalReq}_i$

When a *request from some user* a is $Received$, the receiving data center agent performs two actions:

- $\textsc{Forward}$ the reqest to all data centers $d \in \mathcal{D}_i$ for local handling
  - with info on current data center time and where to report the locally computed answer
    - computed on the basis of the replica data kept in the nodes of d
- Create a response collector agent c and $\textsc{Initialize}$ it
  - enabling c to $\textsc{ManageResponseCollection}$ asynchronously
    - including counting the number of locally inspected replicas, as reported from the data center agents, and needed to perform the global compliance check

Definition of $\textsc{DelegateExternalReq}_i$

$\textsc{DelegateExternalReq}_i =$

if $Received(req,$ **from** $a)$ **then**

let $t_{current} = clock_{\textbf{self}}$ -- retrieve current data center time

let $c =$ **new** $(Agent)$ -- create response collector

$\textsc{Initialize}_i(c, (req, a, \textbf{self}))$ -- and initialize it

$\textsc{Forward}_i(req, c, t_{current})$ -- delegate response

$\textsc{Consume}(req,$ **from** $a)$

where

$\textsc{Forward}_i(r, c, t) =$ -- delegate response throughout the cluster

forall $d \in \mathcal{D}_i$ **do** $\textsc{Send}((r, c, t),$ **to** $d)$ -- including $d =$ **self**

$req \in \{read(p_i, \varphi), write(p_i, p)\}$ **forsome** φ, p (with fixed p_i)

Information needed to $\textsc{ManageResponseCollection}$

- local record of the user, its *req*uest, the mediating data center d and of the set $ReadVal$ of values received for a read req from
- a counter for the number of inspected fragment replicas
  - needed for the policy compliance check

$\textsc{Initialize}_i(c, (r, usr, d)) =$

$pgm(c) := \textsc{ManageResponseCollection}_i$

$ReadVal_c := \emptyset$ -- initialize set where to collect responses

forall $d \in \mathcal{D}_i$ **forall** $1 \leq j \leq q_i$

$count_c(j, d) := 0$ -- inspected $Frag_{j,i}$-replicas at d

$count_c(j) := 0$ -- inspected $Frag_{j,i}$-replicas

$request_c := r$ -- record user request

$requestor_c := usr$ -- record user

$mediator_c := d$ -- record $home(usr)$

Management of Internal Requests

$\textsc{ManageInternalReq}_i =$

if $Received(req, c, t)$ **then**

$\textsc{HandleLocally}_i(req, c, t)$

$\textsc{Consume}(req, c, t)$

The ground model read/write actions are refined at each data center d by $\textsc{HandleLocally}_i(req, c, t)$ which

- performs the read/write action only for the nodes of d
- performs the read/write action for *all* nodes which are $Alive$ in d
- includes into the response the number of locally inspected nodes
  - because the policy compliance can only be performed globally, at the cluster level (here by the response collector c)

Therefore there are two versions $\textsc{HandleLocally}_i(read(p_i, \varphi), c, t)$ and $\textsc{HandleLocally}_i(write(p_i, p), c, t)$ to define.

$\textsc{HandleLocally}_i(\mathit{read}(p_i, \varphi), c, t)$

let $d =$ **self** $\in \mathcal{D}_i$ -- at the local data center d

forall $j \in \{1, \ldots, q_i\}$ -- for each fragment

let $G_{i,j,d} = \{j' \mid \mathit{HoldsReplica}(j', d, j, i)$ **and** $\mathit{Alive}(j', d)\}$

-- inspect replicas at all Alive d-nodes

let $t_{max}(k) =$ -- to compute their most recent timestamp

$\max\{t \mid p_{i,j,d,j'}(k) = (v', t)$ **forsome** $v', j' \in G_{i,j,d}\}$

let $\rho_{i,j,d} = \{(k, v, t_{max}(k)) \mid k \in \mathit{Frag}_{j,i} \uparrow \varphi$ **and** $v \neq \mathit{null}$ **and**

$p_{i,j,d,j'}(k) = (v, t_{\max}(k))$ **forsome** $j' \in G_{i,j,d}\}$

-- collect at d most recent defined values in j-th fragment

let $\rho_d = \bigcup_{j=1}^{q_i} \rho_{i,j,d}$ -- collect those values from all fragments

let $x_d = (|G_{i,1,d}|, \ldots, |G_{i,q_i,d}|)$ -- count inspected replicas

$\textsc{Send}(\mathit{LocalValFor}(\mathit{read}(p_i, \varphi), \rho_d, x_d),$ **to** $c)$

-- send local values to response collector

$\textsc{HandleLocally}(\mathit{write}(p_i, p), c, t)$

let $d =$ **self** $\in \mathcal{D}_i$ -- at the local data center d

forall $j \in \{1, \ldots, q_i\}$ -- for each fragment

let $G_{i,j,d} = \{j' \mid \mathit{HoldsReplica}(j', d, j, i)$ **and** $\mathit{Alive}(j', d)\}$

-- inspect replicas at all Alive d-nodes

forall $j' \in G_{i,j,d}$ -- for each of those replicas

forall $(k, v) \in p$ **with** $k \in \mathit{Frag}_{j,i}$ -- for each update value in p

if $p_{i,j,d,j'}(k) = (v', t')$ **with** $t' < t$ **forsome** v', t'

then $p_{i,j,d,j'}(k) := (v, t)$ -- update older values to p-value

if $\mathit{clock}_d < t$ **then** $\textsc{AdjustClock}(d, t)$

let $x = (|G_{i,1,d}|, \ldots, |G_{i,q_i,d}|)$ -- count inspected replicas

$\textsc{Send}(\mathit{LocalAckFor}(\mathit{write}(p_i, p), x),$ **to** $c)$

-- send local ack to response collector

Response collection and policy check

$\textsc{ManageResponseCollection}_i$ must

- collect received internal local read/write responses
- send the final read/write response once the local responses collected so far turn out to be $\mathit{ResponsesSufficientFor}$ the given policy

$\textsc{ManageResponseCollection}_i =$

$\textsc{CollectLocalReadResponses}_i$

$\textsc{SendReadResponse}$

$\textsc{CollectLocalWriteResponses}_i$

$\textsc{SendWriteResponse}$

CollectLocalWriteResponses

Collecting write responses means to RefreshReplicaCount

- NB. Write requests trigger only an ack, no values are sent back

$\textsc{CollectLocalWriteResponses}_i =$

if $Received(LocalAckFor(write(p_i, p), x),$ **from** $d)$ **then**

$\quad \textsc{RefreshReplicaCount}(x, d)$

$\quad \textsc{Consume}(LocalAckFor(write(p_i, p), x),$ **from** $d)$

where

$\quad \textsc{RefreshReplicaCount}(x, d) =$

$\quad\quad$ **let** $(x_1, \ldots, x_{q_i}) = x$ -- per fragment: replicas inspected at d

$\quad\quad\quad$ **forall** $j \in \{1, \ldots, q_i\}$ -- for each fragment

$\quad\quad\quad\quad count(j) := count(j) + x_j$ -- inspected $Frag_{j,i}$-replicas

$\quad\quad\quad\quad count(j, d) := count(j, d) + x_j$

-- inspected $Frag_{j,i}$-replicas at d

CollectLocalReadResponses

CollectLocalReadResponses =

if *Received*(*LocalValFor*(*read*(p_i, φ), ρ, x), **from** d) **then**

forall k **if thereissome** $(k, v, t) \in \rho$ **then** -- for each key k

let $(k, v, t) \in \rho$ -- with received local value v

if thereisno $(k, v', t') \in ReadVal$ -- if key k new for collection

then Insert($(k, v, t), ReadVal$) -- collect received value

else let $(k, v', t') \in ReadVal$ -- for k-value v' in collection

if $t' < t$ **then** -- with older timestamp

Delete($(k, v', t'), ReadVal$) -- replace old value

Insert($(k, v, t), ReadVal$) -- by new value

RefreshReplicaCount(x, d)

Consume(*LocalValFor*(*read*(p_i, φ), ρ, x), **from** d)

$\textsc{SendRead/WriteResponse}$ **if policy satisfied**

$\textsc{SendReadResponse} =$

if $ResponsesSufficientFor(readPolicy)$ **and**

$IsReadReq(request_{\mathbf{self}})$

then let $\rho = \{(k, v) \mid (k, v, t) \in ReadVal$ **forsome** $t\}$

$\textsc{Send}(ValFor(request_{\mathbf{self}}, \rho),$ -- send the collected values

from $mediator_{\mathbf{self}}$, **to** $requestor_{\mathbf{self}})$

$\textsc{Delete}(\mathbf{self}, Agent)$ -- collector kills itself

$\textsc{SendWriteResponse} =$

if $ResponsesSufficientFor(writePolicy)$ **and**

$IsWriteReq(request(\mathbf{self}))$ **then**

$\textsc{Send}(AckFor(request_{\mathbf{self}}),$ -- send an acknowledgement

from $mediator(\mathbf{self})$, **to** $requestor_{\mathbf{self}})$

$\textsc{Delete}(\mathbf{self}, Agent)$

The meaning of $ResponsesSufficientFor(policy)$

$ResponsesSufficientFor(num)$ **iff**

forall $1 \leq j \leq q_i \;\; count(j) \geq num$ -- for each fragment

NB. $\geq$ (instead of $=$) is due to the concurrency: when testing $ResponsesSufficientFor$, already more than num answers may have been received.

$ResponsesSufficientFor(all)$ **iff**

forall $1 \leq j \leq q_i \;\; count(j) = |ReplicaNodes_{i,j}|$

$ResponsesSufficientFor(quorum(q))$ **iff**

forall $1 \leq j \leq q_i \;\; q \cdot |ReplicaNodes_{i,j}| < count(j)$

where

$num \in \{one, two, three\}$ **and** $0 < q < 1$

A note on $Alive$ replica nodes

- Without restrictig in the policy definition the set $ReplicaNodes_{i,j}$ to $Alive$ nodes, the delegate can send its answer only if all relevant replica nodes are $Alive$ during the response providing process.
- This would contradict however the spirit of using replicas, namely to be on the safe side even when some replica holding node happens to be unreachable.
- To describe a satisfactory solution of the problem more information is needed on how not $Alive$ nodes are treated for the various policies.
- NB. A typical meaning of being $Alive$ is that a node in a data center is accessible and 'replies fast enough' to the request, providing the values they record.

Relating $\textsc{ClusterUserView}_i$ and $\textsc{ClusterManagement}_i$

- For each user request to its home data center in a concurrent run with the abstract machine $\textsc{ClusterUserView}_i$
  – where the response is computed using the abstract program $\textsc{DataCenterUserView}_i$

  one can construct an equivalent interaction (i.e. with same read/write effect) between the user and its home data center in a concurrent run with the refined machine $\textsc{ClusterManagement}_i$
  – where the response is computed interactively by the data center agents $d \in \mathcal{D}_i$ using the refined programs $\textsc{DataCenterManagement}_i$
- *The inverse holds only under additional assumptions* on the serializability of read/write requests and answers, using some form of transaction
  – see the example below
  – for a detailed analysis see the references

An (undesirable?) run example for $\textsc{ClusterManagement}_i$

- let replicas x_1 at d_1 and x_2 at d_2 of x be initialized by 0
- assume write policy all and read policy one
- let a_1 issue a write request $x := 1$ and thereafter a_2 issue two successive read requests for x

The following $\textsc{ClusterManagement}_i$ scenario one wouldn't expect from the ground model $\textsc{ClusterUserView}_i$ is possible:

- the write request by a_1 leads to
  - an update of x_1 to 1 before a_2 issues its first read request for x
    - which is answered by the value of replica x_1, namely 1
  - a later update of x_2 to 1 after the second read request has been answered
    - namely by the initial value 0 of replica x_2

  although meantime no new write request has been sent.

References

- Klaus-Dieter Schewe and Andreas Prinz and Egon Börger: Concurrent Computing with Shared Replicated Memory.
  - Springer LNCS 11815 (2019) 211-228
  - extended version in CoRR, vol. abs/1902.04789 (2019)
    `http://arxiv.org/abs/1902.04789`
    eprint arxiv.org/abs/1902.04789
- Apache Cassandra 2.0—Documentation (2016)
  `http://cassandra.apache.org`
- Egon Börger and Alexander Raschke: Modeling Companion for Software Practitioners. Springer 2018
  `http://modelingbook.informatik.uni-ulm.de`

Copyright Notice