Inna Pereverzeva | Linas Laibinis | Elena Troubitsyna |
Markus Holmberg | Mikko Pöri

Formal Modelling of Resilient Data Storage in the Cloud

Turku Centre *for* Computer Science

TUCS Technical Report
No 1076, May 2013

Formal Modelling of Resilient Data Storage in the Cloud

Inna Pereverzeva
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science
inna.pereverzeva@abo.fi

Linas Laibinis
Åbo Akademi University, Department of Computer Science
linas.laibinis@abo.fi

Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
elena.troubitsyna@abo.fi

Markus Holmberg
F-Secure, Helsinki, Finland
markus.holmberg@f-secure.com

Mikko Pöri
F-Secure, Helsinki, Finland
mikko.pori@f-secure.com

TUCS Technical Report
No 1076, May 2013

Abstract

Reliable and highly performant handling of large data stores constitutes one of the major challenges of cloud computing. In this paper, we propose a formalisation of a solution implemented by F-Secure – a provider of secure data storage services. The solution is based on massive replication and write-ahead logging mechanism. The company has abandoned transactional model to achieve high performance. We formally derive a model of the proposed architectural solution and verify data integrity properties under possible failure scenarios. The proposed approach allows the designers to formally define and verify the essential characteristics of architectures for handling large data stores.

Keywords: Formal modelling, Event-B, refinement, replication, data integrity, large data stores.

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Rapid development of digital technology puts a high demand on reliable handling and storage of large volumes of data. It is forecasted that "worldwide consumer digital storage needs will grow from 329 exabytes in 2011 to 4.1 zettabytes in 2016" [4]. Often algorithms for data storage in cloud reuse the ones that have been proposed for databases. The transactional model adopted in databases guarantees ACID properties – Atomicity, Consistency, Isolation and Durability, and as such delivers high resilience guarantees. However, in a pursue of high performance, cloud data storages rarely rely on the transactional model and hence deliver weaker guarantees regarding data integrity. In this paper, we undertake a formal study of data integrity properties that can be guaranteed by several different architectures of cloud data stores.

Our work is motivated by a cloud solution developed by F-Secure – a provider of secure data storage services. To achieve high degree of fault tolerance, the company has combined write-ahead logging (WAL) [7, 10] – a widely used mechanism for database error recovery – and massive data replication. As such, this combination gives very high resilience guarantees (usually in the form of eventual consistency). However, these guarantees are different in non-transactional settings typical for cloud. Moreover, data consistency properties vary in synchronous, semi-synchronous and asynchronous replication modes. Therefore, it is useful to rigorously define and compare the properties that can be ensured by different solutions.

In this paper, we use the Event-B method and the associated Rodin platform to formally model write-ahead logging in replicated data stores. Event-B [1] is a formal framework that is particularly suitable for development of distributed systems. System development starts from an abstract specification that is transformed into a detailed specification in a number of correctness-preserving refinement steps. In this paper, we start from an abstract specification of the write-ahead logging mechanism and refine it into three different models corresponding to the synchronous, semi-synchronous and asynchronous replication modes. Event-B and the Rodin platform [11] allows us to explicitly define the data consistency properties as model invariants and compare them in all three modes.

We believe that the proposed approach allows the designers to gain formally grounded insights on properties of cloud data stores and their resilience.

The paper structured as follows: in Section 2 we give a brief overview of our modelling framework – Event-B. In Section 3 we describe the write-ahead logging mechanism as well as the general architecture of a cloud data store. In Section 4 we present a formal development of the system in the asynchronous replication mode. In Section 5, we briefly (due to similarity with the asynchronous model) overview the models for the synchronous and

semi-synchronous modes and compare data integrity properties of all three replication modes. Finally, in Section 6 we overview the related work, discuss the obtained results and outline future work.

2 Modelling in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [1]. An Abstract State Machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important properties of the systems are defined in the model *invariants*. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. In Event-B, an assignment represents a corresponding next-state relation R_e. The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables. The proof of data refinement is often supported by supplying *witnesses* – the concrete values for the replaced abstract variables and parameters. Witnesses are specified in the event clause **with**.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform

[11] provides an automated support for modelling and verifying. In particular, it automatically generates the required proof obligations and attempts to discharge them.

Event-B adopts an event-based modelling style that facilitate a correct-by-construction development of distributed system. Since cloud data storage is a large-scale distributed system, Event-B is a natural choice for its formal modelling and verification. In the next section, we give a general overview of the cloud data storage architecture that we formally develop in Event-B.

3 Resilient Cloud Data Storage

Essentially, cloud data storage can be seen as a networked online data storage available for its clients as a cloud service. Data are stored in virtualised data stores (data pools) usually hosted by third parties. Physically, the data stores may span across multiple distributed servers. Cloud data storage providers, and in particular F-Secure, should ensure that their customers can safely and easily store their content and access it from their computers and mobile devices. Therefore, there is a clear demand to achieve resilience – data integrity and consistency – and high performance in handling data.

Write-ahead logging (WAL) is a standard data base technique for ensuring data integrity. The main principle of WAL is to apply the requested changes to data files only after they have been logged, i.e., after the log has been stored in the persistent storage (disk). WAL mechanism ensures fault-tolerance because in case of a crash, the system would be able to recover using the log. Moreover, the WAL mechanism helps to optimise performance, since only the log file (rather than all data changes) should be written to the permanent storage to guarantee that a transaction is committed.

The WAL mechanism has been thoroughly studied under the reliable persistent storage assumption, i.e., if the disk containing the log never crashes. However, in cloud implementing such a highly-reliable data store is rather unfeasible. Therefore, to ensure fault tolerance, F-Secure has proposed a solution that combines WAL with replication. The resulting system – distributed data store (DDS) consists of a number of nodes distributed across different physical locations. One of the nodes, called *master*, is appointed to serve incoming data requests from DDS clients and report on success or failure of such requests, i.e., acknowledges success or failure of a transaction. As a result, for instance, the client may receive an acknowledgement that the data have been successfully stored in the system. The remaining nodes, called *standby nodes*, contain replicas of the stored data.

Each request received by the master is translated into a number of reading and writing commands. These commands are first recorded in the *master log* and then applied to the data storage. After this the acknowledgement is sent to the client. (In the non-replicated version of WAL widely used in

the data bases, the acknowledgement to the client is sent already after the request is written in the log). The standby nodes are constantly monitoring and streaming the master log records into their own logs, before applying them to their persistent storage in the same way. Essentially, the standby nodes are continually trying to "catch up" with the master. If the master crashes, one of the standby nodes is appointed to be the master in its stead. At this point, the standby effectively becomes the new master and starts serving all data requests.

DDS can implement different modes of logging. In the asynchronous mode the client request is acknowledged after the master node has performed the required modifications in its persistent storage. The second option – a cascade master-standby – is a semi-synchronous mode. The client receives an acknowledgement after the master and its warm standby (called upper standby) perform the required manipulations with their persistent store. Finally, in the synchronous mode, only after all replica nodes have written in their persistent storage, i.e., fully synchronise d with the master node, the transaction can be committed. Obviously, different modes deliver different resilience guarantee.

In our formal modelling, we aim at formally defining and comparing data integrity properties that can be ensured by each architecture. In the next section, we present a formal development of DDS in the asynchronous mode.

4 Modelling Resilient Data Storage in the Asynchronous Mode

In the asynchronous mode of replication, the standby nodes may stream the master log records only after the required changes have been committed and reported to the client. If master crashes shortly after committing the required modifications, some changes will not be replicated thus leading to an inconsistent system state. In particular, this might happen because a standby node has not yet received (streamed) all the master log records when the master failed. To minimise such data loss, the node that has the freshesest (and hence the most complete) copy of the master log is chosen to become the next master. A graphical representation of the system architecture is shown in Fig.1.

Abstract specification. The initial model – the machine Replication1_m0 abstractly describes the behaviour of the master node – receiving and processing of the received requests. The overall model structure is given on Fig.2.

The variable *comp*, $comp \subseteq COMP$, represents the dynamic set of active system nodes (data stores), where $COMP$ is a set (type) of all available data stores. The variable *master*, such that $master \in comp$, represents the

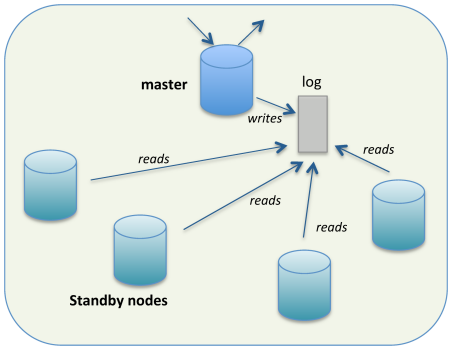

Figure 1: Asynchronous model.

master node.

The other variables *buffer*, *inprocess* and *processed* represent the received data requests at different stages of their processing by the master. They are modelled as disjoints sets of the abstract data type $REQUESTS$. In particular, the variable *buffer* stores the requests that have been received by the master and are waiting to be handled. The variable *inprocess* contains the requests that the master node is currently processing, while the variable *processed* keeps the requests that are completed and acknowledged to the client.

Machine Replication1_m0
Variables $comp, master, buffer, inprocess, processed$
Invariants $comp \subseteq COMP \ \wedge \ master \in comp \ \wedge \ buffer \subseteq REQUESTS$ ...
Events
RequestIn $\widehat{=}$ // arriving of a new request
any r
where $r \in REQUESTS \ \wedge \ r \notin buffer \ \wedge \ r \notin inprocess \ \wedge \ r \notin processed$
then $buffer := buffer \cup \{r\}$
end
Process ... // request processing
RequestOut $\widehat{=}$ // completion of a request
any r
where $r \in inprocess \ \wedge \ r \notin processed$
then $processed := processed \cup \{r\} \parallel inprocess := inprocess \setminus \{r\}$
end
ChangeMaster $\widehat{=}$ // changing the master
any $new_master, n_buffer, n_inprocess, n_processed$
where
$new_master \in comp \ \wedge \ new_master \neq master \ \wedge \ n_inprocess \cap n_buffer = \varnothing \ \wedge$
$n_inprocess \cap n_processed = \varnothing \ \wedge \ n_processed \cap n_buffer = \varnothing \ \wedge$
$n_buffer \cup n_inprocess \cup n_processed \subseteq buffer \cup inprocess \cup processed$
then
$master := new_master \parallel buffer := n_buffer \parallel inprocess := n_inprocess \parallel$
$processed := n_processed \parallel comp := comp \setminus \{master\}$
end
CompActivation ... // activation of a new system component
CompDeactivation ... // deactivation of a system component

Figure 2: Asynchronous model: the abstract model.

The event RequestIn specifies arriving of a new request to the master node. Processing of the received requests and sending notifications to the client are modelled by the events Process and RequestOut respectively. The events update to the variables *buffer*, *inprocess* and *processed* to reflect the progress in request handling.

The event ChangeMaster models a crash of a master node and selection of a new master. Non-deterministically one of the remaining nodes is chosen to become a new master, while the old master is removed from the set of active nodes. Due to possible data loss, the requests being handled by the new master may be only a subset of those of the failed master. This is reflected by the guard condition

$$n_buffer \cup n_inprocess \cup n_processed \subseteq buffer \cup inprocess \cup processed,$$

where *n_buffer*, *n_inprocess*, and *n_processed* are the corresponding data structures of the new master.

Finally, the last two events, CompActivation and CompDeactivation, model a possibility to add new data storage nodes from cloud and remove some currently active nodes from the system respectively. Only standby nodes can be activated and deactivated in this way.

First Refinement. In the first refinement step (defined by the machine Replication1_ref1), we extend the abstract model by explicitly representing the behaviour of the standby nodes.

To accomplish this, we lift the abstract variables *buffer*, *inprocess*, *processed* to become node-dependent functions. In Event-B, this is achieved by data refinement that replaces these variables with the new variables *comp_buffer*, *comp_inprocess* and *comp_processed*. The following gluing invariants are defined to to prove correctness of data refinement:

$$comp_buffer \in comp \to \mathbb{P}(REQUESTS) \wedge comp_buffer(master) = buffer \wedge$$
$$comp_inprocess \in comp \to \mathbb{P}(REQUESTS) \wedge comp_inprocess(master) = inprocess \wedge$$
$$comp_processed \in comp \to \mathbb{P}(REQUESTS) \wedge comp_processed(master) = processed.$$

The overview of the refined model is presented in the Fig. 3. The set of model events includes the refined versions of the abstract events (RequestInMst, ProcessMst, RequestOutMst, ChangeMaster, CompActivation, and CompDeactivation) as well as new events describing the behaviour of standby nodes.

We refine the event ChangeMaster to a deterministic procedure of choosing the node with the freshest log as a new master to the failed master. We formulate this condition as a new guard of the ChangeMaster event in the

Machine Replication1_ref1 refines Replication1_m0
Variables $comp, master, comp_buffer, comp_inprocess, comp_processed, failed, in_transit$
Invariants ...
Events
RequestInMst refines RequestIn ... // arriving of a new request to the master
ProcessMst refines Process ... // request processing by the master
RequestOutMst refines RequestOut ... // completion of a request by the master
RequestInStb $\widehat{=}$ ... // reading the master by a standby
any r
where $c \in comp \ \wedge \ c \neq master \ \wedge \ r \in comp_processed(master) \wedge$
$r \notin comp_buffer(c) \ \wedge \ r \notin comp_inprocess(c) \ \wedge \ r \notin comp_processed(c)$
$c \notin failed \ \wedge \ master \notin failed$
then $comp_buffer(c) := comp_buffer(c) \cup \{r\}$
end
ProcessStb ... // request processing by a standby
RequestOutStb ... // completion of a request by a standby
ChangeMaster refines ChangeMaster $\widehat{=}$ // changing the master
any new_master
where $new_master \in comp \ \wedge \ new_master \neq master \wedge$
$master \in failed \ \wedge \ new_master \notin failed \wedge$
$(\forall c \cdot c \in comp \wedge c \neq new_master \wedge c \neq master \Rightarrow$
$comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq$
$comp_buffer(new_master) \cup comp_inprocess(new_master) \cup$
$comp_processed(new_master))$
with $n_buffer = comp_buffer(new_master)$
$n_inprocess = comp_inprocess(new_master)$
$n_processed = comp_processed(new_master)$
then $master := new_master \parallel comp_buffer := \{master\} \lhd comp_buffer \parallel$
$comp_inprocess := \{master\} \lhd comp_inprocess \parallel$
$comp_processed := \{master\} \lhd comp_processed \parallel$
$comp := comp \setminus \{master\} \parallel failed := failed \setminus \{master\} \parallel in_transit := TRUE$
end
CompActivation ... // activation of a new component into the system
CompDeactivation ... // deactivation of a component from the system
CompFailure ... // modelling a component failure
CompStbRecovery ... // recovery of a standby
TransitionOver ... // all the standby nodes have only the requests already processed by the new master

Figure 3: Asynchronous model: the first refinement model

following way:

$$(\forall c \cdot c \in comp \wedge c \neq new_master \wedge c \neq master \ \Rightarrow \\ comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq \\ comp_buffer(new_master) \cup comp_inprocess(new_master) \cup \\ comp_processed(new_master)). \quad (1)$$

The standby nodes are continuously streaming the master log. Essentially, this means that, as soon as the master node completes the request(s), i.e., performs the required modifications in its persistent storage, the standby nodes start copying the corresponding entries in the master log. This behaviour is modelled by the new event RequestInStb. Similarly as for the master node, the processing of requests and their completion by the standby nodes are respectively modelled by the events ProcessStb and RequestOutStb.

In our model, we assume that the nodes might become temporary unavailable (i.e., crush and recover). The new variable *failed*, $failed \subseteq comp$,

is introduced to store such failed nodes. The new event CompFailure and CompStbRecovery model possible nodes crashes and recoveries correspondingly.

Now we are ready to formulate and prove some data consistency properties expressing the relationships between the requests handled by the master and those handled by the standby nodes. Since any standby node is continuously copying the master log, we can say that any standby node is logically "behind" the master node. Mathematically, this means that all the standby requests (no matter what stage of processing they are in) are subset of those of the master node. Moreover, all the requests that are now handled by a standby node should have been already completed by the master before. We can formulate these two properties as the following system invariants:

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master)), \quad (2)$$

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_processed(master)). \quad (3)$$

As it turns out, the last property cannot be proven as an (unconditional) invariant of the system. Indeed, it can be violated right after one of standby nodes is appointed the new master. A short transitional period may be needed for the new master to "catch up" with some of the standby nodes that got ahead by handling the requests still not committed by the new master. It is easy to show termination of this transitional period, since all such standby nodes are blocked from reading any new requests from the master until the master catches up with them by processing its requests.

We can formally model this transitional stage by introducing the variable $in_transit$, $in_transit \in BOOL$. The variable obtains the value $TRUE$ when a new master is appointed, and reobtains the value $FALSE$ (in the new event TransitionOver) when all the remaining standby nodes have the requests already processed by the new master.

Then we can reformulate the property (3) as a system invariant and prove its preservation:

$$in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_processed(master)). \quad (4)$$

Second Refinement. In the previous refinement step we introduced the standby nodes and their interactions with the master. We also modelled how the received data requests are transferred through the different processing

stages on the master and standby sides. The variables *buffer*, *inprocess* and *processed* were used to store incoming, processing and processed requests. The goal of our second refinement step is explicitly model the write-ahead logging (WAL) mechanism and the resulting interdependencies between the master and standby logs.

Mathematically, any log can be represented as a sequence, i.e., as a function of the type

$$any_log \ \in \ 1..k \ \rightarrow \ ELEMENTS$$

where k is the index of the last written element.

In our case, we want store in the node log all the requests – received, being processed, or completed. This can be represented as partitioning of the component log into three separate parts. To achieve that, we introduce three variables *index_written*, *index_inprocess*, and *index_processed*:

$$index_written \in comp \rightarrow NAT, \ index_inprocess \in comp \rightarrow NAT,$$

$$index_processed \in comp \rightarrow NAT,$$

such that

$$\forall c \cdot c \in comp \ \Rightarrow \ index_inprocess(c) \leq index_written(c),$$
$$\forall c \cdot c \in comp \ \Rightarrow \ index_processed(c) \leq index_inprocess(c).$$

For any component c, $index_written(c)$ defines the index of the last written log entry, $index_inprocess(c)$ – the index of the last request being processed, and $index_processed(c)$ – the index of the last completed request. Graphically, this can be represented as shown in Fig.4.

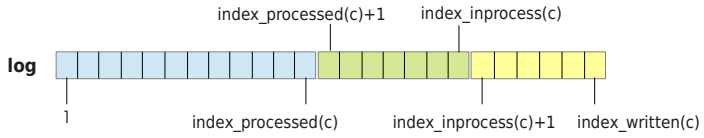

Figure 4: The log partition.

Then the logs of all the components can be defined as the following function:

$$log \in comp \rightarrow (NAT \nrightarrow REQUESTS),$$

such that

$$\forall c \in comp \cdot \ \mathbf{dom}(log(c)) \ = \ 1 \,..\, index_written(c)$$

where **dom** is the functional domain operator.

The function *log* is introduced to replace (data refine) the abstract variables *comp_buffer*, *comp_inprocess*, and *comp_processed*. To prove correctness of such data refinement, the following gluing invariants are added:

$$\forall c \cdot c \in comp \Rightarrow log(c)[index_inprocess(c) + 1 \,..\, index_written(c)] = comp_buffer(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[index_processed(c) + 1 \,..\, index_inprocess(c)] = comp_inprocess(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index_processed(c)] = comp_processed(c),$$

where $R[S]$ denotes relational image of R with respect to the given set S.

An introduction of the sequential representation of the component log allows us to refine some proven invariants as well as prove some new ones. For instance, the invariant property (4) now can be reformulated in terms of new variables as follows:

$$in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow$$
$$log(c)[1 \,..\, index_written(c)] \subseteq log(master)[1 \,..\, index_processed(master)]). \quad (5)$$

The formulated data refinement also affects all the events where the abstract variables were used. For instance, the event RequestOutMst (see Fig. 5) now specifies completion of master request processing by recording this in the node log, i.e., by increasing $index_processed(master)$.

We can refine the procedure of choosing a new master by reformulating the guard condition (1) of the event ChangeMaster as follows:

$$\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq new_master \Rightarrow$$
$$index_written(c) \leq index_written(new_master). \quad (6)$$

Here we check that the new candidate for the master has the largest $index_written$, i.e., the freshest log copy. The other events are refined in a similar way. The overview of the refined model is presented in Fig. 5.

Machine Replication1_ref2 refines Replication1_ref1
Variables $comp, master, comp_buffer, comp_inprocess, comp_processed, failed, in_transit$
Invariants...
Events
RequestInMst $\widehat{=}$ refines RequestInMst ... // arriving of a new request to the master
ProcessMst refines ProcessMst ... // request processing by the master
RequestOutMst $\widehat{=}$ refines RequestOutMst ... // completion of a request by the master
when $index_processed(master) \neq index_inprocess(master) \wedge master \notin failed$
with $r = log(master)(index_processed(master) + 1)$
then $index_processed(master) := index_processed(master) + 1$
end
RequestInStb $\widehat{=}$ refines RequestInStb ... // reading the master by a standby
any c
where $c \in comp \wedge c \neq master \wedge c \notin failed \wedge master \notin failed \wedge$
$index_written(c) < index_processed(master)$
with $r = log(master)(index_written(c) + 1)$
then $log(c) := log(c) \cup \{index_written(c) + 1 \mapsto log(master)(index_written(c) + 1)\} \parallel$
$index_written(c) := index_written(c) + 1$
end
ProcessStb $\widehat{=}$ refines ProcessStb ... // request processing by a standby
RequestOutStb $\widehat{=}$ refines RequestOutStb ... // completion of a request by a standby
ChangeMaster $\widehat{=}$ refines ChangeMaster ... // changing the master
...

Figure 5: Asynchronous model: the second refinement model.

Moreover, we can explicitly formulate and prove the log replication properties as model invariants:

$$\forall c, i \cdot c \in comp \wedge i \in 1 \,..\, index_written(c) \Rightarrow log(c)(i) = log(master)(i),$$

$$\forall c1, c2, i \cdot c1 \in comp \land c2 \in comp \land i \in 1 \mathinner{..} index_written(c1) \land$$
$$i \in 1 \mathinner{..} index_written(c2) \Rightarrow log(c1)(i) = log(c2)(i). \quad (7)$$

These properties state that the corresponding log elements of any two storage (master or standby) nodes are always the same. In other words, all logs are consistent with respect to the log records of the master node.

5 Cascade Master-Standby and Synchronous Models

An alternative, semi-syncronous mode of replication is the *cascade master-standby model*. Besides the *master* node that serves incoming data base requests, we single out another functional node - *upper standby*. The upper standby node starts streaming the master log as soon as the master records the requests in its log. Moreover, the master node waits until the upper standby reads its processed records and, only after that, commits the changes and reports to the client.

In its turn, the other standby nodes are constantly monitoring and streaming the upper standby log records into their own logs and applying them in the same way as described in Section 4. Essentially, the standby nodes are continually trying to catch up with the upper standby.

If the master node goes down, the upper standby node is automatically appointed to be the master in its stead. Moreover, the next candidate for the new upper standby node becomes the node that is closest (with respect to the copied log file) to the current upper standby.

Let us note that this proposed cascade replication mode allows to decrease the possibility of loss of the committed changes if the master node fails. Indeed, at that point, when the master node fails, the upper standby node had already recorded all the changes that were committed and reported to the client by master before. Therefore, such an architectural solution increases the system resilience. A possibility of data loss leading to an inconsistent system state is still present. However, for this to happen, the master node and the upper standby node should both fail in a very short time period. A graphical representation of the system architecture is shown on Fig.6.

The formal development of the proposed replication mode consists of an initial specification and its two refinements. The initial model abstractly describes the system behaviour focusing on the master and the upper standby nodes. The first refinement step introduces the remaining standby nodes and their interoperation with the upper standby, while the second refinement explicitly models the sequential logging mechanism and the interdependencies between the master, the upper standby and others standby logs. Let us note

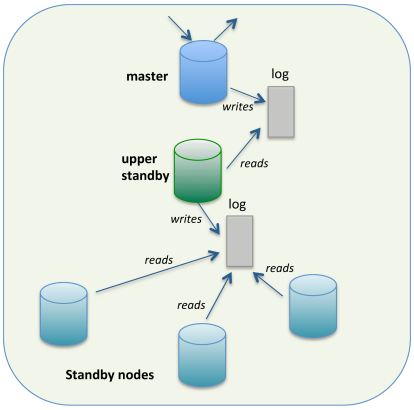

Figure 6: Cascade model.

that the development is similar to that of the asynchronous model. Due to the lack of space we will only highlight the most significant differences between them.

Abstract specification. In the initial model defined by the machine Replication2_m0 we focus on the master and upper standby components and their interoperation. The overall model structure is given on Fig. 7.

Machine Replication2_m0
Variables $comp, master, ups_standby, m_buffer, m_inprocess, m_processed, m_buffer, ...$
Invariants $comp \subseteq COMP \wedge master \in comp \wedge ups_standby \in comp ...$
Events
...
RequestInMst ... // arriving of a new request to the master node
ProcessMst ... // request processing
RequestOutMst $\widehat{=}$ // completion of a request by the master
any r
where $r \in m_inprocess \ \wedge \ (r \in ups_buffer \cup ups_inprocess \cup ups_processed)$
then $m_processed := m_processed \cup \{r\} \parallel m_inprocess := m_inprocess \setminus \{r\}$
end
ProcessStb ... // request processing by the upper standby
ChangeMaster $\widehat{=}$ // changing the master
any $new_ups_standby, n_buffer, n_inprocess, n_processed$
where
$new_ups_standby \in comp \ \wedge \ new_ups_standby \neq ups_standby \wedge$
$new_ups_standby \neq master \ \wedge \ ...$
$(n_buffer \cup n_inprocess \cup n_processed \subseteq$
$ups_buffer \cup ups_inprocess \cup ups_processed)$
then
$master := ups_standby \parallel ups_standby := new_ups_standby \parallel$
$m_buffer := ups_buffer \parallel m_inprocess := ups_inprocess \parallel$
$m_processed := ups_processed \parallel comp := comp \setminus \{master\} ...$
end
ChangeUpsStb ... // changing the upper standby node
CompActivation ... // activation of a new system component
CompDeactivation ... // deactivation of a system component

Figure 7: Cascade model: abstract model

In addition to the master node, we single out one more node to serve as an upper standby node. We model this by introducing the variable *ups_stanbdy*, such that $ups_stanbdy \in comp$ and $ups_standby \neq master$.

The variables *m_buffer*, *m_inprocess*, *m_processed* represent the received data requests at different stages of their processing by the master. Similarly, the variables *ups_buffer*, *ups_inprocess*, *ups_processed* are introduced to model the respective data structures for the the upper standby. The events RequestInMst, ProcessMst, RequestOutMst and RequestInStb, ProcessStb, RequestOutStb specify the corresponding request handling stages for the master and upper standby nodes.

The master node can not commit the changes until the upper standby reads them. We model this requirement by adding the following guard condition in the event RequestOutMst:

$$r \in ups_buffer \cup ups_inprocess \cup ups_processed.$$

The process of changing of the master node by the upper standby is modelled by the event ChangeMaster. The event also specifies the selection procedure of a new upper standby. Due to possible data loss, the requests being handled by the new upper standby may be only a subset of those of the current upper standby:

$$n_buffer \cup n_inprocess \cup n_processed \subseteq ups_buffer \cup ups_inprocess \cup ups_processed.$$

Moreover, a similar event, ChangeUpsStb, models the selection of a new upper standby in the case when the current one fails.

First Refinement. In the first refinement step we extend the abstract model by explicitly introducing the behaviour of the remaining standby nodes as well. Similarly as for the asynchronous model, we data refine the abstract variables *m_buffer*, *m_inprocess*, *m_processed* and *ups_buffer*, *ups_inprocess*, *ups_processed* by the new functional variables *comp_buffer*, *comp_inprocess* and *comp_processed*.

In addition, a number of the new events are added to describe the behaviour of standby nodes, node failures and recovery (RequestInStb, ProcessStb, RequestOutStb, CompFailure, CompStbRecovery).

As for the asynchronous mode, we can formulate and prove data consistency properties between the involved components. The property (2) (stating that a standby node is always behind the master in terms of handled requests) corresponds to two properties for the cascade replication mode: the first one stating this property between any standby and the upper standby, while the second one stating the same between the upper standby and master nodes.

$$\begin{aligned}(\forall c \cdot c \in comp \wedge c \neq master \ \wedge c \neq ups_standby \Rightarrow \\ comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq \\ comp_buffer(ups_standby) \cup comp_inprocess(ups_standby) \\ \cup comp_processed(ups_standby)), \quad (8)\end{aligned}$$

$$comp_buffer(ups_standby) \cup comp_inprocess(ups_standby) \cup comp_processed(ups_standby) \subseteq comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master), \quad (9)$$

The property (4) for the asynchronous mode expressed the relationships between the processed requests of the master node and read requests of the standby nodes. This property again corresponds to two properties for the cascade mode: one between the upper standby and remaining standbys, and the other one between the master and upper standby nodes. In both cases, the properties may be violated for a short period (indicated by $in_transit = TRUE$) right after a new upper standby node is chosen to replace a failed one:

$$in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \Rightarrow (comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_processed(ups_standby)). \quad (10)$$

$$in_transit = FALSE \Rightarrow (comp_processed(master) \subseteq comp_buffer(ups_standby) \cup comp_inprocess(ups_standby) \cup comp_processed(ups_standby)). \quad (11)$$

Note how the requirement that the master node cannot commit a request before it is read by the upper standby reverses the inclusion relationship in the property (11).

Second Refinement. The goal of our second refinement step is explicitly model the write-ahead logging mechanism and the resulting interdependencies between the master, upper standby and other standby logs.

We data refine the abstract variables *comp_buffer*, *comp_inprocess*, and *comp_processed* by the introduced function *log*. The following gluing invariants allow us to prove correctness of such a data refinement:

$$\forall c \cdot c \in comp \Rightarrow log(c)[index_inprocess(c) + 1 \,..\, index_written(c)] = comp_buffer(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[index_processed(c) + 1 \,..\, index_inprocess(c)] = comp_inprocess(c),$$
$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index_processed(c)] = comp_processed(c).$$

Introducing the sequential representation of the component log allows us to reformulate some proven invariants as well as prove some new ones. For instance, the invariant properties (10) and (11) now can be reformulated in terms of the new variables as follows:

$$in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \Rightarrow log(c)[1 \,..\, index_written(c)] \subseteq log(ups_standby)[1 \,..\, index_processed(ups_standby)]). \quad (12)$$

$$in_transit = FALSE \Rightarrow log(master)[1 \,..\, index_processed(master)] \subseteq log(ups_standby)[1 \,..\, index_written(ups_standby)]. \quad (13)$$

Finally, the log data integrity properties (in the exact form as in (7)) are formulated and proved for this replication mode as well.

Synchronous replication mode The last development formalises the *synchronous replication mode*, which can be considered as a combination of both asynchronous and cascade modes. The essential differences of this model are following. The standby nodes start streaming the master log records as soon as master records the commands in its log. Moreover, the master node waits until *all the standby nodes* read processed records from its log and, only after that, commits the corresponding changes and reports to the client. If the master goes down, one of the standby nodes is appointed to be the master in its stead. Essentially, it is a generalisation of the cascade mode where all the standby nodes play the role of upper standby.

This mode allows to avoid a possibility of loss of the committed changes if the master node fails. Indeed, at that point, all the standby nodes have already recorded all the changes that were committed and reported to the client by master. On the other hand, the necessity for the master to synchronise in such a way with all the standbys may negatively affect the performance of this model.

Developing the formal model of this mode, we essentially repeat the refinement steps of the asynchronous model. In particular, the initial model is the same as the abstract model presented on Fig.2. In the first refinement step, in the RequestOutMst event modelling the commitment of the changes by master, we have to impose an additional restriction for this behaviour. Namely, the master node can not commit the changes until all the standby nodes have read them. We model this requirement by adding the following guard condition to the event:

$$\forall c \cdot c \in comp \wedge c \neq master \Rightarrow r \in comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c),$$

where we check that the request r has already been recorded by all the standby nodes. Moreover, in the eventRequestInStb, we relax its guard by allowing to copy the master log as soon as the master records requests in its log.

Similarly as for the first two modes, we formulate and prove log data consistency properties. Specifically, the property (2), stating that that the standby nodes are continuously trying to catch up with the master in terms of handled requests, can be proved for this architecture as well. Moreover, since the master node can not commit the changes until the all standbys have read the corresponding log records, it means that all the requests committed

by the master have been previously read by all standbys. We can formulate this property as follows:

$$in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_processed(master) \subseteq comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c)). \quad (14)$$

Note that, once again, this property can violated right after a new master is appointed and thus a transitional period is needed. This property is very similar to that of (11) (for the cascade architecture) and is inverse, with respect to the inclusion relation, to that of (4) (for the asynchronous architecture).

As in the previous two developments, in the second refinement step we introduce component logs as sequences. In terms of the new variables, the (14) property can be then reformulated as follows:

$$in_transit = FALSE \;\Rightarrow\; (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(master)[1 \,..\, index_processed(master)] \subseteq log(c)[1 \,..\, index_written(c)]). \quad (15)$$

Finally, the log data integrity properties (7), stating that the corresponding log elements of any two storage components are always the same, are proved for this model as well. The full formal developments can be found in Appendix.

Proof Statistics To verify correctness of the presented formal developments, we have discharged more than 400 proof obligations for the first model, more than 750 proof obligations for the second model, and around 400 for the third model. In total, around 90% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. The proof statistics in terms of generated proof obligations for the presented Event B developments is shown in the Table 1. The numbers represent the total number of proof obligations and the percentage of manual effort for each model in each refinement step.

Table 1: The proof statistics.

	Asynchronous model			**Cascade model**			**Synchronous model**		
step	Total	Manual	Manual %	Total	Manual	Manual %	Total	Manual	Manual %
m0	18	0	0	53	0	0	18	0	0
ref1	177	0	0	257	1	0.3	146	0	0
ref2	231	39	16.8	442	75	16.9	232	42	18.1
Overall	426	39	9.1	752	76	10.1	396	42	10.6

6 Conclusions and Related Work

In this paper, we formalised an industrial approach to implementing resilient cloud data storage. To ensure resilience, F-Secure combined the write-ahead loaging mechanism with the log replication. We have formally expressed data consistency properties in three different replication modes and explicitly identified situations that lead to data loss. These properties can inform industry practitioners on resilience guarantees inherent to each solution.

The proposed modelling approach can facilitate early design exploration and evaluate benefits of different fault tolerance mechanisms in implementing resilience requirements.

The problem of data consistency in replicated data stores, im particular for the cloud has been actively studied in both – database and fault tolerance – communities, e.g., see [2, 8]. However, most of these approaches aim at proposing new protocols guaranteeing correctness of data replication. For instance, the work [8] presents a Dynamic Multi-Replica Provable Data Procession scheme. It is based on probabilistic encryption used to periodically verify the correctness and completeness of multiple data copies stored in the Cloud. In our work we aim at modelling architectural aspects of data stores and verifying qualitative characteristics of data handling in the Cloud.

Formal analysis of data base replication protocols have been proposed in [3]. The authors establish safety and liveness correctness criteria that need to be verified by a replication protocol. In contract, in our work we focus on ensuring data integrity and consistency properties based on the concrete, write-ahead logging mechanism.

The write-ahead logging mechanism (WAL) has been investigated in [5, 6]. In those works the authors analyse the performance aspects of this technique. They distinguish four types of the delays that the WAL mechanism can impose on transaction handling and propose an approach to increase log scalability. In our work, we focus on verifying properties concerning data consistency and data loss under possible failure scenarios using the WAL mechanism.

In [12] the authors investigate data properties in the presence of server failures, however considering the full ACID properties. A formal modelling of fault-tolerant transactions for replicated database systems using Event-B has been also undertaken by D. Yadav et al. [13]. The work focuses on data atomic commitment of distributed transaction.

In our future work we are planning to analyse performance characteristics of different replication modes. It particular, it would be interesting to integrate probabilistic verification to evaluate the trade-offs between performance and reliability.

References

[1] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[2] A. F. Barsoum and M. A. Hasan. Integrity Verification of Multiple Data Copies over Untrusted Cloud Servers. In *International Symposium on Cluster, Cloud and Grid Computing 2012*, pages 829–834. IEEE Computer Society, 2012.

[3] J.R.G. de Mendvil, J. E. Armendriz-Iigo, J. R. Garitagoitia, and F. D. Muoz-Esco. Scalability of write-ahead logging on multicore and multisocket hardware. *The Journal of Supercomputing*, 50:121–161, 2009.

[4] F-Secure. http://www.f-secure.com/en/web/operators_global/content-anywhere.

[5] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. volume 3, pages 681–692. VLDB Endowment, 2010.

[6] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.*, 21:239–263, 2012.

[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17:94–162, 1992.

[8] R. Mukundan, S. Madria, and M. Linderman. Replicated Data Integrity Verification in Cloud. *IEEE Data Eng. Bull.*, 35:55–64, 2012.

[9] Inna Pereverzeva, Linas Laibinis, Elena Troubitsyna, Markus Holmberg, and Mikko Pöri. Formal Modelling of Resilient Data Storage in the Cloud. Technical report, Turku Centre for Computer Science, 2013.

[10] PostgreSQL Documentation: WAL. http://www.postgresql.org/docs/9.2/static/wal-intro.html.

[11] Rodin. Event-B Platform, online at http://www.event-b.org/.

[12] Z. Wei, G. Pierre, and Chi-Hung Chi. Scalable Transactions for Web Applications in the Cloud. In *Euro-Par 2009 Parallel Processing*, pages 442–453. Springer-Verlag Berlin Heidelberg, 2009.

[13] D. Yadav and M. Butler. Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event-B. In A. Romanovsky M. Butler, C. Jones and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *LNCS*, pages 343–363. Springer, 2006.

Appendix

Event-B Development

---

1. Asynchronous model

MACHINE replication1_m0

SEES replication_cnt

VARIABLES

buffer

inprocess

comp

master

processed

INVARIANTS

inv1 : buffer $\subseteq$ REQUESTS

inv2 : inprocess $\subseteq$ REQUESTS

inv3 : buffer $\cap$ inprocess $= \varnothing$

inv4 : comp $\subseteq$ COMP

inv5 : master $\in$ comp

inv6 : processed $\subseteq$ REQUESTS

inv7 : inprocess $\cap$ processed $= \varnothing$

inv8 : buffer $\cap$ processed $= \varnothing$

EVENTS

Initialisation

begin

act1 : buffer $:= \varnothing$

act2 : inprocess $:= \varnothing$

act3 : comp := init_comp

act4 : master := init_master

act5 : processed $:= \varnothing$

end

Event *RequestIn* $\widehat{=}$

any

r

where

grd1 : r $\in$ REQUESTS

grd2 : r $\notin$ buffer

grd3 : r $\notin$ inprocess

grd4 : r ∉ processed

then

act1 : buffer := buffer ∪ {r}

end

Event *RequestOut* ≙

any

r

where

grd1 : r ∈ inprocess

then

act1 : inprocess := inprocess \ {r}

act2 : processed := processed ∪ {r}

end

Event *Process* ≙

any

r

where

grd1 : r ∈ buffer

then

act1 : buffer := buffer \ {r}

act2 : inprocess := inprocess ∪ {r}

end

Event *ChangeMaster* ≙

any

new_master

n_buffer

n_inprocess

n_processed

where

grd2 : new_master ∈ comp

grd3 : new_master ≠ master

grd4 : n_buffer ∪ n_inprocess ∪ n_processed ⊆
buffer ∪ inprocess ∪ processed

grd7 : n_inprocess ∩ n_buffer = ∅

grd8 : n_inprocess ∩ n_processed = ∅

grd9 : n_processed ∩ n_buffer = ∅

then

act1 : master := new_master

act2 : buffer := n_buffer

act3 : inprocess := n_inprocess

act4 : processed := n_processed

act5 : comp := comp \ {master}

end

Event *CompActivation* ≙

any

c

where

grd1 : $c \in COMP$

grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$

end

Event *CompDeactivation* $\widehat{=}$

any

c

where

grd1 : $c \in comp$

grd2 : $c \neq master$

then

act1 : $comp := comp \setminus \{c\}$

end

END

CONTEXT replication_cnt

SETS

REQUESTS

COMP

CONSTANTS

init_comp

init_master

AXIOMS

axm1 : finite(REQUESTS)

axm2 : finite(COMP)

axm3 : COMP $\neq \varnothing$

axm4 : init_comp $\subseteq$ COMP

axm5 : init_comp $\neq \varnothing$

axm6 : init_master $\in$ init_comp

END

MACHINE replication1_r1

REFINES replication1_m0

SEES replication_cnt

VARIABLES

comp

comp_buffer

master

comp_inprocess

comp_processed

failed

in_transit

INVARIANTS

inv1 : $comp_buffer \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv2 : $comp_buffer(master) = buffer$

inv3 : $comp_inprocess \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv4 : $comp_inprocess(master) = inprocess$

inv5 : $comp_processed \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv6 : $comp_processed(master) = processed$

inv7 : $\forall c \cdot c \in comp \Rightarrow (comp_buffer(c) \cap comp_inprocess(c) = \varnothing)$

inv8 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_buffer(c) = \varnothing)$

inv9 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_inprocess(c) = \varnothing)$

inv10 : $failed \subseteq comp$

inv11 : $in_transit \in BOOL$

inv12 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master)$

inv13 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_processed(master))$

EVENTS

Initialisation

begin

act1 : $comp_buffer := init_comp \times \{\varnothing\}$

act2 : $comp_inprocess := init_comp \times \{\varnothing\}$

act3 : $comp := init_comp$

act4 : $master := init_master$

act5 : $comp_processed := init_comp \times \{\varnothing\}$

act6 : $failed := \varnothing$

act7 : $in_transit := FALSE$

end

Event *RequestInMst* $\widehat{=}$

refines *RequestIn*

any
r
where
grd1 : $r \in REQUESTS$
grd2 : $r \notin comp_buffer(master)$
grd3 : $r \notin comp_inprocess(master)$
grd4 : $r \notin comp_processed(master)$
grd5 : $master \notin failed$
then
act1 : $comp_buffer(master) := comp_buffer(master) \cup \{r\}$
end

Event *ProcessMst* $\widehat{=}$
refines *Process*
any
r
where
grd1 : $r \in comp_buffer(master)$
grd2 : $master \notin failed$
then
act1 : $comp_buffer(master) := comp_buffer(master) \setminus \{r\}$
act2 : $comp_inprocess(master) := comp_inprocess(master) \cup \{r\}$
end

Event *RequestOutMst* $\widehat{=}$
refines *RequestOut*
any
r
where
grd1 : $r \in comp_inprocess(master)$
grd2 : $master \notin failed$
then
act1 : $comp_inprocess(master) := comp_inprocess(master) \setminus \{r\}$
act2 : $comp_processed(master) := comp_processed(master) \cup \{r\}$
end

Event *RequestInStb* $\widehat{=}$
any
c
r
where
grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : $r \in comp_processed(master)$
grd4 : $r \notin comp_buffer(c)$
grd5 : $r \notin comp_inprocess(c)$
grd6 : $r \notin comp_processed(c)$

grd7 : $c \notin failed$
grd8 : $master \notin failed$

then

act1 : $comp_buffer(c) := comp_buffer(c) \cup \{r\}$

end

Event *ProcessStb* $\widehat{=}$

any

c
r

where

grd2 : $c \neq master$
grd1 : $c \in comp$
grd3 : $r \in comp_buffer(c)$
grd4 : $c \notin failed$

then

act1 : $comp_inprocess(c) := comp_inprocess(c) \cup \{r\}$
act2 : $comp_buffer(c) := comp_buffer(c) \setminus \{r\}$

end

Event *RequestOutStb* $\widehat{=}$

any

c
r

where

grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : $r \in comp_inprocess(c)$
grd4 : $c \notin failed$

then

act1 : $comp_inprocess(c) := comp_inprocess(c) \setminus \{r\}$
act2 : $comp_processed(c) := comp_processed(c) \cup \{r\}$

end

Event *ChangeMaster* $\widehat{=}$

refines *ChangeMaster*

any

new_master

where

grd2 : $new_master \in comp$
grd3 : $new_master \neq master$
grd5 : $master \in failed$
grd6 : $new_master \notin failed$
grd7 : $\forall c \cdot c \in comp \wedge c \neq new_master \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(new_master) \cup comp_inprocess(new_master) \cup comp_processed(new_master)$

with

n_buffer : n_buffer = comp_buffer(new_master)
n_inprocess : n_inprocess = comp_inprocess(new_master)
n_processed : n_processed = comp_processed(new_master)

then

act1 : master := new_master
act2 : comp_buffer := {master} ⩤ comp_buffer
act3 : comp_inprocess := {master} ⩤ comp_inprocess
act4 : comp_processed := {master} ⩤ comp_processed
act5 : comp := comp ∖ {master}
act6 : failed := failed ∖ {master}
act7 : in_transit := TRUE

end

Event *CompActivation* ≙

extends *CompActivation*

any

c

where

grd1 : $c \in COMP$
grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$
act2 : comp_buffer(c) := ∅
act3 : comp_inprocess(c) := ∅
act4 : comp_processed(c) := ∅

end

Event *CompDeactivation* ≙

extends *CompDeactivation*

any

c

where

grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : c ∈ failed

then

act1 : $comp := comp \setminus \{c\}$
act2 : comp_buffer := {c} ⩤ comp_buffer
act3 : comp_inprocess := {c} ⩤ comp_inprocess
act4 : comp_processed := {c} ⩤ comp_processed
act5 : failed := failed ∖ {c}

end

Event *CompFailure* ≙

any

c

where

grd1 : $c \in comp$
grd2 : $c \notin failed$

then

act1 : $failed := failed \cup \{c\}$

end

Event *CompStbRecovery* $\widehat{=}$

any

c

where

grd1 : $c \in failed$
grd2 : $c \neq master$

then

act1 : $failed := failed \setminus \{c\}$

end

Event *TransitionOver* $\widehat{=}$

when

grd1 : $in_transit = TRUE$
grd2 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \subseteq comp_processed(master)$
grd3 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_inprocess(c) \subseteq comp_processed(master)$
grd4 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_processed(c) \subseteq comp_processed(master)$

then

act1 : $in_transit := FALSE$

end

END

MACHINE replication1_r2

REFINES replication1_r1

SEES replication_cnt

VARIABLES

comp

master

failed

log

index_written

index_inprocess

index_processed

n

in_transit

INVARIANTS

inv1 : $n \in \mathbb{N}$

inv2 : $log \in comp \rightarrow (1 \mathinner{..} n \nrightarrow REQUESTS)$

inv3 : $index_written \in comp \rightarrow \mathbb{N}$

inv4 : $index_inprocess \in comp \rightarrow \mathbb{N}$

inv5 : $index_processed \in comp \rightarrow \mathbb{N}$

inv6 : $index_written(master) = n$

inv7 : $log(master) \in 1 \mathinner{..} n \rightarrowtail REQUESTS$

inv8 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_inprocess(c)+1 \mathinner{..} index_written(c)] = comp_buffer(c)$

inv9 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_processed(c)+1 \mathinner{..} index_inprocess(c)] = comp_inprocess(c)$

inv10 : $\forall c \cdot c \in comp \Rightarrow log(c)[1 \mathinner{..} index_processed(c)] = comp_processed(c)$

inv11 : $index_inprocess(master) \leq n$

inv12 : $index_processed(master) \leq index_inprocess(master)$

inv13 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_inprocess(c) \leq index_written(c)$

inv14 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_processed(c) \leq index_inprocess(c)$

inv15 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c) \in 1 \mathinner{..} index_written(c) \rightarrowtail REQUESTS$

inv16 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c) \subseteq log(master)$

inv17 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c)[index_written(c) + 1 \mathinner{..} index_processed(master)] = \varnothing$

inv18 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_written(c) \leq index_written(master)$

inv19 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c)[1 \mathinner{..} index_written(c)] \subseteq log(master)[1 \mathinner{..} index_processed(master)])$

inv20 : $\forall c, i \cdot c \in comp \wedge i \in 1 \mathinner{..} index_written(c) \Rightarrow log(c)(i) = log(master)(i)$

inv21 : $\forall c1, c2, i \cdot c1 \in comp \land c2 \in comp \land$
$i \in 1 \mathinner{..} index_written(c1) \land i \in 1 \mathinner{..} index_written(c2) \Rightarrow$
$log(c1)(i) = log(c2)(i)$

EVENTS

Initialisation

begin

act3 : $comp := init_comp$
act4 : $master := init_master$
act6 : $failed := \varnothing$
act7 : $log := init_comp \times \{\varnothing\}$
act8 : $index_written := init_comp \times \{0\}$
act9 : $index_inprocess := init_comp \times \{0\}$
act10 : $index_processed := init_comp \times \{0\}$
act11 : $n := 0$
act12 : $in_transit := FALSE$

end

Event *RequestInMst* $\widehat{=}$

refines *RequestInMst*

any

r

where

grd1 : $r \in REQUESTS$
grd2 : $r \notin log(master)[1 \mathinner{..} n]$
grd3 : $master \notin failed$

then

act1 : $log(master) := log(master) \cup \{n + 1 \mapsto r\}$
act2 : $n := n + 1$
act3 : $index_written(master) := index_written(master) + 1$

end

Event *ProcessMst* $\widehat{=}$

refines *ProcessMst*

where

grd1 : $index_inprocess(master) \neq index_written(master)$
grd2 : $master \notin failed$

with

r : $r = log(master)(index_inprocess(master) + 1)$

then

act1 : $index_inprocess(master) := index_inprocess(master) + 1$

end

Event *RequestOutMst* $\widehat{=}$

refines *RequestOutMst*

where

grd1 : $index_processed(master) \neq index_inprocess(master)$

grd2 : master $\notin$ failed

with

r : r = log(master)(index_processed(master) + 1)

then

act2 : index_processed(master) := index_processed(master) + 1

end

Event *RequestInStb* $\widehat{=}$

refines *RequestInStb*

any

c

where

grd1 : c $\in$ comp

grd2 : c $\neq$ master

grd3 : c $\notin$ failed

grd4 : master $\notin$ failed

grd5 : index_written(c) $<$ index_processed(master)

with

r : r = log(master)(index_written(c) + 1)

then

act1 : log(c) := log(c) $\cup$
{index_written(c) + 1 $\mapsto$ log(master)(index_written(c) + 1)}

act2 : index_written(c) := index_written(c) + 1

end

Event *ProcessStb* $\widehat{=}$

refines *ProcessStb*

any

c

where

grd2 : c $\neq$ master

grd1 : c $\in$ comp

grd3 : c $\notin$ failed

grd4 : index_inprocess(c) $\neq$ index_written(c)

with

r : r = log(c)(index_inprocess(c) + 1)

then

act1 : index_inprocess(c) := index_inprocess(c) + 1

end

Event *RequestOutStb* $\widehat{=}$

refines *RequestOutStb*

any

c

where

grd1 : c $\in$ comp

grd2 : $c \neq master$
grd3 : $c \notin failed$
grd4 : $index_processed(c) \neq index_inprocess(c)$

with

r : $r = log(c)(index_processed(c) + 1)$

then

act1 : $index_processed(c) := index_processed(c) + 1$

end

Event *ChangeMaster* $\widehat{=}$

refines *ChangeMaster*

any

new_master

where

grd2 : $new_master \in comp$
grd3 : $new_master \neq master$
grd5 : $master \in failed$
grd6 : $new_master \notin failed$
grd7 : $\forall c \cdot c \in comp \land c \neq master \land c \neq new_master \Rightarrow index_written(c) \leq index_written(new_master)$

then

act1 : $master := new_master$
act5 : $comp := comp \setminus \{master\}$
act6 : $failed := failed \setminus \{master\}$
act7 : $log := \{master\} \lhd log$
act8 : $n := index_written(new_master)$
act9 : $index_written := \{master\} \lhd index_written$
act10 : $index_inprocess := \{master\} \lhd index_inprocess$
act11 : $index_processed := \{master\} \lhd index_processed$
act12 : $in_transit := TRUE$

end

Event *CompActivation* $\widehat{=}$

refines *CompActivation*

any

c

where

grd1 : $c \in COMP$
grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$
act2 : $log(c) := \varnothing$
act3 : $index_written(c) := 0$
act4 : $index_inprocess(c) := 0$
act5 : $index_processed(c) := 0$

end

Event *CompDeactivation* $\widehat{=}$

refines *CompDeactivation*

any

c

where

grd1 : $c \in comp$

grd2 : $c \neq master$

grd3 : $c \in failed$

then

act1 : $comp := comp \setminus \{c\}$

act2 : $log := \{c\} \lhd log$

act3 : $index_written := \{c\} \lhd index_written$

act4 : $index_inprocess := \{c\} \lhd index_inprocess$

act5 : $index_processed := \{c\} \lhd index_processed$

act6 : $failed := failed \setminus \{c\}$

end

Event *CompFailure* $\widehat{=}$

extends *CompFailure*

any

c

where

grd1 : $c \in comp$

grd2 : $c \notin failed$

then

act1 : $failed := failed \cup \{c\}$

end

Event *CompStbRecovery* $\widehat{=}$

extends *CompStbRecovery*

any

c

where

grd1 : $c \in failed$

grd2 : $c \neq master$

then

act1 : $failed := failed \setminus \{c\}$

end

Event *TransitionOver* $\widehat{=}$

refines *TransitionOver*

when

grd1 : $in_transit = TRUE$

grd2 : $\forall c \cdot c \in comp \land c \neq master \Rightarrow$
$log(c)[1..index_written(c)] \subseteq log(master)[1..index_processed(master)]$

then

```
        act1 : in_transit := FALSE
    end
END
```

2. Cascade Master-Standby Model

MACHINE replication2_m0

SEES replication_cnt

VARIABLES

comp

master

ups_standby

m_buffer

m_inprocess

m_processed

ups_buffer

ups_inprocess

ups_processed

INVARIANTS

inv1 : m_buffer $\subseteq$ REQUESTS

inv2 : m_inprocess $\subseteq$ REQUESTS

inv3 : m_buffer $\cap$ m_inprocess $= \varnothing$

inv4 : comp $\subseteq$ COMP

inv5 : master $\in$ comp

inv6 : m_processed $\subseteq$ REQUESTS

inv7 : m_inprocess $\cap$ m_processed $= \varnothing$

inv8 : m_buffer $\cap$ m_processed $= \varnothing$

inv9 : ups_standby $\in$ comp

inv10 : ups_standby $\neq$ master

inv11 : ups_buffer $\subseteq$ REQUESTS

inv12 : ups_inprocess $\subseteq$ REQUESTS

inv13 : ups_processed $\subseteq$ REQUESTS

inv14 : ups_buffer $\cap$ ups_inprocess $= \varnothing$

inv15 : ups_inprocess $\cap$ ups_processed $= \varnothing$

inv16 : ups_buffer $\cap$ ups_processed $= \varnothing$

inv17 : ups_buffer $\cup$ ups_inprocess $\cup$ ups_processed $\subseteq$
m_buffer $\cup$ m_inprocess $\cup$ m_processed

EVENTS

Initialisation

begin

act1 : m_buffer $:= \varnothing$

act2 : m_inprocess $:= \varnothing$

act3 : comp $:=$ init_comp

act4 : master $:=$ init_master

act5 : m_processed := ∅
act6 : ups_standby := init_ups_standby
act7 : ups_buffer := ∅
act8 : ups_inprocess := ∅
act9 : ups_processed := ∅

end

Event *RequestIn* ≙

any

r

where

grd1 : r ∈ REQUESTS
grd2 : r ∉ m_buffer
grd3 : r ∉ m_inprocess
grd4 : r ∉ m_processed

then

act1 : m_buffer := m_buffer ∪ {r}

end

Event *ProcessMst* ≙

any

r

where

grd1 : r ∈ m_buffer

then

act1 : m_buffer := m_buffer \ {r}
act2 : m_inprocess := m_inprocess ∪ {r}

end

Event *RequestOutMst* ≙

any

r

where

grd1 : r ∈ m_inprocess
grd2 : r ∈ ups_buffer ∪ ups_inprocess ∪ ups_processed

then

act1 : m_inprocess := m_inprocess \ {r}
act2 : m_processed := m_processed ∪ {r}

end

Event *ChangeMaster* ≙

any

new_ups_standby
n_processed
n_inprocess
n_buffer

where

grd1 : new_ups_standby ∈ comp

grd3 : new_ups_standby $\neq$ ups_standby
grd2 : new_ups_standby $\neq$ master
grd4 : n_buffer $\cup$ n_inprocess $\cup$ n_processed $\subseteq$
ups_buffer $\cup$ ups_inprocess $\cup$ ups_processed
grd5 : n_inprocess $\cap$ n_buffer $= \varnothing$
grd6 : n_inprocess $\cap$ n_processed $= \varnothing$
grd7 : n_processed $\cap$ n_buffer $= \varnothing$

then

act1 : master := ups_standby
act2 : ups_standby := new_ups_standby
act3 : m_buffer := ups_buffer
act4 : m_inprocess := ups_inprocess
act5 : m_processed := ups_processed
act6 : comp := comp $\setminus$ {master}
act7 : ups_buffer := n_buffer
act8 : ups_inprocess := n_inprocess
act9 : ups_processed := n_processed

end

Event *CompActivation* $\widehat{=}$

any

c

where

grd1 : c $\in$ COMP
grd2 : c $\notin$ comp

then

act1 : comp := comp $\cup$ {c}

end

Event *CompDeactivation* $\widehat{=}$

any

c

where

grd1 : c $\in$ comp
grd2 : c $\neq$ master
grd3 : c $\neq$ ups_standby

then

act1 : comp := comp $\setminus$ {c}

end

Event *ChangeUpsStandby* $\widehat{=}$

any

c
new_ups_standby
n_buffer
n_inprocess
n_processed

where

grd1 : $c = ups_standby$
grd2 : $new_ups_standby \in comp$
grd3 : $new_ups_standby \neq master$
grd4 : $new_ups_standby \neq ups_standby$
grd5 : $n_buffer \cup n_inprocess \cup n_processed \subseteq ups_buffer \cup ups_inprocess \cup ups_processed$
grd6 : $n_inprocess \cap n_buffer = \varnothing$
grd7 : $n_inprocess \cap n_processed = \varnothing$
grd8 : $n_processed \cap n_buffer = \varnothing$

then

act1 : $ups_standby := new_ups_standby$
act2 : $ups_buffer := n_buffer$
act3 : $ups_inprocess := n_inprocess$
act4 : $ups_processed := n_processed$
act5 : $comp := comp \setminus \{c\}$

end

Event *RequestInUps* $\widehat{=}$

any

r

where

grd1 : $r \in m_buffer \cup m_inprocess \cup m_processed$
grd2 : $r \notin ups_buffer$
grd3 : $r \notin ups_inprocess$
grd4 : $r \notin ups_processed$

then

act1 : $ups_buffer := ups_buffer \cup \{r\}$

end

Event *ProcessUps* $\widehat{=}$

any

r

where

grd1 : $r \in ups_buffer$

then

act1 : $ups_buffer := ups_buffer \setminus \{r\}$
act2 : $ups_inprocess := ups_inprocess \cup \{r\}$

end

Event *RequestOutUps* $\widehat{=}$

any

r

where

grd1 : $r \in ups_inprocess$

then

act1 : $ups_inprocess := ups_inprocess \setminus \{r\}$

```
        act2 : ups_processed := ups_processed ∪ {r}
    end
END
```

CONTEXT replication_cnt

SETS

REQUESTS

COMP

CONSTANTS

init_comp

init_master

init_ups_standby

AXIOMS

axm1 : finite(REQUESTS)

axm2 : finite(COMP)

axm3 : COMP $\neq \varnothing$

axm4 : init_comp $\subseteq$ COMP

axm5 : init_comp $\neq \varnothing$

axm6 : init_master $\in$ init_comp

axm7 : init_ups_standby $\in$ init_comp

axm8 : init_ups_standby $\neq$ init_master

END

MACHINE replication2_r1

REFINES replication2_m0

SEES replication_cnt

VARIABLES

comp_buffer

comp

master

comp_inprocess

comp_processed

ups_standby

failed

in_transit

INVARIANTS

inv1 : $comp_buffer \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv2 : $comp_buffer(master) = m_buffer$

inv3 : $comp_buffer(ups_standby) = ups_buffer$

inv4 : $comp_inprocess \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv5 : $comp_inprocess(master) = m_inprocess$

inv6 : $comp_inprocess(ups_standby) = ups_inprocess$

inv7 : $comp_processed \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv8 : $comp_processed(master) = m_processed$

inv9 : $comp_processed(ups_standby) = ups_processed$

inv10 : $\forall c \cdot c \in comp \Rightarrow (comp_buffer(c) \cap comp_inprocess(c) = \varnothing)$

inv11 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_buffer(c) = \varnothing)$

inv12 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_inprocess(c) = \varnothing)$

inv13 : $failed \subseteq comp$

inv14 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master)$

inv15 : $in_transit \in BOOL$

inv16 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \Rightarrow (comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_processed(ups_standby))$

inv17 : $in_transit = FALSE \Rightarrow comp_processed(master) \subseteq comp_buffer(ups_standby) \cup comp_inprocess(ups_standby) \cup comp_processed(ups_standby)$

inv18 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(ups_standby) \cup comp_inprocess(ups_standby) \cup comp_processed(ups_standby)$

EVENTS

Initialisation

begin

act1 : comp_buffer := init_comp × {∅}
act2 : comp_inprocess := init_comp × {∅}
act3 : comp_processed := init_comp × {∅}
act4 : comp := init_comp
act5 : master := init_master
act6 : ups_standby := init_ups_standby
act7 : failed := ∅
act9 : in_transit := FALSE

end

Event *RequestInMst* ≙

refines *RequestIn*

any

r

where

grd1 : r ∈ REQUESTS
grd2 : r ∉ comp_buffer(master)
grd3 : r ∉ comp_inprocess(master)
grd4 : r ∉ comp_processed(master)
grd5 : master ∉ failed

then

act1 : comp_buffer(master) := comp_buffer(master) ∪ {r}

end

Event *ProcessMst* ≙

refines *ProcessMst*

any

r

where

grd1 : r ∈ comp_buffer(master)
grd2 : master ∉ failed

then

act1 : comp_inprocess(master) := comp_inprocess(master) ∪ {r}
act2 : comp_buffer(master) := comp_buffer(master) \ {r}

end

Event *RequestOutMst* ≙

refines *RequestOutMst*

any

r

where

grd1 : r ∈ comp_inprocess(master)
grd2 : master ∉ failed
grd3 : r ∈ comp_buffer(ups_standby)∪comp_inprocess(ups_standby)∪
comp_processed(ups_standby)

then

act1 : comp_inprocess(master) := comp_inprocess(master) \ {r}

act2 : comp_processed(master) := comp_processed(master) ∪ {r}

end

Event *ChangeMaster* ≙

refines *ChangeMaster*

any

new_ups_standby

where

grd2 : new_ups_standby ∈ comp

grd3 : new_ups_standby ≠ ups_standby

grd4 : new_ups_standby ≠ master

grd6 : ups_standby ∉ failed

grd7 : new_ups_standby ∉ failed

grd8 : ∀c·c ∈ comp ∧ c ≠ ups_standby ∧ c ≠ new_ups_standby ∧ c ≠ master⇒comp_buffer(c)∪comp_inprocess(c)∪comp_processed(c) ⊆ comp_buffer(new_ups_standby)∪comp_inprocess(new_ups_standby)∪ comp_processed(new_ups_standby)

with

n_buffer : n_buffer = comp_buffer(new_ups_standby)

n_inprocess : n_inprocess = comp_inprocess(new_ups_standby)

n_processed : n_processed = comp_processed(new_ups_standby)

then

act1 : master := ups_standby

act2 : ups_standby := new_ups_standby

act3 : comp_buffer := {master} ⩤ comp_buffer

act4 : comp_inprocess := {master} ⩤ comp_inprocess

act5 : comp_processed := {master} ⩤ comp_processed

act6 : comp := comp \ {master}

act7 : failed := failed \ {master}

act9 : in_transit := TRUE

end

Event *CompActivation* ≙

extends *CompActivation*

any

c

where

grd1 : $c \in COMP$

grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$

act2 : comp_buffer(c) := ∅

act3 : comp_inprocess(c) := ∅

act4 : comp_processed(c) := ∅

end

Event *CompDeactivation* ≙

extends *CompDeactivation*

any

c

where

grd1 : $c \in comp$

grd2 : $c \neq master$

grd3 : $c \neq ups_standby$

grd4 : $\mathtt{c} \in \mathtt{failed}$

then

act1 : $comp := comp \setminus \{c\}$

act2 : $\mathtt{comp_buffer} := \{\mathtt{c}\} \lhd \mathtt{comp_buffer}$

act3 : $\mathtt{comp_inprocess} := \{\mathtt{c}\} \lhd \mathtt{comp_inprocess}$

act4 : $\mathtt{comp_processed} := \{\mathtt{c}\} \lhd \mathtt{comp_processed}$

act5 : $\mathtt{failed} := \mathtt{failed} \setminus \{\mathtt{c}\}$

end

Event *CompFailure* $\widehat{=}$

any

c

where

grd1 : $\mathtt{c} \in \mathtt{comp}$

grd2 : $\mathtt{c} \notin \mathtt{failed}$

then

act1 : $\mathtt{failed} := \mathtt{failed} \cup \{\mathtt{c}\}$

end

Event *ChangeUpsStb* $\widehat{=}$

refines *ChangeUpsStandby*

any

c

new_ups_standby

where

grd1 : $\mathtt{c} = \mathtt{ups_standby}$

grd2 : $\mathtt{new_ups_standby} \in \mathtt{comp}$

grd3 : $\mathtt{new_ups_standby} \neq \mathtt{master}$

grd4 : $\mathtt{new_ups_standby} \neq \mathtt{ups_standby}$

grd6 : $\mathtt{ups_standby} \notin \mathtt{failed}$

grd7 : $\mathtt{new_ups_standby} \notin \mathtt{failed}$

grd8 : $\forall \mathtt{c} \cdot \mathtt{c} \in \mathtt{comp} \land \mathtt{c} \neq \mathtt{master} \land \mathtt{c} \neq \mathtt{new_ups_standby} \land \mathtt{c} \neq \mathtt{ups_standby} \Rightarrow \mathtt{comp_buffer}(\mathtt{c}) \cup \mathtt{comp_inprocess}(\mathtt{c}) \cup \mathtt{comp_processed}(\mathtt{c}) \subseteq \mathtt{comp_buffer}(\mathtt{new_ups_standby}) \cup \mathtt{comp_inprocess}(\mathtt{new_ups_standby}) \cup \mathtt{comp_processed}(\mathtt{new_ups_standby})$

with

n_buffer : $\mathtt{n_buffer} = \mathtt{comp_buffer}(\mathtt{new_ups_standby})$

n_inprocess : $\mathtt{n_inprocess} = \mathtt{comp_inprocess}(\mathtt{new_ups_standby})$

n_processed : $\mathtt{n_processed} = \mathtt{comp_processed}(\mathtt{new_ups_standby})$

then

act1 : ups_standby := new_ups_standby
act2 : comp_buffer := {c} ⩤ comp_buffer
act3 : comp_inprocess := {c} ⩤ comp_inprocess
act4 : comp_processed := {c} ⩤ comp_processed
act5 : comp := comp \ {c}
act6 : failed := failed \ {c}
act7 : in_transit2 := TRUE

end

Event *CompStbRecovery* ≙

any

c

where

grd1 : c ∈ failed
grd2 : c ≠ master
grd3 : c ≠ ups_standby

then

act1 : failed := failed \ {c}

end

Event *RequestInUps* ≙

refines *RequestInUps*

any

r

where

grd1 : r ∈ comp_buffer(master)∪comp_inprocess(master)∪comp_processed(master)
grd2 : r ∉ comp_buffer(ups_standby)
grd3 : r ∉ comp_inprocess(ups_standby)
grd4 : r ∉ comp_processed(ups_standby)
grd5 : ups_standby ∉ failed
grd6 : master ∉ failed

then

act1 : comp_buffer(ups_standby) := comp_buffer(ups_standby) ∪ {r}

end

Event *ProcessUps* ≙

refines *ProcessUps*

any

r

where

grd1 : r ∈ comp_buffer(ups_standby)
grd2 : ups_standby ∉ failed

then

act1 : comp_buffer(ups_standby) := comp_buffer(ups_standby)\{r}

act2 : comp_inprocess(ups_standby) := comp_inprocess(ups_standby)∪ {r}

end

Event *RequestOutUps* ≙

refines *RequestOutUps*

any

r

where

grd1 : r ∈ comp_inprocess(ups_standby)

grd2 : ups_standby ∉ failed

then

act1 : comp_inprocess(ups_standby) := comp_inprocess(ups_standby)\ {r}

act2 : comp_processed(ups_standby) := comp_processed(ups_standby)∪ {r}

end

Event *RequestInStb* ≙

any

c

r

where

grd1 : c ∈ comp

grd2 : c ≠ ups_standby

grd3 : c ≠ master

grd4 : c ∉ failed

grd6 : r ∈ comp_processed(ups_standby)

grd7 : r ∉ comp_buffer(c)

grd8 : r ∉ comp_inprocess(c)

grd9 : r ∉ comp_processed(c)

then

act1 : comp_buffer(c) := comp_buffer(c) ∪ {r}

end

Event *ProcessStb* ≙

any

c

r

where

grd1 : c ∈ comp

grd2 : c ≠ ups_standby

grd3 : c ≠ master

grd4 : r ∈ comp_buffer(c)

grd5 : c ∉ failed

then

act1 : comp_buffer(c) := comp_buffer(c) \ {r}

act2 : comp_inprocess(c) := comp_inprocess(c) ∪ {r}

end

Event *RequestOutStb* $\widehat{=}$

any

c

r

where

grd1 : c ∈ comp

grd2 : c ≠ ups_standby

grd3 : c ≠ master

grd4 : r ∈ comp_inprocess(c)

grd5 : c ∉ failed

then

act1 : comp_inprocess(c) := comp_inprocess(c) \ {r}

act2 : comp_processed(c) := comp_processed(c) ∪ {r}

end

Event *TransitionOver* $\widehat{=}$

when

grd1 : in_transit = TRUE

grd2 : comp_processed(master) ⊆ comp_buffer(ups_standby) ∪
comp_inprocess(ups_standby) ∪ comp_processed(ups_standby)

grd3 : (∀c·c ∈ comp∧c ≠ master∧c ≠ ups_standby⇒(comp_buffer(c)∪
comp_inprocess(c)∪comp_processed(c) ⊆ comp_processed(ups_standby))

then

act1 : in_transit := FALSE

end

END

MACHINE replication2_r2

REFINES replication2_r1

SEES replication_cnt

VARIABLES

comp

master

ups_standby

log

index_inprocess

index_processed

index_written

failed

n

k

in_transit

INVARIANTS

inv1 : $n \in \mathbb{N}$

inv2 : $k \in \mathbb{N}$

inv3 : $log \in comp \rightarrow (1 \mathinner{..} n \nrightarrow REQUESTS)$

inv4 : $index_written \in comp \rightarrow \mathbb{N}$

inv5 : $index_inprocess \in comp \rightarrow \mathbb{N}$

inv6 : $index_processed \in comp \rightarrow \mathbb{N}$

inv7 : $index_written(master) = n$

inv8 : $log(master) \in 1 \mathinner{..} n \rightarrowtail REQUESTS$

inv9 : $index_written(ups_standby) = k$

inv10 : $log(ups_standby) \in 1 \mathinner{..} k \rightarrowtail REQUESTS$

inv11 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_inprocess(c)+1 \mathinner{..} index_written(c)] = comp_buffer(c)$

inv12 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_processed(c)+1 \mathinner{..} index_inprocess(c)] = comp_inprocess(c)$

inv13 : $\forall c \cdot c \in comp \Rightarrow log(c)[1 \mathinner{..} index_processed(c)] = comp_processed(c)$

inv14 : $index_inprocess(master) \leq n$

inv15 : $index_processed(master) \leq index_inprocess(master)$

inv16 : $index_inprocess(ups_standby) \leq k$

inv17 : $index_processed(ups_standby) \leq index_inprocess(ups_standby)$

inv18 : $\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \Rightarrow index_inprocess(c) \leq index_written(c)$

inv19 : $\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \Rightarrow index_processed(c) \leq index_inprocess(c)$

inv20 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow log(c) \in 1 \,..\, index_written(c) \rightarrowtail REQUESTS$

inv21 : $\forall c \cdot c \in comp \land c \neq master \Rightarrow log(c) \subseteq log(master)$

inv22 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow log(c)[index_written(c) + 1 \,..\, index_written(ups_standby)] = \varnothing$

inv23 : $\forall c \cdot c \in comp \land c \neq master \Rightarrow index_written(c) \leq index_written(master)$

inv24 : $log(ups_standby)[1 \,..\, index_written(ups_standby)] \subseteq log(master)[1 \,..\, index_written(master)]$

inv25 : $in_transit \in BOOL$

inv26 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow log(c)[1..index_written(c)] \subseteq log(ups_standby)[1..index_processed(ups_standby)]$

inv27 : $in_transit = FALSE \Rightarrow log(master)[1..index_processed(master)] \subseteq log(ups_standby)[1 \,..\, index_written(ups_standby)]$

inv28 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow log(c) \subseteq log(ups_standby)$

inv29 : $\forall c1, c2 \cdot c1 \in comp \land c2 \in comp \land c1 \neq master \land c2 \neq master \land index_written(c1) \leq index_written(c2) \Rightarrow log(c1)[1..index_written(c1)] \subseteq log(c2)[1 \,..\, index_written(c2)]$

inv30 : $\forall c, i \cdot c \in comp \land i \in 1..index_written(c) \Rightarrow log(c)(i) = log(master)(i)$

inv31 : $\forall c1, c2, i \cdot c1 \in comp \land c2 \in comp \land i \in 1 \,..\, index_written(c1) \land i \in 1 \,..\, index_written(c2) \Rightarrow log(c1)(i) = log(c2)(i)$

inv32 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow index_written(c) \leq index_written(ups_standby)$

inv33 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \Rightarrow log(c) \subseteq log(ups_standby)$

EVENTS

Initialisation

begin

act1 : $comp := init_comp$

act2 : $master := init_master$

act3 : $ups_standby := init_ups_standby$

act4 : $failed := \varnothing$

act5 : $log := init_comp \times \{\varnothing\}$

act6 : $index_written := init_comp \times \{0\}$

act7 : $index_inprocess := init_comp \times \{0\}$

act8 : $index_processed := init_comp \times \{0\}$

act9 : $n := 0$

act10 : $k := 0$

act12 : $in_transit := FALSE$

end

Event *RequestInMst* $\widehat{=}$

refines *RequestInMst*

any

r

where

grd1 : $r \in REQUESTS$

grd2 : $r \notin log(master)[1 \mathinner{..} n]$

grd3 : $master \notin failed$

then

act1 : $log(master) := log(master) \cup \{n + 1 \mapsto r\}$

act2 : $n := n + 1$

act3 : $index_written(master) := index_written(master) + 1$

end

Event *ProcessMst* $\widehat{=}$

refines *ProcessMst*

where

grd1 : $master \notin failed$

grd2 : $index_inprocess(master) \neq index_written(master)$

with

r : $r = log(master)(index_inprocess(master) + 1)$

then

act1 : $index_inprocess(master) := index_inprocess(master) + 1$

end

Event *RequestOutMst* $\widehat{=}$

refines *RequestOutMst*

where

grd1 : $master \notin failed$

grd2 : $index_processed(master) \neq index_inprocess(master)$

grd3 : $index_written(ups_standby) > index_processed(master)$

with

r : $r = log(master)(index_processed(master) + 1)$

then

act1 : $index_processed(master) := index_processed(master) + 1$

end

Event *ChangeMaster* $\widehat{=}$

refines *ChangeMaster*

any

$new_ups_standby$

c

where

grd1 : $c = master$

grd2 : $new_ups_standby \in comp$

grd3 : $new_ups_standby \neq ups_standby$

grd4 : $new_ups_standby \neq master$

grd5 : $ups_standby \notin failed$

grd6 : $new_ups_standby \notin failed$

grd7 : $\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups_standby \wedge c \neq new_ups_standby \Rightarrow$
$index_written(c) \leq index_written(new_ups_standby)$

then

act1 : master := ups_standby
act2 : ups_standby := new_ups_standby
act3 : comp := comp ∖ {c}
act4 : failed := failed ∖ {c}
act6 : log := {c} ⩤ log
act7 : n := index_written(ups_standby)
act8 : index_written := {c} ⩤ index_written
act9 : index_inprocess := {c} ⩤ index_inprocess
act10 : index_processed := {c} ⩤ index_processed
act11 : k := index_written(new_ups_standby)
act13 : in_transit := TRUE

end

Event *CompActivation* $\widehat{=}$

refines *CompActivation*

any

c

where

grd1 : c ∈ COMP
grd2 : c ∉ comp

then

act1 : comp := comp ∪ {c}
act2 : log(c) := ∅
act3 : index_written(c) := 0
act4 : index_inprocess(c) := 0
act5 : index_processed(c) := 0

end

Event *CompDeactivation* $\widehat{=}$

refines *CompDeactivation*

any

c

where

grd1 : c ∈ comp
grd2 : c ≠ master
grd3 : c ≠ ups_standby
grd4 : c ∈ failed

then

act1 : comp := comp ∖ {c}
act2 : failed := failed ∖ {c}
act3 : log := {c} ⩤ log
act5 : index_inprocess := {c} ⩤ index_inprocess
act6 : index_processed := {c} ⩤ index_processed
act4 : index_written := {c} ⩤ index_written

end

Event *CompFailure* $\widehat{=}$
extends *CompFailure*
any
c
where
grd1 : $c \in comp$
grd2 : $c \notin failed$
then
act1 : $failed := failed \cup \{c\}$
end
Event *ChangeUpsStb* $\widehat{=}$
refines *ChangeUpsStb*
any
c
new_ups_standby
where
grd1 : $c = ups_standby$
grd2 : $new_ups_standby \in comp$
grd3 : $new_ups_standby \neq master$
grd4 : $new_ups_standby \neq ups_standby$
grd6 : $ups_standby \notin failed$
grd7 : $new_ups_standby \notin failed$
grd8 : $\forall c \cdot c \in comp \land c \neq master \land c \neq ups_standby \land c \neq new_ups_standby \Rightarrow index_written(c) \leq index_written(new_ups_standby)$
then
act1 : $ups_standby := new_ups_standby$
act2 : $comp := comp \setminus \{c\}$
act3 : $failed := failed \setminus \{c\}$
act4 : $log := \{c\} \lhd log$
act5 : $index_written := \{c\} \lhd index_written$
act6 : $index_inprocess := \{c\} \lhd index_inprocess$
act7 : $index_processed := \{c\} \lhd index_processed$
act8 : $k := index_written(new_ups_standby)$
act9 : $in_transit := TRUE$
end
Event *CompStbRecovery* $\widehat{=}$
extends *CompStbRecovery*
any
c
where
grd1 : $c \in failed$
grd2 : $c \neq master$
grd3 : $c \neq ups_standby$
then

act1 : $failed := failed \setminus \{c\}$

end

Event *RequestInUps* $\widehat{=}$

refines *RequestInUps*

any

r

where

grd1 : ups_standby $\notin$ failed

grd2 : master $\notin$ failed

grd3 : index_written(ups_standby) $<$ index_written(master)

with

r : r = log(master)(index_written(ups_standby) + 1)

then

act1 : log(ups_standby) := log(ups_standby) $\cup$ {index_written(ups_standby)+1 $\mapsto$ log(master)(index_written(ups_standby)+1)}

act2 : index_written(ups_standby) := index_written(ups_standby)+1

act3 : k := k + 1

end

Event *RequestInStb* $\widehat{=}$

refines *RequestInStb*

any

c

where

grd1 : c $\in$ comp

grd2 : c $\neq$ ups_standby

grd3 : c $\neq$ master

grd4 : c $\notin$ failed

grd5 : master $\notin$ failed

grd6 : index_written(c) $<$ index_processed(ups_standby)

with

r : r = log(ups_standby)(index_written(c) + 1)

then

act1 : log(c) := log(c) $\cup$ {index_written(c)+1 $\mapsto$ log(ups_standby)(index_written(c)+1)}

act2 : index_written(c) := index_written(c) + 1

end

Event *ProcessUps* $\widehat{=}$

refines *ProcessUps*

where

grd1 : ups_standby $\notin$ failed

grd2 : index_inprocess(ups_standby) $\neq$ index_written(ups_standby)

with

grd1 : r = log(ups_standby)(index_inprocess(ups_standby) + 1)

then

act1 : index_inprocess(ups_standby) := index_inprocess(ups_standby)+ 1

end

Event *RequestOutUps* $\widehat{=}$

refines *RequestOutUps*

where

grd1 : ups_standby $\notin$ failed

grd2 : index_processed(ups_standby) $\neq$ index_inprocess(ups_standby)

with

r : r = log(ups_standby)(index_processed(ups_standby) + 1)

then

act1 : index_processed(ups_standby) := index_processed(ups_standby)+ 1

end

Event *ProcessStb* $\widehat{=}$

refines *ProcessStb*

any

c

where

grd1 : c $\in$ comp

grd2 : c $\neq$ ups_standby

grd3 : c $\neq$ master

grd4 : c $\notin$ failed

grd5 : index_inprocess(c) $\neq$ index_written(c)

with

r : r = log(c)(index_inprocess(c) + 1)

then

act1 : index_inprocess(c) := index_inprocess(c) + 1

end

Event *RequestOutStb* $\widehat{=}$

refines *RequestOutStb*

any

c

where

grd1 : c $\in$ comp

grd2 : c $\neq$ ups_standby

grd3 : c $\neq$ master

grd4 : c $\notin$ failed

grd5 : index_processed(c) $\neq$ index_inprocess(c)

with

r : r = log(c)(index_processed(c) + 1)

then

act1 : index_processed(c) := index_processed(c) + 1

end

Event *TransitionOver* $\widehat{=}$

refines *TransitionOver*

when

grd1 : in_transit = TRUE

grd2 : log(master)[1..index_processed(master)] ⊆ log(ups_standby)[1..index_written(ups_standby)]

grd3 : (∀c·c ∈ comp∧c ≠ master∧c ≠ ups_standby⇒(comp_buffer(c)∪comp_inprocess(c)∪comp_processed(c) ⊆ comp_processed(ups_standby))

then

act1 : in_transit := FALSE

end

END

3. Synchronous model

MACHINE replication3_m0

SEES replication_cnt

VARIABLES

buffer

inprocess

comp

master

processed

INVARIANTS

inv1 : $buffer \subseteq REQUESTS$

inv2 : $inprocess \subseteq REQUESTS$

inv3 : $buffer \cap inprocess = \varnothing$

inv4 : $comp \subseteq COMP$

inv5 : $master \in comp$

inv6 : $processed \subseteq REQUESTS$

inv7 : $inprocess \cap processed = \varnothing$

inv8 : $buffer \cap processed = \varnothing$

EVENTS

Initialisation

begin

act1 : $buffer := \varnothing$

act2 : $inprocess := \varnothing$

act3 : $comp := init_comp$

act4 : $master := init_master$

act5 : $processed := \varnothing$

end

Event *RequestIn* $\widehat{=}$

any

r

where

grd1 : $r \in REQUESTS$

grd2 : $r \notin buffer$

grd3 : $r \notin inprocess$

grd4 : $r \notin processed$

then

act1 : $buffer := buffer \cup \{r\}$

end

Event *ChangeMaster* $\widehat{=}$

any

n_buffer

n_inprocess
n_processed
new_master

where

grd1 : new_master $\in$ comp
grd2 : new_master $\neq$ master
grd3 : n_buffer $\cup$ n_inprocess $\cup$ n_processed $\subseteq$
buffer $\cup$ inprocess $\cup$ processed
grd4 : n_inprocess $\cap$ n_buffer $= \varnothing$
grd5 : n_inprocess $\cap$ n_processed $= \varnothing$
grd6 : n_processed $\cap$ n_buffer $= \varnothing$

then

act1 : master := new_master
act2 : buffer := n_buffer
act3 : inprocess := n_inprocess
act4 : processed := n_processed
act5 : comp := comp $\setminus$ {master}

end

Event *CompActivation* $\widehat{=}$

any

c

where

grd1 : c $\in$ COMP
grd2 : c $\notin$ comp

then

act1 : comp := comp $\cup$ {c}

end

Event *CompDeactivation* $\widehat{=}$

any

c

where

grd1 : c $\in$ comp
grd2 : c $\neq$ master

then

act1 : comp := comp $\setminus$ {c}

end

Event *RequestOut* $\widehat{=}$

any

r

where

grd1 : r $\in$ inprocess

then

act1 : inprocess := inprocess $\setminus$ {r}
act2 : processed := processed $\cup$ {r}

end

Event *Process* $\widehat{=}$

any

r

where

grd1 : $r \in buffer$

then

act1 : $buffer := buffer \setminus \{r\}$

act2 : $inprocess := inprocess \cup \{r\}$

end

END

CONTEXT replication_cnt

SETS

REQUESTS

COMP

CONSTANTS

init_comp

init_master

init_ups_standby

AXIOMS

axm1 : $finite(REQUESTS)$

axm2 : $finite(COMP)$

axm3 : $COMP \neq \varnothing$

axm4 : $init_comp \subseteq COMP$

axm5 : $init_comp \neq \varnothing$

axm6 : $init_master \in init_comp$

axm7 : $init_ups_standby \in init_comp$

axm8 : $init_ups_standby \neq init_master$

END

MACHINE replication3_r1

REFINES replication3_m0

SEES replication_cnt

VARIABLES

comp

comp_buffer

master

comp_inprocess

comp_processed

failed

in_transit

INVARIANTS

inv1 : $comp_buffer \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv2 : $comp_buffer(master) = buffer$

inv3 : $comp_inprocess \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv4 : $comp_inprocess(master) = inprocess$

inv5 : $comp_processed \in comp \rightarrow \mathbb{P}(REQUESTS)$

inv6 : $comp_processed(master) = processed$

inv7 : $\forall c \cdot c \in comp \Rightarrow (comp_buffer(c) \cap comp_inprocess(c) = \varnothing)$

inv8 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_buffer(c) = \varnothing)$

inv9 : $\forall c \cdot c \in comp \Rightarrow (comp_processed(c) \cap comp_inprocess(c) = \varnothing)$

inv10 : $failed \subseteq comp$

inv11 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master)$

inv12 : $in_transit \in BOOL$

inv13 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_processed(master) \subseteq comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c))$

EVENTS

Initialisation

begin

act1 : $comp_buffer := init_comp \times \{\varnothing\}$

act2 : $comp_inprocess := init_comp \times \{\varnothing\}$

act3 : $comp := init_comp$

act4 : $master := init_master$

act5 : $comp_processed := init_comp \times \{\varnothing\}$

act6 : $failed := \varnothing$

act7 : $in_transit := FALSE$

end

Event *RequestInMst* $\widehat{=}$

refines *RequestIn*

any
r
where
grd1 : $r \in REQUESTS$
grd2 : $r \notin comp_buffer(master)$
grd3 : $r \notin comp_inprocess(master)$
grd4 : $r \notin comp_processed(master)$
grd5 : $master \notin failed$
then
act1 : $comp_buffer(master) := comp_buffer(master) \cup \{r\}$
end

Event *ProcessMst* $\widehat{=}$
refines *Process*
any
r
where
grd1 : $r \in comp_buffer(master)$
grd2 : $master \notin failed$
then
act1 : $comp_buffer(master) := comp_buffer(master) \setminus \{r\}$
act2 : $comp_inprocess(master) := comp_inprocess(master) \cup \{r\}$
end

Event *RequestInStb* $\widehat{=}$
any
c
r
where
grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : $r \in comp_buffer(master) \cup comp_inprocess(master) \cup comp_processed(master)$
grd4 : $r \notin comp_buffer(c)$
grd5 : $r \notin comp_inprocess(c)$
grd6 : $r \notin comp_processed(c)$
grd7 : $c \notin failed$
grd8 : $master \notin failed$
then
act1 : $comp_buffer(c) := comp_buffer(c) \cup \{r\}$
end

Event *RequestOutMst* $\widehat{=}$
refines *RequestOut*
any
r
where
grd1 : $r \in comp_inprocess(master)$

grd2 : master $\notin$ failed
grd3 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow r \in comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c)$

then

act1 : comp_inprocess(master) := comp_inprocess(master) $\setminus$ {r}
act2 : comp_processed(master) := comp_processed(master) $\cup$ {r}

end

Event *ProcessStb* $\widehat{=}$

any

c
r

where

grd2 : c $\neq$ master
grd1 : c $\in$ comp
grd3 : r $\in$ comp_buffer(c)
grd4 : c $\notin$ failed

then

act1 : comp_inprocess(c) := comp_inprocess(c) $\cup$ {r}
act2 : comp_buffer(c) := comp_buffer(c) $\setminus$ {r}

end

Event *RequestOutStb* $\widehat{=}$

any

c
r

where

grd1 : c $\in$ comp
grd2 : c $\neq$ master
grd3 : r $\in$ comp_inprocess(c)
grd4 : c $\notin$ failed

then

act1 : comp_inprocess(c) := comp_inprocess(c) $\setminus$ {r}
act2 : comp_processed(c) := comp_processed(c) $\cup$ {r}

end

Event *ChangeMaster* $\widehat{=}$

refines *ChangeMaster*

any

new_master

where

grd1 : new_master $\in$ comp
grd2 : new_master $\neq$ master
grd3 : $\forall c \cdot c \in comp \wedge c \neq new_master \wedge c \neq master \Rightarrow comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c) \subseteq comp_buffer(new_master) \cup comp_inprocess(new_master) \cup comp_processed(new_master)$
grd4 : master $\in$ failed

grd5 : new_master $\notin$ failed

with

n_buffer : n_buffer = comp_buffer(new_master)
n_inprocess : n_inprocess = comp_inprocess(new_master)
n_processed : n_processed = comp_processed(new_master)

then

act1 : master := new_master
act2 : comp_buffer := {master} $\lhd$ comp_buffer
act3 : comp_inprocess := {master} $\lhd$ comp_inprocess
act4 : comp_processed := {master} $\lhd$ comp_processed
act5 : comp := comp $\setminus$ {master}
act6 : failed := failed $\setminus$ {master}
act7 : in_transit := TRUE

end

Event *CompActivation* $\widehat{=}$

extends *CompActivation*

any

c

where

grd1 : $c \in COMP$
grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$
act2 : comp_buffer(c) := $\varnothing$
act3 : comp_inprocess(c) := $\varnothing$
act4 : comp_processed(c) := $\varnothing$
act5 : in_transit := TRUE

end

Event *CompDeactivation* $\widehat{=}$

extends *CompDeactivation*

any

c

where

grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : c $\in$ failed

then

act1 : $comp := comp \setminus \{c\}$
act2 : comp_buffer := {c} $\lhd$ comp_buffer
act3 : comp_inprocess := {c} $\lhd$ comp_inprocess
act4 : comp_processed := {c} $\lhd$ comp_processed
act5 : failed := failed $\setminus$ {c}

end

Event *CompFailure* $\widehat{=}$

any

c

where

grd1 : $c \in comp$

grd2 : $c \notin failed$

then

act1 : $failed := failed \cup \{c\}$

end

Event *CompStbRecovery* $\widehat{=}$

any

c

where

grd1 : $c \in failed$

grd2 : $c \neq master$

then

act1 : $failed := failed \setminus \{c\}$

end

Event *TransitionOver* $\widehat{=}$

when

grd1 : $in_transit = TRUE$

grd2 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp_processed(master) \subseteq comp_buffer(c) \cup comp_inprocess(c) \cup comp_processed(c)$

then

act1 : $in_transit := FALSE$

end

END

MACHINE replication3_r2

REFINES replication3_r1

SEES replication_cnt

VARIABLES

comp

master

failed

log

index_written

index_inprocess

index_processed

n

in_transit

INVARIANTS

inv1 : $n \in \mathbb{N}$

inv2 : $log \in comp \rightarrow (1 \mathinner{..} n \nrightarrow REQUESTS)$

inv3 : $index_written \in comp \rightarrow \mathbb{N}$

inv4 : $index_inprocess \in comp \rightarrow \mathbb{N}$

inv5 : $index_processed \in comp \rightarrow \mathbb{N}$

inv6 : $index_written(master) = n$

inv7 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_inprocess(c)+1..index_written(c)] = comp_buffer(c)$

inv8 : $\forall c \cdot c \in comp \Rightarrow log(c)[index_processed(c)+1..index_inprocess(c)] = comp_inprocess(c)$

inv9 : $\forall c \cdot c \in comp \Rightarrow log(c)[1..index_processed(c)] = comp_processed(c)$

inv10 : $log(master) \in 1 \mathinner{..} n \rightarrowtail REQUESTS$

inv11 : $index_inprocess(master) \leq n$

inv12 : $index_processed(master) \leq index_inprocess(master)$

inv13 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_inprocess(c) \leq index_written(c)$

inv14 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_processed(c) \leq index_inprocess(c)$

inv15 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c) \in 1 \mathinner{..} index_written(c) \rightarrowtail REQUESTS$

inv16 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c) \subseteq log(master)$

inv17 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c)[index_written(c)+1..index_processed(master)] = \varnothing$

inv18 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow index_written(c) \leq index_written(master)$

inv19 : $in_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(master)[1..index_processed(master)] \subseteq log(c)[1 \mathinner{..} index_written(c)])$

inv20 : $\forall c, i \cdot c \in comp \wedge i \in 1..index_written(c) \Rightarrow log(c)(i) = log(master)(i)$

inv21 : $\forall c1, c2, i \cdot c1 \in comp \land c2 \in comp \land i \in 1 \mathinner{..} index_written(c1) \land i \in 1 \mathinner{..} index_written(c2) \Rightarrow log(c1)(i) = log(c2)(i)$

EVENTS

Initialisation

begin

act3 : $comp := init_comp$

act4 : $master := init_master$

act6 : $failed := \varnothing$

act7 : $log := init_comp \times \{\varnothing\}$

act8 : $index_written := init_comp \times \{0\}$

act9 : $index_inprocess := init_comp \times \{0\}$

act10 : $index_processed := init_comp \times \{0\}$

act11 : $n := 0$

act12 : $in_transit := FALSE$

end

Event *RequestInMst* $\widehat{=}$

refines *RequestInMst*

any

r

where

grd1 : $r \in REQUESTS$

grd2 : $r \notin log(master)[1 \mathinner{..} n]$

grd3 : $master \notin failed$

then

act1 : $log(master) := log(master) \cup \{n + 1 \mapsto r\}$

act2 : $n := n + 1$

act3 : $index_written(master) := index_written(master) + 1$

end

Event *ProcessMst* $\widehat{=}$

refines *ProcessMst*

when

grd1 : $index_inprocess(master) \neq index_written(master)$

grd2 : $master \notin failed$

with

r : $r = log(master)(index_inprocess(master) + 1)$

then

act1 : $index_inprocess(master) := index_inprocess(master) + 1$

end

Event *RequestInStb* $\widehat{=}$

refines *RequestInStb*

any

c

where

grd1 : c ∈ comp
grd2 : c ≠ master
grd3 : c ∉ failed
grd4 : master ∉ failed
grd5 : index_written(c) < index_written(master)

with

r : r = log(master)(index_written(c) + 1)

then

act1 : log(c) := log(c)∪{index_written(c)+1 ↦ log(master)(index_written(c)+1)}
act2 : index_written(c) := index_written(c) + 1

end

Event *RequestOutMst* ≙

refines *RequestOutMst*

when

grd1 : master ∉ failed
grd2 : index_processed(master) ≠ index_inprocess(master)
grd3 : ∀c·c ∈ comp∧c ≠ master⇒index_written(c) > index_processed(master)

with

r : r = log(master)(index_processed(master) + 1)

then

act1 : index_processed(master) := index_processed(master) + 1

end

Event *ProcessStb* ≙

refines *ProcessStb*

any

c

where

grd1 : c ∈ comp
grd2 : c ≠ master
grd3 : c ∉ failed
grd4 : index_inprocess(c) ≠ index_written(c)

with

r : r = log(c)(index_inprocess(c) + 1)

then

act1 : index_inprocess(c) := index_inprocess(c) + 1

end

Event *RequestOutStb* ≙

refines *RequestOutStb*

any

c

where

grd1 : c ∈ comp

grd2 : $c \neq master$
grd3 : $c \notin failed$
grd4 : $index_processed(c) \neq index_inprocess(c)$

with

r : $r = log(c)(index_processed(c) + 1)$

then

act1 : $index_processed(c) := index_processed(c) + 1$

end

Event *ChangeMaster* $\widehat{=}$

refines *ChangeMaster*

any

new_master
c

where

grd1 : $c = master$
grd2 : $new_master \in comp$
grd3 : $new_master \neq master$
grd5 : $master \in failed$
grd6 : $new_master \notin failed$
grd7 : $\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq new_master \Rightarrow index_written(c) \leq index_written(new_master)$

then

act1 : $master := new_master$
act5 : $comp := comp \setminus \{master\}$
act6 : $failed := failed \setminus \{master\}$
act7 : $log := \{master\} \lhd log$
act8 : $n := index_written(new_master)$
act9 : $index_written := \{master\} \lhd index_written$
act10 : $index_inprocess := \{master\} \lhd index_inprocess$
act11 : $index_processed := \{master\} \lhd index_processed$
act12 : $in_transit := TRUE$

end

Event *CompActivation* $\widehat{=}$

refines *CompActivation*

any

c

where

grd1 : $c \in COMP$
grd2 : $c \notin comp$

then

act1 : $comp := comp \cup \{c\}$
act2 : $log(c) := \varnothing$
act3 : $index_written(c) := 0$
act4 : $index_inprocess(c) := 0$

act5 : index_processed(c) := 0
act6 : in_transit := TRUE

end

Event *CompDeactivation* $\widehat{=}$

refines *CompDeactivation*

any

c

where

grd1 : $c \in comp$
grd2 : $c \neq master$
grd3 : $c \in failed$

then

act1 : $comp := comp \setminus \{c\}$
act2 : $log := \{c\} \lhd log$
act3 : $index_written := \{c\} \lhd index_written$
act4 : $index_inprocess := \{c\} \lhd index_inprocess$
act5 : $index_processed := \{c\} \lhd index_processed$
act6 : $failed := failed \setminus \{c\}$

end

Event *CompFailure* $\widehat{=}$

extends *CompFailure*

any

c

where

grd1 : $c \in comp$
grd2 : $c \notin failed$

then

act1 : $failed := failed \cup \{c\}$

end

Event *CompStbRecovery* $\widehat{=}$

extends *CompStbRecovery*

any

c

where

grd1 : $c \in failed$
grd2 : $c \neq master$

then

act1 : $failed := failed \setminus \{c\}$

end

Event *TransitionOver* $\widehat{=}$

refines *TransitionOver*

when

grd1 : $in_transit = TRUE$

grd2 : $\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(master)[1..index_processed(master)] \subseteq log(c)[1 .. index_written(c)]$

then

act1 : $in_transit := FALSE$

end

END

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2887-2
ISSN 1239-1891