Asynchronous Client-Side Coordination of Cluster Service Sessions

Karolis Petrauskas(✉) and Romas Baronas

Vilnius University, Institute of Computer Science, Vilnius, Lithuania
{karolis.petrauskas,romas.baronas}@mif.vu.lt

Abstract. A system-to-system communication involving stateful sessions between a clustered service provider and a service consumer is investigated in this paper. An algorithm allowing to decrease a number of calls to failed provider nodes is proposed. It is designed for a clustered client and is based on an asynchronous communication. A formal specification of the algorithm is formulated in the TLA$^+$ language and was used to investigate the correctness of the algorithm.

Keywords: Session management · Cluster · Formal specification

1 Introduction

Nowadays business applications include a lot of interactions with service providers for handling various operations like order and payment processing, application monitoring and other specialized services [11]. The business applications themselves are often provided as services [12]. This kind of system architecture leads to many system-to-system integrations. Requirements for high availability and fault tolerance impose use of clustered topologies. In the case of system-to-system communications, clustered topologies are often used on the service consumer side as well as on the service provider side. The service providers are often deployed on a cloud or another virtualized infrastructure. Such infrastructure provides a lot of flexibility, but introduces a network instability, connection drops and other disruptions caused node migrations [1,14].

A lot of service providers implement the model of the eventual consistency in order to maintain high availability together with the service scalability [3]. That means the consistency is not guaranteed globally and special requirements are imposed on the service consumers in order to minimize the observed inconsistency. A common requirement for the clients of such services is to maintain session stickiness to particular nodes in the provider cluster [13]. That applies also to the stateless protocols, as requests for the particular end-user should be routed to the same back-end node in order to minimize the primary-node or the cache misses causing data inconsistency for a particular user.

A lot of mainstream protocols have no support for detecting lost connections or server failures immediately [2]. In such cases, the node availability should be

A. Lupeikiene et al. (Eds.): DB&IS 2018, CCIS 838, pp. 121–133, 2018.
https://doi.org/10.1007/978-3-319-97571-9_11

tracked by examining responses to the service requests. Only specific faults can be used as an indication of the failed provider node, excluding all the business faults as well as bad requests. If the service is accessed rarely, additional fake requests can be performed in order to keep the sessions alive or to detect node failures faster, before next user request will be received.

One of the ways to handle failing provider nodes is to consider another server from the remaining list and use it onwards for the session. This strategy can be inefficient if applied for each session separately, without sharing the knowledge on the failed nodes in the case of multiple sessions bound to a single provider node. After detecting the node failure, the error can be propagated to the caller or fail-over to another provider node can be performed silently, without interrupting the caller. Even if the error is handled by the consumer application, usually it has an impact on the behaviour of the system at least as increased execution time of some operations [2,8]. Because of that, the number of calls reaching the failed nodes should be minimized. The optimization usually includes sharing the node availability information between the sessions.

Applications consuming the provider services are often implemented as clusters themselves. The state sharing in the cluster is much more expensive than in a single node, especially if consistency should be preserved [12]. Inconsistency in tracking back-end availability has relatively low cost, as fixing it can only cause several unnecessary calls to the failed nodes. Keeping that in mind, it is reasonable to implement the sharing of the back-end node availability without consistency guarantees, employing the best-effort strategy. One of the ways for implementing it is to use asynchronous messages to share the known information on the provider availability.

Different applications require complex event processing relying on the detection of composite events often formed by logical and temporal combinations of events coming from many sources [9]. Various formal methods handling temporally composed events have been designed and implemented for complex event processing [4,7]. The Temporary Logic of Actions (TLA) is among such methods successfully used to describe behaviours of concurrent systems [5]. The corresponding specification language TLA$^+$ and the TLC model checker help to prevent serious bugs from reaching production as well as to optimize complex algorithms without sacrificing quality [10].

An algorithm for coordinating sessions using asynchronous messages in the consumer cluster is proposed in this paper. In order to avoid misbehaviours in various corner cases, the algorithm was formulated as a formal specification in the TLA$^+$ language [5,6]. The specification was verified by performing model checking [15], employing the TLC tool provided by the TLA$^+$ toolbox. We first provide a direct solution of the problem in Sect. 3 and show its misbehaviour by performing model checking. Then we propose two modifications of the algorithm in Sects. 4 and 5. In Sect. 6 we provide the details on the performed model checking and discuss the difference between the proposed correct solutions.

2 Principal Structure

We consider an interaction between two systems – a service provider and a service consumer. Both systems are assumed to be master-less clusters consisting of several nodes. The nodes of the consumer cluster maintain a set of sessions bound to some nodes in the service provider cluster. A structure of the elements participating in the session management is shown as a UML class diagram in Fig. 1.

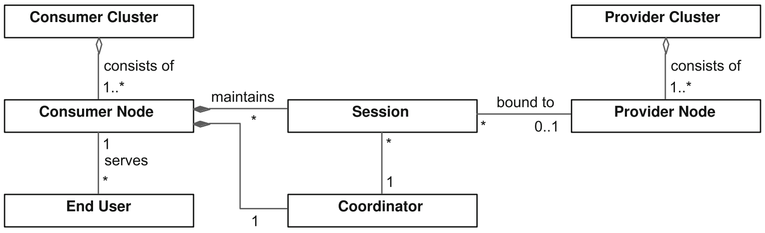

Fig. 1. Principal structure of the modelled subsystem

The main idea of the session management algorithm is that a client-side session process notifies its coordinator when a failure of the provider node is detected. The coordinator then notifies the other sessions bound to the same provider node and the coordinators on the other consumer nodes. The coordinators then notifies the corresponding sessions on their nodes. In that way, all the sessions in the cluster can handle the failure of the provider node gracefully.

We assume that a session can be bound to another node in the case of a provider failure, although re-binding of sessions should be avoided, as the cost of such operation is not negligible. The cost can be expressed in terms of performance drop or a possibility to provide the end-user with inconsistent data, etc. A session can be unbound, i.e. not bound to any of the provider nodes. This can be the case for the sessions that were dropped by the provider and were not reconnected yet.

3 Formal Specification

In this section we define a formal specification for the session management algorithm that relies on the asynchronous communication for sharing the knowledge about the provider node availability. We assume each session to be a separate process in a node. These processes communicate asynchronously with a coordinator process responsible for tracking a state of the provider cluster in the consumer node.

3.1 State of the Model

The specification of the session management algorithm is formulated in the TLA$^+$ language and has several parameters (constants). A constant in the specification does not change during a single simulation (model checking), but can have different values in separate simulations. The following excerpt defines constants and a state structure of the proposed specification:

$$
\begin{array}{l}
\textsc{constants}\ \mathit{PNodes},\ \mathit{CNodes},\ \mathit{SNames}\\
\textsc{variables}\ \mathit{prov},\ \mathit{cons}\\
\mathit{NA} \triangleq \textsc{choose}\ n : n \notin \mathit{PNodes}\\
\mathit{Msg} \triangleq [\mathit{pn} : \mathit{PNodes}]\\
\mathit{TypeOK} \triangleq \mathit{prov} \in [\mathit{PNodes} \rightarrow \textsc{boolean}\ ] \land \mathit{cons} \in [\mathit{CNodes} \rightarrow [\\
\qquad\qquad\qquad c : [\mathit{PNodes}\ \rightarrow [\mathit{st}\ : \textsc{boolean}\ ]],\\
\qquad\qquad\qquad s : [\mathit{SNames} \rightarrow [\mathit{pn} : \mathit{PNodes} \cup \{\mathit{NA}\},\ m : \textsc{subset}\ \mathit{Msg}]],\\
\qquad\qquad\qquad \mathit{sm} : \textsc{subset}\ \mathit{Msg},\ \mathit{cm} : \textsc{subset}\ \mathit{PNodes}]]
\end{array}
$$

In order to keep the specification simple and the state space finite, we consider a number of consumer and provider nodes as well as a number of sessions in each node to be constant. The constant *PNodes* stands for a set of provider nodes. Each node in this set is defined by assigning a unique identifier, e.g. $\mathit{PNodes} = \{\mathrm{p}_1, \mathrm{p}_2\}$. Similarly the constant *CNodes* stands for a set of consumer nodes. The constant *SNames* stands for a session pool in the consumer node and should be assigned with a set of session identifiers.

Systems are modelled as state machines in TLA$^+$. Variables define a state structure of the machine. In this specification the variable *prov* represents the actual state of the provider nodes. This variable is a function with the domain *PNodes* and the range BOOLEAN, where TRUE means the corresponding node is operational, and FALSE – the node is down.

The variable *cons* represents the state of the consumer cluster including its view of the provider nodes. It is a function with a domain *CNames* and therefore describes state for each node in the consumer cluster separately.

A state of the coordinator process is represented by the field c, that holds known states for all the provider nodes in each consumer node. The state of a particular provider node $\mathit{cons}[\mathit{cn}].c[\mathit{pn}].\mathit{st}$ (where $\mathit{cn} \in \mathit{CNodes}$ and $\mathit{pn} \in \mathit{PNodes}$) can differ from $\mathit{prov}[\mathit{pn}]$, because changes of the node availability are not detected immediately by the consumer nodes.

The field $\mathit{cons}[\mathit{cn}].s$ stands for a session pool in a consumer node. Each session $\mathit{cons}[\mathit{cn}].s[\mathit{sn}]$ (where $\mathit{sn} \in \mathit{SNames}$) is bound to a node $\mathit{pn} \in \mathit{PNodes}$ or is unbound, if $\mathit{cons}[\mathit{cn}].s[\mathit{sn}].\mathit{pn} = \mathit{NA}$. The session has also a set of asynchronous messages $\mathit{cons}[\mathit{cn}].s[\mathit{sn}].m$ received from the coordinator in the current node. Synchronous calls are modelled as direct changes of the corresponding variables. In this algorithm we consider messages sent to the sessions by the coordinator to be asynchronous. The set of possible messages is defined as *Msgs*.

The fields *sm* and *cm* in $\mathit{cons}[\mathit{cn}]$ stand for sets of asynchronous messages received by the coordinator process correspondingly from the sessions in the current node and the coordinators in other consumer nodes.

A set of valid states in the specification is defined by the predicate *TypeOK*. This predicate is used to check the type correctness of the specification.

3.2 Behaviour of the Provider Nodes

Transitions of the state machine are defined by the actions – formulas involving primed variables (they stand for the variable values in the next step). Actions describing behaviour of the provider nodes are the following:

$$\begin{aligned}
&\mathit{ProvNodeUp}(pn) \triangleq \neg \mathit{prov}[pn]\\
&\quad \wedge \mathit{prov}' = [\mathit{prov} \text{ EXCEPT } ![pn] = \text{TRUE}]\\
&\quad \wedge \text{UNCHANGED } \langle \mathit{cons} \rangle\\
&\mathit{ProvNodeDown}(pn) \triangleq \mathit{prov}[pn]\\
&\quad \wedge \mathit{prov}' = [\mathit{prov} \text{ EXCEPT } ![pn] = \text{FALSE}]\\
&\quad \wedge \text{UNCHANGED } \langle \mathit{cons} \rangle
\end{aligned}$$

The action $\mathit{ProvNodeUp}(pn)$ states that the provider node $pn \in \mathit{PNodes}$ can become operational at any time if it is currently down. The expression $[\mathit{prov} \text{ EXCEPT } ![pn] = \text{TRUE}]$ stands for a function that is equal to *prov* except that the value of $\mathit{prov}[pn]$ equals to TRUE. The state of the consumer nodes is not affected by this transition (UNCHANGED $\langle \mathit{cons} \rangle$) as the availability of the provider node is only detected by the consumer later by performing some operations. The action $\mathit{ProvNodeDown}(pn)$ correspondingly turns operational node down.

3.3 Behaviour of a Consumer Session

This section describes operation of the session processes. A session can either handle requests, update its state based on messages from the coordinator or connect if it was not bound to any provider node. The latter is modelled by the action $\mathit{SessionConnect}(cn, sn)$, where $cn \in \mathit{CNodes}$ stands for a consumer node and $sn \in \mathit{SNames}$ stands for a session identifier. This action is enabled, if the session is not bound to a provider node ($\mathit{cons}[cn].s[sn] = \mathit{NA}$) and there is a node $pn \in \mathit{PNodes}$ that is operational ($\mathit{prov}[pn] = \text{TRUE}$) and the consumer node knows it is operational ($\mathit{cons}[cn].c[pn].st = \text{TRUE}$),

$$\begin{aligned}
&\mathit{SessionConnect}(cn, sn) \triangleq \mathit{cons}[cn].s[s].pn = \mathit{NA} \wedge \mathit{cons}[cn].c[pn].st\\
&\quad \wedge \exists\, pn \in \mathit{PNames} : \wedge\, \mathit{prov}[pn]\\
&\qquad\qquad\qquad\qquad\quad \wedge\, \mathit{cons}' = [\mathit{cons} \text{ EXCEPT } ![cn].s[sn].pn = pn]\\
&\qquad\qquad\qquad\qquad\quad \wedge \text{UNCHANGED } \langle \mathit{prov} \rangle
\end{aligned}$$

When connected ($\mathit{cons}[cn].s[sn].pn \in \mathit{PNodes}$), a session can be used by the consumer node to issue requests to the service provider. Only failing requests are modelled in this specification, because the successful requests do not affect the state of the modelled subsystem. We consider all the requests ended up with business faults as completed successfully. A request is considered failed only if the corresponding provider node is down ($\mathit{prov}[pn] = \text{FALSE}$) at the moment, when the request is performed. In that case the session marks itself as unbound and sends an asynchronous message indicating the failure of the provider node to the coordinator process. The state of the other sessions as well as the state of the coordinator is not affected in this transition directly,

$$
\begin{aligned}
&\mathit{SessionReqFail}(cn, sn) \triangleq \mathit{cons}[cn].s[sn].pn \in \mathit{PNodes} \land \neg \mathit{prov}[\mathit{cons}[cn].s[sn].pn] \\
&\quad \land \mathit{cons}' = [\mathit{cons} \text{ EXCEPT} \\
&\qquad\qquad\qquad !\,[cn].s[sn].pn = \mathit{NA}, \\
&\qquad\qquad\qquad !\,[cn].sm = @ \cup \{[pn \mapsto \mathit{cons}[cn].s[sn].pn]\}] \\
&\quad \land \text{UNCHANGED } \mathit{prov}
\end{aligned}
$$

The symbol @ in this and other formulas stands for the current value of the function.

Sending an asynchronous message is modelled by adding it to the set of messages $\mathit{cons}[cn].sm$ sent by the sessions to the coordinator. The ordering of messages is not modelled in this specification in order to decrease the space of possible states. This also allows to avoid complicated requirements for an implementation. Duplicated messages are modelled by not removing a message from the set $\mathit{cons}[cn].sm$ after processing it.

A session can receive notifications from the coordinator indicating provider nodes that became down. Upon receiving such a message the session unbinds itself, if the provider node specified in the message matches with the node the session is bound to. This behaviour is modelled by the following action:

$$
\begin{aligned}
&\mathit{SessionUpdate}(cn, sn) \triangleq \\
&\quad \exists\, msg \in \mathit{cons}[cn].s[sn].m : \\
&\qquad \exists\, msgsDeq \in \{\mathit{cons}[cn].s[sn].m,\ \mathit{cons}[cn].s[sn].m \setminus \{msg\}\} : \\
&\qquad\quad \land \mathit{cons}' = \text{LET } consDeq \triangleq [\mathit{cons} \text{ EXCEPT } !\,[cn].s[sn].m = msgsDeq] \\
&\qquad\qquad\qquad\quad \text{IN } \ \text{IF } msg.pn = \mathit{cons}[cn].s[sn].pn \\
&\qquad\qquad\qquad\qquad\quad \text{THEN } [consDeq \text{ EXCEPT } !\,[cn].s[sn].pn = \mathit{NA}] \\
&\qquad\qquad\qquad\qquad\quad \text{ELSE } \ consDeq \\
&\qquad\quad \land \text{UNCHANGED } \mathit{prov}
\end{aligned}
$$

Receiving a message (dequeuing) is modelled by taking any message from the set of sent messages $\mathit{cons}[cn].s[sn].m$ ignoring their order. The set of sent messages is either left unchanged or the selected message is removed from that set. The former case models the situation when there were several identical messages in the queue. This case also models duplication of messages, that can occur because of various retries or message re-deliveries in the software implementing this algorithm. The latter case models dequeuing of the last message of that kind. It also covers the situation, when part of messages can be lost.

3.4 Behaviour of the Consumer Node Coordinator

The coordinator is responsible for maintaining the state of the provider nodes in a single consumer node. It is responsible also for sharing this knowledge between the consumer nodes. The coordinator receives messages indicating failures of the provider nodes from the sessions. Then it updates its internal state ($\mathit{cons}[cn].c[pn].st$) and notifies all the sessions and other consumer nodes about the state changes, if some node becomes unavailable. This is modelled by the following action:

```
$CoordSessionMsg(cn) \triangleq$
  $\exists\, msg \in cons[cn].sm : \exists\, sm \in \{cons[cn].sm,\ cons[cn].sm \setminus \{msg\}\} :$
    $\text{LET } consDeq \triangleq [cons \text{ EXCEPT } ![cn].sm = sm]$
        $consEnq \triangleq [c \in \text{DOMAIN } consDeq \mapsto [consDeq[c] \text{ EXCEPT}$
                    $!.cm = \text{IF } c = cn \text{ THEN } @ \text{ ELSE } @ \cup \{msg.pn\}]]$
        $consUpd \triangleq [consEnq \text{ EXCEPT}$
                    $![cn].c[msg.pn].st = \text{FALSE},$
                    $![cn].s = [s \in \text{DOMAIN } @ \mapsto [@[s] \text{ EXCEPT } !.m = @ \cup \{msg\}]]]$
    $\text{IN} \quad \wedge\, cons' = \text{IF } cons[cn].c[msg.pn].st \text{ THEN } consUpd \text{ ELSE } consDeq$
          $\wedge\, \text{UNCHANGED } prov$
```

The coordinator sends notifications to other consumer nodes when some provider node becomes offline. These notifications are handled by coordinators on the corresponding consumer nodes. Handling these notifications is modelled by the action *CoordClusterMsg*(cn). Main distinction from *CoordSessionMsg*(cn) in this action is that the corresponding provider node is checked explicitly (the conjunct $\neg prov[pn]$) before marking it as offline. This check is kept explicit in order to avoid accidental marking of operational node as unavailable. We assume the network can make a particular provider node visible from one consumer node and not visible from another. Another distinction is that the notification is not propagated to other nodes here,

```
$CoordClusterMsg(cn) \triangleq$
  $\exists\, pn \in cons[cn].cm : \exists\, cm \in \{cons[cn].cm,\ cons[cn].cm \setminus \{pn\}\} :$
    $\text{LET } consDeq \triangleq [cons \text{ EXCEPT } ![cn].cm = cm]$
         $consEnq \triangleq [consDeq \text{ EXCEPT } ![cn].c[pn].st = \text{FALSE},$
           $![cn].s = [s \in \text{DOMAIN } @ \mapsto [@[s] \text{ EXCEPT } !.m = @ \cup \{[pn \mapsto pn]\}]]]$
    $\text{IN} \quad \wedge\, cons' = \text{IF } cons[cn].c[pn].st \wedge \neg prov[pn] \text{ THEN } consEnq \text{ ELSE } consDeq$
           $\wedge\, \text{UNCHANGED } prov$
```

As shown above, the coordinator marks the provider nodes as being down in the consumer state based on messages from the sessions and the other coordinators. The coordinator is also responsible for marking the nodes as being available, when they become operational. This is performed periodically by checking the nodes that are currently marked as down ($cons[cn].c[pn].st = \text{FALSE}$) and marking them available if the checks succeed. This is modelled by the action *CoordProviderCheck*(cn, pn). The check of the provider node is performed synchronously and is modelled here by the conjunct $prov[pn]$,

```
$CoordProviderCheck(cn,\ pn) \triangleq \neg cons[cn].c[pn].st \wedge prov[pn]$
  $\wedge\, cons' = [cons \text{ EXCEPT } ![cn].c[pn].st = \text{TRUE}]$
  $\wedge\, \text{UNCHANGED } prov$
```

3.5 Temporal Properties

The complete specification in TLA$^+$ is represented as a temporal formula

$$Spec \triangleq Init \wedge \Box[Next]_{\langle prov, cons \rangle} \wedge Liveness$$

where *Init* describes the initial state, *Next* defines all the possible transitions at any step and *Liveness* defines requirements for actions to actually occur. Here $\Box$ is a temporal operator "always". The expression $[Next]_{\langle prov, cons \rangle}$ states that either a step *Next* or a step not changing the variables *prov* and *cons* can occur.

The formula *Init* stands for the initial state. It is similar to the *TypeOK* predicate, except that message sets are initialized with empty sets {} and all the provider nodes are assumed to be operational initially. The formula *Next* is a disjunction of all the actions and describes all the possible transitions at any step. This formula straightforward and therefore is omitted in this paper.

Liveness is a temporal formula describing what actions should actually occur in the system if they are enabled (contrary to "can occur"). We assume weak fairness conditions (an action will be performed if it is enabled forever) for all the actions describing behaviour of the consumer nodes (the sessions and the coordinators).

The specification *Spec* can be used to check if it satisfies required properties. A typical property usually checked for any specification is a type correctness invariant

$$TypeInvariant \triangleq Spec \Rightarrow \Box\, TypeOK$$

Apart from simple invariants, TLA$^+$ allows to define temporal properties. These properties imply requirements for the entire behaviour (a sequence of transitions). The following temporal properties are expected to be held in the system:

$$\begin{aligned}
&NodeDownDetected \triangleq \\
&\quad \forall\, pn \in PNodes,\ cn \in CNodes,\ sn \in SNames: \\
&\qquad (cons[cn].s[sn].pn = pn \wedge \neg prov[pn]) \leadsto (cons[cn].s[sn].pn = NA \vee prov[pn]) \\
&SessionsWillReconnect \triangleq \\
&\quad \forall\, pn \in PNodes,\ cn \in CNodes,\ sn \in SNames: \\
&\qquad (cons[cn].s[sn].pn = NA \wedge prov[pn]) \leadsto (cons[cn].s[sn].pn \neq NA \vee \neg prov[pn])
\end{aligned}$$

The temporal property *NodeDownDetected* asserts that if a provider node becomes unavailable, then sessions bound to it will be eventually disconnected, unless the node will become operational again ($\leadsto$ is the temporal operator "leads to"). It was checked that this property holds for the specification by employing the TLC model checker.

The property *SessionsWillReconnect* asserts, that if a session is unbound and there is an operational node, the session will reconnect and will continue to serve requests. These properties should be implied by the specification,

$$TemporalProperties \triangleq Spec \Rightarrow NodeDownDetected \wedge SessionsWillReconnect$$

The TLC model checker was used to check the type correctness invariant as well as the temporal properties defined above. The model checking showed that property *SessionsWillReconnect* is not satisfied by the specification. The misbehaviour is caused by the asynchronous communication between the sessions and the coordinator. One of the counter-examples: a provider node was down, then it becomes available, coordinator process marks it as available and then receives a delayed message from a session indicating node failure. As a consequence, the node is marked as unavailable again till the next $\mathit{CoordProviderCheck}(pn)$. This behaviour can repeat infinitely, making the consumer to consider running provider node as failed thus decreasing availability of the system.

4 Explicit Provider Checks

Another possible solution allowing to avoid the impact of the delayed messages is to check node availability before marking it as offline in the coordinator process. In that case, the $\mathit{CoordSessionMsg}(cn)$ action should be changed by adding expression $\mathit{cons}[cn].c[\mathit{msg}.pn].st \land \neg \mathit{prov}[\mathit{msg}.pn]$ instead of $\mathit{cons}[cn].c[\mathit{msg}.pn].st$ in the IF condition. The changed parts of the action are as follows:

$$\begin{array}{l}
\mathit{CoordSessionMsg}(cn) \triangleq \\
\quad \ldots \\
\quad \land\ \mathit{cons}' = \textsc{if}\ \mathit{cons}[cn].c[\mathit{msg}.pn].st \land \neg \mathit{prov}[\mathit{msg}.pn]\ \textsc{then}\ \mathit{consUpd}\ \textsc{else}\ \mathit{consDeq} \\
\quad \land\ \textsc{unchanged}\ \mathit{prov}
\end{array}$$

With this change the temporal property *SessionsWillReconnect* is fulfilled. The drawback of this approach is that additional calls to the service provider must be performed.

5 Detecting Delayed Messages

In order to avoid the impact of the delayed messages, generations of the provider nodes can be introduced. Each time when a provider node is detected to become online by the coordinator, its generation number is increased. Messages referring to generations older than one known by the coordinator are then ignored. The generations should be tracked in the coordinator process as well as in the sessions and should be included in the messages exchanged between them. The updated structure of the messages and the state of the model are as follows:

$$\begin{array}{l}
\mathit{Msg} \triangleq [pn : \mathit{PNodes},\ \mathit{gen} : \mathit{Nat}] \\
\mathit{TypeOK} \triangleq \\
\quad \land\ \mathit{prov} \in [\mathit{PNodes} \rightarrow \textsc{boolean}\ ] \\
\quad \land\ \mathit{cons} \in [\mathit{CNodes} \rightarrow [\\
\qquad\quad c : [\mathit{PNodes} \rightarrow [st\ : \textsc{boolean}\ ,\ \mathit{gen} : \mathit{Nat}]], \\
\qquad\quad s : [\mathit{SNames} \rightarrow [pn : \mathit{PNodes} \cup \{\mathit{NA}\},\ \mathit{gen} : \mathit{Nat},\ m : \textsc{subset}\ \mathit{Msg}]], \\
\qquad\quad \mathit{sm} : \textsc{subset}\ \mathit{Msg},\ \mathit{cm} : \textsc{subset}\ \mathit{PNodes}]]
\end{array}$$

The observed generations of the provider nodes are tracked inside of the consumer nodes and are not shared between them. Each node can observe different provider node interruptions. Moreover, depending on a network topology, a particular provider node can be accessible from one consumer node and not accessible from other. The message delays between the consumer nodes are handled by the explicit node checks (conjunct $\neg prov[pn]$) in the action $\mathit{CoordClusterMsg}(cn)$.

Some parts of the model should be updated to maintain the observed provider node generations. The initial state can start from any generation. We consider to have $gen \mapsto 0$ in all the sessions and the coordinators.

For the coordinator behaviour, the $\mathit{CoordProviderCheck}(cn, pn)$ action is changed to increment the node generation each time the coordinator detects it became available,

$$\begin{aligned}
&\mathit{CoordProviderCheck}(cn,\ pn) \triangleq \neg cons[cn].c[pn].st \land prov[pn] \\
&\quad \land cons' = [cons \text{ EXCEPT } ![cn].c[pn].st = \text{TRUE},\ ![cn].c[pn].gen = @ + 1] \\
&\quad \land \text{UNCHANGED } prov
\end{aligned}$$

The coordinator then ignores all the messages received with old generations ($msg.gen < cons[cn].c[msg.pn].gen$) in the $\mathit{CoordSessionMsg}(cn)$ action. It also includes the generation into the messages sent to the sessions when node change is detected on a notification from other consumer nodes in $\mathit{CoordClusterMsg}(cn)$. The generation is included in the messages triggered by the session notifications in the $\mathit{CoordSessionMsg}(cn)$ action without changes in the specification as it only forwards received messages (and they include the gen field).

When connecting, a session takes the current provider node generation from the coordinator in the consumer node ($cons[cn].c[pn].gen$), therefore the action $\mathit{SessionConnect}(cn, sn)$ is updated to assign the generation known by the session as follows:

$$\begin{aligned}
&\mathit{SessionConnect}(cn,\ sn) \triangleq cons[cn].s[s].pn = NA \land cons[cn].c[pn].st \\
&\quad \land \exists\, pn \in PNames : \land prov[pn] \\
&\qquad \land cons' = [cons \text{ EXCEPT } ![cn].s[sn].pn = pn, \\
&\qquad\qquad ![cn].s[sn].gen = cons[cn].c[pn].gen] \\
&\qquad \land \text{UNCHANGED } \langle prov \rangle
\end{aligned}$$

The sessions should only consider messages received from the coordinator in the $\mathit{SessionUpdate}(cn, sn)$ action with a generation not less than the current generation known by the session ($cons[cn].s[s].gen \leq msg.gen$) and then remember it as the last known generation ($![cn].s[s].gen = msg.gen$). All the other messages are just dequeued and ignored. The sessions include the generation to the messages sent to the coordinator in the $\mathit{SessionReqFail}(cn, sn)$ action.

6 Model Checking

Three variants of the specification were defined in Sects. 3, 4 and 5. All of them were model-checked for the type correctness as well as for the temporal properties. The model checking was performed on a laptop with 8 CPUs, 16 GB RAM and an SSD disk, using TLA$^+$ Toolbox version 1.5.6 with OpenJDK Java version 1.8.0 running on a Linux OS.

A model checking in TLA$^+$ is performed by defining a model and exploring its possible states. The model instantiates the specification with particular values for the constants. In all the cases the following values were used for the specification constants:

$$\mathit{CNodes} = \{\mathrm{c}_1, \mathrm{c}_2\}, \mathit{PNodes} = \{\mathrm{p}_1, \mathrm{p}_2\}, \mathit{SNames} = \{\mathrm{s}_1, \mathrm{s}_2\}.$$

Small sets were selected in order to decrease state space while keeping the model meaningful.

The specification maintaining observed generations of the provider nodes (Sect. 5) has fields $gen \in Nat$, whose range is infinite. In order to keep the state space finite, additional constraint was used when checking the model,

$$\forall cn \in \mathit{CNodes}, pn \in \mathit{PNodes} : cons[cn].c[pn].gen < 3.$$

For all the variants of the specification the type correctness invariant ($Spec \Rightarrow \Box\, TypeOK$) was checked for the entire space of the state values (with the constraints shown above). The temporal properties were checked only for a fraction of the state space. If a model checking was running for 10 h with no property violations reported, the checking was stopped and considered successful. The violation of the *SessionsWillReconnect* property for the initial specification (Sect. 3) was found in 3 min.

6.1 Modelling Message Queues

Message queues were modelled as sets of sent messages [6]. This approach ignores the order of messages and also cannot represent several identical messages in the queue precisely.

Another way to model a message queue is to use a sequence instead of a set. In that case the order of messages is maintained and multiple messages with the same representation are supported. While this approach is more precise, it increases the state space a lot. The corresponding specifications were designed during this research with message queues represented by sequences. The model checker was unable to explore all the state space (the checking was stopped) even for the type correctness invariant with a constraint for the queues to have length less than 3.

Note also, that the violation of the property *SessionsWillReconnect* was not reproduced with these models in a reasonable time. Possible reasons for not reproducing the violation are: the model was to small (maybe longer message queues should be considered), the model checking was cancelled to early, respect of the message order solves the race condition leading to violation of this property. Particular reason for this was not found in this research.

6.2 Number of Synchronous Provider Checks

The specification variants in Sects. 4 and 5 both solve the race condition found when model checking the initial specification (Sect. 3).

The specification with the explicit checks (Sect. 4) is simpler to implement, though is not efficient. A session needs to check the provider node availability by performing a synchronous call each time it receives notification from the coordinator. The number of calls will be equal to the number of sessions bound to that particular provider node. In the case of uniform distribution of sessions over the provider nodes there will be $|SNames| \times |CNodes|/|PNodes|$ checks performed (usually $|SNames| \gg |PNodes|$). In the worst case (all the sessions are be bound to a single node) the number of checks will be $|SNames| \times |CNodes|$. The number of synchronous checks can cause two kinds of problems:

1. If the provider node is actually online, unnecessary calls will be issued to the provider. All the checks will be performed approximately in the same time and therefore can cause a notable increase in a load on the provider. The sessions cannot serve user requests while performing the check.
2. If the provider node is not reachable, the checks can take long time before failing, causing delays before switching to another node. This can be the case, if the provider becomes unavailable because of network interruptions or misconfiguration, e.g. when packets are dropped instead of rejecting them.

The specification maintaining the observed node generations (Sect. 5) allows to decrease the number of calls to the failing nodes. The number of synchronous checks will be equal to the number of nodes in the consumer cluster, as each node will perform single synchronous check.

7 Conclusions

The proposed algorithm for tracking provider node availability allows to avoid synchronous communication in the consumer cluster as well as inside of the consumer node. That allows to avoid process blocking thus decreasing impact on the performance. The algorithm was formulated by employing formal specification language and was model-checked for its correctness in a subset of its possible states.

The model checking showed that straight-forward solution of the problem works incorrectly at some race-conditions. Explicit node checks can be used to solve the inconsistencies though they introduce a lot of overhead and can cause bottlenecks in the system. The overhead can be decreased by tracking observed generations of the provider nodes. It is meaningful to track the generations in a single consumer node, although its usefulness cluster-wide depend on the network topology.

References

1. Armbrust, M., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010). https://doi.org/10.1145/1721654.1721672
2. Ayari, N., Barbaron, D., Lefevre, L., Primet, P.: Fault tolerance for highly available internet services: concepts, approaches, and issues. IEEE Commun. Surv. Tutor. **10**(2), 34–46 (2008). https://doi.org/10.1109/COMST.2008.4564478

3. Bailis, P., Ghodsi, A.: Eventual consistency today: limitations, extensions, and beyond. Queue **11**(3), 20:20–20:32 (2013). https://doi.org/10.1145/2460276.2462076
4. Hinze, A., Voisard, A.: EVA: an event algebra supporting complex event specification. Inf. Syst. **48**, 1–25 (2015). https://doi.org/10.1016/j.is.2014.07.003
5. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
6. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
7. Li, D., Zhang, Q., Zio, E., Havlin, S., Kang, R.: Network reliability analysis based on percolation theory. Reliab. Eng. Syst. Saf. **142**, 556–562 (2015). https://doi.org/10.1016/j.ress.2015.05.021
8. Lowell, D.E., Chandra, S., Chen, P.M.: Exploring failure transparency and the limits of generic recovery. In: Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, vol. 4, p. 15, OSDI 2000, USENIX Association, Berkeley (2000). Article No. 20
9. Luckham, D.C.: Event Processing for Business: Organizing the Real-Time Enterprise. Wiley, Hoboken (2015). https://doi.org/10.1002/9781119198697
10. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
11. Petcu, D.: Consuming resources and services from multiple clouds. J. Grid Comput. **12**(2), 321–345 (2014). https://doi.org/10.1007/s10723-013-9290-3
12. Tsai, W., Bai, X., Huang, Y.: Software-as-a-service (saas): perspectives and challenges. Sci. Chin. Inf. Sci. **57**(5), 1–15 (2014). https://doi.org/10.1007/s11432-013-5050-z
13. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009). https://doi.org/10.1145/1435417.1435432
14. Xie, R., Wen, Y., Jia, X., Xie, H.: Supporting seamless virtual machine migration via named data networking in cloud data center. IEEE Trans. Parallel Distrib. Syst. **26**(12), 3485–3497 (2015). https://doi.org/10.1109/TPDS.2014.2377119
15. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6