Databases and Information Systems X
A. Lupeikiene et al. (Eds.)

doi:10.3233/978-1-61499-941-6-95

Effectiveness of the Asynchronous Client-Side Coordination of Cluster Service Sessions

Karolis PETRAUSKAS[1] and Romas BARONAS
Institute of Computer Science, Vilnius University, Lithuania

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. An agent-based model was constructed and used to evaluate effectiveness of the proposed algorithm by performing simulations.

Keywords. session management, cluster, simulation

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 [1]. The business applications themselves are often provided as services [2]. 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 [3,4].

A lot of service providers implement the model of the eventual consistency in order to maintain high availability together with the service scalability [5]. 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 [6]. That applies also to the

[1]Corresponding author, Institute of Computer Science, Vilnius University, Didlaukio st. 47, Vilnius, Lithuania; E-mail: karolis.petrauskas@mif.vu.lt.

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 [7]. In such cases, the node availability should be 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 [7,8]. 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 [2]. 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 [10,11]. The Temporal Logic of Actions (TLA) is among such methods successfully used to describe behaviours of concurrent systems [12]. 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 [13].

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 [12,14]. 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 Section 3 and show its misbehaviour by performing model checking. Then we propose two modifications of the algorithm in Sections 3.6 and 4. In Section 5 we describe an agent-based simulation performed in order to assess effectiveness of the proposed algorithm and discuss results of the simulations.

This paper is an extended version of work published in [16]. We extend our previous work by providing a dynamic assessment of the effectiveness of the proposed algorithm and two its variants. The assessment was performed by constructing an agent-based model and performing simulations with it.

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 Figure 1.

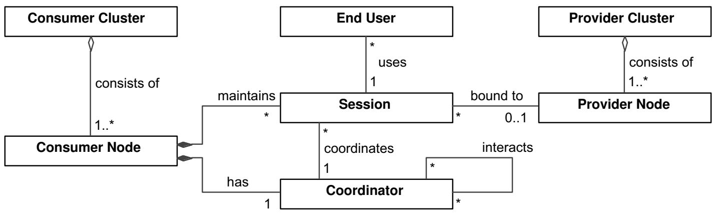

Figure 1. Principal structure of the modelled subsystem

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

The session management algorithm 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 specification:

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

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* 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),

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

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 TRUE. The action $\mathit{ProvNodeDown}(pn)$ correspondingly turns operational node down.

3.3. Behaviour of a Consumer Session

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 ($\mathit{cons}[cn].c[pn].st = \text{TRUE}$),

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

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,

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

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 $cons[cn].sm$ 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. Duplicated messages are modelled by not removing a message from the set $cons[cn].sm$ 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 bound node,

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

Receiving a message (dequeuing) is modelled by taking any message from the set of sent messages $cons[cn].s[sn].m$ ignoring their order. The set of sent messages is either left unchanged or the selected message is removed from that set.

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. The coordinator receives messages indicating failures of the provider nodes from the sessions. Then it updates its internal state ($cons[cn].c[pn].st$) and notifies all the sessions and other consumer nodes about the state changes, if some node becomes unavailable,

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

The coordinator sends notifications to other consumer nodes when some provider node becomes offline,

```
$CoordClusterMsg(cn) \triangleq$
  $\exists\, pn \in cons[cn].cm : \exists\, cm \in \{cons[cn].cm,\ cons[cn].cm \setminus \{pn\}\} :$
    $\text{LET } consDeq \triangleq [cons \text{ EXCEPT } ![cn].cm = cm]$
         $consEnq \triangleq [consDeq \text{ EXCEPT } ![cn].c[pn].st = \text{FALSE},$
           $![cn].s = [s \in \text{DOMAIN } @ \mapsto [@[s] \text{ EXCEPT } !.m = @ \cup \{[pn \mapsto pn]\}]]]$
```

$$\begin{aligned} \text{IN} \quad & \wedge\ cons' = \text{IF } cons[cn].c[pn].st \wedge \neg prov[pn] \text{ THEN } consEnq \text{ ELSE } consDeq \\ & \wedge\ \text{UNCHANGED } prov \end{aligned}$$

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 $prov[pn]$,

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

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),

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

$$(cons[cn].s[sn].pn = NA \land prov[pn]) \leadsto (cons[cn].s[sn].pn \neq NA \lor \neg prov[pn])$$

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,

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

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 $CoordProviderCheck(pn)$. This behaviour can repeat infinitely, making the consumer to consider running provider node as failed thus decreasing availability of the system.

3.6. Explicit Provider Checks

A 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 $CoordSessionMsg(cn)$ action should be changed by adding expression $cons[cn].c[msg.pn].st \land \neg prov[msg.pn]$ instead of $cons[cn].c[msg.pn].st$ in the IF condition. The changed parts of the action are as follows:

$$
\begin{aligned}
&CoordSessionMsg(cn) \triangleq \\
&\quad \ldots \\
&\quad \land cons' = \text{IF } cons[cn].c[msg.pn].st \land \neg prov[msg.pn] \text{ THEN } consUpd \text{ ELSE } consDeq \\
&\quad \land \text{UNCHANGED } prov
\end{aligned}
$$

With this change the temporal property *SessionsWillReconnect* is fulfilled.

4. 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,

$$
\begin{aligned}
&Msg \triangleq [pn : PNodes,\ gen : Nat] \\
&TypeOK \triangleq
\end{aligned}
$$

$$\begin{array}{l}
\quad \land\, prov \in [PNodes \to \text{BOOLEAN}\ ] \\
\quad \land\, cons \in [CNodes \to [\\
\qquad\quad c : [PNodes\ \to [st\ : \text{BOOLEAN}\ ,\ gen : Nat]], \\
\qquad\quad s : [SNames \to [pn : PNodes \cup \{NA\},\ gen : Nat,\ m : \text{SUBSET}\ Msg]], \\
\qquad\quad sm : \text{SUBSET}\ Msg,\ cm : \text{SUBSET}\ PNodes]]
\end{array}$$

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 $CoordClusterMsg(cn)$.

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 $CoordProviderCheck(cn, pn)$ action is changed to increment the node generation each time the coordinator detects it became available,

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

The coordinator then ignores all the messages received with old generations ($msg.gen < cons[cn].c[msg.pn].gen$) in the $CoordSessionMsg(cn)$ 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 $CoordClusterMsg(cn)$. The generation is included in the messages triggered by the session notifications in the $CoordSessionMsg(cn)$ 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 $SessionConnect(cn, sn)$ is updated to assign the generation known by the session as follows:

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

The sessions should only consider messages received from the coordinator in the $SessionUpdate(cn, sn)$ 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 $SessionReqFail(cn, sn)$ action.

5. Simulation

An agent-based simulation was performed in order to investigate behaviour of the proposed algorithm. The agent-based model [17] was implemented using the Erlang programming language [18,19,20]. This language implements an actor model, where a program is constructed as a set of communicating sequential processes. The processes communicate by exchanging asynchronous messages.

The developed simulation model consists of three types of agents: sessions and coordinators in a consumer node and the provider nodes. The agents communicate in synchronous and asynchronous ways thus implementing the proposed algorithms in a direct way.

Three variations of the session management algorithm were implemented and used in the simulations: the algorithm maintaining observed node generations (G), the algorithm with explicit node checks (C) and the algorithm where the sessions are not coordinated between each other (N).

The parameters used in all the experiments as a basis were the following:

$$n_\mathrm{c} = |CNodes| = 10, \quad n_\mathrm{p} = |PNodes| = 100, \quad n_\mathrm{s} = |SNames| = 1000.$$

These parameters describe a topology of the system. The agent-based model also includes the timing of the operations. In order to make the behaviour of the model closer to a real application, the timing parameters were randomized by introducing dispersion of their values. The following timing parameters were used as a basis in all the experiments:

$$t_\mathrm{s} = t_\mathrm{e} = 100 \pm 10\,\mathrm{ms}, \quad t_\mathrm{c} = 1000 \pm 500\,\mathrm{ms},$$

where t_s is a time consumed by a provider node to execute a request successfully [21]. A provider node takes t_e of time to respond with an error. The same parameters were used for the provider operations and the availability checks. The parameter t_c stands for a period of time while the user waits before accessing the session for the next time (the session is accessed approximately $1/t_\mathrm{c}$ times per second).

5.1. Failure Handling

In order to determine, how the variants of the algorithm tackle with the provider node faults, an experiment was carried out using the agent-based simulation. The experiments start with all the nodes up and all the sessions connected to random nodes. In the period $t \in [0, 100]$ ms 10% of the provider nodes were made inaccessible ($prov[pn] = \text{FALSE}$). The experiment was designed so that errors returned by the sessions were counted as is, without performing any retries. Results of the experiments are shown in Figure 2.

As one can see in Figure 2a, the algorithm maintaining the observed node generations (G) caused less errors to be returned by the sessions comparing to the other variations. The variation (C) caused more errors, because the provider nodes were not marked as failed while the explicit check was performed. This figure shows also, that the proposed algorithm (G) handled the errors faster.

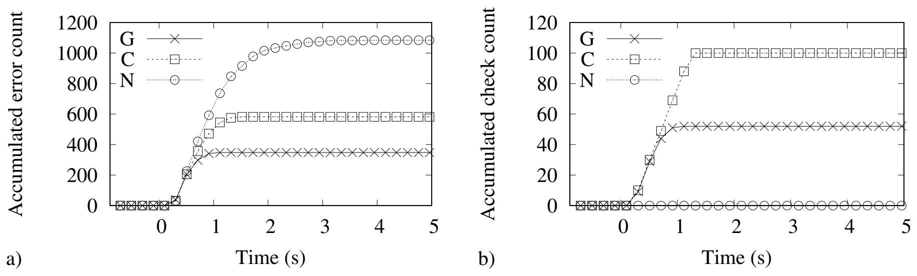

Figure 2. Behaviour of the proposed algorithm (G) and its variations (C, N) in the case of single burst of provider node failures

The error count stopped to grow after 1 second when the provider nodes were made inaccessible, while the algorithm (N) generated errors for 3 seconds after the failures started to appear.

Figure 2b shows that variant (C) being less efficient in error handling, performs twice more availability checks against the provider nodes comparing to the variant (G) of the algorithm.

5.2. Impact of the Request Rate

In order to investigate the impact of the request rate on the behaviour of the proposed algorithm, a number of simulations were performed with different rates of the user requests. The rate of the requests was varied by changing t_c (delay performed by the users between requests) from 0.01 to 16 s. With the base configuration described in Section 5, this range corresponds to the overall rate of the requests from 10^6 down to 625 operations per second. Each of the experiments was performed in the same way, as described in Section 5.1. The results of the investigation are shown in Figure 3.

Dependency of the accumulated number of errors on the request rate is shown in Figure 3a. As one can see in this figure, the variant (G) of the algorithm performs better or similar to the other variants in the investigated range of t_c. For the higher values of t_c (thus the lower rate of the requests), the algorithm with generations (G) has similar performance to the variant with explicit checks (C), through its performance is notably better for lower delays ($t_c \in [0.2, 4]$ s). Accumulated error counts are approaching $n_f = n_p \times 10\%$ when increasing t_c for both algorithms (G) and (C). In the case of the variant (N) the accumulated error count does not depend on the rate of the requests and is approximately equal to $n_s n_c / n_f$, the number of sessions bound to the failed nodes n_f. The variants (G) and (C) approach to this error count when $t_c \to 0$. That means the sessions are accessed faster than the state coordination is performed.

The impact of the request rate on the accumulated number of checks is shown in Figure 3b. The variant (N) does not perform any availability checks and therefore the count is always 0. In the case of the explicit checks (C) the number of checks equals $n_c n_f = 100$, because the check is performed for each failed node on each consumer node. The variant (G) of the algorithm uses checks only for validating notifications from other nodes. In the case of low rate (high t_c), the

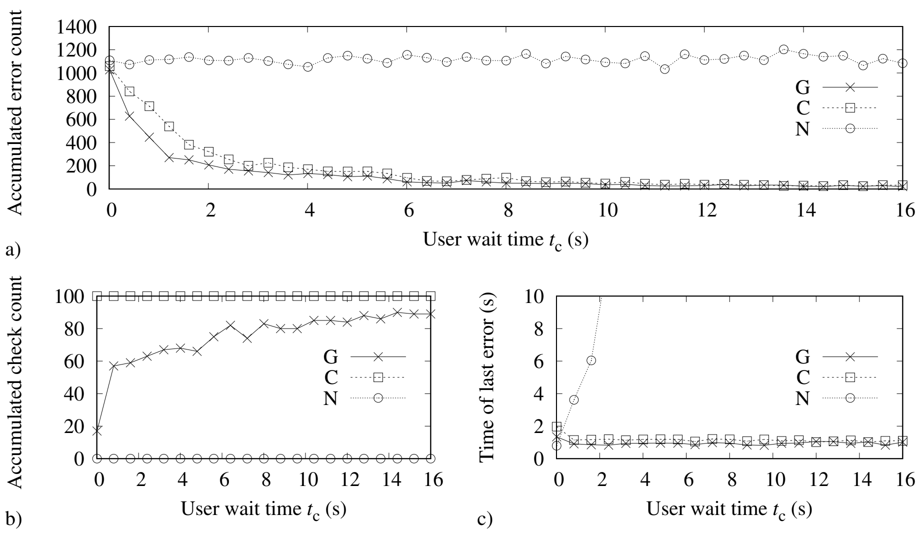

Figure 3. Behaviour of the proposed algorithm (G) and its variations (C, N) in the case of single burst of provider node failures depending on the load, $t_c \in [0.01, 16]$ s

session coordination is performed faster in the entire cluster, than the sessions are accessed. Because of this, the number of checks approaches to $(n_c - 1)n_f = 90$. At a high request rate (low t_c), the sessions are often accessed before the failure notification is received from another node (and checked) leading to less checks performed and more failures reported to the users.

Figure 3c shows impact of t_c on the time interval between the first and the last error, a period in which the users will experience consequences of a failure. In the case of non coordinated sessions (N) the delay increases linearly (higher values are not shown in this graph in order to keep other curves distinguishable). The proportion coefficient is larger than 1 here, because failures can occur also on reconnects. The variant (G) of the algorithm is always faster than the variant (C), because it performs less explicit checks. Each of the consumer nodes perform the checks sequentially because that is done by a single coordinator process. Number of checks on each node is approximately equal to n_f and each check takes t_e time to complete. In this experiment $n_f t_e = 1$ s.

5.3. Impact of the Provider Behaviour

The dependency of the algorithm performance on the the number of failing provider nodes as well as their latency is shown in Figure 4.

As one can see in Figure 4a, the accumulated number of errors increases linearly in the case of variants (C) and (G) while it grows exponentially, when sessions are not coordinated (N). The exponential increase of the errors is observed because the reconnects are performed with no knowledge of the failed nodes.

Figure 4b shows that the number of errors does not depend on the provider node latency in the case of variant (N). The variants (C) and (G) perform better when the latency is smaller, because the first error is received earlier and the

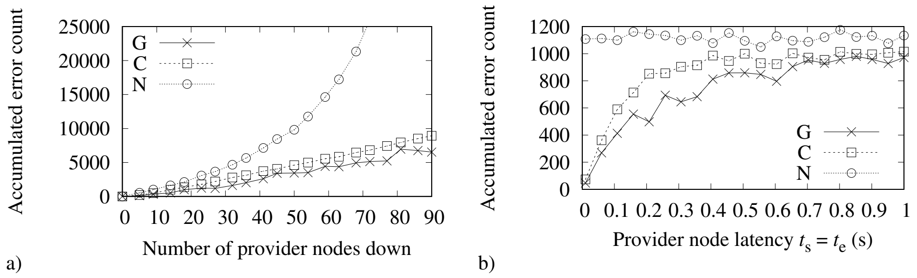

Figure 4. Behaviour of the proposed algorithm (G) and its variations (C, N) at different number of failing nodes (a) and different latency of the provider operations (b)

explicit availability checks are performed faster. The algorithm maintaining the observed node generations (G) performs better than the other variants in all the investigated cases. Its advantage is bigger at the lower latency of the provider nodes ($t < 0.7$ s in this case).

6. 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.

The agent-based simulation showed that the coordination of the sessions can decrease a number of errors exposed to the user considerably. With the configuration used in the simulation, the error count was decreased 50 times at a low rate of operations. By coordinating the sessions the duration in which the users observe consequences of a provider node failure can be changed from linear to constant function of delays between quests. At the very high rate of the user requests (10^6 with the investigated configuration), the performance of the coordinated sessions approaches the performance of uncoordinated case, because the sessions are accessed faster than the coordination is performed.

The variant of the algorithm tracking the observed node generations performs better than the variant with explicit node checks. At some parameters, it handles two times more errors and handles the errors faster. The explicit checks cause higher load on the provider nodes, comparing to the variant maintaining the observed node generations.

References

[1] D. Petcu. Consuming resources and services from multiple clouds. *Journal of Grid Computing*, 12(2):321–345, Jun 2014.

[2] W. Tsai, X. Bai and Y. Huang. Software-as-a-service (saas): perspectives and challenges. *Science China Information Sciences*, 57(5):1–15, May 2014.

[3] M. Armbrust et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, APR 2010.

[4] R. Xie et al. Supporting seamless virtual machine migration via named data networking in cloud data center. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3485–3497, Dec 2015.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[6] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[7] N. Ayari et al. Fault tolerance for highly available internet services: concepts, approaches, and issues. *IEEE Communications Surveys and Tutorials*, 10(2):34–46, 2008.

[8] D. E. Lowell, S. Chandra and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[9] D. C. Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. John Wiley & Sons, Hoboken, New Jersey, 2015.

[10] A. Hinze and A. Voisard. EVA: An event algebra supporting complex event specification. *Information Systems*, 48:1–25, 2015.

[11] D. Li et al. Network reliability analysis based on percolation theory. *Reliability Engineering & System Safety*, 142:556–562, 2015.

[12] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] C. Newcombe et al. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, March 2015.

[14] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] Y. Yu, P. Manolios and L. Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] K. Petrauskas and R. Baronas. Asynchronous client-side coordination of cluster service sessions. In *Databases and Information Systems*, pages 121–133, Cham, 2018. Springer International Publishing.

[17] C. M. Macal and M. J. North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[18] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[19] I. Ribners and G. Arnicans. Concept of client-server environment for agent-based modeling and simulation of living systems. In *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks*, pages 83–88, June 2015.

[20] W. Chun-yu et al. A framework for multi-agent-based stock market simulation on parallel environment. In *Contemporary Research on E-business Technology and Strategy*, pages 561–570, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[21] J. B. Leners et al. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.