Journal of Scientific & Industrial Research
Vol. 80, December 2021, pp. 1078-1090

Formal Verification of Receiver Initiated Load Distribution Protocol with Fault Tolerance and Recovery using Event-B

Pooja Yadav[1]*, Raghuraj Suryavanshi[2] and Divakar Yadav[3]

[1]Dr A P J Abdul Kalam Technical University, Lucknow 226 031, India

[2]Pranveer Singh Institute of Technology, Kanpur 209 305, India

[3]Institute of Engineering and Technology Lucknow 226 021, India

Received 05 May 2021; revised 12 September 2021; accepted 02 December 2021

Load distribution is a process that involves the allocation of tasks to various nodes in the distributed system in such a manner that overall resource utilization is maximized, and overall response time is minimized. This paper presents a formal model for verification of receiver-initiated load balancing and fault tolerance protocol with recovery in distributed systems using the eclipse-based Event-B platform called Rodin. Here, the receiver-initiated load balancing approach is demonstrated along with tolerance of node failure and recovery. In this approach, an underloaded node (receiver) initiates the process of load transfer from an overloaded node (sender). The underloaded node broadcasts a request message to obtain load from the overloaded nodes. The overloaded nodes reply with their load value. The underloaded node then selects the optimal overloaded node for load transfer. The chances of node failure are minimized by reducing the number of overloaded nodes. The process of recovery from failure is also shown in the proposed model. Formal methods are used to mathematically verify the critical properties of the system by developing a model based on its specifications. Our objective is to verify and validate the model for correctness through discharge of proof obligations using Event-B. Event-B is a formal method which is used for verification of a model based on distributed systems. The proof obligations generated by the model are discharged which ensures the correctness of our model.

Keywords: Distributed systems, Formal methods, Load balancing, Proof obligations, Rodin, Verification

Introduction

Distributed systems are a group of autonomous systems connected by a communication network. These autonomous systems interact through message passing. They do not have a common global clock or a shared memory. Due to complexity in the designs of distributed systems, the precise understanding and verification of distributed algorithms becomes difficult. A formal specification of distributed algorithms using mathematical techniques helps in understanding their precise behavior. This paper presents a formal development approach for receiver-initiated load balancing algorithm with fault tolerance and recovery using Event-B. Event-B is an event-based approach for formal development and verification of models based on distributed systems. The users submit their tasks to individual autonomous systems or nodes for processing. Such random submission of tasks leads to unequal load distribution at various nodes. Some of the nodes become heavily loaded which leads to performance degradation while others remain idle or underutilized. Load balancing[1] is the process of allocating tasks to various nodes in the system in such a manner that overall resource utilization is maximized, and overall response time is minimized. In distributed systems, load distribution algorithms can be classified as sender initiated or receiver initiated.[2,3] In sender-initiated algorithm[2] an overloaded node or sender identifies an underloaded node or receiver to share its load while in receiver-initiated algorithm[2,4] an underloaded node or receiver identifies an overloaded node or sender to acquire its load.

In our receiver-initiated load balancing model, the notion of threshold value is considered which indicates optimal load value of a node. Nodes having load value above the threshold value are considered as overloaded nodes while nodes having load value below the threshold value are considered as underloaded nodes. The uniqueness and novelty of the presented algorithm lies in the following factions.

- In order to transfer load from maximum number of overloaded nodes, the node having minimum

*Author for Correspondence
E-mail: poojayadav255@gmail.com

load above threshold value (i.e., minimum overloaded node) will be selected from directory. If the overloaded node having maximum load above threshold value (i.e, maximum overloaded node) is selected then lesser number of overloaded nodes will be catered to by the algorithm. However, as the load balancing process continues as per the algorithm the most overloaded node will also be addressed. Therefore, algorithm is optimized to maximize the number of nodes with optimum load i.e., load value nearing the threshold value.

- Along with load balancing, the notion of fault tolerance[3] and recovery[5] have also been added to the algorithm. In case the receiver or underloaded node fails or does not respond to messages, we choose another underloaded node from the set of underloaded nodes for receiving the load.
- Each step of the algorithm is formalized with the help of events comprising of guards and actions in order to verify the correctness of the model.

Related Work

There has been extensive research in the field of modelling techniques using formal verification and validation methods with respect to distributed systems, distributed databases, distributed transactions, checkpointing, fault tolerance etc. Suryavanshi & Yadav[6] have modelled and verified lazy replication in distributed database systems using formal techniques with the help of Event-B using Rodin platform. Chandra *et al.*[7] highlight the formal verification of distributed checkpointing, distinguishing the local and global checkpoints and marking the transactions which took place before the global checkpoint which may further be used for the purpose of recovery. The checkpointing algorithm is specified through an Event-B model using Rodin platform and its correctness is checked through discharge of proof obligations. Elnozahy *et al.*[5] gives a detailed survey of various rollback recovery protocols in message passing systems. Rigorous design of fault tolerant distributed transactions[8] has been demonstrated and verified with the help of Event-B using Click'n'Prove platform. Data is replicated across multiple sites in this distributed database and replica control measures are exercised to maintain the consistency of the distributed database. Similarly, atomicity of transactions in a cash based digital payment system is verified and validated by specifying the system mathematically using Event-B in Rodin platform through a series of refinement steps and discharge of proof obligations by Chandra & Yadav.[9] Lahbib *et al.*[10] have used Event B for the verification of safety, accuracy and correctness of smart contractions developed using block chain technology. The development of smart contractions is a well-known application of block chain technology and is used for various purposes such industry, trade and commerce, healthcare etc. Lahouij *et al.*[11] relate to the development of a formal model for the integration of various components of cloud services provided by different service providers and having different specifications. The model is verified to check the correctness of the "Cloud composite services". Karmakar *et al.*[12] demonstrate the formulization and validation of a protocol developed for conserving ground water using the Event B platform called Rodin. Event-B is used for the verification of fault tolerance mechanism in cyber physical system by Ali *et al.*[13] Le *et al.*[14] have devised a method for converting database triggers into Event-B constructs and verify their correctness by detecting the presence of infinite loops and preservation of data constraint properties. However, the use of formal methods for the development and verification of distributed load balancing algorithms is relatively unexplored. Formal specification of sender-initiated load balancing mechanism in distributed systems having causal order message delivery is specified by Yadav *et al.*[15] Formal verification of split point load balancing algorithm is presented by Shukla *et al.*[16] Here the overloaded node or sender initiates the process of load transfer. The excess load from the overloaded node is not transferred to a single underloaded node but split among two or more underloaded nodes. In this paper, we have modelled the receiver-initiated load balancing algorithm with fault tolerance property for distributed systems using Event-B. It produces proof obligations and all of them are discharged to check the correctness of the system.

Event-B: A Formal Technique

A formal technique known as Event-B is used for rigorous designing of distributed algorithms in a stepwise manner using mathematical expressions. The algorithm is first expressed as an abstraction model showcasing its basic functionality and then further refinements are modelled by adding finer details and features of the algorithm. The discharge of proof obligations verifies the correctness of the system model. Using this technique first the abstraction problem is defined, and then further details and solutions are introduced in the refinement steps to obtain more concrete specifications leading to verification of the validity of the proposed solutions. The static properties of Event-B models are defined by invariants and the dynamic properties are defined by events. Activation of an event[8] when its guards become valid modifies a list of state variables. Each refinement step strengthens the guards of events.

Contexts and machines are the two major components of an Event-B model.[17,18] Contexts, which consist of constants, sets and axioms and form the static part of the model.[19] Variables define the state of the machine. Invariants are the constraints which are applied to the machine's variables. When state change in a machine occurs during execution, the invariants of the machine which define the properties of those variables must not be violated. All the invariants must be satisfied by each state of the machine.[20] If there is a violation of invariants, it means the model is not working correctly as per the specifications. An event comprises of guards and actions. The guards represent the necessary conditions for the events to occur. When all the guards of an event become true, its list of actions is enabled. An action assigns of new values to variables. Prove obligations can be discharged through interaction or by using automatic prover.[21] The detailed syntax and description of Event-B notations are given by Abrial.[22]

Some of the Event-B tools are B-Toolkit[23], Rodin[21,24], Atelier B[25] Click'n'Prove.[26] Rodin platform[21] is used for this research work. Metayer[21] states that "Rodin platform is an open extensible tool for specification and verification of Event-B models". Modeling elements such as variables, invariants, events and components like context and machines are available in Rodin. Rodin provides a platform for refinement checking and consistency checking through generation and discharge of proof obligations.

System Model

The receiver-initiated algorithm ensures load balancing in a synchronized manner. The communication network in the system is assumed to be reliable. In this algorithm, the process of load balancing and redistribution is initiated by the receiver which is an underloaded node as shown in Fig. 1. Each node maintains its status as underloaded or overloaded which is defined by a threshold value. If the load value of the node is less than or equal to

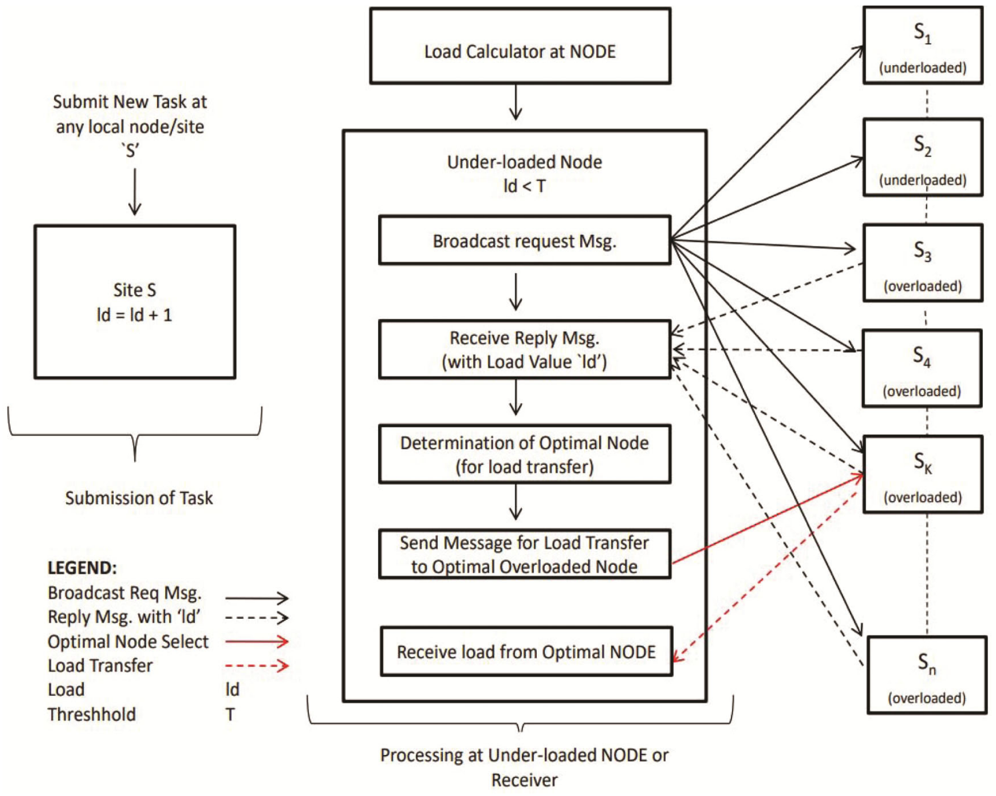

Fig. 1 — Proposed model for receiver-initiated load balancing algorithm

the threshold value, the node is declared as "underloaded node", otherwise the category of the node is "overloaded node". The receiver or underloaded node broadcasts the request message to all other nodes. After receiving the request message, only the overloaded nodes (senders) reply with a message containing their load value. The underloaded node receives the reply messages with load values from all overloaded nodes and selects the optimal overloaded node for load transfer. The optimal overloaded node is the minimal overloaded node which has the least load value above threshold. This approach is chosen to balance the load at maximum possible overloaded nodes. The underloaded node then informs the minimal or optimal overloaded node that it has been selected for load transfer and the activity of load transfer takes place.A brief informal description of the events is given below:

a) **Identify the Underloaded and Overloaded Nodes:** When a new task is submitted at any process, the load value of that process is incremented. The node compares this load value with the threshold value. A node with the load value greater than the threshold value is declared as *"overloaded node"*, otherwise the category of the node is *"underloaded node"*.
b) **Underloaded (Receiver) Node Broadcasts the Request Message:** If the load value of a node is less than the threshold value i.e., it is an underloaded node then a request message is broadcast to find an overloaded node for load transfer. Since the receiver of load or underloaded node initiates the algorithm, it is called the receiver-initiated load balancing algorithm.
c) **Delivery of Request Message and Reply to the Underloaded (Receiver) Node:** After the request is delivered to all the nodes, they will send a reply message to the underloaded node. The reply message contains node id and its corresponding load value.
d) **Receiving the Reply Message and Updating the Directory:** The underloaded node or receiver receives the reply message from other nodes and updates its directory with each node and its corresponding load value.
e) **Find the Least Overloaded Node for Transfer of Load:** Once all the nodes have sent their load value, we find the least overloaded node i.e., the node whose load value is closest to the threshold value (only slightly greater than the threshold value). The underloaded node sends a message to the least overloaded node for transferring the load. This approach is chosen to balance the load at maximum possible overloaded nodes.
f) **Receiving the message for load transfer:** The least overloaded node receives the load transfer message and prepares for transferring the load.
g) **Checking the Status of the Underloaded Node:** After receiving the load transfer message the least overloaded node checks the status of underloaded or receiver node i.e. *"ready to receive"* or *"failed"*.
h) **Transfer of Load from the Overloaded Node to the Underloaded Node:** In order to transfer load, status of underloaded node must be *"ready to receive"*. Further, we compute the excess load value at the overloaded node (i.e., load value of the overloaded node - threshold value) which must be transferred to the underloaded node. It is ensured that the total load value at the receiver node does not exceed the threshold value after load transfer. The received load value is added to the already existing load value of the underloaded node and the transfer status is set to *"completed"*.
i) **Failure of Underloaded Node:** If the status of the underloaded node is *"failed"*, we choose another underloaded node from the set of underloaded nodes which is different from the previous one.
j) **Node Failure/Fault Tolerance:** After choosing another underloaded node whose status is "*ready to receive*", we transfer the load as per the policy discussed above. The fault is tolerated in this manner, and the status of load transfer is set as *"completed"*.

Formal Modelling of Receiver Initiated Load Distribution Protocol

The Event-B model for receiver-initiated load balancing and fault tolerance algorithm contains a context and a machine having several events. Every event consists of guards and actions. The invariants define the properties of the variables that should not be violated,[13] when the state of the machine changes during execution. In this Event-B model, *MESSAGE* and *NODE* are declared as carrier sets. The sets *status, m_status, type, load_progress* and *node_status* are defined as enumerated sets. The machine part of the model, justifies the variables, invariants, and events whose detailed discussion is as follows:

Invariant 1: sender $\in$ MESSAGE $\mapsto$NODE
Invariant2: deliver $\in$ NODE $\leftrightarrow$ MESSAGE
Invariant3: node_status$\in$ NODE $\rightarrow$ status

Invariant4: load_value∈ NODE → N

Invariant 5: reply_msg_send∈(MESSAGE↔ MESSAGE) ↦NODE

Invariant6: reply_msg_rcd∈ NODE ↔ (MESSAGE ↔ MESSAGE)

Invariant7: msg_send⊆ MESSAGE

Invariant8: dir∈ NODE ↔ (MESSAGE ↦N)

Invariant1 defines the variable sender is as a partial function from the set *MESSAGE* to the set *NODE*. It models the sending of message *m* by node *n*. *Invariant2* defines the variable *deliver* as the relation *(mm ↦nn)* ∈ *deliver* which represents the successful delivery of request message *mm* at the overloaded node *nn*. *Invariant3* identifies the status of the node as defined in the context part. The status of the node is either overloaded or underloaded. In *invariant4*, the variable *load_value* is defined as the value of load at each node which must be a natural number. In *invariant5*, the variable *reply_msg_send* is represented as a mapping function *({mm ↦ m}) ↦nn∈reply_msg_send.* The reply message *m* is sent in context to the request message *mm* by the overloaded node *nn*. In *invariant6*, the variable *reply_msg_rcd* shows the receiving of the reply message by the underloaded node and is represented by the mapping function *n ↦ ({mm ↦ m}) ∈reply_msg_rcd.* The reply message *m*, sent by the overloaded node *nn* is received by underloaded node *n* which was sent as a reply to the request message *mm*. *Invariant7* defines the variable *msg_send* as a subset of the set MESSAGE, which identifies the messages sent by a node. In *invariant8*, the variable *dir* represents the directory which contains the node ID and their corresponding load value, which is a natural number. Here, the mapping between the node and the message is also represented.

Invariant9:message_type∈msg_send → type

Invariant10:req_nodes⊆ NODE

Invariant11:request_queue∈ NODE ↔ (MESSAGE ↦NODE)

Invariant12:message_node_value∈ MESSAGE ↦N

Invariant13:reply_node∈ NODE ↔ (MESSAGE ↦ N)

Invariant14:transfer_load∈ NODE → load_progress

Invariant15:message_status∈ MESSAGE → m_status

Invariant16:n_status∈ NODE → node_status

Invariant17:load_val⊆ N

Invariant18:min_load_msg⊆ MESSAGE

Invariant19: load ∈ MESSAGE ↦N

Invariant20:underloaded_nodes⊆ NODE

Invariant21:load_balancer⊆ NODE

In *invariant 9*, the variable *message_type* represents the category of a message. A message can be categorized as either "*request*", "*reply*" or "*minimum load*". *Invariant10* defines the variable *req_nodes* as a subset of the set *NODE*. In *invariant11*, the variable *request_queue* is represented by the set of relations between the set *NODE* and request messages sent by the corresponding nodes. A mapping represented as *n ↦ {mm ↦ n}} ∈request_queue* indicates that the request queue of node *n* has a request message *mm* sent by node *n*. In *invariant12*, the variable *message_node_value* specifies the mapping of a message with the load value of its sender, which is a natural number. *Invariant13* defines the variable *reply_node* which represents those nodes that reply to the corresponding request message. In *invariant14*, The variable *transfer_load* shows the status of load transfer from the overloaded node to the underloaded node. The progress of load transfer can be categorized as "*ready to send*", "*completed*" or "*failed*". In *invariant15,* the variable *message_status* is defined as the status of the delivery of the message which can be "*pending*" or "*successful*". In *invariant16*, the variable *n_status* shows the status of the node which can be either "*active*" or "*expired*". As per *invariant17*, the variable *load_val* is a strict subset of the set of natural numbers *N*. *Invariant18* defines the variable *min_load_msg* is a subset of the set *MESSAGE*. This message contains the node ID of the minimum overloaded node. Variable *load* belongs to the set of natural numbers *N* as per *invariant19*. *Invariant20* defines the set *underloaded_nodes* as a subset of the set *NODE*. It contains the set of underloaded nodes. *Invariant21* defines the set *load_balancer* as a set of nodes which belong to the set *underloaded_nodes* and participate in load balancing. It is a subset of the set *NODE*. Each step of the receiver-initiated load balancing protocol is modelled as events comprising of guards and actions as described below.

Event 1: Submission of Request: The event *Request submission* is given below. This event models the submission of a new request at any node *n*. The load value of node *n* is represented by the variable *load_value(n)*. When a task is submitted at a node, then the increase in the *load_value* of the node is modelled as an increment by one *(action1).*

EVENT *Request submission*
ANY *n*
WHERE
guard1: *n ∈NODE*
THEN
action1:*load_value(n) := load_value(n) + 1*

Events 2 & 3: Decision of Node Status: The event *Underload* is given below. In this event, the load value of node *n* is compared with the threshold value which is a constant. If the load value is less than or equal to the threshold value *(guard4)* the status of the node is set as *"underloaded" (action1).*

The event *Overload* is also given below. The load value of anode is compared with the threshold value. If it exceeds the threshold value *(guard4),* then the status of the node is declared as "*overloaded*" (*action1*).

EVENT*Underload*
ANY *n*
WHERE
guard1: *n ∈NODE*
guard2: *n ∈dom(load_value)*
guard3:*load_value(n) ∈N*
guard4:*load_value(n) ≤ threshold*
THEN
action1:*node_status(n) := underloaded*
EVENT*Overload*
ANY *n*
WHERE
guard1: *n ∈NODE*
guard2: *n ∈dom(load_value)*
guard3: *load_value(n) ∈N*
guard4: *load_value(n) > threshold*
THEN
action1:*node_status(n) := overloaded*

Event 4: Broadcast of Request Message from the Underloaded Node: In a receiver-initiated algorithm, the receiver node or the underloaded node broadcasts the request message to find the overloaded node. The event *Underloaded node broadcast* is modelled as shown below. Node *n* belongs to the set *NODE (guard1).* The status of node *n* is "*underloaded*" *(guard3).* Message *mm* has not yet been sent *(guard4)* and message *mm* is not yet in the domain of sender node is ensured *(guard5).* Message *mm* does not belong to the request queue of the underloaded node *n (guard6)* and node *n* is not in the set *req_nodes (guard7).*

EVENT*Broadcast by underloaded node*
ANY *n, mm*
WHERE
guard1: *n ∈NODE*
guard2: *mm ∈MESSAGE*
guard3:*node_status(n) = underloaded*
guard4: *mm ∉msg_send*
guard5: *mm ∉dom(sender)*
guard6: *mm ↦n ∉request_queue[n]*
guard7: *n ∉req_nodes*

In the action part, *action1* ensures that message *mm* is sent by node *n*. Sending of message *mm* is ensured by *action3*. The type of message *mm* is *"request" (action4).* Node *n* belongs to the set *req_nodes (action5).*

action1:*sender := sender ∪{mm ↦ n}*
action2:*request_queue := request_queue∪{n ↦ {mm ↦ n}}*
action3:*msg_send := msg_send∪{mm}*
action4:*message_type(mm) := request*
action5:*req_nodes := req_nodes∪n*

Event 5: Delivery of Request Message: The delivery of the request message which was broadcast by the underloaded node *n* is shown below. Node *n* is the requesting node. The status of node *n* is "*underloaded*" *(guard3).* Message *mm* is a request message broadcast by the underloaded node *n* (*guard5*). Message *mm* which is sent by node *n*, does not belong to the request queue of the overloaded node *nn(guard6).* The type of message *mm* is "*request*" *(guard7).* The status of the node *nn* is overloaded *(guard9)* and message *mm* is not delivered at node *nn (guard10).*

EVENT*Delivery of request message*
ANY *mm, n, nn*
WHERE
guard1: *n ∈req_nodes*
guard2: *n ∈underloaded_nodes*
guard3:*node_status(n) = underloaded*
guard4: *mm ∈msg_send*
guard5: *(mm ↦n) ∈sender*
guard6: *(mm ↦ n) ∉request_queue[nn]*
guard7:*message_type(mm) = request*
guard8:*nn∈NODE*
guard9:*node_status(nn) = overloaded*
guard10: *(nn↦ mm) ∉ deliver*

If all the guards are valid, then message *mm* is delivered at node *nn (action1).* Delivery of message *mm* is shown in the request queue of the overloaded node *nn (action2).*

action1:*deliver := deliver ∪{nn↦ mm}*
action2:*request_queue := request queue ∪{nn↦ {mm ↦ n}}*

Event 6: Reply from Overloaded Node to the Underloaded Node: All the overloaded nodes reply to the request message broadcast by the underloaded node *n* with *"ready to transfer load"* message. The event *Reply from overloaded node* is shown below.

Message *mm* belongs to *msg_send* and *msg_send* is a subset of the set *MESSAGE*. It contains those messages which are to be sent by the node either in the form of request or reply *(guard3).* Status of node *n* and *nn* is "*underloaded"* and "*overloaded"* respectively *(guard4 & guard5).* We have successfully delivered the message *mm* in node *nn (guard7).* The overloaded nodes have load values which are natural numbers *(guard8 & guard9).* Message *m* does not yet belong to the domain of *msg_send (guard12)*, the type of message is not yet decided *(guard13)* and message *m* does not yet belong to the domain of *reply_msg_send (guard14)*. Message *m* does not belong to the domain of the *sender (guard15)*. Reply message *m* with load value *ld* does not belong to the set *load (guard16)*.

EVENT *Reply from overloaded node*
ANY *n, mm, nn, m, ld*
WHERE
guard1: $n \in NODE$
guard2: $nn \in NODE$
guard3: $mm \in msg\_send$
guard4: $node\_status(n) = underloaded$
guard5: $nod\_status(nn) = overloaded$
guard6: $message\_type(mm) = request$
guard7: $(nn \mapsto mm) \in deliver$
guard8: $ld = load\_value(nn) - threshold$
guard9: $ld \in N$
guard10: $(mm \mapsto n) \in sender$
guard11: $m \in MESSAGE$
guard12: $m \notin msg\_send$
guard13: $m \notin dom(message\_type)$
guard14: $mm \mapsto m \notin dom(reply\_msg\_send)$
guard15: $m \notin dom(sender)$
guard16: $(m \mapsto ld) \notin load$

In the action module, *action1* shows that message *m* belongs to the domain of *msg_send* and message type is a "*reply" (action2)*. Reply message *m* is sent by node *nn* successfully *(action3)*. The load value of node *nn* is sent with reply message *m* and the variable *message_node_value* is updated successfully *(action4)*. Message *m* is sent from overloaded node *nn* to underloaded node *n (action5)*. Message *m* with load value *ld* now belongs to the set *load (action6)* and node *n* which is an underloaded node is the *load_balancer (action7)*.

action1 : $msg\_send := msg\_send \cup \{m\}$
action2 : $message\_type(m) := reply$
action3 : $reply\_msg\_send := reply\_msg\_send \cup \{(mm \mapsto m) \mapsto nn\}$
action4: $message\_node\_value := message\_node\_value \cup (m \mapsto ld)$
action5 : $sender := sender \cup (m \mapsto nn)$
action6 : $load := load \cup (m \mapsto ld)$
action7 : $load\_balancer := \{n\}$

Event 7: Receiving of Reply Message: Reply message is sent by overloaded nodes to the underloaded node. We ensure that only overloaded nodes participate in the load transfer activity, otherwise our system will remain busy unnecessarily and the network traffic congestion will increase.

In the event given below, node *n* is a *load_balancer* and node *nn* belongs to *req_nodes (guard1 & 2)* respectively. Message *mm* is a request message is ensured by *guard4*. Status of node *nn* and *n* is "*overloaded"* and "*underloaded"* respectively *(guard5 & 6)*. The overloaded node *nn* sends the reply message *m* to node *n (guard10)*. As per the request queue, message *mm* is sent by the underloaded node *n (guard11)*. The variable *ld* defines the load value at a node and it is a natural number (*guard12 & 13*). Message *m* is not yet delivered to node *n* (*guard15*). Message *m* is not present in the request queue of node *nn (guard14)*. The directory is not updated with the load value *ld* and corresponding overloaded node *nn* is ensured by *guard17*. The variable *reply_node[{nn}]* is not updated with the reply message *m* and load variable *ld (guard18)*.

EVENT *Receive reply message*
ANY *n, mm, nn, m, ld*
WHERE
guard1: $\{n\} = load\_balancer$
guard2: $nn \in req\_nodes$
guard3: $mm \in msg\_send$
guard4: $message\_type(mm) = request$
guard5: $node\_status(n) = underloaded$
guard6: $node\_status(nn) = overloaded$
guard7: $(mm \mapsto n) \in sender$
guard8: $m \in msg\_send$
guard9: $message\_type(m) = reply$
guard10: $(mm \mapsto m) \mapsto nn \in reply\_msg\_send$
guard11: $(mm \mapsto n) \in request\_queue[n]$
guard12: $ld = load\_value(nn) - threshold$
guard13: $ld \in N$
guard14: $(m \mapsto nn) \notin request\_queue[n]$
guard15: $(n \mapsto m) \notin deliver$
guard16: $(m \mapsto ld) \in load$
guard17: $(m \mapsto ld) \notin dir[n]$
guard18: $(m \mapsto ld) \notin reply\_node[nn]$

In the action part, *action1* represents the delivery of message *m* at node *n* successfully. The variable *reply_msg_rcd* is updated with the reply message *m* corresponding to the request message *mm* at node *n (action2)*. The directory is updated with the message and the load values *(action3)*. The request queue of node *n* is updated with the reply message *m (action4)*. The variable *reply_node[nn]* also has an entry of message *m* and load value *ld* in its queue *(action5)*. The variable *load_val* is the set of load values received from the overloaded nodes. The load value *ld* received through message *m* is added to the set *load_val (action6)*.

action1: $deliver := deliver \cup \{n \mapsto m\}$
action2: $reply\_msg\_rcd := reply\_msg\_rcd \cup \{n \mapsto (mm \mapsto m)\}$
action3: $dir := dir \cup \{n \mapsto (m \mapsto ld)\}$
action4: $request\_queue := request\_queue \cup \{n \mapsto (m \mapsto nn)\}$
action5: $reply\_node := reply\_node \cup \{nn \mapsto (m \mapsto ld)\}$
action6: $load\_val := load\_val \cup \{ld\}$

Event 8: Finding the Minimum Load: After receiving the load value *ld* from the overloaded nodes, we choose the node with minimum load value from among the replying nodes. The *find minimum load* event is demonstrated below.

Node *nn* is in the set *req_nodes*, node *n* is the *load_balancer (guard1& 2)* and message *mm* belongs to the *msg_send* list is ensured by *guard3*. The type of message *mm* is "*request*" and the status of node *n* and node *nn* is "*underloaded"* and "*overloaded"* respectively *(guard5 & 6)*. The type of message *m* is "*reply*" *(guard7 & 8)*. Reply message is received by node *n* which was sent by node *nn* in the form of message *m* corresponding to the request message *mm (guard9)*. The minimum load must be a natural number *(guard10)*. The variable *ld* belongs to the set *load_val* and we compare the load values with each other in the set *load_val*. We take one load value and compare it with each value in the set, if none of the load value is less than the selected load value then we send the load in the set *min_load (guard13, 14 & 15)*. Node *nn* sends the reply message *m* with *min_load* value towards the underloaded nodes is ensured by the guard *(guard16)*. In *guard 17*, we consider all the nodes which are overloaded *nk* with reply message *mk* corresponding to the request message *mm* and the load value of every node is received. This implies that all the overloaded nodes have replied to the request message broadcast by the underloaded node. Message *ml* is not in the domain of node *nn* and the reply of *min_load* message is not yet sent (*guard18 &19*).

EVENT*Find minimum overloaded node*
ANY *n, nn, m, mm, ld, min load, ml*
WHERE
guard1: *{n} = load_balancer*
guard2:*nn∈req_nodes*
guard3: *mm ∈msg_send*
guard4:*message_type(mm) = request*
guard5:*node_status(n) = underloaded*
guard6:*node_status(nn) = overloaded*
guard7: *m ∈msg_send*
guard8:*message_type(m) = reply*
guard9: *(n ↦ (mm ↦ m)) ∈reply_msg_rcd*
guard10:*min_load∈ N*
guard11:*ld∈load_val*
guard12: *(n ↦ (m ↦ld)) ∈dir*
guard13: *∀l · (l ∈load_val) ⇒min_load ≤ l*
guard14:*min_load = min(load_val∪ {0})*
guard15:*min_load∈load_val*
guard16: *(nn↦ (m ↦min_load)) ∈reply_node*
guard17: *∀nk, mk, ld1 · (nk∈ NODE ∧node_status(nk) = overloaded ∧mk∈ MESSAGE ∧mk∈dom(message_type) ∧message_type(mk) = reply ∧ ld1 ∈load_val⇒ (nk↦ (mk↦ ld1)) ∈reply_node)*
guard18: *ml ∉msg_send*
guard19: *(m ↦ ml) ∉dom(reply_msg_send)*

If all the guards are valid, then the actions should be implemented. Message type of *ml* is "*minimum load*" (*action1*). Reply message with minimum load *ml* is sent by node *n* in response to the message *m* from node *nn* (*action2*). Message *ml* is now in the set *msg_send (action3)*.

action1:*message_type(ml) := minimum_load*
action2:*reply_msg_send := reply_msg_send∪ ((m ↦ ml) ↦nn)*
action3:*msg_send := msg_send∪ {ml}*

Event 9: Receiving of Minimum Load Message: When the load value of all the overloaded nodes has reached the underloaded node, then the minimum load value is selected from it. Then a message is sent to the minimum overloaded node. When this message of the minimum load is received, the minimum overloaded node is ready to transfer the load.

The event given below shows the receiving of minimum load message *ml* (*guard10*). The message type of *ml* is "*minimum_load" (guard11)*. The variable *reply_msg_send* sets the message *ml* corresponding to the message *m* in node *nn (guard12*). The reply message *ml* has not yet been received by the node *nn (guard13)*.

EVENT*Receive minimum load message*
ANY *m, mm, ml, nn, n*
WHERE
guard1: *{n} = load_balancer*
guard2:*nn∈NODE*
guard3: *mm ∈msg_send*
guard4:*message_type(mm) = request*
guard5: *m ∈msg_send*
guard6:*message_type(m) = reply*
guard7:*node_status(n) = underloaded*
guard8:*node_status(nn) = overloaded*
guard9: *ml ∈min_load_msg*
guard10: *ml ∈dom(message_type)*
guard11:*message_type(ml) = minimum_load*
guard12: *((m ↦ ml) ↦nn) ∈reply_msg_send*
guard13: *(nn↦ (m ↦ ml)) ∉reply_msg_rcd*

Finally, the reply message is received successfully at the overloaded node which has the minimum load value *(action1)*.

action1:*reply_msg_rcd := reply_msg_rcd∪ (nn↦ (m ↦ml))*

Event 10: Checking the Status of the Underloaded Node: Status of an underloaded node is checked by the overloaded node. The underloaded nodes may be in *"active"* or *"expired"* state. The overloaded node sends the *"ready to send"* message to the underloaded node. If the underloaded node is *"active"* then this *"ready to send"* message is further processed. If the underloaded node is in an expired state, then overloaded node converts the *transfer_load(nn)* status from *"ready to send"* to *"failed"*.

In the event given below, node *n* is the *load_balancer*, *n* is an underloaded node and *n_status* of node *n* is *"active"* is ensured by the guards *(guard1,2 &3)*. At a time only one node is the *load_balancer*. Node *nn* is an overloaded node and exists in the domain of *transfer_load* of node *nn*. The overloaded node *nn* is not in *"ready to send"* state is ensured by the guards *(guard5, 6 &7)*.

EVENT*Check status of underloaded node*
ANY *n, nn*
WHERE
guard1: *{n} = load_balancer*
guard2:*n_status(n) = active*
guard3:*node_status(n) = underloaded*
guard4:*nn ∈ NODE*
guard5:*node_status(nn) = overloaded*
guard6:*nn ∈ dom(transfer_load)*
guard7:*transfer_load(nn) ≠ ready_to_send*

If all the guards are true, then the *transfer_load* of node *nn* is in the *"ready to send"* state.

action1:*transfer_load(nn) := ready_to_send*

Event 11: Transfer of Load from the Overloaded Node: After checking the status of the underloaded node, the load is transferred from the overloaded node which received the minimum load message *ml* from the underloaded node. The transferred load value *ld* is subtracted from the total load value of the overloaded node.

In the event given below, the load is transferred from the overloaded node to the underloaded node. Status of node *nn* and node *n* is "*overloaded"* and "*underloaded"* respectively *(guard4 &7)*. Message *m* is delivered at node *n* successfully *(guard5)*. Load *ld* consists of the load value of node *n* minus the threshold value and the value of *ld* must be a natural number *(guard8 & 9)*. Node *nn* sends the reply message *m* with the load *ld* (*guard10*). *Min_load* is a set that contains a natural number (*guard11*). Message *ml* is a reply message which is a minimum load message *(guard12)*. The reply message *ml* corresponding to the message *m* is received at node *nn (guard13)*. Node *nn* is in the domain of *transfer_load* and the *transfer_load* status of node *nn* is *"ready to send" (guard14&15)*.

EVENT *Load transfer from overloaded node*
ANY *m, mm, nn, n, ml, min load, ld*
WHERE
guard1: *m ∈ msg_send*
guard2: *mm ∈ msg_send*
guard3: *{n} = load_balancer*
guard4:*node_status(n) = underloaded*
guard5: *(n ↦ m) ∈ deliver*
guard6:*nn ∈ NODE*
guard7:*node_status(nn) = overloaded*
guard8:*load_value(n) ∈ N*
guard9:*ld = load_value(nn) − threshold*
guard10: *(nn ↦ (m ↦ ld)) ∈ reply_node*
guard11:*min_load ∈ N*
guard12: *ml ∈ min_load_msg*
guard13: *(nn ↦ (m ↦ ml)) ∈ reply_msg_rcd*
guard14:*nn ∈ dom(transfer_load)*
guard15:*transfer_load(nn) = ready_to_send*

If all the guards stand valid, then the action module is executed, and the load transfer status is *"completed"*. The load value of the overloaded node *nn* is decreased by *ld* as ensured by (*action1* & *2*).

action1:*load_value(nn) := load_value(nn) − ld*
action2:*transfer_load(nn) := completed*

Event 12: Receiving of Load by the Underloaded Node: Load value *ld* is received by the underloaded node *n* and the load value of *n* is increased by *ld*. After receiving the load, the status of *transfer_load* is set as *"completed"*.

According to the event given below, node *nn* is present in the domain of *transfer_load (guard3)*. Load *ld* is total load value of *nn* minus the threshold value *(guard5)*. Message *m* is in the set *msg_send (guard6)* and message *ml* is in the set *min_load_msg (guard7)*. We ensure that the load value of node *n* plus *ld* is less than the threshold value *(guard9)*. Status of underloaded node *n* is *"active"* is ensured by the guard *(guard11)*.

EVENT*Load received by underloaded node*
ANY *n, nn, m, ml, ld*
WHERE
guard1: *{n} = load_balancer*
guard2:*nn ∈ NODE*
guard3:*nn ∈ dom(transfer_load)*
guard4:*transfer_load(nn) = completed*
guard5:*ld = load_value(nn)*
guard6: *m ∈ msg_send*
guard7: *ml ∈ min_load_msg*
guard8:*load_value(n) ∈ N*
guard9:*load_value(n) + ld ≤ threshold*
guard10:*load_value(nn) > threshold*
guard11:*n_status(n) = active*

If all the guards stand valid, then the load value of the underloaded node *n* is increased by *ld* and the status of *transfer_load* of node *n* is set to *"completed" (action1 & 2)*.

action1:*load_value(n) := load_value(n) + ld*
action2:*transfer_load(n) := completed*

Event 13: Failure of the Underloaded Node: The formal specification of the node failure event is given below. In this event, we have modelled the failure of underloaded node. According to this event, *n_status* of the underloaded node *n* is *"expired" (guard5)*. The overloaded node *nn* is in the domain of *transfer_load* and the status of *transfer_load* of node *nn* is *"ready to send"* is ensured by the guards *(guard6 & 7)*.

EVENT*Failure of underloaded node*
ANY *n, nn*
WHERE
guard1: *{n} = load_balancer*
guard2: *nn ∈ NODE*
guard3: *node_status(n) = underloaded*
guard4: *node_status(nn) = overloaded*
guard5: *n_status(n) = expired*
guard6: *nn ∈ dom(transfer_load)*
guard7: *transfer_load(nn) = ready_to_send*

Due to the *"expired"* status of the underloaded node *n*, the *transfer_load* status of the overloaded node *nn* is set as *"failed" (action1).*

action1: *transfer_load(nn) := failed*

Event 14: Selection of Another Underloaded Node: After the failure of the underloaded node *n* which is the *load_balancer*, another underloaded node *ud* is selected as the *load_balancer*. In the event given below, node *n* is in the set *load_balancer* and it is underloaded *(guard1 &2)*. Node *nn* is overloaded and the status of *transfer_load* at node *nn* is *"failed"* due to the *"expired"* status of node *n*. Node *ud* is in the set *NODE* and the status of *ud* is *"underloaded" (guard7 & 8).* We ensure that node *n* with the status *"expired"* is not selected again for receiving the load *(guard9)*.

EVENT *Select another underloaded node*
ANY *n, ud, nn*
WHERE
guard1: *{n} = load_balancer*
guard2: *node_status(n) = underloaded*
guard3: *nn ∈ NODE*
guard4: *node_status(nn) = overloaded*
guard5: *nn ∈ dom(transfer_load)*
guard6: *transfer_load(nn) = failed*
guard7: *ud ∈ underloaded_nodes*
guard8: *node_status(ud) = underloaded*
guard9: *ud ≠ n*

Now, *action1* removes the underloaded node *n* as the *load_balancer* and updates with new node *ud* as the *load_balancer*.

action1: *load_balancer := {ud}*

Event 15: Tolerance of Node Failure: A node is said to be"failed" when it stops responding to messages. When the *transfer_load* status at the overloaded node is *"failed"* and the underloaded node *n* does not respond to messages within time, it is assumed that the underloaded node has failed and therefore it is not able to receive load from the overloaded node. Now, the fault tolerance approach is formalized. The system selects a new underloaded node *ud* for load transfer. Load from the overloaded node *nn* is transferred successfully to the new underloaded node *ud* which is the new *load_balancer*.

In the event given below, another underloaded node *ud* is selected and the status of the node *ud* is *"underloaded" (guard1 &2).* Status of previous load transfer from overloaded node *nn* is *"failed"* is ensured by (*guard6*). Reply message *m* corresponding to the request message *mm* is not received at the node *ud (guard11).* Node *ud* is in the domain of *transfer_load* and its status is *"ready to send" (guard13).* Status of node *ud* is *"active"* and *message_status* is *"pending" (guard14 &15).*

EVENT*Tolerance of node failure*
ANY *mm, nn, m, ud*
WHERE
guard1: *{ud} = load_balancer*
guard2: *node_status(ud) = underloaded*
guard3: *nn ∈ NODE*
guard4: *node_status(nn) = overloaded*
guard5: *nn ∈ dom(transfer_load)*
guard6: *transfer_load(nn) = failed*
guard7: *mm ∈ msg_send*
guard8: *message_type(mm) = request*
guard9: *m ∈ msg_send*
guard10: *message_type(m) = reply*
guard11: *(ud ↦ (mm ↦ m)) ∉ reply_msg_rcd*
guard12: *ud ∈ dom(transfer_load)*
guard13: *transfer_load(ud) = ready_to_send*
guard14: *n_status(ud) = active*
guard15: *message_status(m) = pending*

If all the guards are valid, then *action1* sets the *transfer_load* status of node *ud* as *"completed".* Reply message *m* is delivered successfully at node *ud (action2).* Status of the message is changed from *"pending"* to *"successful" (action3).*

action1: *transfer_load(ud) := completed*
action2: *reply_msg_rcd := reply_msg_rcd ∪ (ud ↦ (mm ↦ m))*
action3: *message_status(m) := successful*

Event 16 & 17: Change the Status of a Node from "Active" to "Expired" State and Vice Versa: When an underloaded node becomes faulty, its status is changed from *"active"* to *"expired"*. In the event given below, node *n* is the *load_balancer* and its status is "*underloaded" (guard2 & 3).* The *n_status* of node *n* is *"active" (guard4).*

EVENT *Change status of underloaded node from "active" to "expired"*
ANY *nn, n*
WHERE
guard1: *nn ∈ NODE*
guard2: *{n} = load_balancer*
guard3: *node_status(n) = underloaded*
guard4: *n_status(n) = active*

Due to the occurrence of this event, the status of the underloaded node is changed from *"active"* to *"expired"*.

action1: *n_status(n) := expired*

After the recovery of an underloaded node, the status of the node is set from *"expired"* to *"active"* so that it can again participate in the process of load balancing. In the event given below, node *n* is the *load_balancer* and its status is "*underloaded"* (*guard2 & 3*). The *n_status* of node *n* is *"expired" (guard4)*. In this state, node n cannot send or receive any message.

EVENT*Change status of underloaded node from "expired" to "active"*
ANY *nn, n*
WHERE
guard1: $nn \in NODE$
guard2: $\{n\} = load\_balancer$
guard3: $node\_status(n) = underloaded$
guard4: $n\_status(n) = expired$

In *action1*, the *n_status* of node *n* is set as *"active"* so that it can again participate in the process of load balancing.

action1: $n\_status(n) := active$

Results and Discussion

The above model of receiver-initiated load balancing algorithm with fault tolerance is verified and validated by the generation and discharge of proof obligations with the help of eclipse-based Event-B platform which assists in the proof management of models based on distributed systems. The existing B tools generate proof obligations for consistency and refinement checking which ensure the safety property in distributed systems. In order to provide fairness for selection of overloaded node, the receiver will select that overloaded node which has minimum load among all available overloaded nodes. We add the following invariants for finding minimum overloaded node:

Invariant 22: $\forall nk, mk, ld1 \cdot (nk \in NODE \land node\_status(nk) = overloaded \land mk \in MESSAGE \land mk \in dom(message\_type) \land message\_type(mk) = reply \land ld1 \in load\_val \Rightarrow (nk \mapsto \{mk \mapsto ld1\}) \in reply\_node)$

The above specification ensures that the receiver must receive the load value from all overloaded nodes for selecting the overloaded node having minimum load value. To verify the fault tolerance property in our model, we add following invariant:

Invariant 23: $\forall nn,n \cdot (nn \in NODE \land nn \in dom(transfer\_load) \land node\_status(nn) = overloaded \land node\_status(n) = underloaded \land transfer\_load(nn) = failed \Rightarrow n\_status(n) = expired)$

It ensures that the load transfer from the overloaded node *nn* fails if the status of the underloaded node *n* is *"expired"*. After failure of load transfer, a new underloaded node *ud* will be selected which shares the extra load of the overloaded node *nn*. The following invariant ensures that while load transfer to node *ud* takes place it will remain in an active state.

Invariant 24: $\forall ud \cdot (ud \in NODE \land ud \in dom(transfer\_load) \land transfer\_load(ud) = completed \land node\_status(nn) = underloaded \Rightarrow n\_status(ud) = active)$

The *Invariant 25* verifies that if load is transferred from an overloaded node to the load balancer node (which is an underloaded node), the load value at the load balancer node will be less than threshold value. Therefore, it ensures balancing of the load value.

Invariant 25: $\forall n \cdot (\{n\} = load\_balancer \land n \in dom(transfer\_load) \land transfer\_load(n) = completed \land n \in dom(load\_value) \Rightarrow load\_value(n) < threshold)$

The invariants added to our model verify the correctness of load transfer and fault tolerance property in the model. Some of the issues and challenges[27] that occur while designing a load balancing algorithm which we have tried to address are as follows:

- The algorithm needs to be stable, scalable[28] and have a low overhead for the system[27] so that optimal processor utilization is achieved along with maximum throughput. The algorithm discussed in the paper is scalable as it does not depend on the number of nodes in the system. The algorithm does not cause instability in the system because there is a high probability that the underloaded node (receiver) will find an overloaded node (sender) quickly. This gives receiver-initiated load balancing algorithm an edge over the sender-initiated load balancing algorithm in which the responsibility of finding the underloaded node or receiver, lies with the overloaded node or sender. This sender or overloaded node is already burdened and at high system loads it becomes difficult to find a receiver or underloaded node.
- The algorithm must ensure that the load (task) is not transferred continuously from one node to another node without being executed.[29] Since the load balancing process in our algorithm is initiated by an underloaded node, the load transfer to this node stops as soon as the load value at this node reaches the threshold load value. Thus, the processes do not get transferred continuously from one node to another without any execution. This is also ensured by *Invariant 25*.

- The overhead of running a load distribution algorithm must not affect the overall efficiency or throughput of the system. Also, the algorithm must be general and transparent to the application.[30] Our algorithm does not affect the efficiency or throughput of the system because the load balancing activity is initiated and implemented by the underloaded node whose processing capability is still underutilized.

The novelty of the algorithm lies in the faction that load is first transferred from the least overloaded node to the underloaded node. This maximizes the number of overloaded nodes which can be addresses by an underloaded node. The algorithm is optimized to maximize the number of nodes with optimum load i.e., load value nearing the threshold value. Each step of the algorithm is formalized with the help of events comprising of guards and actions. Also, the feature of fault tolerance and recovery have been added to make the algorithm more robust.

Complexity Analysis – A formal model for receiver-initiated load balancing protocol for distributed systems or message passing systems has been proposed in this paper. In message passing system, the complexity of the model is measured in terms of communication cost i.e., the number of messages used by the algorithm. In this model, it is assumed that there are n nodes in the system.

The underloaded node (or receiver of load) broadcasts a request message to all the nodes except itself. Therefore, the number of messages required will be $n – 1$

After receiving the request message, only the overloaded nodes reply to the requesting node (underloaded node or load adjusting node). Assuming that there are k overloaded nodes, such that $k \leq n – 1$. Number of messages required for replying to the underloaded node = k.

After determination of optimal or minimum overloaded node, the underloaded node (receiver of load) informs the optimal overloaded node (sender) that it has been selected for load transfer. Number of messages required for this purpose = 1

Total number of messages = $(n – 1) + (k) + 1 = n+k$.

In the worst-case $k = n -1$

Therefore, the total number of messages required in the worst case = $n + (n-1) = 2n - 1$

Hence, the message complexity of the algorithm is $2n-1$ messages in order to find the optimal overloaded node.

Conclusions

The formal specification of the receiver-initiated load balancing protocol with fault tolerance in distributed systems using Event-B is discussed in detail in this paper. Receiver-initiated load balancing is a protocol, where the activity of load balancing is initiated by the underloaded node (receiver) which tries to obtain load from an overloaded node (sender). Event-B is a formal technique for mathematical specification of models of distributed systems step by step and then verifying the correctness of the system through discharge of proof obligations. Event-B supports a refinement-based approach for the development of models of distributed system algorithms and protocols. Eclipse based Rodin platform is used to carry out this work. The generation and discharge of proof obligations ensure consistency checking and refinement checking of the proposed model. The proof obligations are also helpful in creating new invariants which lead to better understanding of the problem and verify the correctness of its proposed solution. All the proofs generated by the model have been discharged. A total of 160 proof obligations are generated during verification of the model out of which 128 are discharged automatically while the remaining 32 are discharged interactively by provers of Event-B tools. It ensures that load balancing and fault tolerance properties are preserved in the model. This model gives a clear insight about the process of load transfer from one node to another for achieving uniform load distribution among nodes.

References

1 Begum S & Prashanth C, Review of load balancing in cloud computing, *Int J Comput Sci Issues* (IJCSI), **10(1)** (2013) 343.

2 Singhal M & Shivaratri N G, *Advanced Concepts in Operating Systems*, McGraw-Hill Science/Engineering/Math, 1994.

3 Eager D L, Lazowska E D & Zahorjan J, A comparison of receiver-initiated and sender-initiated adaptive load sharing, *Perform Eval*, **6(1)** (1986) 53–56.

4 Chou T C K & Abraham J A, Load balancing in distributed systems, *IEEE Trans Soft Eng*, **(4)** (1982) 401–412.

5 Elnozahy E N, Alvisi L, Wang Y-M, Johnson D B, A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys (CSUR)*, **34(3)** (2002) 375–408.

6 Suryavanshi R & Yadav D, Rigorous design of lazy replication system using Event-B, in *Int Conf Contempor Comput*, Springer, 2012, 407–418.

7 Chandra G, Suryavanshi R, Yadav D, Formal verification of distributed checkpointing using Event-B, *Int J Comput Sci Inf Technol*, **7(5)** (2015) 59–73.

8 Yadav D & Butler M, Rigorous design of fault-tolerant transactions for replicated database systems using Event B, in *Rigorous Development of Complex Fault-Tolerant Systems*, Springer, 2006, 343–363.

9 Chandra G & Yadav D, Verification of money atomicity in digital cashbased payment system, *Int Conf Inf Syst Secur*, Springer, 2012, 249–264.

10 Lahbib A, Wakrime A A, Laouiti A, Toumi K & Martin S, An Event-B based approach for formal modelling and verification of smart contractions, in *Int Conf Adv Inf Network Appl*, Springer, 2020, 1303–1318.

11 Lahouij A, Hamel L, Graiet M & el Ayeb B, An Event-B based approach for cloud composite services verification, *Form Asp Comput*, **32(4)** (2020) 361–393.

12 Karmakar R, Sarkar B B & Chaki N, Event-B based formal modeling of a controller: A case study, *Proc Int Conf Frontiers Comput System*, Springer, 2021, 649–658.

13 Ali, Alhaj A, Chramcov B, Jasek R, Katta R & Krayem S, Fault Tolerant Sensor Network Using Formal Method Event-B, in *Comput Sci On-line Conf*, Springer, Cham, 2021, 317–330.

14 Le A H, Van Khanh T & Thuan T N, Formal Analysis of Database Trigger Systems Using Event-B, *Int J Softw Innov (IJSI)*, **9(4)** (2021) 1–16.

15 Yadav P, Suryavanshi R, Singh A K & Yadav D, Formal verification of causal order-based load distribution mechanism using Event-B, in *Data Eng Appl*, Springer, 2019, 229–241.

16 Shukla S, Suryavanshi R S, Yadav D, Split point load balancing algorithm based on Event B, *Int J Innov Technol Explor Eng (IJITEE)* **8(9)** (2019) 2258–2265.

17 Suryavanshi R & Yadav D, Formal development of byzantine immune total order broadcast system using Event-B, in *Int Conf Data Eng Manag*, Springer, 2010, 317–324.

18 Hoang T S, Dghaym D, Snook C & Butler M, A composition mechanism for refinement-based methods, in *22nd Int Conf Eng Complex Comput Syst (ICECCS),* IEEE, 2017, 100–109.

19 Bodeveix J-P, Dieumegard A, Filali M, Event-B formalization of a variability-aware component model patterns framework, *Sci Comput Program*, **199** (2020) 102511

20 Singh A & Yadav D, Formal specification and verification of total order broadcast through destination agreement using Event-B, *Int J Comput Sci Inf Technol*, **7(5)** (2015) 85–95.

21 Metayer C, Abrial J & Voison L, *Event-B Language, RODIN Deliverables 3.2* (2005).

22 Abrial J-R, *The B-book: Assigning Programs to Meanings*, Cambridge university press, 2005.

23 B UK, Core (uk) limited, oxon, *B-toolkit*, *on-line manual*, 1999.

24 Abrial J-R, A system development process with Event-B and the rodin platform, in *Int Conf Formal Eng Methods*, Springer, 2007, 1–3.

25 Steria F. Aix-en-Provence, *Atelier B, User and Reference Manuals*, 2001.

26 Abrial J-R & Cansell D, Click'n' prove: Interactive proofs within set theory, in *Int Conf Theorem Prov Higher Order Logics*, Springer, 2003, 1–24.

27 Rajguru A A & Apte S, A comparative performance analysis of load balancing algorithms in distributed system using qualitative parameters, *Intl J Recent Technol Eng*, **1(3)** (2012) 175–179.

28 Ivanisenko I N & Radivilova T A, Survey of major load balancing algorithms in distributed system, in *Inform Technol Innov Bus Conf (ITIB)*, IEEE, 2015, 89–92.

29 Sharma S, Singh S & Sharma M, Performance analysis of load balancing algorithms, *World Acad Sci Eng Technol*, **38(3)** (2008) 269–272.

30 Salehi M A, Deldari H & Dorri B M, Balancing load in a computational grid applying adaptive, intelligent colonies of ants, *Informatica*, **33(2)** (2009).