Journal of Scientific & Industrial Research
Vol. 83, July 2024, pp. 721-729
DOI: 10.56042/jsir.v83i7.5812

Rigorous Specification of Vector Timestamp Based Load Sharing Mechanism for Distributed Systems

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

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

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

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

Received 25 September 2023; revised 24 May 2024; accepted 11 June 2024

Distributed systems are autonomous computer nodes or sites that are interconnected by a communication network. These sites function independently without a shared memory or global clock, communicating through message exchanges. Any incoming job can be allocated to any of these independent sites within the system. This leads to a non-uniform distribution of load where some sites become highly loaded while others remain lightly loaded or idle. This scenario leads to poor utilization of resources, heightened response time and impaired system performance. This paper presents the formal specification of a load balancing protocol for distributed systems which is based on vector time-stamping of messages using the vector clock at a site. In order to add fairness to the load sharing mechanism, the request for unburdening the heavily laden site is catered to first whose request message has the least vector timestamp from among the request messages present in the request acquisition queue of the lightly loaded site. Formal methods utilise mathematical techniques to methodically detect and correct errors in the development of software in its initial phases. Event-B is a formal method used to create models of proposed algorithms. The approach used is event-driven, employing set-theoretic principles to construct models of distributed system protocols. The procedure involves creating and satisfying proof obligations to validate the correctness of a model. The article presents a rigorous analysis of the vector clock mechanism in distributed systems as well as a load distribution system which uses vector timestamped messages as the underlying message exchange protocol using Event-B.

Keywords: Distributed systems, Event-B, Formal methods, Load balancing, Vector clock

Introduction

As distributed system consists of several autonomous computer nodes or sites which may be located at different geographical locations are interconnected by a communication network, any incoming job can be allocated to any of these independent sites within the system. This leads to the problem of overloading at some sites while the resources at other sites remain underutilized causing degraded system performance and increased response time.[1] Several load balancing protocols have been proposed for distributed systems to overcome this problem of non- uniform load distribution.[2–5] The Event-B specifications a load sharing algorithm based on vector timestamping of messages are presented in this paper. Formal methods use rigorous mathematical reasoning for the construction and verification of models based on software systems.[6] The behavioural aspects of the system are described through its functional specification. Event-B is a formal methodology that offers a systematic mathematical approach for formulation of accurate system specifications. Discharge of proof obligations in Event-B models offers a mechanism for verification of system properties which ensures that the system is working according to its specifications. The vector clock at a site presents the most up to date knowledge which is available at the site about the local clock of all other sites in the system.[7–9] In comparison to vector clock, the scalar clock represents the knowledge of the local clock of the site only. The vector clock is used for vector timestamping of messages.[10,11] In this protocol, an overloaded site broadcasts a request message for unburdening its excess load in which it assigns its own vector clock value as the timestamp of the message. On receiving the request message, the receiving site updates its own vector clock value using the vector timestamp of the received message and puts the received request message in its request acquisition queue. A lightly loaded site may receive several such messages

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

requesting for removal of load from various highly loaded sites. It compares the vector timestamp of all the received request messages present in its request acquisition queue and sends a reply to only that heavily loaded site whose request message has the minimum timestamp value. This adds fairness to the load sharing protocol. After getting a response from a site that is minimally loaded, the requesting site updates its vector clock using the timestamp from the reply message and alleviates the excess load from its queue. This excess load is shifted to the queue of the lightly burdened site which responded with a reply message. Once a request is processed it is removed from the request acquisition queue of the site.

Literature Survey

Event-B has been widely employed in the development of formal models for algorithms based on distributed system. Some of the papers dealing with formal analysis of load distribution protocols for distributed systems are discussed in this section. The Event-B specifications of causal order based load sharing mechanism are given by Yadav *et al*.[12] In this protocol causal order based message delivery is used for exchange of messages. The load sharing systems is triggered by the heavily burdened site, which serves as the load sender. The lightly burdened site responds to the request of that highly loaded site only whose request message causally precedes the request messages of all the other highly loaded sites. A protocol was modelled using the Rodin platform, employing the Event-B paradigm. The model generated 42 proof obligations, out of which 30 were discharged automatically using tool support provided by Rodin platform while 12 required manual interaction. Yadav *et al.*[13] present the formal verification of a receiver initiated load sharing mechanism. In this protocol, the load sharing mechanism is triggered by the lighter burdened node, which acts as the recipient of the load. This protocol has an edge over the sender initiated protocol at high system load as the lightly loaded node finds a sender or heavily loaded node quickly and shares its load. Furthermore, the responsibility for initiating the load balancing process is transferred from the heavily burdened node to the lightly burdened node. Site recovery and fault tolerance during load balancing are also demonstrated in the paper. The Event-B model created for this algorithm generated 160 proof obligations out of which 128 were discharged automatically while 32 required manual intervention. Shukla *et al.*[14] have presented the formal specifications of a load distribution system in which the network is partitioned into several clusters. A site is chosen as the coordinator within each cluster. The excess load at an over burdened site is first adjusted within the cluster. If there is no lightly loaded site present within the cluster for adjustment of load, the coordinator of the cluster finds another cluster for shifting of load. The Event-B model of this protocol generated 93 proof obligations, out of which 24 required manual interventions while 69 were discharged automatically. Yadav *et al.* [15] have introduced an enhanced leader election process that adjusts the priority of a site to become the leader based on the site's load value. A site's priority for becoming the leader decreases as its load value increases, as sites with higher load values are more likely to fail compared to those with lower load values. The formal analysis of split point based load sharing protocol is given by Shukla *et al.*[16] In this algorithm, the excess burden at a heavily burdened site is shared among several minimally burdened sites. This paper throws light on the situation in which a single lightly loaded site may not have the capacity to share the excess load of a heavily loaded site completely, so the load is split among several lightly loaded sites. The Event-B model created for this protocol generated 80 proof obligations, all of which were discharged automatically.

Research Gaps

Vector clocks are one of the major means of coordination of activities among various sites in a distributed system as they lack global clocks. Event-B offers a methodical technique to formally create distributed system protocols using set theoretic constructs and verify them by discharging proof obligations. Some of the mechanisms used for balancing of load in distributed systems and their formal analysis are discussed above however, the formal verification of a protocol which uses vector timestamping of messages as the mechanism for load sharing in distributed systems is scarcely discussed in the available literature. This paper presents a load sharing protocol which uses the vector timestamp of messages for selecting the heavily loaded site whose request for load removal is processed first when there are request messages from several heavily loaded sites. A minimally burdened site compares the vector timestamp of all the received request messages present in its request acquisition queue and sends a

reply to only that heavily loaded site whose request message has the minimum timestamp value. This adds fairness to the protocol which provides an improvement over other load sharing mechanisms. The primary contributions of this article encompass:

- Demonstration of the procedure for updating vector clocks at a site in a distributed system on sending and receiving of messages.
- The use of vector clock for assigning timestamp to messages instead of the Lamport's scalar clock.
- Development of a load sharing protocol which uses vector clock mechanism at a site in a distributed system for vector timestamping of messages.
- The development of Event-B specifications for the proposed vector timestamp based load sharing protocol and verification of its correctness through discharge of proof obligations.

Methodology

System Model

In the proposed model, messages with vector timestamp are used for communication among the sites. Each message is timestamped using the vector clock value of the site which sends the message. A request acquisition queue is maintained at each site containing request messages which are timestamped. A vector clock contains the local clock of the site as well as the latest knowledge a site has about the local clock at all other sites.[17] A vector clock consists of an array of size *n*, because the system consists of *n* sites. The vector clock at a site *si* is represented by *vt_site(si)*. The local time at site *si* is given by *vt_site(si)(si)*. The latest knowledge that site *si* has about the local time at site *sj* is given by *vt_site(si)(sj)*. *Vt_site(si)(sj)* is the j^{th} entry in the vector *vt_site*. When site *si* sends a message, it increases its local time by one unit *vt_site(si)(si)= vt_site(si)(si) + 1*. *Vt_site(si)(si)* is the i^{th} entry in the vector *vt_site*. Site *si* assigns this updated vector clock value with the vector timestamp value attached with message *vtmsg*. On receiving a message, a site amends its own vector clock according to the vector timestamp value attached with the received message. In the proposed load sharing mechanism, the total load at a site is represented by its queue length. If the queue length of the site is greater than the optimal load value then it is considered to be excessively loaded and if the queue length is lesser than the optimal load value then it is lightly loaded. The optimal load value is the ideal load value at which the resources at a site are properly utilized and it is not overloaded or underloaded. An outline of the proposed load sharing protocol which uses vector timestamped messages is demonstrated below in Algorithm 1:

Algorithm 1 — Algorithm for Vector Timestamp based Load Sharing Mechanism

When the arrival of a new job occurs at site *si*, its queue length is incremented by one
  queuelength(si) = queuelength(si) + 1
If *queuelength(si) > optimal load value* then
  vt_site(si)(si) := vt_site(si)(si) + 1
  Broadcast timestamped request message mreq for removal of excess load with timestamp *vtmsg(mreq)*.
  vtmsg(mreq)(sk) := vt_site(si)(sk), ∀k(k ∈ 1, 2, 3.....n), where *n*represents the overall number of sites present in the system.
  Put *mreq* in request acquisition queue of *si*.
On receiving timestamped request message *mreq*
  Site *sj* updates its own vector clock
  vt_site(sj)(sk) := Max(vt_site(sj)(sk), vtmsg(mreq)(sk)), ∀k(k ∈ 1, 2, 3.....n), where *n* represents the overall number of sites present in the system.
  Put *mreq* in request acquisition queue of *sj*.
Site *sj* sends timestamped reply message *mrep* if
  queuelength(sj) < optimal load value
  the vector timestamp of *mreq* is less than the vector timestamps all other request messages present in the request acquisition queue of *sj*.
  vtmsg(mreq)(sk) <vtmsg(mi)(sk), ∀k(k ∈ 1, 2, 3.....n),∀i(i∈ 1, 2, 3.....m), where *n* represents the overall number of sites present in the system and *m* is the total number of request messages present in the request acquisition queue of site *sj*.
On receiving timestamped reply message *mrep* at site *si*
  Site *si* updates its own vector clock *vt_site(si).*
  vt_site(si)(sk) := Max(vt_site(si)(sk), vtmsg(mrep)(sk)), ∀k(k ∈ 1, 2, 3.....n), where *n* represents the overall number of sites present in the system.
  Remove load*queuelength(si) := queuelength(si) − 1*
  Remove *mreq* from request acquisition queue of site *si*.
Add load at site *sj*
  queuelength(sj) := queuelength(sj) + 1
  Remove *mreq* from request acquisition queue of site *sj*.
END

The system model for the vector timestamp based load sharing mechanism is outlined in Fig. 1.

Formal Modelling Approach: Event-B

Formal methods utilise mathematical techniques to precisely establish software specifications, facilitating the identification of errors during the early phases of software development. Event-B is a formal method used to specify the load-sharing protocol. It relies on set theory to construct a software model.[18] Event-B utilises an event-driven methodology for specifying systems, where the model consists of static components known as contexts and dynamic

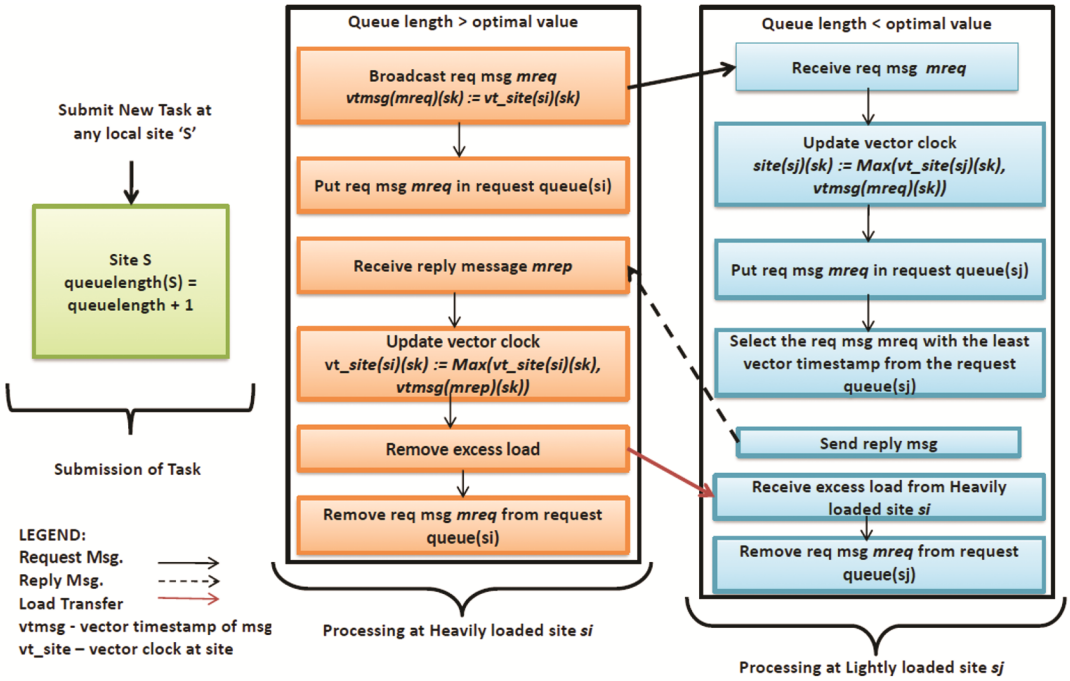

Fig. 1 — Framework of vector timestamp based load sharing algorithm

components known as machines.[19] Contexts are composed of sets, axioms, and theorems, whereas machines include events, variables, and invariants to describe behavioural aspects. The system's state evolution is determined by events, which are executed when all event guards are activated.[20] Event-B facilitates the iterative improvement of software models, beginning with abstract machines and advancing towards more explicit system specifications.[21] Abrial[22] offers a thorough explanation of the notations used in Event-B. Proof obligations established in Event-B must be fulfilled in order to test the accuracy of the model. This can be done using Rodin,[23] an Eclipse-based tool that helps in formulating Event-B specifications and discharging proof obligations according to the proposed protocol.

Event-B Model for Vector Timestamp based Load Sharing Protocol

This section provides a description of the Event-B model for the load sharing protocol based on vector timestamps. The context section of the proposed protocol includes SETOFSITES and SETOFMESSAGES as carrier sets. They denote the set of sites and the set of messages respectively. *Category* is an enumerated set containing the type of messages which are *load_bal_req* and *load_balance_reply*. Invariants and events together make up the machine segment of the model. The machine's invariants are described below:

inv1 : queuelength∈ SETOFSITES →NATURAL
inv2 : optimal_value∈ NATURAL
inv3 : msg_sender∈ SETOFMESSAGES ⇸ SETOFSITES
inv4 : sent_messages⊆ SETOFMESSAGES
inv5 : category_of_msg∈sent_messages→category
inv6 : vt_site∈ SETOFSITES →(SETOFSITES →NATURAL)
inv7 : vtmsg∈ SETOFMESSAGES →(SETOFSITES →NATURAL)
inv8 : requesting_sites⊆ SETOFSITES
inv9 : req_queue∈ SETOFSITES ↔(SETOFMESSAGES⇸SETOFSITES)
inv10 : sent_reply_msg∈ (SETOFMESSAGES ↔SETOFMESSAGES)⇸SETOFSITES
inv11 : rec_reply_msg∈ SETOFSITES ↔(SETOFMESSAGES↔SETOFMESSAGES)
inv12 : msg_deliver∈ SETOFSITES ↔SETOFMESSAGES
inv13 : msg_del_order∈ SETOFSITES ⇸ (SETOFMESSAGES ↔ SETOFMESSAGES)

Invariants

The variable *queuelength* denotes the total number of tasks at a site or the load value of a site. Invariant 1 states that the variable *queuelength* must have a value that is a natural number. The variable *optimal_value* represents the threshold value of load at a site. Invariant 3 describes the variable *msg_sender* as a partial function from *SETOFMESSAGES* to *SETOFSITES* which implies that a mapping *(m1 ↦*

s1) ∈msg_sender means message *m1* is sent by site *s1*. In invariant4, the set *sent_messages*is a set of messages which have been sent by any site and it is a subset of the set *SETOFMESSAGES*. The variable *category_of_msg*, according to invariant 5 is a total function for mapping each message sent to its category which may be *load_bal_req*or *load_balance_reply*. The variable *vt_site* represents the value of vector clock at a site as invariant 6. A total function represents the mapping of each site to a vector. The vector is a total function from the *SETOFSITES* to a natural number. The length of the vector is equivalent to the number of sites in the system. Assuming that the system comprises of *n* sites, the *vt_site(site i)* for *sitei* will be *((Site1 ↦ N1), (Site2 ↦N2),(Site3 ↦ N3).....(Siten↦Nn))*. Each time *site i* sends a message, it increases its own vector clock value *vt_site(site i)(site i)* by one. The variable *vtmsg* represents the vector timestamp of a message. Invariant 7 describes the variable *vtmsg* as the total function mapping each message to a vector. When a *site i* transmits a message *msg*, it increases its own vector clock value by one and assigns this revised vector clock value as the vector timestamp of the message *vtmsg(msg)*. The vector clock value *vt_site(site i)(site i)*denotes the total number of messages transmitted by *site i*. Invariant 8 defines *requesting_sites* as the set of sites which have sent a *load_bal_req* message. It is a subset of the *SETOFSITES*. The variable *req_queue*, as per invariant 9 is a set of relations between *SETOFSITES* and *load_bal_req*messages transmitted by the corresponding requesting sites. The mapping *(sj↦(mreq↦ si)) ∈req_queue* denotes that the request message *mreq* transmitted by site *si* is in the request acquisition queue of site *sj*. *Req_queue(sj)* is the relational image of site *sj* under the relation *req_queue*. For example, if site *sj* receives three *load_bal_req* messages *ma*, *mb*and *mc* from sites *sp*, *sq* and *sr* respectively, then *req_queue(sj)* will be *((ma ↦sp), (mb ↦sc), (mc ↦sr))*. In invariant 10, the variable *sent_reply_msg* denotes the reply sent by a site in correspondence to the request message. The mapping *((mreq↦mrep) ↦sj)* represents that *load_balance_reply* message *mrep* in correspondence to *load_bal_req* message *mreq* is sent by site *sj*. In invariant 11, the variable *rec_reply_msg* indicates the receiving of a *load_balance_reply* message in compliance to a *load_bal_req* message at a site. The delivery of a message at a site is indicated by the variable *msg_deliver* as per invariant 12. The mapping *(si↦ m1) ∈msg_deliver* indicates that message *m1* is delivered at site *si*. According to invariant 13, the variable *msg_del_order* specifies the order of delivery of messages at a site. The mapping *(ma ↦ mb ∈msg_del_order(si))* represents that message *ma* is delivered before message *mb* at site *si*. In the initialization event, the vector clock of all the sites and the vector timestamp of all the messages are set to zero. All the other variables are given null values. The detailed description of the events of the machine is given as follows:

Event 1: Arrival of New Task

The acceptance of a new job at a site is demonstrated by this event. When a new task is submitted at any site *si* belonging to *SETOFSITES*(grd1), the modification in its queue length is represented as an increase by one (act1).

EVENT *New job submission*
ANY*si*
WHERE
grd1 *:si∈SETOFSITES*
THEN
act1 *:queuelength(si) := queuelength(si) + 1*
END

Event 2: Broadcast of Timestamped Request Message

During this event, a heavily burdened site broadcasts a request message to transfer its load to a less burdened website. Site *si* is excessively loaded because its queue length is greater than the optimal load value (grds 1 & 2). Site *si* is not in the set of *requesting_sites* and message *mreq* is a new message i.e., it was never sent previously (grds 4,5 & 6). Site *si* increases its own vector clock value by one unit before broadcasting message *mreq* (grds 7 & 8) and also ensures that message *mreq* is not in its own request acquisition queue (grd9). Action 1 assigns the incremented vector clock value of site *si* as the vector timestamp of message *mreq*. Action 2 ensures that message *mreq* is broadcast by site *si*. Action 3 adds site *si* to the set of *requesting_sites* and action4 adds the message *mreq* to the set of *sent_messages*. The category of message *mreq* is set as *load_bal_req*(act5) and it is added to the request acquisition queue of site *si* (act 6).

EVENT *Broadcast Timestamped Request*
ANY*mreq, si, new_vtsite*
WHERE
grd1 *:si∈SETOFSITES*
grd2 *:queuelength(si) >optimal_value*

grd3 *:si∉requesting_sites*
grd4 *:mreq∈SETOFMESSAGES*
grd5 *:mreq∉sent_messages*
grd6 *:mreq∉dom(msg_sender)*
grd7 *:new_vtsite∈(SETOFSITES →NATURAL)*
grd8 *:new_vtsite∈vt_site(si) ⩤ {si↦vt_site(si)(si) + 1}*
grd9 *: {mreq↦si} ∉req_queue[{si}]*
THEN
act1 *:vtmsg(mreq) := new_vtsite*
act2 *:msg_sender := msg_sender∪{mreq↦si}*
act3 *:requesting_sites := requesting_sites∪{si}*
act4 *:sent_messages := sent_messages∪{mreq}*
act5 *:category_of_msg(mreq) := load_bal_req*
act6 *:req_queue := req_queue∪{si↦{mreq↦si}}*
END

Event 3: Receiving of Timestamped Request Message

This event showcases the reception of a request message with a timestamp at the receiving site *sj*. The *load_bal_req*message *mreq* is transmitted by the requesting site *si* and has not yet been received by site *sj* and is not present in the request acquisition queue of site *sj* (grds 1 to 10). Guard 11 ensures that messages are delivered at site *sj* in FIFO order. All the messages transmitted by site *si* prior to message *mreq* are previously received by site *sj*. Action 1 delivers message *mreq* at site *sj*. The order of message delivery at site *sj* is updated in such a manner that every message received at site*s j* prior to message *mreq* precedes message *mreq* (act2). Message *mreq* is placed in the request acquisition queue of site *sj* (act3). Site *sj* maintains the current knowledge about the system by upgrading its own vector clock value *vt_site(sj)* with the corresponding vector timestamp value *vtmsg(mreq)* of message *mreq* wherever the vector clock value *vt_site(sj)(sk)* is less than the vector timestamp value *vtmsg(mreq)(sk)*. The overload operator ⩤ is used for this purpose.

EVENT *Receive Timestamped Request*
ANY *mreq, si, sj*
WHERE
grd1 *:si∈SETOFSITES*
grd2 *:si∈requesting_sites*
grd3 *:sj∈SETOFSITES*
grd4 *:sj∈dom(msg_del_order)*
grd5 *:mreq∈SETOFMESSAGES*
grd6 *:mreq∈sent_messages*
grd7 *:category_of_msg(mreq) = load_bal_req*
grd8 *: {mreq↦si} ∈msg_sender*
grd9 *: {mreq↦si} ∉req_queue[{sj}]*
grd10 *: (sj↦mreq) ∉msg_deliver*
grd11 *: ∀mi, sk ⋅ (mi ∈SETOFMESSAGES ∧sk∈SETOFSITES ∧(mi ↦si) ∈msg_sender∧vtmsg(mi)(sk) <vtmsg(mreq)(sk) ⇒(sj↦mi) ∈msg_deliver)*
THEN
act1 *:msg_deliver := msg_deliver∪{sj↦mreq}*
act2 *:msg_del_order(sj) := msg_del_order(sj) ∪ (msg_deliver[{sj}] × {mreq})*
act3 *:req_queue := req_queue∪{sj↦{mreq↦si}}*
act4 *:vt_site(sj) := vt_site(sj) ⩤ ({sk|sk∈SETOFSITES∧ vt_site(sj)(sk) <vtmsg(mreq)(sk)} ⩤vtmsg(mreq))*
END

Event 4: Transmitting of Timestamped Reply Message

The dispatch of reply message in answer to the timestamped request message is demonstrated by this event. Site *sj* is a lightly loaded site (grds 1 & 6). The *load_bal_req*message *mreq* has been sent and received at site *sj* (grds 2 to 5). The reply message answering the *load_bal_req* message *mreq* is not yet sent (grds 8 and 9). Site *sj* is not the sender of message *mreq*. Guard 12 is specified as:

∀sk, mi ⋅ (sk∈ SETOFSITES ∧ mi ∈ SETOFMESSAGES ∧ {mi ↦sk} ∈req_queue[{sj}] ⇒ (∀sa ⋅ sa∈ SETOFSITES ∧vtmsg(mreq)(sa) <vtmsg(mi)(sa)))

It ensures that the vector timestamp of the *load_bal_req* message *mreq* is least of all the messages in the request acquisition queue of site *sj*. The lightly loaded site *sj* responds to only that heavily loaded site whose request message has the least timestamp among all the request messages present in the request acquisition queue of site *sj*. This adds fairness to the algorithm as the site which raised the request first is catered to before all other requesting sites. Site *sj* increases the value of its vector clock value by one unit before sending the reply message (grds 13 & 14). The updated vector clock value of site *sj* is assigned as the vector timestamp of message *mrep* (act1). Message *mrep* is inserted in the set of *sent_messages* (act2) and its category is set as *load_balance_reply* (act3). Site *sj* is the sender ofmessage *mrep* (act4). Message *mrep* is sent as a reply to message *mreq* by site *sj* as ensured by action 5.

EVENT *Timestamped Reply*
ANY*mreq,mrep, sj, new_vtsite*
WHERE
grd1 *:sj∈SETOFSITES*
grd2 *:mreq∈SETOFMESSAGES*
grd3 *:mreq∈sent_messages*
grd4 *: (sj↦mreq) ∈msg_deliver*
grd5 *:category_of_msg(mreq) = load_bal_req*
grd6 *:queuelength(sj) <optimal_value*
grd7 *:mrep∈SETOFMESSAGES*
grd8 *:mrep∉sent_messages*
grd9 *:mrep∉dom(msg_sender)*
grd10 *: {mreq↦mrep} ∉dom(sent_reply_msg)*
grd11 *:sj∉{msg_sender(mreq)}*
grd12 *: ∀sk, mi ⋅ (sk∈SETOFSITES ∧mi ∈SETOFMESSAGES∧*

{mi ↦sk} ∈req_queue[{sj}] ⇒ (∀sa · sa∈SETOFSITES∧ vtmsg(mreq)(sa) <vtmsg(mi)(sa)))
grd13 *:new_vtsite∈ (SETOFSITES →NATURAL)*
grd14 *:new_vtsite = vt_site(sj) <⊢ {sj↦vt_site(sj)(sj) + 1}*
THEN
act1 *:vtmsg(mrep) := new_vtsite*
act2 *:sent_messages := sent_messages∪ {mrep}*
act3 *:category_of_msg(mrep) := load_balance_reply*
act4 *:msg_sender := msg_sender∪ {mrep↦sj}*
act5 *:sent_reply_msg := sent_reply_msg∪ {({mreq↦mrep}) ↦sj}*
END

Event 5: Receiving of Timestamped Reply Message

The delivery of the timestamped reply message at the highly loaded site is demonstrated in this event. Site *si* is the requesting site and its queue length is greater than the optimal load value (grds 1, 2 & 3). Site *sj* is the lightly loaded site as its queue length is lesser than the optimal value (grds 4 & 5). The *load_bal_req* message *mreq* has been sent by site *si* and the *load_balance_reply* message *mrep* has been sent by site *sj* as a reply to *mreq* (grds 6 to 12). Message *mreq* is in the request acquisition queue of site *si*. Message *mrep* is not yet received by site *si* (grd 14). Guard 15 ensures that all the messages are received in FIFO order at site *si* i.e., all the messages sent before message *mrep* are delivered before message *mrep* at site *si*. The vector timestamp value of all the messages sent before message *mrep* is less than the vector timestamp value of message *mrep*. Action 1 ensures the delivery of message *mrep* at site *si*. All the messages received before message *mrep* precede message *mrep* in the delivery order at site *si* (act2). Message *mrep* is received as a reply to message *mreq* at site *si*. Site *si* updates its vector clock according to the vector timestamp of the received message *mrep* (act4).

EVENT *Receive Timestamped Reply*
ANY *mreq, mrep, si, sj*
WHERE
grd1 *:si∈SETOFSITES*
grd2 *:si∈requesting_sites*
grd3 *:queuelength(si) >optimal_value*
grd4 *:sj∈SETOFSITES*
grd5 *:queuelength(sj) <optimal_value*
grd6 *:mreq∈sent_messages*
grd7 *:category_of_msg(mreq) = load_bal_req*
grd8 *: (mreq↦si) ∈msg_sender*
grd9 *:mrep∈sent_messages*
grd10 *:category_of_msg(mrep) = load_balance_reply*
grd11 *:si∈dom(msg_del_order)*
grd12 *: {mreq↦mrep} ∈dom(sent_reply_msg)*
grd13 *: {mreq↦si} ∈req_queue[{si}]*
grd14 *: (si↦ ({mreq↦mrep})) ∉rec_reply_msg*
grd15 *: ∀sk,mg · (sk∈ SETOFSITES ∧ mg ∈ SETOFMESSAGES∧ (mg ↦sk) ∈msg_sender∧vtmsg(mg)(sk) <vtmsg(mrep)(sk) ⇒ (si↦ mg) ∈msg_deliver)*
THEN
act1 *:msg_deliver := msg_deliver∪ {si↦mrep}*
act2 *:msg_del_order(si) := msg_del_order(si) ∪ (msg_deliver[{si}]× {mrep})*
act3 *:rec_reply_msg := rec_reply_msg∪ {si↦ ({mreq↦mrep})}*
act4 *:vt_site(si) := vt_site(si) <⊢ ({sk|sk∈ SETOFSITES∧ vt site(si)(sk) <vtmsg(mrep)(sk)} <⊲vtmsg(mrep))*
END

Event 6: Load Removal from Heavily Loaded Site

This event illustrates the process of removing the load from a site that is experiencing a high level of demand. The requesting site *si* is heavily burdened while site *sj* is lightly loaded (grds 1 to 5). The *load_bal_req* message *mreq* has been sent by site *si* (grds 6,7 & 8). The *load_balance_reply* message *mrep* is sent by site *sj* as a reply to message *mreq* (grds 9 to 12). The request message *mreq* is in the request acquisition queue of site *si*(grd 13). Message *mrep* has been received at site *si* as a reply to message *mreq*(grd14). The alleviation of load from site *si* is illustrated as a unit decrease in its queue length (act1). The load balance request message *mreq* is removed from the request acquisition queue of site *si* (act2).

EVENT *Remove Load*
ANY*mreq,mrep, si, sj*
WHERE
grd1 *:si∈ SETOFSITES*
grd2 *:si∈requesting_sites*
grd3 *:queuelength(si) >optimal_value*
grd4 *:sj∈ SETOFSITES*
grd5 *:queuelength(sj) <optimal_value*
grd6 *:mreq∈sent_messages*
grd7 *:category_of_msg(mreq) = load_bal_req*
grd8 *: (mreq↦si) ∈msg_sender*
grd9 *:mrep∈sent_messages*
grd10 *:category_of_msg(mrep) = load_balance_reply*
grd11 *: (mrep↦sj) ∈msg_sender*
grd12 *: {mreq↦mrep} ∈dom(sent_reply_msg)*
grd13 *: {mreq↦si} ∈req_queue[{si}]*
grd14 *: (si↦ ({mreq↦mrep})) ∈rec_reply_msg*
THEN
act1 *:queuelength(si) := queuelength(si) − 1*
act2 *:req_queue := req_queue\{si↦ {mreq↦si}}*
END

Event 7: Addition of Load at the Lightly Loaded Site

The addition of load at the lightly loaded site is shown in this event. Site *si* is the requesting site and site *sj* is the lightly loaded site (grds 1 to 4). The *load_bal_req* message *mreq* has been sent by site *si* (grds 5,6 & 7). The *load_balance_reply* message *mrep* is sent by site *sj* as a reply to message *mreq*

Table 1 — Statistics of Proofs obtained for the Event-B specifications for vector timestamp based load sharing protocol

Element name	Total POs	POs Discharged Automatically	POs Discharged Manually	Undischarged POs
VECTORLB context	0	0	0	0
VECTORLB Machine	76	69	07	0

(grds 8 to 11). The request message *mreq* is not present in the request acquisition queue of *si* (grd 12). Message *mrep* has been received at site *si* as a reply to message *mreq* (grd13). The request message *mreq* dispatched by site *si* is in the request acquisition queue of site *sj* (grd 14). The addition of load at site *sj* is modelled as an increment in its queue length by one (act1). The *load_bal_req* message *mreq* dispatched by site *si* is removed from the request acquisition queue of site *sj* (act2). Site *si* is removed from the set of *requesting_sites* (act3).

EVENT *Addition of Load*
ANY*mreq,mrep, si, sj*
WHERE
grd1 *:si∈SETOFSITES*
grd2 *:si∈requesting_sites*
grd3 *:sj∈SETOFSITES*
grd4 *:queuelength(sj) <optimal_value*
grd5 *:mreq∈sent_messages*
grd6 *:category_of_msg(mreq) = load_bal_req*
grd7 *:(mreq↦si) ∈msg_sender*
grd8 *:mrep∈sent_messages*
grd9 *:category_of_msg(mrep) = load_balance_reply*
grd10 *: (mrep↦sj) ∈msg_sender*
grd11 *: {mreq↦mrep} ∈dom(sent_reply_msg)*
grd12 *: {mreq↦si} ∉req_queue[{si}]*
grd13 *: (si↦ ({mreq↦mrep})) ∈rec_reply_msg*
grd14 *: {mreq↦si} ∈req_queue[{sj}]*
THEN
act1 *:queuelength(sj) := queuelength(sj) + 1*
act2 *:req_queue := req_queue\{sj↦ {mreq↦si}}*
act3 *:requesting_sites := requesting_sites\{si}*
END

Results and Discussion

An Event-B model for a load sharing algorithm which uses vector timestamping of messages for communication among the sites is presented in this paper. In this protocol, the vector clock at a site is used for vector timestamping of messages. A highly burdened site broadcasts a vector timestamped request message for removal of excess load. A lightly loaded site may receive several such request messages and put them in its request acquisition queue. It responds to only that request message whose vector timestamp is lesser than the vector timestamp of all the messages present in its request acquisition queue. This is ensured by the invariant given below:

∀sk, mi · (sk∈ SETOFSITES ∧ mi ∈ SETOFMESSAGES ∧ {mi ↦sk} ∈req_queue[{sj}] ⇒ (∀sa · sa∈ SETOFSITES∧vtmsg(mreq)(sa) <vtmsg(mi)(sa)))

The following invariant ensures that a *load_bal_req*which is processed once is not processed again by ensuring that it is removed from the request acquisition queue of all the sites.

∀si,sj, mi · (si∈ SETOFSITES ∧sj∈ SETOFSITES∧ mi ∈ SETOFMESSAGES∧category_of_msg(mi) = load_bal_req∧{mi ↦si}∈msg_sender∧si∉requesting_ sites⇒{mi ↦si} ∉req_queue[{sj}])

The invariant given below ensures that the messages received at a site are a subset of messages that have already been sent.

ran(msg_deliver) ⊆ dom(msg_sender)

The above invariants are used to strengthen the model. The Rodin platform is utilized for writing these Event-B specifications as it provides an exhaustive package for generation and discharge of proof obligations. The discharge of proof obligations ensures the correctness of the model. All the proof obligations generated by the Event-B model of this load sharing algorithm were discharged through provers embedded in Rodin tool without any anomaly. The statistics of proofs of the Event-B model of the proposed protocol are given in Table 1.

Analysis of Complexity

The message complexity of an algorithm is determined by its communication cost, which refers to the number of messages needed during the it's execution. It is computed as follows:
The system comprises of *n* sites.

Messages required by the highly loaded site for broadcast of timestamped request message = *n – 1*
Messages required for sending of timestamped reply message by the lightly loaded site = *1*

Therefore, the proposed load sharing mechanism utilizes a total of *(n – 1) + 1 = n* messages for alleviation of load from an over burdened site which is the message complexity of the protocol.

Conclusions

The formal analysis of the vector timestamp based load sharing mechanism for distributed systems is presented in this paper. This protocol uses the vector clock at a site for timestamping of messages instead

of the Lamport's scalar clock. The vector clock at a site represents its own local time as well as the latest knowledge it has about the local time of all other sites. The value of this vector clock is assigned as the vector timestamp of a message which is being sent. The formal specifications of this protocol are written using Event-B. Event-B offers a comprehensive framework for developing mathematical models of algorithms for distributed systems. In an Event-B model, invariants are developed according to the specifications and proof obligations are discharged to ensure that these invariants are satisfied. The problem of non uniform load distribution and subsequent site failure due to high load is a prevalent issue in most large distributed computing applications which can be mitigated by the proposed protocol for load distribution. It can be used in systems such as e-banking, railway reservation systems, government portals with heavy loads, digital payment systems etc.

References

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

2 Talaat F M, Ali H A, Saraya M S & Saleh A I, Effective scheduling algorithm for load balancing in fog environment using CNN and MPSO, *Know Info Sys*, **64(3)** (2022) 773–797.

3 Kashani M H & Mahdipour E, Load balancing algorithms in fog computing: A systematic review, *IEEE Trans on Services Computing*, **16(2)** (2023) 1505–1521.

4 Singh P, Kaur R, Rashid J, Juneja S, Dhiman G, Kim J & Ouaissa M, A fog-cluster based load-balancing technique, *Sustainability*, **14(13)** (2022) 7961.

5 Li C, Cai Q & Lou Y, Optimal data placement strategy considering capacity limitation and load balancing in geographically distributed cloud, *Future Gen Comp Sys*, **127** (2022) 142–159.

6 Beek M H, Larsen K G, Ničković D & Willemse T A, Formal methods and tools for industrial critical systems, *Int J Soft Tools Tech Trans*, **24(3)** (2022) 325–330.

7 Beschastnikh I, Liu P, Xing A, Wang P, Brun Y & Ernst M D, Visualizing distributed system executions, *ACM Trans Soft Eng Methodology*, **29(2)** (2020) 1–38.

8 Salem I & Schiller E M, Practically-Self-stabilizing Vector Clocks in the Absence of Execution Fairness, in *Networked Sys NETYS 2018*, Lecture Notes in Computer Science, edited by A Podelski and F Taïani (Springer, Cham) 2019, 11028.

9 Lindsay D, Gill S S, Smirnova D & Garraghan P, The evolution of distributed computing systems: From fundamental to new frontiers, *Computing*, **103(8)** (2021) 1859–1878.

10 Yadav P, Suryavanshi R & Yadav D, Formal verification of liveness properties in causal order broadcast systems using Event-B, *Proc Sec Doc Symp Comp Intel* (Springer Singapore) 2022, 199–210.

11 Schwitanski S, Tomski F, Protze J, Terboven C & Müller M S, An On-the-Fly method to exchange vector clocks in distributed-memory programs, in *IEEE Int Parallel Distrib Proc Symp Workshops* (IEEE) 2022, 530–540.

12 Yadav P, Suryavanshi R, Singh A K & Yadav D, Formal Verification of Causal Order-Based Load Distribution Mechanism Using Event-B, *Data Eng Appl*, **2** (2019) 229–241.

13 Yadav P, Suryavanshi R & Yadav D, Formal Verification of receiver initiated load distribution protocol with fault tolerance and recovery using Event-B, *J Sci Ind Res*, **80** (2021) 1078–1090.

14 Shukla S, Suryavanshi R & Yadav D, Formal modelling of cluster-coordinator-based load balancing protocol using Event-B, *Proc Sec Doc Symp Comp Intel* (Springer Singapore) 2022, 593–603.

15 Yadav P, Suryavanshi R & Yadav D, Formal Specification of dynamic load-based coordinator selection algorithm with recovery in distributed systems, *Proc Third Doc Symp Comp Intel* (Springer Nature Singapore) 2023, 153–163.

16 Shukla S, Suryavanshi R S & Yadav D, Split point load balancing algorithm based on Event B, *Int J Innovative Tech Exploring Eng*, **8(9)** (2019) 2258–2265.

17 Suryavanshi R & Yadav D, Modeling of Distributed Mutual ExclusionSystem Using Event-B, in *Int Conf Comp Sci and info* edited by J Zizka (SIPP, AISC, PDCTA) 2013, 477–491.

18 Abrial J R, *Modeling in Event-B: system and softeng* (Cambridge University Press) 2010.

19 Riviere P, Singh N K & Aït-Ameur Y, EB4EB: A framework for reflexive Event-B, in *Int Conf on Eng of Complex Comp Sys* (IEEE) 2022, 71–80.

20 Aït-Ameur Y, Bogomolov S, Dupont G, Singh N K & Stankaitis P, Reachability analysis and simulation for hybridised Event-B Models, in *Int Conf on Integrated Formal Methods* (Cham: Springer International Publishing) 2022, 109–128.

21 Zhu C, Butler M, Cirstea C & Hoang T S, A fairness-based refinement strategy to transform liveness properties in Event-B models, *Sci Comp Prog*, **225** (2023) 102907.

22 Abrial J R, A system development process with Event-B and the Rodin platform, *Proc Int Conf Formal Eng Methods* (Springer, Berlin, Heidelberg) 14-15 November, 2007.

23 Metayer C, Abrial J R & Voison L, Event-B language, *Deliverables*, **3** (2005).