J Supercomput (2013) 63:612–635
DOI 10.1007/s11227-012-0834-2

Specification and verification of reliability in dispatching multicast messages

Seyed Morteza Babamir

Published online: 1 November 2012

Abstract Multicasting some pieces of information such as messages or packets (called dispatches) from source node(s) to a group of target nodes are governed by a specific sequence in the networked systems. The sequence is called a consensus that indicates an ordering on dispatches to be viewed by the target nodes. Achievement of consensus is a concern in some networked based systems such as distributed ones because the lack of consensus leads to conflict among target nodes reaction. A consensus protocol has some properties to be checked when a source node multicasts a sequence of dispatches to target nodes. The CBCAST protocol is a consensus protocol having properties for ordering and synchronization of dispatches in network communications. This paper thinks of the properties and formulates axioms to check them. The axioms can be practiced for network applications such as group communication and web services. Our approach has two phases consisting of modeling and formulation. The first phase addresses specification of sender and recipient processes by tabular automata. The second phase addresses formulation of axioms using the automaton.

Keywords Asynchronous communication · Client-server · Network reliability · Distributed system · Multicast protocol

1 Introduction

Networked systems as distributed systems use dispatches (a piece of information) for communication between nodes. In such systems, the dispatch sequence is one of the significant concerns. A dispatch sequence indicates ordering of viewing dispatches in target nodes. FIFO (First in First Out), for instance, is a sequence indicating dispatches whose view ordering by target nodes adhere to their transmission

S.M. Babamir (✉)
School of Computer and Electrical Engineering, University of Kashan, Kashan, Iran
e-mail: babamir@kashanu.ac.ir

ordering. If a viewed dispatch sequence by a target node does not adhere to an expected dispatch sequence, the target node is very likely open to failure in achieving its tasks.

The dispatch sequence, on the other hand, is a concern in group communication [5]. It is possible to create a special network address to which multiple machines, called a group, can listen. When a dispatch is sent to one of these addresses, it is automatically delivered to all machines listening to the address. This technique is called multicasting [14]. In a networked system, in fact, a group is collection of nodes that act together when some dispatch is multicasted to them. Thus, on a network, each group has a different multicast address.

The key property is that when a dispatch is transmitted to the group, all members of the group view it. It indicates that members of a group should view same dispatch sequence. Contrary viewed dispatch sequences of the members lead to incompatibility of members decisions. Dissatisfaction of an expected dispatch sequence is due to the presence of faults in the sender or recipient programs, or network failure or delay. It has been shown how these faults may lead to malfunctions such as false deadlock in resources allocation in a distributed system [20]. Accordingly, verification of the interprocess communication in networked systems has appeared as a significant matter.

To make group communication reliable, the expected dispatch sequence should be considered. A networked system with multicasting group communication, in fact, has to include well-defined semantics with respect to the sequence in which dispatches are delivered. Absolute time sequence is not always easy to implement, so a consistent time sequence can be considered. In this sequence, if two dispatches are sent close together in time, the system picks one of them as "first" and delivers it to all group members; however, the chosen sequence may really not be the first dispatch. In effect, dispatches are guaranteed to be viewed by all group members in the same sequence, but that sequence may not be the same as ordering of dispatches transmission.

For some networked systems, weaker semantics called virtual synchrony can be acceptable because of better performance. The semantics is one in which the sequence constraint has been more relaxed than the consistent sequence [5]. The CBCAST protocol is a communication protocol providing virtually synchronous semantics [4].

This study aims to present a method to formulate axioms that can be used for verifying behavior of processes in networked systems in according to the virtually synchronous semantics. In order to describe sequence constraints in the CBCAST protocol, we use event-based logic, a formal method for specifying events. These constraints are our requirements.

When a dispatch is multicasted by a source node, the runtime support of the multicast mechanism is responsible for delivering the dispatch to each target node of the multicast group. To show dispatch sequence and synchronization between dispatches, an event-based specification is used. As each target node may reside on a separate machine, the delivery of dispatches requires the cooperation of the machines. Problems such as failure of network links or network hosts, routing delays, and the time duration between sending and viewing dispatch are causes of dissatisfaction of the synchronization semantics. The differences in dispatch delivery to members of a group are a significant matter.

The remainder of paper is organized as follows. Section 2 deals with related work. The CBCAST protocol is explained in Sect. 3. Section 4 explains properties of the

CBCAT protocol. Section 5 deals with specifying behavior of sender and recipient processes by tabular automata and shows that how axioms are formulated using the automata. The axioms can be used to verify some actual dispatch sequence against an expected dispatch sequence having a specific semantics (Sect. 6). In Sect. 7, we state considerations applied in our implementation. Finally, in Sect. 8, the future work and main conclusions of this study are presented.

2 Related work

Specification of dispatch sequence in networked systems and rule construction for verification of the sequence received considerations. We specified dispatch sequence using Petri-nets and then extracted constrains should govern the sequence [1]. The approach applied to the distributed computation of Fibonacci numbers. Contrasts between our previous and the present work are in: (1) specification of communicating processes behavior and constraints and (2) consideration of implementation. While the former used Petri-nets for specification of the processes behavior and Petri-nets invariants for constraints, the later uses tabular automata and event-based invariants, respectively. Moreover, the later proposes using vector clocks for abstract and concrete implementation satisfaction of dispatch orderings. Another one of our previous work [2] specified properties of the CBCAST multicast protocol using textual and graphical temporal logic, and then mapped the specification to UML Statecharts as state machines. The present work differs from the previous one in two cases: (1) the specification of processes (sender and receiver) behavior and constraints and (2) the inclusion of abstract and the concrete implementation model of causal and FIFO properties in the present work.

Lomazova [15] and Kostin and Ilushechkina [10] suggested Petri-nets and their extension, respectively, to model and verify dispatch sequences in distributed systems. Temporal logics were used by [11, 18, 25, 26]. Sen et al. [18] specified verification rules in PT-DTL, a variant of past time linear temporal logic. Visual models were used by [7, 12, 13] for specification of dispatch sequence. Drusinsky and Shing [7], Kruger et al. [13] and Kruger et al. [12] used Message Sequence Charts (MSCs) in their specification. Drusinsky and Shing [7] extracted assertions from the Message Sequence Chart (MSC) and employed to formalize the actual dispatch sequence showing the system behavior. MSCs are visual models to show the desired dispatch sequence; however, they cannot be employed to show process states in interprocess communication. Accordingly, they used UML Statecharts to resolve this obstacle [8]. Then they extracted verification assertions from the Statecharts. Although using MSC and UML visual models to specify the processes, specification has the advantage of getting a better understanding of the processes behavior; they lack a rigorous formal specification. This is the contrast between such methods and ours and others that used formal methods.

The FiLM tool was introduced by [25] to specify the dispatch sequence obtained by finite traces of execution runs against linear time temporal logic specifications. Then the finite automaton was constructed from LTL specifications to evaluate the trace. This method used linear temporal logic formula to specify the verification rules, similar to the method presented by [18]. The runtime dispatches were observed by [11] for deducing a runtime state transition diagram (STD). Then temporal rules resided in a rule-base were used to verify the correctness of STD.

Event-B is another formal method using a framework for developing formal models of distributed systems. Yadav and Butler [24] and Fulmare and Yadav [9] used the Event-B method for representing formal development of broadcast communication and for causal and total ordering of message delivery. This work separately proposed models for causal and total orderings and dealt with verifying the models. They showed an abstract of implementation of the ordering properties using the vector clocks and described invariants stating the relationship of causal and total orderings using the vector clocks and the sequence numbers. There are similarities and contrasts between the method presented by Yadav and Butler and our proposed method. They considered three properties, called *validity*, *agreement*, and *integrity*, which are close to properties 1, 3, and 4 that we formulate and verify. While they dealt with total and causal orderings of message, we consider FIFO (property 2) and causal orderings. Although both methods propose vector clocks for abstract implementation, Yadav and Butler did not consider a concrete implementation. However, our method presents a model for concrete implementation of the causal ordering property.

Four types of reliable multicast protocols were implemented over IP multicast using the standard UDP interface [14]: the ACK-based, the NAK-based with polling, the ring-based, and the tree-based protocols. The performance of these protocols was evaluated over Ethernet-connected networks. Then the impact of the Ethernet features (such as flow control, buffer management, and error control mechanisms) on the performance of the protocols was studied.

Tsaur and Horng [22] proposed an approach for *auditing* causal relationships of multicast communications in group-oriented distributed systems where the auditing task is to prevent processes from denying or forging causal relationships between events. The auditing was used to identify attacks including the denial of existing causal relationships and the forgery of nonexistent causal relationships. Tsaur and Horng aimed to just deal with causal ordering property and they were unconcerned with other properties we addressed.

Tsuchiya and Schiper [23] used a model checking method as a semi-automatic verification approach for asynchronous *consensus* algorithms. Since the state space of such algorithms is huge, the proposed approach dealt with this difficulty by reducing the verification problem to small model checking problems; the small model involves one phase of the algorithm execution. According to [23], the proposed approach model can check consensus algorithms up to around 10 processes. While Tsuchiya and Schiper used the model checking method, our method considers checking at runtime. Similar to our method, they used invariants to state properties; however, properties that they addressed were different from those we consider.

3 Consensus protocol

A consensus protocol with multicast group communication consists of semantics with respect to the sequence by which dispatches are delivered. The protocol environment is a distributed system where nodes are communicating processes. The working assumptions are constraints relating to the environment, i.e., the distributed system: (1) processes are run on nodes and have no shared memory; so, the interaction among the processes is carried out just via dispatch passing and no process is aware of other processes behavior (2) processes may crash before or after broadcasting or receiving a message, but they will recover from the crash after a while; moreover, the environment, i.e., the network on which dispatches are transmitted may be disconnected, (3) a dispatch may depend on another one, i.e., it will be broadcasted when some other dispatch is viewed, (4) an acknowledgment is sent back to the sender process by a receiver when the receiver views a dispatch and (5) a sender process imposes a deadline for an acknowledgment and waits no longer if the deadline expires.

A synchronous semantics [4, 5] is one in which group members strictly view dispatches sequentially, i.e., broadcasting dispatches takes zero time to complete. Figure 1(a) shows a true synchrony semantics where: (1) p, q, r, s, t, and u are communicating processes and processes p, q, r, and s are organized into one group and (2) arrows indicate dispatch broadcasts. An arrow from process p to process q, for instance, denotes sending a message from p to q. The right arrows mean that the broadcasts are truly carried out in lockstep and there is no delay in delivery of dispatches. The ovals with arrows denote that processes q, r, and s obtain the acceptance of membership from process p. The oval with an explosion mark denotes crashing process q, and accordingly there is no longer interaction between q and other processes. Processes t and u are the nonmembers that can interact with the group members of p. Similar to broadcasting dispatches among the group members, the right arrows mean that the broadcasting dispatches between members and nonmembers are truly carried out in lockstep and there is no delay between sending and delivery of dispatches. Adherence to true synchrony is difficult, because absolute time sequence is not easy to implement.

A loosely synchronous [4, 5] semantics is one in which transmission of dispatches takes a finite amount of time, but all members view dispatches in the same sequence (the slanted allows in Fig. 1(b)). Such semantics is easy to implement. The ABCAST [4] communication protocol provides loosely synchronous communication and is used for transmitting data to the members of a group. However, this protocol is complex and expensive.

Each arrow denotes sending a dispatch from a process to another one. However, while the right arrows in Fig. 1(a) denote no delay in delivery of dispatches to the target process, the diagonal ones mean that there is a delay in delivery of dispatches. The ovals have the same meaning as Fig. 1(a).

For some networked systems, even weaker semantics is acceptable to obtain better performance. Virtual synchrony [4, 5] is the one in which the sequence constraint is more relaxed than a loosely synchrony. In this semantics, two dispatches are causally related if one of them is influenced by another. Figure 1(c) shows dispatches m_1 and m_2 where m_1 is broadcasted by process P_1 to all processes. Having viewed dispatch m_1, process P_2 broadcasts dispatch m_2 to all processes. Because process P_2 broadcasts dispatch m_2 when it viewed dispatch m_1, m_1 is the cause of m_2. In fact, if a process sends a dispatch to another process and the second process views the dispatch and then sends a new dispatch to a third process, the second dispatch is

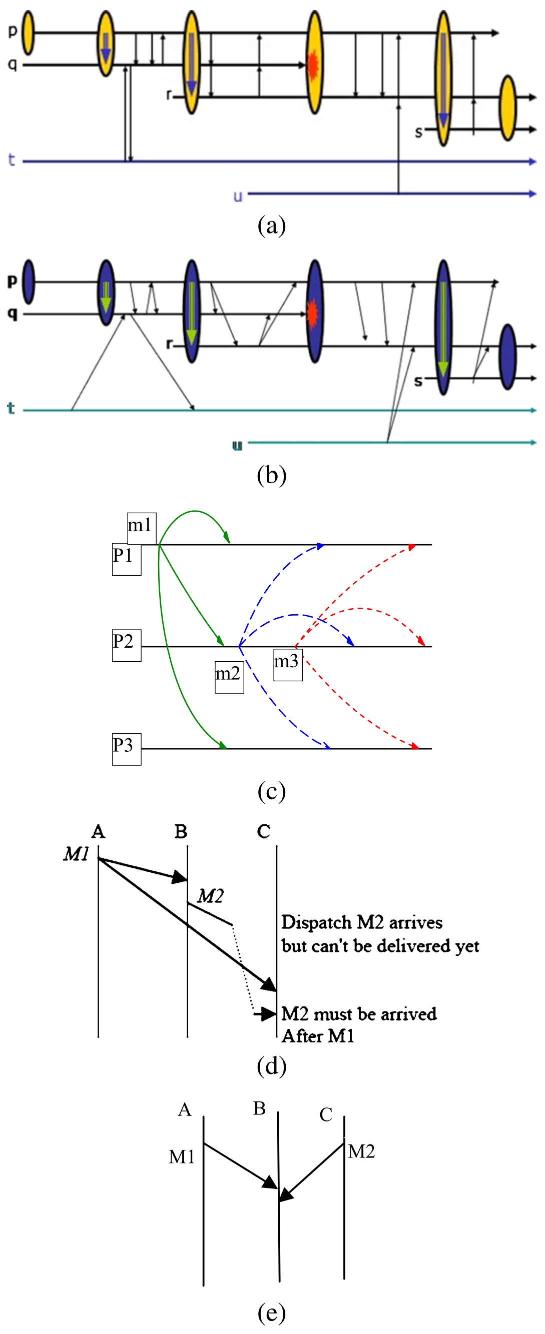


Fig. 1 (**a**) True synchrony. (**b**) Loosely synchrony. (**c**) Virtual synchrony (causal dispatches). (**d**) Dispatch deferment in causality. (**e**) Concurrent dispatches

causally related to the first one because its contents might have been derived from the first one. In Fig. 1(c), since dispatch m_1 is sent out before sending out dispatch m_2 and m_1 is the cause of m_2, each recipient process should deliver m_1 before m_2.

Similarly, since m_2 is causally sent out before sending out m_3, each recipient process should deliver m_2 before m_3.

Figure 1(d) shows the transmission of dispatches in the CBCAST protocol. When process "B" transmits a dispatch of its own, i.e., M_2 to process "C," and it arrives before M_1, process "C" views that process "B" had already received one dispatch from process "A" before M_2 was sent, and since it has not yet received anything from process "A," M_2 is buffered until a dispatch from process "A" arrives.

If neither dispatch influences each other, they are unrelated and called concurrent. For instance, if node "A" sends a dispatch, say M_1 to node "B," and at about the same time, node "C" sends a dispatch, say M_2 to node "B." The dispatches are concurrent because neither can influence the other (Fig. 1(e)). So, the view ordering of M_1 and M_2 is not a concern.

Semantics of virtual synchrony says that if two dispatches are causally related, all processes must receive them in the same sequence. If, however, they are concurrent, the system is free to deliver them in a different sequence to different nodes. CBCAST protocol is a communication protocols providing virtually synchronous communication. To implement this protocol, each process maintains a vector with n components, one per group member. The ith component of this vector is the number of the last dispatch viewed in sequence from process "i."

4 Protocol analysis

This section aims to deal with presenting formal and event-based specification of constraints and should be adhered to the reliable dispatch transmission based on virtual synchrony in networked systems. Virtual synchrony is a model for reliable multicasting in group communication. The reliable multicast system is one which guarantees that each dispatch is eventually delivered correctly to each process in the group. The definition of reliable multicast requires that each participating process views exactly one copy of each dispatch sent but the dispatches are causally related, and must be delivered in order.

Property 1 Each sent dispatch is eventually viewed in a particular interval of time by each group member.

Property 2 FIFO Sequence: if dispatch m_1 is sent before dispatch m_2, then m_1 should be viewed before m_2.

Property 3 Dispatch entirety. It indicates that each dispatch belongs to some sender; so, no dispatch should be produced spontaneously; noises and unwanted messages among others are spontaneous ones.

Property 4 Dispatch distinctiveness. It indicates that no dispatch should be repeated. Such dispatches may be nonidempotent. An idempotent dispatch such as "reading a specific record of a file" changes nothing when it unaskedly is rehashed but a non-idempotent dispatch such as "updating a specific record of a file" leads to incorrect results when it is unwantedly rehashed.

If sender or network fails in transmitting some dispatch to a group member or if the member or network fails in giving out acknowledgment to sender, the dispatch should be warmed-over. To adhere to the reliable communication, property "exactly once semantics" should be considered. This property ensures that a dispatch is viewed by a group member just one time. It features properties "at least once semantics" and "at most once semantics" and has no their problems. They ensure that a transmitted dispatch is viewed at least/at most once by the dispatch recipient. While protocol "at least" is used for idempotent dispatches, protocol "at most" is used for nonidempotenti ones.

5 Formulating axioms

This section aims to derive axioms from the consensus protocol whose properties explained in Sect. 4. The axioms can be used to verify adherence of dispatches in a distributed system to properties of the consensus protocol. A distributed system includes some sequential processes having no shared memory and interprocess communication is generally carried out through dispatches. Every process behavior includes changing local states in transmitting dispatches to other processes. A process action is specified by a reaction to view of a dispatch. Since process behavior is sequential, happening events in a process have a general sequence.

Events are divided into two types: (1) local or internal such as process crash that may happen independently in every process and (2) concurrent or external such as multicasting dispatches. An event causes a state transition in a process. Local events cause just state transition in one process and concurrent events cause state transition of source and target ones. Since communication media may not send dispatches to target in the sequence sent by source, it is necessary to establish a consistent sequence between dispatches. To this end, the virtual synchrony semantics is used. As stated in Sect. 3, virtual synchrony focuses on causally related events. Two events are causally related, when one of them causes the other. This means that the second dispatch is transmitted by some process when the process viewed the first one. This characteristic, which is significant to ensure reliability in the sequence of viewing dispatches by group members, indicates that two dispatches can be viewed in any order if they are not related. Such dispatches are called concurrent. Establishing causal sequence and releasing concurrent ordering are constraints are considered by the virtual synchrony. Virtual synchrony, used in the consensus protocol CBCAST, improves the performance of reliable dispatch communication.

The first step of axiom formulation We specify sender/recipient behavior by tabular automata. The automaton for dispatch sender is shown by two tables; mode (state) transition and event (Tables 1 and 2). The *mode* indicates the state where the process is doing some action. The event table shows the actions should be carried out by the dispatch sender.

Modes (States) of dispatch sender are as follows: Ready, Sending, Awaiting, Success, Failed, and Lifeless, of which the Ready mode is the primary one. The second column in Tables 1 and 3 denotes events and conditions. "Send" (sending a dispatch),

Table 1 Sender mode (state) transition

Current mode	Send	R_{ACK}	TOut	Crash	New mode
Ready	@T	–	–	–	Sending
Sending	@F	–	–	–	Awaiting
Awaiting	f	@T	f	–	Success
Awaiting	f	f	@T	–	Failed
Ready/sending/ awaiting/success/ failed	–	–	–	@T	Lifeless
Lifeless	–	–	–	@F	Ready

Table 2 Sender events

Mode	Event	
Sending	@F(Send)	*false*
Awaiting	False	$@T(R_{ACK})$ *when* $TOut_{ACK} = false \vee @T(TOut_{ACK})$ *when* $R_{ACK} = false$
SetTimer$_{ACK}$	*True*	*false*

Table 3 Recipient mode (state) transition

Current mode	Recv	Crash	New mode
Ready	@T	–	Receiving
Receiving	@F	–	Successful
Ready/receiving/success	–	@T	Lifeless
Lifeless	–	@F	Ready

"R_{ACK}" (receiver acknowledgment), "Tout" (expiring time of acknowledgment view) and "Crash" (process fail) denote an event when they are stated by "@T" or "@F" and play the role of temporal predicate or condition when they are stated by "t" or "f." The third row of Table 1, for instance, shows that if the sender is awaiting acknowledgment of the receiver (Awaiting mode) and Send = f (sending dispatch has finished) and Tout = f (the acknowledgment deadline has not expired), then sender makes a transition to the Success state.

If sender is in mode Sending and the dispatch transmission finished (shown by event @F(SEND)), then the sender shall turn the acknowledgment timer on. However, if the sender is in mode Sending, it has no role in setting the timer to zero (false). If the sender is in mode Awaiting and receives the acknowledgment from receiver (shown by event @T(R_{ACK})), the sender will reset the timer to zero for the next cycle provided that the acknowledgment deadline has not expired. Similarly, if the acknowledgment deadline expires, the timer is reset. Similar to Tables 1 and 2, Tables 3 and 4 show state transition and event tables for the receiver.

The receiver is always in one of four modes: Ready, Receiving, Success, and Lifeless, which the primary mode is Ready. If the receiver is in mode Ready and a dis-

Table 4 Recipient event

Mode	Event
Receiving	@F(Recv)
Send_{ACK}	true

patch arrives, indicated by event @T(Recv), the recipient goes to mode Receiving. But, if the recipient is in mode Receiving and receiving the dispatch comes to an end (shown by @F(Recv)), then the recipient shall start sending an acknowledgment to the sender.

Now, we deal with formulating axioms by considering the Tables 1 to 4. First, we specify temporal predicates (the second row of Tables 1 and 3) as fluents of the sender.

$$\text{Fluents} = \{\text{Send}, \text{R}_{\text{ACK}}, \text{TOut}, \text{Crash}\}$$

The value of a fluent is specified by "t," "f," "@T," and "@F" indicating conditions "true" and "false" and change events "false→true" and "true→false." The predicate "Send = t," for instance, says: "fluent Send is true" and the predicate "Send = @T," for instance, says: "the truth value of fluent Send has changed from false (in current mode) to true (in new mode)" (see the explanation of Table 1). This means that fluent "Send" was false in the current mode of the sender and it becomes true in the new mode.

We categorize axioms as (1) State Invariant (MI indicating Mode Invariant) and Transition Invariant (TI). We use predicate HoldsAt(S, t) for representing the process state at time "t." Since, at a time instant, the sender can only be in one state, the first MI represents this situation. Formula (1) shows Axiom "exclusive state" for sender where "⊕" indicates the exclusivity of modes.

$$\begin{aligned}\text{MI}_0(t) \overset{\text{def}}{\equiv} \big[&\text{HoldsAt}(\text{Ready}, t) \oplus \text{HoldsAt}(\text{Sending}, t)\\ &\oplus \text{HoldsAt}(\text{Awaiting}, t) \oplus \text{HoldsAt}(\text{Success})\\ &\oplus \text{HoldsAt}(\text{Failed}, t) \oplus \text{HoldsAt}(\text{Lifeless}, t)\big] \end{aligned} \tag{1}$$

Other MIs for the sender are specified as Formulas (2)–(5) indicating various states of the sender. Each formula shows connection between a sender state (the left-hand side of the formula) and its corresponding fluent (the right-hand side of the formula). Consider event @T below "Send" in the first row of Table 1. It means that "Send" is false in the current mode (Ready) and will change to true when the sender makes a transition to the next state (Sending). This relation is stated as Formulae (2) and (3). In other words, "If the sender is in mode Ready at time t, the fluent Send is false at this time."

$$\text{MI}_1(t) \overset{\text{def}}{\equiv} \text{HoldsAt}(\text{Ready}, t) \rightarrow {\sim}\text{HoldsAt}(\text{Send}, t) \tag{2}$$

$$\text{MI}_2(t) \overset{\text{def}}{\equiv} \text{HoldsAT}(\text{Sending}, t) \rightarrow \text{HoldsAt}(\text{Send}, t) \tag{3}$$

Consider rows 2 to 4 of Table 1. Notation @F (true→false) under "Send" in row 2 denotes "Send" becomes false when the sender goes to mode Awaiting and notation "f" under "Send" in rows 3 and 4 states that "Send" stays false both in mode Awaiting and in its next modes. This fact is stated by the first predicate of the right-hand side of Formula (4). Moreover, rows 3 and 4 of Table 1 include three values for fluents Send, $\mathrm{R_{ACK}}$ and TOut. Similar to what we stated for fluent "Send" above, two other predicates $\sim\mathrm{HoldsAt}(\mathrm{R_{ACK}}, t)$ and $\sim$HoldsAt(TOut) at the right-hand side of Formula (4) appear for fluents $\mathrm{R_{ACK}}$ and TOut. Similarly, Formula (5) is stated for fluent "Crash."

$$\begin{aligned}\mathrm{MI}_3(t) &\overset{\mathrm{def}}{\equiv} \mathrm{HoldsAT}(\mathrm{Awaiting}, t) \rightarrow \\ &\quad \left[\sim\mathrm{HoldsAt}(\mathrm{Send}, t) \wedge \sim\mathrm{HoldsAt}(\mathrm{R_{ACK}}, t) \wedge \sim\mathrm{HoldsAt}(\mathrm{TOut})\right] \quad (4)\\ \mathrm{MI}_4(t) &\overset{\mathrm{def}}{\equiv} \mathrm{HoldsAT}(\mathrm{Lifeless}, t) \rightarrow \mathrm{HoldsAt}(\mathrm{Crash}, t) \quad (5)\end{aligned}$$

5.1 Formulas (2) to (5): Axioms "state specification" for sender

Formulas (2)–(5) are axioms in form of temporal predicate used to verify the sender behavior when some event happens. When a condition (temporal predicate) is false (indicated by "f") in the state transition table, it means that the predicate is false in both current and new states, and when an event happens, it means that the predicate is: (1) false in the current mode and (2) true in the new mode or vice versa.

Now, we deal with axioms TI. For being in a process state, we use predicate HoldsAt(β, τ), which β indicates the state and to represent an event, we use predicate Happens(α, τ)), which α indicates the event. To enter a state, we use the predicate Initiates(α, β, τ) and to exit from a state, we use predicate Terminates(α, β, τ). So, we use the predicates Initiates(α, β, τ) and Terminates(α, β, τ) for every row of the state transition table. Each pair of predicates results from predicates HoldsAt(β, τ) and Happens(α, τ).

Owing to similarity, we present axioms TI only for rows 3 and 4 of Table 1. Transition from the mode Awaiting is represented as axiom (6) and entering modes Success and Failed are represented as axioms (7) and (8), respectively.

$$\begin{aligned}\mathrm{TI}_1(t) \overset{\mathrm{def}}{\equiv}\quad & \mathrm{Terminates}\big(@\mathrm{T}(\mathrm{RACK}), \mathrm{Awaiting}, t\big) \leftarrow \\ & \text{if } \mathrm{HoldsAt}(\mathrm{Awaiting}, t) \wedge \\ & \big[\sim\mathrm{HoldsAt}(\mathrm{Send}, t) \wedge \sim\mathrm{Happens}\big(@\mathrm{T}(\mathrm{Send}), t\big) \wedge \\ & \sim\mathrm{HoldsAt}(\mathrm{Tout}, t) \wedge \sim\mathrm{Happens}\big(@\mathrm{T}(\mathrm{Tout}), t\big) \wedge \\ & \sim\mathrm{HoldsAt}(\mathrm{R_{ACK}}, t) \wedge \mathrm{Happens}\big(@\mathrm{T}(\mathrm{R_{ACK}}), t\big)\big] \\ \vee\quad & \\ & \big[\sim\mathrm{HoldsAt}(\mathrm{Send}, t) \wedge \sim\mathrm{Happens}\big(@\mathrm{T}(\mathrm{Send}), t\big) \wedge \\ & \sim\mathrm{HoldsAt}(\mathrm{R_{ACK}}, t) \wedge \sim\mathrm{Happens}\big(@\mathrm{T}(\mathrm{R_{ACK}}), t\big) \wedge \\ & \sim\mathrm{HoldsAt}(\mathrm{TOut}, t) \wedge \mathrm{Happens}\big(@\mathrm{T}(\mathrm{TOut}), t\big)\big] \quad (6)\end{aligned}$$

5.2 Formula (6): Axiom "exit of the Waiting state" for sender

$$\begin{aligned}\mathrm{TI}_2(t) \overset{\mathrm{def}}{\equiv} & \mathrm{Initiates}\big(@\mathrm{T}(\mathrm{RACK}), \mathrm{Success}, t\big) \leftarrow \\ & \text{if } \mathrm{HoldsAt}(\mathrm{Awaiting}, t) \wedge\end{aligned}$$

$$\begin{aligned}
&\big[\sim\text{HoldsAt}(\text{Send}, t) \wedge \sim\text{Happens}\big(@\text{T}(\text{Send}), t\big) \wedge \\
&\sim\text{HoldsAt}(\text{Tout}, t) \wedge \sim\text{Happens}\big(@\text{T}(\text{Tout}), t\big) \wedge \\
&\sim\text{HoldsAt}(\text{RACK}, t) \wedge \text{Happens}\big(@\text{T}(\text{RACK}), t\big)\big] \qquad (7)
\end{aligned}$$

$$\begin{aligned}
\text{TI}_3(t)^{\text{def}} \equiv\, &\text{Initiates}\big(@\text{T}(\text{TOut}), \text{Failed}, t\big) \leftarrow \\
&\text{if HoldsAt}(\text{Awaiting}, t) \wedge \\
&\big[\sim\text{HoldsAt}(\text{Send}, t) \wedge \sim\text{Happens}\big(@\text{T}(\text{Send}), t\big) \wedge \\
&\sim\text{HoldsAt}(\text{RACK}, t) \wedge \sim\text{Happens}\big(@\text{T}(\text{RACK}), t\big) \wedge \\
&\sim\text{HoldsAt}(\text{TOut}, t) \wedge \text{Happens}\big(@\text{T}(\text{TOut}), t\big)\big] \qquad (8)
\end{aligned}$$

5.3 Formulae (7) and (8): Axioms "entering modes Success and Failed"

In every axiom, the happening of an event is specified as a pair of predicates, Happens and ∼HoldsAt. This means that when an event happens, the fluent in predicate ∼HoldsAt has not already held, but it holds now (indicated by predicate Happens).

Altogether, 23 axioms are acquired for specifying the rightful sender behavior from Tables 1 and 2 where 5 and 18 axioms are respectively considered as MIs and Tis. From 18 TIs, 15 axioms are obtained from the transition table (Table 1) and 3 axioms are obtained from the event table (Table 2). The disjunction of these axioms totally constructs specification of the rightful sender behavior represented as the general axiom (9). Also, we acquired 15 axioms to specify the rightful receiver behavior where 4 and 11 axioms are respectively considered as MIs and TIs (10 axioms are obtained from the transition table (Table 3) and one axiom is obtained from the event table (Table 4).

$$\begin{aligned}
I_{\text{total}}^{\text{def}} \equiv\, &\text{MI}_0(t) \wedge \text{MI}_1(t) \wedge \cdots \wedge \text{MI}_5(t) \wedge \\
&\text{TI}_1(t) \wedge \text{TI}_2(t) \wedge \cdots \wedge \text{TI}_{18}(t) \quad \text{where } t \in \text{Time} \qquad (9)
\end{aligned}$$

The second step of axiom formulation In this step, we formulate axioms for the constraints are considered in reliability of dispatch communication in the consensus protocol. They are called safety axioms including 4 properties.

Formulating axiom for Property 1 As stated in Sect. 4, it should be guaranteed that each sent dispatch eventually is viewed in a particular interval of time by the dispatch recipient. To formulate the axiom for Property 1, we consider Tables 1 and 3. The axiom, in fact, constructs connection between modes Sending in Table 1 and Receiving in Table 3.

$$\begin{aligned}
P_1^{\text{def}} \equiv\, &\text{Happens}\big(@\text{T}\big(\text{Send}(p, m)\big), t_i\big) \rightarrow \\
&\exists q \exists t_{j \in \text{Time}}\colon \text{Happens}\big(@\text{T}\big(\text{Receive}(q, m)\big), t_j\big) \wedge t_j < t_i \\
&\text{where } t_i \text{ and } t_j \in \text{Time}
\end{aligned}$$

Formulating axiom for Property 2 This property stated as FIFO Sequence in Sect. 4 indicating if dispatch m_1 is sent before dispatch m_2, then m_1 should be viewed before m_2. To formulate the axiom for Property 2, we consider Tables 1 and 3. The axiom, in fact, connects ordering of two Sending modes of Table 1 with ordering of two Receiving modes of Table 3.

$$\begin{aligned} P_2 \overset{\text{def}}{\equiv} \text{if } &\text{Happens}\big(@\text{T}\big(\text{Send}(p, m_1)\big), t_1\big) \wedge \\ &\quad \text{Happens}\big(@\text{T}\big(\text{Send}(p, m_2)\big), t_2\big) \wedge t_1 < t_2 \rightarrow \\ &\exists q \text{Happens}\big(@\text{T}\big(\text{Receive}(q, m_1)\big), t_3\big) \wedge \\ &\quad \text{Happens}\big(@\text{T}\big(\text{Receive}(q, m_2)\big), t_4\big) \wedge t_3 < t_4 \\ &\text{where } t_1, t_2, t_3 \text{ and } t_4 \in \text{Time} \end{aligned}$$

Formulating axiom for Property 3 (dispatch entirely) As stated in Sect. 4, this property denotes that no dispatch shall be produced spontaneously. This property shows that for every dispatch view, a dispatch sending has been already occurred. This axiom is used to recognize any unwished dispatch or noise (P_3). Similar to axiom P_1, P_3 constructs the connection between the two modes in Tables 1 and 3; however, this connection is in reverse order of P_1.

$$\begin{aligned} P_3 \overset{\text{def}}{\equiv} \text{if } &\text{Happens}\big(@\text{T}\big(\text{Receive}(q, m)\big), t_i\big) \rightarrow \\ &\exists p\text{: Happens}\big(@\text{T}\big(\text{Send}(p, m)\big), t_j\big) \wedge t_j < t_i \quad \text{where } t_i \text{ and } t_j \in \text{Time} \end{aligned}$$

Formulating axiom for Property 4 (dispatch distinctiveness) This axiom denotes that two different events of viewing dispatches with same content should not happen in the receiver side (P_4).

$$\begin{aligned} P_4 \overset{\text{def}}{\equiv} \text{if } &\text{Happens}\big(@\text{T}\big(\text{Receive}(p, m)\big), t_i\big) \wedge \\ &\text{Happens}\big(@\text{T}\big(\text{Receive}(p, m)\big), t_j\big) \rightarrow i = j \quad \text{where } t_i \in \text{Time} \end{aligned}$$

As the last rows of Tables 1 and 3 show, it is assumed that the sender or recipient will eventually recover when crashes. This property is called runtime integrity. In other words, the first event after failure is recovery (P_5).

$$\begin{aligned} P_5 \overset{\text{def}}{\equiv} &\text{Happens}\big(@\text{T}(\text{Crash}), t_i\big) \rightarrow \\ &\quad \text{Happens}\big(@\text{T}(\text{Recover}), t_j\big) \wedge t_i < t_j \wedge \\ &\exists!\, \alpha \in \text{Event: Happens}\big(@\text{T}(\alpha), t_k\big) \wedge t_i < t_k < t_j \\ &\text{where } \exists! \text{ means “There is not.”} \end{aligned}$$

The last property to be addressed is virtual synchrony stated as causal sequence in Sect. 3. As stated in Sect. 3, the sequence considering just the order between two causal dispatches is used to increase the performance in distributed systems.

First, we define the causal relation between two events as P_6 where Pid is process ID and α, β and δ are events. The first two predicates in the right-hand side of P_6 denote that event β *causally relates* to α (indicated as $\alpha \Rightarrow \beta$) if those were caused by same process. The next two predicates in the right-hand side of P_6 state that the event “receiving a dispatch” *causally relates* to event of sending same dispatch. The third part of P_6 states that there exists an event like δ causally relating to α, and there exists another event like β causally relating to δ.

$$\begin{aligned} P_6^{\text{def}} \equiv \alpha \quad \Rightarrow \quad \beta^{\text{def}} \equiv \quad & \big[\text{Happens}(\alpha, t_i) \wedge \text{Happens}(\beta, t_j) \wedge \\ & \text{Pid}(\alpha) = \text{Pid}(\beta) \wedge t_j \geq t_j\big] \\ \vee & \\ & \big[\text{Happens}\big(@\text{T}\big(\text{Send}(p, m)\big), t_i\big) \wedge \\ & \text{Happens}\big(@\text{T}\big(\text{Receive}(q, m)\big), t_i\big)\big] \\ \vee & \\ & [\exists\delta{:}\ \alpha \quad \Rightarrow \quad \delta \wedge \\ & \quad\ \ \delta \quad \Rightarrow \quad \beta] \end{aligned}$$

Regarding the definition of causal relation between two events, we define property *causal delivery* between two dispatches and afterwards define the property *reliable causal delivery.*

Causal delivery If dispatch m_0 is sent before dispatch m_1 and m_1 is related to m_0, then every process receiving those should view m_0 before m_1 (P_7). Relation $\alpha \Rightarrow \beta$ denotes that β is causally related to α.

$$\begin{aligned} P_7^{\text{def}} \equiv \big[&\text{Happens}(\alpha, t_1) \wedge \text{Happens}(\beta, t_2) \wedge t_1 < t_2 \wedge (\alpha \Rightarrow \beta) \wedge \\ &\text{Happens}(\delta, t_3) \wedge \text{Happens}(\sigma, t_4)\big] \rightarrow t_3 < t_4 \end{aligned}$$

where

$$\begin{aligned} &\alpha = @\text{T}\big(\text{Send}(p_0, m_0)\big), \qquad \beta = @\text{T}\big(\text{Send}(p_1, m_1)\big), \\ &\delta = @\text{T}\big(\text{Receive}(q, m_0)\big), \qquad \sigma = @\text{T}\big(\text{Receive}(q, m_1)\big) \\ &\quad \text{where } t_1, t_2, t_3 \text{ and } t_4 \in \text{Time} \end{aligned}$$

Reliable causal delivery If dispatch m_0 is sent before dispatch m_1 and m_1 is causally related to m_0, then every recipient viewing m_1 shall view m_0 before m_1 (P_8). Relation $\alpha \Rightarrow \beta$ denotes that β is causally related to α. While P_7 shows just the constraint of causal relation, P_8 states that both cause and effect dispatches should be viewed by recipients.

$$\begin{aligned} P_8^{\text{def}} \equiv \big[&\big(\text{Happens}(\alpha, t_1) \wedge \text{Happens}(\beta, t_2) \wedge t_1 < t_2\big) \wedge (\alpha \Rightarrow \beta) \wedge \\ &\big(\text{Happens}(\sigma, t_4)\big)\big] \rightarrow \text{Happens}(\delta, t_3) \quad \text{where } t_3 < t_4, \end{aligned}$$

$$\begin{aligned} &\alpha = @\text{T}\big(\text{Send}(p_0, m_0)\big), \qquad \beta = @\text{T}\big(\text{Send}(p_1, m_1)\big), \\ &\delta = @\text{T}\big(\text{Receive}(q, m_0)\big), \qquad \sigma = @\text{T}\big(\text{Receive}(q, m_1)\big), \\ &\quad \text{where } t_1, t_2, t_3 \text{ and } t_4 \in \text{Time} \end{aligned}$$

In reliable causal property, if a related dispatch is viewed, then a cause dispatch has already been viewed. This property states two conclusions: (1) the existence of a causal dispatch and (2) the ordering of causal dispatches. In an unreliable causal delivery, just the latter is considered.

6 Practicing axioms

This section aims to show how to use the axioms formulated in Sect. 5. They could be used to verify actual behavior sender and recipient processes against the expected behavior and properties of reliable dispatching. To this end, we acquire constraints should be satisfied by the sender and recipient when they are involved in sending or viewing some dispatch. To verify the constraints, the verifier should receive a notification including the status of sender/recipient from sender/receiver processes. Therefore, the notification codes should weave into the processes places that send or receive a dispatch.

The verification process of the constraints is centrally carried out by practicing axioms acquired in Sect. 5. Each constraint connects two properties so that its premise and conclusion parts hold in two separate properties. The nondistributed verifier makes verification process of the constraints smooth. Therefore, if the premise of a constraint is in property like Property 3 (dispatch entirely axiom) and its conclusion is in another property, we can verify the constraint as it exists if the verifier is centralized. Consider, for instance, property dispatch entirely stated as P_3. The premise part of this property shows the event "viewing a dispatch by a recipient":

$$\text{Happens}\big(@\text{T}\big(\text{Receive}(q,m)\big),t_i\big)\quad \text{(the } P_3 \text{ premise part)}$$

and the conclusion part shows event "sending a dispatch by sender":

$$\text{Happens}\big(@\text{T}\big(\text{Send}(p,m)\big),t_j\big)\wedge t_j<t_i\quad \text{(the } P_3 \text{ conclusion part)}$$

Holding the premise part will mean that a recipient has received a dispatch and given that the notification code has been weaved into the recipient, the recipient: (1) informs the verifier of its mode by sending a message and (2) transmits an acknowledgment to the sender process. Having received the message from the recipient, the verifier verifies holding the conclusion part of P_3 (stated above).

By the conclusion part of P_3, the verifier understands: "If the sender has already sent a dispatch, it would be in the Awaiting mode." Accordingly, the verifier expects to hold the axiom (4) (which stated in Sect. 5):

$$\begin{aligned}\text{MI}_3(t)^{\text{def}}\equiv{}&\text{HoldsAT}(\text{Awaiting},t)\rightarrow\\&\big[{\sim}\text{HoldsAt}(\text{Send},t)\wedge{\sim}\text{HoldsAt}(R_{\text{ACK}},t)\wedge{\sim}\text{HoldsAt}(\text{TOut})\big]\end{aligned}$$

Also, by the conclusion part of P_3, the verifier understands the sender would receive an acknowledgment. Accordingly, the verifier expects to hold the axiom (7) (which stated in Sect. 5):

$$\begin{aligned}\text{TI}_2(t)^{\text{def}}\equiv{}&\text{Initiates}\big(@\text{T}(\text{R}_{\text{ACK}}),\text{Success},t\big)\leftarrow\\&\text{HoldsAt}(\text{Awaiting},t)\wedge\\&\big[{\sim}\text{HoldsAt}(\text{Send},t)\wedge{\sim}\text{Happens}\big(@\text{T}(\text{Send}),t\big)\wedge\\&{\sim}\text{HoldsAt}(\text{Tout},t)\wedge{\sim}\text{Happens}\big(@\text{T}(\text{Tout}),t\big)\wedge\\&{\sim}\text{HoldsAt}(\text{R}_{\text{ACK}},t)\wedge\text{Happens}\big(@\text{T}(\text{R}_{\text{ACK}}),t\big)\big]\end{aligned}$$

Table 5 History structure

Event name	
Absolute index	
Index	
Time	Content
Time	Content
Time	Content
⋮	⋮
Time	Content

Since the axiom (7) includes axiom (4), just (7) is considered. To hold the axiom (7), its predicates constituting 7 *constraints* should hold:

$$\begin{array}{ll} 1\text{: HoldsAt(Awaiting}, t), & 2\text{: }{\sim}\text{HoldsAt(Send}, t), \\ 3\text{: }{\sim}\text{Happens}\big(\text{@T(Send)}, t\big), & 4\text{: }{\sim}\text{HoldsAt(Tout}, t), \\ 5\text{: }{\sim}\text{Happens}\big(\text{@T(Tout)}, t\big), & 6\text{: }{\sim}\text{HoldsAt}(\text{R}_{\text{ACK}}, t), \\ 7\text{: Happens}\big(\text{@T}(\text{R}_{\text{ACK}}), t\big) & \end{array}$$

These constraints are of two kinds: temporal predicates represented by the HoldsAt predicate and happening events represented by the Happens predicate. As the constraints show, it is clear that in the event of receiving an acknowledgment, constraint 7 takes effect (shown by @T(R_{ACK}), t). Following this event, the verifier monitors variables related to temporal predicates Send and Tout. If the values of these variables are false (constraints 2 and 4), then it verifies that whether the sequel of constraint 1 has already been observed or not. As stated in the second step, the sequel of this constraint is the event of sending a dispatch by sender in the past, i.e.,

$$\text{Happens}\big(\text{@T}\big(\text{Send}(p, m)\big), t_j\big) \wedge t_j < t_i$$

Accordingly, there should be a mechanism by which previous events can be remembered. To accomplish this mechanism, we use a data structure as an event history managed by the verifier. Table 5 shows the implementation of the event history structure managed by the verifier. The history is constructed as a finite circular queue consisting of time and value pairs where: (1) the *index* refers to the most current event, (2) the *absolute index* counts the number of events and (3) the *event ID* is specified individually by "the name of event." When an event happens, time and its content (for example, the dispatch content) are stored in the queue.

7 Considering implementation

This section deals with cases we have in implementation of the constraints partly acquired in Sect. 6. We used an MPI (Message Passing Interface) called MPICH2 on a clustered multicore computer for our work. MPI is a communication protocol used in cluster systems and recently on clustered multicore computers [6]. MPI purveys a message passing interface for the programmers, with a semantic to force constraints in an implementation.

Process 0	Process 1
define TAG 99	MPI_Status status;
float a[10];	float b[20];
int dest=1;	int sender=0;
MPI_Send(a, 10, MPI_FLOAT, dest, TAG, MPI_COMM_WORLD);	MPI_Recv(b, 20, MPI_FLOAT, sender, TAG, MPI_COMM_WORLD)

Fig. 2 Sending and receiving dispatches in MPI

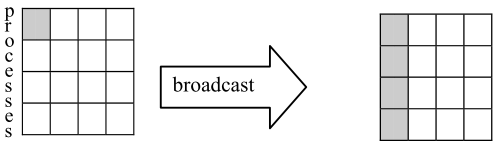


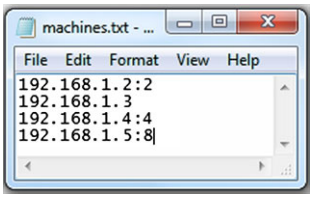
Fig. 3 Broadcasting data to 4 processors in MPI

MPI has become a standard for interprocess communication running on cluster systems and Implementations of MPI usually have routines to use in the C++ and Java programming languages. MPI is able to purvey virtual topology, synchronization, and communication between processes of clustered multicore computers. Although programs using MPI work with processes, programmers understand the processes as processors. Therefore, each core is assigned a single process at runtime.

The MPI library consists of send/receive operations by exchanging data between the sender and recipient processes. It also is able to synchronize nodes and provides information such as the number of processes involving the computation. Interprocess communication may be synchronous, asynchronous, and buffered enabling us to use strong and weak semantics for the synchronization purpose. MPI uses commands *send* and *receive* to point-to-point connection and the command *broadcast* to one-to-all connection where *address* is a pointer to dispatch address, *count* is the number of dispatches to be sent, *type* is the dispatch type, *dest* is the target process to receive the sent dispatch, and *comm* is a communicator defining a group of process.

```
MPI_Send(address, count, type, dest, tag, comm)
MPI_Recv(address, count, type, dest, tag, comm, status)
MPI_Bcast(data, count, type, scr, comm)
```

Figure 2 shows the code of point to point connection between processes 0 and 1 and Fig. 3 shows broadcasting a dispatch to 4 processors in MPI. MPICH2 is an implementation of MPI for efficient support of communication on clustered multicore architectures [21]. To exploit MPICH2, we used Visual C++ in Visual Studio 2008 configured by MPICH2 on four multicore (octa-core) computers. Spawning processes for execution is under tutelage of a station called head node. Via the head node, we specified the number of processes and computers on which the processes to be executed. The head node distributes processes on computers, collects the execution results of each process and shows the results on the head node screen. The command to be executed on the head node has the form of *mpiexec -f machinefile -n 32 prog* where *machinfile* denotes a file on the head node including 4 IP addresses. Each



```
192.168.1.2:2
192.168.1.3
192.168.1.4:4
192.168.1.5:8
```

Fig. 4 Machine file in MPICH2 consisting of 4 host nodes with different assigning processes

Table 6 Network and expected sequence of dispatches

Dispatch	M_1	M_2	M_3	M_4	M_5	M_6
Network sequence	1	2	3	4	5	6
Expected sequence	3	2	4	1	5	6

address indicates a host machine on which the program *prog* is executed as a process. Considering an octa-core CPU in each host machine, totally 32 concurrent processes will be executing on 4 nodes. If each host node CPU has different core, we can assign different processes to each node depending on CPU cores of nodes. Given that the machine file consists of 4 host nodes specified by IP addresses (Fig. 4). By running the command *mpiexec -machinefile -n 15 prog*, Process ranks 0 and 1 are to run on the first, rank 2 on the second, ranks 3–6 on the third and ranks 7–14 on the last IP.

For brevity, we state cases we considered for orderings FIFO and causal (stated in Sects. 3 and 4) in MPICH2. Each dispatch in MPICH2 will take a label called *tag* when the dispatch is transmitted. We used the tags to force FIFO sequence when the verifier notifies us of dissatisfaction of the sequence against constraints we acquired as axioms (See Sect. 6). Table 6 shows tagged dispatches in the first row (M_1 to M_6) and the sequence was intended to be delivered by the system is in the second row. The third row shows how sequence we expected in the recipient side for the dispatches. As the third row of Table 5 shows, M_4 should be first delivered. Accordingly, we buffered dispatches viewed before M_4 and delivered M_4 to the recipient on receiving. Afterward, dispatches M_2, M_1, and M_3 were respectively delivered to the recipient. MPICH2 facilities such as tag, buffering, and making sequencing enabled us to force the FIFO sequence.

To exercise the causal ordering, we used vector clock mechanism explained in [3] where a vector is thought for each process. The vector has n rooms long that each room is thought for a logical clock of a process. A room is represented as $VC_i[j]$ where index "i" denotes the process number and "j" is the logical clock value of process "j." Accordingly, $VC_i[i]$ denotes the logical clock value of process "i" in vector "i."

The first logical clock value of each process is zero and it will be increased when the process sends a dispatch. Consider Fig. 5 where vectors VC_0, VC_1, and VC_2 are thought for processes P_0, P_1, and P_2. Since there are 3 processes, each clock vector has 3 rooms. First, vectors are set to (0, 0, 0), i.e., value of all logical clocks is set to zero. On transmitting dispatch m, $VC_0[1]$ is increased and it is transmitted with m to P_1. Having viewed m, process P_1 means to transmit dispatch m^*. This means that m^* is causally related to m; so, P_1 sets $VC_1[1]$ to $VC_0[1]$. Also, it increases its logical clock value, i.e., $VC_1[2]$ because it means to transmit a dispatch called m^*.

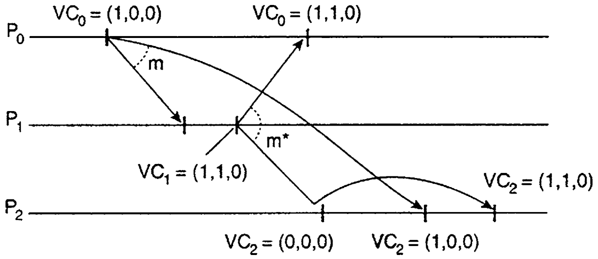


Fig. 5 Vector clocks in three processes

Given that the second dispatch, m^*, arrives before dispatch m; delivery of dispatch m^* to process P_2 should be postponed until delivery of dispatch m.

According to the description mentioned above, each process is to update its vector. To exercise the causal ordering, P_i should verify conditions C_1 and C_2 when it receives dispatch m with time stamp ts(m) from P_j. Time stamp ts(m) indicates the vector clock. Delivery of dispatch m is postponed until the conditions are met; so, satisfaction of the conditions denotes viewing the causal dispatch.

$$C_1.\ \text{ts(m)}[j] = \text{VC}_i[j] + 1$$

$$C_2.\ \text{ts(m)}[k] = \text{VC}_i[k] \quad \text{for all } k \neq j$$

C_1 specifies the next dispatch that process P_i expects to be sent by process P_j and C_2 specifies that all connected dispatches with m should be viewed by P_i before m. In other words, before delivery of m to process P_i, the process should view all dispatches that P_j viewed prior to m.

Figure 4 shows the causal ordering we implemented. The boxes tagged by Causal are object classes where dispatches are objects and class methods are used to satisfy the causal ordering. In Fig. 4, the process “Rank 0”, broadcasts “Msg 0” to the processes “Rank 1” and “Rank 2” and the process “Rank 1” transmits “Msg 1” to process “Rank 2” when it viewed “Msg 0”. Adherence to ordering in delivery of causal dispatches “Msg 0” and “Msg 1” was considered by the second Causal class.

As Fig. 4 shows, process “Rank 0” sends dispatch “Msg 0”, process “Rank 1” sends dispatch “Msg 1” after receiving dispatch “Msg 0” and process “Rank 2” receives dispatches “Msg 0” and “Msg 1.” However, since dispatch “Msg 1” arrives prior to dispatch “Msg 0” and “Msg 0” is the cause of “Msg 1,” the second Causal class first arranges the dispatches in their timestamp using a vector and then delivers them to process “Rank 2.”

Figure 5 shows Message and Causal classes whose objects are included in each process and consists of three attributes for: (1) numbering processes send dispatches (*senderindex*), the dispatch content and broadcast time (indicated by a counter) of dispatches (*timestamp*). The methods *toString*() and *fromString*() have been intended to implement actions (the read and set) of the vector clock. Figure 6 shows structure of the causal structure depicted in Fig. 7 consisting of attributes *currindex* (the indicator of the latest arrived dispatch), *messageCount* (the number of dispatches), *messages* (a vector for saving and arranging dispatches), *timestamp* (an array of dispatches counters) and *arrivedmsg* (an auxiliary array of dispatches counters). Figure 11 in the Appendix shows the Message class code consisting of the described attributes and methods. The Causal class has a lengthy code and was not shown.

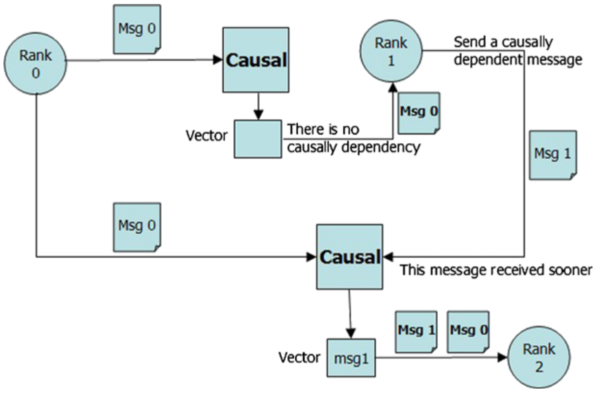

Msg 0
Rank 0
Causal
Rank 1
Send a causally dependent message
Msg 0
Vector
There is no causally dependency
Msg 1
Msg 0
Causal
This message received sooner
Msg 1
Msg 0
Vector
msg1
Rank 2

Fig. 6 Causal classes for implementing the causal ordering

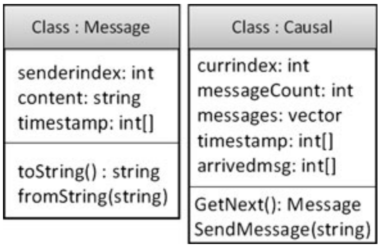
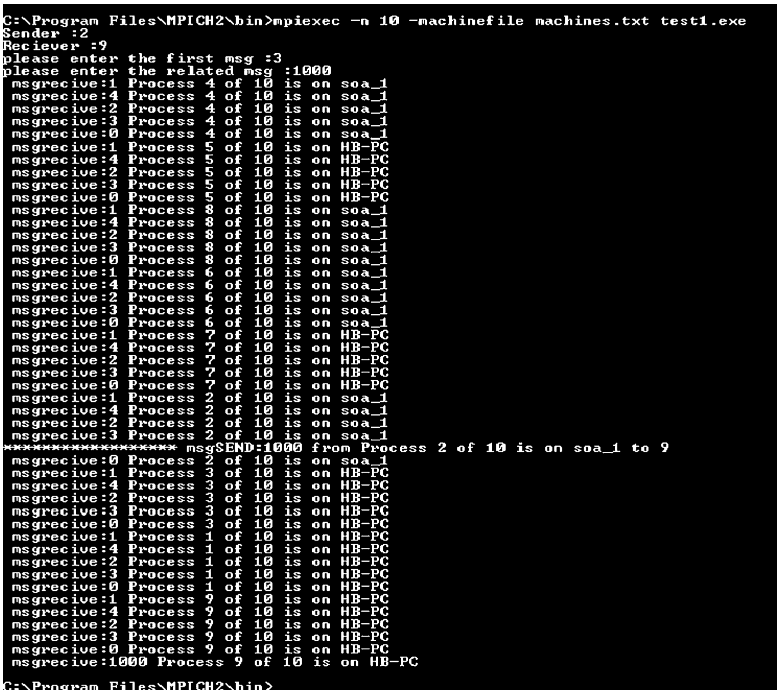


Fig. 7 The structure of message and causal classes

Figure 8 shows the result of the causal ordering implemented by causal classes (Fig. 6) on a cluster with 2 sender and 9 receiver processes. In Fig. 8, *megrecive n* indicates the number of dispatch has been received by a receiver process where dispatch *megrecive* 1 is the cause of *megrecive* 3. As the figure shows, our causal implementation of causal classes executed the causal ordering because delivery of dispatch *megrecive* 1 is always carried out before dispatch *megrecive* 3.

To consider the execution time of practicing the orderings, we configured a cluster with 5, 20, 50, and 100 processes. Figure 9 shows the execution time for applying the FIFO, causal and total orderings. Note that we used interfaces for implementation, which take a few seconds and included in the execution time. The x and y axes indicates the number of processes and the consumed execution time in seconds.

An application of causal and FIFO orderings is execution of consistency in data replication in distributed systems. To make distributed systems fault tolerant, data are replicated across distributed nodes. However, data replication is subject to data inconsistency because of network delay in updating data. To prevent such an inconsistency, causal and FIFO orderings should be considered. Consider Fig. 10 where process P_1 has updated variable x by storing value a in x (indicated by $\mathrm{W}(x)a$); afterward, process P_2 reads value a from variable x (indicated by $\mathrm{R}(x)a$) in order to update x by storing value b in it. This scenario shows that value b is causally related to value a. Accordingly, each process receives values a and b, it should receive a before b. Since such ordering has been respected by processes P_3 and P_4, we say that the data store is causally consistent. Since there is no causal relation between values



```
C:\Program Files\MPICH2\bin>mpiexec -n 10 -machinefile machines.txt test1.exe
Sender :2
Reciever :9
please enter the first msg :3
please enter the related msg :1000
 msgrecive:1 Process 4 of 10 is on soa_1
 msgrecive:4 Process 4 of 10 is on soa_1
 msgrecive:2 Process 4 of 10 is on soa_1
 msgrecive:3 Process 4 of 10 is on soa_1
 msgrecive:0 Process 4 of 10 is on soa_1
 msgrecive:1 Process 5 of 10 is on HB-PC
 msgrecive:4 Process 5 of 10 is on HB-PC
 msgrecive:2 Process 5 of 10 is on HB-PC
 msgrecive:3 Process 5 of 10 is on HB-PC
 msgrecive:0 Process 5 of 10 is on HB-PC
 msgrecive:1 Process 8 of 10 is on soa_1
 msgrecive:4 Process 8 of 10 is on soa_1
 msgrecive:2 Process 8 of 10 is on soa_1
 msgrecive:3 Process 8 of 10 is on soa_1
 msgrecive:0 Process 8 of 10 is on soa_1
 msgrecive:1 Process 6 of 10 is on soa_1
 msgrecive:4 Process 6 of 10 is on soa_1
 msgrecive:2 Process 6 of 10 is on soa_1
 msgrecive:3 Process 6 of 10 is on soa_1
 msgrecive:0 Process 6 of 10 is on soa_1
 msgrecive:1 Process 7 of 10 is on HB-PC
 msgrecive:4 Process 7 of 10 is on HB-PC
 msgrecive:2 Process 7 of 10 is on HB-PC
 msgrecive:3 Process 7 of 10 is on HB-PC
 msgrecive:0 Process 7 of 10 is on HB-PC
 msgrecive:1 Process 2 of 10 is on soa_1
 msgrecive:4 Process 2 of 10 is on soa_1
 msgrecive:2 Process 2 of 10 is on soa_1
 msgrecive:3 Process 2 of 10 is on soa_1
****************** msgSEND:1000 from Process 2 of 10 is on soa_1 to 9
 msgrecive:0 Process 2 of 10 is on soa_1
 msgrecive:1 Process 3 of 10 is on HB-PC
 msgrecive:4 Process 3 of 10 is on HB-PC
 msgrecive:2 Process 3 of 10 is on HB-PC
 msgrecive:3 Process 3 of 10 is on HB-PC
 msgrecive:0 Process 3 of 10 is on HB-PC
 msgrecive:1 Process 1 of 10 is on HB-PC
 msgrecive:4 Process 1 of 10 is on HB-PC
 msgrecive:2 Process 1 of 10 is on HB-PC
 msgrecive:3 Process 1 of 10 is on HB-PC
 msgrecive:0 Process 1 of 10 is on HB-PC
 msgrecive:1 Process 9 of 10 is on HB-PC
 msgrecive:4 Process 9 of 10 is on HB-PC
 msgrecive:2 Process 9 of 10 is on HB-PC
 msgrecive:3 Process 9 of 10 is on HB-PC
 msgrecive:0 Process 9 of 10 is on HB-PC
 msgrecive:1000 Process 9 of 10 is on HB-PC

C:\Program Files\MPICH2\bin>
```

Fig. 8 Result of causal ordering implemented by causal classes on a cluster with 11 processes

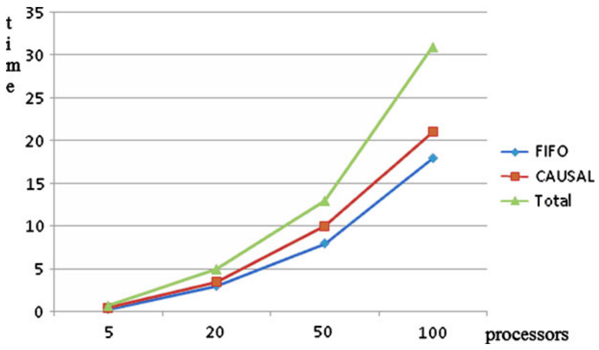
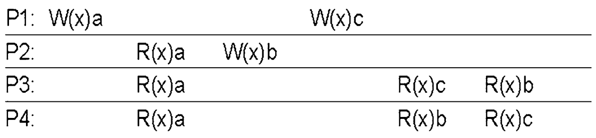


Fig. 9 The execution time of applying orderings in dispatch passing on a cluster



P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Fig. 10 The causally consistent data store

a and c, those may be received by processes in any ordering. In other words, Fig. 10 shows: (1) causally related writes have been seen by all processes in same order and (2) concurrent writes have been seen in a different order.

8 Conclusions and future work

This study presented an approach for formulating axioms can be used to verify broadcasting reliable dispatches. Contribution of this study was presenting a constructive method by which one can produce axioms that are susceptible verification of inter-process communications in the cluster systems.

Safety and reliability of processes depend on consistency rules we derived and specified as invariants to be verified against runtime behavior of processes. Since some invariants are difficult to verify using static analysis, runtime verification of invariants against dynamic behavior of processes is suggested. Contribution of this paper was description of a framework for deriving invariants and runtime verification of communicating processes against the invariants. The framework derived formal specification of state and transition invariants using tabular automata where state invariants were used to determine rightful synthesis of values of variables at states of the processes and transition invariants were used to determine rightful changes of values of variables after the state transitions. Runtime verification was used to check holding: (1) state invariants on current runtime states of processes and (2) transition invariants after making the transitions associated with the current state.

In dynamic systems, on the other hand, the system behavior is governed by: (1) the events happened in the system environment and (2) constraints. The events cause changing the system states and the constraints denote properties to be satisfied in the system states and states change. Since normally the events and constraints are independently specified, other contribution of this paper was establishing connection between them by invariants.

Similar to our work, Drusinsky and Shing formulated axioms called assertions from Message Sequence Chart (MSC) [7]. However, MSC cannot show the state of the communicating processes because it is thought just to show sequencing dispatches. This is why [7, 11, 18, 25, 26] exploited some automaton in their works. To resolve this drawback, Drusinsky et al. used UML Statecharts [8] to specify the expected behavior of distributed applications and formulated verification axioms called assertions from the Statecharts.

The connection between formal methods and MPI explained in Sect. 7 is a new trend and future work in province of distributed dispatch passing in networked systems. Although some researches have already carried out studies in this field [16, 17], according to [19] they did next to nothing for helping MPI by formal methods.

The approach we presented in this paper used a centralized verifier, but the improvement of performance is necessary when we aim to carry out timely verification of temporal constraints in the observation of real-time systems. One way to improve this type of performance is to minimize the time duration of transmitting events to the central verifier so that the verifier can manage timely control of the system environment. To this end, we suggest the idea of distributed verifiers for future work. By the distribution of verifiers, there is no need to spread verifiable messages between communicating processes and the verifier.

Acknowledgement The author is grateful to University of Kashan for supporting this work by grant No. 3055/2.

Appendix: Figure 11

```
public class Message
{
public int senderindex;
public int[] timestamp;
public String content;
public Message()
{
senderindex=-1;
timestamp=null;
content="";
}
public Message(int senderindex, int[] timestamp, String content)
{
this.senderindex= senderindex;
this.timestamp= timestamp;
this.content= content;
}
@Override
public String toString()
{
String strtimestamp= "";
for (int i=0; i<timestamp.length; i++)
strtimestamp += timestamp[i] + ",";
return strtimestamp + "///" + senderindex + "///" + content;
}
public void fromString(String message)
{
String[] parts= message.split("///");
senderindex= Integer.valueOf(parts[1]);
content= parts[2];
parts = parts[0].split(",");
timestamp= new int[parts.length];
for (int i=0; i<parts.length; i++)
timestamp[i] = Integer.valueOf(parts[i]);
}
}
```

Fig. 11 The implementation code of the Message class

References

1. Babamir SM (2012) Constructing formal rules to verify message communication in distributed systems. J Supercomput 59(3):1396–1418
2. Babamir SM (2012) Specifying and modeling multicast communication in CBCAST protocol. Proc Rom Acad, Ser A : Math Phys Tech Sci Inf Sci 13(3):261–268

3. Baldoni R, Raynal M (2002) Fundamentals of distributed computing: a practical tour of vector clock systems. IEEE Distrib Syst, 3
4. Birman KP (2005) Reliable distributed systems: technologies, Web Services and applications. Springer, Berlin
5. Chockler GV, Keidar I, Vitenberg R (2001) Group communication specifications: a comprehensive study. ACM Comput Surv 33(4):427–469
6. Dózsa G, Kumar S, Balaji P, Buntinas D, Goodell D, Gropp W, Ratterman J, Thakur R (2010) Enabling concurrent multithreaded MPI communication on multicore petascale systems. In: Proc of the 17th European MPI users' group meeting conference on recent advances in the message passing interface, Heidelberg, 2011. Springer, Berlin, pp 11–20
7. Drusinsky D, Shing M (2007) Verifying distributed protocols using MSC-assertions, run-time monitoring and automatic test generation. In: the 18th IEEE/IFIP international workshop on rapid system prototyping (RSP'07), pp 82–88
8. Drusinsky D, Shing M, Demir KA (2007) Creating validating embedded assertion statecharts. IEEE Distrib Syst Online 8(5):1–12
9. Fulmare N, Yadav D (2010) Rigorous analysis of byzantine causal order using event-B. In: Proc of the international conference and workshop on emerging trends in technology, pp 723–726
10. Kostin A, Ilushechkina L (2010) Modeling and simulation of distributed systems. World Scientific, Singapore
11. Khanna G, Varadharajan P, Bagchi S (2006) Automated online monitoring of distributed applications through external monitors. IEEE Trans Dependable Secure Comput 3(2):115–129
12. Kruger IH, Meisinger M, Menarini M (2007) Runtime verification of interactions: from MSCs to aspects. In: Proc. of the 7th international workshop on runtime verification (RV 2007). Lecture notes in computer science, vol 4839. Springer, Berlin, pp 63–74
13. Kruger IH, Meisinger M, Menarini M (2010) Interaction-based runtime verification for systems of systems integration. J Log Comput 20(3):725–742
14. Lane RG, Daniels S, Yuan X (2007) An empirical study of reliable multicast protocols over ethernet-connected networks. Journal of Performance Evaluation 64(3):210–228
15. Lomazova IA (1997) On proving large distributed systems: Petri net modules verification. In: Proceedings of the 4th international conference on parallel computing technologies. Lecture notes in computer science, vol 1277. Springer, Berlin, pp 70–75
16. Li G, Delisi M, Gopalakrishnan G, Kirby RM (2008) Formal specification of the MPI-2.0 standard in TLA+. In: Proc of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, pp 283–284
17. Li G, Palmer R, DeLisi M, Gopalakrishnan G, Kirby RM (2011) Formal specification of MPI 2.0: case study in specifying a practical concurrent programming API. Sci Comput Program 76(2):65–81
18. Sen K, Vardhan A, Agha G, Rosu G (2004) Efficient decentralized monitoring of safety in distributed systems. In: Proc of 26th international conference on software engineering, pp 418–427
19. Siegel SF, Gopalakrishnan G (2011) Formal analysis of message passing. In: Proc of the 12th international conference on verification, model checking, and abstract interpretation (VMCAI'11). Springer, Berlin, pp 2–18
20. Tanenbaum AS, Steen MV (2007) Distributed systems: principles and paradigms. Prentice Hall, New York
21. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in MPICH. Int J High Perform Comput Appl 1(19):49–66
22. Tsaur W, Horng S (2001) Auditing causal relationships of group multicast communications in group-oriented distributed systems. J Supercomput 18:25–45
23. Tsuchiya T, Schiper A (2011) Verification of consensus algorithms using satisfiability solving. J Parallel Distrib Comput 23:341–358
24. Yadav D, Butler M (2005) Application of event-B to global causal ordering for fault tolerant transactions. In: Workshop on rigorous engineering of fault tolerant systems (REFT2005), pp 93–102
25. Zhang F, Qi Z, Guan H, Liu X, Yang M, Zhang Z (2009) FiLM: a runtime monitoring tool for distributed systems. In: the 3rd IEEE international conference on secure software integration and reliability improvement, pp 40–46
26. Zulkernine M, Seviora RE (2002) A compositional approach to monitoring distributed systems. In: Proc of the international conference on dependable systems and networks, pp 763–772