Turkish Journal of Computer and Mathematics Education Vol.12 No. 11 (2021), 2748- 2762
Research Article

Formal Development of a Fault Tolerant Distributed Checkpoint Process Using Event-B

Bal Krishna Saraswat [a], Dr. Raghuraj Suryavanshi [b] , Dr. Divakar Yadav

[a] Assistant Professor, SRM Institute of Science & Technology, NCR Campus, Modinagar.
[b] Assistant Professor, Pranveer Singh Institute of Technology, Kanpur.
[c] Professor, Institute of Engineering & Technology, Lucknow.

Article History: Received: 11 January 2021; Revised: 12 February 2021; Accepted: 27 March 2021; Published online: 10 May 2021

Abstract: It is really a challenging effort to develop and verify the distributed fault tolerant system. This is because of the communication primitives available for distributed system are not very strong. In Distributed systems processes cooperate and communicate with each other by message exchange. Fault tolerant broadcast mechanisms are must for the construction of the reliable Distributed system. We can achieve the fault tolerance by introducing the Total and causal order broadcast. Use of formal approaches allows us to build fault tolerant distributed systems. In this article, we introduce a systematic system development in which processes communicate through broadcast and messages are exchanged using a total order broadcast mechanism. Total order broadcast primitives were proposed to enable broadcasting system for checkpoint process and allow fault-tolerant co-operation in a distributed network between the sites. We explain how an Event-B refinement-based methodology can be used for the systematic creation of Checkpoint process using total order broadcast models. We introduce a model of broadcasting system for total order and then ensure that the model maintains appropriate ordering attributes while taking the checkpoints. Consequently, in a succession of refinement stages we illustrate how to accurately integrate an abstract total order utilizing the concept of sequence number. This approach allows one to discharge proof obligations due to consistency and refinement checking. To fulfil the proof obligations, it is necessary to find out invariant which define the interconnection among the abstract total order and the process responsible for it.

Keywords: Event-B, Formal Verification, Distributed systems, Recovery, Checkpoint, Total Order, Fault Tolerant, Formal Specifications, Checkpoint Number, Formal Methods, Consistent State, Global State, Rodin

1. Introduction

Checkpointing is the method of saving the failure free state of process on to the stable storage and restart the computation from that failure free state which was saved on the stable storage in the process of Rollback recovery (Kim & Park, 1993). For the Rollback recovery, all the processes must be agreed on to some Consistent Global Checkpoint to restart the computation (Manivannan, Netzer, & Singhal, 1997). To make the consensus on the consistent global checkpoint, the processes have to communicate with each other through message passing. In Distributed Systems the interaction among the processes is only through the exchange of messages because they do not share memory and for the rollback recovery of the distributed system all the processes must form a global state and that must be communicated to all the processes (Leu & Bhargava, 1988) (Bal Krishna Saraswat, D. Y., & Raghuraj Suryavanshi, 2018).

Implementation of distributed systems and applications are very hard. This is because of the communication primitives available for the broadcast and may also be because of the inevitable concurrency in distributed system, associated with the complexity of offering global control. This challenge would be significantly overcome by depending on group communication primitive that delivers better assurances than conventional point-to-point communication. The weakest group communication primitive is the Reliable Broadcast. In reliable broadcast, all the processes make their consensus on the set of messages they deliver (Hadzilacos & Toueg, 1994). One of the well-known communication primitives is Total Order Broadcast (Defago, Schiper, & Urban, 2004), which guarantees that all such processes deliver messages to a group of processes in the same sequence. Total order broadcast is a powerful primitive that plays a key role in the execution of state machine method, (also called active replication) (Lamport, 1978) (Schneider, 1990). It is also shown that the use of Total Order Broadcast primitive can improve the performance of the replicated databases (Pedone, Guerraoui, & Schiper, 1998) (Kemme, Pedone, Alonso, Schiper, & Wiesmann, 2003).

Distributed system's reliability is a key design measure of success for building or upgrading established distributed services. Reliability applies both to a system's resistance to different sorts of failure and its ability to survive from them (Holliday, 2001). By displaying well-defined behaviors that support the recovery-friendly operation, a process can be configured to be fault tolerant. In Order to ensure the reliability of distributed system we need to ensure the reliability of each independent parameter or factors involved in distributed systems before forecasting or determining the reliability of the whole system, and incorporating a consistent mechanism for detecting faults and restoring faults to provide end users a smooth interaction (Ahmed & Wu, 2013).

In this research article we are designing a robust checkpoint algorithm with the help of total order broadcast primitive using Event-B. The main characteristics of Event-B include the use of set theory as a modeling language, description of various degrees of abstraction using refinement, and the use of mathematical proof to validate uniformity among various degrees of refinement. Event-B is a successor of B Method (J. Abrial, 1996) and action system (Back, 1989). We are using the Rodin (J.-R. Abrial, 2007) (J.-R. Abrial et al., 2010) (J.-R. Abrial, Butler, Hallerstede, & Voisin, 2006), which is a model creation tool for Event-B that is open source and expandable.

2. Preliminaries

In this section we are specifying about the process failures, and the message ordering properties available for distributed system. We are also defining some B-notations which are used throughout our model.

2.1 Process Failures

There are many types of failures that are supposed to take place in the Distributed System. A general class of failure models are acknowledged below:

- *Crash failures:* If a system fails, it will cease to work indefinitely. It implies that it avoids writing, distributing, or accepting some response from any operation.
- *Omission failures:* Some activities, such as exchange of messages, are omitted when omission failure occurs.
- *Timing failures:* If a mechanism contradicts the synchronization presumption, it is called a timing fault. In asynchronous structures, this form of error is meaningless.
- *Byzantine failures:* It is the most common class of failure. Any illogical actions are permitted byzantine element. For example, A faulty mechanism may alter the message content, repeat messages, submit spam messages, or even attempt to maliciously break down the whole system.

A correct process is described as a process which does not express any of the shortcomings listed above.

2.2 Ordering Properties

The *Reliable Broadcast* is the weakest class of the broadcast technique available. Hadzilacos and Toueg (Hadzilacos & Toueg, 1994) described the properties of reliable broadcast as-

1. **Agreement:** There is a mutual agreement between all the authenticated process which are defining the system on the set of messages they will deliver to each other.
2. **Validity:** Each and every message will be delivered if it is broadcasted by the authenticated process.
3. **Integrity:** No fake or fraudulent message will ever be delivered.

The properties defined for the *Reliable Broadcast* is sufficient for most of the applications, where ordering of messages is not important. But there are many applications where the ordering of messages is very crucial for the fault tolerant and reliable functioning of the system. There are many broadcast primitives available for the ordering of messages.

The *FIFO* Broadcast technique is the same as *Reliable Broadcast* with the addition of an ordering characteristic, that *messages will be delivered in the same order as they were broadcasted*. But this definition is also having a very fine problem. Suppose a process *pp* broadcasted three messages, namely mm_1, mm_2, and mm_3, in that order, and an authenticated process *pq* delivers the message mm_1, then mm_3, but the process never delivers message mm_2. This is possible as while broadcasting the message mm_2, the process *pp* is suffering from the transient failure (Hadzilacos & Toueg, 1994).

Every message has a *context* without which we can wrongly interpret the message delivered. In some cases, the casual precedence of the messages is important. *FIFO* broadcast technique is sufficient when the context of message *mm* is related to the message which was broadcasted by the same process before broadcasting the message *mm* (Hadzilacos & Toueg, 1994). But if the context of the message mm_1, which is broadcasted by the process *pp*, is related to the context of the message mm_2, which was delivered to the process *pp* before broadcasting the message mm_1. Then *FIFO* broadcast order will no longer work (Baldoni, Mostefaoui, & Raynal, 1996).

Causal Order broadcast technique is the stronger form of the *FIFO* Order broadcast by setting up a *causal precedence* relation message broadcast and delivery (Pedone & Schiper, 1999). Following the *Causal Precedence* relation between the message broadcast and delivery, the *Causal Order* broadcast can be defined as, "If the broadcast of a message *m* causally precedes the broadcast of a message *m`*, then no correct process delivers *m`* unless it has previously delivered *m* (Hadzilacos & Toueg, 1994)."

Every message exchanged in the distributed system is not *causally related* with the broadcast and delivery of the message, and *Causal Order* broadcast does not imply any order of delivery on these messages (Cristian, Aghili, Strong, & Dolev, 1995). Being more precise, the two correct processes may be delivered the messages in different order. This type of discord between the delivery order of the messages is not at all acceptable in some application (Luan & Gligor, 1990). To avoid such type of vulnerabilities, *Total Order* broadcast ensures that all the processes deliver all the messages in same order. *Total Order* broadcast can be defined as "If two correct process *p* and *q*

both delivers messages *mm* and *mm`*, then *q* delivers *mm* before *mm'* if and only if *p* delivers *mm* before *mm'* (Hadzilacos & Toueg, 1994).

2.3 B – Notation

Here we annotate some B notations that are majorly used by our model. A more descriptive justification of these notations is defined in (J. Abrial, 1996; Boulanger, 2014). We are assuming that there are two sets named A & B, then the sign ↔ defines set of relations between A and B as

$$A \leftrightarrow B = \mathbb{P}(A \times B)$$

here × is defined as the Cartesian product of sets A and B. When there is a mapping of elements like a $\in A$ and $b \in B$ in a relation R $\in$ A $\leftrightarrow$ B, this is denoted as $a \mapsto b$. We define the domain of a relation $R \in A \leftrightarrow B$, is set of elements of A that R relates to some set of elements in B is denoted as

$$dom(R) = \{a \mid a \in A \wedge \exists\, b.(b \in B \wedge a \mapsto b \in R)\}$$

Additionally, the range of relation $R \in A \mapsto B$ is expressed as set of elements in B related to some element in A

$$ran(R) = \{\, b \mid b \in B \wedge \exists\, a\,.(a \in A \wedge a \mapsto b \in R\,)\,\}$$

A function is defined as a relation with some constraints. A function is having two types: partial function (⇸) and a total function (→). A partial function from set A to B (A ⇸ B) is a relation which relates an element in A to at most one element in B.

A total function from set A to B (A → B) is a partial function where dom(f)=P, i.e., each element of set A is related to exactly one element of set B. Given f $\in$ A ⇸ B and a $\in$ dom(f), f(a) represents the unique value that a is mapped to by f.

3. System Model

In this section we give a theoretical design of a distributed system. The distributed system which is considered in this paper is defined as:

1. processes do not have any shared memory and they communicate with each other by sending messages over channels.
2. Channels do not lose messages and this is guaranteed by certain end-to - end transmission protocol that helps to make the channels lossless (virtually) and first-in-first-out in message delivery.
3. Processes can fail, and all other processes will be notified in the finite time of failure when a process fails.

Our model system comprises of a set of sites where the set of processes is running. We assumed a set of sites coordinating their checkpoints in a way that the resulting global state is consistent. We have used the logical clock of the Lamport to allocate the timestamp to communication sites and the messages associated. Our prototype saves two forms of checkpoints on stable storage:

- Permanent
- Tentative

A permanent checkpoint cannot be ruled out. It ensures that it does not replay the calculation necessary to enter the checkpointed state. Additionally, a provisional checkpoint may be undone or altered to be a permanent checkpoint.

Besides that, we presume that the algorithm is invoked by an only one process to take a permanent checkpoint. Our model communicates by messages transfer through lossless (virtually) networks and via *Reliable broadcast*. The algorithm we are modeling here is conceptually based on the *two-phase commit protocol*. During the first step, the checkpoint coordinator *q* takes a preliminary checkpoint and tells all other processes to take preliminary checkpoints.

We create a hypothetical process named *daemon* to assume the initiating and decision-making function of the checkpoint coordinator. When *q* notices that all systems have taken provisional checkpoints, q wants to officially make all provisional checkpoints permanent, else *q* shall opt to dispose of the provisional checkpoints. In the phase two, *q's* decision is broadcasted and all processes complete the *q's* choice. Our latest set of checkpoints is indeed consistent because either every or none of the process take permanent checkpoints This decision about checkpoint is delivered in the same manner like the request is provided to take a provisional checkpoint. The process discards its old checkpoint after taking a new permanent checkpoint (Koo & Toueg, 1987) (Bal Krishna Saraswat, Raghuraj Suryavanshi, & Divakar Yadav, 2021).

4. Informal Specifications of a Total Order Broadcast

Eventually a reliable broadcast (Hadzilacos & Toueg, 1994) sends the messages to all the participating processes. A total order broadcast (Defago et al., 2004) (Hadzilacos & Toueg, 1994) is a tougher notion of a robust broadcast which delivers messages in the same order of delivery to all processes. If the following requirements are satisfied than the reliable broadcast can be termed as Total order broadcast.

"If processes p and q both deliver messages m1 and m2, then q delivers m1 before m2 if and only if p delivers m1 before m2."

We can define the Total Order broadcast by the following properties:

- Validity: If a correct process Total Order broadcasts a message *m*, then it eventually Total Order delivers *m*.
- Uniform Agreement: If a process Total Order delivers a message *m*, then all correct processes eventually Total Order delivers *m*.
- Uniform Integrity: For any message *m*, every process Total Order delivers *m* at most once, and only if *m* was previously Total Order broadcast by *sender(m).*
- Uniform Total Order: If processes *p* and *q* both Total Order delivers messages *m* and *m`*, then *p* Total Order delivers *m* before *m`*, if and only if *q* Total Order delivers *m* before *m`*.

A basic broadcast that satisfies all of these properties is considered a reliable broadcast excluding Uniform Full Order (i.e., that offers no ordering guarantee). The agreement features of a reliable transmission and total order specifications means that all valid processes ultimately deliver the same message sequence (Hadzilacos & Toueg, 1994). As you can see in the Fig. 1, that all the messages have been delivered in the same order at every process. In the Fig. 2 message delivery order violates the total order, as delivery order at process P1, and P2 are different.

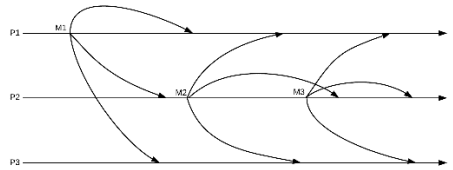

Figure. 1 Total Order

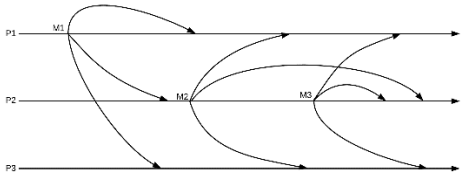

Figure. 2 Total Order Violation

There can be four different roles for the participating process in our model: *sender, destination, daemon* and *sequencer*. The message originates from the *sender* process. The message which originates from the *sender* destined to destination process. The *daemon* is responsible for checkpoint process and *sequencer* is responsible for ordering of messages. The *sequencer* process is responsible for making the *Total Order*. We will be using the *Broadcast Broadcast (BB)* (Defago et al., 2004) variant Fig. 3 of the *Fixed sequencer-based* algorithms.

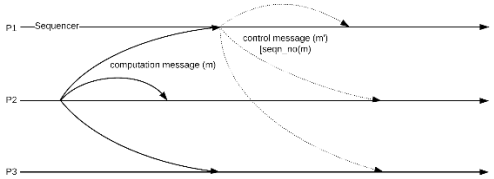

Figure. 3 Total Order - Broadcast Broadcast Variant

4.1 Total Order Implementation Framework

The most important information regarding the total order broadcast algorithm is that who will build the total order, how to build the total order, and what other information is required to define the total order.

In the *Fixed Sequencer based Total Order* algorithms, one process takes the role of *sequencer* and is responsible building of message ordering. At first the *sender* broadcasts a message *m* to all the processes including *sequencer* process. when the *sequencer* process receives the message *m*, it generates a *sequence number* for message *m*. Now *sequencer* broadcasts the sequence number for message *m*. All the *destination* processes deliver the message *m* according the sequence number generated by *sequencer*. As shown in the Fig 3 process *P2* is a *sender* and sends a *computation message m*. As soon as the *computation message m* reaches the *sequencer*, *sequencer* assigns a unique sequence number to *m* and broadcasts its sequence number to all the processes by a *control message m`*. When the *destination* process receives the *control message m`* from sequencer, it delivers their *computation message m* according to the sequence number.

MACHINE Checkpoint_Machine
SEES Checkpoint_Context
VARIABLES
daemon
daemon_status
sender
sent_msg
time_sent_msg
msg_category
deliver
time_response_msg
ckpt_state
no_of_responded_process
tentative_ckpt_no
permanent_ckpt_no
total_order
INVARIANTS
inv11: $tentative\_ckpt\_no \in PROCESS \rightarrow \mathbb{N}$
inv12: $permanent\_ckpt\_no \in \mathbb{N}$
inv1: $daemon \subseteq PROCESS$
inv2: $daemon\_status \in daemon \rightarrow state$
inv4: $sent\_msg \subseteq PROCESS\_MSG$
inv5: $time\_sent\_msg \in PROCESS\_MSG \rightarrow \mathbb{N}$
inv6: $msg\_category \in sent\_msg \rightarrow category$
inv7: $deliver \in PROCESS \leftrightarrow PROCESS\_MSG$
inv8: $time\_response\_msg \subseteq \mathbb{N}$
inv9: $ckpt\_state \in PROCESS \rightarrow checkpointstate$
inv10: $no\_of\_responded\_process \in \mathbb{N}$
inv3: $sender \in PROCESS\_MSG \twoheadrightarrow PROCESS$
inv13: $total\_order \in PROCESS\_MSG \leftrightarrow PROCESS\_MSG$
EVENTS
Initialisation
begin
act1: $ckpt\_state := \varnothing$
act2: $daemon\_status := \varnothing$
act3: $deliver := \varnothing$
act4: $sender := \varnothing$
act5: $sent\_msg := \varnothing$
act6: $time\_sent\_msg := \varnothing$
act7: $msg\_category := \varnothing$
act8: $time\_response\_msg := \varnothing$
act9: $no\_of\_responded\_process := 0$
act10: $tentative\_ckpt\_no := \varnothing$
act11: $permanent\_ckpt\_no := 0$
act12: $daemon := \varnothing$
act13: $total\_order := \varnothing$
end

Figure. 4 *Initial Part - Level 0*

5. Abstract Model of Checkpoint Process Using Total Order Broadcast

We are presenting here the abstract model of the checkpoint process. We incorporated the *total order broadcast* technique for communication between the participating processes, as presented in Fig 4. The dynamic part of the model is represented by *Machine* in *Event-B* (J.-R. Abrial et al., 2010). *Event-B Machine* consists of the *Variables*, *Invariants*, *Guards* and *Action*. More detailed specification about the *Event-B Machine* can be found in (J.-R. Abrial, 2007). Many variables have been defined in the machine of our abstract model, which are listed below, and the detailed definition can be found in Fig. 4:

- ***sender:*** We have defined a variable *sender*, that is a partial function from the set *PROCESS_MSG* to *PROCESS* as defined in Invariant *Inv:3*. A notation of the form $(mm \mapsto pp) \in sender$, denotes that message *mm* was sent by process *pp*.
- ***daemon:*** daemon is checkpoint coordinator process, can be any process from the set *PROCESS*. This is defined in Invariant *Inv:1*.
- ***daemon status:*** Represents the status of the *daemon* (initiator process). Any process can be a *daemon*. It is a *Total function* from *daemon* to set *state*. It is defined in Invariant *Inv: 2*.
  Daemon status can be any one of the four values:

- awaiting
- received_all_responses
- permanent_ckpt_broadcast
- idle

A representation of the form $daemon\_status \in daemon \mapsto awaiting$ represents that *daemon* is waiting to receive the response from checkpoint cohorts and did not received the response from all the cohorts.

- ***sent_msg:*** Represents the messages which was sent by the process.
- ***time_sent_msg:*** Represents the timestamp of the sent messages.
- ***msg_category:*** Represents the category of the sent message. Sent messages can of three types:
  - tentative_ckpt_req
  - tentative_ckpt_response
  - permanent_ckpt_msg

  This is given in Fig.4

$$inv6 : msg\_category \in sent\_msg \rightarrow category$$

A representation of the form $msg\_category \in mm \mapsto tentative\_ckpt\_req$ represents that the category of the message *mm* is tentative checkpoint request.

- ***deliver:*** *deliver* represents the delivery of the message to a process following the *Total Order*. It is a Relation between *PROCESS* and *PROCESS_MSG*. It is defined in Fig.4

$$inv7 : deliver \in PROCESS \leftrightarrow PROCESS\_MSG$$

A representation of the form $\text{deliver} \in \text{pp} \mapsto \text{mm}$ represents that message *mm* has been delivered to process *pp* according to the *Total Order*.

- ***time_response_msg:*** represents the timestamp of all reply messages to the coordinator.
- ***ckpt_state:*** represents the checkpoint state of each process. ckpt_state can be any one of the three values.
  - *open*
  - *tentative*
  - *permanent*

  ckpt_state is a Total function from set *PROCESS* to set *checkpointstate*. It is given in Fig.4. A representation of the form $\text{pp} \mapsto \text{tentative}$ represents that process *pp* has taken *tentative* checkpoint.
- ***no_of_responded_process:*** represents the number of process responded for the request message for checkpoint creation. It is a set of Natural numbers.
- ***tentative_ckpt_no:*** represents tentative checkpoint number of each process. It is a Total function from *PROCESS* to Natural number. it is defined in the Fig.4.

$$inv11: tentative\_ckpt\_no \in PROCESS \rightarrow \mathbb{N}$$

A representation of the form $pp \mapsto 1$ represents that process *pp* has taken a Tentative checkpoint number 1. tentative_ckpt_no store all the events which happens on the process and used for recovery purpose for that process.

- ***permanent_ckpt_no:*** Represents the permanent checkpoint number.
- ***total order:*** The variable total order represents the relationship among the messages. A representation of the form $(m1 \leftrightarrow m2) \in total\_order$ indicates that message *m1* is totally ordered before message *m2*. It is defined in Fig.4.

$$inv13 : total\_order \in PROCESS\_MSG \leftrightarrow PROCESS\_MSG$$

5.1 Events in the Abstract model

5.1.1 Events:

Various events have been defined in the machine. Informal information about the events is given below:

1. **Broadcasting checkpoint request message to the cohorts:**
   In the event *Daemon_checkpoint_request_broadcast* in Fig.5 *daemon* process *pp* broadcast a checkpoint request message *mm* to all the cohorts to take the tentative checkpoint.

Event Daemon_ckpt_request_Broadcast ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \in daemon$
grd2: $mm \in PROCESS\_MSG$
grd3: $daemon\_status(pp) = idle$
grd4: $mm \notin dom(sender)$
then
act1: $tentative\_ckpt\_no(pp) := tentative\_ckpt\_no(pp) + 1$
act2: $time\_sent\_msg(mm) := tentative\_ckpt\_no(pp)$
act3: $sender := sender \cup \{mm \mapsto pp\}$
act4: $daemon\_status(pp) := awaiting$
act5: $sent\_msg := sent\_msg \cup \{mm\}$
act6: $msg\_category(mm) := tentative\_ckpt\_req$
end

Figure. 5 *Broadcast operation of checkpoint process model - Level 0*

The *guard4:* mm ∉ dom(sender) ensures that *daemon* always send a fresh checkpoint request message to all the cohorts. The *guard3:* daemon_status(pp) = idle ensures that the status of the *daemon* should be idle to send a checkpoint request message. If the given guards are true then, timestamp is assigned to the checkpoint request message by incrementing the tentative checkpoint number of daemon process by 1, as given in *action1:* tentative_ckpt_no(pp) ≔ tentative_ckpt_no(pp) + 1 and in *action2:* time_sent_msg(mm) ≔ tentative_ckpt_no(pp) .The status of the *daemon* is set to *awaiting* and category of the request message is set to tentative_ckpt_req.

2. **order:**

The event Order models the construction of an abstract total order on message *mm* its first ever delivery to a process *pp*, as it is given in Fig. 6. The *guard2:* mm ∉ ran(deliver) ensures that the message *mm* has not been delivered to any other process and *guard2:* ran(deliver) ⊆ deliver[{pp}]ensures that any message who has been delivered to any other process must be delivered to this process *pp*.
In the coming refinements of the model, we will get to know that this is the function of a designated process *sequencer* to put each message in the sequence. When all the given guards are true then message *mm* delivered to process *pp* and the variable *total_order* updated by the *action2:* total_order ≔ total_order ∪ (ran(deliver) × {mm}). This action implies that all the messages that are delivered to any process are ordered before *mm*.

Event order ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $mm \in dom(sender)$
grd2: $mm \notin ran(deliver)$
grd3: $ran(deliver) \subseteq deliver[\{pp\}]$
grd4: $pp \in PROCESS$
then
act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $total\_order := total\_order \cup (ran(deliver) \times \{mm\})$
end
END

Figure. 6 *Ordering of message according to the Total Order - Level 0*

3. **Checkpoint Request Message Receive by Checkpoint Cohort:**

The event Cohort Checkpoint Request Receive in Fig. 7 models that the checkpoint cohorts receive the checkpoint request broadcast for the first time and then updates its tentative checkpoint number with timestamp of the request message or its current timestamp value, whichever is higher. In Fig. 7 the *guard4:* $mm \notin deliver[\{pp\}]$ ensures that checkpoint request message *mm* is not delivered to the process *pp*, this is a fresh request message and *guard10:* $\text{mm} \in \text{ran}(\text{deliver})$ ensures that the message *mm* has been delivered to at least one process and the *Total Order* on the message *mm* has also been constructed. *Guard8:* $\text{mm} \mapsto \text{dp} \in \text{sender}$ ensures that sender of the checkpoint request message is *daemon*. *Guard6:* $\text{ckpt\_state}(\text{pp}) = \text{open}$ ensures that state of receiver process of the checkpoint message should be *open*.

Event Cohort_Ckpt_Request_Receive ⟨ordinary⟩ $\widehat{=}$

any

pp
mm
dp

where

grd1: $pp \notin daemon$
grd7: $dp \in daemon$
grd8: $mm \mapsto dp \in sender$
grd9: $pp \mapsto mm \notin deliver$
grd2: $mm \in sent\_msg$
grd3: $msg\_category(mm) = tentative\_ckpt\_req$
grd4: $mm \notin deliver[\{pp\}]$
grd10: $mm \in ran(deliver)$
grd5: $finite(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
grd6: $ckpt\_state(pp) = open$
grd11: $mm \in dom(sender)$
grd12: $\forall m \cdot (m \in PROCESS\_MSG \wedge (m \mapsto mm) \in total\_order \Rightarrow (pp \mapsto m) \in deliver)$

then

act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $tentative\_ckpt\_no(pp) := max(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$

end

Figure. 7 Checkpoint request receive operation of cohorts - Level 0

If all the given guards are true then message *mm* is delivered to process *pp* in *action1:* $deliver := deliver \cup \{pp \mapsto mm\}$.

In *action2:* $\text{tentative\_ckpt\_no}(\text{pp}) := \max(\{\text{time\_sent\_msg}(\text{mm}), \text{tentative\_ckpt\_no}(\text{pp}) + 1\})$ process *pp* takes tentative checkpoint number as maximum value of timestamp of the message *mm* or the tentative checkpoint number of the process incremented by 1.

4. Checkpoint request Response by Cohort:

In the event Checkpoint_Cohort_Response given in Fig. 8 every cohort process sends a timestamped response message to daemon. To assign the timestamp to response message process increments its tentative checkpoint number by 1 and that value is assigned as a timestamp to response message.
In *guard5:* $\text{pp} \mapsto \text{m} \in \text{deliver}$, it is ensured that request message must be delivered to cohort. The response message *mm* must be a fresh response message, it is ensured by *guard3:* $\text{mm} \notin \text{dom}(\text{sender})$. To send the response message to *daemon*, the checkpoint state of the cohort must be *open*.

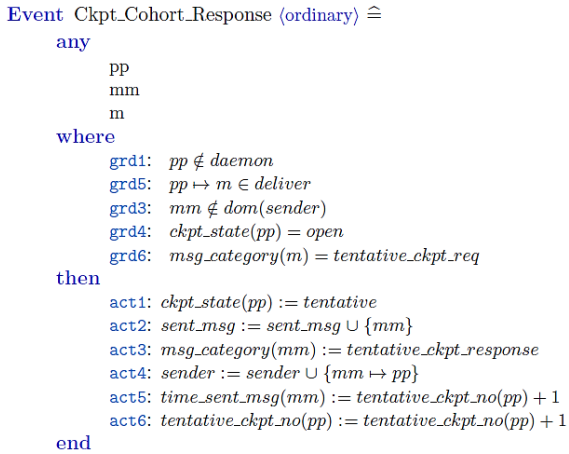

***Figure. 8** Checkpoint response send operation of cohorts*

If all the guards are true then the checkpoint state of the process *pp* is set to *tentative* and message *mm* is added to the set *sent_msg*. Message category of the *mm* is set to *tentative_ckpt_response*, which represents that category is tentative checkpoint response *(action3:)*. Timestamp of the response message *mm* is set to tentative checkpoint number incremented by value 1, it is given in *action3:* (msg_category(mm) ≔ tentative_ckpt_response).

5. Cohort Response Submission at Daemon:

In the event *Cohort_Response_submission_At_Daemon*, response messages from all the cohorts are delivered to the *daemon* process. It is given in Fig. 9. Whenever a daemon process receives a message from a cohort with category *tentative_ckpt_response*, it updates the *no_of_responded_sites* with 1. The value of the timestamp of the response message is also stored for the purpose of Permanent checkpoint computation. *Guard3:* $mm \notin deliver[\{pp\}]$ ensures that the response message *mm* is a fresh response message, and it is not already delivered to process *pp*. Message category of the response message *mm* should be *tentative_ckpt_response*, it is defined in *guard4*: $msg\_category(mm) = tentative\_ckpt\_respons$. Status of the daemon should be *awaiting*.

If all the given guards are true then the response message *mm* should be delivered to the *daemon* in *action1:* deliver ≔ deliver ∪ {pp ↦ mm}. Whenever *daemon* receives the response from any cohort, *daemon* updates the *no_of_responded_process*, it is defined in *action2:* no_of_responded_process ≔ no_of_responded_process + 1 . In *action3:* $(time\_response\_msg \coloneqq time\_response\_msg \cup \{time\_sent\_msg(mm)\})$ timestamp of the response message is added to the pool of all timestamp of the response messages.

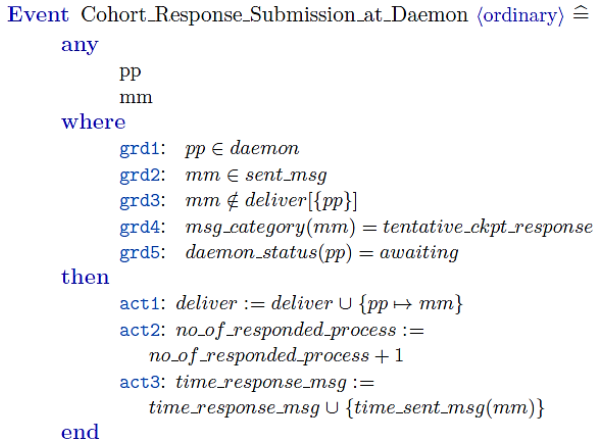

Figure. 9 Cohort response submission at Daemon

6. Permanent Checkpoint Computation:

In the event *permanent_ckpt_computation daemon* must ensure that it has received the response messages from the all the cohorts, it is implied by the *guard2:* $no\_of\_responded\_process = card(PROCESS) - 1$, as it is shown in Fig. 10.

Event permanent_ckpt_computation ⟨ordinary⟩ ≙
any
pp
where
grd1: $pp \in daemon$
grd2: $no\_of\_responded\_process = card(PROCESS) - 1$
grd3: $daemon\_status(pp) = awaiting$
then
act1: $daemon\_status(pp) := received\_all\_responses$
act2: $permanent\_ckpt\_no := max(time\_response\_msg)$
end

Figure. 10 *Permanent Checkpoint Computation by daemon*

daemon must ensure that its status should be *awaiting* before computing the permanent checkpoint number, it is defined in *guard3:* daemon_status(pp) = awaiting.
When all the given guards are true then the *daemon* changes its state from *awaiting* to *received_all_responses*, is defined in *action2:* daemon_status(pp) ≔ received_all_responses, and maximum timestamp value from the received response messages is assigned to the *permanent_ckpt_no*, according to the *action2:* permanent_ckpt_no ≔ max(time_response_msg).

7. Broadcast of Permanent Checkpoint Number

In the event *Broadcast Permanent_Ckpt_No*, permanent checkpoint number is broadcasted by *daemon* to all cohorts. It is given in It is given in Fig.11. *Guard2:* mm ∉ dom(sender) ensures that message *mm* is a fresh permanent checkpoint number message. We put the *guard3:* permanent_ckpt_no = max(time_response_msg) to ensure that the value of permanent checkpoint number should be maximum of the timestamp of received messages. *guard6:* daemon_status(pp) = received_all_responses to ensure that status of the *daemon* should be *received_all_responses.*

When all the given guards are true, message *mm* is added to the *sent_msg* in *action2:* sent_msg ≔ sent_msg ∪ {mm} . Category of the sent permanent checkpoint message *mm* is set to the *permanent_ckpt_msg* in *action3:* msg_category(mm) ≔ permanent_ckpt_msg. The timestamp of the message *mm* is set to the value of *permanent_ckpt_no* in *action4:* time_sent_msg(mm) ≔ permanent_ckpt_no . The status of the *daemon* is set to *permanent_ckpt_broadcast* in *action5:* daemon_status(pp) ≔ permanent_ckpt_broadcast.

Event Broadcast_Permanent_Ckpt_No ⟨ordinary⟩ ≙
any
pp
mm
where
grd1: $pp \in daemon$
grd2: $mm \notin dom(sender)$
grd5: $time\_response\_msg \neq \varnothing$
grd3: $permanent\_ckpt\_no = max(time\_response\_msg)$
grd6: $daemon\_status(pp) = received\_all\_responses$
then
act1: $sender := sender \cup \{mm \mapsto pp\}$
act2: $sent\_msg := sent\_msg \cup \{mm\}$
act3: $msg\_category(mm) := permanent\_ckpt\_msg$
act4: $time\_sent\_msg(mm) := permanent\_ckpt\_no$
act5: $daemon\_status(pp) := permanent\_ckpt\_broadcast$
end

Figure. 11 *Broadcast operation of permanent checkpoint number - Level 0*

8. Permanent Checkpoint Message receive by Cohort:

In the event *Cohort_Permanent_Ckpt_Message_Receive* when cohort process receives the permanent checkpoint number message from the *daemon*, it updates its tentative checkpoint number with the received permanent checkpoint message timestamp, it is given in Fig. 12.
In the *guard4:* $msg\_category(mm) = permanent\_ckpt\_msg$ of the Fig. 12 it is checked that the category of the received permanent checkpoint message *mm* must be *permanent_ckpt_msg*. In the *guard6:* $pp \mapsto mm \notin deliver$ it is ensured that message *mm* should be a fresh permanent checkpoint message, it is not the duplicate message already received by the cohort and *guard8:* $mm \in ran(deliver)$ ensures that the message *mm* has been delivered to at least one process and the *Total Order* on the message *mm* has also been constructed. It is also ensured that the checkpoint state of the cohort must be *tentative*, as implied in the *guard7:* $ckpt\_state(pp) = tentative$.

Event Cohort_permanent_Ckpt_Message_Receive ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \notin daemon$
grd2: $mm \in sent\_msg$
grd4: $msg\_category(mm) = permanent\_ckpt\_msg$
grd5: $mm \in dom(sender)$
grd6: $pp \mapsto mm \notin deliver$
grd7: $ckpt\_state(pp) = tentative$
grd8: $mm \in ran(deliver)$
grd9: $\forall m \cdot (m \in PROCESS\_MSG \wedge (m \mapsto mm) \in total\_order \Rightarrow (pp \mapsto m) \in deliver)$
then
act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $tentative\_ckpt\_no(pp) := time\_sent\_msg(mm)$
end

Figure. 12 *Permanent checkpoint message receive operation by cohort - Level 0*

If all the guards are true then message *mm* is delivered to cohort *pp* and tentative checkpoint number of cohort *pp* is set to the timestamp of the received permanent checkpoint number message *mm*.

9. Switching the Checkpoint State from Tentative to Permanent:
In Fig.13 the event *Switching_from_Tentative_to_Permanent_State*} cohort's checkpointing state is changed from *tentative* to *permanent*. Before switching the checkpoint state from *tentative* to *permanent* it is ensured that received permanent checkpoint message *mm* should have the category *permanent_ckpt_msg* and it is also delivered to the cohort process *pp*, it is defined in *guard3:* $(msg\_category(mm) = permanent\_ckpt\_msg)$ and *guard5:* $(pp \mapsto mm \in deliver)$. The checkpoint state of the cohort process must be *tentative*.

Event Switching_from_Tentative_to_Permanent_state ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \notin daemon$
grd2: $mm \in sent\_msg$
grd3: $msg\_category(mm) = permanent\_ckpt\_msg$
grd4: $ckpt\_state(pp) = tentative$
grd5: $pp \mapsto mm \in deliver$
then
act1: $ckpt\_state(pp) := permanent$
end

Figure. 13 *Switching_from_Tentative_to_Permanent_state*

When the given guards are true then the checkpoint state of the cohort process *pp* is changed to *permanent* to *tentative*.

This is the level-0 of the refinement of abstract model of total order broadcast. In this model, abstract total order is constructed when a message is delivered to a process for the first time. At all other processes a message is delivered in the total order. In the next level, we will introduce the notion of sequencer.

5.2 First Refinement - Introducing the notion of sequencer

In the first refinement of the abstract model, we are going to add the notion of *sequencer*. The Fig. 14 is the refinement of the events given in Fig. 6, Fig. 7 and Fig. 12. The *sequencer* is a constant and defined as sequencer ∈ PROCESS. As we can see in the Fig. 14, the refinement of the abstract model, in the *order* event, a message is first delivered to the *sequencer*. We can also see in event *order* that the guards mm ∉ ran(deliver) and ran(deliver) ⊆ deliver[{pp}] are replaced by the guards pp = sequencer and (sequencer ↦ mm) ∉ deliver. In the events *Checkpoint Request Message Receive by Checkpoint Cohort* and *Permanent Checkpoint Message receive by Cohort* the *guard* pp ≠ sequencer ensures that message *mm* should be delivered to the processes other than *sequencer*.

5.3 Second Refinement - Refinement of order event

It is a rather basic improvement which provides more concrete description to the *order* event definition. Through this refinement we demonstrate that the messages sent to the sequencer can be used to construct a total order. As it is given in Fig. 14, a total order is constructed as, total_order ≔ $total\_order \cup (ran(deliver) \times \{mm\})$. It specifies that all messages which were sent to any process are ordered before the latest message *mm*. The total order is built as total_order ≔ total_order ∪ (deliver[{sequencer}] × {mm}) in the refined *order* event. It specifies that all messages sent to the sequencer are ordered before the latest message *mm*. The refined *order* event is given in Fig. 15.

Event order ⟨ordinary⟩ ≙
  any
    pp
    mm
  where
    grd1: $mm \in dom(sender)$
    grd4: $pp \in PROCESS$
    grd5: $pp = sequencer$
    grd6: $(sequencer \mapsto mm) \notin deliver$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $total\_order := total\_order \cup (ran(deliver) \times \{mm\})$
  end

Event Cohort_Ckpt_Request_Receive ⟨ordinary⟩ ≙
  any
    pp
    mm
    dp
  where
    grd1: $pp \notin daemon$
    grd7: $dp \in daemon$
    grd8: $mm \mapsto dp \in sender$
    grd9: $pp \mapsto mm \notin deliver$
    grd2: $mm \in sent\_msg$
    grd3: $msg\_category(mm) = tentative\_ckpt\_req$
    grd4: $mm \notin deliver[\{pp\}]$
    grd10: $mm \in ran(deliver)$
    grd5: $finite(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
    grd6: $ckpt\_state(pp) = open$
    grd11: $mm \in dom(sender)$
    grd12: $\forall m \cdot (m \in PROCESS\_MSG \wedge (m \mapsto mm) \in total\_order \Rightarrow (pp \mapsto m) \in deliver)$
    grd13: $pp \neq sequencer$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $tentative\_ckpt\_no(pp) := max(\{time\_sent\_msg(mm tentative\_ckpt\_no(pp) + 1\})$
  end

Event Cohort_permanent_Ckpt_Message_Receive ⟨ordinary⟩ ≙
  any
    pp
    mm
  where
    grd1: $pp \notin daemon$
    grd2: $mm \in sent\_msg$
    grd4: $msg\_category(mm) = permanent\_ckpt\_msg$
    grd5: $mm \in dom(sender)$
    grd6: $pp \mapsto mm \notin deliver$
    grd7: $ckpt\_state(pp) = tentative$
    grd8: $mm \in ran(deliver)$
    grd10: $pp \neq sequencer$
    grd9: $\forall m \cdot (m \in PROCESS\_MSG \wedge (m \mapsto mm) \in total\_order \Rightarrow (pp \mapsto m) \in deliver)$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $tentative\_ckpt\_no(pp) := time\_sent\_msg(mm)$
  end

Figure. 14 *Checkpoint Process using Total Order Broadcast: Level-1*

Event order ⟨ordinary⟩ $\widehat{=}$
  any
    pp
    mm
  where
    grd1: $mm \in dom(sender)$
    grd4: $pp \in PROCESS$
    grd5: $pp = sequencer$
    grd6: $(sequencer \mapsto mm) \notin deliver$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $total\_order := total\_order \cup (deliver[\{sequencer\}] \times \{mm\})$
  end

Figure. 15 *Checkpoint Process using Total Order Broadcast: Level-2*

5.4 Third Refinement - Introducing Sequence Numbers

In the third refinement we introduce the concept of sequence numbers. The new variables *counter* and *seqn_no* is introduced in the third refinement. The variable *counter* is defined as counter ∈ ℕ and variable *seqn_no* is defined as seqn_no ∈ sent_msg⇸ℕ. The variable *seqn_no* is used to assign the sequence numbers to the sent messages.

Event order ⟨ordinary⟩ $\widehat{=}$
  any
    pp
    mm
  where
    grd1: $mm \in dom(sender)$
    grd4: $pp \in PROCESS$
    grd5: $pp = sequencer$
    grd6: $(sequencer \mapsto mm) \notin deliver$
    grd7: $mm \in sent\_msg$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $total\_order := total\_order \cup (deliver[\{sequencer\}] \times \{mm\})$
    act3: $seqn\_no := seqn\_no \cup \{mm \mapsto counter\}$
    act4: $counter := counter + 1$
  end

Event Cohort_Ckpt_Request_Receive ⟨ordinary⟩ $\widehat{=}$
  any
    pp
    mm
    dp
  where
    grd1: $pp \notin daemon$
    grd7: $dp \in daemon$
    grd8: $mm \mapsto dp \in sender$
    grd9: $pp \mapsto mm \notin deliver$
    grd2: $mm \in sent\_msg$
    grd3: $msg\_category(mm) = tentative\_ckpt\_req$
    grd4: $mm \notin deliver[\{pp\}]$
    grd10: $mm \in ran(deliver)$
    grd5: $finite(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
    grd6: $ckpt\_state(pp) = open$
    grd11: $mm \in dom(sender)$
    grd12: $\forall m \cdot (m \in sent\_msg \wedge (seqn\_no(m) < seqn\_no(mm)) \Rightarrow (pp \mapsto m) \in deliver)$
    grd13: $pp \neq sequencer$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $tentative\_ckpt\_no(pp) := max(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
  end

Event Cohort_permanent_Ckpt_Message_Receive ⟨ordinary⟩ $\widehat{=}$
  any
    pp
    mm
  where
    grd1: $pp \notin daemon$
    grd2: $mm \in sent\_msg$
    grd4: $msg\_category(mm) = permanent\_ckpt\_msg$
    grd5: $mm \in dom(sender)$
    grd6: $pp \mapsto mm \notin deliver$
    grd7: $ckpt\_state(pp) = tentative$
    grd8: $mm \in ran(deliver)$
    grd10: $pp \neq sequencer$
    grd9: $\forall m \cdot (m \in sent\_msg \wedge (seqn\_no(m) < seqn\_no(mm)) \Rightarrow (pp \mapsto m) \in deliver)$
  then
    act1: $deliver := deliver \cup \{pp \mapsto mm\}$
    act2: $tentative\_ckpt\_no(pp) := time\_sent\_msg(mm)$
  end

Figure. 16 *Checkpoint Process using Total Order Broadcast: Level-3*

The *counter*, initialized with zero, is managed by the sequencer process, and incremented by one each time the sequencer process sends out a control message. Throughout the *Checkpoint Request Message Receive by Checkpoint Cohort* and *Permanent Checkpoint Message receive by Cohort* event definition it can be noticed that such messages are delivered in their sequence numbers to processes other than the sequencer.
It can be noticed that guard in the abstract event *Checkpoint Request Message Receive by Checkpoint Cohort* and *Permanent Checkpoint Message receive by Cohort*,

$$\forall m.\,(m \in \text{PROCESS\_MSG} \wedge (m \mapsto mm) \in \text{total\_order} \Rightarrow (pp \mapsto m) \in \text{deliver})$$

is replaced by the *guard*

$$\forall m.\,\big(m \in \text{sent\_msg} \wedge \big(\text{seqn\_no}(m) < \text{seqn\_no}(mm)\big) \Rightarrow (pp \mapsto m) \in \text{deliver}\big).$$

6. Conclusion

In this paper we proposed formal design of a broadcast system in total order. In the abstract model we detail how to create a conceptual total order on the messages. Subsequently in a set of optimization steps we explain how the notion of control messages and sequence numbers can be used to properly enforce an abstract total order. Instead of model checking, proving theorems by hand, or proving trace behaviour correctness, our strategy is to define problem in the abstract model and introduce approaches or design features in the refining phases. We verify that the variables in the refinement are true refinements of abstract variables by refinement checking. We used the Rodin tool for proof management. This tool generates proof obligations as a result of refinement and consistency tests, accounts for complex proof obligations for relatively simple proofs, and assists in the fulfilment of proof obligations through automated and interactive provers.

References

1. Abrial, J. (1996). The b-book: assigning programs to meanings cambridge university press. London.
2. Abrial, J.-R. (2007). A system development process with event-b and the rodin platform. In International conference on formal engineering methods (pp. 1-3).
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in event-b. International journal on software tools for technology transfer, 12 (6), 447-466.
4. Abrial, J.-R., Butler, M., Hallerstede, S., & Voisin, L. (2006). An open extensible tool environment for event-b. In International conference on formal engineering methods (pp. 588-605).
5. Ahmed, W., & Wu, Y. W. (2013). A survey on reliability in distributed systems. Journal of Computer and System Sciences, 79 (8), 1243-1255.
6. Back, R.-J. (1989). Re_nement calculus, part ii: Parallel and reactive programs. In Work-shop/school/symposium of the rex project (research and education in concurrent systems) (pp.67-93).
7. Baldoni, R., Mostefaoui, A., & Raynal, M. (1996). Causal delivery of messages with real-time data in unreliable networks. Real-Time Systems, 10 (3), 245-262.
8. Bal Krishna Saraswat, Raghuraj Suryavanshi, Divakar Yadav (2021). Formal Specification & Verification of Checkpoint Algorithm for Distributed Systems using Event - B International Journal of Engineering Trends and Technology, *69(4),1-9.*
9. Bal Krishna Saraswat, D. Y., Raghuraj Suryavanshi. (2018). A comparative study of checkpointing algorithms for distributed systems. International Journal of Pure and Applied Mathematics, 118, 1595-1603.
10. Boulanger, J.-L. (2014). Formal methods applied to complex systems: implementation of the b method. John Wiley & Sons.
11. Cristian, F., Aghili, H., Strong, R., & Dolev, D. (1995). Atomic broadcast: From simple message diffusion to byzantine agreement. Information and Computation, 118 (1), 158-179.
12. Defago, X., Schiper, A., & Urban, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR), 36 (4), 372-421.
13. Hadzilacos, V., & Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems (Tech. Rep.). Cornell University.
14. Holliday, J. (2001). Replicated database recovery using multicast communication. In Proceedings ieee international symposium on network computing and applications. nca 2001 (pp. 104-107).
15. Kemme, B., Pedone, F., Alonso, G., Schiper, A., & Wiesmann, M. (2003). Using optimistic atomic broadcast in transaction processing systems. IEEE Transactions on Knowledge and Data Engineering, 15 (4), 1018-1032.
16. Kim, J. L., & Park, T. (1993). An e_cient protocol for checkpointing recovery in distributed systems. IEEE Transactions on Parallel and Distributed Systems, 4 (8), 955-960.

17. Koo, R., & Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. IEEE Transactions on software Engineering(1), 23-31.
18. Lamport, L. (1978). The implementation of reliable distributed multiprocess systems. Computer Networks (1976), 2 (2), 95-114.
19. Leu, P.-J., & Bhargava, B. (1988). Concurrent robust checkpointing and recovery in distributed systems. In Proceedings. fourth international conference on data engineering (pp. 154-163).
20. Luan, S.-W., & Gligor, V. D. (1990). A fault-tolerant protocol for atomic broadcast. IEEE Transactions on Parallel & Distributed Systems(3), 271-285.
21. Manivannan, D., Netzer, R. H. B., & Singhal, M. (1997). Finding consistent global checkpoints in a distributed computation. IEEE Transactions on Parallel and Distributed Systems, 8 (6), 623-627.
22. Pedone, F., Guerraoui, R., & Schiper, A. (1998). Exploiting atomic broadcast in replicated databases. In European conference on parallel processing (pp. 513-520).
23. Pedone, F., & Schiper, A. (1999). Generic broadcast. In International symposium on distributed computing (pp. 94-106).
24. Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR), 22 (4), 299-319.