Formal Model of Crash Tolerant Moving Sequencer Atomic Broadcast (UUB based)

Prateek Srivastava[1]

[1]*Graphic Era Hill University, Clement Town, Dehradun, Uttarakhand, India*

Abstract

The proposed work investigates crash failure in moving sequencer atomic broadcast that relies on unicast unicast broadcast (UUB) variant of fixed. A few points are evident from various existing kinds of literature, like; (i) Different existing moving sequencer atomic broadcast algorithms (like Reliable multicast protocol, Dynamic token-passing scheme, and Pinwheel) are based on broadcast broadcast (BB) variant. The BB-based mechanism always introduces more messages in comparison to UUB. In a distributed environment, it is also possible that any process (or processes) might be crashed, hence the computing environment should be verified so that such types of failures can be handled. This work is an attempt to extend the mechanism given for atomic broadcast and to make it capable to tolerate crash failure. The "B" formal language is popular for the development of distributed models. The ProB tool is considered here for checking and verification of models. The proposed model investigates crash tolerance in UUB based atomic broadcast environment. The result shows all the processes have delivered messages in the correct order, even in the case of the crashes of some processes.

Keywords

Verification, crash, unicast unicast broadcast (UUB)

1. Introduction

The reliable broadcast ensures that all the correct processes in the system should deliver all the messages that were broadcasted earlier by any process [1]. However, it never restricts the order of delivery of messages hence atomic broadcast is required that ensures the order of delivery also. The UUB (Unicast Unicast Broadcast) variant of fixed sequencer atomic broadcast consists of three steps; At first, a sender process unicasts request, asking for a sequence number from the sequencer for the message (Msg1). In the second step, the sequencer unicasts a unique number for the message (seqno(Msg1)) to the sender and this unique number is known as the sequence number. In the third step, the sender broadcasts the computation message (Msg1) along with its sequence number (seqno(Msg1)). This work considers an asynchronous distributed computing environment for investigation.

2. Related work

The various categories of atomic broadcast algorithms are discussed in depth in [3]. Here, a detailed discussion of almost all such algorithms and their comparison is given. This paper also focuses on various types of failures that may happen in a distributed computing environment. The work given in [4] presents that Amoeba group communication protocol works on two points, (i) it works with a sequencer-based protocol with negative acknowledgment numbers, and (ii) the user can

ACI'22: Workshop on Advances in Computation Intelligence, its Concepts & Applications at ISIC 2022, May 17-19, Savannah, United States

EMAIL: psrivastava@gehu.ac.in (A. 1)

choose the degree of fault tolerance as per their choice. The protocol proposed in [5]; describes a reliable multicast protocol that utilizes the multicast capability of applicable lower-level network architecture. The Tandem [6] provides a highly available, general-purpose, and fault-tolerant computer. The work given in [7] enhances multicast communication by propagation graph algorithm that ensures strong ordering properties in multiple groups. The research proposed in [8] ensures total order in multiple groups using propagation trees. The research given in [9] presents a new set of algorithmss that works in support of the group communication approach. This work focuses on two primitives; (i) a fault-tolerant causally ordered delivery system and (ii) it is extended to totally ordered multicast primitive. The work given in [10] gives an implementation of a reliable group communication mechanism that ensures atomicity in message delivery by all or no correct processes of the group. The [11] investigates different definitions of total order and proposes a hierarchy of them. Here Weak and Strong total order definitions are introduced which are extreme of the proposed hierarchy. The Rampart [12] investigates new failures and improves system performance. It is a group communication protocol that suggests trusting a set of processes is better than trusting a single process in the distributed system. The papers discussed so far are based on a fixed sequencer process hence they suffer from a single point of failure. However, Reliable multicast protocol [13], Dynamic token passing [14], and Pinwheel [15] are based on moving sequencers hence more reliable. The research done in [2] presents a model which delivers messages in total order to either all or none of the processes. But this model doesn't handle any failure.

3. Objectives

This research provides a refinement of work given in [2] and improves the earlier proposal with crash tolerant capabilities.

4. Model refinement

4.1. Investigation of crash tolerance

In a computing environment, a process can be crashed anytime and remain halted. For example, it stops sending or receiving messages. The processes in a crash state are known as faulty processes otherwise correct processes. The correct processes behave correctly all the time but the crashed processes can neither send nor receive messages. Hence when a crashed process recovers then it is mandatory that it must deliver all those messages that were broadcasted during its crash state so that total order (atomic broadcast) in the system should be maintained. So, for this purpose many new invariants have been added, a few new events are also added, and earlier events (proposed in [2]) have been strengthened with new capabilities.

4.2. New invariants

To achieve crash tolerance, the following new specifications (invariants) have been introduced in the proposed refined model.

List_of_Crashed_Processes∈ POW(Process) /* It represents the list of crashed processes. */

List_of_Alive_Processes ∈ POW(Process) /* It represents list of alive processes. */

List_of_Crashed_Processes /\ List_of_Alive_Processes={} /* If a process is confirmed as crashed then it will not belong to the alive list and vice versa. */

my_trusted_seq ∈ POW(Process) /* The sequencer is determined as trusted once it will be checked for being alive by all other processes positively. */

List_of_Crashed_Receivers∈ POW(Process) /* It represents a list of destination processes that are crashed and hence can't receive the messages. */

List_of_Correct_Receivers∈ POW(Process) /* It represents a list of destination processes that are alive and hence ready to receive the messages. */

List_of_Correct_Receivers /\ List_of_Crashed_Receivers={} /* If a receiver process is correct then it will not belong to the list of crashed receivers list and vice versa. */

msg_delivered∈ Process ↔ Message /* This shows a list of messages delivered to receiving process. */

Seq_Checks_All_Processes_Alive_OR_Crashed∈ Process ↔ Process /* This list is maintained by the sequencer process that keeps those processes whose state is checked by sequencer. */

Ack_After_Recovering_From_Crash∈ POW(Process) /* This list contains those processes which are recovered from the crash state. */

check_heartbeat_seq∈ Process ⇸ (Process ↔ BOOL) /* All the processes will check the state of the sequencer process and this entry will be kept here. */

total_sequencer_votes∈ INTEGER /* It represents the total number of votes casted for sequencer by other processes. */

Seq_+_votes∈ INTEGER /* It represents the total positive votes casted for the sequencer process. If a process finds sequencer is alive then this variable will be increased by one. */

Seq_-_votes∈ INTEGER /* It represents the total negative votes casted for the sequencer process. If a process finds sequencer is crashed then this variable will be increased by one. */

Start_Unicast ∈ BOOL /* If a process became trusted then it will TRUE which informs that now processes can unicast their message to sequencer. Initially it is FALSE. */

Seq_is_ON_or_OFF_Check_is_Done∈BOOL /* If this variable is TRUE then indicates that all the processes have checked the state of the sequencer process. Initially it is FALSE. */

last_round_of_voting_4_processes∈ POW(Process) /* Once half of the processes will cast their votes for the sequencer then the sequencer will become part of this list and once the sequencer will be added to this list then it will not be allowed to change its state. */

4.3. Empowering sequencer selection event

The following new guard and action have been added to the sequencer selection event [2] to strengthen it.
/*Guard */ p /: List_of_Crashed_Processes.
/*Action */ Seq_is_ON_or_OFF_Check_is_Done := FALSE.

4.4. Check sequencer's heartbeat event

As the sequencer is elected, other processes start to check the sequencer's heartbeat and voting will happen to check the sequencer's correctness. This event helps processes to cast their vote.
Check_sequencer_heartbeat (sender, sequencer_process) = /*Guard*/ sender: Process
/*Guard */ sender/: selected_seq

/*Guard*/ sender/: dom(check_heartbeat_seq)
/*Guard */ sender/: List_of_Crashed_Processes /*Guard */ sequencer_process: Process
/*Guard */ sequencer_process: selected_seq
/*Guard */ (sequencer_process: List_of_Alive_Processes or sequencer_process: List_of_Crashed_Processes)
/*Guard */ Seq_is_ON_or_OFF_Check_is_Done= FALSE
/*Guard */ card(unicast_message)/= card(Message)
/*Guard */ my_trusted_seq = {}THEN
/*Action */ total_sequencer_votes:= total_sequencer_votes +1
/*Condition */ IF sequencer_process:List_of_Alive_Processes THEN
/*Action */ check_heartbeat_seq(sender):= {sequencer_process↦TRUE}
/*Action */ Seq_+_votes:= Seq_+_votes+1 END
/* Condition */ IF sequencer_process: List_of_Crashed_Processes THEN
/*Action */ check_heartbeat_seq(sender):= {sequencer_process↦FALSE} /*Action */ Seq_-_votes:= Seq_-_votes+1
/* Condition */ IF total_sequencer_votes= card(List_of_Alive_Processes)/2 THEN
/*Action */ last_round_of_voting_4_processes:= {sequencer_process}
END END

4.5. Vote for sequencer event

As all the processes have been checked heartbeat of the sequencer process and casted their votes then this event comes into existence. It enables to compare the total positive and negative votes casted for the sequencer process in order to ensure the sequencer is correct or faulty. The *Vote for sequencer event* is given as follows:

Vote_for_Seq (p) =/*Guard */ p: Process
/*Guard */ p: selected_seq
/*Guard */ p/: my_trusted_seq
/*Guard */ (p: List_of_Alive_Processes or p: List_of_Crashed_Processes)
/*Guard */ total_sequencer_votes= card(List_of_Alive_Processes) -1
/*Guard */ card(my_trusted_seq)= 0 THEN
/*Condition */ IF Seq_-_votes>= Seq_+_votes THEN
/*Action */ sequencer_selection:= FALSE
/*Action */ check_heartbeat_seq:= {}
/*Action */ Seq_is_ON_or_OFF_Check_is_Done:= TRUE ELSE
/*Action */ Unicast_Start:= TRUE
/*Action */ my_trusted_seq:= {p}
/*Action */ List_of_Correct_Receivers:= List_of_Correct_Receivers \/ {p} END
/*Action */ Seq_-_votes := 0
/*Action */ Seq_+_votes:= 0
/*Action */ total_sequencer_votes:= 0
/*Action */ last_round_of_voting_4_processes:= {} END

4.6. Strengthening unicast event

The unicast event [2] has been strengthened by the addition of some more guards like,

/*Guard */ p/:List_of_Crashed_Processes
/*Guard */ Unicast_Start=TRUE.
/*Guard */ card(my_trusted_seq)/=0.
/*Guard */ m/:ran(msg_delivered).
/*Condition */ IF card(unicast_message)= card(Message)-1 THEN
/*Action*/ Unicast_Start:=FALSE

4.7. Strengthening re unicast event

The re-unicast event [2] is strengthened with inclusion of following guard.
/*Guard*/ sender /: List_of_Crashed_Processes.

4.8. Strengthening broadcast event

There are some more guards have been introduced in order to empower it, like;

/*Guard */ p: my_trusted_seq
card(List_of_Correct_Receivers) + card(List_of_Crashed_Receivers) = card(Process).
The condition 1 (IF card(acknowledged_msg)-1=0) given in abstract model has been updated with some more actions like,
/*Action */ my_trusted_seq:={},
/*Action */ Seq_is_ON_or_OFF_Check_is_Done:=TRUE ,
/*Action */ check_heartbeat_seq:={}
/*Action */ unicast_message:={}

4.9. Strengthening deliver event

In this refined model deliver event [2] has been assigned with some more capabilities (by strengthening) with help of new guards and some new conditions like,
/*Guard */ p/:List_of_Crashed_Receivers
/*Action */ msg_delivered:=msg_delivered ∨ {p↦m}
/*Condition */ IF card(msg_delivered[{p}])+1= card(msg_sent) & card(acknowledged_msg)=0 THEN
/*Action */ List_of_Correct_Receivers:=List_of_Correct_Receivers-{p}
/*Action*/ Ack_After_Recovering_From_Crash:=Ack_After_Recovering_From_Crash-{p} END
/*Condition*/
IF card(msg_delivered~[{m}])+1=card(List_of_Correct_Receivers) THEN /*Action */ temporary_receive:= temporary_receive |->> {m} END

4.10. Check heartbeat of receiver processes

At first, the trusted sequencer will evaluate the state of all the receivers and build a list accordingly to show correct and faulty separately.
Seq_Checks_All_Processes_Alive_OR_Crashed(pr, Any_Process) =/*Guard */ p: Process
/*Guard */ pr: selected_seq
/*Guard */ pr: my_trusted_seq
/*Guard */ Any_Process: Process
/*Guard */ Any_Process/: selected_seq
/*Guard */ (Any_Process /:List_of_Crashed_Receivers or Any_Process/: List_of_Correct_Receivers)
/*Guard */ Any_Process/: Ack_After_Recovering_From_Crash
/*Guard */ card(acknowledged_msg)/= 0 THEN
/*Condition */ IF Any_Process: List_of_Crashed_Processes&Any_Process: dom(unicast_message) THEN
/*Action */ unicast_message:= {Any_Process}<<-| unicast_message
/*Action*/ acknowledged_msg:=
acknowledged_msg|->>unicast_message[{Any_Process}]
/*Action */ Unicast_Start:= TRUE END
/*Condition */ IF Any_Process: List_of_Crashed_Processes THEN
/*Action */ List_of_Crashed_Receivers:= List_of_Crashed_Receivers ∨ {Any_Process} END
/*Condition */ IF Any_Process/: List_of_Crashed_Processes THEN

/*Action */ List_of_Correct_Receivers:= List_of_Correct_Receivers \/ {Any_Process}
/*Action */ Seq_Checks_All_Processes_Alive_OR_Crashed:= Seq_Checks_All_Processes_Alive_OR_Crashed\/{pr↦Any_Process}
END END

4.11. Crash investigation event

The crash_ *Investigation* event is introduced to demonstrate the crash tolerance feature of the system.

Crash_Investigation (pr) = pr: Process pr/: my_trusted_seq
/*Guard */ pr/: List_of_Crashed_Processes
/*Guard */ pr/: List_of_Correct_Receivers
/*Guard */ pr/: Ack_After_Recovering_From_Crash
/*Guard */ pr/: last_round_of_voting_4_processesTHEN
/*Action */ List_of_Crashed_Processes:= List_of_Crashed_Processes\/{pr}
/*Action*/ List_of_Alive_Processes:= List_of_Alive_Processes-{pr}
/*Condition */ IF pr/: selected_seq & pr: dom(check_heartbeat_seq) &ran(check_heartbeat_seq(pr))= {TRUE}&total_sequencer_votes> 0
THEN
/*Action */ total_sequencer_votes:= total_sequencer_votes-1
/*Action 2*/ Seq_+_votes:= Seq_+_votes-1
/*Action */ check_heartbeat_seq:={pr}<<-| check_heartbeat_seq END
/*Condition */ IF pr/: selected_seq& pr: dom(check_heartbeat_seq) &ran(check_heartbeat_seq(pr))={FALSE} & total_sequencer_votes> 0
THEN
/*Action */ Seq_-_votes:= Seq_-_votes + 1
/*Action */ total_sequencer_votes:= total_sequencer_votes-1
/*Action */ check_heartbeat_seq:= {pr}<<-|check_heartbeat_seq END
/*Condition */ IF pr: selected_seq THEN
/*Action */ acknowledged_msg:= {}
/*Action */ unicast_message:= {}END END

4.12. Recover_crashed_processes event

This event runs itself and recovers crashed processes.

Recover_Crashed_Processes(Pr)=pr: Process
/*Guard */ pr: List_of_Crashed_Processes
/*Guard */ pr/: List_of_Alive_Processes
/*Guard */ pr/: last_round_of_voting_4_processes THEN
/*Action */ List_of_Crashed_Processes:= List_of_Crashed_Processes-{pr}
/*Action */ List_of_Alive_Processes:= List_of_Alive_Processes\/{pr}
/*Action */ List_of_Crashed_Receivers:= List_of_Crashed_Receivers-{pr}
/*Condition */ IF card(selected_seq)/= 0 & card(msg_sent)/= 0 THEN
/*Action */ List_of_Correct_Receivers:= List_of_Correct_Receivers \/ {pr}
/*Action*/Ack_After_Recovering_From_Crash:= Ack_After_Recovering_From_Crash\/{pr} END
END

5. Results and discussion

To animate this model for 500 animations, the ProB [16] has been applied here. In such animations no deadlock or invariant violation has been observed for any transition. The current state of different variables has been shown in Table 1. The Table 1 entails that finally, all the processes have delivered messages in the same sequence (values of receive variable) even in case of failures also. Since the

order of receiving of the message at each process (Finally_Msg_delivered variable) is the same hence confirming the definition of atomic broadcast.

Table 1

Result

Variables	Values
selected_seq	{p1}
unicast_message	{(p2↦m2)}
Message_order	{(2↦1),(3↦1),(3↦2)}
msg_sent	{({(p2↦m1)}↦2),({(p2↦m3)}↦3),({(p2↦m4)}↦1)}
Sequence_number	4
Finally_Msg_delivered	{({(p1↦m1)}↦2),({(p1↦m3)}↦3),({(p1↦m4)}↦1),({(p2↦m1)}↦2),({(p2↦m3)}↦3),({(p2↦m4)}↦1),({(p3↦m1)}↦2),({(p3↦m3)}↦3),({(p3↦m4)}↦1),({(p4↦m1)}↦2),({(p4↦m3)}↦3),({(p4↦m4)}↦1)}
Broadcasted_message_with_seq	{(m1↦2),(m3↦3),(m4↦1)}
acknowledged_msg	{(p1↦m2)}
List_of_Crashed_Processes	{}
List_of_Alive_Processes	{p1,p2,p3,p4}
my_trusted_seq	{p1}
check_heartbeat_seq	{(p3↦{(p1↦TRUE)}),(p4↦{(p1↦TRUE)})}
unicast_Start	TRUE
List_of_Crashed_Receivers	{}
List_of_Correct_Receivers	{p2,p3,p4}
Seq_Checks_All_Processes_Alive_OR_Crashed	{(p2↦p3),(p2↦p4)}
msg_delivered	{(p1↦m1),(p1↦m3),(p1↦m4),(p2↦m1),(p2↦m3),(p2↦m4),(p3↦m1),(p3↦m3),(p3↦m4),(p4↦m1),(p4↦m3),(p4↦m4)}
Ack_After_Recovering_From_Crash	{p2,p3,p4}

6. Conclusion and future scope

The proposed model is an extension of [2] and provides the capability of crash tolerance to this model. The results show that the properties of atomic broadcast are maintained and no case of invariant violation, deadlock, or inconsistent view has been reported. The model is verified on ProB [16] tool and codes are written with the help of the B formal language [17].

The proposed model investigates only crash failure in moving sequencer atomic broadcast but is unable to tolerate omission and byzantine failures. A system can't be fully reliable unless it handles all types of failures. So, there is a scope for further investigation of omission and byzantine failures so that a system can be more reliable.

7. References

[1] Hadzilacos V. and Toueg S.: A modular approach to fault-tolerant broadcasts and related Problems, TR 94 -1425, Cornell University, New York (1994).

[2] Srivastava P. and Sharma A. : Rigorous design of moving sequencer atomic broadcast in distributed systems, ACM-ICTCS, Udaipur, Rajasthan, India (2016).

[3] D´efago X., Schiper, A., and Urb´an, P. : Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36, 372– 421 (2004).

[4] Kaashoek, M. F. and Tanenbaum, A. S. :An evaluation of the Amoeba group communication system. In Proceeding of 16th International Conference on Distributed Computing Systems. pp. 436–447. Hong Kong, (1996).
[5] Armstrong, S., Freier, A., and Marzullo, K. : Multicast transport protocol. RFC 1301, IETF (2004).
[6] Carr, R. :The Tandem global update protocol. Tandem Systems Review. 1, 74–85 (1985).
[7] Molina, G. H., and Spauster, A.: Ordered and reliable multicast communication. ACM Trans. Comput. Syst. 9, 242-271 (1991).
[8] Jia, X. : A total ordering multicast protocol using propagation trees. IEEE Trans. Parall. Distrib. Syst. 6, 617–627 (1995).
[9] Birman, K. P., Schiper, A., and Stephenson, P. : Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. 9, 272–314 (1991).
[10] Navaratnam, S., Chanson, S. T. and Neufeld, G. W.: Reliable group communication in distributed systems. In Proceeding of 8th International Conference on Distributed Computing Systems. pp. 439–446. San Jose, CA, USA, (1988).
[11] Wilhelm, U. and Schiper, A. : A hierarchy of totally ordered multicasts. In Proceeding of 14th Symp. on Reliable Distributed Systems. IEEE, pp. 106–115. Bad Neuenahr, Germany, (1995).
[12] Reiter, M. K.: Distributing trust with the Rampart toolkit. Commun. ACM. 39, 71–74 (1996).
[13] Jia, W., Kaiser, J., and Nett, E. 1996. RMP: Fault–Tolerant Group Communication. Micro, IEEE. 16, 59 – 67 (1996).
[14] Kim, J., and Kim, C. : A total ordering protocol using a dynamic token-passing scheme. Distrib. Syst. Eng. 4, 87–95 (1997).
[15] Cristian, F., Mishra, S., and Alvarez, G.: High-performance asynchronous atomic broadcast. Distributed System Engineering Journal. 4, 109–128 (1997).
[16] Leuschel, M., and Butler, M.: Pro B: A model checker for B. In FME, K. Araki, S., Gnesi, and D., Mandrioli, Eds. LNCS. Springer, Heidelberg, 855-874 (2003).
[17] Abrial, J., R.: The B-book: assigning programs to meanings. Cambridge University Press New York. USA, ISBN:0-521-49619-5 (1996).