Rigorous Design of Moving Sequencer Omission Tolerant Atomic Broadcast

Prateek Srivastava
Department of Computer Science and Engineering
Sir Padampat Singhania University
Udaipur, Rajasthan, India

Prasun Chakrabarti
Department of Computer Science and Engineering
Sir Padampat Singhania University
Udaipur, Rajasthan, India

Avinash Panwar
Department of Computer Science and Engineering
Sir Padampat Singhania University
Udaipur, Rajasthan, India

ABSTRACT
This article investigates a mechanism to tolerate omission failure in moving sequencer based atomic broadcast at distributed systems. Various mechanisms are already given for moving sequencer based atomic broadcast like RMP [1], DTP [2], Pin Wheel [3] and mechanism proposed by [4]. But none of these mechanisms are efficient to tolerate different failure. Scholarly observation is that, these algorithms can tolerate only crash failure but not capable to tolerate omission or byzantine failure. This work is an extension of [4]. This work proposes a mechanism to tolerate omission failure in moving sequencer based atomic broadcast. Hence this work is a refined version of [4]. This work relies on unicast broadcast hence it will introduce a very less number of messages in comparison to previous mechanisms [5].
B [6] has been used as formal technique for development of proposed model. B uses set theory as a modeling notation, refinements to represent system at different abstraction level. Pro B [7] has been used as model checker and animator for constraint based checking, to discover errors due to invariant violation and for deadlocks, thereby, validating the specifications.

General Terms
Distributed Systems, Model Verification

Keywords
Broadcast, Atomic Broadcast, Total Order, Unicast, Sequencer, Crash, Omission, Model Checking, B formal method.

1. INTRODUCTION
Atomic broadcast (also known as total order broadcast) is an important abstraction in fault tolerant distributed computing [8]. It ensures that messages broadcasted by different processes are delivered by all destination processes in same order [9]. Lamport has proposed state machine replication [10] for implementing fault tolerant services. Basically state machine replication is way to achieve highly available system. These systems are available in any case whether very high load or any failure. So the question arises that what is the role of atomic broadcast in context to highly available systems. To answer this question one has to understand the functioning of state machine replication. A state machine is set of state variable which implements its state and commands, which transform its state [11]. The client interacts with replicated servers by submitting same order of input commands. The replicas are in same initial state, after receiving input they will go through same state of execution and generate same result and finally go to same final state. The voting will be there for correctness of result and then correct result will be given back to client. In Distributed environment it is very difficult to achieve same order (or sequence) on input commands due to lackness of global clock in distributed systems. To achieve this, variety of algorithms have been given by different scholars. Different scholars use to classify these algorithms on their own assumptions and requirements. Considering the criteria that "who is responsible for sequencing?" then one can classify theses algorithms in following categories[5]: (a) fixed sequencer atomic broadcast (b) moving sequencer atomic broadcast (c) privilege based atomic broadcast (d) communication history based atomic broadcast and (e) destination agreement based atomic broadcast mechanisms. Fixed sequencer is the easiest, where one dedicated process is there for sequencing of messages but at high load or in case of sequencer failure the whole system will suffer. Though mechanisms like, Amoeba [12], MTP [13], Tandem [14], [15], Jia [16], ISIS [17], [18], Phoenix [19] and Rampart [20, 21] are fixed sequencer based and can tolerate crash but for any researcher it's always a conundrum to face sequencer failure and bad performance at high load. So to get rid of this problem moving sequencer is a best option where not a fixed process will be sequencer. RMP[1], DTP[2] ,pin wheel[3] and [4] are based on moving sequencer and tolerate crash failure but not capable to tolerate the omission Failure. So this work has proposed a new mechanism to build atomic broadcast that is based on moving sequencer and will tolerate the omission failure.

The process can be crash due to network disconnection, system restart, buffer overflow or due to any other temporary reasons. The system must be efficient enough to tolerate such type of failure so that the reliability should be maintained.

The failure may be different types as (i) *Crash failure;* where process gets crashed at all and not responding. (ii) *Omission failure;* where process is omitting to do some work. (iii) *Timing failure;* it is due to time out. It occurs in synchronous system and (iv) *Byzantine failure;* where process is behaving completely maliciously. It means there is no fix pattern of its behavior.

This paper focuses only on omission tolerance. Since this paper is extension of [4] hence it will also tolerate crash failure.

2. CONTRIBUTION OF THE PAPER
The paper contributes a tranche in direction to achieve the fault tolerant systems. It presents omission tolerance in moving sequencer based atomic broadcast. The B [6] formal method is used to design this model. Pro B [7] model animator and checker tool is used to verify this model for any deadlock, constraint violations, error and inconsistencies. The results are obtained in sequential steps.

3. SYSTEM MODEL

This work assumes an asynchronous system composed of n processes belongs to a set π = {P1, P2 ... Pn}. For simplicity, this model considers a set of three processes as: Process belongs to π and Process = {P0, P1, P2}. The processes communicate by message passing over reliable channels. Message is a set of messages, for simplicity, this model considers a set of three messages as: Message = {M1, M2, M3}.

Since this work is an extension of [4] hence, Network is reliable, uses unicast broadcast (UB) variant of fixed sequencer atomic broadcast, based on moving sequencer and by default crash tolerant.

3.1 Agreement Problem

The agreement problem considered in this paper is presented below.

3.1.1 Atomic Broadcast

Atomic broadcast problem is defined by primitive [8] a_broadcast and a_delivers, the processes have to agree on a common order on a set of messages. Formally atomic broadcast (uniform) can be defined by four properties [5];

Validity: if a correct process a_broadcast any message m then it eventually a_delivers m.

Uniform agreement: If a process a_delivers m then all the correct processes a_deliver m.

Uniform integrity: For any message m, every process p, a_delivers m at most once and only if m was previously a_broadcast.

Uniform total order: If some process, a_delivers m before m' then every process a_delivers m' only after it has a_delivered m.

3.1.2 Sequencer Based Algorithms

The sequencer based atomic broadcast [3] is simplest one and provides best delivery time (in absence of failure) while the protocols based on privilege provide best stability time in system where logical ring is formed and message is passed along with token. This work relies on sequencer based approach where any process can be elected as sequencer.

4. RELATED WORK

There is lot of work have been done since 25 years in area of atomic broadcast. The RMP [1], DTP [2], Pin Wheel [3] and [4, 22] are the various mechanisms to achieve moving sequencer based atomic broadcast. In moving sequencer mechanisms, there must be some process that is responsible for sequencing. But this sequencer will not be fixed for whole time. Each process will be a sequencer in a rotation manner. It is somewhat easier that privilege based atomic broadcast mechanisms. All these mechanisms help to build atomic broadcast but they can tolerate only crash failures.

Different authors have given various mechanisms base on communication history (where sender processes are itself responsible for sequencing) to build atomic broadcast but most of these algorithms can only tolerate crash failure. HAS [23] can tolerate crash and omission both type of failures and Quick-S [24] (for synchronous system) can tolerate crash, omission and Byzantine failures.

A variety of algorithms are also given for atomic broadcast based on destination agreement where the destination processes are responsible for arranging the messages before delivery. But most of these algorithms can tolerate only crash failure except Le –Lann Bres [25] and Quick-A [24]. Le-Lann Bres [25] can tolerate crash and omission both while Quick-A [24] (for asynchronous system) is capable for tolerating byzantine failure also. Rampart [20, 21] is based on fixed sequencer and can tolerate crash, omission and byzantine failures. Scholarly observation of these algorithms is that, there is still a space to achieve omission and byzantine tolerance in case on moving sequencer atomic broadcast. This work focuses on [4] and present a mechanism to tolerate omission failures.

5. ARCHITECTURE OF PROPOSED WORK

This work relies on incremental approach (see fig. 1) to design a model of atomic broadcast. The work that has been done in [4] will be used as abstract model. This work is a refinement of abstract model [4] that tolerates omission failure.

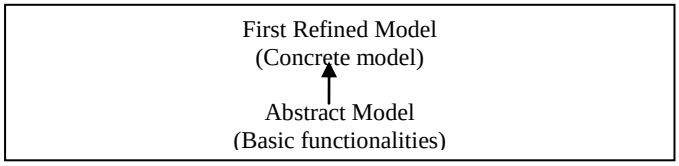

Figure 1 Architecture of proposed work

6. ABSATRACT MODEL

An abstract model represents the basic functionality of any system. This became more accurate when refines in next versions. Here, [4] has been considered as an abstract model (it is based on unicast broadcast (UB) variant of fixed sequencer and tolerates crash failure in order to build moving sequencer based atomic broadcast) and introduced refined version that will tolerate omission failure. Table 1 represents the various B symbols used in model.

Table 1. B symbols used in model

B symbols	Description
:	Element of
/:	Not element of
<:	Subset
/<:	Not subset of
!	For every
X	Cartesian product
POW	Power Set
<->	Relation
+->	Partial function
-->	Total Function
R~[A]	Relational Inverse
\ /	Set union
/ \	Set intersection
: =	Assignment
||	Parallel substitution
PRE	Pre-condition
BOOL	Boolean
NATURAL1	Non zero natural number
Card	Cardinality
Ran	Range of realtion
Dom	Domain of Relation

The following section presents the informal definition of different events given in abstract model [4]. The B model is build up with sets, constants, variables, Invariant and events. The fig. 2 summarizes all the abstract machine variables with their corresponding initial values and constraints (or invariant).

6.1 Events

This section presents informal definition of different events given in [4].

6.1.1 Sequencer Selection Event

The sequencer selection event will elect any process as sequencer. This event will ensure that no crashed process will participate in election.

6.1.2 Check Sequencer's Heartbeat Event

This event is used by all processes (except sequencer) to decide sequencer is crashed or alive.The processes will check heartbeat of sequencer and cast their vote for sequencer to confirm whether sequencer is alive or crashed.

6.1.3 Voting for Sequencer Event

After casting of vote for sequencer this event comes into existence. Based on votes it decides whether sequencer is alive or not.

If more processes are casting their vote for alive nature of sequencer than crash nature then it will be a trusted sequencer and ready to accept messages.

6.1.4 Unicast Event

If any process (except sequencer) needs to broadcast any message then at first it will use unicast event to unicast its message to sequencer.

6.1.5 Acknowledgement By Sequencer Event

After receiving the message sequencer will send an acknowledgement to sender.

6.1.6 Check Heartbeat Event

Before any broadcast sequencer will check heartbeat of all the processes (receivers) such that it can prepare list of alive and crashed receivers.

6.1.7 Broadcast Event

Broadcast event will be used by trusted sequencer to broadcast all acknowledged messages with proper sequence number to all alive processes.

6.1.8 Deliver Event

This event will occur at every alive process to deliver the messages. The messages will deliver in same order and this order is specified by *follow* variable.

6.1.9 Crash Event

This event is used to introduce crash nature of processes. Any process can be crash due to system shutdown, network disconnection or due to some other temporary reasons.

If any process has been crashed then it is not suppose to send or receive any message.

6.1.10 Get Alive Event

This event is used to recover any crashed process. As any crash process get recover it will intimate sequencer (if exists) about its recovery, and ask to sequencer for all previously broadcasted messages. If it founds any difference between its receiving list and sequencer's "sent message" list then it will deliver all old messages, if there is no difference in messages then still it will work as usual.

```
MACHINE Abstract1
SETS
Process= {P1, P2, P3};  Message={M1, M2, M3}
VARIABLES
selected_sequencer,sequencer_selection, unicast_message,
temporary_receive, follow, sent, seq_no, receive,
msg_with_seq_no, acknowledged_message
INVARIANT
selected_sequencer :        POW(Process)
sequencer_selection :       BOOL
unicast_message :           Process <-> Message
temporary_receive :         Process <-> Message
follow : NATURAL1<->NATURAL1
sent :  (Process<->Message)<->NATURAL1
seq_no :        NATURAL1
receive :  (Process <-> Message)<-> NATURAL1
msg_with_seq_no :Message<-> NATURAL1
acknowledged_message:Process<->Message
crash_list:POW(Process)
alive_list:POW(Process)
crash_list /\ alive_list={}
trusted_sequencer:POW(Process)
Receiver_is_Crashed:POW(Process)
Receiver_is_OK:POW(Process)
Receiver_is_OK /\ Receiver_is_Crashed={}
received_msg:Process<->Message
Heart_Beat_Check: Process<->Process
Re_Unicasted_msg:Process<->Message
Crash_Recoverd_Ack : POW(Process)
Message_diff:Process+->INTEGER
check_seq_heartbeat:Process+->(Process<->BOOL)
vote_for_sequencer:INTEGER
Positive_vote_for_sequencer:INTEGER
Negative_vote_for_sequencer:INTEGER
Start_unicast:BOOL
Sequencer_heart_beat_check_is_over:BOOL
voting_at_final_stage_for_process:POW(Process)
INITIALISATION
selected_sequencer :={} ||sequencer_selection :=FALSE ||
unicast_message :={} || temporary_receive :={} ||
follow :={} || sent :={} seq_no :=1 ||receive :={}||
msg_with_seq_no :={}||crash_list:={}|| alive_list:=Process ||
trusted_sequencer:={} ||Receiver_is_Crashed:={} ||
Receiver_is_OK:={}                          ||received_msg:={}||
Heart_Beat_Check:={} ||Re_Unicasted_msg:={} ||
Crash_Recoverd_Ack:={} ||Message_diff:={} ||
check_seq_heartbeat:={} ||vote_for_sequencer:=0 ||
Positive_vote_for_sequencer:=0
|| Negative_vote_for_sequencer:=0                          ||
Start_unicast:=FALSE||
Sequencer_heart_beat_check_is_over:=FALSE ||
voting_at_final_stage_for_process:={}
```

Figure 2 Variables, Invariants and their initial value in abstract model

7. REFINED MODEL

The different events, variables and invariants (see fig. 2) discussed in section VI constitutes moving sequencer atomic broadcast that tolerates crash failure. This refined version presents omission tolerance. For this purpose this work has introduced some new constants and variables (see fig. 3).

The *unicast_message_size* is a new constant that will put a restriction on sequencer's buffer to control the send omission.

The *Receiver_buffer_size* is a constant that will put a restriction on receiving processes' buffer to control receive omission.

Variable *Receiver_buffer_size_counter* contains the list of receiver processes along with available buffer.

Variable *final_updated_msg_list* contains finally delivered message corresponding to each process.

Variable *local_order* is similar to follow variable but contains the sequence of messages at receiver end. It helps during *final updation* of messages in total order at receivers. The sequence of *delivered message list* and *final updated message list* must be same and in Total order.

Omission failure can be of three types; (i) *Send omission:* due to sender's buffer overflow. (ii) *Receive omission:* due to receiver's buffers overflow. (iii) *Network omission:* due to unreliable network.

Since this work assumes reliable network hence it will not consider network omission.

```
REFINEMENT Refine2_Omission_Tolerant
REFINES Abstract1
CONSTANTS
unicast_message_size, Receiver_buffer_size
PROPERTIES
unicast_message_size:NATURAL1 &
Receiver_buffer_size:Process-->NATURAL1
VARIABLES
Receiver_buffer_size_counter,
final_updated_msg_list,local_order
INVARIANT
Receiver_buffer_size_counter:Process+->NATURAL
final_updated_msg_list:Process<->Message
local_order:Message<->Message
card(acknowledged_message)<=unicast_message_size
INITIALISATION
Receiver_buffer_size_counter:=Receiver_buffer_size||
final_updated_msg_list:={}|| local_order:={}
```

Figure 3 Variables, Invariants and their initial value in first refined version

7.1 Procedure TO Tolerate Send Omission

Sequencer is responsible for broadcast hence this work has been focused on sequencer for any send omission. This model has been designed in such a way that sequencer will never be in such a case where buffer will overflow.

The sequencer has limited size of buffer where it keeps the acknowledged messages. Any process can unicast its message to sequencer but sequencer cannot receive all such messages since it has limited buffer. When any process unicast its message this message stores into *unicast message list*.

The *acknowledged message list* is sequencer's buffer and size of acknowledge message list varies from zero *unicast_message_size*. Initially sequencer picks some messages (maximum upto *unicast_message_size*) from *unicast message list* and keeps into acknowledged message list. If there is no free space at acknowledged message list (means if size of acknowledged message list is equals to *unicast_message_size*) then it will not pick any more messages from unicast message list. As sequencer broadcasts any message it clears its entry from acknowledged message list.

In this way buffer will available to receive some more message from unicast message list. So there will be no case of sequencer buffer overflow and hence no send omission.

7.1.1 Strengthening of Acknowledgement by Sequencer Event

Before any acknowledgement sequencer will check its buffer. For this new guard has been introduced to strengthen the acknowledgment by sequencer event.

Guard:

Cardinality (acknowledged_message)<=unicast_message_size

7.1.2 Strengthening of Broadcast Event

As sequencer will broadcast any message it will clear its entry from acknowledged message list. For this new functionality has been added to broadcast event.

Action:

acknowledged_message:=acknowledged_message-{p|->m}

As it will delete messages from acknowledged message list the free space will be there to keep another message.

7.2 Procedure to Tolerate Receive Omission

Each receiver has a limited buffer capacity. Receive omission occurs at receivers due to overflow of receive buffer. This model has been designed in such a way that no receive omission will occur.

Before any delivery the receiver will check its available buffer. If there is free space then it will deliver that message otherwise wait until space will free. As any message has been delivered by each process that message will be finally updated at each site. As any message will finally update at any site the space taken by it at receiver buffer will get free and it can deliver new message.

To achieve this; the receiver's buffer size has been represented with a constant (i.e. *Receiver_buffer_size*), Variable *Receiver_buffer_size_counter* that contains the list of receiving processes with corresponding free buffer size. Informally, if *Receiver_buffere_size_counter* is zero means there is no free space to deliver and if *Receiver_buffere_size_counter* is equal to *Receiver_buffer_size* then it means buffer at corresponding process is completely free to deliver.

For each process buffer size (*Receiver_buffer_size_counter*) varies from zero to *Receiver_buffer_size*.

Initially *Receiver_buffere_size_counter* has been initialized by *receiver_buffer_size* (it means initially the buffer of all receivers is empty).

Before any delivery processes will check their available buffer (*Receiver_buffere_size_counter*); if it is zero then it will not deliver (since buffer is full) otherwise it can deliver. After delivery of each message it will decrease its buffer size by one.

In this way there will be no case of receiver's buffer overflow and hence no receive omission.

7.2.1 Strengthening of Deliver Event

To strengthen deliver event such that receive omission can be tolerated this work introduces new guard.

Gurad: (p|->0) /: Receiver_buffer_size_counter

Guard ensures that if some process p having available buffer size is zero (i.e. p|->0), it cannot deliver message. Two new functionalities have been added to deliver event.

Action 1:

*IF m/: ran(received_msg) THEN local_order:=local_order\/ {m} * ran(received_msg) END*

Received_msg, variable (see fig. 1) keeps the list of processes and corresponding delivered messages. This action 1 specifies that for every newly delivering message there will be a precedence relationship with previously delivered messages will build, and this relationship is represented with *local_order*. For example *local_order {m2|->m1}* indicates that *m2* is proceeding to *m1; m2* will deliver only after delivery of *m1* at any process.

The *local_order* will helpful for *final updation* of messages. Since local delivery and final delivery must be in same order such that total order should be maintained.

Action 2: Receiver_buffer_size_counter(p):=Receiver_buffer_size_counter (p)-1

As any process deliver any message it will decrease its available buffer size by one.

7.2.2 Final Updation Event

As any message will locally deliver by each process then it will finally update at each process (see fig.4). Final updation means message is finally delivered at the process.

```
final_updation(p,m)=
PRE
p:Process & p:dom(received_msg) & m:ran(received_msg)&
card(received_msg~[{m}])=card(alive_list) &
(p|->m)/:final_updated_msg_list
& !(mm).(mm:Message &(m|->mm):local_order =>(p|->mm):final_updated_msg_list)
THEN
Action1:
final_updated_msg_list:=final_updated_msg_list\/{p|->m}
Action 2:
Receiver_buffer_size_counter(p):=Receiver_buffer_size_counter(p)+1
END END
```

Figure 4 Final updation event

Guard 1 and 2: p: Process & p: dom(received_msg) illustrates that any process in system that is participating in delivery (means it must not be crashed process) will deliver message.

Guard 3 and 4: m: ran(received_msg) & card(received_msg~[{m}])=card(alive_list)

These guards ensure that *final updation event* will occur only for those messages that have been delivered by each processes.

Guard 5: (p|->m)/: final_updated_msg_list

This guard ensures that any process cannot repeatedly deliver same message.

Guard 6:

!(mm).(mm:Message &(m|->mm):local_order =>(p|->mm):final_updated_msg_list)

This guard ensures that message will finally deliver in total order.

Action 1:

final_updated_msg_list:=final_updated_msg_list\/{p|->m}

As any process will finally deliver any message then *final_updated_msg_list* will be updated.

Action2:

Receiver_buffer_size_counter(p):=Receiver_buffer_size_counter (p)+1

As any process will finally deliver any message then buffer will be free to receive new message.

8. RESULT

The model has been verified by Pro B [7] model checker and animator tool. No invariant violations, errors and deadlock have been found. The B model animated through Pro B worked very well. The Pro B managed to explore the entire state space of the B-machine in 59 seconds, covering 37322 states and 58329 transitions. This model has been animated for various numbers of random operations. The table 2 presents the status of various variables after 500 random operations.

The model has randomly assumed the size of constant *unicast_message_size = 2* (it is sequencer's buffer size) and *Receiver_buffer_size = {(p1|->2),(p2|->3),(p3|->1),(p4|->1)}* (i.e. Process p1 has been assigned with receive buffer capacity=2, similarly p2 with 3, p3 with 1 and p4 with 1). *selected_sequencer = {p3} and trusted_sequencer = {p3}* indicates that p3 is a trusted sequencer. The *sent = {({(p4|->m1)}|->4),({(p4|->m2)}|->3),({(p4|->m3)}|->2),({(p4|->m4)}|->1)}* indicates that previous trusted sequencer p4 has broadcasted m4 with sequence number 1, m3 with sequence number 2, m2 with sequence number 3 and m1 with sequence number 4. The *follow = {(2|->1),(3|->1),(3|->2),(4|->1),(4|->2),(4|->3)}* variables shows a precedence relationship among the different sequence numbers. As (2|->1) indicates sequence number 2 is following to sequence number 1. And at receiver, message having sequence number 2 will only deliver to some receiver if that receiver has delivered message having sequence number 1. The follow variable guaranties delivery of messages in same sequence to each receiver.

Investigate the *final_updated_msg_list = {(p1|->m1),(p1|->m2),(p1|->m3),(p1|->m4),(p2|->m1),(p2|->m2),(p2|->m3),(p2|->m4),(p3|->m1),(p3|->m2),(p3|->m3),(p3|->m4),(p4|->m1),(p4|->m2),(p4|->m3),(p4|->m4)}*, this list clearly indicates that messages delivered at each process are same.

And investigation of *receive= {({(p1|->m1)}|->4),({(p1|->m2)}|->3),({(p1|->m3)}|->2),({(p1|->m4)}|->1),({(p2|->m1)}|->4),({(p2|->m2)}|->3),({(p2|->m3)}|->2),({(p2|->m4)}|->1),({(p3|->m1)}|->4),({(p3|->m2)}|->3),({(p3|->m3)}|->2),({(p3|->m4)}|->1),({(p4|->m1)}|->4),({(p4|->m2)}|->3),({(p4|->m3)}|->2),({(p4|->m4)}|->1)}* list reports that messages have been delivered to each process and in same order (or sequence); hence confirming to atomic broadcast definition.

Table 2. Evaluation view

Constants	Variables	Values	Variables	Values
unicast_message_size		2	Message_diff	{(p1|->0),(p2|->0),(p4|->0)}
Receiver_buffer_size		{(p1|->2),(p2|->3),(p3|->1),(p4|->1)}	check_seq_heartbeat	{(p1|->{(p3|->TRUE)}),(p2|->{(p3|->TRUE)}),(p4|->{(p3|->TRUE)})}
	selected_sequencer	{p3}	vote_for_sequencer	0
	sequencer_selection	TRUE	Positive_vote_for_sequencer	0
	unicast_message	{}	Negative_vote_for_sequencer	0
	temporary_receive	{}	Start_unicast	TRUE
	follow	{(2|->1),(3|->1),(3|->2),(4|->1),(4|->2),(4|->3)}	Sequencer_heart_beat_check_is_over	FALSE
	sent	{({(p4|->m1)}|->4),({(p4|->m2)}|->3),({(p4|->m3)}|->2),({(p4|->m4)}|->1)}	voting_at_final_stage_for_process	{}
	seq_no	5	non_deletable_ack_msg_log	{}
	Receive	{({(p1|->m1)}|->4),({(p1|->m2)}|->3),({(p1|->m3)}|->2),({(p1|->m4)}|->1),({(p2|->m1)}|->4),({(p2|->m2)}|->3),({(p2|->m3)}|->2),({(p2|->m4)}|->1),({(p3|->m1)}|->4),({(p3|->m2)}|->3),({(p3|->m3)}|->2),({(p3|->m4)}|->1),({(p4|->m1)}|->4),({(p4|->m2)}|->3),({(p4|->m3)}|->2),({(p4|->m4)}|->1)}	final_updated_msg_list	{(p1|->m1),(p1|->m2),(p1|->m3),(p1|->m4),(p2|->m1),(p2|->m2),(p2|->m3),(p2|->m4),(p3|->m1),(p3|->m2),(p3|->m3),(p3|->m4),(p4|->m1),(p4|->m2),(p4|->m3),(p4|->m4)}
	msg_with_seq_no	{(m1|->4),(m2|->3),(m3|->2),(m4|->1)}	Receiver_buffer_size_counter	{(p1|->2),(p2|->3),(p3|->1),(p4|->1)}
	acknowledged_message	{}	local_order	{(m1|->m2),(m1|->m3),(m1|->m4),(m2|->m3),(m2|->m4),(m3|->m4)}
	crash_list	{}	Receiver_is_Crashed	{}
	alive_list	{p1,p2,p3,p4}	Receiver_is_OK	{p1,p2,p4}
	trusted_sequencer	{p3}	received_msg	{(p1|->m1),(p1|->m2),(p1|->m3),(p1|->m4),(p2|->m1),(p2|->m2),(p2|->m3),(p2|->m4),(p3|->m1),(p3|->m2),(p3|->m3),(p3|->m4),(p4|->m1),(p4|->m2),(p4|->m3),(p4|->m4)}

No invariant violation, error or deadlock has been observed. During animation buffer overflow has been introduced at sender and receivers but this model has successfully tolerated such problems; Hence also confirming to omission tolerance.

9. CONCLUSION

This paper presents mechanism to tolerate omission failure in moving sequencer atomic broadcast. Since this paper is an extension of [4] hence it rely upon unicast broadcast (UB) variant of fixed sequencer to build moving sequencer atomic broadcast and also tolerates crash failures. However in future there is also a good scope to work with byzantine failures. For

any message loss one can also use negative and positive acknowledgement [26] to recover it. Pro B [7] model checker and animator tool has been used for modeling and step by step checking. This model has been checked for invariant violation or for any deadlock occurrence. The B machine animated through Pro B worked very well. On injecting a subtle fault into the specifications, to verify the model, Pro B captured them automatically thereby substantiating the results.

10. ACKOWLEDGMENT

We are grateful to Dr. Divakar singh yadav for his valuable guidance. It gives us immense pleasure to express our deep sense of gratitude to Dr. S. L. Srivastava for encouragements during work. Last but not the least; we extend our heartiest gratefulness to our parents and all family members.

11. REFERENCES

[1] Jia, W., Kaiser, J., and Nett, E. 1996. RMP: Fault–Tolerant GroupCommunication. Micro, IEEE, Oxford, Clarendon, 16(2) , 59 – 67.

[2] Kim, J., and Kim C. 1997. A total ordering protocol using a dynamic token-passing scheme. Distributed System Engineering. 4(2), 87–95.

[3] Cristian, F., Mishra, S., and Alvarez, G. 1997. High performance asynchronous atomic broadcast. Distributed System Engineering 4(2), pp. 109-128.

[4] Srivastava, P., Lakhtaria, K., Panwar A., and Jain, A. 2014. Rigorous design of moving sequencer crash tolerant atomic broadcast with unicast broadcast. IEEE International Conference on Recent Advances and Innovations in Engineering – ICRAIE, Rajasthan, India.

[5] D´efago, X., Schiper, A., and Urb´an, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36(4), 372– 421.

[6] Abrial, J., R. 1996. The B-book: assigning programs to meanings Cambridge University Press New York. USA, ISBN:0-521-49619-5.

[7] Leuschel, M., Butler, M. 2003. Pro B: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME. Springer, Heidelberg, LNCS, 2805, 855-874.

[8] Ekwall, R., and Schiper, A. 2011. A Fault-Tolerant Token-Based Atomic Broadcast Algorithm. Dependable and Secure Computing, IEEE Transactions. 8(5), 625–639.

[9] Hadzilacos, V., and Toueg, S. 1993. Fault-Tolerant Broadcasts and Related Problems. Distributed systems (2nd Ed.), ACM Press/Addison- Wesley Publishing Co., New York, USA, 97-145.

[10] Lamport, L., 1978. The Implementation of Reliable Distributed Multiprocess Systems. Computer Networks. 2(2), 95–114.

[11] Schneider., and F. B. 1990. Implementing fault tolerant services using the state machine approach: a tutorial. ACM Computing Survey. 22(4), 299-319.

[12] Kaashoek, M. F. and Tanenbaum, A. S. 1996. An evaluation of the Amoeba group communication system. In Proceeding of 16th International Conference on Distributed Computing Systems (ICDCS-16). Hong Kong, 436–447.

[13] Armstrong, S., Freier, A., and Marzullo, K., 1992. Multicast transport protocol. Network working group. RFC 1301, IETF.

[14] Carr, R., 1985. The Tandem global update protocol. Tandem Systems Review. 74–85.

[15] Garcia-Molina, H. and Spauster, A. 1991. Ordered and reliable multicast Communication. ACM Trans. Comput. Syst. 9(3), 242–271.

[16] Jia, X. 1995. A total ordering multicast protocol using propagation trees. IEEE Trans. Parall. Distrib. Syst. 6, 617–627.

[17] Birman, K. P., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. 9(3), 272–314.

[18] Navaratnam, S., Chanson, S. T., and Neufeld, G. W. 1988. Reliable group communication in distributed systems. In Proceeding of 8th International Conference on Distributed Computing Systems (ICDCS-8). San Jose, CA, USA, 439–446.

[19] Wilhelm, U. and Schiper, A. 1995. A hierarchy of totally ordered multicasts. In Proc. 14th Symp. on Reliable Distributed Systems (SRDS), Bad Neuenahr, Germany, 106–115.

[20] Reiter, M. K. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In Proceeding of 2nd ACM Conference on Computer and Communications Security (CCS-2). 68–80.

[21] Reiter, M. K. 1996. Distributing trust with the Rampart toolkit. Communications of the ACM. 39(4), 71–74.

[22] Srivastava, P.,Lakhtaria, K., Jain, A. 2013. Rigorous design of moving sequencer atomic broadcast with unicast broadcast. In Proceeding of International Conference on Advances in computer science. Elsevier. 484-491.

[23] Cristian, F., Aghili, H., Strong, R., and Volev, D. 1995. Atomic broadcast: From simple message diffusion to Byzantine agreement. IEEE, Proceeding of FTCS-25, 431.

[24] Berman, P., and Bharali, A. A. 1993. Quick Atomic broadcast. Springer Berlin Heidelberg. LNCS. 725, 189-203.

[25] Le Lann, G. and Bres, G. 1991. Reliable atomic broadcast in distributed systems with omission faults. ACM Operating Systems Review. SIGOPS 25, 80–86.

[26] Chang, J. M., and Maxemchuk, N. F. 1984. Reliable broadcast protocols. ACM Trans. Comput. Syst. 2(3), 251–273.