Rigorous Design of Moving Sequencer Atomic Broadcast with Malicious Sequencer

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 malicious nature of sequencer in moving sequencer based atomic broadcast in 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 Srivastava et al. [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 (malicious) failure. This work proposes a mechanism to tolerate byzantine failure (malicious nature) of sequencer in moving sequencer based atomic broadcast. The mechanism proposed in [4], has been considered as an abstract model and design refined model in order to fulfill objective. Since it relies on unicast broadcast hence it will introduce a very less number of messages in comparison to previous mechanisms [5]. B [6] formal technique has been used for development of this model and Pro B [7] model checker tool for constraint based checking to discover errors due to invariant violation and deadlocks, thereby, validating the specifications. The models have been verified for invariant violations, errors and deadlock occurrence. The B machine animated through Pro B worked very well. The Pro B managed to explore the entire state space of the B-machine in few minutes and confirming the specifications.

General Terms
Distributed Systems, Model Verification

Keywords
Broadcast, Atomic Broadcast, Total Order, Unicast, Sequencer, Crash, Byzantine, 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. In result of this question that "who is responsible for sequencing?" these algorithms can be classified into 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 mechanism proposed in [4] are based on moving sequencer and tolerate crash failure but not capable to tolerate the byzantine failure. So this work proposes a new mechanism to build atomic broadcast that is based on moving sequencer and will tolerate the byzantine failure of sequencer. Subsequently this mechanism can apply to whole system in order to get byzantine resistant system.

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. Even in case of failures the system must be efficient enough to tolerate any failure such that availability and reliability should be maintained. This work focuses on byzantine nature of sequencer.

2. CONTRIBUTION OF THE PAPER
The paper contributes a tranche in direction to achieve the fault tolerant systems. It presents a mechanism that tolerates byzantine nature of sequencer 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 mechanism proposed in [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. Quick-S [23] (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 only Quick-A [23] (for asynchronous system) is capable for tolerating byzantine failure.

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 byzantine tolerance in case on moving sequencer atomic broadcast. This work focuses on mechanism proposed in [4] and presents a mechanism to tolerate byzantine nature (malicious nature) of sequencer.

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 byzantine failure (malicious nature) of sequencer.

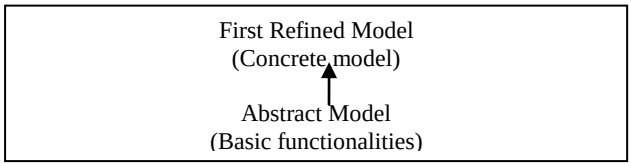

Figure 1 Architecture of proposed work

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 byzantine failure (malicious nature) of sequencer process. 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 discussed in section VI constitute moving sequencer atomic broadcast (see fig. 2) that tolerates crash failure. In refined version solution for byzantine sequencer has been given. For this purpose some variables and invariants have been taken (see fig. 3).

Variable *non_deletable_ack_msg_log* contains the list of acknowledged messages at sequencer. Variable *process_delivered_message_with_sequence_no* contains the list of all sequence numbers that have been delivered to some process. Variable *just_previously_delivered* contains the sequence number that has been just delivered at some process.

This work assumes following malicious cases that a sequencer can show:

i) Malicious sequencer can broadcast different messages with same sequence number.

ii) A malicious sequencer can broadcast same message with different sequence numbers.

iii) A malicious sequencer can broadcast any message with some jumping sequence number (sequence number must increase by one but if it is increasing by more than one then It is jumping. Sequence number is also not allowed in decreasing order).

Since it has been assumed that sequencer will broadcast the messages with unique and increasing sequence number (increasing by one only) hence occurrence of any of the above case will report for malicious nature of sequencer. Then immediately a new correct process will be elected as sequencer. After election heartbeat of sequencer will check by every correct process and cast their vote in order to elect a trusted sequencer. After election of trusted sequencer it will broadcast all such messages for which previous sequencer was reported as malicious.

```
REFINEMENT Refine2_Byzantine_Tolerant
REFINES Abstract_Moving_Sequencer_Atomic_Broadcast
VARIABLES
non_deletable_ack_msg_log, just_previously_delivered,
process_delivered_message_with_sequence_no
INVARIANT
non_deletable_ack_msg_log:POW(Message)
process_delivered_message_with_sequence_no:Process<-
>NATURAL1
just_previously_delivered:Process+->NATURAL
INITIALISATION
non_deletable_ack_msg_log:={}||
process_delivered_message_with_sequence_no:={}||
just_previously_delivered:=Process*{0}
```

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

7.1 Procedure TO Tolerate Byzantine Nature of Sequencer

In moving or fixed sequencer based atomic broadcast sequencer is a very important component. If this is incorrect then whole system will suffer. This work assumes that sequencer can be malicious and can introduce problems into the system. But receivers are so smart that they can understand that whether sequencer is malicious or not. If they found sequencer is malicious then they will not deliver such messages and report for malicious nature of sequencer. Subsequently new sequencer will be elected. As any new correct process will be elected as sequencer then heartbeat check for it will happen by all correct processes and subsequently voting will be done in order to elect a trusted sequencer. Now new trusted sequencer will broadcast all those messages for which previous sequencer has been reported as malicious.

To introduce malicious nature of sequencer one new event *faulty_broadcast* has been added. And for re broadcast of messages for which previous sequencer was reported as faulty a new event (named as *re_broadcast*) has been introduced. The receivers have been strengthened such that they can identify malicious sequencer. In this way Sequencer can introduce errors into the system but receivers can tolerate this.

7.1.1 Strengthening of Acknowledgement by Sequencer Event

As sequencer will acknowledge any message it will also update a list *non_deletable_ack_msg_log*. For this a new action has been introduced:

Action 1:

non_deletable_ack_msg_log:=non_deletable_ack_msg_log \/{m}

7.1.2 Faulty Broadcast Event

By this event (see fig. 4) sequencer can introduce errors into the system like, it can broadcast different messages with same sequence number, re broadcast same message with different sequence number or broadcast with jumping sequence number.

```
faulty_broadcast(p,m,number)=
PRE p:selected_sequencer & m/:ran(temporary_receive) &
number: NATURAL1 & p:trusted_sequencer &
m:ran(acknowledged_message) &
card(Receiver_is_OK) + card(Receiver_is_Crashed) =
card(Process) &
number /: ran(sent) & m/:ran(final_updated_msg_list)
THEN
temporary_receive:=temporary_receive\/{p|->m}
||sent:=sent\/{{p|->m}|->number} ||
follow:=follow \/{number} * ran(sent)||
msg_with_seq_no:=msg_with_seq_no\/{m|->number}
acknowledged_message:=acknowledged_message-{p|->m} ||
IF card(acknowledged_message)-1=0
THEN
sequencer_selection:=FALSE ||
trusted_sequencer:={}||
Sequencer_heart_beat_check_is_over:=TRUE ||
check_seq_heartbeat:={}||
unicast_message:={}
END
END
```

Figure 4 Faulty broadcast event.

7.1.3 Strengthening Deliver Event

There are some more capabilities have been provided to receivers such that before any delivery they will decide whether message is coming with correct sequence number or not. For this purpose some conditions have been applied to check before any delivery.

Condition 1: *card(receive)=0 & sequence_num /=1*

Condition 2: *(p:dom(just_previously_delivered)& ((num-just_previously_delivered(p) >1) or(num-just_previously_delivered(p)<0))*

Condition 3: *(p|->m): received_msg*

Condition 4:
(sequence_no:process_delivered_message_with_sequence_no[{p}])

Condition 1 provides capability to receivers to not deliver such message that is very first message to deliver and sequence number not equal to one, Condition 2 provides capability to receivers to not deliver any new message having jumping sequence number, Condition 3 provides capability to receivers to not deliver any new message that has been already delivered (It means old message is again coming with some different sequence number) and Condition 4 provides capability to receivers to not deliver any new message that is coming with some old sequence number (It is the case of new message coming with some old sequence number).

If any of above condition will found true then for such message delivery will not happen and sequencer will reported as malicious and new sequencer will be elected.

Action 1:
process_delivered_message_with_sequence_no:=process_delivered_message_with_sequence_no\/{p|->num}

Action 2: *just_previously_delivered(p):=num*

Action 3:
non_deletable_ack_msg_log:=non_deletable_ack_msg_log-{m}

Action 1 maintains a list of sequence numbers that have been delivered to some process, Action 2 maintains immediate delivered sequence number at different processes and Action 3 maintains a list of such messages that have been broadcasted but not delivered. This list will helpful in case of re broadcast of messages.

7.1.4 Re Broadcast Event

As sequencer reported as malicious new sequencer will be elected. After election all other processes will check heartbeat of sequencer and cast their vote. Finally there will be a new trusted sequencer. Now this trusted sequencer will broadcast all those messages for which old sequencer was reported as malicious (see fig. 5). The question arises, how sequencer comes to know that which messages need to re broadcast? For this purpose sequencer will check *non_deletable_ack_msg_log* list (*see deliver event*) that keeps those messages for which previous sequencer was reported as malicious.

Guard 1: *p: selected_sequencer*

Guard 2: *p: trusted_sequencer*

Guard 1 and 2 ensures that sequencer must be trusted. It means all the correct processes in the system have checked sequencer's heartbeat and casted their votes. Any sequencer can be trusted only if it has secured more positive votes in comparison to negative votes.

Guard 3: *number: NATURAL1*

Guard 4: *number/: ran (sent)*

Guard 3 and 4 ensures that sequencer will broadcast a sequence number that must be positive and unique natural number.

Guard 5: *m: non_deletable_ack_msg_log*

Guard 5 ensures that new trusted sequencer will broadcast all those messages for which sequencer were reported malicious.

```
Re_broadcast(p,m,number)=
PRE p: selected_sequencer & p: trusted_sequencer &
number: NATURAL1 &  m:non_deletable_ack_msg_log
& number/:ran(sent)
THEN
temporary_receive:=temporary_receive\/{p|->m}                ||
sent:=sent\/{{p|->m}|->number} ||
follow:=follow      \/{number}      *      ran(sent)      ||
msg_with_seq_no(m):=number ||
seq_no:=seq_no+1
END
```

Figure 5 Re broadcast event

8. RESULT

The models have been verified by Pro B [7] model checker and animator tool. No invariant violations, errors and deadlock have been found. The B machine animated through Pro B worked very well. The Pro B managed to explore the entire state space of the B-machine in few minutes, covering 2265 states and 2623 transitions. The values of *sent* and *receive list* obtained after covering all the operations are:

sent = {({(p4|->m1)}|->2),({(p4|->m2)}|->3),({(p4|->m3)}|->1),({(p4|->m4)}|->4)}

receive = {({(p1|->m1)}|->2),({(p1|->m2)}|->3),({(p1|->m3)}|->1),({(p1|->m4)}|->4),({(p2|->m1)}|->2),({(p2|->m2)}|->3),({(p2|->m3)}|->1),({(p2|->m4)}|->4),({(p3|->m1)}|->2),({(p3|->m2)}|->3),({(p3|->m3)}|->1),({(p3|->m4)}|->4),({(p4|->m1)}|->2),({(p4|->m2)}|->3),({(p4|->m3)}|->1),({(p4|->m4)}|->4)}

The *sent list* specifies that *sequencer p4* has broadcasted *m3* with sequence number *1*, *m1* with sequence number *2*, *m2* with sequence number *3*, *m4* with sequence number *4*. The *receive list* specifies that all the processes in system have received all messages in the same order (as broadcasted) without any invariant violation, error or deadlock; confirming to definition of total order.

9. CONCLUSION

This paper presents mechanism to tolerate byzantine failure (malicious nature) of sequencer 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. For any message loss one can also use negative and positive acknowledgement [24] 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] Berman, P., and Bharali, A. A. 1993. Quick Atomic broadcast. Springer Berlin Heidelberg. LNCS. 725, 189-203.

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