An Agreement Protocol to exploit and handle Byzantine Faulty Nodes in Authenticated Hierarchical Configuration

Poonam Saini and Awadhesh Kumar Singh

Department of Computer Engineering, National Institute of Technology, Kurukshetra, 136119, India
nit.sainipoonam@gmail.com, aksinreck@rediffmail.com

Abstract

Consensus algorithms that, essentially, endeavor agreement or commit on a particular transaction, are preeminent building blocks of distributed systems. Outside the FLP *impossibility results in an asynchronous environment to failure detectors and many more tactics for synchrony supplements, consensus has always been a major part of concern. It poses more severe threats, in case, the distributed network possesses some arbitrary behaving (malicious) nodes. The proposed article adds up a non-faulty agreement decision to the requesting client nodes from the coordinator replicas. The work is divided into two-phases, namely, a fault-free and fair cluster formation which employs authenticated key management scheme and secondly, an authenticated agreement among the cluster heads resulting in secure and correct outcome of a transaction. We assume a two-layer hierarchy with different clusters of replicas in one layer associated with their cluster heads on another layer. Multiple levels of encryption are incorporated by means of two keys: (1) a unique pair-wise key between the processes, and (2) a communication key that provides more authenticity and enables a secure communication among the processes. The necessary correctness proof has also been presented. The protocol is robust and exhibits better efficiency for long-lived systems.*

Keywords: *Authentication, Byzantine Failures, Consensus, Distributed Transactions, Hierarchical Cluster Formation, Key Management.*

1. Introduction

The Consensus is fundamental for various distributed computing applications *e.g.*, atomic commitment, atomic broadcast, files replication [3, 4, 5]. Informally, the consensus problem is for a set of processes to agree on a value proposed by one or more processes [1]. A process is said to be correct if it behaves according to an agreed specification. Otherwise, a failure is said to prevail. The consensus protocols rely on the following three correctness properties:

- *Termination*: Eventually, every correct process decides some value.
- *Agreement*: All non-faulty processes decide the same value.
- *Validity*: The agreed value must be proposed by some processe(s).

The problem of agreement has been widely studied by many research groups in crash-stop as well as Byzantine model for distributed transaction systems. If the system is fault-free, it makes the solution easier; however, in case of failures, it becomes challenging [2]. Moreover,

in an asynchronous system, consensus is unsolvable even with the failure of only one process [6]. One way to elude the famous FLP impossibility result relies on the assumption of some element of synchrony supported by the presence of unreliable failure detectors [7]. A failure detector can be visualized as a distributed oracle that indicates (probably incorrect) about a crashed process so far. Some more primitives have also been used in the agreement protocols to ensure fault tolerance, namely, reliable snapshot, checkpointing, quorum replication, eventual bisource and server replication (also called primary-backup approach). However, the presence of a malicious replica, which may act in a superfluous manner, may disrupt the complete transaction processing.

In the proposed work, we impose a hierarchical-structure in order to minimize the overall communication cost and message overhead. Moreover, it will help in reducing the latency of the transaction processing, from request initiation to final decision, and speed up the agreement among the processes. Furthermore, the proposed authenticated cluster formation technique will enhance the reliability and safety of the protocol by segregating out the faulty processes. Although, the logical hierarchy has been extensively used in mobile and wireless environments, the same has not been used in static environment, especially for secure consensus.

1.1. Background

Traditional Byzantine fault tolerant protocols such as PBFT [8], BFTDC [9] use a combination of primary-backup and quorum replication technique to order requests. The replicas move through a succession of configurations, named as a view. In a particular view, one of the replicas is chosen as primary and other replicas work as backups. In the middle of agreement, if time out occurs for current view because of delay in message propagation or the primary is found faulty then view change occurs. It uses MAC to make the system faster. The major limitation of the said protocols outbreak, in case a new view is formed. At every initiation for a fresh view, the timeout is doubled directing an exponentially growing timeout and resulting in overall performance degradation of the system. Moreover, the authors assume a (fault-free) coordinator and a (fault-free) quorum to maintain consistency and to enhance the system performance. Also, it may so happen that a faulty replica may initiate a view change. Another protocol named Zyzzyva [10] employs a speculative execution to improve the performance in terms of latency and number of messages. It intends to reduce the cost and simplify the design of Byzantine state machine replication. It maintains integrity in the system using signed checkpoint messages. Although, the protocol shows an overall improved efficiency over some conventional protocols, the speculation cost could not be avoided. It is worth mentioning that the protocols, which rely on optimism, usually suffer heavy cost of system throughput in case of long-running applications.

1.2. Motivation

In the literature, every BFT state machine replication approach, which is designed for consensus protocol in a distributed transaction system, confronts the effect of malicious activities of faulty processes. We attempt to design a system where the focus of the study, primarily, is to restrict the entry of an arbitrary behaving process into the formation of the primary and backup replicas and then reflect the faulty actions, if any, along with their remedies. In our protocol specification, a large number of replicas are collated into some clusters to be placed at one level and elect cluster heads from each cluster, maintained in a ring structure, at another level, thus, assembling the system into a two-level hierarchy. While

configuring a cluster, it is authenticated by utilizing random key pre-distribution [11] and a key-management scheme [12, 13]. The protocol also incorporates a second level of authentication for the cluster heads resulting in an elongated fault-free structure. Afterwards, the agreement is executed where a primary process, from among the ring of cluster heads, is responsible to finally declare the outcome of the transaction.

The approach is *proactive* and provably more efficient, in a sense; it detects the faulty processes in advance and constrains them from participating in the agreement process at initial stage. In comparison, the existing commit protocols [8, 9, 10] do not imply the notion of proactive fault tolerance as they rely on the specification of the faults to circumvent them which makes them *reactive*. The proactive fault tolerance minimizes the transaction discontinuity and latency. To the best of our knowledge, this is the first hierarchical cluster-based consensus protocol for distributed transaction processing system where the agreement is required among a finite number of processes.

1.3. Organization of the Chapter

In section 2, we formalize the computation model. Section 3 describes the terms and assumptions. The detailed approach of authenticated cluster and authenticated ring formation has been discussed in section 4. Section 5 explains the authenticated agreement protocol. Following this, in section 6, we analyze the security of the authenticated key management scheme. Section 7 describes the correctness proof and section 8 shows the simulation results. In section 9, we present the conclusion which summarizes the work and explores future aspects.

2. Computation Model and Design Goals

We impose a two-layer hierarchical structure with clusters containing replicas in one layer and cluster head, arranged in a unidirectional ring, in another. The schematic view is shown in figure 1.

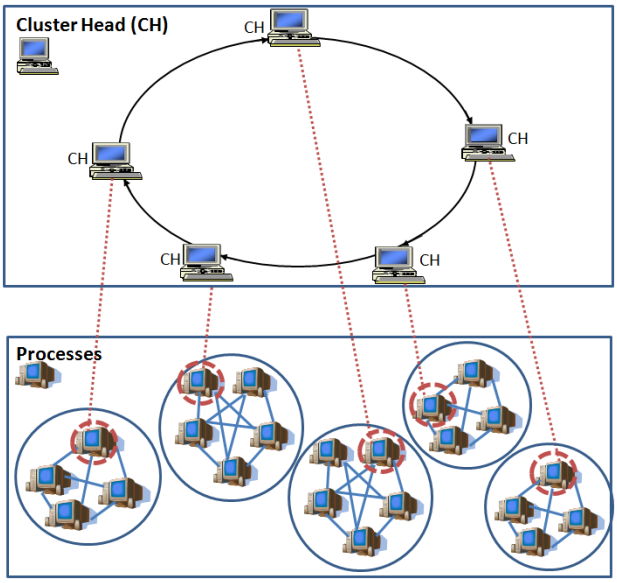

Fig. 1: Schematic View of 2-level Hierarchy

The protocol is based on server replication technique, i.e., primary backup. Here, the clients have been considered correct and hence they are not replicated. The Byzantine faults may occur only at the coordinator site. The cluster formation is authenticated by means of key pre-distribution method that improves the resilience and reduces the compromised communication among a number of processes. In our computation model, the basic idea of key pre-distribution scheme includes two phases, key pre-distribution phase for the purpose of secure cluster formation and pair-wise key setup phase for the purpose of authenticated agreement. In key pre-distribution phase, initially, a randomly chosen reliable process, say P_o (essentially an oracle among the processes) generates a key pool of random symmetric keys and distributes them to all other processes before entering into cluster formation round. On the reception of key, every process exchange the key information to establish a unique pair-wise key setup named as an authentication key. The process P_o also uses a secure broadcast key to encrypt its message, which is updated periodically, known as a communication key. The process P_o is used as oracle only for a secure and fair cluster formation not for the agreement purpose. In this way, we reduce the dependency on the oracle. Our cluster formation protocol follows a *leader-first* approach [14] where, first a leader is selected followed by the formation of clusters with backup replicas. Next, the agreement protocol is started for a transaction whenever a commit request is received by a primary process, say C_P, from among the ring of cluster heads. The C_P propagates the client's decision to next cluster head along with its cluster members. The process is repeated by every cluster head, C_H. After finite rounds of authenticated communication, a final decision is available with C_P who is responsible to forward the outcome of the transaction to the client.

2.1. Design Goals

Our primary goal has been to reach a safe agreement among the processes. Nevertheless, the formation of a hierarchical and secure cluster has been introduced because it has the following advantages:

- It is unlikely for a benign process to safely cross the multi-level authentication steps before participating in agreement process.
- It is difficult for a malicious process to join many clusters in order to mislead other processes.
- It is difficult for a faulty process to enter into the ring of cluster heads and initiate agreement before authenticating its presence.

Note that a comprehensive solution to the agreement problem, in the presence of arbitrary processes, must successfully handle the issues like secure cluster formation, authenticated cluster head ring formation and authenticated agreement. In this paper, we begin with the secure cluster formation which serves as the start point towards a desired solution to reach safe agreement. Moreover, the hierarchical approach is inherently scalable.

3. Terms and Assumptions

The proposed scheme includes some assumptions and we have introduced some new terms that are explained as follows:

3.1. Threat Model

We assume that an adversary may try to enter into the cluster formation. Furthermore, a process may be fail-stop *i.e.*, crash. A Byzantine cluster replica may change the decision forwarded by its cluster head. Also, a faulty cluster head may propagate wrong information about the outcome of the transaction to distract the other cluster heads as well as its own cluster members.

3.2. Authentication Key (K_{Auth})

The authentication key is a unique pair-wise shared key between a process and the oracle process P_O. The function of authentication key is two-fold. Formerly, during initial authentication phase *i.e.*, after key pre-deployment phase, each process authenticates itself with the process P_O using its corresponding authentication key. Later, after the selection of cluster heads, all cluster heads will reportedly authenticate themselves with process P_O accessing P_O's updated communication key. The authentication key can be used once in the lifetime of a process and cluster head, and the key is deleted after authentication from the process P_O.

3.3. Communication Key (K_{Com})

The process P_O uses a communication key to provide a secure statement with the processes. All messages transmitted by the process P_O are encrypted by the communication key. Further, the key is updated periodically in order to maintain a longer integrity into the system.

3.4. Cluster Key (K_{CH})

Once the cluster heads are elected, each cluster head will generate a cluster member key to establish a confidential communication with its cluster members.

3.5. Key Pool (P)

A key pool is a large pool of random symmetric keys which is generated and stored at process P_O. The process P_O assigns a particular number of keys to each process, where keys are randomly chosen from the key pool without replacement. Say, S represents the number of keys in the pool.

A summary of notations and symbols used in the proposed mechanism are listed in Table 1.

4. The Approach

This section describes the authenticated cluster formation of the processes and an authenticated ring formation of the cluster heads.

4.1. The Authentication Phase

The authentication phase is subdivided into two-phases, namely, key pre-deployment phase and an initial authentication phase. This phase is responsible to authenticate all the processes prior to the formation of cluster and selection of cluster heads. It helps to achieve a trustworthy replicated system with reliable primary as well as reliable backup processes.

4.2. Key Pre-deployment Phase

A finite number of processes enter into the transaction processing system with unique identifiers (N_{ID}). First, a random process is chosen as an oracle, P_O, which generates a large pool P of S symmetric keys. Also, every process id is recognized by process P_O. Then, the key is drawn from the pool randomly and given to different processes without replacement. Each process, now, avails a unique pair-wise key (K_{Auth}) shared with the process P_O. Following this, every process undergoes an initial authentication phase. An additional task of process P_O is to allocate a time slot, containing a finite sequence of equal time outs, to every process. The objective of the task can be seen in cluster formation phase.

Table 1: Notations and Symbol Definition

Notation	Definition
P_O	Oracle process
C_H	Cluster head
C_M	Cluster member
P	A pool of keys
S	The number of keys in P
P_{ID}	Identity of the process
K_{Auth}	Authentication key
$K_{Com(O)}$	Communication key of P_O
$K_{Upd(O)}$	Updated communication key of P_O
$K_{CH(i)}$	Cluster key generated by cluster head i
CH_P	Primary process in cluster head's ring
$E_{K(m)}$	An encryption of message m with key K
$D_{K(m)}$	A decryption of message m with key K
$mac_{K(m)}$	MAC calculated with key K
$nonce(t_s)$	A (time-variant) random number string
R_C	Client's decision request (commit/abort)
$D_{CR(i)}$	Decision of the cluster replica i
$D_{CH(i)}$	Decision of cluster head process i
$AM(o)$	Final agreement message with final outcome o.

4.3. Initial Authentication Phase

During this phase, all processes are authenticated. A process is authenticated by using the latest communication key ($K_{Cur(O)}$). In order to get the latest communication key, a process P_i sends a request to the process P_O. The request consists of process identity P_{ID}, [1]nonce(t_s) and MAC, where MAC is calculated by using the authentication key $K_{Auth(i)}$ of that process as shown in the following figure 2:

$P_i \Rightarrow P_O$	:	$Msg\ (P_{ID(i)},\ \text{nonce}(t_s), \text{MAC}_{KAuth(i)}\ (P_{ID(i)} \| \text{nonce}(t_s)))$
$P_O \Rightarrow P_i$	:	$E_{KAuth(i)}(K_{Com(O)})$
P_i	:	$D_{KAuth(i)}(K_{Com(O)})$

Fig. 2: Authentication of Process Pi

[1] Nonce, a cryptographic cookie, is a random *number used once*. We apply nonce with time-stamp t_s to ensure its one-time usage.

The process P_O authenticates the process P_i by verifying the MAC using the authentication key of that process associated with its process identity, P_{ID}. The process P_O replies to the process P_i with its current communication key, $K_{Com(O)}$ which is encrypted with $K_{Auth(i)}$ of the requesting process and mark the $K_{Auth(i)}$ of that process as *utilized* in its database. Afterwards, the process P_i receives the message from P_O and uses its $K_{Auth(i)}$ to decrypt the communication key.

Although, the process P_i has also used the authentication key, it will maintain the key for a second level of authentication with the process P_O's updated communication key, in case, it is selected as a cluster head. Otherwise, the process P_i would also be required to delete $K_{Auth(i)}$ from its memory before participating in the agreement round.

5. Cluster Formation Phase

After authentication and receiving communication key K_{Com} from the process P_O, this phase considers only the non-faulty processes as participants whereas all the faulty ones are disallowed to participate in the transaction processing. The cluster formation phase is subdivided into two phases, namely, an initial cluster organization phase and ring formation phase. Both the phases are interleaved to form a two-layer hierarchy.

5.1. Initial Cluster Organization Phase

All the processes enter into the cluster organization phase. Figure 3 shows the pseudocode of the algorithm which is executed on every process in the system. The steps of the algorithm are as follows:

Cluster Organization Phase

```
/* Initial State*/
χ : Maximum limit on processes in one cluster.
t_slot : time slot with sequences of fixed timeouts.
N : Network Size, i.e., total number of nodes
Max(N_ID): Function to find a process with maximum identity
m : No. of cluster members to be accommodate in a cluster

procedure: InitCluster()
begin
   for (P = 1 to p = maximum processes available)
      begin
      t_s1= get time;                //first time-out from the timeslot
      broadcast (N_ID).t_s1 to p-1;
      receive (N_ID) from p-1;
      call Max(N_ID);
      Max(p) ⇒ C_H(1);               //first cluster head
      call ClusterSize();
   end for;
end;
procedure: ClusterSize()
At C_H(1):
begin
   χ = 0.2N;
   If (m ≤ χ)
      goto InitCluster();
      replace t_s1 to t_s2;          //next(second) time-out from the timeslot
    else
      select χ processes;
      broadcast C_H message;
   endIf;
end;
```

Fig. 3: Pseudocode for Cluster Organization Phase

i. Every process undergoes 'N_{ID}-exchange' round where each process will exchange its identity to every other process appended with a timeout from the timeslot (given by P_O) to specify the time period in order to receive their identities.

ii. After receiving N_{ID}'s from other processes within the specified slot, each process will call a $Max(N_{ID})$ function to declare the maximum identity process to be the first cluster head ($C_{H(1)}$) that is head of the first cluster.

iii. Now, the selected cluster head will set a threshold say, $\chi = 0.2N$, on the number of members to accommodate into the cluster, where N is the total number of processes in the system. Accordingly, there will be two cases:

Case I: If the members are less than or equal to the threshold value, the cluster head will reinitiate the algorithm from step 2 with a next timeout period from the timeslot.

Case II: Otherwise, it will select the members randomly in accordance with the χ and broadcast a cluster member message to its cluster processes. Following this, the remaining process' undergoes the same rounds of execution.

iv. The steps (i) through (iii) are repeated until every non-faulty process becomes either a cluster member or a cluster head.

The timeout is, essentially, incorporated in order to evacuate the fail-silent and crashed processes. In the end, all the cluster heads will generate and distribute a cluster key (K_{CH}) to authenticate its members. Figure 4 shows the message transferred between the cluster heads C_H and cluster member C_M.

$C_H \Rightarrow P_i$	:	K_{CH}, nonce, <*join-cluster*>, MAC($K_{Upd(O)}$(nonce))
$P_i \Rightarrow C_H$	:	N_{ID}, <*approve*>, nonce, MAC(<*approve*> ‖ nonce)

Fig. 4: Authenticated Message Transfer between CH and CM

First, a cluster head broadcast a *join-cluster* message with a cluster key, K_{CH}. The communication between cluster member and cluster head is protected using two keys: updated $K_{Upd(O)}$ and $K_{CH(i)}$, where $K_{Upd(O)}$ is the updated communication key of the oracle process P_O. This key is to be available with the all the processes before entering into cluster organization. Next, the process chooses the cluster head C_H with whom it shares a key and has received a *join-cluster* message to join the cluster. Then, it sends an *approve* message to the selected C_H; the approval request is protected by the MAC, using $K_{CH(i)}$ and include the nonce (to prevent replay attacks) from C_H. It sends the approval message with its N_{ID} (so that the receiving C_H knows which key to use to verify the MAC). Eventually, every non-faulty process becomes a participant in some cluster with its respective cluster heads.

5.2. The Ring Formation Phase

Once the organization of authenticated clusters with their respective leaders is complete, this phase organizes all the cluster heads in a logical ring. Afterwards, there is another level of authentication is performed among the cluster heads. It executes the following sequence of steps:

1. The first cluster head ($C_{H(1)}$) will intimate its presence to the succeeding, *i.e.*, the next cluster head with its id and a time-stamped nonce protected with MAC.
2. The successor cluster head *i.e.*, $C_{H(2)}$ will install its identity in the [2]migrating array in the descending order of the identities along with the protected MAC message from the originator.
3. The same operation is repeated at every cluster head on the reception of migrating array.
4. Finally, the cluster head $C_{H(1)}$ will receive an authenticated and sorted list of all clusters heads contained in the migrating array and declares the maximum identity process(*i.e.*, the process at first position in the array) to be primary cluster head, CH_P.

The CH_P is responsible to receive the client's decision request and propagate it to all other cluster heads in the ring that further multicasts the same request among their respective cluster replicas to reach an agreement. In case, any cluster head maliciously introduces a send-omission failure *i.e.*, it excludes its id while forwarding the information to the adjoining cluster head, it is not included in the current agreement round. However, such possibility is infrequent after multi-rounds of authentication for the selection of cluster heads. The state diagram for ring formation is shown in the following figure 5.

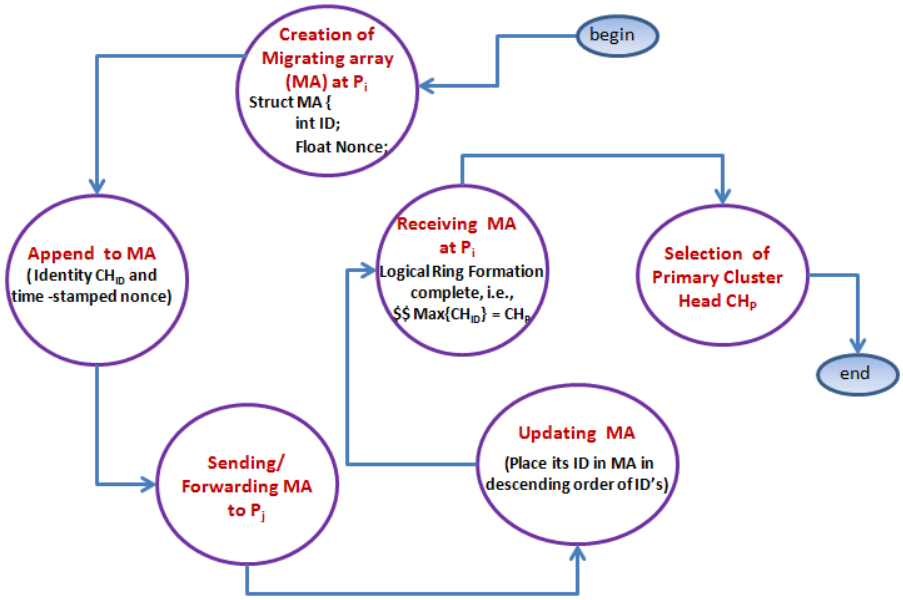

Fig. 5: State Diagram for Ring Formation Phase

6. Authenticated Agreement Protocol

After the organization of the processes in a two-layer hierarchy, the agreement begins for the requested transaction. The main data structures and messages used in the agreement process are as follows:

[2] Migrating array is an array which is circulated among all the cluster heads in the ring where the processes append their identity and a MAC protected message in order to prove their authenticity as well liveness in the system.

i. r_i: serial number of the current round in which a cluster head C_H and a cluster replica C_R are participating.

ii. req_c: the decision value v i.e., commit/abort from the client to the primary cluster head, CH_P.

iii. fg_i: the flag indicating whether the cluster head $C_{H(i)}$ has taken the decision. Initially, the value is set to *false*.

iv. ts_i: the timeslot indicating time-out for a particular communication round.

v. ph_i : phase number of the current phase in which process P_i is participating.

vi. *PROP*(r, req_c): the proposed value sent from the primary cluster head CH_P to all other cluster heads C_H which further propagates among their respective cluster replicas C_R. req_c is the current request kept by the primary cluster head. For each round r, the CH_P tries to impose req_c as the final decision by sending the proposed values.

vii. *DEC*(*req*): the message sent by a cluster replica C_R to its respective cluster head C_H.

The proposed protocol consists of two procedures executing the tasks, namely, *achieving consensus* and *broadcast reliable decision*. The pseudocode of both the procedures is given in figure 6.

-------Proc 1: Achieving Consensus------

//The code is executed by every non-faulty cluster head C_H as well as cluster replicas C_R
CH_P is the primary cluster head;
C_H represents rest of the cluster heads;
BEGIN:
$r_i \leftarrow 0$; $req_c \leftarrow v_i$; $ts_i \leftarrow 0$; $fg_i \leftarrow$ *false*;
while($fg_i \neq$ true){
$r_i \leftarrow r_i + 1$; $ph_i \leftarrow 1$;
----------Phase 1; round r_i: from CH_P to all C_H's----------
At CH_P:
receive req_c from client;
repeat at every C_H:
broadcast *PROP*(r_i, v) to next C_H with ts_i;
broadcast *PROP*(r_i, v) to C_R with ts_i;
wait until *PROP*(r_i, v) is received or time-out;
if(*PROP*(r_i, v) is received and $v \neq \perp$) {$req_c \leftarrow v_i$; $ts_i \leftarrow 1$;}
----------Phase 2; round r_i: from all C_R to respective C_H's----------
$ph_i \leftarrow 2$;
At C_R:
Exchange *PROP*(r_i, v) to C_{R-1};
Call *MAJ*(); //Declares the majority value of the decision
DEC(req_i) = v(*MAJ*)
DEC(req_i) = $E_{KCH}(v(MAJ)$ //Encrypts decision using cluster key K_{CH}
Broadcast to C_H;
At C_H:
$D_{KCR}(v(MAJ)$; //Decrypts decision using corresponding replica key K_{CR}
If (*DEC*(req_i) is received from 0.5χ) within ts_i
Decides final value //writes the decision on the migrating array in the ring in order to send to the CH_P
endIf;

-------Proc 2: Reliable Broadcast------

Upon reception of *DEC*(req_i) from every C_H to CH_P:
$fg_i \leftarrow$ *true*; send *AM*(o) to the requesting client;
END

Fig.6: Authenticated Agreement

Procedure 1 consists of two phases. At the beginning of round r, the primary cluster head CH_P receives a client's decision value (commit/abort) about the current transaction. On reception of the decision, the primary cluster head CH_P sends $PROP(r_i, req_c)$ to its succeeding cluster head as well to its cluster replicas with a time-stamp ts_i. The same action is executed by every cluster head. Every cluster head and cluster replica waits for the proposed decision and enters into the second phase after receiving the values from the corresponding leader processes. However, in case, if sender fails to broadcast the message, it is declared faulty. Next, in the second phase, the cluster replicas will enter a message exchange round to know each other's decision value. Afterwards, every replica will call a $MAJ(\text{req}_i)$ function to decide on a single outcome of the transaction. The final decision $DEC(req_i)$, encrypted with the cluster key K_{CH} is forwarded to the cluster head where the message received within the stipulated time (as mentioned in the timeslot) is considered as a bound to reach agreement in order to avoid the fail-silent replicas. Also, a faulty replica may take two arbitrary actions: (1) it may send the correct decision value encrypted with its own generated key so as to instill ambiguity as well as latency; (2) It may send the wrong decision value encrypted with the allotted unique pair-wise cluster key K_{CH}. Therefore, the cluster head verifies the decision message by using the C_{Ri}'s corresponding cluster key available in its database. In case, any cluster replica fails the verification, it is evacuated from decision making round and if the cluster head receives more than 0.5χ matching replies from the remaining non-faulty replicas, where χ is the number of cluster replicas, it decides on a single outcome. Further, following the same procedure, the decision is propagated in the ring by every cluster head. After multi-level authentication, assuming all cluster heads to be fault-free during the final agreement round, the primary cluster head CH_P receives the agreed decision value and takes action accordingly.

Task 2 is the simple broadcast mechanism. When primary cluster head CH_P decides the outcome of the transaction, it simply forwards the agreement message AM exhibiting the final outcome to the requesting client. This marks the end of the authenticated agreement among the non-faulty processes.

7. Security Analysis

In the following, we will discuss how the proposed protocol achieves the two important security goals identified in section 4 and 5.

Security goal 1: This security goal requires that only legitimate leaders will form a cluster and selects the non-faulty members to be part of the secure cluster. In other words we have to ensure that the unauthorized nodes that don't have the leader membership cannot join as a cluster replica and cannot impersonate any benign cluster member. This property is achieved by using key pre-distribution scheme by an oracle process P_O. The cryptographic communication key K_{CH} to establish a secure communication makes it impossible, for an arbitrary process, to join the cluster organization.

We further explain that the faulty process from any other cluster cannot forge the communication between the cluster replicas and their respective cluster head. This is ensured because the announcement between a cluster replica C_R and C_H is encrypted by a unique pair-wise cluster key generated by the cluster head for secure communication. Therefore, the adversary would not have any way to make its forged commitments accepted by any other cluster member.

Security goal 2: Note that a compromised process can always behave normally and get elected as the cluster head at some point during the cluster organization phase.

Although, a fail-silent or crashed process may be diagnosed with an ease, there are no effective ways to identify those "passive" malicious processes since there are no evidences of them being malicious. As a result, the second security goal focuses on making sure that a cluster head, if chosen, is not straightaway allowed to control the cluster activities without further authentication. In our protocol, the cluster heads too undergo a second-level authentication, as described in section (4) to prove their loyalty with the protocol, only after which they are eligible to enter into the agreement round. Thus, the protocol ensures a safe communication to reach consensus on a requested value.

8. Correctness Proof

Theorem: The protocol satisfies the agreement and termination requirements.
Proof: We prove the above theorems with the help of following lemmas.

Lemma 1: *No correct cluster head will be blocked forever.*
Argument: The proof is by contradiction. Assume that some correct cluster head, say CH_i is blocked in a round, say, *R.* A cluster head can only be blocked in a wait statement. Now, there are two possibilities:

Case 1: The blocked cluster head CH_i is a part of the ring and it the primary cluster head *i.e.*, CH_P. However, it cannot be blocked as it has already received the proposed value from the client and forwarded the same to the next successor cluster head in the ring. Therefore, CH_i is not blocked, which is a contradiction.

Case 2: The cluster head CH_i is not the primary cluster head. Hence, it will eventually receive the proposed value from the non-faulty primary cluster head CH_P that can never be blocked as prove in case 1. However, CH_P may crash. Under this situation, CH_i will eventually suspect it and a new CH_P is chosen with the same procedure as described in the ring formation phase. Therefore, a CH_i cannot be blocked. This is a contradiction.

Therefore, from case 1 and case 2, it is obvious that lemma 1 holds. □

Lemma 2: *If a cluster head* C_H *is correct* (*non-faulty*), *it eventually decides*.
Proof: The proof is by contradiction. Assume a correct cluster head CH_i never decides. According to our protocol, this situation will be eventually detected by some neighbor of CH_i in the ring. Now, this can happen only when CH_i dose not forward the desired information to its neighbor within stipulated time. This indicates that either CH_i is faulty or CH_i has not received the decision value from its predecessor. However, in the later case, the predecessor process will be detected incorrect by CH_i itself and this information will be forwarded further in the ring. Therefore, a correct process will always forward its decision value and *id* of incorrect process.

Therefore, lemma 2 holds. □

Lemma 3: *No two correct cluster heads decide differently in any round r.*

Proof: The proof is by contradiction. Assume that any two correct cluster heads say, CH_1 and CH_2 decides differently. If a cluster head CH_2 decides a value, then this value must have been decided and forwarded by its predecessor cluster head CH_1 in the ring. According to the accuracy and completeness of the underlying primary cluster head, *i.e.*, CH_P, there is a time *t* after which all the non-faulty cluster heads C_H are never suspected by any non-faulty cluster head and all the crashed cluster heads are permanently suspected by every non-faulty cluster head. Since, there is at least one correct cluster head i.e., CH_P in the ring, so every non-faulty

cluster head is associated with the non-faulty cluster primary CH_P (implemented with the help of migrating array). Let r be any round of communication among the ring members coordinated by CH_P and started after t. So, after receiving the proposed decision value from the CH_P, it is forwarded to the successor cluster head, which decides the same value, in case it is a correct process and forward the decision to next. Therefore, any two correct cluster heads will eventually agree on the same decision value in the round r. This contradicts the assumption that two correct cluster heads decides differently.

Therefore, lemma 3 holds. □

9. Simulation Results

We used BFTSim [15] to analyze the performance of our protocol. In one of our previous works [16], we have analyzed the performance of reactive protocols with the proactive protocols to reach consensus on a particular transaction request. The following figure 7 exhibits the average response time *i.e.*, latency incurred to reach agreement among different number of processes. It is noteworthy that, using the property of authentication via secure cluster formation, the proposed scheme drastically reduces the overall latency overhead of the protocol as the communication rounds among the processes are minimized. The lesser the communication rounds, the message overhead is reduced accordingly.

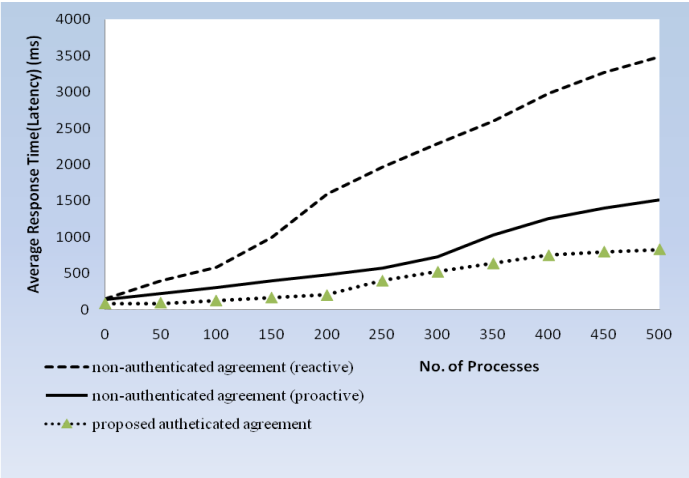

Fig 7. Latency Measurement for Different Approaches

In the next experiment, we evaluated our protocol to show the effect of cluster size *i.e.*, the average number of group members in a cluster on the communication overhead. If the cluster size is kept constant, the communication overhead also remains confined to some average value. It may be noted that the nodes being static, it is practically feasible to maintain constant cluster size because the failure rate is usually very low. However, in case of random sized clusters, the communication overhead is also random with an increasing trend as shown in following figure 8.

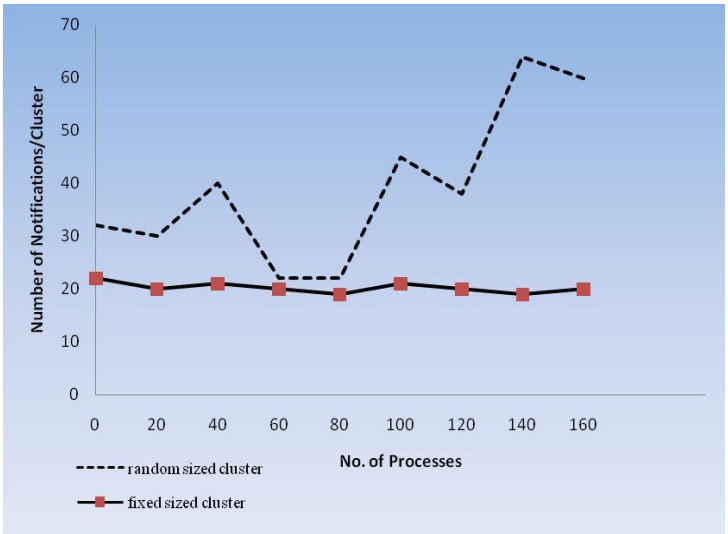

Fig 8. Cluster Size vs Communication Overhead

9. Conclusion

Our scheme provides a hierarchical solution for Byzantine resilient agreement using secure cluster formation. Unlike, many protocols on hierarchy-based consensus for mobile and wireless environments, the proposed approach is proactive for reaching agreement among finite number of processes. We impose a hierarchical-structure in order to minimize the overall communication cost and reduce the latency of the transaction processing, from request initiation to final decision in order to speed up the agreement among the processes. The approach is *proactive* and provably more efficient, in a sense, that it detects the faulty processes in advance and constrains them from participating in the agreement process at an early stage. In comparison, the existing commit protocols do not imply the notion of proactive fault tolerance as they rely on the specification of the faults to circumvent them which makes them *reactive*. Furthermore, the proposed authenticated cluster formation technique will enhance the reliability without violating the safety of the protocol by segregating out the faulty processes. Furthermore, the approach being hierarchical, it is inherently scalable.

References

[1] G. Coulouris, J. Dollimore, and T. Kindberg, Distrbuted Systems: Concepts and Design (third edition), *Addison-Wesley*, 2001.

[2] N. A. Lynch, Distributed Algorithms, *Morgan Kaufmann*, 1996.

[3] M. Hurfin, and M. Raynal, A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector, *Distributed Computing*, Sep 1999.

[4] R. Guerraoui, M. Hurfin, et. al., Consensus in Asynchronous Distributed Systems: A Concise Guided Tour, *LNCS 1752*, 2000.

[5] R. Guerraoui, and A. Schiper, Consensus: The Big Misunderstanding, *the sixth IEEE workshop on Future Trends of Distributed Computing Systems*, 1997.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson, Impossibility of distributed consensus with one faulty process. *J.ACM*, 32(2):374-382, Apr. 1985.

[7] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 225-267, 1996.

[8] M. Castro, and B. Liskov, Practical Byzantine Fault Tolerance and Proactive Recovery. In: *ACM Transactions on Computing Systems*, vol. 20, pp. 398-461. DOI: 10.1145/571637.571640, (2002).

[9] W. Zhao, A byzantine fault tolerant distributed commit protocol, *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp. 37-44, Sep, 2007.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, Zyzzyva: Speculative Byzantine Fault Tolerance, *In ACM Proceedings of twenty-first Symposium on Operating Systems and Principles*, vol. 41, no. 6, pp. 45-48, Oct, (2007).

[11] H. Chan, et.al., Random Key Predistribution Schemes for Sensor Networks, *IEEE Symposium on Security and Privacy*, pp. 197-213, DOI: 10.1109/SECPRI.2003.1199337, May, (2003).

[12] H. Rifa-Pous, and J. Herrera-Joancomarti, A Fair and Secure Cluster Formation Process for Ad Hoc Networks, *Wireless Pers Commun*, vol. 56, pp. 625-636, (2011).

[13] Y. Cheng, and D.P. Agrawal, Efficient Pairwise Key Establishment and Management in Static Wireless Sensor Networks, *IEEE International Conference on Mobile Adhoc and Sensor Systems*, 7 pp.-550, Dec, (2005).

[14] Q. Dong, and D. Liu, Resilient Cluster Leader Election for Wireless Sensor Networks, *IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON*, June, (2009).

[15] BFTSim, http://bftsim.mpi-sws.org/, April 2008.

[16] P. Saini, A.K. Singh, A preemptive view change for fault tolerant agreement using single message propagation. *International Conference on Advances in Information Technology and Mobile Communication* (AIM), LNCS-CCIS 147, pp 507-512. Springer, Heidelberg, Apr, (2011).

Authors

Poonam Saini received B.Tech degree in Information Technology from Kurukshetra University, Kurukshetra, India in 2003. She received M.Tech degree in the area of Software Engineering from the same university in 2006. She is pursuing PhD from National Institute of Technology and has submitted final PhD synopsis in the area of Fault Tolerant Distributed Computing. Currently, she is Assistant Professor in the department of Computer Science and Engineering, NITTTR, Chandigarh, India. Her present research interest is Fault Tolerant Distributed Computing.

Awadhesh Kumar Singh received B. E. degree in Computer Science & Engineering from Gorakhpur University, Gorakhpur, India in 1988. He received M.E. and PhD (Engg) in the same area from Jadavpur University, Kolkata, India. Currently, he is Associate Professor in the Department of Computer Engineering, National Institute of Technology, Kurukshetra, India. His present research interest is mobile distributed computing systems.