Dynamic Distributed Computing with Byzantine Failures

Cheng Wang
LPD, I&C, EPFL

Abstract—Byzantine failure tolerance (BFT) is one of the most fundamental problem in distributed computing and has been researched extensively. However, due to the famous impossible result of FLP, there is no perfect way to solve it. Modern BFT protocols are usually proposed with practical assumptions such as partially synchrony, encrypted messages. Yet, designing dynamic BFT system with large number of nodes remains challenge. In this write-up, we will explore and review the techniques in the core of modern BFT protocols, and discuss the recent new approach by combining clustering and random walk to obtain reliable and efficient dynamic BFT system.

Index Terms—thesis proposal, candidacy exam write-up, EDIC, EPFL

I. INTRODUCTION

In distributed computing, one of the ultimate goal is to ensure that the system performs reliable and efficient. Due to the significant characteristics of distributed system such as concurrency of components, lack of global clock, and failures of component, it is hard to achieve reliability and efficiency perfectly. Since the appearance of the field of distributed computing, researchers have proposed plenty of important concepts, ideas and protocols to cope with the difficulty. In this write-up, we will describe some of them like the celebrated Paxos algorithm, the practical Byzantine fault tolerance, and highly dynamic distributed computing, and then conclude it with a research proposal.

In distributed system research different models with different assumptions are exploited according to the scenarios considered. But typically, the general frameworks presented are almost the same. In the rest of this section, we will browser the popular general model of distributed computing to provide a preliminary for next sections.

A. Message-passing System

In a message-passing distributed system, processes communicate by sending messages over communication channels which is called the topology (or network) of the system (usually the topology is connected). Depends on research interests and specification, the topology can be directed or indirected, complete or incomplete, static or dynamic. In this write-up, only complete topology is considered, which means every two processes are connected directly. An algorithm (or protocol) for a message-passing distributed system refers to the set of local programs designed for all the processes inside. The local programs or algorithms, which can be deterministic or random, defines the way for the processes to perform local computation and to send messages to and receive messages from its neighbors in the given network.

More formally, a system consists of \( n \) processes \( p_0, p_1, \ldots, p_{n-1} \), and \( i \) is the unique identifier of \( p_i \). Each process \( p_i \) is modeled as a state machine (usually infinite in theoretic discussion) with local state \( \text{State}_i \). \( \text{State}_i \) contains a initial state \( S_{i,0} \), input values, output registers, and a transition function \( f_i \), and other changeable memory for storing information. The transition function \( f_i \) takes the \( \text{State}_i \) as input and change it to a new value, which is referred as local algorithm informally. It also produces the set of messages to be sent to the neighbors of \( p_i \). Another component of the system is the message system. The message system maintains a multiset, called the message buffer, of messages that have been sent but not yet delivered. Note that the function of message system to delivery message is independent of the sender and receiver of the message. Therefore the messages previously sent by \( p_i \) can not influence \( p_i \)'s current step. There are two kinds of message type: one is oral message which is very normal in pure theoretic research, and the other is unforgeable signed message which attracts a lot of attention in practice.
A *configuration* (or *view* sometimes) of the system consists of the internal state of each processes as well as the contents of the message buffer. And the initial configuration is the configuration in which all the processes are in the initial states and the message buffer is empty. The execution of a system is modeled as a sequence of configurations alternating with events. Basically, this sequence must satisfy two kinds of conditions, i.e. safety and liveness conditions. These two conditions are specified by the problem to be solved and we will define them for our. Informally, a safety condition stipulates that something bad will not happen and a liveness condition stipulates that something good will happen.

### B. Synchrony and Asynchrony

According to the communication protocol, system can be synchronous and asynchronous. In the synchronous model processes proceed in lockstep, i.e. the execution is partitioned into rounds and in each round, every process can send messages to neighbors, and the messages are delivered to and received by destination processes. This model, though not achievable in practice, is very convenient for designing algorithms because an algorithm need to to consider the uncertainty of message delivery. Yet, the asynchronous model is more practical and captures the characters of nowadays networks like internet. A system is said to be asynchronous if there is no fixed upper bound on how long it takes to be delivered or how fast the processes are. Because of this indefinite property, designing an asynchronous algorithm is always difficult but usually attractive and desirable.

### C. Failure Model

It is convenient to build the failure model as an adversary model, and define the ability of the adversary to describe the faulty behavior of the system. In most research, the adversary controls a set of processes to work incorrectly according to the failure type. In this write-up, we only discuss the following failures:

- **Crash failure**: the processes controlled by the adversary might crash in a particular time and send no message after that. Most practical protocols deal with crash failure which is proved easier to handle.

- **Byzantine failure**: the processes controlled by the adversary might send arbitrarily messages. Byzantine failure is considered as the toughest failure system can experience. It has been extensively researched since the seminal paper [6].

A process always behavior following the protocol is called *correct*.

### D. Consensus and State Machine Replication

A fundamental problem in distributed computing is to achieve reliability in the presence of failures. This often requires processes to agree on computation. The consensus problem is a very common model to capture this by requiring agreement among a number of processes for a single value.

Consider a system in which each process has an input value (usually called proposal value). A *consensus* protocol tolerating failures should satisfy the following properties.

**Termination**: Every correct process decides some value.

**Validity**: The decided value must be proposed by some process.

**Agreement**: Every correct process must decide the same value.

Applications of consensus include leader selection, state machine replication and atomic broadcasts etc. Consensus protocols are the basis for the state machine approach to distributed computing, as suggested by Leslie Lamport [4]. In this write-up, we will explore state machine replication.

State machine replication is a general method introduced by Leslie Lamport in the 1970 as a general framework for implementing fault tolerant servers by replicating servers and coordinating client interactions with servers. The process inside such protocol is usually called *replica*. A state machine replication based approach involves replicating a state machine on multiple instances of replica. Each replica begins with the same initial state. When the system receives a client request each of the replicas should process the same request in the same order and update their local states following the protocol. So generally speaking it tries to emulate a single highly available process.

### II. Paxos Algorithm

Paxos is a family of protocols for solving consensus in an asynchronous distributed system which might contains failures, first published by Lamport in 1989. The Paxos protocols includes a spectrum of trade-offs between message delays, number of message sent, and types of failures, trying to make the protocol as practical and simple as possible. Although the famous result by FLP [2] states there is no deterministic fault-tolerant consensus protocol in an asynchronous network, Paxos guarantees safety, as well as liveness in most practical case. In this section, we describe the Basic Paxos [5]. Figure 1 depicts the message flow of Basic Paxos with a successful first round.

Paxos presents the action of the processes by their roles in the protocol: proposers, acceptors, learners. The functions of roles are the same as suggested by their names. In an implementation, one process may act as more than one roles. This enables Paxos algorithm applicable to many scenarios not only the consensus problem. The protocol of Basic Paxos contains two phases as follows.

**Phase 1a**: A proposer creates a proposal identified with a number $n$ and sends a *prepare* request containing this proposal to a majority of acceptors. The number $n$ should be greater than any previous proposal number used by this proposer. To prevent confusion, different numbers for different proposals are required.

**Phase 1b**: If an acceptor receives a *prepare* request with number $n$ greater than that of any prepare request to which it has already responded, then it responds to the request with a promise to ignore all future proposals numbered less than $n$ and
Theorem 1: The famous result of Fischer, Lynch, and Patterson [2] is the following: If a value is chosen at proposal number $n$, then any value sent out by the proposal in phase 2 of any later proposal numbers must be $v$.

Proof: The proof is by induction. Suppose $m > n$, and for all $k (n \leq k < m)$ the value sent out in phase 2 of proposal number $k$ is $v$. So there must be a subset $S$ of acceptors (more than half of total) whose highest-numbered accepted values are $v$, and the other acceptors have accepted no values.

When a learner finds out that a proposal has been accepted by a majority of acceptors, we say the proposal value has been chosen. Though the algorithm presented above seems complex, it can be deduced from consensus condition and the functions of roles. Here we prove a theorem to show the safety of Basic Paxos.

Phase 2a: If the proposer receives a response to its prepare requests (numbered $n$) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered $n$ with a value $v$, where $v$ is the value of the highest-numbered accepted proposal among the responses, or if any value if the responses reported no proposals.

Phase 2b: If an acceptor receives an accept request for a proposal numbered $n$, it accepts the proposal unless it has already responded to a prepare request having a number greater than $n$. Usually it should send an accepted message to the proposal and every learner.

When a learner finds out that a proposal has been accepted by a majority of acceptors, we say the proposal value has been chosen. Though the algorithm presented above seems complex, it can be deduced from consensus condition and the functions of roles. Here we prove a theorem to show the safety of Basic Paxos.

III. PRACTICAL BYZANTINE FAULT TOLERANCE

This section describes the first replication algorithm proposed by Castro and Liskov [1] that is able to tolerate Byzantine faults in practical asynchronous system. Indeed, this algorithm can be viewed as a variation of Paxos protocol by using a combination of primary-backup mechanism. Here we discuss about state machine replication, so we use notation like replica (for process) following the convenience.

We assume an asynchronous distributed system with Byzantine failure model. Also unforgeable signed message is assumed because in practice one can encrypt messages. Suppose $R$ is the set of replicas of the system, $f$ is the maximum number of replica that may be faulty. In order to withstand $f$ failures, $3f+1$ replicas are necessary. For simplicity, we assume $|R| = 3f + 1$. But the algorithm can be easily generalized to the case with more than $3f+1$ replicas.

Every replica moves through a succession of configurations called views. In a view one replica is the primary and the others are backups. In this algorithm, primary plays the role of proposer in Paxos, and backups plays the role of accepters and learners. The primary of a view with view number $v$ is replica $p$ such that $p \equiv v \mod |R|$. The primary selects the operations to be executed. It does this by attaching a sequence number to a request and sending the number together with the request to the backups. The sequence number here is similar to the proposal number in Basic Paxos. The backups check the message sent by the primary and use timeouts to detect when it stops. They trigger view changes to elect a new primary when it seems the current one has failed.

A. Normal Case Operation

For a normal operation, a three-phase protocol is designed to broadcast requests to the replicas. The three phases are prepare, prepare, and commit. The first phase is to broadcast request, the next two phases are to ensure the request is broadcast correctly. Basically, the first two phases corresponds to the two phase of Paxos, and the third phase is for the purpose of preserving consistency in different views. Figure 2 from [1] provides an overview of the algorithm in normal case of no faults.

The state of each replica includes a message log containing messages the replica has accepted or sent, and an integer denoting the replica’s current view. It is possible to truncate the log for optimization (see [1]).
When the primary $p$ receives a request from a client, it attaches a sequence number $n$ to it and then broadcasts a pre-prepare message $m$ with number $n$ to the backups and inserts this message into log.

A backup $i$ accepts the pre-prepare message provided the message is corrected and it has not accepted a pre-prepare for view $v$ and sequence number $n$. If $i$ accepts the pre-prepare message, it logs the request and enters the prepare phase by broadcasting a prepare message with sequence number, view number and request. The prepare message states that the backup agreed to select sequence number $n$ to the request.

Each replica collects message until it got $2f$ matching pre-prepare message for sequence number $n$, view number $v$, and message $m$. In such case, we say that the replica is prepared for the request. The requirements guarantees that it is impossible to be prepared for the same view and same sequence number but different requests. The reason is quiet simple. If two different requests are prepared for same view and same sequence number, then by $2(2f+1) - f > |R|$ there must be one non-faulty replica which has sent prepare message for both requests, which is impossible. Therefore this ensures that replicas agree on a total order for requests in the same view. But it may happens that replicas is prepared for the same sequence number and different requests in different views. In order to solve this, another phase is introduced.

When a replica is prepared for a request, it broadcasts a commit message saying that it has been prepared for the request and adds the message to its log. Then each replica collects messages until it has received $2f+1$ commits requests (possibly including its own) for which we say the replica is committed-local for the request. Each replica $i$ executes the request $m$ when $m$ is committed and the replica has executed all requests with lower sequence numbers. So this ensures that all non-faulty replicas agree the same sequence number for the same request in the same view.

### B. View Changes

The view-change protocol ensures liveness by allowing the system to make progress when the primary fails. In order not to wait indefinitely long, a backup starts a timer when it receives a request and the timer is not running, and it stops the time when it is no longer waiting to execute the request.

The backup broadcasts a view-change message for view $v+1$ when its timer expires, and stop working in the current view $v$. The view-change message contains the highest-committed sequence number and other prepared messages (if it have). The new view will have view number $v+1$. When the new primary receives $2f$ correct view-change message for view $v+1$ from other replicas, it broadcasts a new-view message to all other replicas. The new-view message contains all the view-change messages and all prepared messages without being committed. The view-change messages attached are used as proofs of correctness by other replicas. After a replicas accepts the new-view messages, it enters into the new view, updates local state accordingly and recomputes as in the above section. By view-change protocol, it ensures same sequence number can not be assigned to different requests in different views. So PBFT always guarantee the safety property by assigning a unique sequence number to every request.

### IV. Dynamic System with Byzantine Failures

In the paper [3], Guerraoui, Huc and Kermarrec show the first time that distributed computing can be both reliable and efficient in an environment that is both highly dynamic and hostile. More specifically, it shows how to maintain clusters of size $O(\log N)$, each containing more than two thirds of corrected nodes with high probability, with a dynamic system whose size can vary polynomially with respect to its initial size. In this section we will describe the basic idea of the algorithm.

Distributed system are usually designed to emulate a single highly available process. Yet, with a large number of nodes, this method is very expensive. One way to reduce the complexity is clustering. In short, the idea of clustering is to reduce the system to several reliable processes, each corresponding to one of the clusters with smaller subset of process. Clustering actually can not reduce complexity until it is combined with random walking on expander graph. If every cluster is reliable with high probability, then random walking can provider efficiency while preserving reliability.

The algorithm can be divided into two phases. The first phase initializes the clustering graph and random expander graph with good properties. The second phase maintains such graph in a dynamic setting. Suppose the number of processes in the system varies between $\sqrt{N}$ and $N$.

#### A. Initialization Phase

**a) Network Discovery:** The algorithm begins with this to inform each node of the identifiers of all other nodes. The complexity of network discovery is very expensive. So it is better to start with a small graph (for instance, of size $\sqrt{N}$), and use join and leave operations to enlarge the graph.

**b) Clustering:** This protocol exploits Byzantine agreement protocol to divide the system. First a representative cluster of logarithmic size containing more than two thirds of honest nodes is selected. Afterwards, we use this representative cluster to randomly partition the network into clusters, each of size $k \log N$, for some constant $k$. Every section is random and use Byzantine agreement on the nodes of the representative cluster to get same decision. Random choosing ensures there is more than two thirds of honest nodes in each cluster. Then, an expander graph $G$ is initialized on top of the partition.
Finally, the representative cluster informs each node the cluster it belongs to, the other nodes in the same cluster, and the neighbor clusters. Since then, the inter-cluster communication is from all to all. A node receiving a message from another cluster considers it correct if and only if, it receives the same message from more than half (or even one third) of the nodes of this cluster. So after the initialization phase, the correctness is ensured with high probability.

B. Maintenance Phase

As the initialization phase ensures the desired properties for clusters and expander graph, in this section we describe how to maintain these under high dynamics.

In order to preserve these properties in the presence of join and leave operations, shuffling the network is very crucial. The shuffling is implemented by the algorithm denoted exchange, for which the cluster exchange its nodes with nodes chosen at random from other clusters. The involved operations in maintenance phase are as follows. All of them have a communication cost of polylog(N).

Join: When a node joins the network, it gets contact with a cluster of the overlay. This cluster chooses another cluster using random walk. The chosen cluster proceeds by inserting the node and apply exchange to its nodes for one time.

Split: This operation is started when the size of a cluster is larger than \(lk\log N\) for some fixed parameter \(l\) \((l \text{ is a constant greater than } \sqrt{2})\). The cluster is split into two, the old and the new clusters. The old cluster keeps the neighbors information in \(G\), whereas the new cluster is added into \(G\) by adding \(O(\log N)\) random edges to connect to the graph.

Leave: When a node leaves the network or when the other nodes in the same cluster detect its absence, the cluster removes the node from its view and apply the algorithm exchange. Then a cluster receiving nodes from this cluster also executes exchange for all of its nodes.

Merge: This operation is initiated when the size of a cluster \(C\) is less than \(k \log N/l\) \((l \text{ described previously})\). In such case, choose a random cluster \(C'\) at random and join the nodes of \(C\) into \(C'\).

Random walk employed in exchange ensures correctness with high probability, while the clustering technique provides efficiency. This algorithm can be leveraged to implement various applications such as broadcast, agreement, sampling in the context of highly dynamic networks. For instance, a broadcast algorithm using this algorithm would only have a message complexity of \(O(N)\).

V. Future Work

In last section, we described a dynamic distributed computing algorithm for large-scale system withstanding Byzantine failures. The algorithm is reliable and efficient in a synchronous environment. So, a natural question is whether this can be generalized to asynchronous network in theory, and whether we can design a practical algorithm for such asynchronous dynamic large network. In this section we briefly describe the challenges along with possible solutions to the problem.

In theory, we can replace the synchronous Byzantine agreement algorithm with asynchronous random Byzantine agreement algorithm. But in asynchronous case, the initialization phase might not complete in bounded time. The framework of maintenance phase can be applied to this problem by solving collisions between asynchronous operations. One challenge is the probabilistic analysis in asynchronous case which might be more complex.

In practical, the problem is not very difficult because we can exploit timeouts and unforgeable messages. We can set a bound to timeout and complete the initialization phase in bounded time, and then proceed to the maintenance phase. We believe it is possible to simplify the maintenance phase protocol and Byzantine fault tolerance with practical assumptions.

REFERENCES