Generalised Lattice Agreement

B.Tech Project Stage 1 Report

Sagar Chordia
Roll No : 09005013

under the guidance of

Prof. S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
November 2012

Abstract

Chapter 1

Introduction

Lattice agreement is a key decision problem in distributed systems. In this problem, each process starts with an input value belonging to a lattice, and must learn an output value belonging to the lattice. The goal is to ensure that each process learns a value that is greater than or equal to its input value, each learnt value is the join of some set of input values, and all learnt values form a chain.
Consider the lattice agreement problem in asynchronous, message passing systems. A wait-free algorithm for solving a single instance of lattice agreement is proposed [1]. We also study a generalization of lattice agree- ment, where each process receives a (potentially unbounded) sequence of values from an infinite lattice, and the goal is to learn a sequence of values that form a chain. As before, we wish to ensure that every learnt value is the join of some set of received values and that every received value is eventually included in a learnt value. We present a wait-free algorithm for solving generalized lattice agreement.
Finally, we show that generalized lattice agreement can be used to implement a special class of replicated state ma- chines, the main motivation for our work. State machine replication [8] is used to realize data-structures that toler- ate process failures by replicating state across multiple processes
We address fundamental questions concerning the possibility of implementing data-structures that can tolerate process failures in an asynchronous distributed set- ting, while guaranteeing correctness (linearizability with re- spect to a given sequential specification) and progress (wait- freedom). We consider a class of data-structures whose op- erations can be classified into two kinds: update operations that can modify the data-structure but do not return a value and read operations that return a value, but do not modify the data-structure. We show that if the set of all update operations satisfy certain algebraic properties, then waitfree linearizable replication is possible for the data-structure. We also show that under certain conditions waitfree linearizable replication is not possible.

This report is organized as follows. Chapter 2 describes Genearalized Lattice Agreement in detail. In Chapter 3 we will describe Application of GLA for view maintenance of database systems . In Chapter 4 we will describe system design for generalized lattice agreement. We discuss the implementation details and challenges involved. The Chapter

5 describes future work.

Chapter 2

Generalised Lattice Agreement: Algorithm

2.1 Problem Definition

Let (L, ,) be a semi-lattice with a partial order and join (least-upper-bound) . We say that two values u and v in L are comparable iff u v or v u. In lattice agreement, each of the N processes starts with an initial value from the lattice and must learn values that satisfies the following conditions:

1. **Validity** Any value learnt by a process is a join of some set of initial values that includes its own initial value.
2. **Stability** A process can learn at most one value.
3. **Consistency** . Values learnt by any two processes are comparable.
4. **Liveness** Every correct process eventually learns a value

2.2 Algorithm

Our algorithm for lattice agreement is shown in Algorithm 1. For convenience, we consider two kinds of processes, (1) proposer processes, each of which has an initial value initialValue, and learns a single output value outputValue, and (2) acceptor processes, which help proposer processes learn output values. Let Np denote the number of proposer processes and let Na denote the number of acceptor processes. In this formulation, our liveness guarantees require a majority of acceptor processes to be correct.

Proposer

Every proposer begins by proposing its value to all acceptors (see Propose in Algorithm 1). Each proposal is associated with a proposal number that is unique to each proposer.

Acceptor

An acceptor may accept or reject a proposed value. Every acceptor ensures that all values that it accepts form a chain in the lattice. It does this by tracking the largest value it has accepted so far in the variable acceptedValue. When it gets a proposal, it accepts the proposed value iff it is greater than or equal to acceptedValue, and send an acknowledgment back to the proposer. If it rejects a proposed value, it sends back the join of the proposed value and acceptedValue back to the proposer along with the rejection.

Comparable values

This guarantees that all values accepted by a single acceptor are comparable. However, different acceptors may accept incomparable values. A value is said to be a chosen value iff it is accepted by a majority of acceptors. Note that any two chosen values have at least one common acceptor. Hence, any two chosen values are guaranteed to be comparable. Proposers simply count the number of acceptances every proposed value gets and whenever a proposed value gets a majority of acceptances, it knows that its proposed value is a chosen value. The proposer then executes the Decide() action, and declares the current proposed value as its output value.

Liveness and Termination

This approach ensures the safety requirements of the problem. We now show how proposers deal with rejections to ensure liveness and termination. A proposer waits for a quorum of acceptors to respond to its proposal. If all these responses are acceptances, the proposer is done (since its value has been included in an output value). Otherwise, it refines its proposal by replacing its current value with the join of its current value and all the values it received with its rejection responses. It then goes through this process all over again, using its current value as the new proposed value. Once a proposer proposes a new value, it ignores responses it may receive for all previous proposals. This approach ensures termination since all values generated in the algorithm belong to a finite sublattice of L (namely, the lattice L consisting of all values that can be expressed as the join of some set of input values).

Guarantees

Following are guarantees required in distributed systems in their increasing order of strongness. "Witness" means sequential execution of the same set of operations which potentially occured concurrently at possibly different servers

- **Eventual Consistency** Replicated states eventually agree on final state but in between they can have inconsistent states.
- **Serializability** Observed behavior of the replicated state machine on some set of (possibly concurrent) operations is the same as the behavior of the state machine (with no replication) for some "witness"

- **Sequential Consistency** Serializability and additional guarantee that program order of operations appears to be maintained in "witness"
- **Linearizability**: Serializability and provides the additional guarantee that any two temporally non-overlapping operations (in the execution) occur in the same order in the witness.

GLA can be used to get sequential consistency and linearizability

2.3 Algorithm

```
Algorithm 1: Lattice Agreement
1: // Proposer process
2: int UID // Unique id for a process
3: enum {passive, active} status = passive
4: int ackCount, nackCount, activeProposalNumber = 0
5: L initialValue // Initial value of the process
6: L proposedValue, outputValue =
7:
8:  action Propose()
9:  guard: activeProposalNumber = 0
10: effect:
11:   proposedValue = initialValue
12:   status = active
13:   activeProposalNumber ++
14:   ackCount = nackCount = 0
15:   Send Proposal(proposedValue, activeProposalNumber , UID) to all Acceptors
16:
17: action ProcessACK(proposalNumber , value, id )
18: guard: ?ACK(proposalNumber , value, id ) && proposalNumber = activeProposalNumb
19: effect: ackCount++
20:
21: action ProcessNACK(proposalNumber , value)
22: guard: ?NACK(proposalNumber , value) && proposalNumber = activeProposalNumber
23: effect:
24:   proposedValue= proposedValue  value
25:   nackCount++
26:
27: action Refine()
28: guard: nackCount > 0 && nackCount + ackCount  (Na + 1)/2 && status = active
29: effect:
30:   activeProposalNumber ++
31:   ackCount = nackCount = 0
```

```
32:   Send Proposal(proposedValue, activeProposalNumber , UID) to all Acceptors
33:
34: action Decide()
35: guard: ackCount  (Na + 1)/2 && status = active
36: effect:
37:   outputValue= proposedValue
38:   status = passive
39:
40: // Acceptor process
41: L acceptedValue =
42:
43: action Accept(proposalNumber , proposedValue, proposerId )
44: guard: ?Proposal(proposalNumber , proposedValue, proposerId ) && acceptedValue
45: effect:
46:   acceptedValue := proposedValue
47:   Send ACK(proposalNumber , proposedValue, proposerId ) to proposerId
48:
49: action Reject(proposalNumber , proposedValue, proposerId )
50: guard: ?Proposal(proposalNumber , proposedValue, proposerId ) && acceptedValue
51: effect:
52:   acceptedValue := acceptedValue  proposedValue
53:   Send NACK(proposalNumber , acceptedValue) to proposerId
```

2.4 State Machine Replication

State machine replication is a general approach for imple- menting data-structures that can tolerate process failures by replicating state across multiple processes.Consider a special class of state machines. We first assume that operations of the state machine can be classified into two kinds: updates (operations that mod- ify the state) and reads (operations that do not modify the state, but return a value).

Other Approaches

There are several approaches for implementing such state machines, each with different consistency and performance characteristics.

One approach is to allow each replica pro- cess to process reads and updates in arbitrary order; all correct processes eventually reach the same state upon coordination among themselves. Thus we get sequential consistency only here.

One approach to guarantee linearizability, based on generalized consensus, is for processes to agree on a partial order on the commands that totally orders every read command with every update command that it does not commute with. This alternative guarantees

linearizability but requires the use of consensus to compute the partial order, which is impossible in the presence of failures.

Using GLA

Following algorithm describes a wait-free algorithm for state machine replication based on generalized lattice agreement that guarantees serializability and linearizability. In this algorithm, the lattice L is defined to be the power set of all update commands with the partial order defined to be set inclusion.

```
Algorithm 2 Serializable ReplicatedStateMachine
 1: procedure ExecuteUpdate(cmd )
 2:   ReceiveValue( {cmd })
 3:
 4: procedure State Read()
 5:   return Apply(LearntValue())
```

```
 Algorithm 3 Linearizable ReplicatedStateMachine
 1: procedure ExecuteUpdate(cmd )
 2:   ReceiveValue( {cmd })
 3:   waituntil cmd  LearntValue()
 4:
 5: procedure State Read()
 6:   ExecuteUpdate(CreateNop())
 7:   return Apply(LearntValue())
```

Chapter 3

Implementation and System Design

3.1 Essential components

3.1.1 Transport Layer

This layer is responsible tp handle all connections between client-server and server-server. GLA assumes that message delivery is reliable. So we use TCP as transport layer protocol. GLA needs a primitive of reliable multicast which should be acknowledged by quorum(majority) of nodes involved in protocol. Additional logic is required to build this primitive. Further proposed messages may exceed packet length and so packet splitting and reassembling is required. All this should be reliable.

3.1.2 GLA layer

This layer uses underlying transport layer and provides implementation of operations shown in Algorithm 1. API exposed by this layer is proposeValue and ReadValue. This layer should be generic to handle any type of proposed message with its corresponding join.

3.1.3 State Machine Replication Layer

Actual operations from clients are received here. This layer creates a map of operation and its uniqueId called as *glaId*. This ID should be uniques across proposers. MACaddress of machine followed by counter is one possible implementation of *glaId*. Only glaId are passed to underlying GLA layer whereas map of operations and glaId is shipped to other acceptors through separate reliable connection. This is done since GLA only takes join of operations and doesnot need to know about its semantics. Thus minimal payload is passed to GLA layer and hence its execution is faster. Once operations are read from GLA layer then a witness of operations is created back using map of operations. This sequence of operations is used to process read requests.

3.1.4 Translator Layer

GLA needs operations to be commutative. But many real world data structues seldom have commutative operations. But there are tricks where operations are transformed to other operations preserving semantics of data structures [2]. These transformed operations are commutative and hence works with GLA. Also execution of these transformed commutative operations corresponds to some sequential execution of original operations guaranteeing correctness. This layer receives operations from client and transforms them and passed them to state machine replication layer.

3.1.5 Client Layer

Operations to be executed on distributed system are received from client. Each client establishes connection with a proposer and all operations executed by that client are proposed by this proposer. Multiple clients can connect to single proposer.

3.2 Possible Implementations

Some of functionalities needed for our system design are quite similar to other distributed frameworks. For instance the transport layer is available in various other implementations. So we considered following possibilities as base to build our system.

3.2.1 Windows Fabric

This is distributed framework provided by Microsoft something similar to Windows Azure. Fabric allows to create multiple coordinted servers and provide two kind of services. In stateless service there is no replication among servers whereas in stateful service there is underlying replication of server state. We started with implementation of GLA as stateless service where we built our own replication using GLA protocol.

The motivation behind using Fabric was its ease of use with Visual C# and .NET framework. WCF model was used to communicate between clients and servers. I built the Client Layer and State Machine Replication Layer for Fabric in summer 2012 at MSR Bangalore. But Microsoft is yet to release Fabric to public and so we couldn't continue with Fabric implementation once my intern was over. Thus we needed a fresh start.

3.2.2 Open Source Implementaions

We inspected some open source distributed framework implementations. We realised that open-source paxos implementation uses common primitives needed by GLA. We studied LibPaxos(C++)[3] and Concord system(Python)[4] but both didn't meet out requirements.

We studied research papers of No-SQL services like Yahoo PNUTS and Google MegaStore to gain some more insight in various implementation details. Facebook's Cassandra and Apache's Zookeeper also provide distributed framework and have made there implementation open-source. So with further study we decided to use ZooKeeper as our base system implementation.

3.3 ZooKeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by GLA. ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers (we call these registers znodes), much like a file system.It guarantees high throughput, low latency, highly availability and strictly ordered access to the znodes. Its strict ordering allows sophisticated synchronization primitives to be implemented at the client.

The service itself is replicated over a set of machines that comprise the service. These machines maintain an in-memory image of the data tree along with a transaction logs and snapshots in a persistent store. The servers that make up the ZooKeeper service must all know about each other. As long as a majority of the servers are available, the ZooKeeper service will be available. Clients only connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses and sends heartbeats. If the TCP connection to the server breaks, the client will connect to a different server.

Read requests sent by a ZooKeeper client are processed locally at the ZooKeeper server to which the client is connected. Write requests are forwarded to other ZooKeeper servers and go through consensus before a response is generated.

GLA supports only minimal operations as API to maintain simplicity:

- **create** creates a node at a location in the tree
- **delete** deletes a node
- **exists** tests if a node exists at a location
- **get data** reads the data from a node
- **set data** writes data to a node
- **get children** retrieves a list of children of a node
- **sync** waits for data to be propagated

3.3.1 GLA using ZooKeeper

ZooKeeper can be used to implement higher order functions at client side itself without any changes to server side. To start with we implemented GLA as recipe(client-side library) of ZooKeeper. This implementation requires that atleast one server is running.

GLA with BareBone zooKeeper

We maintain the single znode called as *gla* in zookeeper which is intialised to null value. Note that this znode is replicated across all servers. In Zookeeper, an update is materialized only after going through consensus of servers. Thus value contained in *gla* znode at any replica is also present at quorum(majority) of other servers. Thus znode keeps record of all accepted operations by quorum of acceptors. Hence value read from znode at any time corresponds to learnt value of GLA system.

Thus readValue of Gla is processed by simply returning value of znode from zookeeper. For proposeValue we need to take join of current znode value with inputValue and replace znode's value with this joined value. API exposed by Zookeeper allows to setData in a znode with check of version. If version of znode is same as one specified in setData then data is replaced otherwise it fails. So in GLA proposer first needs to find out the current version of znode. It then takes join of current znode value and its inputValue; and proposes this joinedValue to zooKeeper. But in meantime some other proposer might update znode making current version present with proposer stale. In such case proposer repeats the entire procedure. It reads zNode value, takes join with inputValue and proposes it back again. This repeats till either proposedValue is accepted or maximum retryCount is over.

```
GLA:  byte[] readValue() uses
ZooKeeper: byte[] getData(String znodePath) where znodePath is "\gla"

GLA: void proposeValue(byte[] inputvalue) uses
ZooKeeper: void setData(String znodePath, byte[] newProposedValue,
  Version oldVersion) and
ZooKeeper: void getData(String znodePath)
```

GLA with Modified ZooKeeper

The previous implementation required repeated setData usage since joining of input-Value and current znode value was done at client side. Since there are multiple clients joint value of a particular client may become stale due to concurrent updates by other clients. So one possible solution is to do the joining of inputValue and current znodeValue at leader server itself.

I modified codebase and added some functionalities in ZooKeeper, to provide function called ProposeData as API. In ProposeData, value proposed is routed to leader server.

Leader server takes join of its current value and proposedValue and updates its zNode value with this joint value. This modified value is then propagated to other servers. Since update of zNode value takes at a central place and is synchronized so there are no conflicts. Thus ProposeData is executed in only one join at Leader server.

```
GLA:  byte[] readValue() uses
ZooKeeper: byte[] getData(String znodePath) where znodePath is "\gla"

GLA: void proposeValue(byte[] inputvalue) uses
ZooKeeper: byte[] proposeData(String znodePath, byte[] inputValue)
```

3.3.2 Testing

Setup: 3 zooKeeperServers form our GLA service. Each server maintains a znode called */gla* We have three clients(proposers) independent of each other. Each client submits 1000 operations with 50% operations being ProposeValue and remaining 50% as ReadValue. Clients are running concurrently so there is possibility of jointValue getting stale as in implementation 1 (Gla with barebone ZooKeeper). In total 3000 operations are processed by our GLA service (underlying zooKeeper service)

3.3.3 Performance Comparison

```
Time are in milliseconds.
Time for implementation 1 (Barebone ZooKeeper):
Thread 0 is: 76256
Thread 1 is: 107071
Thread 2 is: 109196

Time for implementation 2 (Modified ZooKeeper)
Thread 0 is: 33425
Thread 1 is: 37890
Thread 2 is: 37512
```

50% ProposeValue in Implementation 1 typically requires 3-4 retries of setData. Hence time required is more in barebone ZooKeeper. Note time requied for Implementation 2 acts as a benchmark for best possible time we can attain.

3.3.4 Code Technicalities

New functionality added to ZooKeeper API is

```
byte[] ProposeData ( byte[] proposedValue )
   returns jointValue of proposedValue and current zNode value.
```

Major files in which proposeData functionalities are added

- **Zookeeper Client side**
  - *ZooKeeper.java* Creates Request and Response object (uses SetDataRequest and GetDataResponse classes)
  - *StateVector.java* Exposes GLA API to user. It uses implementation 1.
  - *StateVectorImplicitJoin.java* - GLA API is made available through this. It uses implementation 2.
- **Zookeeper Server side** RequestObject created at client side is passed through chain of RequestProcessors on server side.
  - *PrepRequestProcessor.java* prequest method contains a switch case which handles what to do if opType.proposeData is true
  - *SyncRequest.java* No processing at this RequestProcessor.
  - *FinalRequestProcessor:java* Calls zookeeperServer.processTxn() which returns ProcessTxnResult. This return object is modifed by adding class member byte[] outputValue. This outputValue is filled with new value after processData is called. It also creates a getDataResponse object which is returned finally to client side.
- *ZooKeeperServer* - Class managing all requestProcessors.
- *ZkDataBase* - Class maintaining in memory state of znode.
- *DataTree* - Actual logic of functionalities goes here. Join logic for proposeData is implemented here.
- **Testing**
  - *GenerateLoad* Creates a testbed for implementation 1
  - *GenerateLoadImplicitJoin* Creates a testbed for implementation 2

Chapter 4

Future Work

4.1 Implementation

Currently we are using zooKeepers underlying replication mechanism for GLA. This replication is primary master based, So in case when primary(leader) fails, ZooKeeper runs leader re-election. But leader election in reducible to consensus in asynchronous systems and hence leader election can fail in extreme conditions. True GLA protocol doesnt need leader election. True GLA is waitfree in all circumstances unlike other protocols.

The ultimate goal is to have full-fledged GLA system which doesn't use zooKeepers centralised syncronisation primitives. Currently as per Section 3.1 out of 5 layers, we are using transport layer of zooKeeper whereas Gla layer is built partially over ZooKeeper.In future Gla layer needs to be made completely independent. Once entire GLA system is ready then extensive testing needs to be done to compare performance of GLA protocol and primary-based protocol.

Finally we want to achieve best of two worlds. In scenario where leader is functioning properly there we will use primary-based protocol. Once leader fails then GLA protocol is used for time till leader is re-elected. So in case if leader-election fails then system is still available because of GLA protocol.

4.2 GLA as State Machine Replication

There is possibility of implementing data-structures that can tolerate process failures in an asynchronous distributed setting, while guaranteeing correctness (linearizability with respect to a given sequential specification) and progress (waitfreedom).[7]. For this effect we need to implement State Machine replication layer and Translator layer above GLA layer as explained in Section 3.1

4.3 GLA for distributed View maintenance

We are also inspecting on how GLA or its modification can be used to for asynchronous distributed view maintenance of database systems. It is possible to implement key-value table with GLA having following API.

- set(key, value) - Update operation
- delete(key) - Update operation
- get(key) - Read operation

Since set and delete have nullifying semantics it fits into category of data-structures that can be implemented using GLA.Partitioned key-value where partitioning is done on primary key can also be implemented by having seprate GLA instance for every partition of table.

Database tables can be viewed as key-value table with (key-primary attribute, value-all other attributes). Operations on database spanning only one partition (eg- set(key, value), delete(key)) thus can be implemented. But operations spanning multiple partitions require special treatment. Views like (Index, Projection, Selection) can be maintaining by having index on partitioning attribute. [9] Lets say base table B is partitioned on k whereas view W is partitioned on v. set(k, v) operation in B may need update on two replicas of W. It may lead to deleting (k, v") from partition p1 and inserting (k, v) in partition p2 of W.

Solution 1 Ship set(k, v) to all partitions of W. All partitions will check if key k is their table with some v". Partition which finds key k will execute remove(k). Other partitions will ignore this operation. And partition where v belongs will insert it. But non-optimal solution.

Solution 2 Set (k, v, v") implies change value of key k to v if previous value was v". In this case go to partition corresponding to v" in W and see if key matches to k. If success then insert(k, v) in partition corresponding to v. So here we need feedback from one GLA instance to execute operation completely. This needs to be worked out in detail further.

References:

1. Generalized Lattice Agreement, PODC-2012 by Jose M. Falerio, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, Kapil Vaswani

2. M. Shapiro, N. Pregui a, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. Bulletin of the European Association for Theoretical Computer Science (EATCS), (104):6788, 2011.

3. LibPaxos http://libpaxos.sourceforge.net/

4. Concord System http://openreplica.org/doc/

5. PNUTS: Yahoo!'s hosted data serving platform, VLDB 2008

6. Megastore: Providing Scalable, Highly Available Storage for Interactive Services, CIDR 2011

7. ZooKeeper: Wait-free coordination for Internet-scale systems, Patrick Hunt,Mahadev Konar,Flavio P. Junqueira and Benjamin Reed

8. Asynchronous WaitFree Linearizability: Sagar Chordia, G. Ramalingam, Kapil Vaswani, Kaushik Rajan.

9. Asynchronous view maintenance for VLSD database- Agarwal, Yahoo)