Abstractions for Concurrent Consensus - Go Implementation

Nithin Thomas

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Future Networked Systems)

Supervisor: Dr. Vasileios Koutavas

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Nithin Thomas

August 19, 2022

Permission to Lend and/or Copy

Abstractions for Concurrent Consensus - Go Implementation

Nithin Thomas, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Dr. Vasileios Koutavas

Consensus is a problem in any system using concurrency, such as systems distributed over a network, multi-core machines, or single-core machines with multi threading. This dissertation utilizes the theoretical foundations provided for abstractions of Concurrent Communicating processes and proposes a library designed to support consensus among communicating processes. The library is implemented in Go programming language, which supports communication over channels and shared memory between threads. The library is designed to provide an abstraction where a user can easily design the processes while the library handles communication needed for the transactional nature of these processes. The challenges and design approach of the library is discussed. Further, the analysis of implementation is performed in terms of its correctness, performance and usability.

Acknowledgments

I would like to extend my gratitude to my supervisor Dr. Vasileios Koutavas for his regular feedback and support, making this dissertation an iterative process. His fresh perspective and time spent on design discussions helped me learn new approaches and ideas. I thank Dr. Andrew Butterfield for his feedback on the design and its analysis.

I would like to thank my parents and sister for their unconditional support. I thank Anandu Pavanan for sharing his knowledge of Elixir programming language.

Nithin Thomas

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Distributed applications aim to provide users with abstraction while the complexities of the underlying system are hidden away. Such applications replicate data across participants for the concern of availability. In such an approach, consistency is a crucial aspect so that any data modification is done on all of the replicas. The application, as a whole, should move from one uniform state to another.

This problem of consensus between participants is common in Distributed Systems and dealt with in different ways according to the level of consistency required[1]. A generalized way of achieving consensus is provided by Paxos [7]. A set of state machines that can take up the roles of *proposer, acceptor,* and *learner* communicate with each other. Each proposer can propose a new value, which has to be accepted by the majority of acceptors. This value is learned by all learners involved. In a practical implementation, usually, all processes involved play all three roles of *proposer, acceptor,* and *learner.*

A more practical algorithm such as RAFT[8], demonstrates this through a number of processes out of which one can be elected as a *leader.* A leader acts as the proposer, all processes act as acceptors, and a majority of processes have to accept a value for a *commit.* In either case, consistency is ensured by making sure the majority of processes agree on a new proposed value. A system of distributed processes can also aim for complete acceptance instead of a majority for stronger consistency. All implementations thus require techniques to handle consensus problems.

Generally, there are two types of this problem, the consensus problem and the interactive consensus problem[2]. In the consensus the problem, all non-faulty processes that start with the same initial data should eventually decide on the same final values. An example is the 3-way rendezvous algorithm, where 3 concurrent processes, starting with the same initial state should agree on the value of the next state. All processes involved will transition to the new state, only if all of them converge to the same value of the new state. The state can be a value of a data variable or the content of a file. One of the three processes will be designated as a leader, and others take up the role of followers. The leader will verify the new state of followers, and if they do not converge, decide to roll back to the initial state. This can be extended to an n-way rendezvous algorithm, to fit a group of n processes.

In an interactive consensus problem, all non-faulty processes will agree on data for individual processes. These processes may not converge on the same data but will agree on the end state of all processes involved. This is a more general version of the consensus problem. An example is the Saturday Night Out(SNO) problem[3]. In such problems, all processes involved would require some conditions to be satisfied by other concurrent processes. These conditions may be regarding communication over the network, writing to persistent storage, and so on. If one such process can find concurrent processes that satisfy all the requirements, this group of processes can move to a new state and commit. The groups formed here can be dynamic and with a variable number of members.

In either case, the concurrent processes need to communicate with each other to reach a consensus. The communication could be with a designated leader, following a server-client behavior, or a peer-to-peer manner of communication. In this context, communicating transactions construct [6] is useful in ensuring consensus between processes, or rolling back if no consensus is achieved. The All or Nothing property of transactions can ensure a consistent transition between states. Communicating transactions can also use programmable aborts, which are triggered by the process. These may be triggered due to an external dependency specific to the process, or due to disagreement with other processes.

The TCML language [3] gives abstractions for modeling such communicating processes, that are transactional. An interpreter was developed using Concurrent Haskell, which creates threads and a *Trie* data structure to maintain a global view of all processes. With the global view available, a scheduler process will choose from available actions, such as commit or abort, among processes. The actions of the processes will notify a gatherer process, which will update the *Trie* global data. This can model consensus problems similar to 3-way rendezvous and SNO scenarios can be modeled using *restarting processes.*

The Elixir implementation [9] of this abstraction uses the actor model and message passing provided within the Elixir language. It uses a protocol structure to indicate how a group of processes in the consensus problem be created from available processes. A global coordinator handles the creation of groups and creates local coordinators. The processes are modeled into lightweight threads available in Elixir called *process.* The *process* has isolated memory space and can communicate with other processes by asynchronous messages.

1.2 3 Way Rendezvous

In a multi-party agreement scenario[4], a number of participants have to come to an agreement regarding an operation and thereby a new state of the system. They form groups and communicate among themselves to exchange necessary information, for example, the result of the proposed operation.

This is also considered to be a server-client model. In this paradigm, the server provides services, while managing the exclusive access to a state, effectively providing the effect of a locking mechanism. This ensures only one client accesses server data at any time. However, it can be proven that abstract implementations for n-way rendezvous using asynchronous communication are impossible [5]. Hence synchronous communication is the desired communication primitive for such abstraction.

In a 3-way Rendezvous protocol, one participant is designated as the server. The server process is designated so when the group is created. The server process and client processes are hence different. This asymmetry in synchronous communication is not desirable in all cases in general. The server can then communicate with other participants, and ensure that the participants are in synchronization with each other and the leader itself. If all participants are in agreement and ready to commit, the group can agree and thus accept the new state or data. If any of them fail or propose a different value, the group will roll back and each participant will revert to an initial state.

The Go programming language(Golang) provides lightweight threads called *goroutines*. However, synchronization is not straightforward in Golang as the memory can be shared among such goroutines, and techniques involving mutex and channels are used for communication. The goroutines are also not assigned exclusively to an operating system thread. Golang does not provide functionalities to explicitly kill a goroutine or to identify any failure in one. Due to these reasons, implementing a protocol similar to 3-way rendezvous would require the use of channels for communication and mutexes for safe access to shared memory. However, as discussed, this form of blocking, synchronous communication is not the best fit for a general abstraction of this problem. But, Golang does not support the message passing primitive which is a better choice for rendezvous problems in general[5].

1.3 Motivation

A coroutine is a concurrent computation, that can be suspended. However, a coroutine can be executed in different operating system threads. It can be suspended from one operating system thread and resume in another.

Similar to Elixir, Golang enables the creation of lightweight threads. Even though the concurrency model of Elixir and Golang is derived from

Communicating Sequential Processes(CSP), the lightweight threads -*goroutines*- are not isolated. Goroutines help in the easy implementation of concurrency by multiplexing coroutines while keeping the overhead to a few kilobytes. They can communicate via shared memory or channels. This required use of locking and mutex to avoid race conditions. Hence the communication is synchronous and is a blocking operation. Golang achieves concurrency by keeping active threads on a separate operating system thread.

The Haskell implementation of concurrent transactional processes uses a *Scheduler* to choose actions from processes, which is done randomly. However, a defined *Protocol* given in the Elixir implementation can improve the performance of the scheduler. This implementation relies on the isolation provided by Elixir. In case of a faulty process, this means that the fault will be ignored and will not affect other running processes. However, if a process fails, while execution within a group, the isolation, and ignored failure can lead to inconsistent states or for the members of the group to wait indefinitely.

Due to the shared memory model of goroutines, failure of one goroutine can halt the runtime. However, such failures can be identified using *recover* construct available in Golang and used to signal a process failure within a consensus scenario. This could also be useful in implementing programmable aborts within processes.

Additionally, an analysis of performance in terms of commits or rollbacks to the number of groups created for consensus scenarios is required. This would be particularly helpful in identifying the overheads in creating the groups out of available processes, and how a defined *Protocol* would help improve the implementation in this regard.

1.4 Research Question

This dissertation proposes an implementation of abstractions for concurrent transactional processes discussed above. The Go programming language(Golang) is used to implement these abstractions into a library. This library provides easy abstractions for a user to create a group of concurrent processes taking part in a consensus problem. The library provides functionalities for communication between processes, and to implement aborts within the process computation. The library also handles faults in processes from external dependencies. The concurrency features provided by goroutines are utilized in process computation, while *channel* construct in Golang can be used for communication between processes. The performance of the library can be analyzed against the number of concurrent processes. The correctness and usability of this library can be investigated by implementing particular scenarios.

1.5 Structure of the Dissertation

The following sections explain the background work which helped design this library. Section 3 explains the approach followed for the development of this library and the reasons for design decisions in creating these abstractions. The abstractions and the enclosed functionalities are discussed. Section 4 explains the implementation details of the library. The choice of constructs available in Golang is explained. The major components designed are discussed along with the overall computation logic. Some challenges faced in creating some functionalities and the workarounds are discussed. Few guidelines to avoid such challenges are listed. Section 5 analyses the library on its performance, correctness, and its usability. The performance of the library is discussed in throughput and the behavior when scaled up. Section 6 draws conclusions from the implementation and its analysis and discusses areas of improvement, possible new features, and future work.

Chapter 2

Background

Concurrent ML(CML) is introduced by John Reppy in [4] which supports synchronous operations as first-class values. It provides responsiveness with the preemptive scheduling of threads. CML uses synchronous message passing for communication between threads. Message passing provides a more robust model than using shared memory for communication. synchronous communication is better suited for detecting errors and enables threads to share information quicker.

Various synchronization and communication methods are discussed in the design choices of CML. While shared memory can be used for communication between threads, it brings in the complexities of access to shared values. As a solution, a locking mechanism has to be introduced. For this reason, as well as for better abstraction, message passing is considered to be a better approach in CML. In message passing, the abstraction can be the same for threads on a single processor and threads running on different machines on a network. The same cannot be the case for shared memory abstraction. Similarly, the design chooses synchronous communication over asynchronous mode.

To accommodate selective communication using synchronous communication, as well as provide a necessary abstraction for communication over the channels, CML introduces *event* to represent an abstract communication protocol. CML also extends the message passing communication to include a negative acknowledgment. This ensures the threads communicate about failures too, which is useful in terms of a transaction.

The implementation detail of using CML for terminal-based applications where concurrency is achieved through a single processor or parallelism achieved by different terminals on a network is discussed. The capability of CML in designing an interactive window application is also discussed, where threads handle different aspects of the application to enable responsiveness with the graphical interface.

In [10] *Transactor* programming model is introduced to design distributed components. The model is based on the actor programming model and focuses on the durability aspect of transactions. The components communicate via message passing, and the model can work in an unreliable environment with component failures, network failures , and dropped messages. The model involves a construct called a transactor which can create checkpoints and roll back to a previous checkpoint if needed. A transactor can be dependent on other transactors, and hence model the dependency within a multi-component distributed system. A transactor will be able to send asynchronous messages to others and hold information about its dependencies.

A transactor holds a state and provides facilities to update it through messages. The state is volatile, but a checkpoint can be created which is persistent. A checkpoint can be created only if all the dependent transactors hold valid states. This helps in creating a distributed checkpoint, where all components are in consistent, persistent states. In case of failures after a checkpoint, the transactors can roll back to this persistent checkpoint.

The transactors also hold transitive dependencies and the history of states for them. A transactor moves to a stabilized state before creating a checkpoint. Stabilized state is volatile still, where the transactor still processes messages but does not modify its state. When all dependencies are in agreement, the transactor can checkpoint its state, thus creating a new persistent history.

In [11] an abstraction called *stabilizers* are introduced to CML for creating checkpoints in CML. This aims at dealing with temporary faults within distributed and concurrent systems where the action of rolling back has to consider the previous global checkpoint. This is also necessary to avoid an inconsistent global state. Due to the concurrent nature of components, performing a set of local operations after a failure does not guarantee a consistent state of the larger system.

The Stabilizer is created using a stable section in the code which is monitored and is handled to be atomic. In the event of a revert, the program reverts to the state before the stable section. A *stabilize* method when used reverts the state of the program, which was immediately before the stable section. A stabilizer also includes a *cut* primitive, which defines a point in the program before which it cannot revert to.

The checkpoints are captured only at the entry into a stable section. A stable section can be dependent on other stable sections through communication. In this case, if the stable section reverts, the dependent stable section has to revert as well. The dependencies are tracked using a communication graph. To revert actions of a thread, the value of references is also maintained. For the need for reverting, the initial value of a variable is stored. For subsequent changes, a version list is stored for reference and the value of data. This also considers the use of locks for variable access. The order of locking and unlocking by threads are kept track of for the need for stabilization.

Communicating, non-isolated transactions are introduced in TransCCS[6], where transactions can coordinate to create a distributed checkpoint. All transactions involved have to commit before creating a checkpoint. The transactions communicate state information and their environment. In the event of a rollback, the transactions involved do not have to revert all changes, but instead, roll back to the last checkpoint created among them.

A transaction can be embedded into another, modeling the communication between them. This also enables the composition of distributed transactions consisting of several such transactions. TransCCS also introduces *programmable abort* language primitive, where a deterministic abort is caused by a transaction through its computation logic.

Abstractions are introduced in [3] through TCML, which can implement generic and interactive cases of consensus problems such as SNO. Processes communicate over channels, and a communicating transaction is represented in terms of the process P and P' which is the process to be executed if P aborts. It also provides primitives to create a channel within the scope of a transaction, as well as to send and receive data through channels. It also defines rules to embed a communicating transaction within another. This deals with the ordering of processes in the event of an abort in embedded transactions.

Further rules are defined to mark a transaction as *atomic* and commit primitive is provided to mark a communicating transaction commit. In the case of an embedded transaction, the innermost transaction has to commit for the outer transaction to be able to do the same. This ensures that all transactions involved in the communication have to agree to commit for the whole transaction to do so. The aborts are non-deterministic and transactions can be restarted for implementing scenarios such as SNO.

An implementation architecture is also introduced which holds all transaction information in the form of a Trie data structure. Any side effect creates a notification and is sent to a component called *gatherer*. The main thread is a *Scheduler*. Each process has a local transaction state. The Scheduler will send messages to threads to update their local state. The update creates an acknowledgment that is sent to the gatherer. This helps keep the Trie data containing the global view updated.

Various approaches are followed in designing a scheduler. The basic approach is a random scheduler which selects random operations from available processes. A staged scheduler selects actions according to a priority. A commit will be chosen over an abort or embed. A communication-driven scheduler will embed processes only if they are to communicate with each other. A delayed aborts scheduler will abort a process only if there is no non-sequential operation for a specific time. The analysis shows the random scheduler performed worse by an order of magnitude. All schedulers deteriorate exponentially with a scaling number of processes.

Work by Reppy on CML provides design approaches and arguments for synchronous communication between threads and demonstrates how the communicating threads can be used to achieve concurrency and parallelism. Stabilizers are introduced on CML to enable transactional properties of threads by creating a monitored section and global checkpoints. The work by Koutavas et al [3] introduces communicating transactions that can be used to model consensus scenarios similar to SNO or n-way rendezvous. This can be achieved by primitives provided for embedding, committing and creating, and using channels for communication. These lay theoretical foundations for communicating transactions without providing an effective method that they can be implemented.

The implementation of these primitives in the Go programming language poses a further challenge due to the shared memory model of Go. Communications are primarily through channels, and shared memory is used to communicate with the help of a locking mechanism. As the theoretical primitives mostly rely on message passing and actor programming model, a direct and simple implementation into Go is not entirely achievable.

Chapter 3

Approach & Design

3.1 Development Approach

The development of this library was completed by following an iterative software development approach. Regular discussions and correctness checks helped in creating the core functionalities and set direction for features to make it more seamless. The initial phase of development was focused on creating a complete working program in Golang that demonstrates the concurrent execution of goroutines and implements a correct consensus mechanism among them through communication via channels.

Following the initial implementation of the complete program, the approach was to separate the functionalities to be offered specifically by the library. This involved deciding the level of flexibility offered for a user while implementing a process and what fixed functionalities would be provided by the library. With the separation of library functionalities, the approach for giving the users to run their generic implementation of a process was discussed. Golang supports Generic data types since version 1.18 which was introduced in March of 2022. But as the development of this library was ongoing since November of 2021, the design approach used other techniques to allow generic process logic, which is discussed in section 4.

3.2 Process Abstraction

As Golang does not support isolated processes and message passing, the design of this library embraces the motto of Golang, ”share memory by communicating”. In this approach, any data to be shared between goroutines is passed around through channels. A goroutine is then designed to accept a memory reference to data, perform computation with it, and send it out of the goroutine using another channel. This approach of memory sharing helps design concurrent goroutines, while ensuring that data is accessed by only one goroutine at any given time.

By design, a goroutine can be only controlled by the program logic within that goroutine. Considering this, a communicating process(*Process*) is designed to be a Golang object. References to this *Pocess* objects are used to handle grouping and monitoring completion. The *Process* object will also contain necessary data for the execution of a goroutine, which will execute the *Process* logic implemented by the user. This is useful in explicitly executing the process only after it is grouped for the consensus problem. Since the *Process* is handled as a reference to the object, the library can insert group-specific data into Process when required. This can include information about peer processes, as well as the designated role of the process in the created group. The Process object should also provide functionalities for communication within groups, as well as to signal aborts from within the Process logic implementation.

Complete execution of Process logic should indicate its readiness to commit. A commit operation will not involve any more computation to avoid the possibility of failure after the Process is ready to commit. However, any fault within Process function logic or a programmed abort should indicate a failure in consensus. At this point, the operations performed by the Process have to be reverted and the data state has to be reverted to the previous, consistent state. For this reason, the Process object will also hold a rollback function, which cleans up the effect of Process operations in case of a rollback.

In any consensus scenario, communication between processes is necessary. To facilitate this, the library provides functionalities for a Process to identify its peers and their roles in the group. The library also provides functionalities for a Process to send data to another, using an *Inbox* channel for each Process. This is designed to be of generic data type and can be used by processes to even define new channels and bootstrap for further communication.

```
Processes
{
    Roles(Possible roles to play in a group)
    Role(selected role to play in the group)
    Function
    Rollback Function
    Inbox(channel)
}
```

3.3 Protocol

A *Protocol* can be useful in defining how a group involved in the consensus problem will be created. The design of *Protocol* considers the possibility of a process that can play different roles in a group. Such a *Process* can provide a list of different roles it can play in the group. However, it is up to the library to choose the role the process will play. The library will follow a greedy approach in this case, such that a process will be assigned a role from its list of roles, which will help in creating a group and starting Process execution. For this reason, the Protocol will also need to rank the priority of possible roles. Consider a group is to be created for a 3-way rendezvous problem, where a Process will be *verifier* and two Processes will play the role of *client*. The verifier will receive new values from clients, and if any of them are different from the new value proposed by the verifier, the group should roll back. The Protocol for this group could be:

```
Protocol
{
    roles -> (verifier: 1), (client: 2)
    priority -> verifier, client
}
```

This explains what roles, and how many of each role are required to build a group. The priority helps the library with its greedy approach to creating groups. If a Process can be a verifier as well as a client, the library will impose the role of verifier on the Process.

3.4 Monitor

A *Monitor* is the process (Fig. 3.1) that will keep the pool of all available processes, and their roles, and will be responsible for the creation of groups. Every new Process created will be registered with the Monitor. A process will be created with a selection of roles it can perform. The monitor will hold a pool of all available processes. When the Processes get registered to the Monitor, it can check the pool and with the use of Protocol, check if there are enough available participants for creating a group. If there are enough processes with the appropriate roles available, Monitor can create a *Group Monitor*. A Group Monitor can be executed in a separate goroutine, which can handle the consensus within the group.

```
Monitor
{
    protocol;
    available processes: <List of Process>
}
```

When the group is created, the new goroutine specific to the group runs concurrently, and the involved processes are removed from the list in Monitor. These can be put into the Monitor again in case of an abort or rollback. This can be useful in problems similar to SNO. The Monitor will wait for all the groups to complete execution, which involves either a commit decision or a rollback decision. The only functionality of Monitor is to group processes and hand the grouped processes to a new goroutine.

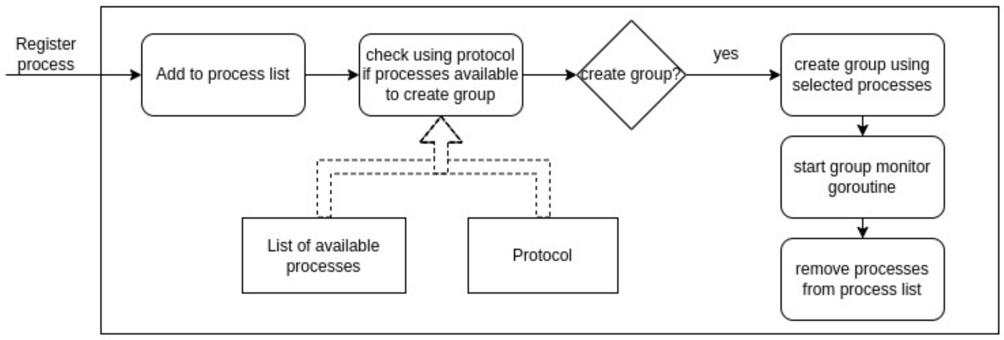
Register process
Add to process list
check using protocol if processes available to create group
create group?
yes
create group using selected processes
start group monitor goroutine
remove processes from process list
List of available processes
Protocol

Figure 3.1: Logic flow of Monitor when a new Process is registered

3.5 Group Monitor

A *Group Monitor* is the component in the library that handles the execution of a group and checks for consensus. As Monitor creates a Group Monitor, the list of processes in that group is embedded into it. Hence, a Group Monitor can begin by setting up means for communication between these processes, as well as ways for each Process to signal readiness to commit or abort.

```
Group
{
    processes (as part of the group)
    completion Channel (to listen for completion messages)
    fault Channel (to listen for failures or abort messages)
}
```

Since each Process object consists of the Process logic, as well as any data parameters needed for execution, the Group Monitor (Fig. 3.2) can start Process computation on a new goroutine. After setting up common channels, the Group Monitor will start a goroutine for each Process within the group. The Group Monitor would then listen for messages from each Process goroutine. These messages will signal either a readiness to commit, or a fault. If all Processes signal readiness to commit, the Group Monitor will broadcast a commit message to all Processes in the group. This should complete the execution of all Process goroutines, and no further computation should be carried out in these goroutines.

If any of the Processes involved in the group sends an abort message, the Group Monitor will broadcast a rollback message to all Process goroutines involved. The Process goroutines will halt after execution of the rollback function which is part of the Process object. If the group arrives at a consensus and commits to the new value, the Group Monitor goroutine can end its execution. If the group aborts and reverts to the previous value, the Group Monitor will send the Process objects involved back to the Monitor goroutine. This will help in restarting a Process if needed and creating a new group for consensus, thus starting a new Group Monitor goroutine.

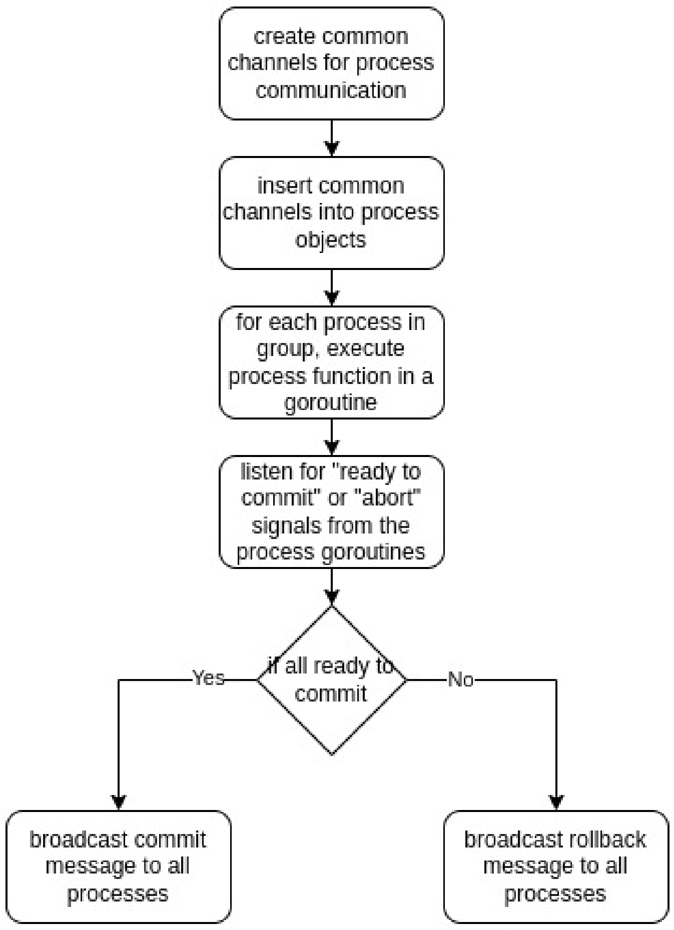
create common channels for process communication
insert common channels into process objects
for each process in group, execute process function in a goroutine
listen for "ready to commit" or "abort" signals from the process goroutines
if all ready to commit
Yes
No
broadcast commit message to all processes
broadcast rollback message to all processes

Figure 3.2: Logic flow of Group Monitor thread to monitor Process executions

3.6 Overall Design

The library groups and executed the processes, and follows a nested approach on goroutine invocation. This creates a tree pattern (Fig. 3.3) of goroutine creation, where each node represents a running goroutine and all children of a node are goroutines created by the parent node. Each node waits for the completion of all children nodes. The Monitor goroutine is the root of this tree, which holds all available Processes, and the Protocol to follow in creating groups. It creates groups and starts Group Monitor goroutines when enough Processes are available to fit the Protocol. Each Group Monitor goroutine will setup up necessary channels and create a goroutine for each of the processes in the group.

This hierarchy, however, does not mean an implicit monitoring effect. If a goroutine fails, the failure is not communicated to the parent goroutine. By design, Golang does not have a parent-child relationship between goroutines. However, completion of a goroutine can be monitored by a channel, which is, however, a blocking operation. In the structure to be designed, the Group Monitor goroutines need to know failure and aborts within Process goroutines. To achieve this failures have to be detected within these goroutines and use channels to signal it. The implementation detail of this technique is discussed in section 4.

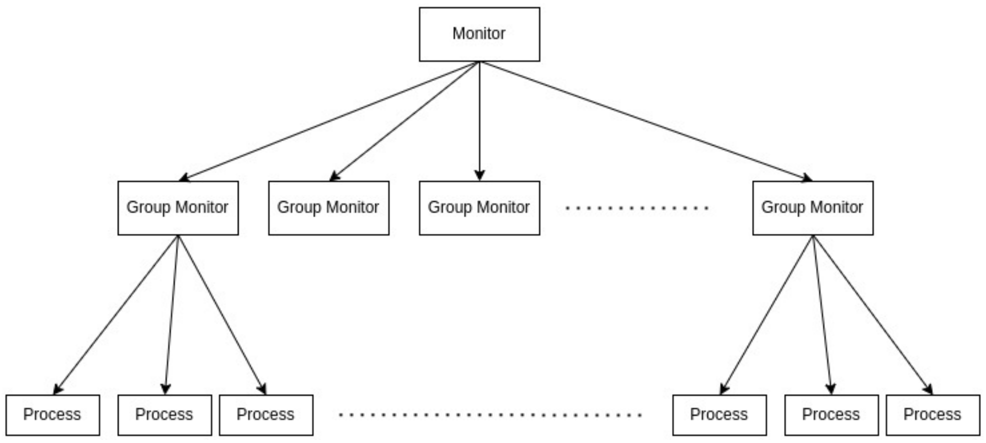

Figure 3.3: Tree structure formed by the creation of goroutines

The Monitor goroutine does not depend on the failure or success of the goroutines it created. The Group Monitor goroutine has to handle failures and arrive at either a commit decision or rollback decision. However, the Monitor goroutine requires completion of all groups created. For this reason, the goroutine has to wait for all children to complete execution. This is achieved by using a counter of all actively running Group Monitors. The *WaitGroup* feature of Golang is used to implement this and the details are discussed in section 4.

3.7 Abstraction as a library

The library designed here hides the complexities of creating a group, and communication within a group from a user. Any failure within processes, as well as the consensus result, is handled by the Group Monitor, which is completely decoupled from user code. The user has to define Protocols and create a Monitor using any of these Protocols. The user is also required to create a function that holds the execution logic of Process, as well as a Rollback function which will clean up the effects of Process operations in case of a rollback decision.

After the creation of the Monitor goroutine, the user is only required to create Processes with appropriate roles according to the Protocol, and register them with the Monitor. While registering these Processes, any parameters required for the execution of the Process function also have to be passed on to the Monitor. After the processes are registered, the user only has to wait for Monitor execution to complete. This happens when all groups created from Monitor finish execution.

If utilizing the library, the user is hence only concerned about the computation logic of each process within a consensus problem, conditions for successful completion , and conditions to program aborts. The overhead of grouping, synchronization for commits, or rollbacks is hidden away. The library provides additional functionalities for a user to have their processes communicate within a group. The grouping of processes is dynamic using the library. To an extent, it depends on the order in which Processes are registered with the Monitor. But if processes are restarted, this cannot be the right assumption, as the Monitor works on a greedy approach and any available Processes can be grouped.

Chapter 4

Implementation

4.1 Process

The Process is the object which encapsulates all necessary information required to execute one process and information about peers and channels to communicate with them. The Process structure created in Golang is as below:

```
type Process struct {
    Roles map[string]bool
    Role string

    function interface{}
    cleanup  interface{}
    args []reflect.Value

    peers map[*Process]bool
    Inbox chan (Message)

    errorChan    chan (int)
    successChan  chan (int)
    commitChan   chan (int)

    ExecCount int
    GroupNum  int
    Ctx       context.Context
}
```

The *Roles* attribute holds all the possible roles the Process object can play in a group. this is useful for the monitor while creating a group. The *Role* attribute is the actual role the Process will play in the group. This is modified in the Process object by the Monitor, only after the group is created and the role of the Process is confirmed. If the process is restarted, the Monitor will group it again, and edit the Role attribute as needed.

For the use of this library to create processes, a process function has to be created in Golang along with a cleanup function which would be executed if the group of processes decides to roll back. Any arguments required for the execution of these processes have to be registered with Monitor. To keep these attributes generic, *Reflection* can be used. Reflection refers to the capability of a program to analyze its structure. In Golang, using Reflection[13], the raw value of a variable and its type can be examined. This capability is used here to save the arguments as raw values of type *reflect.Value*.

Interfaces are used in Golang for composability. Interfaces ensure that an object behaves in a particular way. This is achieved by defining function signatures in an interface. If an object defines methods with these function signatures, it implements the interface. In Golang, interfaces can be used for function parameters, and any object that implements a certain interface can be used with the function. In such cases, an empty interface would mean there are no necessary methods to be implemented in an object. So any object can take the place of an empty interface. Here, an empty interface is used to store the process function and the cleanup function. When the function has to be executed, the reflection capability of Golang is used and the value of arguments provided while the process was registered as :

```
...
reflect.ValueOf(p.function).Call(p.args)
...
```

The *peers* attribute of the Process object stores a set of all peer objects in the group it belongs to. As Golang does not have an inbuilt set data type the map data type is used, where keys are the memory address to the project object, and the value is always True. This data will be added to the Process object by the Group Monitor goroutine, while it sets up common channels before starting the execution of each process. The *Inbox* channel will be created in the Process object. This can be used by processes to send messages to peers. The channel is of type *Message*, which is a structure defined within the library to enable data of any kind. Message is essentially a map data type, with keys to be of type string, and the values can be any type, which is again achieved by an empty interface:

```
type Message map[string]interface{}
```

This helps the user to exchange information between peers of any type. Another channel could be exchanged too, as a way to bootstrap specific communication between peers.

The channels, *errorChan, successChan, commitChan* are created by Group Monitor and injected into the Process object. Process can use signal an abort through errorChan, and readiness to commit through successChan. The commitchan is for the Group Monitor to broadcast the commit signal to all processes involved in the group. The Process object provides methods for a user of the library to signal failure or readiness to commit.

The *ExecCount* in Process keeps track of how many times the Process has been tried to be executed. The protocol defined for Monitor includes a maximum retry count. This helps in avoiding retrying a process forever. The Monitor can check the execution count of Process and if it's above the threshold, choose not to start it again. The Group number is modified when a Process is put into a group. It is useful in logging and analyzing how many and which groups the Process was put into.

The *Context* type is defined in context package [12]. It is used to stop unnecessary goroutines and has an associated cancellation signal. Context is created and carried by the goroutines. While the goroutines cannot be killed from the outside, the goroutine can be designed in a way to check if the context is still active. Goroutines can check the status and decide to stop if the context is inactive. A cancellation signal is received through an associated *Done* channel of context. The select construct can be used to design goroutines to check context as :

```
...
go func{
    select {
        case <- ctx.Done(): // context was canceled
                            // and a signal is
                            // received
            return
        case intData <- integerChannel:
            // use intData to perform computation
    }
}(ctx context.Context)
...
```

When any process in a group aborts, the Group Monitor cancels the shared context. This can be used to stop the computation of other processes. The library does not implicitly kill the processes, but the user has to design process functions to check the cancellation of context. While the user is free to implement the process function logic in any way, the process function should follow a specific signature to use the Process methods, and if needed check the context of the group to ensure active context. The process function and cleanup function should use the following signature:

```
...
//func(p *lib.Process, /* arguments*/)

//eg:
```

```
func(p *lib.Process, num *int, name *string){
    ...
    //checking context
    select {
        case i <- numChannel:
            if i==0{
                p.Fail("received 0")
                //using Process method to signal
                //failure or program abort
            }
        case <-p.Ctx.Done():
            return
            // as context is killed
            // signal received on Done channel
            // stops goroutine
        }
}
...
```

4.2 Monitor

The monitor is the object that handles all the available processes, groups them with a defined grouping Protocol, and starts the execution of Group Monitor goroutine for each of these groups. The Monitor structure is designed as:

```
type Monitor struct {
    counter              int
    protocol             Protocol
    state                map[int]*Group

    processChan          chan *Process
    retryChan            chan *Process
    newProcessChan       chan *Process

    wg                   *sync.WaitGroup
}
```

The counter attribute keeps track of how many groups were created by the Monitor. This is also used in assigning an identifier to the group when its created. The state attribute holds references to all groups created. It is a map with the group identifier as the key and the memory address of the Group object as value. The *protocol* attribute holds the grouping protocol the Monitor has to follow. It is implemented as :

```
type Protocol struct {
    roles    map[string]uint
    priority []string
    maxRetry uint
}
```

The roles attribute defines the possible roles accepted in the group. Any process to be put into the group should be playing one of these roles. the priority attribute of Protocol helps the Monitor to follow its greedy approach in the grouping. If a process is available, which can play multiple roles, the Monitor assigns a role to the process according to this priority defined in the Protocol. The *maxRetry* attribute is used while restarting processes. If Monitor has a process available, whose execution count is above maxRetry in the Protocol, the process is not executed again. This avoids infinite retries for a faulty process.

After the Monitor is created with the required Protocol, the Monitor can be started with the *Start* method. New processes can be registered with the Monitor after its started, using the *RegisterProcess* method. It requires the roles the process can play in the group, the process function, the rollback function and any arguments for process function.

```
RegisterProcess(role map[string]bool,
    f interface{},
    c interface{},
    args ...interface{})
```

The Monitor has to be started as a goroutine, and it listens on a *NewProcessChan* channel to accept a new process object, and add it to its list of available processes. Internally, the Monitor goroutine starts another goroutine, named *ProcessHandler* that will listen for new processes in NewProcessChan channel as well as *retryChan* channel. All groups that rollback will send the involved processes to Monitor through retryChan channel. These also have to be added to the list of available channels, if they have not been executed more times than specified by the maximum retry count.

To work with both channels, the select construct is used in ProcessHandler, which listens to both NewProcessChan and retryChan. A fan in design is then created with the *processChan* channel so that all processes coming in from either channel are sent to processChan channel. The Monitor goroutine listens for processes on the processChan channel.

```
func (m *Monitor) Start() {
    //starting process handler goroutine
    go m.ProcessHandler()
    ...
    //listening on processChan
    p := <- monitor.processChan
    ...
}
ProcessHandler() {
    for{
        //listens on NewProcessChan
        // and retryChan
        // sends any incoming to processChan
        select{
            case p := <-m.newProcessChan:
                ...
                //sends to processChan
                monitor.processChan <- p
            case p := <- monitor.retryChan:
                ...
                monitor.processChan <- p
        }
    }
}
```

4.3 Group Monitor

The Group object holds information about processes involved in a group and common channels required for communication with them.

```
type Group struct {
    Gid int
    processes map[*Process]bool

    errorChan    chan (int)
    successChan  chan (int)
    commitChan   chan (int)
}
```

The group is identified by *Gid* which is given while creation by the Monitor. The *processes* are a set of processes that belong to the group. These processes are provided by Monitor and removed from its set of available processes.

A *Waitgroup* is part of the sync package in Golang. It enables the program to wait for a group of goroutines to finish. The Waitgroup is initialized, and for every goroutine to start execution, Waitgroup is incremented. the goroutine will take the Waitgroup as an argument or shared data, and on completion of its computation, the *Done* method is used to decrement the Waitgroup.

Once the Group is created, the Group Monitor goroutine is executed by the Monitor. This goroutine also takes a Waitgroup created at the Monitor. This keeps track of all running Group Monitor goroutines. Thus Monitor waits for all Group Monitor goroutines to stop its execution. In the Group Monitor, the channels for communication with processes are created. The *errorChan* and *successChan* channels are created for processes to indicate failure and readiness to commit respectively. The *commitChan* channel is created for Group to signal processes to commit. All these channels are created and injected into Process objects while creating the Group.

The Group Monitor also creates a context to be shared by all member processes and injects this context into them, before the execution of process functions. When a process aborts, the Group Monitor receives this information through errorChan channel. In such an event, the Group Monitor will cancel this shared context, which will signal other processes to stop execution.

Every Group Monitor also creates a Waitgroup in its scope and increments it for each process goroutine it invokes. This is used to have the Group Monitor wait for all processes to stop execution. On finishing execution, all processes will decrement the WaitGroup. The Group Monitor goroutine is designed to be short and only communicates with processes over channels. This is done to avoid failures from the group monitor itself. It will only listen for signals from processes, and send a commit or roll back broadcast to processes. Each process goroutine can recover from a runtime panic. This will be signaled to Group Monitor and handled.

```
GroupMonitor(wg *sync.WaitGroup, retryChan chan *Process) {
    defer func(){
        //decrementing Waitgroup
        //from Monitor after execution
        wg.Done()
    }()

    //creating Waitgroup
    //to track process execution
    groupWG := sync.WaitGroup{}

    //creating shared context
    ctx, cancel := context.WithCancel(context.Background())
    for p := range g.processes {
        p.InsertContext(ctx)
    }

    //executing all processes
    for p := range g.processes {
        groupWG.Add(1)
        go p.Exec(&groupWG, ctx)
    }
    ...
}
```

Once the process executions are started the Group Monitor listens for signals from processes. It would either wait for all processes to indicate readiness to commit or for one process to signal a failure. On receiving a failure, Group Monitor cancels the shared context. A broadcast is implemented through channels by closing the channels. When a channel is closed, a message is received to any goroutine reading from the channel.

```
GroupMonitor(wg *sync.WaitGroup,
            retryChan chan *Process)
{
    ...
    select{
        //cancel context on receiving
        //failure signal
        case <- group.errorChan:
            cancel()
        case <- group.successChan:
            completionCount += 1
            //broadcast commit signal if all
            // processes complete
            if completionCount == len(group.processes){
                close(group.commitChan)
            }
    }
    ...
}
```

4.4 Challenges

Signalling Abort

The primary challenge faced was to identify a failure or programmed abort from a process. Goroutines cannot be monitored as in an actor model of a program. The shared memory of Golang instead has to use channels and explicit steps to send a failure status. In the initial phases of this dissertation where the programs were designed to model transactional behavior of processes, channels were used extensively for this purpose. However, when the functionalities of the library are separated, and the user has to be offered flexibility in designing the process, the approach had to be modified.

The *defer* functionality of Golang enables you to set a block of code to be executed at the end of a function. This happens even if there is a panic(failure) in the function. The built-in *recover* function enables the identification of panic in a function when used within a defer block. In the updated approach, the user is free to design a process function as they need. The process function is executed within a wrapper function, which has a defer statement to identify any failures. This could be a runtime failure or a programmed abort. In either case, the group is set to roll back.

```
func (p *Process) Exec(
    groupWG *sync.WaitGroup,
    ctx context.Context) {
    defer groupWG.Done()
    //setting execution count of process
    p.ExecCount += 1

    //wrapper function
    func(){
        defer func() {
            if r := recover(); r != nil {
            //sending failure to group Monitor
            //through shared group channel
                p.errorChan <- 1
            } else {
                p.successChan <- 1
            }
        }()
        //execution of user defined process function
        reflect.ValueOf(p.function).Call(p.args)
    }()

    //waiting for commit or rollback
    // signal from group monitor
    //after process function execution
    select{
        case <- p.commitChan:
            //commit - no further action
        case <- p.Ctx.Done():
            // context was killed
            // indicating roll back
            // execute rollback/ cleanup function to
            // cleanup changes from process function
            reflect.ValueOf(p.cleanup).Call(p.args)
        }
}
```

Controlling Goroutines

The Group Monitor starts a goroutine for each process, and if it receives an abort from any of them, other processes have to be stopped. Situations where a process might wait for a long period for an external dependency, such as a network request, are possible. In such cases, it's better to kill the group of processes when at least one abort message is received. However, the goroutines cannot be killed from outside its scope.

To achieve this, the context package in Golang is used. This helps the user check for a shared context, and continue operations in process functions only if the context is active. This solution needs mindful design from the user and does not kill a goroutine. Instead, the goroutine logic actively checks for the validity of context.

Race conditions

The library spawns several goroutines to support the collective execution of process functions. To maintain the groups and the available processes, the data has to be maintained between the Monitor and Group Monitors. In the initial implementations, this caused issues of race conditions, as both Monitor as well as Group Monitor tried to access data about the same set of Processes. However, the use of mutex for read and write access to this data solves the issue. But as this is a blocking operation.

To avoid the blocking operation, and thus avoid any waits in these goroutines, the subsequent implementations separated the data as needed by each goroutine. In this design, the Monitor only keeps information required for all available Processes. When a group is created, and the Group Monitor goroutine has to be spawned, the Processes belonging to this group are removed from the set of available ones. On the other side, Group Monitor will only have to handle data regarding these particular Processes in its Group. This separation avoids the need for a mutex.

Deadlocks

As the process goroutines require communication with each other for reaching an agreement, the library provides a method to use peer information and send data of type *Message* as defined in the library. This also enables the user to send channels over the provided channel and bootstrap further communication. However, the channel communications are blocking operations. While using an unbuffered channel, a sender waits for an active recipient, or the recipient waits for a message to come through. This leads to situations where all goroutines are waiting for data from the channel, or all of them are trying to send data in the channel. Either case leads to a deadlock.

This is more relevant in the case of rollback in a group. If one process failed or signaled an abort, and the goroutine has completed its execution, the other peers may end up waiting to receive something from the particular process, or send information to the process. But as the process has already completed execution, the peers will end up waiting, thus creating a deadlock.

This is solved by using the shared context, and canceling context when a rollback has to be performed. This also requires the user to check if the context is active within the process function. This helps in avoiding deadlock where a process is waiting for data from a peer through a channel. When a process is completed, the channels within the process function will be dead, and the process *Inbox* the channel will be killed. In Golang, trying to send data on a closed channel causes failure. This is useful in avoiding deadlock, as the failure will only signal a rollback to the group.

4.5 Guidelines

The library is implemented to enable consensus through the shared memory model and functionalities provided by Golang. While it lets the user design process functions for their use case, the following are some guidelines to use the library and to avoid running into deadlocks.

Structure and arguments of Process function

The Process object contains methods to use in Process function, such as for sending a Message to peers, or signaling abort. To use these methods, the Process function should be able to access the Process it is part of. To enable this, the process function should expect a pointer to a Process as the first argument. When *RegisterProcess* is used, a Process object is created, and the process function becomes part of it. The cleanup function works similarly. The signature of these functions should be as :

```
//Process function and cleanup function
//func(p *lib.Process, /*additional arguments*/)
//eg:
processFunction = func(p *lib.Process,
                        value *string,
                        result *string){}

processRollback = func(p *lib.Process,
                        value *string,
                        result *string){}
```

It is also important that the arguments used in both process function as well as its cleanup function are same. The RegisterProcess function is used as :

```
monitor.RegisterProcess(
    roles,
    processFunction,
    processRollback,
    &valueString,
    &resultString,
)
```

The process function will be invoked with the arguments provided in RegisterProcess, and if the process has to roll back, the corresponding cleanup function will also be invoked with the same arguments.

A suggestion to achieve transactional nature in these processes is to use pointers as arguments to process function. The process function could write data to the memory pointed in the course of its operation. If the group rolls back, the cleanup would involve clearing the pointer. This can be easily achieved by setting the argument to a null pointer in the rollback function.

```
processFunction = func(p *lib.Process, result *string)
{
    ...
    //communicate with peers
    //reach agreement on result
    result = &aStringVariable
}

processRollback = func(p *lib.Process, result *string)
{
    //clear the pointer
    result = nil
}
```

Avoiding Deadlocks

As discussed in the previous section, the context package is used to control and stop goroutines if the group fails to commit. When a Process function is designed, the user can choose to create new channels and communicate over them. To avoid deadlocks its safer to use the *Inbox* channel provided by the library, and the associated methods, while using *Message* type to communicate. However, if a user creates new channels, the usage should be combined with context checks. If a peer has failed, the process waiting for data over such a channel will lead to deadlock. The context is inserted into the Process and can be accessed by the **lib.Process* argument of process function. The *Done* function associated with the context should be used to verify the aliveness of the context.

```
func(p *lib.Process, result *string){
    ...
    //channel intChannel created by the user
    ...

    select{
    case intValue <- intChannel:
        ...
    case <- p.Ctx.Done():
        return
    }
}
```

Chapter 5

Analysis

5.1 Correctness

The analysis for the correctness of this implementation checks if the library can group the processes correctly according to a protocol. Further, the behavior of the processes is checked to ensure the commits are done only when all processes agree, and if not they roll back. For this analysis, the library is used to create a simple consensus scenario where three processes exchange values. The protocol defines an asymmetric group where one process is a *verifier*, and two processes are labeled *peer*. To verify the grouping behavior, a certain number of processes can be registered with the Monitor, and check how many groups were created from the available processes. In the protocol defined here, a group needs one *verifier* and two *peers*. To avoid regrouping when a group rolls back, the retry count on the protocol is set to zero.

The verifier process generates a random number. A probabilistic abort is introduced by generating this random number in the range [0-100) and aborting if the number is less than the imposed probability of failure. In case of no failure, the number is sent to peer processes. Each peer process receives the number and ends its execution. The designed scenario is executed with different numbers of verifiers and peers registered to the Monitor. Combinations, where excess processes are present, are also used. Following are the observations from different executions:

As observed in 5.1, the number of groups formed in each combination is according to the availability of processes and obeying the protocol defined. the excess processes are not executed at all as they are never put into a group. The protocol defined here does not retry execution on abort, but if it had the number of groups would be higher. The commit counts also show how many groups were able to commit while a certain percentage of abort was introduced. The abort is probabilistic and introduced using random number generation.

Table 5.1: Number of groups created with respect to different combinations of verifier and peer process counts

# verifiers	# peers	groups	abort %	commits
5	5	2	20	2
5	10	5	20	3
10	10	5	20	4
10	15	7	20	7
10	15	7	50	5
10	20	10	20	9
10	20	10	50	5

5.2 Performance

The consensus scenario defined earlier for the correctness check is further used in the analysis for performance. To analyze performance, the Monitor is started with the defined protocol, and several verifier and peer processes are registered with the Monitor to create a certain number of groups. The time required for the Monitor to group all processes and wait for the completion of all groups is noted. This information is also extrapolated to calculate commits per second. Here the Processes are registered in numbers convenient for Monitor to group according to the Protocol defined. That is, for every verifier process registered, two peer processes are registered with the Monitor. The growth of execution time to the increasing number of processes and thus the increasing number of groups are analyzed to study the scalability of the implementation.

These readings were obtained by execution on a Lenovo ThinkPad E14 Notebook with 16GB RAM(SODIMM DDR4 Synchronous 2667 MHz (0.4 ns)) and Intel i5 Processor (Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz), running Ubuntu 20.04.2. the implementation is made using Go version 1.13.8. Five executions were carried out per combination and the average value is used for analysis.

Figure 5.1 shows the execution time, where the processes are grouped statically, without using the library. In this design, each group is started as a group of goroutines, and channels are used to monitor readiness to commit. Figure 5.2 shows the execution times for the same consensus scenario but implemented using the Go library. As observed, there is a significant difference in execution time. The static implementation completed the execution in less than 200 microseconds for 20 groups, while the implementation with this library required nearly 1 millisecond for completion. This difference is due to the setup of Monitor, channels, and additional goroutines for each group. The communication between these goroutines is synchronous and blocking operation.

While the static implementation is faster, it has to be done on a case-

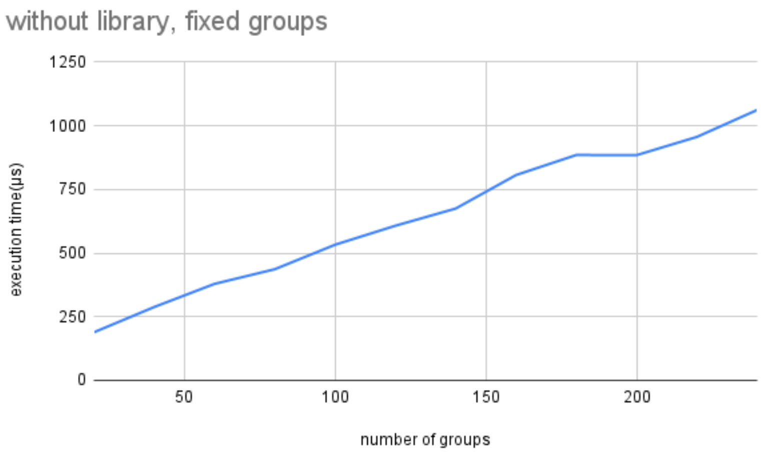

Figure 5.1: performance on creating concurrent processes for consensus without the library

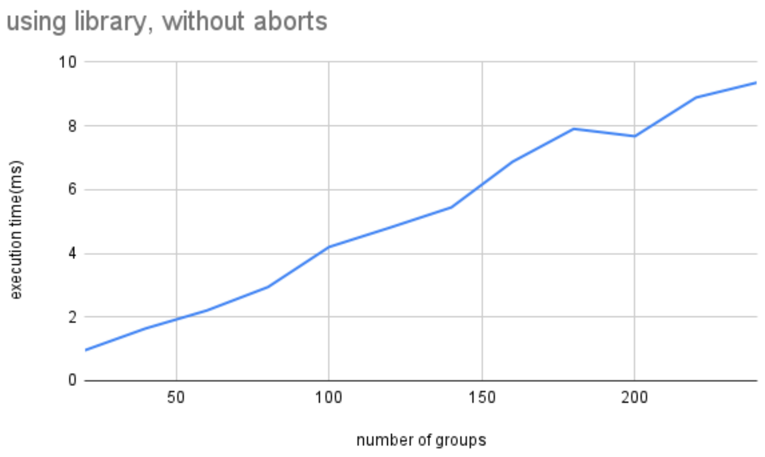

Figure 5.2: performance using the library with no retry for processes

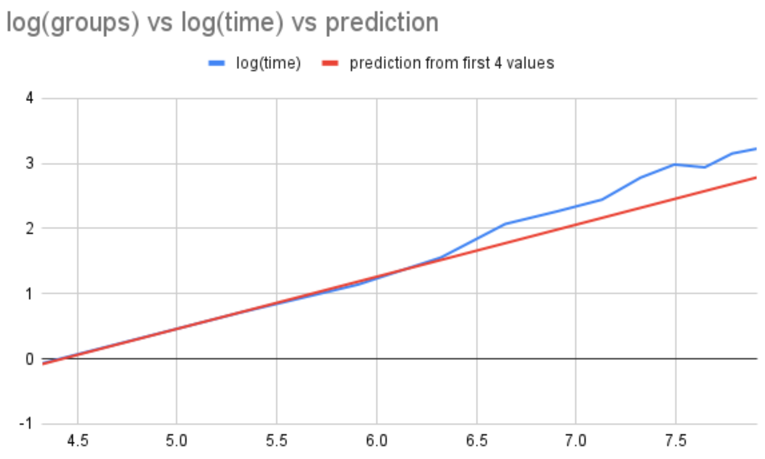

Figure 5.3: log log analysis of implementation using library

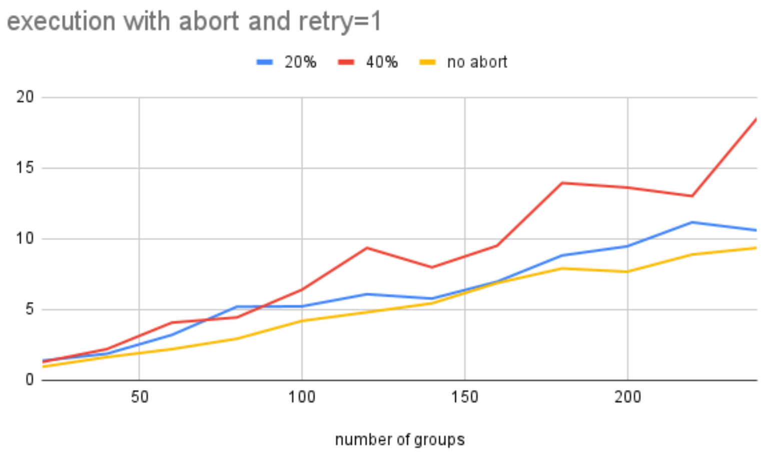

Figure 5.4: execution time of implementation using library and programmed aborts

by-case basis. The library provides an effective abstraction for easy design and implementation of consensus scenarios in general. While there is a performance difference due to the additional setup required in the library, it is observed that execution time for both implementations grows similarly with the scaling number of groups. The time of execution grows linearly with the increasing number of groups.

Figure 5.3 shows the log-log analysis of the implementation using the library. As observed, the overall analysis shows a slope of 0.965. However, some deviation is observed with a higher number of groups. To demonstrate it, predictions made from the first four readings and the actual readings are compared against it. This deviation could be attributed to the higher number of goroutines additionally required for each group as well as its blocking communication with each process goroutines. This could also be contributed to by the data structures such as *slice* used in the implementation. In [3] the performance deteriorated exponentially with the increasing number of processes. However, the scheduler defined in [3] chose actions randomly or driven by communication requirements. The approach here brings an improvement through grouping processes according to a protocol and executing them only after they are put into groups.

when aborts are programmed in the processes and the protocol is defined to retry processes after aborts, the effective number of groups increases. Aborted processes are grouped again, creating more goroutines. This also increases the time of execution. As observed in figure5.4, the time of execution increases as the percentage of abort increases. This is due to the restarting of processes, and regrouping them. This spawns a new goroutine for Group Monitors and process executions.

5.3 Usability

To demonstrate usability, the library is used to implement a scenario similar to SNO where all parties involved require something from peers. Consider a scenario involving a group of people who can donate organs and require donations too. A group of people can be formed where everyone benefits from the group, creating a circle of dependencies. One person may donate to another, but receive a donation from a third member of the group. Following the interactive consensus behavior, the group size can be dynamic in this scenario. However, in the design of this library, the protocol defines the size of each group.

To model this scenario without the dynamic group size, we use the library to define a protocol with some processes with role *person* and a process labeled *matcher* in every group. Here the number of persons in a group is set to 3.

```
organMatchingProtocol := lib.NewProtocol(
    map[string]uint{"matcher": 1, "person": 3},
    []string{"matcher", "person"},
    0, //retry count
)
```

We also define a *Person* object to pack the details of each person. If a group is formed and requirements are met, the details of the group members can be saved in this object.

```
type Person struct {
    Id        string
    Needs     string
    CanDonate string
    Matches   *string
}
```

When a group is created, the Matcher can receive information from all members in the group and keep track of what donations are required, and what donations are possible within the group. This information can be used to validate if all required donations are being satisfied within the group. The matcher process could be designed as :

```
func(p *lib.Process) {
    //keep track of donations possible
    offers := make(map[string]int)
    //keep track of donations required
    require := make(map[string]int)
    //list of all peers
    peers := []string{}
```

```
        for i:=0; i< groupsize; i++ {
            message, err := p.ReadMessage()
            if err != nil {
                return
            }

            // get peerId, requirement
            // and donation from message
            peerID := message["id"].(string)
            donation := message["canDonate"].(string)
            requirement := message["need"].(string)

            peers = append(peers, peerId)
            offers[donation] += 1
            require[requirement] += 1
        }

        for r, num = range require {
            if offers[r] < num{
                p.Fail("requirements do not match donations")
                return
            }
        }
    }
```

The process labeled for a person then only needs to send its requirement and donation details to the matcher process. The *person* process can then wait for a message from the matcher informing about all members in the group. If the group cannot satisfy all requirements, the matcher will use *Fail* method for a programmed abort. This will also kill the *person* processes using context within a process.

```
    func(p *lib.Process, person *Person) {
        p.SendToPeersByType("matcher", lib.Message{
            "id":        person.Id,
            "canDonate": person.CanDonate,
            "need":      person.Needs,
        })

        message, err := p.ReadMessage()
        if err != nil {
            return
        }

        matchField := message["matches"]
        matchesSlice := matchField.([]string)
        matches := strings.Join(matchesSlice, ";")
        person.Matches = &matches
    }
```

It has to be noted that the group size is not dynamic in this example. However, the same processes can be used with different monitors, and thus different protocols to find a larger group that can agree with each other.

Chapter 6

Conclusion

Consensus is common in all systems involving concurrent processing. It can be a widely distributed system or an interactive application with multiple concurrent processes to provide responsiveness. This variety of scenarios of concurrency requires different approaches. A system has multiple choices for approach regarding communication, and synchronization among such concurrent threads. The programming languages currently available decide to prioritize one method for communication as well as synchronization capabilities. Consensus is not supported by programming languages and is instead left to the programmer to implement.

This dissertation studies the abstraction for concurrent and transactional processes provided by the work of Koutavas et al [3]. The introduced abstraction is implemented in the Go programming language. Go is a popular programming language with inbuilt support for concurrency and is used widely in distributed system development. The library implemented in this dissertation provides abstractions for easy implementation of concurrent processes and modeling consensus scenarios among them. It enables to design of the processes in a transactional approach. The library handles the complexities of monitoring, communication, and the commit or rollback of processes. The library also handles any runtime panics within the designed processes and uses one goroutine per group to monitor its status. This library used synchronous message passing through channels, as well as shared mutex for its implementation.

The challenges posed by the concurrency and communication functionalities in Golang, such as goroutines and channels are discussed. The performance of the library is evaluated against a case-specific, from-scratch implementation of a consensus scenario. It is observed that there is a performance deficit due to the extra goroutines and communication abstracted away by the library. The implementation using the library also scales almost linearly to a scaling number of processes.

6.1 Future work

Future work is to focus on enabling a dynamic grouping of processes. Within the current implementation, the groups are created by protocols, and protocols strictly define the number of processes and their roles in a group. This restricts the library from being used for a more generic and interactive problem like SNO where the group can grow if more willing participants are available. Further improvements can be made by giving more abstractions to steps detailed in the guidelines of the library. Cleaner abstractions for avoiding deadlocks and sending messages can be introduced. The latest version of Go also introduced generics. This has to be explored to provide a better abstraction to define a process. A proper analysis of the data structures used, and optimizations will be useful in improving the performance generally, as well as when dealing with a larger number of groups and processes.

Bibliography

[1] Herlihy, M., Shavit, N.: The art of multiprocessor programming. Kaufmann (2008)

[2] Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2008)

[3] Spaccasassi, C., Koutavas, V. (2014). Towards Efficient Abstractions for Concurrent Consensus. In: McCarthy, J. (eds) Trends in Functional Programming. TFP 2013. Lecture Notes in Computer Science, vol 8322. Springer, Berlin, Heidelberg.

[4] Reppy, J. H. (1999). Concurrent programming in ML. Cambridge University Press (1999)

[5] Panangaden, P., Reppy, J. (1997). The Essence of Concurrent ML. In: Nielson, F. (eds) ML with Concurrency. Monographs in Computer Science. Springer, New York, NY.

[6] de Vries, E., Koutavas, V., Hennessy, M.: Communicating Transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583.

[7] Lamport, L., 2001. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pp.51-58.

[8] Ongaro, D. and Ousterhout, J., 2014. In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference (Usenix ATC 14) (pp. 305-319).

[9] Choudhary, N.,Effective Abstraction for Transactional Concurrent Consensus(Elixir), 2021.

[10] Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: POPL. pp. 195–208.

[11] Ziarek, L., Schatz, P., Jagannathan, S.: Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In: Reppy,

J.H., Lawall, J.L. (eds.) ICFP. pp. 136–147. ACM, NY (2006) ACM, NY (2005)

[12] "Context - Documentation", Go standard library, August 2, 2022, [Online], Available: https://pkg.go.dev/context [Accessed 18 - August - 2022]

[13] "Reflect - Documentation", Go standard library, August 2, 2022, [Online], Available: https://pkg.go.dev/reflect [Accessed 18 - August - 2022]

Appendix A

Source Code

The repository containing the source code of the implementation and examples demonstrating usage is available at:
https://github.com/Brotchu/concurrentConsensus