Decentralized Group Messenger with Causally Ordered Multicast

Ahmet Uysal
auysal16@ku.edu.tr

Date: 15th of March

Contents

1 Introduction

A decentralized unstructured peer-to-peer group messenger application is implemented based on Causally Ordered Multicasting algorithm using Go programming language and RPC protocol. AWS EC2 instances are used as the distributed platform.

1.1 Assumptions

- Multicast group size is static.
- Every peer has a file that contains addresses of all peers.
- Peers are assumed to function correctly once the initial connection is established. Therefore, the system does not implement any fault tolerance mechanisms.

2 Implementation

2.1 `MessageDTO` Struct

`MessageDTO` struct is how the messages are represented in my implementation. It consists of the identifier (IP address + port) of the sender, message timestamp, and the message string.

```go
type MessageDTO struct {
    Transcript string
    OID        string
    TimeStamp  []int
}
```

Code snippet 1: `MessageDTO` struct

`MessageDTO` struct also implements a method that checks whether a message should be delivered to application based on the Causally Ordered Multicasting algorithm. This method returns true if $ts(m)[sender] = VC_{receiver}[sender] + 1$ and $ts(m)[k] \leq VC_{receiver}[k]$ for all $k \neq sender$, and false otherwise.

```go
func (message MessageDTO)
    shouldDeliverMessageToApplication(vectorClock []int,
    senderProcessIndex int) bool {
    if message.TimeStamp[senderProcessIndex] !=
        vectorClock[senderProcessIndex]+1 {
        return false
    }
    for k, time := range vectorClock {
        if k == senderProcessIndex {
            continue
        }
        if message.TimeStamp[k] > time {
            return false
        }
    }
    return true
}
```

Code snippet 2: `shouldDeliverMessageToApplication` Implementation

2.2 `ConcurrentMessageSlice` Struct

Decentralized group messenger needs to postpone the delivery of the messages that does not satisfy the requirements of `shouldDeliverMessageToApplication` method 2. In order to implement this, `MessengerProcess` structs need store queued messages in a data structure. However, Go Slices does not handle concurrency by themselves and mutexes are required to organize concurrent access and update scenarios. `ConcurrentMessageSlice` struct is created for this purpose. Implementation is based on a blog post by M. Nikolov. [1]

```
type ConcurrentMessageSlice struct {
    sync.RWMutex
    messages []MessageDTO
}
func (cms *ConcurrentMessageSlice) append(message MessageDTO) {
    cms.Lock()
    defer cms.Unlock()
    cms.messages = append(cms.messages, message)
}
// Note: Removal of messages is implemented in MessengerProcess
```

Code snippet 3: `ConcurrentMessageSlice` Struct

2.3 `MessengerProcess` Struct

`MessengerProcess` struct is the main type that is responsible for delivery and interpretation of `MessageDTO` instances. It represents a peer process that is connected to Decentralized Group Messenger service. Each `MessengerProcess` stores the identifier (IP address + port) of the peer, current vector clock of the peer, queued messages that are waiting for to be delivered to the application, and a map that relates identifiers of the peers to indices of the vector clock.

```
type MessengerProcess struct {
    OID                 string
    QueuedMessages      ConcurrentMessageSlice
    VectorClock         []int
    VectorClockIndexMap map[string]int
}
```

Code snippet 4: `MessengerProcess` Struct

2.3.1 `PostMessage` RPC Method

`PostMessage` RPC method allows peers to transmit `MessageDTO` instances using RPC and handles received `MessageDTO` instances on peers. Go "net/rpc" package requires methods to follow a specific signature structure to be registered as an RPC call. In order to make a method publicly available to RPC calls, a method needs to:

- Have a type that is exported (`MessengerProcess` type is exported)
- Method itself is exported
- Method has two arguments, both exported (or builtin) types.
- The second argument is a pointer

- Method has the return type `error`

`PostMessage` is the only method of `MessengerProcess` type that satisfies all these requirements, and therefore it is can be registered. All other functions are helpers to implement message handling and does not need to be registered as RPC calls. RPC registration is implemented in the `main` function as shown below.

```
me := MessengerProcess{
    OID: myAddress,
    QueuedMessages: ConcurrentMessageSlice{
        RWMutex:  sync.RWMutex{},
        messages: nil,
    },
    VectorClock:        vectorClock,
    VectorClockIndexMap: peers,
}
_ = rpc.Register(&me)
```

Code snippet 5: RPC call registration

`PostMessage` method checks the received message for the requirements of being delivered to application. If the message satisfies the condition, it is delivered to application and the vector clock of the `MessengerProcess` is updated. Otherwise, the message is appended to the message queue. If a message is delivered to application and the vector clock is updated, `PostMessage` checks the message queue for messages that satisfy the delivery conditions based on the updated vector clocks. This procedure continues until there are no messages in the queue that satisfy delivery condition after a vector clock update. The `PostMessage` function implementation is given below along with `shouldDeliverMessageToApplication`, `deliverMessageToApplication`, and `popNextMessageToDeliver` helper methods.

```
func (messengerProcess *MessengerProcess) PostMessage(message
    MessageDTO, isSuccessful *bool) error {
    if messengerProcess.shouldDeliverMessageToApplication(message){
        messengerProcess.deliverMessageToApplication(message)
        // if a message is delivered vector clock is updated
        // we need to check queued messages
        nextMessageToDeliver, shouldDeliver :=
            messengerProcess.popNextMessageToDeliver()
        for shouldDeliver {
            messengerProcess
                .deliverMessageToApplication(nextMessageToDeliver)
            nextMessageToDeliver, shouldDeliver =
                messengerProcess.popNextMessageToDeliver()
        }
    } else {
        // add message to queue
        messengerProcess.QueuedMessages.append(message)
    }
    *isSuccessful = true
    return nil
}
```

Code snippet 6: `PostMessage` RPC method

2.3.2 Helper Methods

`shouldDeliverMessageToApplication` is just a wrapper method around the `MessageDTO`'s `shouldDeliverMessageToApplication` method.

```
func (messengerProcess MessengerProcess)
    shouldDeliverMessageToApplication(message MessageDTO) bool {
    i := messengerProcess.VectorClockIndexMap[message.OID]
    return message.shouldDeliverMessageToApplication(
        messengerProcess.VectorClock, i)
}
```

Code snippet 7: `shouldDeliverMessageToApplication` helper method

`deliverMessageToApplication` updates the vector clock of the `MessengerProcess` and prints the received message to standard output.

```
func (messengerProcess *MessengerProcess)
    deliverMessageToApplication(message MessageDTO) {
    // update the vector clock
    for index, time := range messengerProcess.VectorClock {
        var maxTime int
        if time > message.TimeStamp[index] {
            maxTime = time
        } else {
            maxTime = message.TimeStamp[index]
        }
        messengerProcess.VectorClock[index] = maxTime
    }
    fmt.Printf("%s: %s\n", message.OID, message.Transcript)
}
```

Code snippet 8: `deliverMessageToApplication` helper method

`popNextMessageToDeliver` iterates over queued messages, pops and returns the first message that satisfies the delivery condition. It also returns a second flag value to indicate whether it found a message that satisfy the conditions.

```
func (messengerProcess *MessengerProcess) popNextMessageToDeliver() (MessageDTO, bool) {
    var nextMessageToDeliver MessageDTO
    var nextMessageToDeliverIndex int
    messengerProcess.QueuedMessages.Lock()
    defer messengerProcess.QueuedMessages.Unlock()
    \\ iterate over queued messages until a match is found
    for index, queuedMessage := range messengerProcess.QueuedMessages.messages {
        if messengerProcess.shouldDeliverMessageToApplication(queuedMessage) {
            nextMessageToDeliver = queuedMessage
            nextMessageToDeliverIndex = index
            break
        }
    }
    // no message satisfies the condition
    if nextMessageToDeliver.OID == "" {
        return MessageDTO{}, false
    }
    messengerProcess.QueuedMessages.messages =
          append(messengerProcess.QueuedMessages.messages[:nextMessageToDeliverIndex],
        messengerProcess.QueuedMessages.messages[nextMessageToDeliverIndex+1:]...)
    return nextMessageToDeliver, true
}
```

Code snippet 9: `popNextMessageToDeliver` helper method

2.4 Execution Flow of the Program

The program starts by getting its own IP address and reads the peers.txt file. It gives an error and exits if it couldn't find its own identifier inside this file. Every `MessengerProcess` starts with a vector clocks consisting of all zeros. Then, it registers the `PostMessage` method to RPC as explained in the previous sections. A goroutine is created to continuously listen for incoming RPC connections. Each `MessengerProcess` then waits until all peers listed inside peers.txt file is available by continously trying to connect to each peer until it responds. After the connection is established, it informs the user and starts to accept message inputs. Vector clock of the `MessengerProcess` is increased by one only on message send events. Messages are multicasted by RPC calls to each peer.

3 Deployment to AWS EC2 Instances

Decentralized Group Messenger implementation is tested with five peers using AWS EC2 instances. I used PuTTY to transfer source files and connect to each client over SSH connection.

Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
i-0097952516c871711	t2.micro	us-east-1b	running	2/2 checks ...	None	ec2-52-206-76-106.co...	52.206.76.106
i-04b332a0d2eed04...	t2.micro	us-east-1b	running	2/2 checks ...	None	ec2-54-172-5-91.comp...	54.172.5.91
i-05132f1a1488f2b8f	t2.micro	us-east-1b	running	2/2 checks ...	None	ec2-18-208-107-40.co...	18.208.107.40
i-06245518336700e...	t2.micro	us-east-1b	running	2/2 checks ...	None	ec2-54-234-28-3.comp...	54.234.28.3
i-06fb2213cef410e7e	t2.micro	us-east-1b	running	2/2 checks ...	None	ec2-3-87-4-159.comput...	3.87.4.159

Figure 1: Five AWS EC2 instances that are used in the deployment phase.

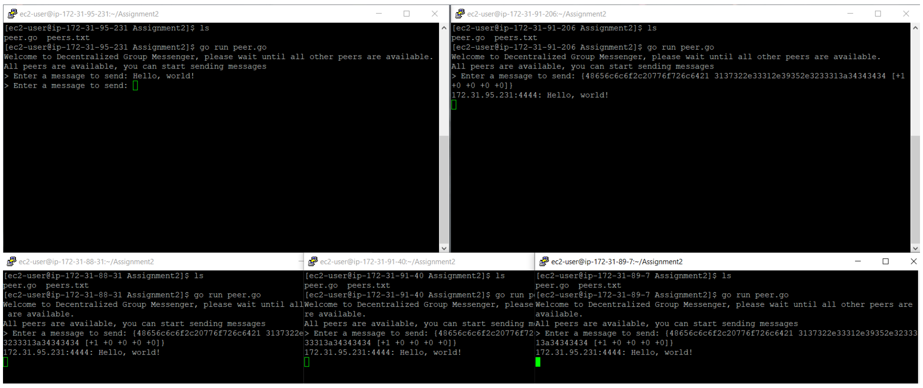

Figure 2: Initial connection and messaging between group of five peers running on AWS EC2 instances

4 Possible Message Delivering Scenarios

In order to illustrate different scenarios for message delivery, I added a random delay between 0-5 seconds to each message transfer in `postMessageToPeer` function. Since this function is called in seperate goroutines for each peer, sender will send the messages to each peer in a random order with different delays. I also added some parts to print out the received message and vector clock information at the time of the event. On my trials, every peer delivered the message to application in the correct order.

```go
func postMessageToPeer(peerAddress string, message MessageDTO) {
    randomDelay := rand.Float64() * 5000
    time.Sleep(time.Duration(randomDelay) * time.Millisecond)
    peerRpcConnection, err := rpc.Dial(CONNECTION_TYPE,
        peerAddress)
    handleError(err)
    var isSuccessful bool
    _ = peerRpcConnection.Call("MessengerProcess.PostMessage",
        message, &isSuccessful)
    _ = peerRpcConnection.Close()
}
```

Code snippet 10: `postMessageToPeer` helper function

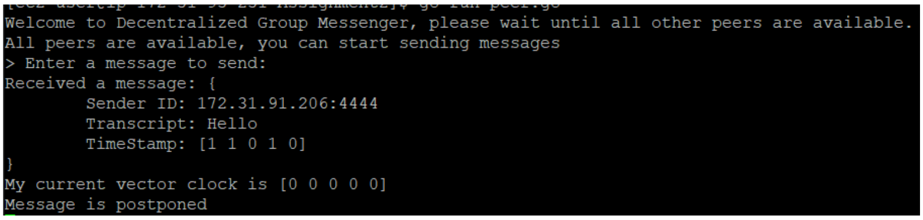

Figure 3: A message that is postponed since it does not satisfy the condition.

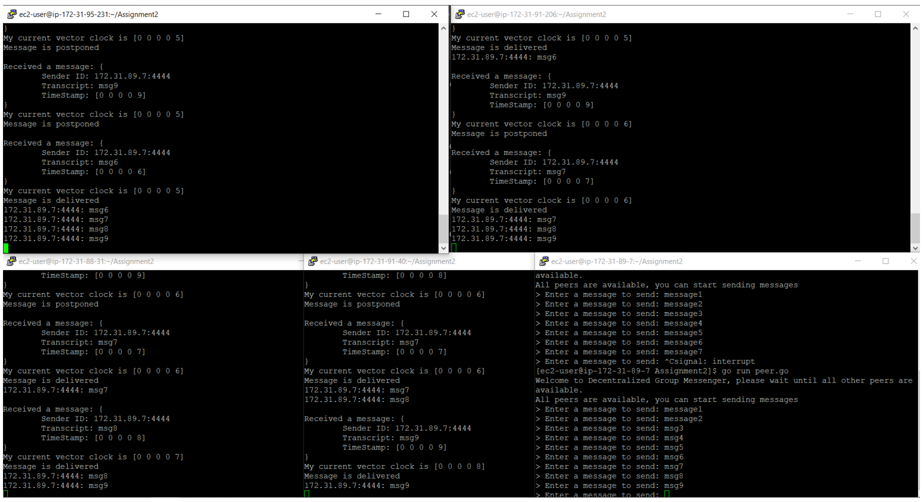

Figure 4: Sending of 9 consequent messages with random delays.

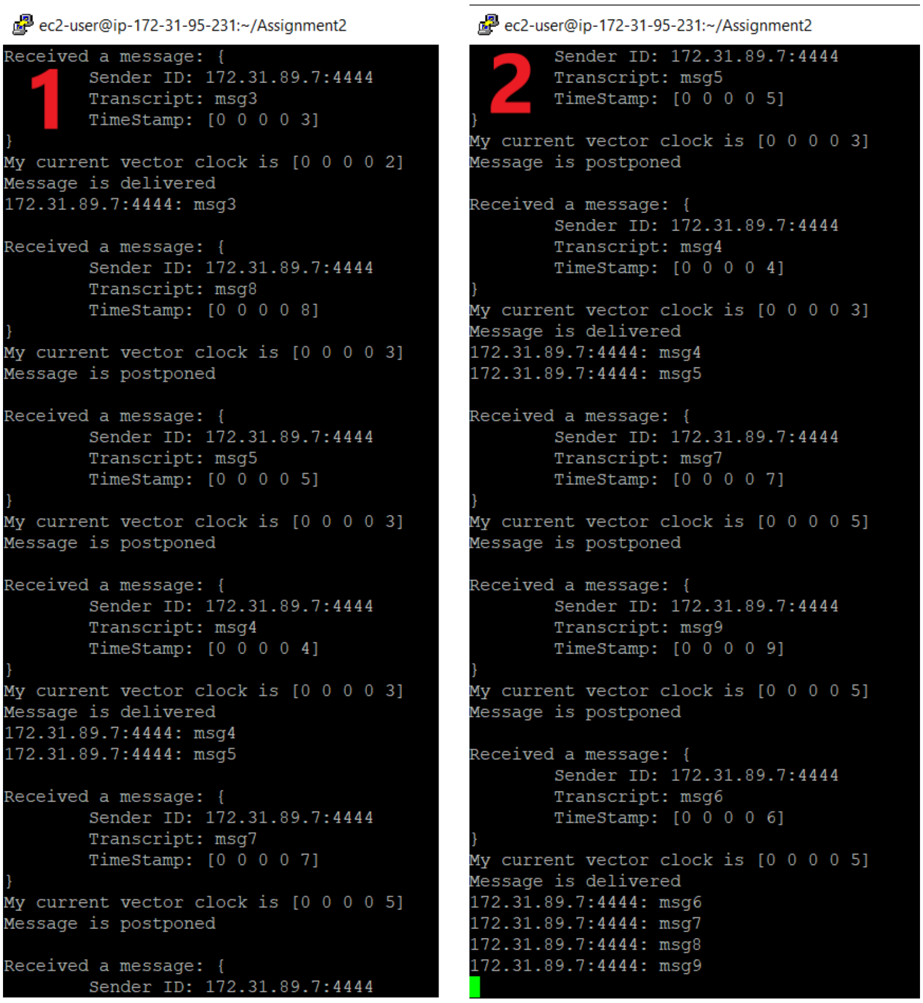

Figure 5: Peer 1 Delivering the messages in the correct order

References

[1] Concurrent map and slice types in Go. Retrieved March 15, 2020, from https://dnaeon.github.io/concurrent-maps-and-slices-in-go/