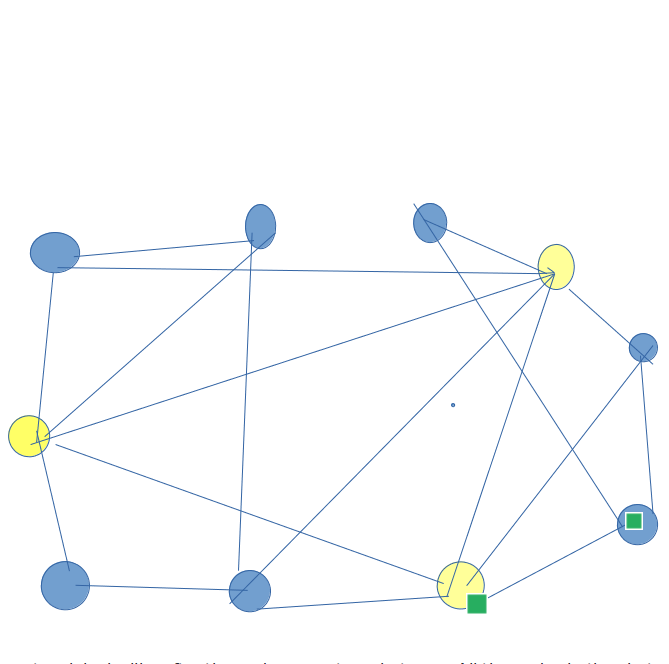


This is how the network looks like after a node joins the Kademlia network. We're the yellow node on the lower right. Yellow nodes are participants of a chatroom.



This is how the network looks like after the node requests a chatroom. All the nodes in the chatroom are now connected directly and separately from the Kademlia network.

**Chat Node Structure**

Type node struct {

logger \*zap.Logger

host libp2phost.Host

kadDHT \*dht.IpfsDHT

bootstrapOnly bool

storeIdentity bool

ps \*pubsub.PubSub

roomManager \*RoomManager

eventPublishers []events.Publisher

eventPublishersLock sync.RWMutex

nicknameStoreMutex sync.Mutex

nicknameStoreWaiting bool

}

**API supported by the chat node**

rpc Ping(PingRequest) returns (PingResponse);

rpc SendMessage(SendMessageRequest) returns (SendMessageResponse);

rpc GetNodeID(GetNodeIDRequest) returns (GetNodeIDResponse);

rpc SetNickname(SetNicknameRequest) returns (SetNicknameResponse);

rpc GetNickname(GetNicknameRequest) returns (GetNicknameResponse);

rpc JoinRoom(JoinRoomRequest) returns (JoinRoomResponse);

rpc GetRoomParticipants(GetRoomParticipantsRequest) returns (GetRoomParticipantsResponse);

rpc SubscribeToEvents(SubscribeToEventsRequest) returns (stream Event);

**API Request Response Structure**

a)

message PingRequest {}

message PingResponse {}

b)

message SendMessageRequest {

string room\_name = 1;

string value = 2;

}

message SendMessageResponse {

bool sent = 1;

}

c)

message ChatMessage {

string sender\_id = 1;

int64 timestamp = 2;

string value = 3;

}

d)

message ModeratorMessage {

string sender\_id = 1;

int64 timestamp = 2;

string value = 3;

}

e)

message BlockMessage {

string sender\_id = 1;

int64 timestamp = 2;

string value = 3;

}

f)

message BanMessage {

string sender\_id = 1;

int64 timestamp = 2;

string value = 3;

}

g)

message RateLimitMessage {

string sender\_id = 1;

int64 timestamp = 2;

string value = 3;

}

h)

message GetNodeIDRequest {}

message GetNodeIDResponse {

string id = 1;

}

i)

message SetNicknameRequest {

string room\_name = 1;

string nickname = 2;

}

message SetNicknameResponse {}

j)

message GetNicknameRequest {

string room\_name = 1;

string peer\_id = 2;

}

message GetNicknameResponse {

string nickname = 1;

}

k)

message JoinRoomRequest {

string room\_name = 1;

string nickname = 2;

}

message JoinRoomResponse {}

l)

message RoomParticipant {

string id = 1;

string nickname = 2;

}

message GetRoomParticipantsRequest {

string room\_name = 1;

}

message GetRoomParticipantsResponse {

repeated RoomParticipant participants = 1;

}

// Room holds room event and pubsub data.

type Room struct {

name string

topic \*pubsub.Topic

subscription \*pubsub.Subscription

// Map is counter of moderation messages for user in a chat room

usermoderationCounter sync.Map

ratelimitstore ratelimit.Store

lock sync.RWMutex

participants map[peer.ID]\*participantsEntry

}

// RoomManager manages rooms through pubsub subscription and implements room operations.

type RoomManager struct {

logger \*zap.Logger

ps \*pubsub.PubSub

node Node

kadDHT \*dht.IpfsDHT

bannedIPlist \*block.Blocklist

moderatorConfig \*vec.Model

rooms map[string]\*Room

eventPublisher events.Publisher

lock sync.RWMutex

}

// RoomMessageOut holds data to be published in a topic.

type RoomMessageOut struct {

Type RoomMessageType `json:"type"`

Payload interface{} `json:"payload,omitempty"`

}

**Architecture**

// SendChatMessage sends a chat message to a given room.

// Fails if it has not yet joined the given room.

func (r \*RoomManager) SendChatMessage(ctx context.Context, roomName string, msg entities.Message) error {

room, found := r.getRoom(roomName)

if !found {

return errors.New(fmt.Sprintf("must join the room before sending messages"))

}

rm := &RoomMessageOut{

Type: RoomMessageTypeChatMessage,

Payload: msg,

}

if err := r.publishRoomMessage(ctx, room, rm); err != nil {

return err

}

return nil

}

rmJSON, err := json.Marshal(rm)

if err != nil {

return errors.Wrap(err, "marshalling message")

}

if err := room.topic.Publish(ctx, rmJSON); err != nil {

return err

}

Output Message is marshalled in Json and published to chatroom topic

// RoomMessageIn holds data to be received from a topic.

//

// The Payload field is lazily unmarshalled because it depends on the type of message published.

type RoomMessageIn struct {

Type RoomMessageType `json:"type"`

Payload json.RawMessage `json:"payload,omitempty"`

}

After receiving message from subscription channel, it is unmarshalled in room message input,

message category is derived.

subMsg, err := room.subscription.Next(context.Background())

if err != nil {

r.logger.Error("failed receiving room message", zap.Error(err))

continue

}

if subMsg.ReceivedFrom == r.node.ID() {

continue

}

var rm RoomMessageIn

if err := json.Unmarshal(subMsg.Data, &rm); err != nil {

r.logger.Warn("ignoring room message", zap.Error(err))

}

**Messaging Layer**

1)New Chat Message

2)New Moderate Message

3)New Block Message

4)New Ban Message

5)New Rate limit Message

6)New Advertising Message

7)Reaction Message

There is handler for each different type of message.

When message is received -> different Algorithms play a role as illustrated below -

**DDOS/Malicious Attack Prevention Algorithm**

First Banned IP list role comes into play

This Banned IP list is loaded in segment tree, for faster querying based on IP range.

If p2p message is coming from one of the Banned IP Addresses, then peer is banned and message from this peer will automatically be dropped.

**Messaging cooling period Algorithm**

If user is posting too many messages in short period, user is rate limited using token bucket Algorithm as per cooling period algorithm.

**Content Moderation Algorithm**

After analysing message – following categories are determined which need moderation

**Negative Categories**

1) Spam

2) Abuse

3) Adult content

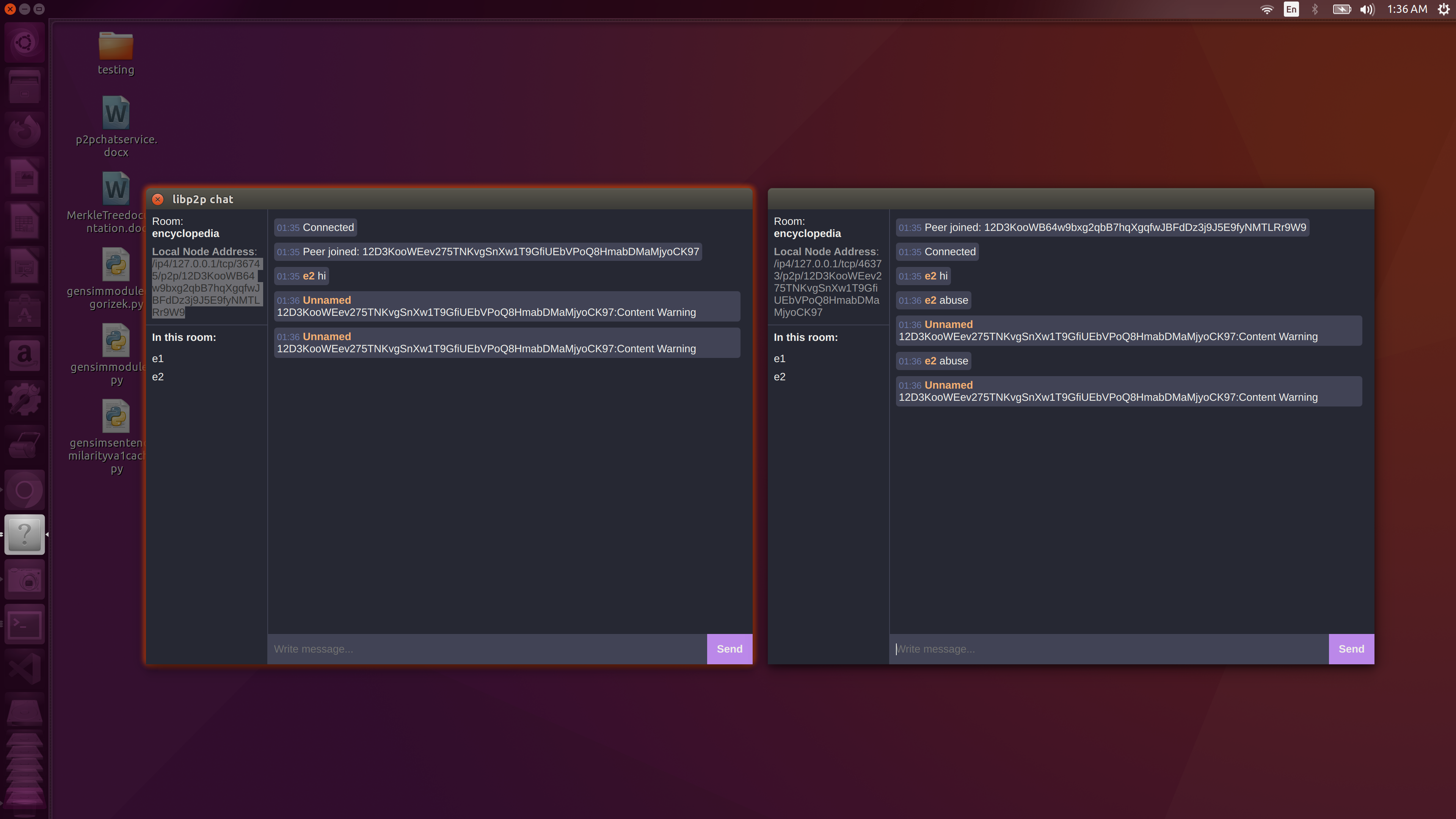
**Positive Categories**

1)Highly relevant – quality content

2)Less relevant

3)Fun messages

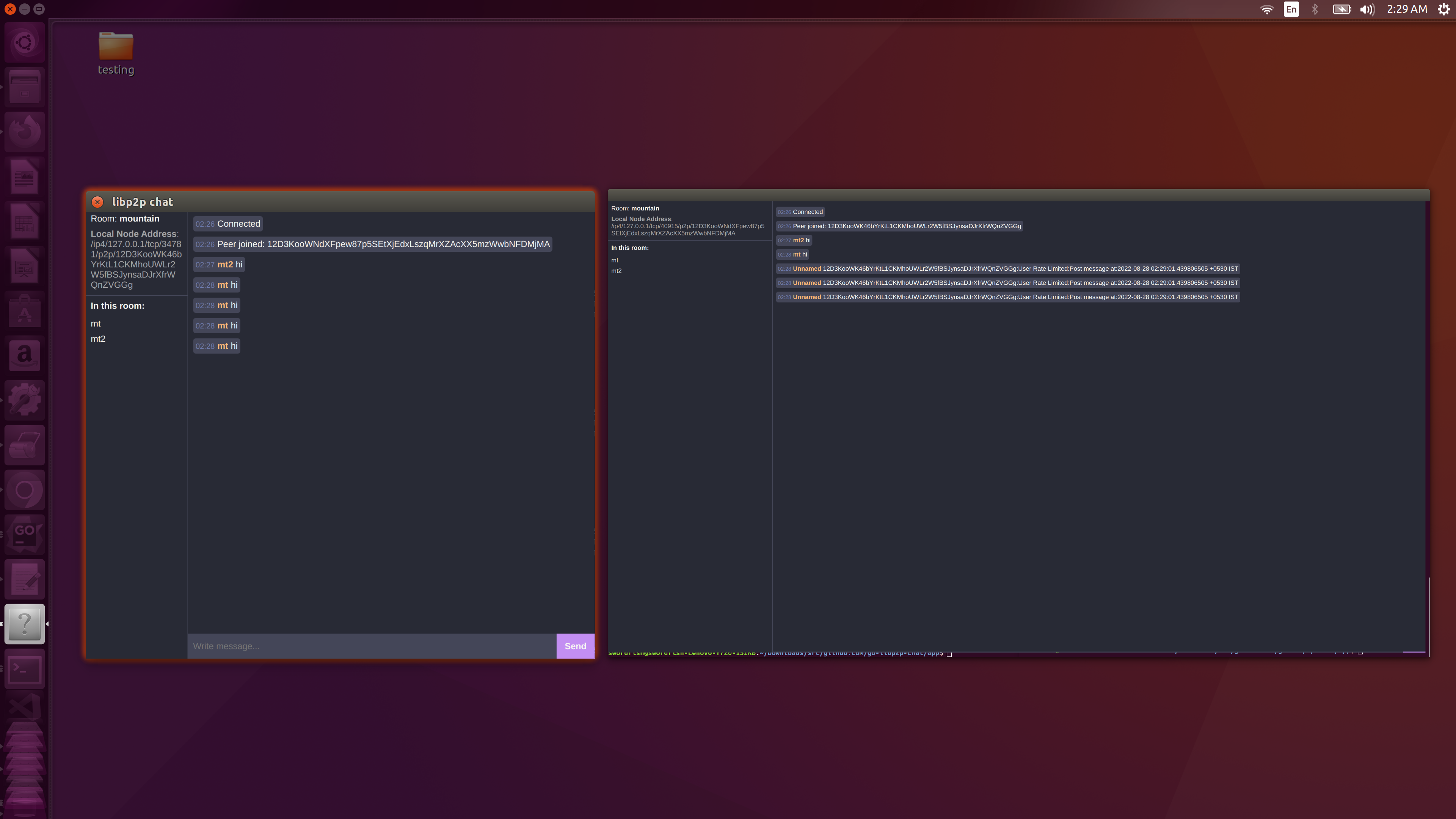
Moderator is implemented within node and is implemented using ML Model, v

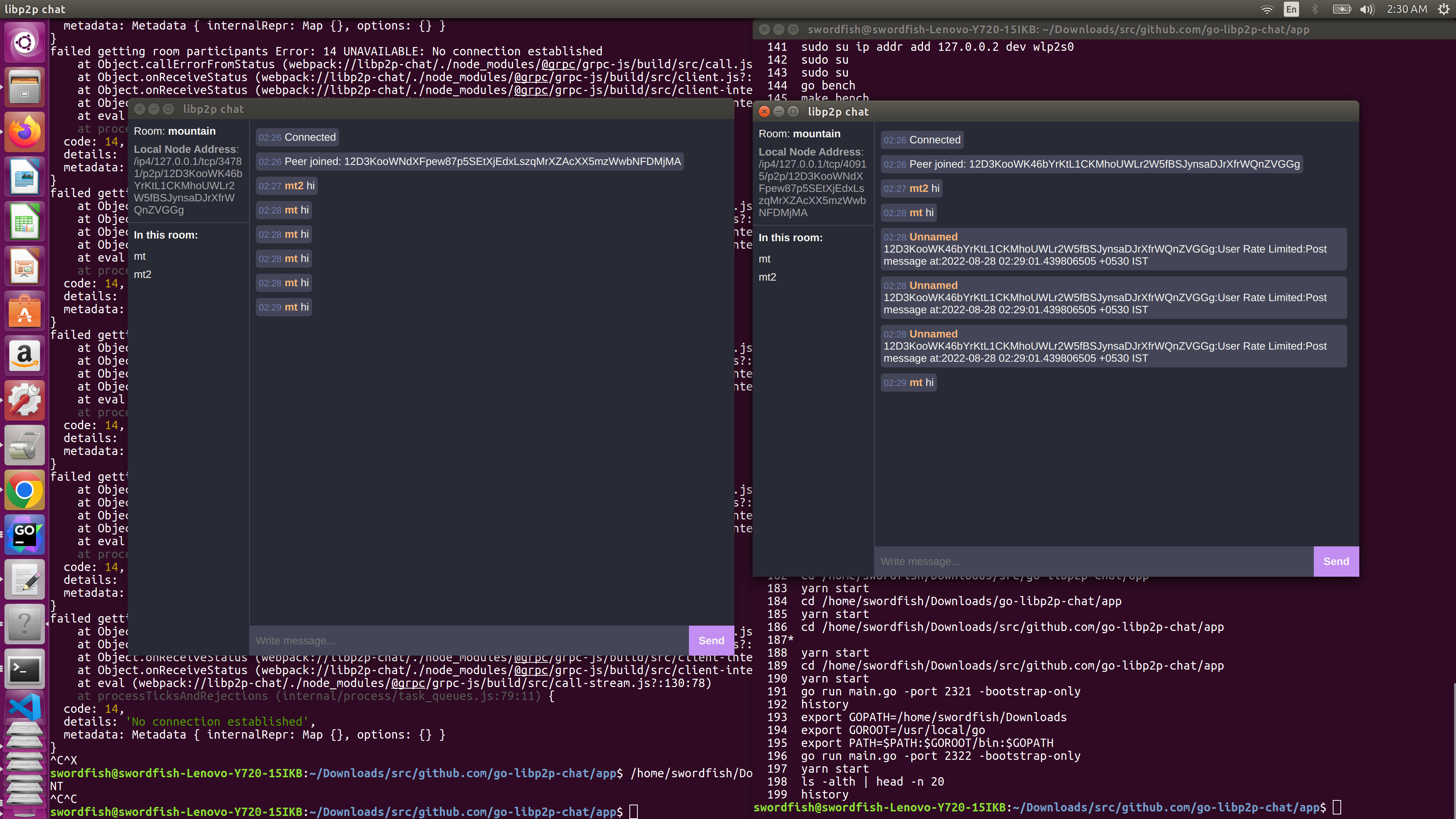


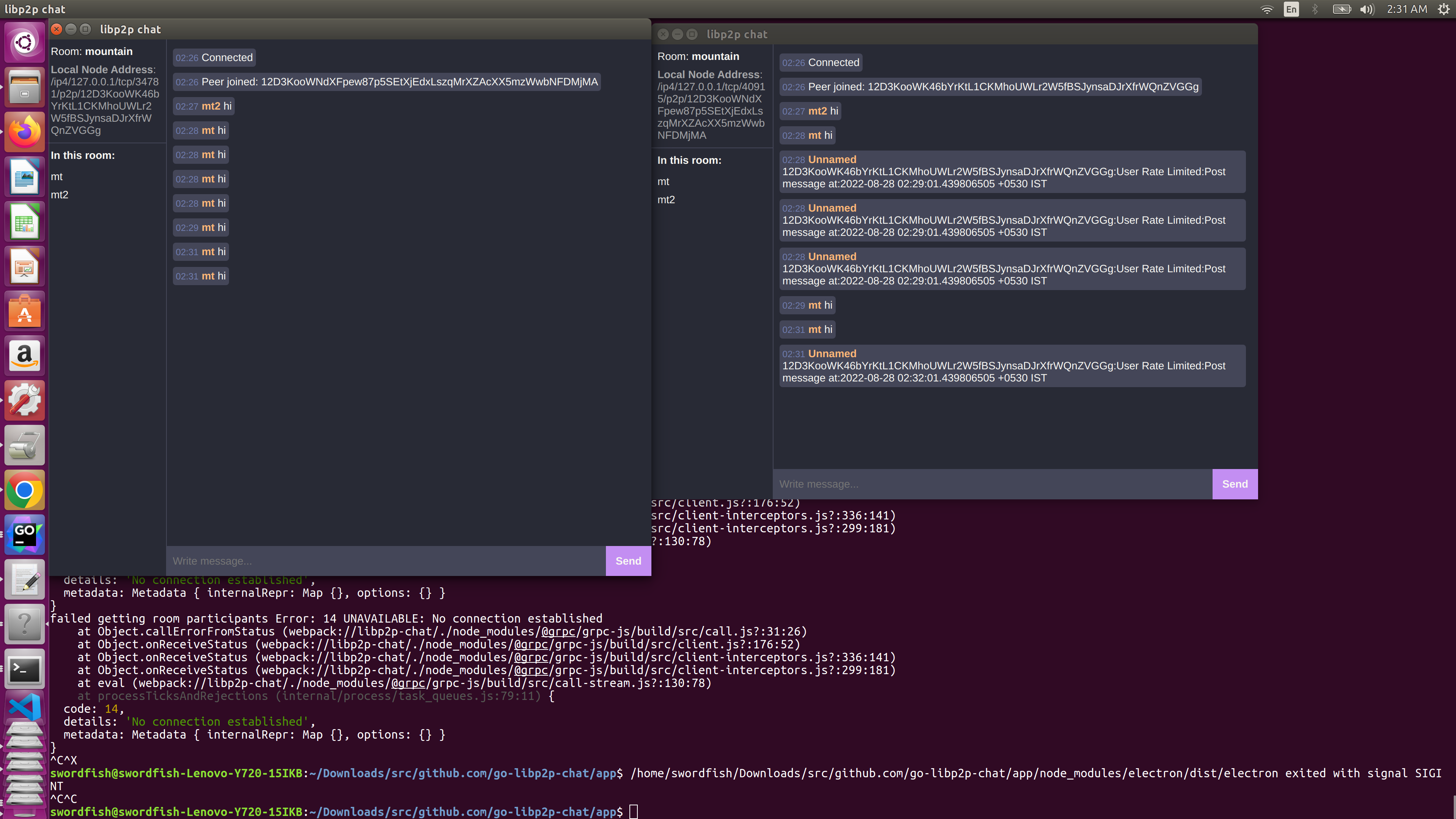
**Content moderation warning posted by moderation bot.**

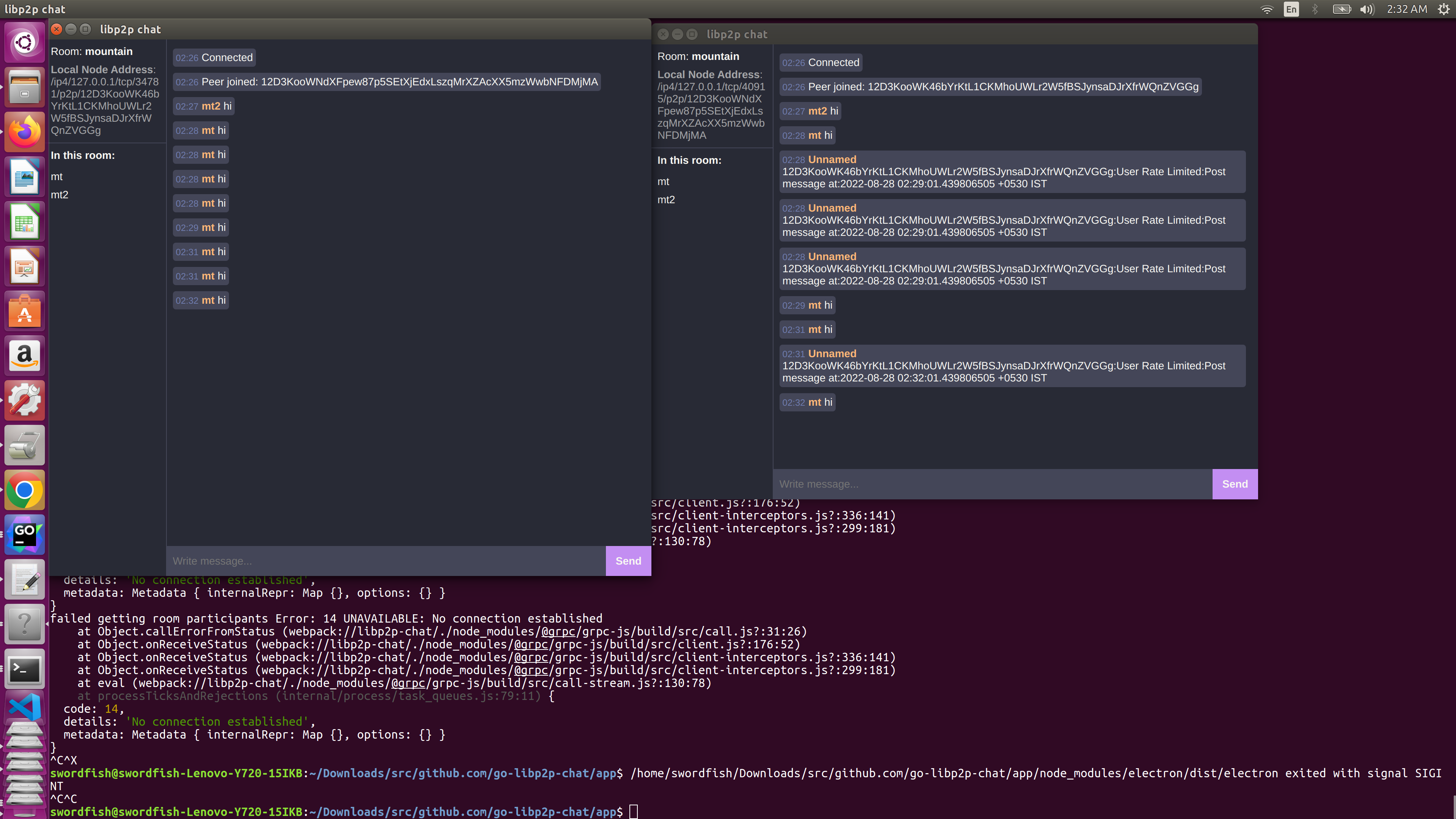
**Abusive Messages are blocked and not reachable to other user.**

**Cooling Period Feature demo in p2p chat service**









**Refreshing participants Algorithm**

**Categories**

1)Inactivity

2)Unblock

3)Unban

Role of different TTL come into play

roomParticipantsTTL = time.Second \* 300

roomBlockTTL = time.Hour \* 12

roomBanTTL = time.Hour \* 168

Different Maps are maintained for Normal Participants, Blocked Participants, Banned Participants.

If user is inactive for roomParticipantsTTL, user will automatically be evicted from the topic room.

If user is in blocked participant list and block ttl has expired, user will automatically be unblocked.

If user is in banned participant list and banned ttl has expired, user will automatically be unbanned.

Refreshing Participants Algorithm for each of the category -

It is implemented using longer running go routines for each of the categories during life cycle of p2p chat session for a particular topic room.

**User alias feature**

User can use fancy nicknames or user handle for display at the time of chat

**User Message history and P2P chat session storage and recovery/resumption in peers**

Bolt db is used for serialisation of p2p chat session and p2p chat session resumption in case peer goes offline temporarily.

**Bolt DB Key value specification**

// Spec contains fields for serialising topic information

type Topic struct {

ChatMessage string

ModeratorMessage string

AdvertisingMessage string

ReactionMessage string

UserList []string

BlockuserList []string

BanuserList []string

}

// Keys for the persistent storage.

// Chat room data storage

var TopicKeys = struct {

ChatMessage []byte

ModeratorMessage []byte

AdvertisingMessage []byte

ReactionMessage []byte

UserList []byte

BlockuserList []byte

BanuserList []byte

}{

ChatMessage: []byte("chatmessage"),

ModeratorMessage: []byte("moderatormessage"),

AdvertisingMessage: []byte("advertisingmessage"),

ReactionMessage: []byte("reactionmessage"),

UserList: []byte("userlist"),

BlockuserList: []byte("blockuser"),

BanuserList: []byte("banuser"),

}

**Storing User semantics for P2P CHAT sessions to be plugged in analytics engine**.

// Keys for User semantics

// Peer semantics data storage

var UserKeys = struct {

Geography []byte

UserContentCategories []byte

UserPersonality []byte

AverageUserSessionLength []byte

UserFrequency []byte

PeerId []byte

UserHandle []byte

}{

Geography: []byte("geography"),

UserContentCategories: []byte("usercontentcategories"),

UserPersonality: []byte("userpersonality"),

AverageUserSessionLength: []byte("averageusersessionlength"),

UserFrequency: []byte("userlist"),

}

type User struct {

Geography string

UserContentCategories []string

UserPersonality string

AverageUserSessionLength int

UserFrequency string

PeerId string

UserHandle string

}

Geography, Internet bandwidth is derived from IP Address present in node boostrapped address using maxmind databases for classfying peer in normal and light (very limited frequency) peer.

User content categories and User Personality are derived using saved user chat session data using ML models.

Average user chat session length derived from serialisation of user chat history session in bold db,helps determined user chat engagement time.

Bold db database is serialised in master data store like elasticsearch for analysing user frequency category like new, returning, loyal (friends category) and analysing different chat sessions in which different nodes participated for forming user graph.

These all properties can be used to assign badges to user as give in discord.

**Event Driven Architecture**

For each API called in GRPC node, events are emitted for main operations, all these events can be sent to event tracker urls hosted by p2p chat service servers for storing event information and for plugging this data in notification system (social network eg facebook notification), analytics system and more.

message SubscribeToEventsRequest {}

// Events

message EvtNewChatMessage {

ChatMessage chat\_message = 1;

string room\_name = 2;

}

message EvtNewModeratorMessage {

ModeratorMessage moderator\_message = 1;

string room\_name = 2;

}

message EvtNewBlockMessage {

BlockMessage block\_message = 1;

string room\_name = 2;

}

message EvtNewBanMessage {

BanMessage ban\_message = 1;

string room\_name = 2;

}

message EvtNewRateLimitMessage {

RateLimitMessage rate\_limit\_message = 1;

string room\_name = 2;

}

message EvtPeerJoined {

string room\_name = 1;

string peer\_id = 2;

}

message EvtPeerLeft {

string room\_name = 1;

string peer\_id = 2;

}

message EvtModerationRemoved {

string room\_name = 1;

string peer\_id = 2;

}

message EvtSetNickname {

string room\_name = 1;

string peer\_id = 2;

string nickname = 3;

}

**Publish and Subscribe Pumps for Topic based chatrooms**

Longer running Publish and Subscribe go routines i.e Pumps are started for publishing and subscribing to topic in chatrooms.

As they are longer running they can be optimised using heartbeats and healing go routines concurrency pattern.

// A method of ChatRoom that publishes a chatmessage

// to the PubSub topic until the pubsub context closes

func (cr \*ChatRoom) PubPump() {

for {

select {

case <-cr.psctx.Done():

return

case message := <-cr.Outbound:

// Create a ChatMessage

m := chatmessage{

Message: message,

SenderID: cr.selfid.Pretty(),

SenderName: cr.UserName,

}

// Marshal the ChatMessage into a JSON

messagebytes, err := json.Marshal(m)

if err != nil {

cr.Logs <- chatlog{logprefix: "puberr", logmsg: "could not marshal JSON"}

continue

}

// Publish the message to the topic

err = cr.pstopic.Publish(cr.psctx, messagebytes)

if err != nil {

cr.Logs <- chatlog{logprefix: "puberr", logmsg: "could not publish to topic"}

continue

}

}

}

}

// A method of ChatRoom that continously reads from the subscription

// until either the subscription or pubsub context closes.

// The recieved message is parsed sent into the inbound channel

func (cr \*ChatRoom) SubPump() {

// Start loop

for {

select {

case <-cr.psctx.Done():

return

default:

// Read a message from the subscription

message, err := cr.psub.Next(cr.psctx)

// Check error

if err != nil {

// Close the messages queue (subscription has closed)

close(cr.Inbound)

cr.Logs <- chatlog{logprefix: "suberr", logmsg: "subscription has closed"}

return

}

// Check if message is from self

if message.ReceivedFrom == cr.selfid {

continue

}

// Declare a ChatMessage

cm := &chatmessage{}

// Unmarshal the message data into a ChatMessage

err = json.Unmarshal(message.Data, cm)

if err != nil {

cr.Logs <- chatlog{logprefix: "suberr", logmsg: "could not unmarshal JSON"}

continue

}

// Send the ChatMessage into the message queue

cr.Inbound <- \*cm

}

}

}

A libp2p host with TLS encrypted secure transportation that works over a TCP

transport connection using a Yamux Stream Multiplexer and uses UPnP for the NAT traversal.

A Kademlia DHT is then bootstrapped on this host using the default peers offered by libp2p

and a Peer Discovery service is created from this Kademlia DHT. The PubSub handler is then

created on the host using the peer discovery service created prior.

// Setup a P2P Host Node

nodehost, kaddht := setupHost(ctx)

// Debug log

logrus.Debugln("Created the P2P Host and the Kademlia DHT.")

// Bootstrap the Kad DHT

bootstrapDHT(ctx, nodehost, kaddht)

// Debug log

logrus.Debugln("Bootstrapped the Kademlia DHT and Connected to Bootstrap Peers")

// Create a peer discovery service using the Kad DHT

routingdiscovery := discovery.NewRoutingDiscovery(kaddht)

// Debug log

logrus.Debugln("Created the Peer Discovery Service.")

// Create a PubSub handler with the routing discovery

pubsubhandler := setupPubSub(ctx, nodehost, routingdiscovery)

// Debug log

**Advertising Service**

// A method of P2P to connect to service peers.

// This method uses the Advertise() functionality of the Peer Discovery Service

// to advertise the service and then disovers all peers advertising the same.

// The peer discovery is handled by a go-routine that will read from a channel

// of peer address information until the peer channel closes

func (r \*RoomManager) advertise() {

tick := time.Tick(time.Second \* 5)

for {

<-tick

func() {

r.lock.RLock()

defer r.lock.RUnlock()

for \_, room := range r.rooms {

r.advertiseToRoom(room)

}

}()

}

}

//In case advertisement message illustrated below from a user to chat on a topic in chatroom is not coming for a specified ttl, then user will be evicted from chat participants list and participants map will be refreshed.

func (r \*RoomManager) advertiseToRoom(room \*Room) {

// fetch this node's nickname

thisNickname, \_ := room.getNickname(r.node.ID())

rm := RoomMessageOut{

Type: RoomMessageTypeAdvertise,

Payload: thisNickname,

}

if err := r.publishRoomMessage(context.Background(), room, &rm); err != nil {

r.logger.Error(

"failed publishing room advertise",

zap.Error(err),

zap.String("room", room.topic.String()),

)

}

}