Go-Socket.IO Library Implementation PlanGo-Socket.IO Library Implementation Plan

1. Overview

This document outlines the plan to build a Go module that replicates the core API and functionality of the Socket.IO library. The module will be split into a server package and a client package, intended to run on different machines and communicate over the network. Shared types and protocol definitions will be placed in the root package of the module (e.g., in sockets.go) as requested. The primary transport mechanism will be WebSocket, using gorilla/websocket for the underlying implementation.
The primary goal is to achieve an API "feel" similar to the Node.js version, particularly regarding its event-based nature (.On, .Emit).

2. Guiding Principles

- **API Similarity:** Strive for method and event names that mirror the socket.io-js library to provide a familiar developer experience (e.g., io.On("connection", ...)).
- **WebSocket First:** The initial version (V1) will focus exclusively on WebSocket as the transport. Engine.IO (long-polling) is a complex subsystem and will be considered a future enhancement.
- **Modularity:** server and client are distinct packages. Shared logic (like the event emitter) will be developed in an internal package.
- **Concurrency:** The server must be fully thread-safe, capable of handling thousands of concurrent connections, each running in its own goroutine.
- **Testing:** Each milestone will be accompanied by a clear testing strategy, building from unit tests to full end-to-end (E2E) integration tests.

3. Project Structure

The module (e.g., github.com/your-org/gosock) will have the following structure:

```
/
├── go.mod
├── sockets.go        # Root package (e.g., "gosock") containing shared types
├── server/
│   ├── server.go     # Defines the main Server and HTTP handler
│   └── socket.go     # Defines the server-side Socket struct
├── client/
│   ├── client.go     # Defines the Connect function and Manager
```

```
│   └── socket.go       # Defines the client-side Socket struct
├── internal/
│   ├── emitter/
│   │   └── emitter.go  # Generic, reflection-based event emitter
│   └── parser/
│       └── parser.go   # Handles encoding/decoding of the Socket.IO packet protocol
└── examples/
    └── chat/
        ├── server/main.go
        └── client/main.go
```

4. Core Components

- **Root Package (sockets):**
  - Defines the shared Packet struct, PacketType constants (e.g., CONNECT, DISCONNECT, EVENT, ACK), and other shared error types.
- **internal/emitter:**
  - A generic, concurrent-safe event emitter.
  - It will use sync.Map to store listeners and reflect to dynamically call callback functions with a variable number of arguments. This is the heart of the .On and .Emit magic.
- **internal/parser:**
  - Handles the Socket.IO packet protocol (e.g., 42["event", {"data": 1}]).
  - Provides Encode(Packet) ([]byte, error) and Decode([]byte) (Packet, error) functions.
- **server.Server:**
  - The main server struct. Manages namespaces, rooms, and active connections.
  - Provides the http.Handler (or ServeHTTP method) to upgrade HTTP connections.
- **server.Socket:**
  - Represents a single connected client on the server.
  - Embeds internal/emitter.EventEmitter.
  - Holds the *websocket.Conn.
  - Provides methods like Emit, On, Join, Leave, Broadcast.
- **client.Socket:**
  - Represents the client-side connection.
  - Embeds internal/emitter.EventEmitter.
  - Holds the *websocket.Conn.
  - Provides methods like Emit, On.

5. Implementation Milestones

Milestone 1: Core Types & Protocol

Task: Define the foundation of the library. **Location:** sockets.go, internal/parser/

1. **sockets.go:**
    - Define PacketType as an int (CONNECT, DISCONNECT, EVENT, ACK, BINARY_EVENT, ERROR).
    - Define Packet struct:

```
type Packet struct {
    Type      PacketType
    Namespace string
    Data      json.RawMessage
    ID        *uint64 // Pointer for optional ACK ID
}
```

2. **internal/parser/parser.go:**
    - Implement Encode(p Packet): Serializes a Packet into the Socket.IO string/binary format. (e.g., EVENT packet: 42["event_name", "data"]).
    - Implement Decode(data []byte): Deserializes the raw message into a Packet. This will involve regex or careful string/byte parsing.

Testing:

- Unit tests for parser. Test Encode and Decode with all packet types, including edge cases (empty data, no namespace, with ACK ID).

Milestone 2: Internal Event Emitter

Task: Create the generic, reflection-based event system. **Location:** internal/emitter/

1. **emitter.go:**
    - Define EventEmitter struct. It will hold a sync.Map where keys are event names (string) and values are slices of reflect.Value (the callbacks).
    - On(event string, callback interface{}): Registers a new event listener. This will use reflect.ValueOf(callback) to store the function.
    - Emit(event string, args ...interface{}): Fires an event. This will:
        - Look up listeners for the event.
        - Create a slice of reflect.Value from args.
        - Loop through listeners and call listener.Call(reflectedArgs).
        - Include panic recovery within the Call to prevent one bad listener from crashing the loop.
    - Implement Once(event string, callback interface{}).
    - Implement Off(event string, callback interface{}).

Testing:

- Heavy unit testing.
- Test On and Emit with various function signatures (no args, one arg, multiple args).

- Test concurrent On and Emit calls.
- Test that Once only fires once.
- Test Off correctly removes listeners.
- Test panic recovery.

Milestone 3: Server - Connection & Basic Events

Task: Get a server running that can accept a WebSocket connection and handle basic events.
Location: server/

1. **server.go:**
    - Define Server struct. It will hold connection/socket maps and a *websocket.Upgrader.
    - NewServer(...) *Server: Constructor function.
    - Server.ServeHTTP(w http.ResponseWriter, r *http.Request):
        - Uses gorilla/websocket.Upgrader to upgrade the connection.
        - On success, creates a new server.Socket.
        - Generates a UUID for the socket (google/uuid).
        - Spawns socket.readLoop() and socket.writeLoop() as goroutines.
        - Emits the connection event: server.Emit("connection", socket).
2. **socket.go:**
    - Define Socket struct. It embeds emitter.EventEmitter.
    - Holds *websocket.Conn, ID string, and a buffered channel for outbound messages (the write queue).
    - readLoop():
        - Runs in a loop: websocket.ReadMessage().
        - Decodes the message using parser.Decode().
        - switch packet.Type:
            - EVENT: socket.Emit(eventName, ...data). (Will use reflection to unpack packet.Data into args for Emit).
            - DISCONNECT: Close connection, socket.Emit("disconnect", "client request").
    - writeLoop():
        - Ranges over the outbound message channel.
        - Encodes the Packet using parser.Encode().
        - Writes to the WebSocket: websocket.WriteMessage().
    - Emit(event string, args ...interface{}):
        - Creates an EVENT packet.
        - Marshals args into json.RawMessage for packet.Data.
        - Pushes the packet onto the write-queue channel.
    - Close(): Closes the connection and cleans up.

Testing:

- **E2E Test:**
    - Start a server.Server in a test.
    - Register a handler: server.On("connection", func(s *server.Socket) { s.On("ping",

func() { s.Emit("pong") }) }).
    - Use a *basic* gorilla/websocket client (not our client package yet) to connect.
    - Manually craft and send a ping packet.
    - Assert that a pong packet is received.

Milestone 4: Client Implementation

Task: Create the client package that can connect to the server. **Location:** client/

1. **client.go:**
    - Connect(url string, ...) (*Socket, error):
        - Uses gorilla/websocket.Dialer to connect to the server.
        - On success, creates a new client.Socket.
        - Spawns socket.readLoop() and socket.writeLoop().
        - Emits connect on the socket: socket.Emit("connect").
2. **socket.go:**
    - Define client.Socket struct. Embeds emitter.EventEmitter.
    - Similar to server.Socket, it has readLoop, writeLoop, Emit, and Close.
    - The readLoop will decode packets and fire events (e.g., pong from the server).
    - Emit will encode packets and send them (e.g., ping).

Testing:

- **Full E2E Test:**
    - Use the server from Milestone 3.
    - Use the client from this milestone.
    - client.Connect() and client.On("connect", ...).
    - client.On("pong", ...).
    - client.Emit("ping").
    - Verify the full "connect -> ping -> pong" flow works using our own library on both ends.

Milestone 5: Rooms, Broadcasting, and Namespaces

Task: Implement the advanced dispatching features. **Location:** server/

1. **Namespaces:**
    - Refactor server.Server to hold a sync.Map of *Namespace objects (key is namespace name).
    - Define Namespace struct. It embeds emitter.EventEmitter.
    - Server.Of(path string) *Namespace: Gets or creates a namespace.
    - Server.ServeHTTP now needs to parse the namespace from the request and dispatch the new Socket to the correct Namespace, emitting connection on *that namespace*.
    - client.Connect must be updated to support connecting to specific namespaces.
2. **Rooms:**
    - Namespace will manage rooms (e.g., map[string]map[string]bool // roomName -> socketID -> true).
    - server.Socket.Join(room string): Adds socket ID to the namespace's room map.

    - server.Socket.Leave(room string): Removes socket ID.
3. **Broadcasting:**
    - server.Socket.Broadcast() *BroadcastOperator: Returns a helper struct that excludes the sender's ID.
    - Namespace.To(room string) *BroadcastOperator: Returns a helper struct for a specific room.
    - BroadcastOperator.Emit(event string, args ...interface{}):
        - Creates the packet.
        - Iterates over the target sockets (all in namespace, all in room, etc.) and pushes the packet to each socket's write queue.

Testing:

- **E2E Test:**
    - Client A connects to / and joins "room1".
    - Client B connects to / and joins "room1".
    - Client C connects to / and joins "room2".
    - Client A emits server.On("msg", func(m string) { s.Broadcast().To("room1").Emit("broadcast", m) }).
    - Verify: Client B receives "broadcast", but Client A and Client C do not.

Milestone 6: ACKs (Callbacks)

Task: Implement request/response semantics. **Location:** internal/emitter/, server/, client/

1. **Emitter/Socket Changes:**
    - Emit(event string, args ...interface{}):
        - Check if the *last* item in args is a function (using reflect).
        - If it is, this is an ACK.
        - Store this callback in a sync.Map on the socket: ackMap[ackID]callback.
        - Generate a new ackID (using atomic.AddUint64).
        - Pass this ackID in the Packet.ID field.
        - Remove the callback from args before marshaling.
2. **Handler Changes:**
    - On(event string, callback interface{}):
        - When an event *with an ackID* arrives, the emitter must be smart enough to pass an ack function as the *last* argument to the user's callback.
        - server.On("request_data", func(data string, ack func(string, int)) { ack("response", 123) })
        - This ack function, when called, creates an ACK packet, sets Packet.ID to the *original* ackID, and Emits it.
3. **ReadLoop Changes:**
    - In readLoop, switch packet.Type:
        - ACK: This is a *response* to an Emit we sent.
            - Look up packet.ID in the ackMap.
            - If found, decode packet.Data and reflect.Call the callback.
            - Delete the callback from the map.

    - Implement timeouts for ACKs (e.g., using a time.AfterFunc).

Testing:

- **E2E Test:**
    - client.Emit("get_data", "foo", func(response string) { ... })
    - server.On("get_data", func(d string, ack func(string)) { ack("echo:" + d) })
    - Assert the client's callback is fired with the value "echo:foo".

Milestone 7: Documentation & Example App

Task: Make the library usable by others.

1. **GoDoc:** Write comprehensive GoDoc comments for all public types, methods, and packages (server, client, and root).
2. **README.md:** Create a root README.md with:
    - Installation instructions.
    - Basic usage example (server and client).
    - Link to the examples directory.
3. **Chat Example:**
    - Create examples/chat/ that fully implements the socket.io/get-started/chat application.
    - This will be the ultimate E2E test and gold-standard example.

6. Future Enhancements (Out of Scope for V1)

- **Engine.IO:** Implement the full Engine.IO protocol, including HTTP long-polling as a fallback for when WebSockets are not available.
- **Middleware:** Implement a server.Use(...) method for middleware, similar to Express/Socket.IO.
- **Binary Data:** Natively support []byte in Emit using BINARY_EVENT packets.
- **Gzip/Deflate:** Support for transport-level compression.

1. Overview

This document outlines the plan to build a Go module that replicates the core API and functionality of the Socket.IO library. The module will be split into a server package and a client package, intended to run on different machines and communicate over the network. Shared types and protocol definitions will be placed in the root package of the module (e.g., in sockets.go) as requested. The primary transport mechanism will be WebSocket, using gorilla/websocket for the underlying implementation.
The primary goal is to achieve an API "feel" similar to the Node.js version, particularly regarding its event-based nature (.On, .Emit).

2. Guiding Principles

- **API Similarity:** Strive for method and event names that mirror the socket.io-js library to provide a familiar developer experience (e.g., io.On("connection", ...)).

- **WebSocket First:** The initial version (V1) will focus exclusively on WebSocket as the transport. Engine.IO (long-polling) is a complex subsystem and will be considered a future enhancement.
- **Modularity:** server and client are distinct packages. Shared logic (like the event emitter) will be developed in an internal package.
- **Concurrency:** The server must be fully thread-safe, capable of handling thousands of concurrent connections, each running in its own goroutine.
- **Testing:** Each milestone will be accompanied by a clear testing strategy, building from unit tests to full end-to-end (E2E) integration tests.

3. Project Structure

The module (e.g., github.com/your-org/gosock) will have the following structure:

```
/
├── go.mod
├── sockets.go        # Root package (e.g., "gosock") containing shared types
├── server/
│   ├── server.go     # Defines the main Server and HTTP handler
│   └── socket.go     # Defines the server-side Socket struct
├── client/
│   ├── client.go     # Defines the Connect function and Manager
│   └── socket.go     # Defines the client-side Socket struct
├── internal/
│   ├── emitter/
│   │   └── emitter.go  # Generic, reflection-based event emitter
│   └── parser/
│       └── parser.go   # Handles encoding/decoding of the Socket.IO packet protocol
└── examples/
    └── chat/
        ├── server/main.go
        └── client/main.go
```

4. Core Components

- **Root Package (gosock):**
  - Defines the shared Packet struct, PacketType constants (e.g., CONNECT, DISCONNECT, EVENT, ACK), and other shared error types.
- **internal/emitter:**
  - A generic, concurrent-safe event emitter.
  - It will use sync.Map to store listeners and reflect to dynamically call callback functions with a variable number of arguments. This is the heart of the .On and .Emit magic.

- **internal/parser:**
  - Handles the Socket.IO packet protocol (e.g., 42["event", {"data": 1}]).
  - Provides Encode(Packet) ([]byte, error) and Decode([]byte) (Packet, error) functions.
- **server.Server:**
  - The main server struct. Manages namespaces, rooms, and active connections.
  - Provides the http.Handler (or ServeHTTP method) to upgrade HTTP connections.
- **server.Socket:**
  - Represents a single connected client on the server.
  - Embeds internal/emitter.EventEmitter.
  - Holds the *websocket.Conn.
  - Provides methods like Emit, On, Join, Leave, Broadcast.
- **client.Socket:**
  - Represents the client-side connection.
  - Embeds internal/emitter.EventEmitter.
  - Holds the *websocket.Conn.
  - Provides methods like Emit, On.

5. Implementation Milestones

Milestone 1: Core Types & Protocol

Task: Define the foundation of the library.
Location: sockets.go, internal/parser/

1. **sockets.go:**
   - Define PacketType as an int (CONNECT, DISCONNECT, EVENT, ACK, BINARY_EVENT, ERROR).
   - Define Packet struct:

```
type Packet struct {
    Type      PacketType
    Namespace string
    Data      json.RawMessage
    ID        *uint64 // Pointer for optional ACK ID
}
```

2. **internal/parser/parser.go:**
   - Implement Encode(p Packet): Serializes a Packet into the Socket.IO string/binary format. (e.g., EVENT packet: 42["event_name", "data"]).
   - Implement Decode(data []byte): Deserializes the raw message into a Packet. This will involve regex or careful string/byte parsing.

Testing:

- Unit tests for parser. Test Encode and Decode with all packet types, including edge

cases (empty data, no namespace, with ACK ID).

Milestone 2: Internal Event Emitter

Task: Create the generic, reflection-based event system.
Location: internal/emitter/

1. **emitter.go:**
    - Define EventEmitter struct. It will hold a sync.Map where keys are event names (string) and values are slices of reflect.Value (the callbacks).
    - On(event string, callback interface{}): Registers a new event listener. This will use reflect.ValueOf(callback) to store the function.
    - Emit(event string, args ...interface{}): Fires an event. This will:
        - Look up listeners for the event.
        - Create a slice of reflect.Value from args.
        - Loop through listeners and call listener.Call(reflectedArgs).
        - Include panic recovery within the Call to prevent one bad listener from crashing the loop.
    - Implement Once(event string, callback interface{}).
    - Implement Off(event string, callback interface{}).

Testing:

- Heavy unit testing.
- Test On and Emit with various function signatures (no args, one arg, multiple args).
- Test concurrent On and Emit calls.
- Test that Once only fires once.
- Test Off correctly removes listeners.
- Test panic recovery.

Milestone 3: Server - Connection & Basic Events

Task: Get a server running that can accept a WebSocket connection and handle basic events.
Location: server/

1. **server.go:**
    - Define Server struct. It will hold connection/socket maps and a *websocket.Upgrader.
    - NewServer(...) *Server: Constructor function.
    - Server.ServeHTTP(w http.ResponseWriter, r *http.Request):
        - Uses gorilla/websocket.Upgrader to upgrade the connection.
        - On success, creates a new server.Socket.
        - Generates a UUID for the socket (google/uuid).
        - Spawns socket.readLoop() and socket.writeLoop() as goroutines.
        - Emits the connection event: server.Emit("connection", socket).
2. **socket.go:**
    - Define Socket struct. It embeds emitter.EventEmitter.
    - Holds *websocket.Conn, ID string, and a buffered channel for outbound messages (the write queue).

    - readLoop():
        - Runs in a loop: websocket.ReadMessage().
        - Decodes the message using parser.Decode().
        - switch packet.Type:
            - EVENT: socket.Emit(eventName, ...data). (Will use reflection to unpack packet.Data into args for Emit).
            - DISCONNECT: Close connection, socket.Emit("disconnect", "client request").
    - writeLoop():
        - Ranges over the outbound message channel.
        - Encodes the Packet using parser.Encode().
        - Writes to the WebSocket: websocket.WriteMessage().
    - Emit(event string, args ...interface{}):
        - Creates an EVENT packet.
        - Marshals args into json.RawMessage for packet.Data.
        - Pushes the packet onto the write-queue channel.
    - Close(): Closes the connection and cleans up.

Testing:

- **E2E Test:**
    - Start a server.Server in a test.
    - Register a handler: server.On("connection", func(s *server.Socket) { s.On("ping", func() { s.Emit("pong") }) }).
    - Use a *basic* gorilla/websocket client (not our client package yet) to connect.
    - Manually craft and send a ping packet.
    - Assert that a pong packet is received.

Milestone 4: Client Implementation

Task: Create the client package that can connect to the server.
Location: client/

1. **client.go:**
    - Connect(url string, ...) (*Socket, error):
        - Uses gorilla/websocket.Dialer to connect to the server.
        - On success, creates a new client.Socket.
        - Spawns socket.readLoop() and socket.writeLoop().
        - Emits connect on the socket: socket.Emit("connect").
2. **socket.go:**
    - Define client.Socket struct. Embeds emitter.EventEmitter.
    - Similar to server.Socket, it has readLoop, writeLoop, Emit, and Close.
    - The readLoop will decode packets and fire events (e.g., pong from the server).
    - Emit will encode packets and send them (e.g., ping).

Testing:

- **Full E2E Test:**
    - Use the server from Milestone 3.

- Use the client from this milestone.
- client.Connect() and client.On("connect", ...).
- client.On("pong", ...).
- client.Emit("ping").
- Verify the full "connect -> ping -> pong" flow works using our own library on both ends.

Milestone 5: Rooms, Broadcasting, and Namespaces

Task: Implement the advanced dispatching features.
Location: server/

1. **Namespaces:**
   - Refactor server.Server to hold a sync.Map of *Namespace objects (key is namespace name).
   - Define Namespace struct. It embeds emitter.EventEmitter.
   - Server.Of(path string) *Namespace: Gets or creates a namespace.
   - Server.ServeHTTP now needs to parse the namespace from the request and dispatch the new Socket to the correct Namespace, emitting connection on *that namespace*.
   - client.Connect must be updated to support connecting to specific namespaces.
2. **Rooms:**
   - Namespace will manage rooms (e.g., map[string]map[string]bool // roomName -> socketID -> true).
   - server.Socket.Join(room string): Adds socket ID to the namespace's room map.
   - server.Socket.Leave(room string): Removes socket ID.
3. **Broadcasting:**
   - server.Socket.Broadcast() *BroadcastOperator: Returns a helper struct that excludes the sender's ID.
   - Namespace.To(room string) *BroadcastOperator: Returns a helper struct for a specific room.
   - BroadcastOperator.Emit(event string, args ...interface{}):
     - Creates the packet.
     - Iterates over the target sockets (all in namespace, all in room, etc.) and pushes the packet to each socket's write queue.

Testing:

- **E2E Test:**
  - Client A connects to / and joins "room1".
  - Client B connects to / and joins "room1".
  - Client C connects to / and joins "room2".
  - Client A emits server.On("msg", func(m string) { s.Broadcast().To("room1").Emit("broadcast", m) }).
  - Verify: Client B receives "broadcast", but Client A and Client C do not.

Milestone 6: ACKs (Callbacks)

Task: Implement request/response semantics.
Location: internal/emitter/, server/, client/

1. **Emitter/Socket Changes:**
    - Emit(event string, args ...interface{}):
        - Check if the *last* item in args is a function (using reflect).
        - If it is, this is an ACK.
        - Store this callback in a sync.Map on the socket: ackMap[ackID]callback.
        - Generate a new ackID (using atomic.AddUint64).
        - Pass this ackID in the Packet.ID field.
        - Remove the callback from args before marshaling.
2. **Handler Changes:**
    - On(event string, callback interface{}):
        - When an event *with an ackID* arrives, the emitter must be smart enough to pass an ack function as the *last* argument to the user's callback.
        - server.On("request_data", func(data string, ack func(string, int)) { ack("response", 123) })
        - This ack function, when called, creates an ACK packet, sets Packet.ID to the *original* ackID, and Emits it.
3. **ReadLoop Changes:**
    - In readLoop, switch packet.Type:
        - ACK: This is a *response* to an Emit we sent.
            - Look up packet.ID in the ackMap.
            - If found, decode packet.Data and reflect.Call the callback.
            - Delete the callback from the map.
    - Implement timeouts for ACKs (e.g., using a time.AfterFunc).

Testing:

- **E2E Test:**
    - client.Emit("get_data", "foo", func(response string) { ... })
    - server.On("get_data", func(d string, ack func(string)) { ack("echo:" + d) })
    - Assert the client's callback is fired with the value "echo:foo".

Milestone 7: Documentation & Example App

Task: Make the library usable by others.

1. **GoDoc:** Write comprehensive GoDoc comments for all public types, methods, and packages (server, client, and root).
2. **README.md:** Create a root README.md with:
    - Installation instructions.
    - Basic usage example (server and client).
    - Link to the examples directory.
3. **Chat Example:**
    - Create examples/chat/ that fully implements the socket.io/get-started/chat application.
    - This will be the ultimate E2E test and gold-standard example.

6. Future Enhancements (Out of Scope for V1)

- **Engine.IO:** Implement the full Engine.IO protocol, including HTTP long-polling as a fallback for when WebSockets are not available.
- **Middleware:** Implement a server.Use(...) method for middleware, similar to Express/Socket.IO.
- **Binary Data:** Natively support []byte in Emit using BINARY_EVENT packets.
- **Gzip/Deflate:** Support for transport-level compression.