Planning Poker System Design

Real-time Collaborative Estimation Platform

Author: Mike Conlen

Version: 0.7.5

Date: June 29, 2025

Abstract

This document describes the architectural design and implementation of a real-time Planning Poker system built with Go and WebSockets. The system provides a collaborative platform for Agile development teams to perform story point estimation in distributed environments. This specification covers the system architecture, WebSocket message protocols, session management, and security considerations.

Planning Poker Project Documentation
https://github.com/meconlen/planning_poker

Contents

1 Introduction

Planning Poker is a consensus-based, gamified technique for estimating effort or relative size of development goals in software development [5]. This system implements a real-time, web-based Planning Poker platform that enables distributed teams to collaborate effectively on story point estimation, following Agile methodologies [6].

1.1 Purpose and Scope

The Planning Poker system addresses the need for remote Agile teams to conduct estimation sessions with the same effectiveness as in-person meetings. The system provides:

- Real-time collaborative estimation sessions
- Moderator-controlled session flow
- Standard Fibonacci voting scales
- Session state persistence
- Multi-user support with role-based permissions

1.2 System Overview

The system follows a client-server architecture with WebSocket-based real-time communication [2]. The server is implemented in Go [3], leveraging the Gorilla WebSocket library [4] for efficient bidirectional communication. The client interface is a modern web application using vanilla JavaScript and WebSocket APIs.

2 System Architecture

2.1 High-Level Architecture

The Planning Poker system employs a three-tier architecture:

1. **Presentation Layer**: Web-based user interface
2. **Application Layer**: Go-based server with WebSocket handling
3. **Data Layer**: In-memory session management with future persistence options

2.2 Technology Stack

Component	Technology
Backend Language	Go 1.24+
WebSocket Library	Gorilla WebSocket v1.5.3
UUID Generation	Google UUID v1.6.0
Frontend	HTML5, CSS3, JavaScript (ES6+)
Containerization	Docker
CI/CD	GitHub Actions
Documentation	LaTeX

Table 1: Technology Stack

2.3 Package Structure

The Go application follows clean architecture principles with clear separation of concerns:

```
planning-poker/
|-- main.go                  # Application entry point
|-- internal/
|   |-- server/
|   |   `-- server.go          # HTTP and WebSocket handlers
|   `-- poker/
|       |-- session.go         # Session management logic
|       |-- session_test.go    # Unit tests
|       `-- session_unit_test.go # Comprehensive test suite
|-- web/
|   `-- index.html             # Frontend application
|-- test/
|   `-- client.go              # Integration test client
|-- scripts/
|   |-- workflow.sh            # Development workflow
|   |-- monitor-actions.sh     # CI/CD monitoring
|   `-- check-actions.sh       # Status checking
`-- docs/
    |-- design.tex             # This document
    |-- references.bib         # Bibliography
    `-- Makefile               # Documentation build
```

Listing 1: Project Structure

3 Session Management

3.1 Session Lifecycle

Sessions progress through distinct states that control user interactions and system behavior:

1. **Waiting**: Initial state where participants join but cannot vote
2. **Active**: Session is running and users can participate in voting
3. **Completed**: Session has ended (future enhancement)

3.2 User Roles

The system implements role-based access control with two primary roles:

Role	Permissions
Moderator	• Start and control session flow • Reveal votes • Set story descriptions • Initiate new voting rounds • Manage session state
Participant	• Submit votes • View session state • Wait in waiting room until session starts • Receive real-time updates

Table 2: User Roles and Permissions

4 WebSocket Message Protocol

4.1 Message Format

All WebSocket messages follow a standardized JSON format [1] for consistency and ease of parsing:

```
{
    "type": "message_type",
    "data": {
        // Message-specific payload
    }
}
```

Listing 2: Base Message Format

4.2 Client-to-Server Messages

4.2.1 Vote Submission

Participants submit their estimates using the vote message:

```
{
    "type": "vote",
    "data": {
        "vote": "5"
    }
}
```

Listing 3: Vote Message

Validation: The vote value must be from the standard Fibonacci sequence: 0, 0.5, 1, 2, 3, 5, 8, 13, 21, ?, coffee

4.2.2 Session Control

Moderators control session flow with the following messages:

```
{
    "type": "start_session"
}
```

Listing 4: Start Session Message

```
{
    "type": "reveal"
}
```

Listing 5: Reveal Votes Message

```
{
    "type": "new_round"
}
```

Listing 6: New Round Message

```
{
    "type": "set_story",
    "data": {
        "story": "User story description"
    }
}
```

Listing 7: Set Story Message

4.3 Server-to-Client Messages

4.3.1 Session State Broadcast

The server broadcasts complete session state to all connected clients:

```
{
    "type": "session_state",
    "data": {
        "status": "active",
        "currentStory": "As a user, I want to...",
        "votesRevealed": false,
        "users": {
            "user-uuid-1": {
                "id": "user-uuid-1",
                "name": "Alice",
                "vote": "5",
                "isModerator": true,
                "isOnline": true
            },
            "user-uuid-2": {
                "id": "user-uuid-2",
```

```
                "name": "Bob",
                "vote": null,
                "isModerator": false,
                "isOnline": true
            }
        }
    }
}
```

Listing 8: Session State Message

4.3.2 Waiting Room Notification

Non-moderator users receive waiting room notifications:

```
{
    "type": "waiting_room",
    "data": {
        "message": "Waiting for moderator to start the session..."
    }
}
```

Listing 9: Waiting Room Message

4.3.3 Session Start Notification

When a session begins, all participants receive:

```
{
    "type": "start_session",
    "data": {
        "message": "Session has started! You can now participate in voting."
    }
}
```

Listing 10: Session Start Message

4.3.4 User Presence Updates

The system broadcasts user join/leave events:

```
{
    "type": "user_joined",
    "data": {
        "userId": "user-uuid",
        "userName": "Charlie"
    }
}
```

Listing 11: User Joined Message

```json
{
    "type": "user_left",
    "data": {
        "userId": "user-uuid",
        "userName": "Charlie"
    }
}
```

Listing 12: User Left Message

5 Security Considerations

5.1 Authentication and Authorization

Currently, the system uses session-based user identification with the following security measures:

- UUID-based session and user identification
- Role-based permission enforcement
- Server-side validation of all user actions
- Protection against unauthorized moderator actions

5.2 Input Validation

All client inputs undergo server-side validation:

- Vote values restricted to valid Fibonacci sequence
- User names sanitized for display
- Message types validated against known protocols
- Session IDs validated for format and existence

5.3 Future Security Enhancements

Planned security improvements include:

- JWT-based authentication
- Rate limiting for message frequency
- HTTPS enforcement
- Session timeout mechanisms
- Audit logging

6 Deployment and Operations

6.1 Container Deployment

The system supports Docker-based deployment with multi-stage builds for optimization:

```
FROM golang:1.24-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o planning-poker

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/planning-poker .
COPY web/ ./web/
EXPOSE 8080
CMD ["./planning-poker"]
```

Listing 13: Docker Configuration

6.2 CI/CD Pipeline

The project employs GitHub Actions for continuous integration and deployment:

- Automated testing on Go 1.24+
- Static analysis with `go vet` and `staticcheck`
- Docker image building and testing
- Automated releases with semantic versioning

7 Testing Strategy

7.1 Test Coverage

The system implements comprehensive testing at multiple levels:

Test Type	Coverage
Unit Tests	Session management, user roles, voting logic
Integration Tests	WebSocket message flows, client-server interaction
End-to-End Tests	Complete user scenarios via test client

Table 3: Testing Coverage

7.2 Test Client

A dedicated Go test client simulates browser behavior for integration testing:

```
// Create test clients
moderator := NewTestClient("Alice", sessionID, true)
participant := NewTestClient("Bob", sessionID, false)

// Connect and test workflow
moderator.Connect(serverURL)
participant.Connect(serverURL)
moderator.StartSession()
participant.Vote("5")
```

Listing 14: Test Client Usage

8 Performance Considerations

8.1 Scalability

Current design considerations for scalability:

- In-memory session storage for low latency
- Efficient WebSocket connection management
- Minimal message overhead with JSON protocol
- Stateless server design for horizontal scaling

8.2 Resource Usage

Typical resource requirements:

- Memory: 10MB base + 1KB per active user
- CPU: Minimal when idle, spikes during message broadcasts
- Network: 1KB per message, scales with user count

9 Future Enhancements

9.1 Planned Features

- Persistent session storage (Redis/PostgreSQL)
- Custom voting scales
- Session analytics and reporting
- Mobile-responsive design improvements
- Integration with project management tools

9.2 Architecture Evolution

Long-term architectural considerations:

- Microservices decomposition
- Event-driven architecture with message queues
- Multi-region deployment support
- Real-time analytics dashboard

10 Conclusion

The Planning Poker system successfully implements a robust, real-time collaborative estimation platform suitable for distributed Agile teams. The WebSocket-based architecture provides low-latency communication while maintaining simplicity and reliability.

The system's modular design and comprehensive testing strategy ensure maintainability and extensibility for future enhancements. The documented message protocols and security considerations provide a solid foundation for integration and deployment in production environments.

References

[1] Tim Bray. RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, 2014. Available at: https://tools.ietf.org/html/rfc7159.

[2] Ian Fette and Alexey Melnikov. RFC 6455: The WebSocket Protocol. RFC 6455, 2011. Available at: https://tools.ietf.org/html/rfc6455.

[3] Google Inc. The go programming language. Available at: https://golang.org/, 2024. Version 1.24.

[4] Gorilla Web Toolkit. Gorilla websocket: A fast, well-tested and widely used websocket implementation for go. Available at: https://github.com/gorilla/websocket, 2024. Version 1.5.3.

[5] James Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, 3:22–23, 2002.

[6] Ken Schwaber and Jeff Sutherland. The scrum guide: The definitive guide to scrum: The rules of the game. 2020. Available at: https://scrumguides.org/scrum-guide.html.