# Chat Application RPC Protocol

## Overview

Binary RPC protocol using network byte order (big-endian), fixed-length integers, and length-prefixed strings/arrays. The system implements a client-server architecture where clients connect via TCP, send serialized RPC requests, and receive binary responses.

Each request and response is framed with a 4-byte length prefix to handle TCP stream boundaries.

## Transport Layer

**Protocol**: TCP/IP (IPv4) - Server listens for incoming connections on a specified port - Client initiates connection to `host:port` - Each connection is handled synchronously (server handles one request per connection, then closes) - Maximum message size: 65536 bytes (enforced at transport layer)

**Message Framing**: `[4 bytes: message length (uint32_t, network byte order)] [N bytes: message data]`

The length field represents the size of the message data only (not including the 4-byte length prefix itself).

**Byte Order**: All multi-byte integers use network byte order (big-endian). Helpers: - Send: `htonl()` for uint32_t, `htons()` for uint16_t - Receive: `ntohl()` for uint32_t, `ntohs()` for uint16_t

## Client/Server Communication Flow

### Server Architecture

1. **Initialization**: `net_listen(port)` creates a listening socket with `SO_REUSEADDR`
2. **Main loop**:
    - `net_accept()` blocks waiting for client connection
    - On connection, read 4-byte length prefix
    - Read exactly N bytes of request data
    - Deserialize binary request via `request_from_bytes()`
    - Dispatch to appropriate handler (login, logout, tell, say, receive)
    - Serialize response via `response_to_bytes()`
    - Send 4-byte length prefix + response data
    - Connection closes after single request/response exchange

### Client Architecture

1. **Connection**: `net_connect(host, port)` to server
2. **Request**:
    - Build Request struct from command-line arguments
    - Serialize via `request_to_bytes()`
    - Send 4-byte length prefix + serialized request
3. **Response**:
    - Read 4-byte length prefix
    - Read exactly N bytes of response data
    - Deserialize via `response_from_bytes()`
    - Print result or error to stdout
4. **Cleanup**: Close socket, free allocated memory

### Request/Response Lifecycle

```
Client                           Server
  |                                |
  +------ TCP SYN ----- SYN-ACK ---+
  |                                |
  +-- [length][request] ---------->|
  |                                | request_from_bytes()
  |                                | handle_request()
  |                                | response_to_bytes()
  |<------ [length][response] -----+
  |                                |
  +------ TCP FIN ----- FIN-ACK ---+
    |                                  |
```

## General Encoding Rules

- **Byte order**: Network byte order (big-endian)
- **Message framing**: `[4-byte length (uint32_t)][message data]` where length is the size of message data only
- **Strings**: `[2-byte length (uint16_t)][data]`
- **Arrays**: `[2-byte count (uint16_t)][element1][element2]...`
- **Integers**: All multi-byte integers are in network byte order

## Request Format

All requests start with a 1-byte RPC type identifier:

```
[1 byte: RpcType]
[type-specific payload]
```

### RPC Type Values

```
0 = RPC_LOGIN
1 = RPC_LOGOUT
2 = RPC_TELL
3 = RPC_SAY
4 = RPC_RECEIVE
```

### LOGIN Request

```
[1 byte: 0]
[2 bytes: username length]
[variable: username bytes]
```

### LOGOUT Request

```
[1 byte: 1]
[2 bytes: username length]
[variable: username bytes]
```

### TELL Request

```
[1 byte: 2]
[2 bytes: username length]
[variable: username bytes]
[2 bytes: target length]
[variable: target bytes]
[2 bytes: message length]
[variable: message bytes]
```

### SAY Request

```
[1 byte: 3]
[2 bytes: username length]
[variable: username bytes]
[2 bytes: message length]
[variable: message bytes]
```

### RECEIVE Request

```
[1 byte: 4]
[2 bytes: username length]
[variable: username bytes]
```

## Response Format

All responses start with a 1-byte status code:

```
[1 byte: Status]
[type-specific payload]
```

### Status Code Values

```
0 = STATUS_OK
1 = STATUS_ERR_USER_EXISTS
2 = STATUS_ERR_USER_NOT_FOUND
3 = STATUS_ERR_TARGET_NOT_FOUND
4 = STATUS_ERR_MALFORMED
5 = STATUS_ERR_NETWORK
```

### Success Responses (LOGIN, LOGOUT, TELL, SAY)

```
[1 byte: 0]
```

### RECEIVE Success Response

```
[1 byte: 0]
[2 bytes: message count]
[message 1]:
  [2 bytes: from length]
  [variable: from bytes]
  [2 bytes: text length]
  [variable: text bytes]
```

### Error Response (all RPC types)

```
[1 byte: error code (1-5)]
```

## Implementation Constraints

**Limits**: - Maximum username length: 32 bytes (enforced by `MAX_USERNAME`) - Maximum message size: 65536 bytes total (enforced at frame level) - Maximum string field length: 65535 bytes (uint16_t max) - Server storage: In-memory only (not persistent across restarts)

**Validation**: - Message length must be ≤ 65536 bytes - Username must be < 32 bytes - RPC type must be 0-4 - All required fields must be present in message - Target user must exist for TELL requests - User must exist for LOGOUT, SAY, RECEIVE requests - User must NOT exist for LOGIN requests

**Memory**: - All dynamic allocations use `malloc()` / `realloc()` - Request/response objects must be freed via `request_destroy()` / `response_destroy()` - Server maintains vectors of users and messages (grow by 2x on overflow, starting at 4 elements)

## Examples

### TELL Request: alice sends “hi” to bob

Message data (17 bytes total): `02 # RPC_TELL 00 05 # username length = 5 61 6c 69 63 65 # "alice" 00 03 # target length = 3 62 6f 62 # "bob" 00 02 # message length = 2 68 69 # "hi"`

On wire (with 4-byte length prefix): `00 00 00 11 # message length = 17 02 00 05 61 6c 69 63 65 00 03 62 6f 62 00 02 68 69`

### TELL Error Response: target not found

Message data (1 byte): `03 # STATUS_ERR_TARGET_NOT_FOUND`

On wire: `00 00 00 01 # message length = 1 03`

### RECEIVE Request: charlie receives messages

Message data (10 bytes): `04 # RPC_RECEIVE 00 07 # username length = 7 63 68 61 72 6c 69 65 # "charlie"`

On wire: `00 00 00 0a # message length = 10 04 00 07 63 68 61 72 6c 69 65`

## RECEIVE Success Response: 2 messages

Message data (32 bytes): `00 # STATUS_OK 00 02 # message count = 2 00 05 # from length = 5 61 6c 69 63 65 # "alice" 00 05 # text length = 5 68 65 6c 6c 6f # "hello" 00 04 # from length = 4 64 61 76 65 # "dave" 00 03 # text length = 3 68 69 21 # "hi!"`

On wire: `00 00 00 20 # message length = 32 00 00 02 00 05 61 6c 69 63 65 00 05 68 65 6c 6c 6f 00 04 64 61 76 65 00 03 68 69 21`