BLINK DB

1.0 (24CS60R77 Ishan Rai)

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Client Class Reference

Client for BLINK DB server using RESP-2 protocol.

```
#include <client.h>
```

Public Member Functions

- Client (const std::string &host="localhost", int port=9001)

  Construct a new Client object.
- ∼Client ()

  Destroy the Client object.
- bool connect ()

  Connect to the BLINK DB server.
- void disconnect ()

  Disconnect from the server.
- bool is_connected () const

  Check if client is connected.
- std::string execute (const std::string &command, const std::vector< std::string > &args={})

  Execute a command.
- void run_interactive (std::function< void(const std::string &)> on_response=nullptr)

  Run interactive mode.

3.1.1 Constructor & Destructor Documentation

3.1.1.1 Client()

```
Client::Client (
            const std::string & host = "localhost",
            int port = 9001 )
```

Parameters

host	Server hostname or IP
port	Server port
host	Server hostname or IP address
port	Server port number

3.1.1.2 ∼Client()

```
Client::∼Client ( )
```

Destroy the Client object and close any open connection.

3.1.2 Member Function Documentation

3.1.2.1 connect()

```
bool Client::connect ( )
```

Establish a connection to the BLINK DB server.

Returns

true if connection successful, false otherwise

Creates a TCP socket, resolves the server hostname, and connects to the server. Sets a 5-second timeout for socket operations to prevent blocking indefinitely.

Returns

true if connection was successful, false otherwise

3.1.2.2 disconnect()

```
void Client::disconnect ( )
```

Closes the socket connection if it's open

3.1.2.3 execute()

```
std::string Client::execute (
             const std::string & command,
             const std::vector< std::string > & args = {} )
```

Execute a command on the server.

Parameters

command	Command to execute (e.g., "SET", "GET", "DEL")
args	Command arguments

Returns

Human-readable response from the server

Encodes the command and arguments using RESP-2 protocol, sends it to the server, receives the response, and decodes it to a human-readable format.

Parameters

command	Command to execute (e.g., "SET", "GET", "DEL")
args	Vector of command arguments

Returns

Human-readable response string or error message

3.1.2.4 is_connected()

```
bool Client::is_connected ( ) const
```

Check if client is connected to server.

Returns

true if connected, false otherwise

3.1.2.5 run_interactive()

```
void Client::run_interactive (
            std::function< void(const std::string &)> on_response = nullptr )
```

Run an interactive client session.

Parameters

on_response	Callback for displaying responses

Prompts the user for commands, parses them, executes them on the server, and displays the results. Continues until the user enters "exit" or "quit".

Parameters

on_response	Optional callback function to handle response display

3.2 Connection Class Reference

Manages a single client connection.

```
#include <connection.h>
```

Public Types

- enum class State { CONNECTED , CLOSING , CLOSED }

    Connection state enumeration.

Public Member Functions

- bool has_pending_writes () const

    Check if connection has pending data to write.
- Connection (int fd, Server ∗server)

    Construct a new Connection object.
- ∼Connection ()

    Destroy the Connection object.
- bool handle_read ()

    Handle data available to read.
- bool handle_write ()

    Handle socket ready for writing.
- void add_response (const std::string &response)

    Add response to output buffer.
- bool check_timeout (std::chrono::milliseconds timeout_ms)

    Check if connection has timed out.
- int get_fd () const

    Get socket file descriptor.
- State get_state () const

    Get connection state.

3.2.1 Detailed Description

Handles all aspects of a client connection including:

- Buffering for partial reads and writes
- Command parsing and execution
- Connection state management
- Activity tracking for timeout detection

Each Connection instance is associated with a specific client socket and communicates with the server to execute commands.

3.2.2 Member Enumeration Documentation

3.2.2.1 State

```
enum class Connection::State  [strong]
```

Represents the lifecycle states of a client connection

Enumerator

CONNECTED	Connection established and active.
CLOSING	Connection is being closed.
CLOSED	Connection is closed.

3.2.3 Constructor & Destructor Documentation

3.2.3.1 Connection()

```
Connection::Connection (
            int fd,
            Server * server )
```

Constructs a new Connection object.

Initializes a connection with the specified socket file descriptor and server reference. Sets initial state to CONNECTED.

Parameters

fd	Socket file descriptor for this connection
server	Pointer to server instance that owns this connection

Initializes a connection with the provided socket file descriptor and server pointer. Sets the connection state to CONNECTED and records the current time as the last activity.

Parameters

fd	Socket file descriptor for this connection
server	Pointer to the server instance that owns this connection

3.2.3.2 ∼Connection()

```
Connection::∼Connection ( )
```

Destroys the Connection object.

Closes the socket if still open and releases resources

Closes the socket file descriptor if it's still open

3.2.4 Member Function Documentation

3.2.4.1 add_response()

```
void Connection::add_response (
            const std::string & response )
```

Adds a response to the output queue.

Queues a response to be sent to the client when the socket is ready for writing. The server should register for write events when this method adds items to an empty queue.

Parameters

response	Response data to send to the client

Enqueues a response to be sent to the client when the socket is ready for writing. The server needs to update the epoll registration to include EPOLLOUT when there are pending responses.

Parameters

response	The response string to send to the client

3.2.4.2 check_timeout()

```
bool Connection::check_timeout (
             std::chrono::milliseconds timeout_ms )
```

Checks if the connection has timed out.

Compares the time since last activity against the provided timeout value to determine if the connection should be considered inactive.

Parameters

timeout_ms	Maximum allowed idle time in milliseconds

Returns

true if connection has been idle longer than timeout_ms, false otherwise

Compares the elapsed time since the last activity with the provided timeout value to determine if the connection should be considered timed out.

Parameters

timeout_ms	Timeout duration in milliseconds

Returns

true if the connection has timed out, false otherwise

3.2.4.3 get_fd()

```
int Connection::get_fd ( ) const  [inline]
```

Returns the file descriptor associated with this connection. Used by the server for epoll management.

Returns

int Socket file descriptor

3.2.4.4 get_state()

```
State Connection::get_state ( ) const  [inline]
```

Returns the current state of the connection (CONNECTED, CLOSING, CLOSED). Used by the server to determine how to handle the connection.

Returns

State Current connection state

3.2.4.5 handle_read()

```
bool Connection::handle_read ( )
```

Handles data available for reading from the socket.

Called by the server when the socket is ready for reading. Reads data from the socket, appends to input buffer, and processes any complete commands found in the buffer.

Returns

true on success or would block, false on error/connection close

Performs a non-blocking read from the socket, appends data to the input buffer, and processes any complete commands found in the buffer. Implements protection against buffer overflow attacks by limiting the maximum buffer size.

Returns

true if read was successful or would block, false on error or connection closed

3.2.4.6 handle_write()

```
bool Connection::handle_write ( )
```

Handles socket ready for writing.

Called by the server when the socket is ready for writing. Writes pending data from the output queue to the socket, handling partial writes appropriately.

Returns

true on success or would block, false on error

Sends data from the output queue to the client. Handles partial writes by keeping track of what's been sent and what remains to be sent.

Returns

true if write was successful or would block, false on error

3.2.4.7 has_pending_writes()

```
bool Connection::has_pending_writes ( ) const  [inline]
```

Used by the server to determine whether to register the socket for write events in epoll.

Returns

true if there is data in the output queue, false otherwise

3.3 RespProtocol Class Reference

Encoder/decoder for RESP-2 protocol.

```
#include <resp.h>
```

Classes

- class RespValue

  Represents a RESP value of any supported type.

Public Types

- enum class Type {
  SIMPLE_STRING , ERROR , INTEGER , BULK_STRING ,
  ARRAY }

  RESP data types.

Static Public Member Functions

- static std::string encode (const RespValue &value)

  Encode a RESP value to string for transmission.
- static std::string encodeCommand (const std::string &command, const std::vector< std::string > &args={})

  Encode a command with arguments to RESP array format.
- static std::optional< RespValue > parse (const std::string &data, size_t &bytes_consumed)

  Parse RESP data from input buffer.

3.3.1 Detailed Description

Provides static methods for encoding and parsing data in the RESP-2 format. The implementation is focused on efficiency and correctness, supporting all five RESP data types and handling edge cases like null values and incremental parsing.

3.3.2 Member Enumeration Documentation

3.3.2.1 Type

```
enum class RespProtocol::Type  [strong]
```

Enumerates the five data types defined in the RESP-2 protocol specification. Each type has a specific wire format and usage scenarios.

Enumerator

SIMPLE_STRING	Simple string prefixed with "+" (e.g., "+OK\r\n")
ERROR	Error message prefixed with "-" (e.g., "-ERR message\r\n")
INTEGER	Integer prefixed with ":" (e.g., ":1000\r\n")
BULK_STRING	Bulk string prefixed with "$" (e.g., "$6\r\nfoobar\r\n")
ARRAY	Array prefixed with "*" (e.g., "*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n")

3.3.3 Member Function Documentation

3.3.3.1 encode()

```
std::string RespProtocol::encode (
             const RespValue & value )  [static]
```

Encode a RESP value to wire format.

Converts a RespValue object to its wire format representation according to the RESP-2 protocol specification. The resulting string can be sent over a network connection.

Parameters

value	RESP value to encode

Returns

String encoded in RESP format

Exceptions

std::runtime_error	if encoding fails or if value has an unknown type

Converts the given RESP value object to its string representation according to the RESP-2 protocol specification

Parameters

value	RESP value to encode

Returns

String containing the RESP-2 encoded data

Exceptions

std::runtime_error	if the value has an unknown type

3.3.3.2 encodeCommand()

```
std::string RespProtocol::encodeCommand (
            const std::string & command,
            const std::vector< std::string > & args = {} )  [static]
```

Encode a command with arguments as a RESP array.

Convenience method for creating a command in RESP format. Commands are represented as arrays where the first element is the command name and subsequent elements are arguments.

Parameters

command	Command name (e.g., "SET", "GET", "DEL")
args	Command arguments

Returns

String encoded in RESP format

Formats a command and its arguments as a RESP array of bulk strings

Parameters

command	Command name
args	Command arguments

Returns

RESP-encoded command string

3.3.3.3 parse()

```
std::optional< RespProtocol::RespValue > RespProtocol::parse (
            const std::string & data,
            size_t & bytes_consumed )  [static]
```

Parse RESP-2 data from a string.

Attempts to parse a complete RESP value from the provided data buffer. If a complete value cannot be parsed (e.g., due to incomplete data), returns std::nullopt. This enables incremental parsing of RESP protocol data.

Parameters

	data	Input buffer containing RESP data
out	*bytes_consumed*	Number of bytes consumed from input

Returns

Parsed RESP value if complete, or std::nullopt if more data needed

Exceptions

std::runtime_error	if the data contains invalid RESP format

Incrementally parses RESP-2 formatted data from the input string. Returns std::nullopt if more data is needed to complete parsing.

Parameters

	data	String containing RESP-2 formatted data
`out`	*bytes_consumed*	Output parameter that will contain the number of bytes processed

Returns

std::optional<RespValue> Parsed value if complete, std::nullopt if more data needed

Exceptions

std::runtime_error	if the data contains invalid RESP format

3.4 RespProtocol::RespValue Class Reference

Represents a RESP value of any supported type.

```
#include <resp.h>
```

Public Member Functions

- RespValue ()

  Construct a null RESP value.
- Type getType () const

  Get the type of this RESP value.
- bool isNull () const

  Check if this RESP value is null.
- std::string getString () const

  Get string value (for SIMPLE_STRING, ERROR, BULK_STRING)
- int64_t getInteger () const

  Get integer value (for INTEGER)
- const std::vector< RespValue > & getArray () const

  Get array values (for ARRAY)

Static Public Member Functions

- static RespValue createSimpleString (const std::string &value)

  Construct a Simple String RESP value.
- static RespValue createError (const std::string &value)

  Construct an Error RESP value.
- static RespValue createInteger (int64_t value)

  Construct an Integer RESP value.
- static RespValue createBulkString (const std::string &value)

  Construct a Bulk String RESP value.
- static RespValue createNullBulkString ()

  Construct a Null Bulk String RESP value.
- static RespValue createArray (const std::vector< RespValue > &values)

  Construct an Array RESP value.
- static RespValue createNullArray ()

  Construct a Null Array RESP value.

3.4.1 Detailed Description

This class encapsulates a value in the RESP protocol, handling all five data types. It provides type-safe access to the value contents through getter methods that perform runtime type checking. The class supports null values for both Bulk Strings and Arrays.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 RespValue()

```
RespProtocol::RespValue::RespValue ( )  [inline]
```

Default constructor creates a null Bulk String value. Equivalent to $-1\r
in RESP protocol.

3.4.3 Member Function Documentation

3.4.3.1 createArray()

```
RespProtocol::RespValue RespProtocol::RespValue::createArray (
             const std::vector< RespValue > & values )  [static]
```

Create an Array RESP value.

Creates an Array value containing other RESP values. Arrays can contain mixed types and are used for commands and complex responses.

Parameters

values	Vector of RESP values to include in the array

Returns

RespValue object representing an Array

Parameters

values	Vector of RESP values to include in the array

Returns

RespValue object representing an Array

3.4.3.2 createBulkString()

```
RespProtocol::RespValue RespProtocol::RespValue::createBulkString (
            const std::string & value )  [static]
```

Create a Bulk String RESP value.

Creates a Bulk String value in RESP format. Bulk strings are binary-safe and can contain any byte sequence including null bytes.

Parameters

value	String content (can be binary)

Returns

RespValue object representing a Bulk String

Parameters

value	String content (can be binary)

Returns

RespValue object representing a Bulk String

3.4.3.3 createError()

```
RespProtocol::RespValue RespProtocol::RespValue::createError (
            const std::string & value )  [static]
```

Create an Error RESP value.

Creates an Error value prefixed with "-" in RESP format. Typically used to represent error conditions.

Parameters

value	Error message

Returns

RespValue object representing an Error

Parameters

value	Error message

Returns

RespValue object representing an Error

3.4.3.4 createInteger()

```
RespProtocol::RespValue RespProtocol::RespValue::createInteger (
            int64_t value )  [static]
```

Create an Integer RESP value.

Creates an Integer value prefixed with ":" in RESP format. Used to represent numeric values such as counters or response codes.

Parameters

value	Integer value

Returns

RespValue object representing an Integer

Parameters

value	Integer value

Returns

RespValue object representing an Integer

3.4.3.5 createNullArray()

```
RespProtocol::RespValue RespProtocol::RespValue::createNullArray ( )  [static]
```

Create a Null Array RESP value.

Creates a special Array value representing NULL. Encoded as "∗-1\r\n" in RESP protocol.

Returns

RespValue object representing a Null Array

RespValue object representing a Null Array (∗-1\r
)

3.4.3.6 createNullBulkString()

```
RespProtocol::RespValue RespProtocol::RespValue::createNullBulkString ( )  [static]
```

Create a Null Bulk String RESP value.

Creates a special Bulk String value representing NULL. Encoded as "$-1\r\n" in RESP protocol.

Returns

RespValue object representing a Null Bulk String

RespValue object representing a Null Bulk String ($-1\r
)

3.4.3.7 createSimpleString()

```
RespProtocol::RespValue RespProtocol::RespValue::createSimpleString (
             const std::string & value )  [static]
```

Create a Simple String RESP value.

Creates a Simple String value prefixed with "+" in RESP format. Simple strings cannot contain CR or LF characters.

Parameters

value	String content

Returns

RespValue object representing a Simple String

Parameters

value	String content

Returns

RespValue object representing a Simple String

3.4.3.8 getArray()

```
const std::vector< RespProtocol::RespValue > & RespProtocol::RespValue::getArray ( ) const
```

Get the array values stored in this RESP value.

Retrieves the vector of elements in this array. Only valid for the Array type.

Returns

Vector of RESP values

Exceptions

std::runtime_error	if type is not ARRAY or if array is null

Can only be called on ARRAY type

Returns

Reference to the vector of RESP values

Exceptions

std::runtime_error	if type is not ARRAY or array is null

3.4.3.9 getInteger()

```
int64_t RespProtocol::RespValue::getInteger ( ) const
```

Get the integer value stored in this RESP value.

Retrieves the integer content of this value. Only valid for the Integer type.

Returns

Integer value

Exceptions

std::runtime_error	if type is not INTEGER or if value is null

Can only be called on INTEGER type

Returns

The stored integer value

Exceptions

std::runtime_error	if type is not INTEGER or value is null

3.4.3.10 getString()

```
std::string RespProtocol::RespValue::getString ( ) const
```

Get the string value stored in this RESP value.

Retrieves the string content of this value. Only valid for string-compatible types (Simple String, Error, Bulk String).

Returns

String value

Exceptions

std::runtime_error	if type is not string-compatible or if value is null

Can only be called on SIMPLE_STRING, ERROR, or BULK_STRING types

Returns

The stored string value

Exceptions

std::runtime_error	if type is not string-compatible or value is null

3.4.3.11 getType()

```
Type RespProtocol::RespValue::getType ( ) const [inline]
```

Returns the type enum indicating which of the five RESP types this value represents.

Returns

Type enum value

3.4.3.12 isNull()

```
bool RespProtocol::RespValue::isNull ( ) const [inline]
```

Returns whether this value is a null representation. Only relevant for Bulk String and Array types.

Returns

true if null, false otherwise

3.5 Server Class Reference

Implements a TCP server using epoll for efficient I/O multiplexing.

```
#include <server.h>
```

Public Types

- using CommandHandler = std::function< std::string(const std::vector< std::string > &)>

  Function type for command handlers.

Public Member Functions

- Server (int port=9001, int max_connections=1024)

  Construct a TCP server.
- ~Server ()

  Destroy the server and release resources.
- bool init ()

  Initialize the server.
- void run ()

  Start the server and run the event loop.
- void stop ()

  Stop the server gracefully.
- void register_command (const std::string &command, CommandHandler handler)

  Register a command handler.
- std::string execute_command (const std::string &command, const std::vector< std::string > &args)

  Execute a command and return the response.

3.5.1 Detailed Description

The Server class is responsible for:

- Setting up and managing a TCP socket listening on a configured port
- Using epoll() to efficiently handle multiple client connections
- Accepting new connections and creating Connection objects to manage them
- Dispatching received commands to appropriate handlers
- Maintaining the server lifecycle (initialization, running, shutdown)
- Integrating with the Storage Engine to execute commands

This implementation follows an event-driven architecture using edge-triggered epoll for optimal performance with non-blocking I/O operations.

3.5.2 Member Typedef Documentation

3.5.2.1 CommandHandler

```
Server::CommandHandler
```

Defines the signature for command handler functions that process client commands and return RESP-formatted responses.

3.5.3 Constructor & Destructor Documentation

3.5.3.1 Server()

```
Server::Server (
            int port = 9001,
            int max_connections = 1024 )
```

Constructs a new Server instance.

Initializes a new server instance with the specified configuration. The server won't start listening until init() is called.

Parameters

port	Port to listen on (default: 9001)
max_connections	Maximum number of concurrent connections (default: 1024)

Initializes server parameters but does not start listening until init() is called

Parameters

port	Port number to listen on (default: 9001)
max_connections	Maximum number of concurrent connections allowed

3.5.3.2 ∼Server()

```
Server::∼Server ( )
```

Destroys the Server instance.

Closes all connections, releases socket and epoll resources, and ensures the server is properly shut down.

Ensures proper cleanup by calling stop()

3.5.4 Member Function Documentation

3.5.4.1 execute_command()

```
std::string Server::execute_command (
            const std::string & command,
            const std::vector< std::string > & args )
```

Executes a command and returns the response.

Looks up the appropriate handler for the command and executes it, returning the RESP-formatted response. If the command is not recognized or an error occurs, returns an appropriate error response.

Parameters

command	Command name (e.g., "SET", "GET", "DEL")
args	Vector of command arguments

Returns

Response string in RESP format

Looks up the appropriate handler for the command and executes it, returning the RESP-formatted response or an error message.

Parameters

command	Command name (e.g., "SET", "GET", "DEL")
args	Vector of command arguments

Returns

RESP-formatted response string

3.5.4.2 init()

```
bool Server::init ( )
```

Initializes the server.

Creates the listening socket, binds to the configured port, initializes epoll, and registers command handlers. This must be called before run().

Returns

true on successful initialization, false on failure

Sets up the listening socket, binds to the specified port, initializes epoll, and registers command handlers for SET, GET, and DEL operations.

Returns

true if initialization was successful, false otherwise

3.5.4.3 register_command()

```
void Server::register_command (
            const std::string & command,
            CommandHandler handler )
```

Registers a command handler function.

Associates a command name (e.g., "SET") with a handler function that will be called when clients send that command. The handler receives the command arguments and returns a RESP-formatted response.

Parameters

command	Command name (e.g., "SET", "GET", "DEL") - case-sensitive
handler	Function to handle the command

Associates a command name with a function that will process it

Parameters

command	Command name (e.g., "SET", "GET", "DEL")
handler	Function to handle the command

3.5.4.4 run()

```
void Server::run ( )
```

Runs the server main event loop.

Enters the main server loop, listening for events using epoll and processing them accordingly. This method blocks until stop() is called from another thread or a signal handler.

Continuously monitors for events using epoll, accepting new connections and processing I/O for existing connections. This method blocks until stop() is called.

3.5.4.5 stop()

```
void Server::stop ( )
```

Stops the server gracefully.

Terminates the main event loop, closes all client connections, and releases server resources. Can be called from a signal handler or another thread to shut down the server.

Closes all client connections, cleans up resources, and terminates the event loop

Chapter 4

File Documentation

4.1 client.cpp File Reference

Implementation of BLINK DB client.

```
#include "client.h"
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <iostream>
#include <sstream>
#include <cstring>
#include <algorithm>
#include <netdb.h>
```

Variables

- const size_t **MAX_BUFFER_SIZE**

  Maximum receive buffer size.

4.1.1 Detailed Description

This file contains the implementation of the Client class which establishes a connection to the BLINK DB server, encodes and sends commands using the RESP-2 protocol, and decodes responses from the server.

4.2 client.h File Reference

Simple client for BLINK DB that communicates using RESP-2 protocol.

```
#include <string>
#include <vector>
#include <functional>
```

Classes

- class Client

  Client for BLINK DB server using RESP-2 protocol.

4.3 client.h

Go to the documentation of this file.

```
#pragma once

#include <string>
#include <vector>
#include <functional>

class Client {
public:
    Client(const std::string& host = "localhost", int port = 9001);

    ~Client();

    bool connect();

    void disconnect();

    bool is_connected() const;

    std::string execute(const std::string& command, const std::vector<std::string>& args = {});

    void run_interactive(std::function<void(const std::string&)> on_response = nullptr);

private:
    std::string host_;
    int port_;
    int socket_fd_;

    std::string encode_command(const std::string& command, const std::vector<std::string>& args);

    std::string decode_response(const std::string& resp_data);

    bool send_data(const std::string& data);

    std::string receive_data();

    bool parse_command_line(const std::string& command_line, std::string& command,
std::vector<std::string>& args);
};

```

4.4 client_main.cpp File Reference

Entry point for BLINK DB client.

```
#include "client.h"
#include <iostream>
#include <string>
```

Functions

- int main (int argc, char *argv[ ])

  Main entry point for the BLINK DB client application.

4.4.1 Detailed Description

This file implements the main entry point for the BLINK DB client application, handling command-line argument parsing, server connection, and interactive command processing.

4.4.2 Function Documentation

4.4.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

Initializes the client, connects to the specified server, and runs the interactive client interface for sending commands.

Parameters

argc	Number of command-line arguments
argv	Array of command-line arguments

Returns

0 on successful execution, 1 on error

4.5 connection.cpp File Reference

Implementation of Connection management layer.

```
#include "connection.h"
#include "server.h"
#include "resp.h"
#include <unistd.h>
#include <errno.h>
#include <cstring>
#include <iostream>
#include <sys/socket.h>
#include <algorithm>
```

Variables

- const size_t **MAX_READ_SIZE**

  Maximum size for a single read operation.
- const size_t **MAX_INPUT_BUFFER_SIZE**

  Maximum allowed input buffer size to prevent memory exhaustion attacks.

4.5.1 Detailed Description

This file implements the Connection class which manages individual client connections to the BLINK DB server. It handles buffer management for partial reads/writes, connection state tracking, command processing, and timeout detection.

4.6 connection.h File Reference

Connection management layer for BLINK DB.

```
#include <string>
#include <vector>
#include <deque>
#include <chrono>
```

Classes

- class Connection

  Manages a single client connection.

4.6.1 Detailed Description

Manages client connections including buffering for partial reads/writes, connection state tracking, and timeout handling. Acts as an intermediary between the server's socket handling and the RESP protocol processing. Implements non-blocking I/O patterns to efficiently handle multiple concurrent clients.

4.7 connection.h

Go to the documentation of this file.

```
#pragma once

#include <string>
#include <vector>
#include <deque>
#include <chrono>

// Forward declarations
class Server;
class RespProtocol;

class Connection {
public:
    bool has_pending_writes() const { return !output_queue_.empty(); }

    enum class State {
        CONNECTED,
        CLOSING,
        CLOSED
    };

    Connection(int fd, Server* server);

    ~Connection();

    bool handle_read();

    bool handle_write();

    void add_response(const std::string& response);

    bool check_timeout(std::chrono::milliseconds timeout_ms);

    int get_fd() const { return fd_; }

    State get_state() const { return state_; }

private:
    int fd_;
    Server* server_;
    State state_;
    std::string input_buffer_;
    std::deque<std::string> output_queue_;
    std::chrono::steady_clock::time_point last_activity_;

    bool process_commands();

    void update_last_activity();

    void reset();
};

```

4.8 main.cpp File Reference

Entry point for BLINK DB server (Part B)

```
#include "server.h"
#include <iostream>
#include <csignal>
#include <cstring>
```

Functions

- void signal_handler (int sig)

  Signal handler for graceful shutdown.
- void print_usage (const char ∗prog_name)

  Print usage information.
- int main (int argc, char ∗argv[ ])

  Main function.

Variables

- Server ∗ **g_server**

  Global server instance for signal handling.

4.8.1 Detailed Description

Initializes and runs the BLINK DB server with the TCP layer and RESP-2 protocol implementation. This file handles:

- Command-line argument parsing
- Signal handling for graceful shutdown
- Server initialization and execution
- Error reporting and usage information

The server implements a Redis-compatible key-value store that communicates using the RESP-2 protocol and listens on port 9001 by default.

4.8.2 Function Documentation

4.8.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Entry point for the BLINK DB server. Processes command-line arguments, sets up signal handlers, initializes the server, and starts the main event loop. The server continues running until stopped by a signal or fatal error.

Server configuration can be customized through command-line options including:

- Port number (-p, –port)
- Maximum concurrent connections (-c, –connections)

Parameters

argc	Number of command-line arguments
argv	Array of command-line argument strings

Returns

0 on successful execution and graceful shutdown, 1 on error

4.8.2.2 print_usage()

```
void print_usage (
            const char * prog_name )
```

Displays the command-line options and their descriptions to help users understand how to configure the server.

Parameters

prog_name	Name of the program executable

4.8.2.3 signal_handler()

```
void signal_handler (
            int sig )
```

Handles SIGINT (Ctrl+C) and SIGTERM signals to ensure the server shuts down gracefully, closing all connections and releasing resources properly.

Parameters

sig	Signal number received from the operating system

4.9 resp.cpp File Reference

Implementation of RESP-2 protocol encoder/decoder.

```
#include "resp.h"
#include <sstream>
#include <stdexcept>
```

Variables

- const std::string **CRLF**

  CRLF sequence used in RESP-2 protocol.

4.9.1 Detailed Description

This file implements the Redis Serialization Protocol (RESP-2) for BLINK DB. It provides classes and methods for encoding and decoding data in the RESP-2 format, supporting all five data types: Simple Strings, Errors, Integers, Bulk Strings, and Arrays. The implementation handles incremental parsing, allowing for processing of partial messages.

4.10 resp.h File Reference

RESP-2 protocol encoder/decoder for BLINK DB.

```
#include <string>
#include <vector>
#include <optional>
```

Classes

- class RespProtocol

  Encoder/decoder for RESP-2 protocol.
- class RespProtocol::RespValue

  Represents a RESP value of any supported type.

4.10.1 Detailed Description

Implements the Redis Serialization Protocol (RESP-2) for communication between clients and the BLINK DB server. Handles all five RESP data types: Simple Strings, Errors, Integers, Bulk Strings, and Arrays.

This implementation provides both encoding and incremental parsing capabilities for complete RESP-2 protocol support, enabling efficient client-server communication with the same wire format used by Redis.

4.11 resp.h

Go to the documentation of this file.

```
#pragma once

#include <string>
#include <vector>
#include <optional>

class RespProtocol {
public:
    enum class Type {
        SIMPLE_STRING,
        ERROR,
        INTEGER,
        BULK_STRING,
        ARRAY
    };

    class RespValue {
    public:
        RespValue() : type_(Type::BULK_STRING), is_null_(true) {}

        static RespValue createSimpleString(const std::string& value);

        static RespValue createError(const std::string& value);

        static RespValue createInteger(int64_t value);

        static RespValue createBulkString(const std::string& value);

        static RespValue createNullBulkString();

        static RespValue createArray(const std::vector<RespValue>& values);

        static RespValue createNullArray();

        Type getType() const { return type_; }

        bool isNull() const { return is_null_; }

        std::string getString() const;

        int64_t getInteger() const;

        const std::vector<RespValue>& getArray() const;

    private:
        Type type_;
        bool is_null_ = false;
        std::string string_value_;
        int64_t int_value_ = 0;
        std::vector<RespValue> array_values_;

        RespValue(Type type, bool is_null) : type_(type), is_null_(is_null) {}
    };

    static std::string encode(const RespValue& value);

    static std::string encodeCommand(const std::string& command,
                                     const std::vector<std::string>& args = {});

    static std::optional<RespValue> parse(const std::string& data, size_t& bytes_consumed);

private:
    static std::string encodeSimpleString(const std::string& value);

    static std::string encodeError(const std::string& value);

    static std::string encodeInteger(int64_t value);

    static std::string encodeBulkString(const std::string& value);

    static std::string encodeNullBulkString();

    static std::string encodeArray(const std::vector<RespValue>& values);

    static std::string encodeNullArray();

    static std::optional<RespValue> parseSimpleString(const std::string& data, size_t& pos);

    static std::optional<RespValue> parseError(const std::string& data, size_t& pos);

    static std::optional<RespValue> parseInteger(const std::string& data, size_t& pos);
    static std::optional<RespValue> parseBulkString(const std::string& data, size_t& pos);

    static std::optional<RespValue> parseArray(const std::string& data, size_t& pos);

    static size_t findCRLF(const std::string& data, size_t start);
};

```

4.12 server.cpp File Reference

Implementation of TCP server with epoll() for efficient I/O multiplexing.

```
#include "server.h"
#include "connection.h"
#include "resp.h"
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <cstring>
#include <errno.h>
```

4.12.1 Detailed Description

This file implements a high-performance TCP server using the epoll() mechanism to handle multiple concurrent client connections efficiently. It integrates with the Storage Engine from Part A and uses the RESP-2 protocol for client communication.

4.13 server.h File Reference

TCP server with epoll() for I/O multiplexing.

```
#include <string>
#include <unordered_map>
#include <functional>
#include <sys/epoll.h>
#include <atomic>
#include <vector>
#include "StorageEngine.h"
```

Classes

- class Server

    Implements a TCP server using epoll for efficient I/O multiplexing.

4.13.1 Detailed Description

This header defines the Server class, which is the core component of the BLINK DB networking layer. It implements a high-performance TCP server using the epoll() mechanism for efficient I/O multiplexing, allowing it to handle thousands of concurrent connections with minimal resource usage. The server integrates with the storage engine from Part A and communicates with clients using the RESP-2 protocol.

4.14 server.h

Go to the documentation of this file.

```
#pragma once

#include <string>
#include <unordered_map>
#include <functional>
#include <sys/epoll.h>
#include <atomic>
#include <vector>
#include "StorageEngine.h"

// Forward declaration
class Connection;
class RespProtocol;

class Server {
public:
    using CommandHandler = std::function<std::string(const std::vector<std::string>&)>;

    Server(int port = 9001, int max_connections = 1024);

    ~Server();

    bool init();

    void run();

    void stop();

    void register_command(const std::string& command, CommandHandler handler);

    std::string execute_command(const std::string& command, const std::vector<std::string>& args);

private:
    int port_;
    int listen_fd_;
    int epoll_fd_;
    int max_connections_;
    std::atomic<bool> running_;

    StorageEngine storage_engine_;
    std::unordered_map<std::string, CommandHandler> command_handlers_;
    std::unordered_map<int, Connection*> connections_;

    bool set_nonblocking(int fd);

    bool accept_connection();

    void handle_event(int fd, uint32_t events);

    void close_connection(int fd);
};

```

Index