CPT Chat Server/Client Design

Jordan Godau

Giwoun Bae

April 11, 2022

Structures

Cpt protocol related structs

cpt_server_info

Field	Purpose
current_id	*Current user ID*
channel_id	*Current channel ID the user is in*
cpt_response * res	A pointer to cpt_resonse struct for the server response
Channels * ch	*List of channels*
Channel* gc	Pointer to global channel

cpt_client_info

Holds all relevant information for client side application programming.

Field	Purpose
fd	Socket fd of the client
code	Server response code
ip	Client ip information
port	Client port information
name	Client username
channel	Active user channel of the client
packet	A pointer to cpt packet struct
channels	List of users the client is part of

cpt_request

Field	Purpose
version	Version of the protocol
command	command
channel_id	Channel_id the user is currently on
msg_len	Message length
msg	Message

cpt_response

Contains properties for preparing cpt response packets

Field	Purpose
uint8_t code	Server response code
uint8_t * data	Server response data

user_state

Hold information about user_state on application framework for client.

Field	Purpose
name	Name of the client
Logged in	Boolean state of the user logged in
channel	Channel the user is currently in.
CptClientInfo * client_info	Cpt_client_info struct for client information.

Utility objects

Channel object (individual channel)

Field	Purpose
fd	File descriptor
id	Id of the channel
users	List of users that

Channels object (multiple channels)

Field	Purpose
length	Number of channels
channel_head	The first channel of the list (LinkedList)
channel_tail	The last channel of the list (LinkedList)

channel_node object

Field	Purpose
chan	Current Channel
next_chan	The next channel in the linked list
channel_size	Size of the channel

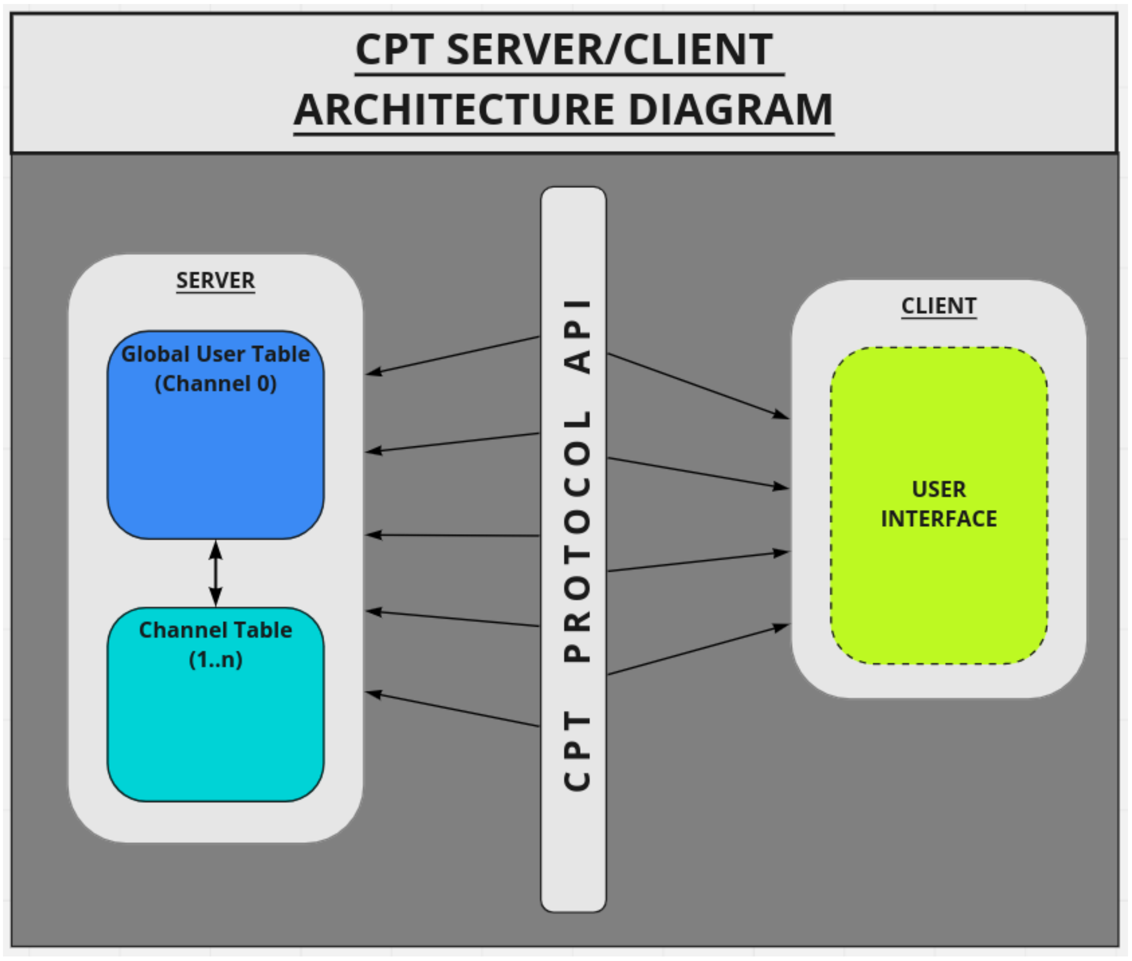
Miscellaneous

Valid commands

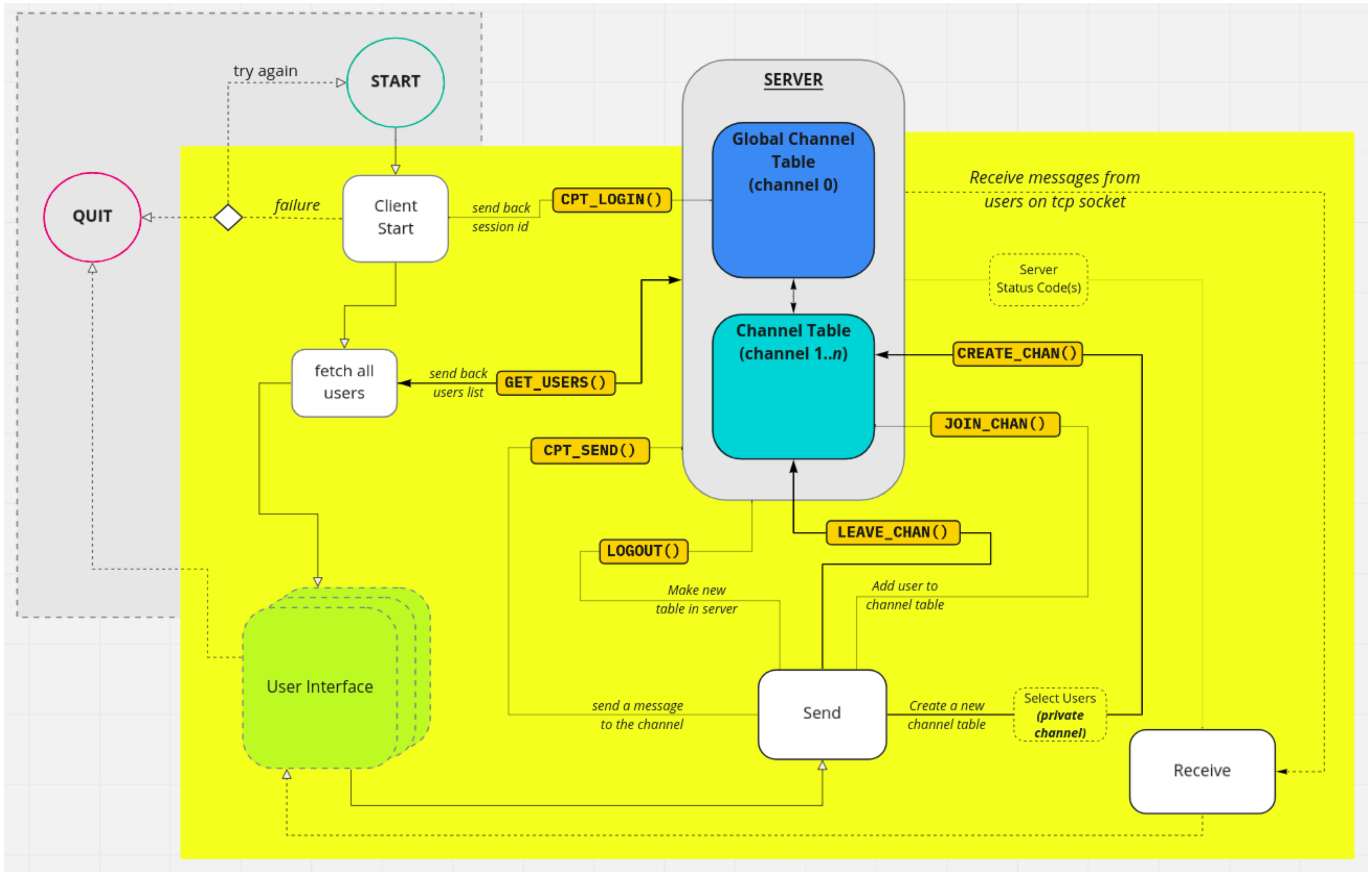
Command	Purpose
@get-users	Returns list of users in the channel
@create-channel	Create a channel with users
@join-channel	Join currently open channel
@leave-channel	Leave current channel
@join-vchannel	Join voice channel
@logout	Disconnect from server
@menu	Prints menu

CPT server/client architecture

Overview of CPT server/client


CPT SERVER/CLIENT ARCHITECTURE DIAGRAM
SERVER
Global User Table (Channel 0)
Channel Table (1..n)
CPT PROTOCOL API
CLIENT
USER INTERFACE

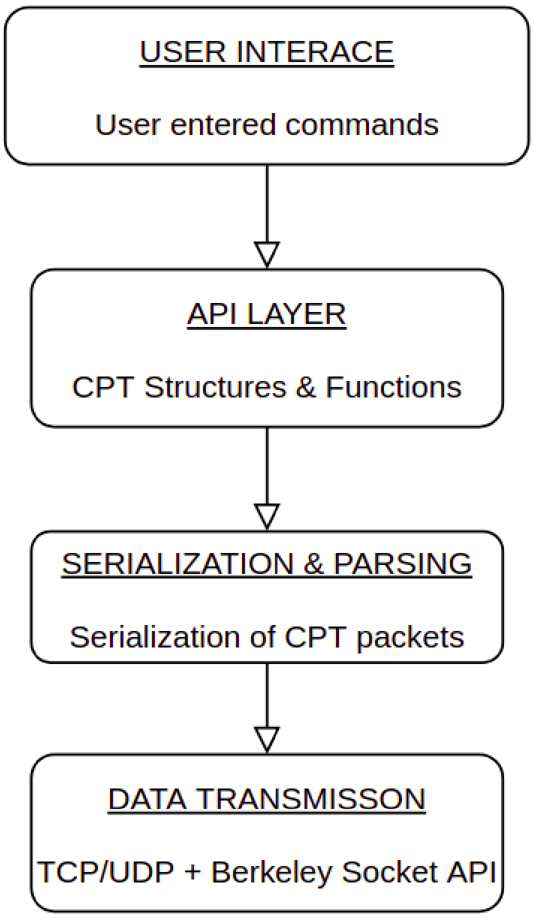
CPT Client-side Design



Code Architecture Design

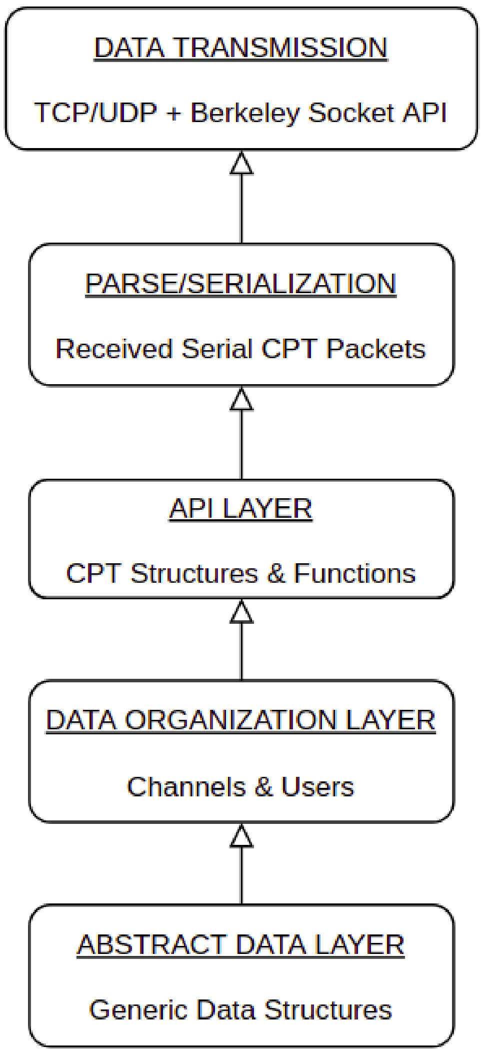
Client-side code architecture

The code structure is designed to have layers according to its responsibility. For the client side, the input obtained from the user interface is processed in the API layer to follow CPT protocol version 2.0. The package structure is serialised to be put on data transmission through the TCP protocol. For the CPT protocol version 2.0, the voice transmission is enabled. The voice transmission is transmitted over UDP protocol.


USER INTERACE
User entered commands
API LAYER
CPT Structures & Functions
SERIALIZATION & PARSING
Serialization of CPT packets
DATA TRANSMISSON
TCP/UDP + Berkeley Socket API

Server-side code architecture

The code structure is designed to have layers according to its responsibility. The server-side code is responsible for managing the list of channels created and deleted. That responsibility is done through the data organisation layer. The data structure used for this version of the application is linkedlist.



Functions

Common utility functions

linked-list

```
/**
 * Create a Node.
 *
 * Use a void pointer to a struct to
 * initialise and return a Node.
 *
 * @param data      Void pointer to a struct.
 * @param data_size Size of <data> in bytes.
 * @return The new Node.
 */
Node * create_node(void * data, size_t data_size);


/**
 * Destroy a Node.
 *
 * Destroys the Node, freeing any allocated
 * memory and setting pointers to NULL.
 *
 * @param node
 */
void destroy_node(Node * node);


/**
 * Initialise a LinkedList.
 *
 * Initialises a LinkedList object using
 * <data> as head. Function will allocate
 * necessary memory, initialise data members,
 * and set the HEAD/TAIL pointers for the list.
 *
 * @param data
 * @param data_size
```

```c
 * @return
 */
LinkedList * init_list_data(void * data, size_t data_size);


/**
 * Initialise a LinkedList.
 *
 * Initialises a LinkedList object using
 * <data> as head. Function will allocate
 * necessary memory, initialise data members,
 * and set the HEAD/TAIL pointers for the list.
 *
 * @param data
 * @param data_size
 * @return
 */
LinkedList * init_list_node(Node * node);


/**
 * Destroy the LinkedList.
 *
 * Destroys the LinkedList, freeing any allocated
 * memory and setting pointers to NULL.
 *
 * @param list A LinkedList object.
 */
void destroy_list(LinkedList * list);


/**
 * Iterate through every node of LinkedList.
 *
 * @param list      A LinkedList object.
 * @param consumer  A Consumer function pointer.
 */
void for_each(LinkedList * list, Consumer consumer);


/**
```

```c
 * Get the head node from the linked list.
 *
 * Retrieves the de-referenced head node from the
 * given linked list. If the linked list has no
 * head node, function returns NULL.
 *
 * @param list An initialized LinkedList object.
 * @return The head Node object, or NULL on failure.
 */
Node * get_head_node(LinkedList * list);


/**
 * Push a node to the front of hte linked list.
 *
 * @param list      A LinkedList object.
 * @param data      Data for the LinkedList.
 * @param data_size Size od the data in bytes.
 */
int push_data(LinkedList * list, void * data, size_t data_size);


/**
 * Push a node to the front of hte linked list.
 *
 * @param list      A LinkedList object.
 * @param data      Data for the LinkedList.
 * @param data_size Size od the data in bytes.
 */
int push_node(LinkedList * list, Node * node);


/**
 * Filter linked list based on input parameters.
 *
 * Filters a LinkedList object based on the <comparator>
 * and <params> given to the function. Returns a new
 * LinkedList list with matches on success. If no
 * matches are found, the function returns NULL.
 *
 * @param list       A LinkedList object.
 * @param comparator  A comparator function pointer.
 * @param params     The search params for comparator.
 * @param num_params The number of params in params.
 * @return A LinkedList or NULL.
 */
LinkedList * filter(LinkedList * list, Comparator comparator, void *
params, size_t num_params);


/**
 * Delete a Node from a LinkedList object.
 *
 * Deletes a node from a linked list object,
 * returning 0 on success, and -1 on failure.
 *
 *  - If a <Supplier> is provided, the function will
 *    delete the first instance that matches the
 *    supplier's return value.
 *  - If a <Supplier> is not provided, the function will
 *    delete the node by comparing the addresses of each
 *    Node to the <target>
 *
 * @param list     A LinkedList object.
 * @param target   The target Node to delete.
 * @param supplier A supplier function pointer.
 * @return 1 on success, 0 on failure.
 */
int delete_node(LinkedList * list, Comparator comparator, void *
test_param);


/**
 * Find a node in the LinkedList.
 *
 * @param list        A LinkedList object.
 * @param comparator   comparator function pointer.
 * @param test_param  Test parameter for comparator.
 * @return Pointer to Node or NULL on failure.
 */
Node * find_node(LinkedList * list, Comparator comparator, void *
test_param);
```

CPT_API functions

Cpt_client

```
/**
 * Prepare a LOGIN request packet for the server.
 *
 * Prepares a LOGIN request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to
 * persist the client's information until cpt_logout() is called.
 *
 * @param cpt            CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param name           Client login name.
 * @return The size of the serialized packet in <serial_buf>.
*/
size_t cpt_login(void * client_info, uint8_t * serial_buf, char *
name);


/**
 * Prepare a GET_USERS request packet for the server.
 *
 * Prepares a GET_USERS request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to
 * send back a list of users specified by the <channel_id>.
 *
 * @param client_info    CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param channel_id     The ID of the CHANNEL to get users from.
 * @return Size of the resulting serialized packet in <serial_buf>
*/
size_t cpt_get_users(void * client_info, uint8_t * serial_buf,
uint16_t channel_id);
```

```
/**
 * Prepare a CREATE_CHANNEL request packet for the server.
 *
 * Prepares a CREATE_CHANNEL request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to create
 * a new channel.
 *
 *      > <user_list> may be optionally passed as user selection
 *        parameters for the new Channel.
 *      > If <members> is not NULL, it will be assigned to the
 *        MSG field of the packet.
 *
 * @param cpt            CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param user_list      Whitespace seperated user IDs as a string.
 * @return Size of the resulting serialized packet in <serial_buf>
*/
size_t cpt_create_channel(void * client_info, uint8_t * serial_buf,
char * user_list);


/**
 * Prepare a LOGOUT request packet for the server.
 *
 * Prepares a LOGOUT request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to remove
 * any instance of the requesting client's information.
 *
 * @param cpt            CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @return Size of the resulting serialized packet in <serial_buf>
*/
size_t cpt_logout(void * client_info, uint8_t * serial_buf);
```

```
/**
 * Prepare a SEND request packet for the server.
 *
 * Prepares a SEND request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to SEND
 * the message specified by <msg> to ever user in the channel
 * specified within the packet CHAN_ID field.
 *
 * @param client_info    CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param msg            Intended chat message.
 * @return Size of the resulting serialized packet in <serial_buf>.
*/
int cpt_send(void * client_info, uint8_t * serial_buf, char * msg);

//TODO this doxygen needs to be updated.
/**
 * Prepare a JOIN_CHANNEL request packet for the server.
 *
 * Prepares a JOIN_CHANNEL request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to instruct the server to add
 * the client's information to an existing channel.
 *
 * @param client_info    CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param channel_id     The target channel id.
 * @return Size of the resulting serialized packet in <serial_buf>
*/
size_t cpt_join_channel(void * client_info, uint8_t * serial_buf,
uint16_t channel_id);


/**
 * Prepare a LEAVE_CHANNEL request packet for the server.
 *
 * Prepares a LEAVE_CHANNEL request to the server. If successful,
 * the resulting data in <serial_buf> will contain a CPT packet
 * with the necessary information to remove the client's information
 * from the channel specified by <channel_id>.
 *
 * @param client_info    CPT packet information and any other
necessary data.
 * @param serial_buf     A buffer intended for storing the result.
 * @param channel_id     The target channel id.
 * @return Size of the resulting serialized packet in <serial_buf>
*/
size_t cpt_leave_channel(void * client_info, uint8_t * serial_buf,
uint16_t channel_id);


/**
 * Create a CptClientInfo object.
 *
 * Contains necessary information to
 * to create a TCP connection with the
 * server and send data back and forth.
 *
 * Initializes all necessary fields and
 * allocates any memory necessary for the
 * object.
 *
 * @param port  Server port number.
 * @param ip    Server IP or URL.
 * @return CptClientInfo object.
 */
CptClientInfo * cpt_init_client_info(char * port, char * ip);


/**
 * Destroy CptRequestInfo object.
 *
 * Frees any memory necessary for the
 * object and set all pointers to NULL.
 *
 * @param packet_info CptRequestInfo object.
 */
void cpt_destroy_client_info(CptClientInfo * client_info);
```

Cpt_packet_builder

```
/**
 * Initialize CptRequest object.
 *
 * Dynamically allocates a cpt struct and
 * initializes all fields.
 *
 * @return Pointer to cpt struct.
*/
CptRequest * cpt_request_init(void);


/**
 * Free all memory and set fields to null.
 *
 * @param req   Pointer to a cpt structure.
*/
void cpt_request_destroy(CptRequest * req);


/**
 * Set the command value for the cpt header block.
 *
 * @param cpt   Pointer to a cpt structure.
 * @param cmd   From enum commands.
*/
void cpt_request_cmd(CptRequest * cpt, uint8_t cmd);


/**
 * Set major and minor version for the cpt header block.
 *
 * @param cpt           Pointer to a cpt structure.
 * @param version_major From enum version.
 * @param version_minor From enum version.
*/
void cpt_request_version(CptRequest * cpt, uint8_t version_major,
uint8_t version_minor);


/**
 * Set the message length for the cpt header block.
 *
 * @param cpt       Pointer to a cpt structure.
 * @param msg_len   An 8-bit integer.
*/
void cpt_request_len(CptRequest * cpt, uint8_t msg_len);


/**
 * Set the channel id for the cpt header block.
 *
 * @param cpt           Pointer to a cpt structure.
 * @param channel_id    A 16-bit integer.
*/
void cpt_request_chan(CptRequest * cpt, uint16_t channel_id);


/**
 * Set the MSG field for the cpt packet.
 *
 * Also appropriately updates the MSG_LEN field.
 *
 * @param cpt  Pointer to a cpt structure.
 * @param msg  Pointer to an array of characters.
*/
void cpt_request_msg(CptRequest * cpt, char * msg);


/**
 * Check serialized cpt to see if it is a valid cpt block.\n\n
 *
 * @param packet    A serialized cpt protocol message.
 * @return 0 if no issues, otherwise CPT error code.
*/
```

```
int cpt_validate(void * packet);


/**
 * Convert CptRequest object to a string representation.\n\n
 *
 * Converts a CptRequest object to a string, returning
 * a char pointer to the dynamically allocated memory.
 * If fields are missing from the CptRequest, they will
 * be filled with default values to signify as such.
 *
 * If call fails for any reason, function returns NULL.
 *
 *
 * @param packet    A serialized cpt protocol message.
 * @return Char pointer to dynamically allocated string.
*/
char * cpt_to_string(CptRequest * cpt);


/**
 * Reset packet parameters.
 *
 * Reset the packet parameters,
 * and free memory for certain params.
 *
 * @param packet    A CptRequest struct.
*/
void cpt_request_reset(CptRequest * packet);


/**
 * Initialize CptResponse server-side packet.
 *
 * Initializes a CptResponse by initializing data
 * members, and returning a dynamically allocated
 * pointer to a CptResponse struct.
 *
 * @param res_code    Received client-side packet.
 * @return CptResponse object.
 */
```

```
CptResponse * cpt_response_init();


/**
 * Destroy CptResponse object.
 *
 * Destroys CptResponse object, freeing any allocated memory
 * and setting all pointers to null.
 *
 * @param res  Pointer to a CptResponse object.
 */
void cpt_response_destroy(CptResponse * res);


/**
 * Reset packet parameters.
 *
 * Reset the response parameters,
 * and free memory for certain params.
 *
 * @param res    A CptResponse struct.
*/
void cpt_response_reset(CptResponse * res);


/**
 * Initialize CptMsgResponse server-side sub-packet.
 *
 * @param packet    Received client-side packet.
 * @param data      Data being sent to client.
 * @return          CptResponse object.
 */
CptMsgSubPacket * cpt_msg_sp_init();


/**
 * Destroy CptMsgResponse object.
 *
 * Destroys CptMsgResponse object, freeing any allocated memory
 * and setting all pointers to null.
 *
 * @param msg_res  Pointer to a CptResponse object.
 */
void cpt_msg_sp_destroy(CptMsgSubPacket * msg_res);
```

Cpt_parse

```
/**
 * @brief Parse serialized server response.
 *
 * @param response  Address to a CptResponse object.
 * @param buffer    Serialized response from server.
 * @return Pointer to filled CptResponse.
 */
CptResponse * cpt_parse_response(uint8_t * buffer, size_t
data_size);


/**
* Create a cpt struct from a cpt packet.
*
* @param packet    A serialized cpt protocol message.
* @return A pointer to a cpt struct.
*/
CptRequest * cpt_parse_request(uint8_t * req_buf, size_t req_size);
```

Cpt_serialize

```
/**
* Serialize a CptRequest struct for transmission.
*
* @param cpt    A CptRequest struct.
* @return       Size of the serialized packet.
*/
size_t cpt_serialize_packet(CptRequest * cpt, uint8_t * buffer);


/**
* Serialize a CptResponse object for transmission.
```

```
*
* @param cpt    A CptResponse object.
* @return       Size of the serialized packet.
*/
size_t cpt_serialize_response(CptResponse * res, uint8_t * buffer);


/**
* Serialize a CptMsgResponse response sub-packet object.
*
* @param cpt    A CptResponse object.
* @return       Size of the serialized packet.
*/
size_t cpt_serialize_msg(CptMsgSubPacket * response, uint8_t *
buffer);
```

Cpt_server

```
/**
 * Handle a received 'LOGIN' protocol message.
 *
 * Use information in the CptRequest to handle
 * a LOGIN protocol message from a connected client.
 *
 * If successful, the protocol request will be fulfilled,
 * updating any necessary information contained within
 * <server_info>.
 *
 * @param server_info   Server data structures and information.
 * @param name          Name of user in received Packet MSG field.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_login_response(void * server_info, char * name);


/**
 * Handle a received 'LOGOUT' protocol message.
 *
 * Uses information in a received CptRequest to handle
 * a LOGOUT protocol message from a connected client.
 *
 * If successful, will remove any instance of the user
 * specified by the user <id> from the GlobalChannel
 * and any other relevant data structures.
 *
 * @param server_info   Server data structures and information.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_logout_response(void * server_info);


/**
 * Handle a received 'LOGOUT' protocol message.
 *
 * Uses information in a received CptRequest to handle
 * a GET_USERS protocol message from a connected client.
 *
 * If successful, the function should collect user information
 * from the channel in the CHAN_ID field of the request packet
 * in the following format.
 *      < user_id >< whitespace >< username >< newline >
 *
 * Example given:
 *      1 'Clark Kent'
 *      2 'Bruce Wayne'
 *      3 'Fakey McFakerson'
 *
 * @param server_info   Server data structures and information.
 * @param channel_id    Target channel ID.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_get_users_response(void * server_info, uint16_t channel_id);


/**
 * Handle a received 'CREATE_CHANNEL' protocol message.
 *
 * Uses information in a received CptRequest to handle
 * a CREATE_CHANNEL protocol message from a connected client.
 *
 * If a <user_list> was received in the MSG field of the packet,
 * function will also parse the <user_list> string and attempt
 * to add the requested user IDs to the channel.
 *
 * If <id_list> is NULL, function will create a new channel with
 * only the requesting user within it.
 *
 * @param server_info   Server data structures and information.
 * @param id_list       ID list from MSG field of received CPT
packet.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_create_channel_response(void * server_info, char * id_list);


/**
 * Handle a received 'JOIN_CHANNEL' protocol message.
 *
 * Uses information in a received CptRequest to handle
 * a JOIN_CHANNEL protocol message from a connected client.
 * If successful, function should add the requesting client
 * user into the channel specified by the CHANNEL_ID field
 * in the CptRequest <channel_id>.
 *
 * @param server_info   Server data structures and information.
 * @param channel_id    Target channel ID.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_join_channel_response(void * server_info, uint16_t
channel_id);


/**
 * Handle a received 'LEAVE_CHANNEL' protocol message.
 *
 * Use information in the CptRequest to handle
 * a LEAVE_CHANNEL protocol message from a connected client.
 * If successful, will remove any instance of the user
 * specified by the user <id> from the GlobalChannel
 * and any other relevant data structures.
 *
 * @param server_info   Server data structures and information.
 * @param channel_id    Target channel ID.
 * @return Status Code (SUCCESS if successful, other if failure).
 */
int cpt_leave_channel_response(void * server_info, uint16_t
channel_id);

/**
 * Generate a simple response message for client.
 *
 * Creates a serialized response packet containing
 * generic messages for operation success or failure.
 *
 * @param res   Pointer to CptResponse object.
 * @return      Pointer to Serialized CptResponse object.
 */
size_t cpt_simple_response(CptResponse * res, uint8_t * res_buf);


/**
 * Initialize and allocate all necessary server data structures.
 *
 * Initializes and allocates necessary memory for data structure
 * object pointers.
 *
 * @param gc   Pointer to the GlobalChannel object.
 * @param dir  Pointer to a LinkedList of Channel objects.
 * @return     Pointer to CptServerInfo object.
 */
CptServerInfo * cpt_server_info_init(Channel * gc, Channels * dir);


/**
 * Destroy te ServerInfo object.
 *
 * Free any allocated memory and set pointer to NULL.
 *
 * @param gc   Pointer to the GlobalChannel object.
 * @return     Pointer to CptServerInfo object.
 */
CptServerInfo * cpt_server_info_destroy(CptServerInfo *
server_info);
```

Cpt_app

```
/**
 * Initialize the GlobalChannel object.
 *
 * A Channel Object with predefined
 * properties, which are restricted
 * from regular channels.
 *
 * @return Pointer to GlobalChannel object.
 */
Channel * init_global_channel();


/**
 * Initialize channel directory object.
 *
 * Creates a ChannelDirectory object, i.e.
 * A LinkedList of Channel objects.
 *
 * @param gc Global Channel object.
 * @return ChannelDirectory object.
 */
Channels * init_channel_directory(Channel * gc);


/**
 * Initialize Channel object.
 *
 * Initialize a Channel object, allocating
 * any necessary memory and initializing data
 * members.
 *
 * @param id         The channel id.
 * @param users      A Users object (pointer to a LinkedList).
 * @return Pointer to a Channel object.
 */
Channel * channel_init(uint16_t id, Users * users);
```

```
/**
 * Destroy Channel object.
 *
 * Initialize a Channel object, allocating
 * any necessary memory and initializing data
 * members.
 *
 * @param id         The channel id.
 * @param users      A Users object (pointer to a LinkedList).
 * @param name       Name of the channel.
 * @param is_private Privacy setting of channel.
 * @return Pointer to a Channel object.
 */
void channel_destroy(Channel * channel);


/**
 * Remove a ChannelNode from a LinkedList of Channel objects.
 *
 * @param channels    Pointer to a LinkedList of Channel objects.
 * @param channel_id
 * @return
 */
int channel_delete(Channels * channels, int channel_id);


/**
 * Push a user onto a Users list.
 *
 * @param users Pointer to a linked list of User objects.
 * @param user  New user to add.
 */
int push_channel(Channels * channels, Channel * channel);


/**
 * Create a ChannelNode object.
 *
 * Creates a ChannelNode object which can be added
 * to a LinkedList. This function mostly provides
 * a wrapper for the Node object, with additional
 * semantic meaning.
 *
 * @param user A pointer to a Channel object.
 * @return     A pointer to a ChannelNode object.
 */
ChannelNode * create_channel_node(Channel * channel);


/**
 * Push a User object onto the Channel->users.
 *
 * An interface for pushing users onto a channel by
 * referencing the Channel, rather than the Channel->users.
 * Behaviour is identical to push_user().
 *
 * @param channel   Pointer to a Channel object.
 * @param user      Pointer to a User object.
 */
void push_channel_user(Channel * channel, User * user);



/**
 * Get head node from a LinkedList of Channel objects.
 *
 * Gets the de-referenced head node from a Channels
 * object - a LinkedList of type Channel, and returns
 * the ChannelNode at the head of the list.
 *
 * @param channels  Pointer to a LinkedList of Channel objects.
 * @return Pointer to the ChannelNode object at the head.
 */
ChannelNode * get_head_channel(Channels * channels);


/**
 * Initialize Channels object.
 *
 * A wrapper for the LinkedList object
 * that mainly provides semantics.
 *
 * @param channel A pointer to a Channel object.
 * @return a Channels object (Pointer to a LinkedList).
 */
Channels * channels_init(ChannelNode * channel_node);



/**
 * Destroy Channels object.
 *
 * Destroy Channels object, freeing any dynamically
 * allocated memory.
 *
 * @param channels  A LinkedList of Channel objects.
 */
void channels_destroy(Channels * channels);


/**
 * Find a channel in a list of channels.
 *
 * Searches for a specific channel based on the
 * Comparator function pointer
 *
 * @param channel A pointer to a Channel object.
 * @return a Channels object (Pointer to a LinkedList).
 */
Channel * find_channel(Channels * dir, uint16_t id);


/**
 * Print details about the Channel object.
 *
 * @param channel A Channel object.
 */
char * channel_to_string(Channel * channel);


/**
```

```
 * A helper for cpt_leave_channel.
 *
 * comparator function pointer that compares
 * channel IDs in a LinkedList object.
 *
 * @param channel_id      A channel ID.
 * @param target_channel  Another channel ID.
 * @return
 */
bool compare_channels(const uint16_t * channel_id, const uint16_t *
target_channel);


/**
 * Parse requested user IDs from body of
 * CREATE_CHANNEL cpt protocol message.
 *
 * @param id_list  MSG field of received CREATE_CHANNEL request.
 * @return Number of requested IDs in the id_buf.
 */
Users * filter_channel_users(Channel * channel, uint16_t * id_buf,
char * id_list);


/**
 *
 * @param id_list
 * @param id_buf
 * @return
 */
uint16_t parse_channel_query(char * id_list, uint16_t * id_buf);
```

Cpt_chat_server

```
/**
 * Drive the program.
 */
void run();
```

```
/**
 * Event handler for new connection.
 *
 * @return
 */
int handle_new_accept();


/**
 * Returns true if the incoming revent is POLLIN.
 *
 * @return
 */
bool is_revent_POLLIN(int index);


/**
 *  Initializes the socket for global channel gc.
 *
 * @param gc Channel object for global channel
 */
void listen_socket_init(Channel * gc);


/**
 *
 * @param poll_result
 * @return
 */
bool should_end_event_loop(int poll_result);


/**
 *  Event handler for login command.
 *
 * @param info
 * @return
 */
int login_event(CptServerInfo * info);
```

```
/**
 * Event handler for logout command.
 *
 * @param info
 */
void logout_event(CptServerInfo * info);


/**
 * Event handler for get-users command.
 *
 * @param info
 * @param chan_id
 */
void get_users_event(CptServerInfo * info, int chan_id);


/**
 * Event handler for send command.
 *
 * @param info
 * @param msg
 */
void send_message_event(CptServerInfo * info, char * msg);


/**
 *  Event handler for leave-channel command.
 *
 * @param info
 * @param id
 */
void leave_channel_event(CptServerInfo * info, uint16_t id);


/**
 * Event handler for create-channel command
 *
 * @param info
 * @param id_list
 */
```

```
void create_channel_event(CptServerInfo * info, char * id_list);


/**
 * Event handler for join-channel command
 *
 * @param info
 * @param channel_id
 */
void join_channel_event(CptServerInfo *info, uint16_t channel_id);


/**
 * Event handler for send command.
 *
 * @param info
 * @param msg
 * @param channel_id
 */
void send_event(CptServerInfo * info, char * msg, int channel_id);
```

Interface

```
/**
 * Check user input against valid CLI commands.
 *
 * @param cmd     string from get_user_input().
 * @return true or false.
 */
bool is_valid_cmd(Command * cmd);


/**
 * Initialize command object.
 *
 * Allocates necessary memory and initialize all
 * pointers to NULL.
 *
 * @return Pointer to command object.
 */
Command * cmd_init();
```

```
/**
 * Destroy Command object.
 *
 * Free any dynamically allocated memory
 * and set all pointer to NULL.
 *
 * @param cmd   Pointer to Command object.
 */
void cmd_destroy(Command * cmd);


/**
 * Execute user entered commands.
 *
 * @param user_commands Commands from user input.
 */
void handle_command(Command * cmd);

/**
 * Parse user input from stdin.
 *
 * Calls parse_cmd_args() and parse_cmd().
 *
 * @param cmd   Pointer to Command object.
 */
void parse_cmd_input(Command * cmd);


/**
 * Parse command from user input.
 *
 * @param cmd   Pointer to Command object.
 */
void parse_cmd(Command * cmd);


/**
 * Parse arguments from user input.
 *
 * @param cmd   Pointer to Command object.
 */
void parse_cmd_args(Command * cmd);


/**
 * Handle parsed commands.
 *
 * @param command   Pointer to Command object.
 */
void * handle_cmd_thread(void * cmd);


/**
 * Check if user input is a command.
 *
 * @param command
 * @param cli_cmd
 * @return
 */
bool is_cmd(Command * command, char * cli_cmd);
```

TCP_networking

Client

```
/**
 * @brief Get client/server address information for network data
transfer.
 *
 * A wrapper for posix setsockopt() system call
 * that includes some minor error handling.
 *
 * @param host  IP or url of target destination.
 * @param port  Port of target destination.
 * @return      Pointer to addrinfo struct or NULL if failure.
 */
struct addrinfo * tcp_client_addr(const char * host, const char *
port);
```

```
/**
 * @brief Create a socket to write to server.
 *
 * A wrapper for posix socket() system call
 * that includes some minor error handling.
 *
 * @param serv_info Result returned from setup_client_addr().
 * @return          Socket file descriptor.
 */
int tcp_client_socket(struct addrinfo * serv_info);


/**
 * @brief Set custom socket options.
 *
 * A wrapper for posix setsockopt() system call
 * that includes some minor error handling.
 *
 * @param sock_fd   Socket file descriptor.
 * @param sock_opt  Socket option to set (see Posix Manual for
details.)
 * @return          0 if success, -1 if failure.
 */
int tcp_client_sock_opt(int sock_fd, int sock_opt);


/**
 * @brief Establish TCP connection to server.
 *
 * A wrapper for posix connect() system call
 * that includes some minor error handling.
 *
 * @param sock_fd   Socket file descriptor.
 * @param serv_info Return value from tcp_init_addr().
 * @return          0 if successful, -1 if failure.
 */
int tcp_client_connect(int sock_fd, struct addrinfo * serv_info);
```

```
/**
 * @brief Send a message over TCP connection.
 *
 * @param sock_fd   Socket file descriptor.
 * @param msg       Data to send.
 */
int tcp_client_send(int sock_fd, uint8_t * data, size_t data_size);


/**
 * @brief Initialize a TCP connection.
 *
 * Makes all necessary TCP posix system calls and
 * returns a file descriptor for writing to the
 * connected socket.
 *
 * @param host  IP or URL of target destination.
 * @param port  Port of target destination.
 * @return      0 if successful, -1 if failure.
 */
int tcp_init_client(const char * host, const char * port);


/**
 * @brief Wait until a certain point in time.
 *
 * @param time_point start time.
 */
void countdown(long time_point);


/**
 * @brief Get a timepoint in seconds.
 *
 * @param client_time   An absolute point in time.
 * @return              Time in seconds.
 */
long time_in_sec(char * client_time);


/**
```

```
 * @brief Receive data from tcp client.
 *
 * @param sock_fd   Socket file descriptor.
 * @return          Pointer to received data.
 */
ssize_t tcp_client_recv(int sock_fd, uint8_t * buff);
```

Server

```
/**
 * @brief Get client/server address information for network data
transfer.\n\n
 *
 * @param host  IP or url of target destination.
 * @param port  Port of target destination.
 * @return      Pointer to addrinfo struct or NULL if failure.
 */
struct addrinfo * tcp_server_addr(const char * ip, const char *
port);


/**
 * @brief Create a socket to write to server.\n\n
 *
 * @param serv_info Result returned from setup_client_addr().
 * @return          Socket file descriptor.
 */
int tcp_listen_socket(struct addrinfo * serv_info);


/**
 * Set custom socket options.\n\n
 *
 * @param sock_fd   Socket file descriptor.
 * @param serv_info Return value from tcp_server_addr().
 * @param sock_opt  Socket option to set (see Posix Manual for
details.)
 * @return          0 if success, -1 if failure.
 */
int tcp_server_sock_opt(int sock_fd, int sock_opt);
```

```
/**
 * @brief Bind server address to listener socket.\n\n
 *
 * @param listen_fd Listener socket file descriptor.
 * @param serv_info Result returned from setup_client_addr().
 * @return          Socket file descriptor.
 */
int tcp_server_bind(struct addrinfo * serv_info, int listen_fd);


/**
 * @brief Bind server address to listener socket.\n\n
 *
 * @param listen_fd Listener socket file descriptor.
 * @param serv_info Result returned from setup_client_addr().
 * @return          Socket file descriptor.
 */
int tcp_server_listen(int listen_fd);


/**
 * @brief Accept incoming tcp connection.\n\n
 *
 * @param listen_fd Listener socket file descriptor.
 * @param serv_info Result returned from setup_client_addr().
 * @return          Socket file descriptor.
 */
int tcp_server_accept(struct sockaddr_storage * client_addr, int
listen_fd);


/**
 * @brief Receive data from tcp client.\n\n
 *
 * @param sock_fd   Socket file descriptor.
 * @return          Pointer to received data.
 */
ssize_t tcp_server_recv(int sock_fd, uint8_t * req_buf);
```

```
/**
 * @brief Drive the TCP init functions.\n\n
 *
 * Calls all tcp server init functions and returns file
 * descriptor for te listener socket.
 *
 * @param host  IP or url of target destination.
 * @param port  Port of target destination.
 * @return      Pointer to received.
 */
int tcp_server_init(char * host, const char *port);


/**
 * @brief Send data to connected client.\n\n
 *
 * Sends data a TCP connected client.
 *
 * @param sock_fd   Client socket file descriptor.
 * @param data      Data being sent.
 * @return          0 if success, -1 if failure.
 */
int tcp_server_send(int sock_fd, uint8_t * data, size_t data_size);
```

UDP_networking

Client

```
/**
* @brief Get client/server address information for network data
transfer.
*
* @param port  Port of target destination.
* @return      Pointer to addrinfo struct or NULL if failure.
*/
struct addrinfo * udp_client_addr(const char * host, const char *
port);
```

```
/**
* @brief Create a socket to write to server.
*
* @param client_addr Result returned from setup_client_addr().
* @return          Socket file descriptor.
*/
int udp_client_socket(struct addrinfo * client_addr);


/**
* @brief Send bytes to server.
*
* @param client_addr Result returned from setup_client_addr().
* @param sock      Socket file descriptor.
* @param msg       Message to be sent.
*/
void udp_client_sendto(struct addrinfo * client_addr, int sock, char *
msg);


/**
*
*
* @param sock_fd
* @param data
* @param data_size
* @return
*/
int udp_client_send(int sock_fd, uint8_t * data, size_t data_size);


/**
*
* @param sock_fd
* @param buf
* @return
*/
ssize_t udp_client_recv(int sock_fd, uint8_t * buf);


/**
* Create a UDP connection.
```

```
*
* @param udp_fd
* @param client_addr
* @return
*/
int udp_client_connect(int udp_fd, struct addrinfo * client_addr);
```

Server

```
/**
* @brief Create a server socket for listening.
*
* @param serv_addr Server address info.
* @return          Socket file descriptor.
*/
int udp_server_socket(struct addrinfo * serv_addr);


/**
* @brief Bind to server socket to port.
*
* @param serv_addr Server address info.
* @param sock      Server size socket file descriptor.
* @return          0 if successful, -1 if failed.
*/
int udp_server_bind(int sock, struct addrinfo * serv_addr);


/**
* Wrapper function for sending a udp packet.
*
* @param sock_fd       connected udp fd
* @param data          data
* @param data_size     size of the data
* @return
*/
int udp_server_send(int sock_fd, uint8_t * data, size_t data_size);



/**
```

```
* Wrapper function for connecting udp client_addr to the udp_fd
*
* @param udp_fd        udp file descriptor to connect.
* @param client_addr   a pointer to addrinfo to connect to the udp file
descriptor.
* @return              udp file descriptor.
*/
int udp_server_connect(int udp_fd, struct addrinfo * client_addr);


/**
* Creating the socket fd for the receiving of the udp packet.
*
* @param host      host for receiving udp packets on server-side.
* @param port      port for receiving udp packets on server-side.
* @return          udp file descriptor
*/
int udp_server_sock_r(const char * host, const char * port);


/**
* Creating the socket fd for the sending of the udp packet.
*
* @param host      host for sending udp packets on server-side.
* @param port      port for sending udp packets on server-side.
* @return          udp file descriptor
*/
int udp_server_sock_s(const char * host, const char * port);
```