ChatProgramm_Final

Generated by Doxygen 1.14.0

Chapter 1

ChatProgramm_Final

A decentralized, peer-to-peer chat application developed in Python for the **Betriebssysteme und Rechnernetze** (Operating Systems and Computer Networks) course. This project implements a custom protocol called SLCP (Simple Local Chat Protocol) to demonstrate real-time communication over local networks without relying on central servers.

1.1 Project Description

The project aims to simulate a real-world decentralized chat application that supports direct peer-to-peer communication without relying on a central server. By using both UDP and TCP sockets, the system enables:

- Dynamic discovery of other peers
- Text-based chat communication
- Image file sharing
- Away-from-keyboard (AFK) autoreply functionality
- Two fully-featured interfaces: a command-line interface (CLI) and a graphical user interface (GUI)

This system mirrors the principles of modern decentralized applications where reliability, independence, and peer autonomy are key. The communication protocol (SLCP) was developed specifically for this project, ensuring messages are structured, lightweight, and interpretable.

1.1.1 Challenges Faced

Developing a decentralized peer-to-peer application came with several challenges:

- **Concurrency Management:** Synchronizing multiple processes (network, discovery, GUI, CLI) required careful use of multiprocessing, threading, and pipes.
- **Peer Discovery:** Broadcasting JOIN and WHO messages while avoiding duplicate peers or stale data demanded a reliable protocol logic.
- **Image Transfer:** Transferring large binary files over TCP while signaling over UDP required a clean and fail-safe handshaking mechanism.
- **Interface Parity:** Maintaining full feature parity between CLI and GUI was non-trivial, especially with user interactions.
- **AFK Logic:** Automatically replying with a custom message while avoiding spam loops required careful state tracking.
- **Cross-platform Compatibility:** Ensuring the system runs reliably on Windows, macOS, and Linux involved path handling, port availability checks, and encoding robustness.

1.2 Features

- **Peer Discovery:** Broadcast-based peer discovery using `JOIN`, `WHO`, and `KNOWUSERS` messages.
- **Message Exchange:** Real-time message delivery over UDP.
- **Image Transfer:** TCP-based file transfer with UDP notification handshakes.
- **AFK Mode:** Automatic autoreplies when a user is away.
- **Graphical Interface:** Built using PyQt5 with dark/light theme support.
- **Settings Dialog:** Runtime configuration for user handle, port, autoreply message, and image folder.
- **CLI Interface:** Text-based command-line chat interface with full feature parity.
- **Fully Documented:** Comprehensive technical documentation generated using Doxygen.

1.3 Architecture

1.3.1 GUI-Based Execution Flow

1.3.2 CLI-Based Execution Flow

1.3.3 Core Modules

Module	Description
`main.py`	Application entry point with GUI interface
`cli.py`	Alternative CLI-based interface
`discovery.py`	Broadcast-based peer discovery logic
`network.py`	Handles UDP messaging, AFK logic, TCP images
`gui.py`	PyQt5-based user interface logic
`config.toml`	TOML configuration for clients and settings

1.4 Team and Responsibilities

Name	Matrikelnummer	Responsibilities
Aashir Ahtisham	1447390	Main contributor to `discovery.py`, contributed to `network.py`
Bratli Metuka	1505429	Main contributor to `network.py`, helped with `gui.py`
Jalal Eddin Alhaj Ahmad	1428348	Main contributor to `cli.py` and `gui.py`, also worked on `discovery.py`
Joseph Bolaños Beyloune	1534591	Main contributor to documentation, contributed to `cli.py`, `gui.py`, `network.py`
Ömer Faruk Capraz	1522507	Main contributor to documentation, helped with `discovery.py` and `cli.py`

1.5 Installation & Setup

1.5.1 1. Install Python 3.10+

Check your version:

```
python3 --version
```

If it's lower than 3.10, update it via your platform's instructions.

1.5.2 2. Install Dependencies

Use `pip` to install the required packages:

```
pip install PyQt5 toml qdarkstyle
```

1.5.3 Required Packages

Package	Purpose
PyQt5	GUI framework (required for `main.py`)
toml	Configuration file parser
qdarkstyle	Optional dark mode for GUI theme

1.5.4 3. Configure Clients

Update the `config.toml` file to define your client settings:

```
[[clients]]
handle = "Aashir"
port = [5008, 6000]
whoisport = 4000
autoreply = "Back in one hour"
away = false
imagepath = "./images/aashir"
```

Each client must have a unique `handle`, and ports must not conflict.
The clients can also be updated after the application was launched.

1.5.5 4. Launch the Application

To run with the GUI:

```
python3 main.py Aashir
```

To run via CLI:

```
python3 cli.py Aashir
```

Replace `"Aashir"` with any configured handle in `config.toml`.

1.6 Platform-Specific Instructions

1.6.1 Windows

- Use **PowerShell** or **CMD**.
- Ensure Python is added to PATH.
- Use `python` instead of `python3` if needed.

```
python main.py Aashir
```

1.6.2 macOS

- Open **Terminal**.
- Use the default Python 3 installation (or via Homebrew).

```
python3 main.py Aashir
```

1.6.3 Linux (Ubuntu/Debian)

- Open **Terminal**.
- Make sure Python 3 and pip are installed:

```
sudo apt update
sudo apt install python3 python3-pip
```

- Install dependencies:

```
pip3 install PyQt5 toml qdarkstyle
```

- Run the application:

```
python3 main.py Aashir
```

1.7 GUI Controls

- **Send Message**: Press Enter or click "Send"
- **Send Image**: Select an image via "Send Image" button
- **Clients**: Show connected peers
- **AFK Toggle**: Enable/disable AFK autoreply
- **Settings**: Edit configuration interactively
- **Leave Chat**: Graceful exit

1.8 Testing

You can simulate multiple clients by:

- Opening multiple terminal sessions with different handles
- Running on separate machines in the same LAN
- Observing image transfers and peer join/leave messages

1.9 Documentation

Doxygen documentation is located at:
`docs/html/index.html`
To regenerate:

```
doxygen Doxyfile
```

The documentation includes:

- Function and class reference
- Namespace and file structure
- Inline code documentation

1.10 Technologies

- **Python 3.10+**
  The main programming language used for the entire application. Python's high-level syntax and standard library make it ideal for rapid development and academic projects.
- **PyQt5**
  Used to create the graphical user interface (GUI). Offers support for event-driven programming through signals and slots, as well as cross-platform compatibility.
- **TOML**
  A minimal and human-readable configuration file format used to define client settings such as handles, ports, autoreplies, and image directories.
- **Socket Programming (UDP/TCP)**
  Enables real-time communication between peers. UDP is used for message broadcasting (`JOIN`, `WHO`, `MSG`, etc.), while TCP is used for transferring binary image files.

- **Doxygen**
  Automatically generates HTML-based technical documentation from docstrings and markdown files such as `README.md`.
- **Multiprocessing Pipes (`multiprocessing.Pipe`)**
  Used for inter-process communication (IPC) between the GUI/CLI and the network process. Allows sending structured messages like `(MSG, handle, message)` through unidirectional or bidirectional channels.
- **Threads (`threading.Thread`)**
  Used for non-blocking background tasks like periodic broadcasting (`JOIN`, `WHO`) and image transfers. Ensures responsiveness of the GUI and CLI.
- **Shared Memory (via `multiprocessing` state)**
  Certain configuration states (e.g., AFK status) and message buffers are indirectly synchronized across processes by sharing references during process creation.
- **QDarkStyle**
  A ready-made dark mode theme applied to the PyQt5 interface for improved aesthetics and readability.

1.11 Security Notice

This application is designed for academic purposes only.
It does **not** implement encryption, authentication, or secure transport mechanisms.
Use only on trusted local networks.

1.12 License

MIT License – For educational use only.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 cli Namespace Reference

Functions

- ts ()

  Returns a formatted timestamp string.
- port_in_use (port)

  Checks if a UDP port is currently in use.
- print_commands ()

  Prints a list of available CLI commands.
- main ()

  Main function that initializes the CLI chat client.

Variables

- str CONFIG_FILE = "config.toml"
- str COLOR_RESET = "\033[0m"
- str COLOR_GREEN = "\033[92m"
- str COLOR_RED = "\033[91m"
- str COLOR_YELLOW = "\033[93m"

6.1.1 Function Documentation

6.1.1.1 main()

```
cli.main ()
```

Main function that initializes the CLI chat client.
Loads configuration, starts discovery and network processes, and runs an input loop.
Definition at line 73 of file cli.py.

```
00073 def main():
00074     cfg_all = toml.load(CONFIG_FILE)
00075     clients = cfg_all.get("clients", [])
00076     if not clients:
00077         print("No [[clients]] section found in config.toml.")
00078         sys.exit(1)
00079 
00080     if len(sys.argv) != 2:
00081         handles = [c["handle"] for c in clients]
00082         print("Usage: python cli.py <Handle>")
00083         print("Available handles:", ", ".join(handles))
00084         sys.exit(1)
00085 
00086     chosen = sys.argv[1]
00087     client_index = next((i for i, c in enumerate(clients) if c["handle"] == chosen), None)
00088     if client_index is None:
00089         print(f"Handle '{chosen}' not found.")
00090         sys.exit(1)
00091 
```

```
00092     # Prepare client configuration and shared state
00093     config = clients[client_index]
00094     manager = multiprocessing.Manager()
00095     config["peers"] = manager.list()
00096     config["__cfg_all"] = cfg_all
00097     config["__cfg_index"] = clients.index(config)
00098
00099     # Inter-process communication pipes
00100     ui2net_p, ui2net_c = multiprocessing.Pipe()
00101     net2ui_p, net2ui_c = multiprocessing.Pipe()
00102     disc_ctrl_parent, disc_ctrl_child = multiprocessing.Pipe()
00103
00104     # Start discovery process if not already running
00105     p_disc = None
00106     if not port_in_use(config["whoisport"]):
00107         p_disc = multiprocessing.Process(target=discovery_process, args=(config, disc_ctrl_child))
00108         p_disc.start()
00109         print(f"[INFO] Discovery service started on port {config['whoisport']}")
00110     else:
00111         print(f"[INFO] Discovery already running on port {config['whoisport']}")
00112
00113     # Start network process
00114     p_net = multiprocessing.Process(target=network_process, args=(config, ui2net_c, net2ui_p))
00115     p_net.start()
00116
00117     stop_event = threading.Event()
00118     left_peers = set()
00119
00120
00122     def poll_network():
00123         while not stop_event.is_set():
00124             while net2ui_c.poll():
00125                 typ, src, payload = net2ui_c.recv()
00126                 if typ == 'MSG':
00127                     print(f"\n{COLOR_GREEN}{ts()} [{src}] {payload}{COLOR_RESET}\n")
00128                 elif typ == 'IMG':
00129                     print(f"\n{COLOR_YELLOW}{ts()} [{src}] sent image → {payload}{COLOR_RESET}\n")
00130                 elif typ == 'LEAVE':
00131                     if src not in left_peers:
00132                         print(f"\n{COLOR_RED}{ts()} [{src}] left the chat.{COLOR_RESET}\n")
00133                         left_peers.add(src)
00134                     config['peers'][:] = [p for p in config['peers'] if p[0] != src]
00135             time.sleep(0.05)
00136
00137     threading.Thread(target=poll_network, daemon=True).start()
00138
00139     print(f"\n========== SLCP CLI Chat started as '{chosen}' ==========")
00140     print_commands()
00141
00142     try:
00143         while True:
00144             cmd = input("> ").strip()
00145             if not cmd:
00146                 continue
00147
00148             parts = cmd.split(" ", 2)
00149             action = parts[0].lower()
00150
00151             if action == "leave":
00152                 print("Sending LEAVE...")
00153                 ui2net_p.send(("LEAVE", "", ""))
00154
00155                 if p_disc:
00156                     disc_ctrl_parent.send("STOP")
00157                     p_disc.join()
00158                     print("[INFO] Discovery stopped.")
00159
00160                 ui2net_p.send(("EXIT", "", ""))
00161                 p_net.join()
00162                 stop_event.set()
00163                 time.sleep(0.1)
00164                 break
00165
00166             elif action == "clients":
00167                 peers = [(h, ip, pt) for (h, ip, pt) in config['peers'] if h != chosen]
00168                 if not peers:
00169                     print("No other clients found.")
00170                 else:
00171                     print("\nActive clients:")
00172                     for (h, ip, pt) in peers:
00173                         print(f"  {h} ({ip}:{pt})")
00174                     print()
00175
00176             elif action == "msg" and len(parts) >= 3:
00177                 dest = parts[1]
00178                 msg = parts[2]
00179                 ui2net_p.send(("MSG", dest, msg))
```

```
00180                 print(f"[SEND] to {dest}: {msg}")
00181
00182             elif action == "img" and len(parts) >= 3:
00183                 dest = parts[1]
00184                 path = parts[2]
00185                 if not os.path.isfile(path):
00186                     print(f"[ERROR] File not found: {path}")
00187                     continue
00188                 ui2net_p.send(("IMG", dest, path))
00189                 print(f"[SEND IMG] to {dest}: {path}")
00190
00191             elif action == "afk" and len(parts) == 2:
00192                 mode = parts[1].lower()
00193                 if mode in ("on", "off"):
00194                     ui2net_p.send(("AFK", chosen, mode.upper()))
00195                     print(f"[AFK] set to {mode.upper()}")
00196                 else:
00197                     print("[ERROR] Usage: afk on|off")
00198
00199             elif action == "help":
00200                 print_commands()
00201
00202             else:
00203                 print("[ERROR] Unknown command. Type 'help' for commands.")
00204
00205             time.sleep(0.05)
00206
00207     finally:
00208         stop_event.set()
00209         time.sleep(0.1)
00210
00211         if p_disc:
00212             disc_ctrl_parent.send("STOP")
00213             p_disc.join()
00214
00215         ui2net_p.send(("EXIT", "", ""))
00216         p_net.join()
00217
00218
```

6.1.1.2 port_in_use()

```
cli.port_in_use (
              port)
```

Checks if a UDP port is currently in use.

Parameters

port	UDP port to test.

Returns

True if in use, False otherwise.

Definition at line 52 of file cli.py.

```
00052 def port_in_use(port):
00053     with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
00054         try:
00055             s.bind(("", port))
00056             return False
00057         except OSError:
00058             return True
00059
```

6.1.1.3 print_commands()

```
cli.print_commands ()
```

Prints a list of available CLI commands.

Definition at line 62 of file cli.py.

```
00062 def print_commands():
00063     print("\nAvailable commands:")
00064     print("  msg <handle> <text>")
00065     print("  img <handle> <path_to_image>")
00066     print("  clients")
00067     print("  afk on|off")
00068     print("  leave\n")
00069
```

6.1.1.4 ts()

```
cli.ts ()
```

Returns a formatted timestamp string.
Definition at line 45 of file cli.py.

```
00045 def ts():
00046     return datetime.now().strftime("[%H:%M:%S]")
00047
```

6.1.2 Variable Documentation

6.1.2.1 COLOR_GREEN

```
str cli.COLOR_GREEN = "\033[92m"
```

Definition at line 39 of file cli.py.

6.1.2.2 COLOR_RED

```
str cli.COLOR_RED = "\033[91m"
```

Definition at line 40 of file cli.py.

6.1.2.3 COLOR_RESET

```
str cli.COLOR_RESET = "\033[0m"
```

Definition at line 38 of file cli.py.

6.1.2.4 COLOR_YELLOW

```
str cli.COLOR_YELLOW = "\033[93m"
```

Definition at line 41 of file cli.py.

6.1.2.5 CONFIG_FILE

```
str cli.CONFIG_FILE = "config.toml"
```

Definition at line 35 of file cli.py.

6.2 discovery Namespace Reference

Functions

- get_local_ip ()

  Get the local machine's IP address.
- discovery_process (config, ctrl_pipe)

  Main discovery process function.

6.2.1 Function Documentation

6.2.1.1 discovery_process()

```
discovery.discovery_process (
              config,
              ctrl_pipe)
```

Main discovery process function.
This function is designed to run in its own multiprocessing process. It listens and sends UDP broadcasts to discover peers on the network. It maintains a list of active peers by interpreting SLCP commands like:

- `JOIN` (a peer has joined),
- `LEAVE` (a peer has left),

- `WHO` (a peer is asking who is online),
- `KNOWUSERS` (a reply containing known peers).

One client that successfully binds to the discovery port acts as the WHO responder. The process listens to a control pipe for termination.

Parameters

config	A shared dictionary (Manager.dict) containing client configuration and a shared peer list. Required fields: `handle`, `port`, `whoisport`, `peers`.
ctrl_pipe	A multiprocessing pipe used by the main process to send control signals (e.g., "STOP").

Definition at line 51 of file discovery.py.

```
00051 def discovery_process(config, ctrl_pipe):
00052     handle    = config["handle"]
00053     port      = config["port"][0]
00054     whoisport = config["whoisport"]
00055     peers     = config["peers"]
00056 
00057     responder = False  # Only one client becomes WHO responder
00058 
00059     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00060     sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
00061     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
00062 
00063     try:
00064         sock.bind(("", whoisport))
00065         responder = True
00066         print(f"[Discovery] WHO responder active on {whoisport}")
00067     except OSError as e:
00068         print(f"[Discovery] Not WHO responder - {e}")
00069 
00070     sock.settimeout(1.0)
00071 
00072 
00075     def broadcast(msg: str):
00076         sock.sendto(msg.encode("utf-8"), ("255.255.255.255", whoisport))
00077 
00078     while True:
00079         # Handle stop command from main process
00080         if ctrl_pipe.poll():
00081             cmd = ctrl_pipe.recv()
00082             if cmd == "STOP":
00083                 print("[Discovery] Terminated by main process.")
00084                 break
00085 
00086         # Broadcast JOIN and WHO messages
00087         broadcast(f"JOIN {handle} {port}\n")
00088         broadcast("WHO\n")
00089 
00090         start = time.time()
00091         while time.time() - start < 1.0:
00092             try:
00093                 data, addr = sock.recvfrom(4096)
00094             except socket.timeout:
00095                 break
00096 
00097             try:
00098                 text = data.decode("utf-8").strip()
00099             except UnicodeDecodeError:
00100                 continue
00101 
00102             if not text:
00103                 continue
00104 
00105             parts = text.split()
00106             cmd = parts[0]
00107 
00108             # Handle JOIN message: add new peer
00109             if cmd == "JOIN" and len(parts) == 3:
00110                 peer, pport = parts[1], int(parts[2])
00111                 entry = (peer, addr[0], pport)
00112                 if peer != handle and entry not in peers:
00113                     peers.append(entry)
00114                     print(f"[Discovery] New peer detected: {entry}")
00115 
00116             # Handle LEAVE message: remove peer
00117             elif cmd == "LEAVE" and len(parts) == 2:
00118                 peer = parts[1]
00119                 peers[:] = [p for p in peers if p[0] != peer]
```

```
00120
00121             # Handle WHO request: respond with known users
00122             elif cmd == "WHO" and responder:
00123                 all_known = [(handle, get_local_ip(), port)] + list(peers)
00124                 payload = ",".join(f"{h} {ip} {pt}" for h, ip, pt in all_known)
00125                 sock.sendto(f"KNOWUSERS {payload}".encode("utf-8"), addr)
00126
00127             # Handle KNOWUSERS response: merge peer list
00128             elif cmd == "KNOWUSERS":
00129                 rest = text[len("KNOWUSERS "):]
00130                 for chunk in rest.split(','):
00131                     if not chunk.strip():
00132                         continue
00133                     try:
00134                         h, ip, pt = chunk.strip().split()
00135                         entry = (h, ip, int(pt))
00136                         if h != handle and entry not in peers:
00137                             peers.append(entry)
00138                     except ValueError:
00139                         continue
00140
00141         time.sleep(3)
```

6.2.1.2 get_local_ip()

`discovery.get_local_ip ()`

Get the local machine's IP address.

This utility function connects to a known external IP (Google DNS) to determine the outbound interface. If no connection is possible, falls back to loopback.

Returns

A string representing the local IP address.

Definition at line 25 of file discovery.py.

```
00025 def get_local_ip():
00026     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00027     try:
00028         s.connect(("8.8.8.8", 80))
00029         return s.getsockname()[0]
00030     except Exception:
00031         return "127.0.0.1"
00032     finally:
00033         s.close()
00034
```

6.3 gui Namespace Reference

Classes

- class SettingsDialog

  Dialog window for editing and saving user configuration.

Functions

- ts ()

  Generate a timestamp string for message labeling.
- open_file (path)

  Open a file with the default system application.
- get_local_ip ()

  Get the current machine's local IP address.
- gui_process (config, to_network, from_network)

  Launches the SLCP GUI as a separate process.

Variables

- int MAX_DISPLAY_CHUNK = 200
- str CONFIG_FILE = "config.toml"

6.3.1 Function Documentation

6.3.1.1 get_local_ip()

```
gui.get_local_ip ()
```

Get the current machine's local IP address.
Attempts a connection to a public DNS server to infer outbound interface IP. Fallback is 127.0.0.1 if no internet connection exists.

Returns

Local IP address as string.

Definition at line 68 of file gui.py.

```
00068 def get_local_ip():
00069     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00070     try:
00071         s.connect(("8.8.8.8", 80))
00072         return s.getsockname()[0]
00073     except Exception:
00074         return "127.0.0.1"
00075     finally:
00076         s.close()
00077
```

6.3.1.2 gui_process()

```
gui.gui_process (
              config,
              to_network,
              from_network)
```

Launches the SLCP GUI as a separate process.
Connects user inputs and received messages through inter-process communication (IPC) with the network process. Implements all chat interface elements (send, AFK toggle, image sending, client listing, etc.). Polls for incoming network data and handles state updates accordingly.

Parameters

config	A dictionary containing client state and runtime configuration.
to_network	A pipe object to send data to the network process.
from_network	A pipe object to receive data from the network process.

Definition at line 151 of file gui.py.

```
00151 def gui_process(config, to_network, from_network):
00152     handle = config['handle']
00153     img_path = config['imagepath']
00154     os.makedirs(img_path, exist_ok=True)
00155
00156     app = QApplication(sys.argv)
00157     wnd = QWidget()
00158     wnd.setWindowTitle(f"SLCP Chat - {handle}")
00159
00160     # Main layout
00161     vlayout = QVBoxLayout()
00162     chat = QTextEdit(); chat.setReadOnly(True)
00163     vlayout.addWidget(chat)
00164
00165     # Control layout (buttons + inputs)
00166     controls = QHBoxLayout()
00167     dest_input = QLineEdit(); dest_input.setPlaceholderText("Recipient handle")
00168     msg_input = QLineEdit(); msg_input.setPlaceholderText("Message...")
00169     btn_send = QPushButton("Send")
00170     btn_img = QPushButton("Send Image")
00171     btn_clients = QPushButton("Clients")
00172     btn_leave = QPushButton("Leave Chat")
00173     btn_afk = QPushButton("AFK: OFF"); btn_afk.setCheckable(True)
00174     btn_afk.setStyleSheet("background-color: #666; color: white;")
00175     btn_dark = QPushButton("Dark Mode"); btn_dark.setCheckable(True)
00176     btn_settings = QPushButton("Settings")
00177
00178     for w in (dest_input, msg_input, btn_send, btn_img, btn_clients, btn_settings, btn_leave, btn_afk,
      btn_dark):
00179         controls.addWidget(w)
```

```
00180     vlayout.addLayout(controls)
00181     wnd.setLayout(vlayout)
00182
00183     local_peers = set()
00184     afk_mode = False
00185
00186
00190     def append(text, color="#010202"):
00191         chat.setTextColor(QColor(color))
00192         chat.append(f"{ts()} {text}")
00193         chat.moveCursor(QTextCursor.End)
00194
00195
00197     def send_message():
00198         dest = dest_input.text().strip()
00199         msg = msg_input.text().strip()
00200         if not dest or not msg:
00201             return
00202         append(f"{handle}: {msg}", "#2A8940")
00203         to_network.send(("MSG", dest, msg))
00204         msg_input.clear()
00205
00206
00208     def send_image():
00209         path, _ = QFileDialog.getOpenFileName(wnd, "Select image", "", "Images (*.png *.jpg *.bmp
    *.webp)")
00210         if not path:
00211             return
00212         dest = dest_input.text().strip()
00213         if not dest:
00214             QMessageBox.warning(wnd, "Error", "Please enter recipient handle!")
00215             return
00216         append(f"{handle} → {dest} [Image]", "#2A8940")
00217         to_network.send(("IMG", dest, path))
00218
00219
00221     def show_clients():
00222         peers = [(h, ip, pt) for (h, ip, pt) in config['peers'] if h != handle]
00223         if not peers:
00224             QMessageBox.information(wnd, "Clients", "No other clients found.")
00225         else:
00226             local_ip = get_local_ip()
00227             local_port = config["port"][0]
00228             info = "\n".join(f"{h} ({ip}:{pt})" for (h, ip, pt) in peers)
00229             QMessageBox.information(wnd, "Clients", f"You: {handle}
    ({local_ip}:{local_port})\n\nActive clients:\n{info}")
00230
00231     already_closing = False
00232
00233
00235     def leave_chat():
00236         nonlocal already_closing
00237         if already_closing:
00238             return
00239         already_closing = True
00240         to_network.send(("LEAVE", handle, ""))
00241         to_network.send(("EXIT", "", ""))
00242         wnd.close()
00243
00244
00246     def toggle_afk():
00247         nonlocal afk_mode
00248         if btn_afk.isChecked():
00249             btn_afk.setText("AFK: ON")
00250             btn_afk.setStyleSheet("background-color: #cc5500; color: white;")
00251             afk_mode = True
00252             to_network.send(("AFK", handle, "ON"))
00253             append("[System] AFK mode enabled", "#c22809")
00254         else:
00255             btn_afk.setText("AFK: OFF")
00256             btn_afk.setStyleSheet("background-color: #666; color: white;")
00257             afk_mode = False
00258             to_network.send(("AFK", handle, "OFF"))
00259             append("[System] AFK mode disabled", "#31c209")
00260
00261
00263     def toggle_dark():
00264         if btn_dark.isChecked():
00265             app.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
00266             btn_dark.setText("Light Mode")
00267         else:
00268             app.setStyleSheet("")
00269             btn_dark.setText("Dark Mode")
00270
00271
00273     def open_settings():
00274         dlg = SettingsDialog(config)
```

```
00275         dlg.exec_()
00276
00277     # Connect GUI controls to functionality
00278     btn_send.clicked.connect(send_message)
00279     msg_input.returnPressed.connect(send_message)
00280     btn_img.clicked.connect(send_image)
00281     btn_clients.clicked.connect(show_clients)
00282     btn_leave.clicked.connect(leave_chat)
00283     btn_afk.clicked.connect(toggle_afk)
00284     btn_dark.clicked.connect(toggle_dark)
00285     btn_settings.clicked.connect(open_settings)
00286
00287     already_left = set()
00288
00289
00291     def poll_network():
00292         current = {h for (h, _, _) in config['peers'] if h != handle}
00293         newcomers = current - local_peers
00294         for h in sorted(newcomers):
00295             append(f"{h} joined the chat.", "#2A8940")
00296         local_peers.update(newcomers)
00297
00298         while from_network.poll():
00299             typ, src, payload = from_network.recv()
00300             if typ == 'MSG':
00301                 append(f"{src}: {payload}", "#204EB4")
00302             elif typ == 'IMG':
00303                 append(f"{src} sent image → {payload}", "#204EB4")
00304                 open_file(payload)
00305             elif typ == 'LEAVE':
00306                 if src in already_left:
00307                     return
00308                 already_left.add(src)
00309                 append(f"WARNING {src} left the chat.", "#D60C0C")
00310                 if src in local_peers:
00311                     local_peers.remove(src)
00312                 config['peers'][:] = [p for p in config['peers'] if p[0] != src]
00313
00314     append(f"Welcome, {handle}!", "#000000")
00315
00316
00319     def on_close_event(event):
00320         nonlocal already_closing
00321         if not already_closing:
00322             to_network.send(("LEAVE", handle, ""))
00323             to_network.send(("EXIT", "", ""))
00324         already_closing = True
00325         event.accept()
00326
00327     # Use QTimer instead of threads to poll for new data
00328     timer = QTimer()
00329     timer.timeout.connect(poll_network)
00330     timer.start(50)
00331
00332     wnd.show()
00333     app.exec_()
```

6.3.1.3 open_file()

```
gui.open_file (
                path)
```

Open a file with the default system application.

Parameters

path	Path to the file to open.

Definition at line 53 of file gui.py.

```
00053 def open_file(path):
00054     if platform.system() == 'Windows':
00055         os.startfile(path)
00056     elif platform.system() == 'Darwin':
00057         subprocess.Popen(['open', path])
00058     else:
00059         subprocess.Popen(['xdg-open', path])
00060
```

6.3.1.4 ts()

```
gui.ts ()
```

Generate a timestamp string for message labeling.

Returns

Timestamp in the format [HH:MM:SS]

Definition at line 46 of file gui.py.

```
00046 def ts():
00047     from datetime import datetime
00048     return datetime.now().strftime("[%H:%M:%S]")
00049
```

6.3.2 Variable Documentation

6.3.2.1 CONFIG_FILE

```
str gui.CONFIG_FILE = "config.toml"
```

Definition at line 41 of file gui.py.

6.3.2.2 MAX_DISPLAY_CHUNK

```
int gui.MAX_DISPLAY_CHUNK = 200
```

Definition at line 40 of file gui.py.

6.4 main Namespace Reference

Functions

- port_in_use (port)

  Checks whether the specified UDP port is already in use.
- save_config_to_file (cfg_all)

  Saves the full client configuration back to the TOML file.
- main ()

  Entry point for launching the GUI-based SLCP chat client.

6.4.1 Function Documentation

6.4.1.1 main()

```
main.main ()
```

Entry point for launching the GUI-based SLCP chat client.
Loads user configuration, initializes inter-process pipes, and spawns child processes. Waits for the GUI to terminate before performing graceful shutdown.
Definition at line 67 of file main.py.

```
00067 def main():
00068     # Load configuration from TOML file
00069     cfg_all = toml.load("config.toml")
00070     clients = cfg_all.get("clients", [])
00071     if not clients:
00072         print("No [[clients]] section found in config.toml.")
00073         sys.exit(1)
00074
00075     # Ensure a handle argument is provided
00076     if len(sys.argv) != 2:
00077         handles = [c["handle"] for c in clients]
00078         print("Usage: python main.py <Handle>")
00079         print("Available handles:", ", ".join(handles))
00080         sys.exit(1)
00081
00082     # Match provided handle to a client config
00083     chosen = sys.argv[1]
00084     client_index = next((i for i, c in enumerate(clients) if c["handle"] == chosen), None)
00085     if client_index is None:
00086         print(f"Handle '{chosen}' not found.")
00087         sys.exit(1)
00088
00089     config = clients[client_index]
```

```
00090     manager = multiprocessing.Manager()
00091     config["peers"] = manager.list()         # Shared list of known peers
00092     config["__cfg_all"] = cfg_all            # Full config for saving later
00093     config["__cfg_index"] = clients.index(config)  # Index of this client in the TOML file
00094
00095     # Create pipes for inter-process communication
00096     ui2net_p, ui2net_c = multiprocessing.Pipe()     # GUI → Network
00097     net2ui_p, net2ui_c = multiprocessing.Pipe()     # Network → GUI
00098     disc_ctrl_parent, disc_ctrl_child = multiprocessing.Pipe()  # Main → Discovery (for stopping)
00099
00100     # Start discovery process only if port is free
00101     if not port_in_use(config["whoisport"]):
00102         p_disc = multiprocessing.Process(target=discovery_process, args=(config, disc_ctrl_child))
00103         p_disc.start()
00104         print(f"[INFO] Discovery service started on port {config['whoisport']}")
00105     else:
00106         p_disc = None
00107         print(f"[INFO] Discovery service already running on port {config['whoisport']}, not starting
      again.")
00108
00109     # Start network and GUI processes
00110     p_net = multiprocessing.Process(target=network_process, args=(config, ui2net_c, net2ui_p))
00111     p_gui = multiprocessing.Process(target=gui_process,       args=(config, ui2net_p, net2ui_c))
00112
00113     p_net.start()
00114     p_gui.start()
00115
00116     # Wait for GUI process to exit (user closed window)
00117     p_gui.join()
00118
00119     # Stop discovery process if we started it
00120     if p_disc:
00121         disc_ctrl_parent.send("STOP")
00122         p_disc.join()
00123
00124     # Notify network process to exit cleanly
00125     ui2net_p.send(("EXIT", "", ""))
00126     p_net.join()
00127
```

6.4.1.2 port_in_use()

```
main.port_in_use (
              port)
```

Checks whether the specified UDP port is already in use.
This function attempts to bind to the given port to detect conflicts.

Parameters

port	UDP port number to test.

Returns

True if the port is currently in use, False if available.

Definition at line 44 of file main.py.

```
00044 def port_in_use(port):
00045     with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
00046         try:
00047             s.bind(("", port))  # Try to bind to the port
00048             return False        # Success: port not in use
00049         except OSError:
00050             return True         # Port already in use
00051
```

6.4.1.3 save_config_to_file()

```
main.save_config_to_file (
              cfg_all)
```

Saves the full client configuration back to the TOML file.
Used when users update settings via the GUI's configuration dialog.

Parameters

cfg_all	Full configuration dictionary (includes all [[clients]]).

Definition at line 58 of file main.py.

```
00058 def save_config_to_file(cfg_all):
00059     with open("config.toml", "w") as f:
00060         toml.dump(cfg_all, f)
00061
```

6.5 network Namespace Reference

Functions

- send_image_via_tcp (config, dest_handle, filepath, peer_ip, peer_port)
- network_process (config, ui2net, net2ui)

Variables

- int MAX_UDP_SIZE = 65507

6.5.1 Detailed Description

```
@file network.py
@brief Handles SLCP networking logic including peer discovery, messaging, AFK handling, and image transfer ove

This module implements the core networking layer of the SLCP protocol. It allows clients to send and receive m
```

6.5.2 Function Documentation

6.5.2.1 network_process()

```
network.network_process (
              config,
              ui2net,
              net2ui)
```

Definition at line 51 of file network.py.

```
00051 def network_process(config, ui2net, net2ui):
00052     handle     = config["handle"]
00053     port       = config["port"][0]
00054     whoisport  = config["whoisport"]
00055     peers      = config["peers"]
00056     autoreply  = config["autoreply"]
00057     away       = config.get("away", False)
00058     img_path   = config["imagepath"]
00059     os.makedirs(img_path, exist_ok=True)  # Ensure image directory exists
00060
00061     afk_replied_to = set()  # Tracks who we've already sent AFK autoreplies to
00062
00063     # Create and bind UDP socket
00064     udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00065     udp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
00066     udp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
00067     udp_sock.bind(("", port))
00068     udp_sock.setblocking(False)
00069
00070     # Periodically broadcast JOIN
00071     def send_periodic_join():
00072         while True:
00073             try:
00074                 msg = f"JOIN {handle} {port}".encode("utf-8")
00075                 udp_sock.sendto(msg, ("255.255.255.255", whoisport))
00076             except Exception as e:
00077                 print(f"[JOIN] Error while sending: {e}")
00078             time.sleep(5)
00079
00080     # Periodically broadcast WHO
00081     def send_periodic_who():
00082         while True:
00083             try:
```

```
00084                 udp_sock.sendto(b"WHO", ("255.255.255.255", whoisport))
00085             except Exception as e:
00086                 print(f"[WHO] Error while sending: {e}")
00087             time.sleep(5)
00088
00089     # Start periodic broadcast threads
00090     threading.Thread(target=send_periodic_join, daemon=True).start()
00091     threading.Thread(target=send_periodic_who, daemon=True).start()
00092
00093     # Main event loop
00094     while True:
00095         # Handle UI commands
00096         if ui2net.poll():
00097             cmd, dest, payload = ui2net.recv()
00098
00099             if cmd == "EXIT":
00100                 print("[NETWORK] EXIT received. Notifying peers and shutting down.")
00101                 # Send LEAVE to all known peers before shutdown
00102                 for h, ip, pt in peers:
00103                     try:
00104                         udp_sock.sendto(f"LEAVE {handle}".encode("utf-8"), (ip, pt))
00105                     except Exception as e:
00106                         print(f"[LEAVE] Error notifying {h}: {e}")
00107                 break  # Exit main loop and shut down process
00108
00109             if cmd == "MSG":
00110                 # Standard SLCP message
00111                 header = f"MSG {handle} {dest} {payload}".encode("utf-8")
00112                 for h, ip, pt in peers:
00113                     if h == dest:
00114                         udp_sock.sendto(header, (ip, pt))
00115
00116             elif cmd == "IMG":
00117                 for h, ip, pt in peers:
00118                     if h == dest:
00119                         send_image_via_tcp(config, dest, payload, ip, pt)
00120
00121             elif cmd == "LEAVE":
00122                 for h, ip, pt in peers:
00123                     udp_sock.sendto(f"LEAVE {handle}".encode("utf-8"), (ip, pt))
00124
00125             elif cmd == "AFK":
00126                 # AFK status toggling
00127                 status = payload.strip().upper()
00128                 away = (status == "ON")
00129                 config["away"] = away
00130                 if not away:
00131                     afk_replied_to.clear()
00132                 print(f"[NETWORK] AFK mode {'enabled' if away else 'disabled'}.")
00133
00134
00135         # Handle incoming UDP packets
00136         try:
00137             data, addr = udp_sock.recvfrom(MAX_UDP_SIZE)
00138         except BlockingIOError:
00139             time.sleep(0.01)
00140             continue
00141
00142         try:
00143             text = data.decode("utf-8").strip()
00144         except UnicodeDecodeError:
00145             continue
00146
00147         if not text:
00148             continue
00149
00150         parts = text.split()
00151         cmd = parts[0]
00152
00153         # Handle incoming chat message
00154         if cmd == "MSG" and len(parts) >= 4:
00155             src, dest = parts[1], parts[2]
00156             msg = ' '.join(parts[3:])
00157             if dest == handle:
00158                 net2ui.send(("MSG", src, msg))
00159
00160                 # Auto-reply if in AFK mode
00161                 if away and src not in afk_replied_to:
00162                     udp_sock.sendto(
00163                         f"MSG {handle} {src} {autoreply}".encode("utf-8"),
00164                         addr
00165                     )
00166                     afk_replied_to.add(src)
00167
00168         # Handle incoming image transfer initiation
00169         elif cmd == "IMG" and len(parts) == 5:
00170             src, dest, tcp_port_s, size_s = parts[1], parts[2], parts[3], parts[4]
```

```
00171             if dest != handle:
00172                 continue
00173
00174             tcp_port = int(tcp_port_s)
00175             size     = int(size_s)
00176
00177             # Connect to sender's temporary TCP server and download image
00178             client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
00179             client.connect((addr[0], tcp_port))
00180
00181             buf = b""
00182             while len(buf) < size:
00183                 chunk = client.recv(4096)
00184                 if not chunk:
00185                     break
00186                 buf += chunk
00187             client.close()
00188
00189             # Save image to file and notify UI
00190             fn = os.path.join(img_path, f"{src}_{int(time.time())}.png")
00191             with open(fn, "wb") as f:
00192                 f.write(buf)
00193             net2ui.send(("IMG", src, fn))
00194
00195         # Handle LEAVE notifications
00196         elif cmd == "LEAVE" and len(parts) == 2:
00197             leaver = parts[1]
00198             net2ui.send(("LEAVE", leaver, ""))
00199             peers[:] = [p for p in peers if p[0] != leaver]
00200
00201         # Handle KNOWUSERS message to update peer list
00202         elif cmd == "KNOWUSERS":
00203             rest = text[len("KNOWUSERS "):]
00204             for chunk in rest.split('.'):
00205                 if not chunk.strip():
00206                     continue
00207                 try:
00208                     h, ip, pt = chunk.strip().split()
00209                     entry = (h, ip, int(pt))
00210                     if h != handle and entry not in peers:
00211                         peers.append(entry)
00212                         print(f"[KNOWUSERS] New peer: {entry}")
00213                 except ValueError:
00214                     continue
00215
00216         time.sleep(0.01)
```

6.5.2.2 send_image_via_tcp()

```
network.send_image_via_tcp (
                config,
                dest_handle,
                filepath,
                peer_ip,
                peer_port)
```

Definition at line 19 of file network.py.

```
00019 def send_image_via_tcp(config, dest_handle, filepath, peer_ip, peer_port):
00020     handle   = config["handle"]
00021     img_path = config["imagepath"]
00022     data = open(filepath, "rb").read()   # Read image file as bytes
00023     size = len(data)
00024
00025     # Set up temporary TCP server to send image
00026     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
00027     server.bind(("", 0))                  # Bind to a random free port
00028     server.listen(1)
00029     tcp_port = server.getsockname()[1]   # Get the chosen port
00030
00031     # Notify recipient via UDP
00032     udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00033     udp.sendto(f"IMG {handle} {dest_handle} {tcp_port} {size}".encode("utf-8"),
00034                (peer_ip, peer_port))
00035     udp.close()
00036
00037     # Serve the image in a separate thread
00038     def _serve():
00039         conn, _ = server.accept()
00040         conn.sendall(data)   # Send image data
00041         conn.close()
00042         server.close()
00043
00044     threading.Thread(target=_serve, daemon=True).start()
```

```
00045
00046 # Main network process responsible for handling all networking logic
00047 #
00048 # @param config   Client configuration dictionary.
00049 # @param ui2net   Pipe for receiving commands from the UI (CLI or GUI).
00050 # @param net2ui   Pipe for sending events back to the UI.
```

6.5.3 Variable Documentation

6.5.3.1 MAX_UDP_SIZE

```
int network.MAX_UDP_SIZE = 65507
```

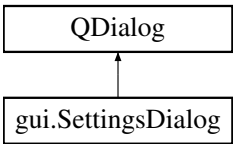
Definition at line 10 of file network.py.

Chapter 7

Class Documentation

7.1 gui.SettingsDialog Class Reference

Dialog window for editing and saving user configuration.
Inheritance diagram for gui.SettingsDialog:



Public Member Functions

- __init__ (self, config, parent=None)

  Constructor for SettingsDialog.
- save (self)

  Validate and persist user configuration to disk.

Public Attributes

- handle_field = QLineEdit(config["handle"])
- port_field = QLineEdit(str(config["port"][0]))
- autoreply_field = QLineEdit(config["autoreply"])
- imagepath_field = QLineEdit(config["imagepath"])
- save
- config = config

7.1.1 Detailed Description

Dialog window for editing and saving user configuration.
Provides editable fields for handle, port, autoreply message, and image directory. Saves updated values directly into the configuration TOML file and notifies the user.
Definition at line 84 of file gui.py.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 __init__()

```
gui.SettingsDialog.__init__ (
              self,
              config,
              parent = None)
```

Constructor for SettingsDialog.

Parameters

config	Reference to the active client configuration.
parent	Parent QWidget, if any.

Definition at line 89 of file gui.py.

```
00089     def __init__(self, config, parent=None):
00090         super().__init__(parent)
00091         self.setWindowTitle("Settings")
00092         layout = QFormLayout(self)
00093 
00094         self.handle_field = QLineEdit(config["handle"])
00095         self.port_field = QLineEdit(str(config["port"][0]))
00096         self.autoreply_field = QLineEdit(config["autoreply"])
00097         self.imagepath_field = QLineEdit(config["imagepath"])
00098 
00099         layout.addRow("Handle:", self.handle_field)
00100         layout.addRow("Port:", self.port_field)
00101         layout.addRow("Autoreply:", self.autoreply_field)
00102         layout.addRow("Image-Ordner:", self.imagepath_field)
00103 
00104         save_btn = QPushButton("Save")
00105         save_btn.clicked.connect(self.save)
00106         layout.addWidget(save_btn)
00107 
00108         self.config = config
00109 
```

7.1.3 Member Function Documentation

7.1.3.1 save()

```
gui.SettingsDialog.save (
            self)
```

Validate and persist user configuration to disk.

Updates local config dictionary and re-writes the TOML file. Informs user via message box on success or error.

Definition at line 115 of file gui.py.

```
00115     def save(self):
00116         try:
00117             self.config["handle"] = self.handle_field.text()
00118             self.config["port"][0] = int(self.port_field.text())
00119             self.config["autoreply"] = self.autoreply_field.text()
00120             self.config["imagepath"] = self.imagepath_field.text()
00121             os.makedirs(self.config["imagepath"], exist_ok=True)
00122 
00123             all_cfg = self.config["__cfg_all"]
00124             index = self.config["__cfg_index"]
00125 
00126             clean_config = {
00127                 k: v for k, v in self.config.items()
00128                 if k not in ("peers", "__cfg_all", "__cfg_index")
00129             }
00130 
00131             all_cfg["clients"][index] = clean_config
00132             with open("config.toml", "w") as f:
00133                 toml.dump(all_cfg, f)
00134 
00135             QMessageBox.information(self, "Saved", "Configuration saved. Please restart the program.")
00136             self.accept()
00137 
00138         except Exception as e:
00139             QMessageBox.warning(self, "Error", f"Invalid input: {e}")
00140 
```

7.1.4 Member Data Documentation

7.1.4.1 autoreply_field

```
gui.SettingsDialog.autoreply_field = QLineEdit(config["autoreply"])
```

Definition at line 96 of file gui.py.

7.1.4.2 config

```
gui.SettingsDialog.config = config
```

Definition at line 108 of file gui.py.

7.1.4.3 handle_field

```
gui.SettingsDialog.handle_field = QLineEdit(config["handle"])
```
Definition at line 94 of file gui.py.

7.1.4.4 imagepath_field

```
gui.SettingsDialog.imagepath_field = QLineEdit(config["imagepath"])
```
Definition at line 97 of file gui.py.

7.1.4.5 port_field

```
gui.SettingsDialog.port_field = QLineEdit(str(config["port"][0]))
```
Definition at line 95 of file gui.py.

7.1.4.6 save

```
gui.SettingsDialog.save
```
Definition at line 105 of file gui.py.
The documentation for this class was generated from the following file:

- processes/gui.py

Chapter 8

File Documentation

8.1 cli.py File Reference

Command-line interface for SLCP peer-to-peer chat.

Namespaces

- namespace cli

Functions

- cli.ts ()

  Returns a formatted timestamp string.
- cli.port_in_use (port)

  Checks if a UDP port is currently in use.
- cli.print_commands ()

  Prints a list of available CLI commands.
- cli.main ()

  Main function that initializes the CLI chat client.

Variables

- str cli.CONFIG_FILE = "config.toml"
- str cli.COLOR_RESET = "\033[0m"
- str cli.COLOR_GREEN = "\033[92m"
- str cli.COLOR_RED = "\033[91m"
- str cli.COLOR_YELLOW = "\033[93m"

8.1.1 Detailed Description

Command-line interface for SLCP peer-to-peer chat.
This CLI allows users to start a chat client, send messages and images, toggle AFK mode, and list active peers on the network. It communicates with discovery and network subprocesses and uses inter-process communication (IPC) pipes to send and receive events.

8.1.2 Features

- Text and image messaging
- Peer discovery
- AFK autoreply toggle
- Dynamic client configuration from config.toml

8.1.3 Usage

Run the CLI as:

```
python cli.py <Handle>
```

The handle must be defined in `config.toml`.
Definition in file cli.py.

8.2 cli.py

Go to the documentation of this file.

```
00001
00022
00023 import socket
00024 import os
00025 import multiprocessing
00026 import sys
00027 import threading
00028 import time
00029 from datetime import datetime
00030
00031 import toml
00032 from processes.discovery import discovery_process
00033 from processes.network import network_process
00034
00035 CONFIG_FILE = "config.toml"
00036
00037 # ANSI escape codes for terminal colors
00038 COLOR_RESET  = "\033[0m"
00039 COLOR_GREEN  = "\033[92m"
00040 COLOR_RED    = "\033[91m"
00041 COLOR_YELLOW = "\033[93m"
00042
00043
00045 def ts():
00046     return datetime.now().strftime("[%H:%M:%S]")
00047
00048
00052 def port_in_use(port):
00053     with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
00054         try:
00055             s.bind(("", port))
00056             return False
00057         except OSError:
00058             return True
00059
00060
00062 def print_commands():
00063     print("\nAvailable commands:")
00064     print("  msg <handle> <text>")
00065     print("  img <handle> <path_to_image>")
00066     print("  clients")
00067     print("  afk on|off")
00068     print("  leave\n")
00069
00070
00073 def main():
00074     cfg_all = toml.load(CONFIG_FILE)
00075     clients = cfg_all.get("clients", [])
00076     if not clients:
00077         print("No [[clients]] section found in config.toml.")
00078         sys.exit(1)
00079
00080     if len(sys.argv) != 2:
00081         handles = [c["handle"] for c in clients]
00082         print("Usage: python cli.py <Handle>")
00083         print("Available handles:", ", ".join(handles))
00084         sys.exit(1)
00085
00086     chosen = sys.argv[1]
00087     client_index = next((i for i, c in enumerate(clients) if c["handle"] == chosen), None)
00088     if client_index is None:
00089         print(f"Handle '{chosen}' not found.")
00090         sys.exit(1)
00091
00092     # Prepare client configuration and shared state
00093     config = clients[client_index]
00094     manager = multiprocessing.Manager()
00095     config["peers"] = manager.list()
00096     config["__cfg_all"] = cfg_all
00097     config["__cfg_index"] = clients.index(config)
00098
00099     # Inter-process communication pipes
```

```
00100     ui2net_p, ui2net_c = multiprocessing.Pipe()
00101     net2ui_p, net2ui_c = multiprocessing.Pipe()
00102     disc_ctrl_parent, disc_ctrl_child = multiprocessing.Pipe()
00103
00104     # Start discovery process if not already running
00105     p_disc = None
00106     if not port_in_use(config["whoisport"]):
00107         p_disc = multiprocessing.Process(target=discovery_process, args=(config, disc_ctrl_child))
00108         p_disc.start()
00109         print(f"[INFO] Discovery service started on port {config['whoisport']}")
00110     else:
00111         print(f"[INFO] Discovery already running on port {config['whoisport']}")
00112
00113     # Start network process
00114     p_net = multiprocessing.Process(target=network_process, args=(config, ui2net_c, net2ui_p))
00115     p_net.start()
00116
00117     stop_event = threading.Event()
00118     left_peers = set()
00119
00120
00122     def poll_network():
00123         while not stop_event.is_set():
00124             while net2ui_c.poll():
00125                 typ, src, payload = net2ui_c.recv()
00126                 if typ == 'MSG':
00127                     print(f"\n{COLOR_GREEN}{ts()} [{src}] {payload}{COLOR_RESET}\n")
00128                 elif typ == 'IMG':
00129                     print(f"\n{COLOR_YELLOW}{ts()} [{src}] sent image → {payload}{COLOR_RESET}\n")
00130                 elif typ == 'LEAVE':
00131                     if src not in left_peers:
00132                         print(f"\n{COLOR_RED}{ts()} [{src}] left the chat.{COLOR_RESET}\n")
00133                         left_peers.add(src)
00134                     config['peers'][:] = [p for p in config['peers'] if p[0] != src]
00135             time.sleep(0.05)
00136
00137     threading.Thread(target=poll_network, daemon=True).start()
00138
00139     print(f"\n========== SLCP CLI Chat started as '{chosen}' ==========")
00140     print_commands()
00141
00142     try:
00143         while True:
00144             cmd = input("> ").strip()
00145             if not cmd:
00146                 continue
00147
00148             parts = cmd.split(" ", 2)
00149             action = parts[0].lower()
00150
00151             if action == "leave":
00152                 print("Sending LEAVE...")
00153                 ui2net_p.send(("LEAVE", "", ""))
00154
00155                 if p_disc:
00156                     disc_ctrl_parent.send("STOP")
00157                     p_disc.join()
00158                     print("[INFO] Discovery stopped.")
00159
00160                 ui2net_p.send(("EXIT", "", ""))
00161                 p_net.join()
00162                 stop_event.set()
00163                 time.sleep(0.1)
00164                 break
00165
00166             elif action == "clients":
00167                 peers = [(h, ip, pt) for (h, ip, pt) in config['peers'] if h != chosen]
00168                 if not peers:
00169                     print("No other clients found.")
00170                 else:
00171                     print("\nActive clients:")
00172                     for (h, ip, pt) in peers:
00173                         print(f"  {h} ({ip}:{pt})")
00174                     print()
00175
00176             elif action == "msg" and len(parts) >= 3:
00177                 dest = parts[1]
00178                 msg = parts[2]
00179                 ui2net_p.send(("MSG", dest, msg))
00180                 print(f"[SEND] to {dest}: {msg}")
00181
00182             elif action == "img" and len(parts) >= 3:
00183                 dest = parts[1]
00184                 path = parts[2]
00185                 if not os.path.isfile(path):
00186                     print(f"[ERROR] File not found: {path}")
00187                     continue
```

```
00188                 ui2net_p.send(("IMG", dest, path))
00189                 print(f"[SEND IMG] to {dest}: {path}")
00190
00191             elif action == "afk" and len(parts) == 2:
00192                 mode = parts[1].lower()
00193                 if mode in ("on", "off"):
00194                     ui2net_p.send(("AFK", chosen, mode.upper()))
00195                     print(f"[AFK] set to {mode.upper()}")
00196                 else:
00197                     print("[ERROR] Usage: afk on|off")
00198
00199             elif action == "help":
00200                 print_commands()
00201
00202             else:
00203                 print("[ERROR] Unknown command. Type 'help' for commands.")
00204
00205             time.sleep(0.05)
00206
00207     finally:
00208         stop_event.set()
00209         time.sleep(0.1)
00210
00211         if p_disc:
00212             disc_ctrl_parent.send("STOP")
00213             p_disc.join()
00214
00215         ui2net_p.send(("EXIT", "", ""))
00216         p_net.join()
00217
00218
00219 if __name__ == "__main__":
00220     main()
```

8.3 config.toml File Reference

8.4 config.toml

Go to the documentation of this file.

```
00001 ##
00002 # @file config.toml
00003 # @brief Configuration file for SLCP clients.
00004 #
00005 # This configuration file defines the list of available clients for the
00006 # SLCP peer-to-peer chat application. Each client section contains
00007 # the networking and UI preferences used to start an instance.
00008 #
00009 # @section format_sec Format
00010 # The file uses the TOML format and contains a list of [[clients]] tables.
00011 #
00012 # @section fields_sec Fields
00013 # - handle: Unique name/identifier for the client.
00014 # - port: List of two ports. First is the UDP port for peer communication,
00015 #         second is typically used for image transfer (or a reserved fallback).
00016 # - whoisport: Broadcast port used for WHO and JOIN messages.
00017 # - autoreply: Message sent automatically when the user is AFK.
00018 # - away: Boolean flag indicating whether the user starts in AFK mode.
00019 # - imagepath: Path to the local folder where received images will be stored.
00020 #
00021 # @note All clients share the same whoisport for discovery purposes.
00022
00023 [[clients]]
00024 handle = "Aashir"                    # Unique name for the client
00025 port = [ 5008, 6000 ]                # UDP port for chat, secondary port
00026 whoisport = 4000                     # Broadcast port for WHO/JOIN messages
00027 autoreply = "in einer Stunde da"     # AFK auto-response message
00028 away = false                         # Whether the client is initially AFK
00029 imagepath = "./images/aashir"        # Directory for storing received images
00030
00031 [[clients]]
00032 handle = "Bratli"
00033 port = [ 5004, 6000 ]
00034 whoisport = 4000
00035 autoreply = "Bin AFK."
00036 away = false
00037 imagepath = "./images/bratli"
00038
00039 [[clients]]
00040 handle = "Jalal"
00041 port = [ 5005, 6000 ]
00042 whoisport = 4000
00043 autoreply = "Bin in 10 Minuten wieder da."
```

```
00044 away = false
00045 imagepath = "./images/jalal"
```

8.5 main.py File Reference

Main graphical launcher for the SLCP (Simple Local Chat Protocol) peer-to-peer chat application.

Namespaces

- namespace main

Functions

- main.port_in_use (port)

  Checks whether the specified UDP port is already in use.
- main.save_config_to_file (cfg_all)

  Saves the full client configuration back to the TOML file.
- main.main ()

  Entry point for launching the GUI-based SLCP chat client.

8.5.1 Detailed Description

Main graphical launcher for the SLCP (Simple Local Chat Protocol) peer-to-peer chat application.
This script serves as the entry point for launching the GUI version of the SLCP client. It reads configuration from a TOML file, initializes inter-process communication pipes, and starts three key processes:

- `discovery_process` (UDP broadcast peer discovery)
- `network_process` (handles SLCP messages, AFK state, and file/image transfers)
- `gui_process` (PyQt5 graphical chat interface)

Main responsibilities:

- Validates user input and config structure
- Manages subprocess startup and shutdown
- Coordinates inter-process communication
- Persists updated configuration settings to `config.toml`

Usage:

```
python main.py <Handle>
```

where `<Handle>` corresponds to a user listed in `config.toml` under [[clients]].

Date

June 2025

Author

SLCP

Definition in file main.py.

8.6 main.py

Go to the documentation of this file.

```
00001
00027
00028 import sys
00029 import toml
00030 import multiprocessing
00031 import socket
00032 import os
00033 from processes.discovery import discovery_process
00034 from processes.network   import network_process
00035 from processes.gui       import gui_process
00036
00037
00044 def port_in_use(port):
00045     with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
00046         try:
00047             s.bind(("", port))  # Try to bind to the port
00048             return False        # Success: port not in use
00049         except OSError:
00050             return True         # Port already in use
00051
00052
00058 def save_config_to_file(cfg_all):
00059     with open("config.toml", "w") as f:
00060         toml.dump(cfg_all, f)
00061
00062
00067 def main():
00068     # Load configuration from TOML file
00069     cfg_all = toml.load("config.toml")
00070     clients = cfg_all.get("clients", [])
00071     if not clients:
00072         print("No [[clients]] section found in config.toml.")
00073         sys.exit(1)
00074
00075     # Ensure a handle argument is provided
00076     if len(sys.argv) != 2:
00077         handles = [c["handle"] for c in clients]
00078         print("Usage: python main.py <Handle>")
00079         print("Available handles:", ", ".join(handles))
00080         sys.exit(1)
00081
00082     # Match provided handle to a client config
00083     chosen = sys.argv[1]
00084     client_index = next((i for i, c in enumerate(clients) if c["handle"] == chosen), None)
00085     if client_index is None:
00086         print(f"Handle '{chosen}' not found.")
00087         sys.exit(1)
00088
00089     config = clients[client_index]
00090     manager = multiprocessing.Manager()
00091     config["peers"] = manager.list()         # Shared list of known peers
00092     config["__cfg_all"] = cfg_all            # Full config for saving later
00093     config["__cfg_index"] = clients.index(config)  # Index of this client in the TOML file
00094
00095     # Create pipes for inter-process communication
00096     ui2net_p, ui2net_c = multiprocessing.Pipe()     # GUI → Network
00097     net2ui_p, net2ui_c = multiprocessing.Pipe()     # Network → GUI
00098     disc_ctrl_parent, disc_ctrl_child = multiprocessing.Pipe()  # Main → Discovery (for stopping)
00099
00100     # Start discovery process only if port is free
00101     if not port_in_use(config["whoisport"]):
00102         p_disc = multiprocessing.Process(target=discovery_process, args=(config, disc_ctrl_child))
00103         p_disc.start()
00104         print(f"[INFO] Discovery service started on port {config['whoisport']}")
00105     else:
00106         p_disc = None
00107         print(f"[INFO] Discovery service already running on port {config['whoisport']}, not starting
      again.")
00108
00109     # Start network and GUI processes
00110     p_net = multiprocessing.Process(target=network_process, args=(config, ui2net_c, net2ui_p))
00111     p_gui = multiprocessing.Process(target=gui_process,       args=(config, ui2net_p, net2ui_c))
00112
00113     p_net.start()
00114     p_gui.start()
00115
00116     # Wait for GUI process to exit (user closed window)
00117     p_gui.join()
00118
00119     # Stop discovery process if we started it
00120     if p_disc:
00121         disc_ctrl_parent.send("STOP")
00122         p_disc.join()
```

```
00123
00124     # Notify network process to exit cleanly
00125     ui2net_p.send(("EXIT", "", ""))
00126     p_net.join()
00127
00128
00132 if __name__ == "__main__":
00133     main()
```

8.7 processes/discovery.py File Reference

Peer discovery module for SLCP (Simple LAN Chat Protocol)

Namespaces

- namespace discovery

Functions

- discovery.get_local_ip ()

  Get the local machine's IP address.
- discovery.discovery_process (config, ctrl_pipe)

  Main discovery process function.

8.7.1 Detailed Description

Peer discovery module for SLCP (Simple LAN Chat Protocol)
This module implements the decentralized peer discovery mechanism used by the SLCP chat application. It uses UDP broadcasting to detect other clients on the same network via `JOIN`, `WHO`, and `KNOWUSERS` messages. One process takes the role of a WHO responder and provides a list of all known clients upon request.
The discovery process is run in a separate process and periodically sends discovery messages to update the local peer list.

Author

Group SLCP

Date

June 2025

Definition in file discovery.py.

8.8 discovery.py

Go to the documentation of this file.

```
00001
00014
00015 import socket
00016 import time
00017
00018
00025 def get_local_ip():
00026     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00027     try:
00028         s.connect(("8.8.8.8", 80))
00029         return s.getsockname()[0]
00030     except Exception:
00031         return "127.0.0.1"
00032     finally:
00033         s.close()
00034
00035
00051 def discovery_process(config, ctrl_pipe):
00052     handle    = config["handle"]
00053     port      = config["port"][0]
00054     whoisport = config["whoisport"]
```

```
00055     peers     = config["peers"]
00056
00057     responder = False  # Only one client becomes WHO responder
00058
00059     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00060     sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
00061     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
00062
00063     try:
00064         sock.bind(("", whoisport))
00065         responder = True
00066         print(f"[Discovery] WHO responder active on {whoisport}")
00067     except OSError as e:
00068         print(f"[Discovery] Not WHO responder - {e}")
00069
00070     sock.settimeout(1.0)
00071
00072
00075     def broadcast(msg: str):
00076         sock.sendto(msg.encode("utf-8"), ("255.255.255.255", whoisport))
00077
00078     while True:
00079         # Handle stop command from main process
00080         if ctrl_pipe.poll():
00081             cmd = ctrl_pipe.recv()
00082             if cmd == "STOP":
00083                 print("[Discovery] Terminated by main process.")
00084                 break
00085
00086         # Broadcast JOIN and WHO messages
00087         broadcast(f"JOIN {handle} {port}\n")
00088         broadcast("WHO\n")
00089
00090         start = time.time()
00091         while time.time() - start < 1.0:
00092             try:
00093                 data, addr = sock.recvfrom(4096)
00094             except socket.timeout:
00095                 break
00096
00097             try:
00098                 text = data.decode("utf-8").strip()
00099             except UnicodeDecodeError:
00100                 continue
00101
00102             if not text:
00103                 continue
00104
00105             parts = text.split()
00106             cmd = parts[0]
00107
00108             # Handle JOIN message: add new peer
00109             if cmd == "JOIN" and len(parts) == 3:
00110                 peer, pport = parts[1], int(parts[2])
00111                 entry = (peer, addr[0], pport)
00112                 if peer != handle and entry not in peers:
00113                     peers.append(entry)
00114                     print(f"[Discovery] New peer detected: {entry}")
00115
00116             # Handle LEAVE message: remove peer
00117             elif cmd == "LEAVE" and len(parts) == 2:
00118                 peer = parts[1]
00119                 peers[:] = [p for p in peers if p[0] != peer]
00120
00121             # Handle WHO request: respond with known users
00122             elif cmd == "WHO" and responder:
00123                 all_known = [(handle, get_local_ip(), port)] + list(peers)
00124                 payload = ",".join(f"{h} {ip} {pt}" for h, ip, pt in all_known)
00125                 sock.sendto(f"KNOWUSERS {payload}".encode("utf-8"), addr)
00126
00127             # Handle KNOWUSERS response: merge peer list
00128             elif cmd == "KNOWUSERS":
00129                 rest = text[len("KNOWUSERS "):]
00130                 for chunk in rest.split(','):
00131                     if not chunk.strip():
00132                         continue
00133                     try:
00134                         h, ip, pt = chunk.strip().split()
00135                         entry = (h, ip, int(pt))
00136                         if h != handle and entry not in peers:
00137                             peers.append(entry)
00138                     except ValueError:
00139                         continue
00140
00141         time.sleep(3)
```

8.9 processes/gui.py File Reference

Graphical User Interface (GUI) for SLCP Chat.

Classes

- class gui.SettingsDialog

  Dialog window for editing and saving user configuration.

Namespaces

- namespace gui

Functions

- gui.ts ()

  Generate a timestamp string for message labeling.
- gui.open_file (path)

  Open a file with the default system application.
- gui.get_local_ip ()

  Get the current machine's local IP address.
- gui.gui_process (config, to_network, from_network)

  Launches the SLCP GUI as a separate process.

Variables

- int gui.MAX_DISPLAY_CHUNK = 200
- str gui.CONFIG_FILE = "config.toml"

8.9.1 Detailed Description

Graphical User Interface (GUI) for SLCP Chat.
This module provides a complete graphical frontend for the SLCP (Simple LAN Chat Protocol) client. It allows users to send and receive messages and images, manage peer visibility, toggle AFK (away-from-keyboard) mode, access configuration settings, and quit the session gracefully.
Communication with the network and discovery processes is handled via multiprocessing pipes. Built using PyQt5 and styled optionally using QDarkStyle.
Key GUI Features:

- Display chat log with timestamps and color-coded messages
- Input fields for recipient and message
- Send image button (opens file dialog)
- View active clients
- AFK mode toggle with autoreply functionality
- Dark mode toggle
- In-app configuration management

Author

SLCP Team

Date

June 2025

Definition in file gui.py.

8.10 gui.py

Go to the documentation of this file.

```
00001
00024
00025 import sys
00026 import os
00027 import subprocess
00028 import platform
00029 import socket
00030 import toml
00031 import qdarkstyle
00032 from PyQt5.QtWidgets import (
00033     QApplication, QWidget, QTextEdit, QVBoxLayout, QHBoxLayout,
00034     QLineEdit, QPushButton, QFileDialog, QMessageBox,
00035     QDialog, QFormLayout
00036 )
00037 from PyQt5.QtCore import QTimer
00038 from PyQt5.QtGui import QTextCursor, QColor
00039
00040 MAX_DISPLAY_CHUNK = 200  # Max characters per chat display chunk
00041 CONFIG_FILE = "config.toml"  # Default path to config file
00042
00043
00046 def ts():
00047     from datetime import datetime
00048     return datetime.now().strftime("[%H:%M:%S]")
00049
00050
00053 def open_file(path):
00054     if platform.system() == 'Windows':
00055         os.startfile(path)
00056     elif platform.system() == 'Darwin':
00057         subprocess.Popen(['open', path])
00058     else:
00059         subprocess.Popen(['xdg-open', path])
00060
00061
00068 def get_local_ip():
00069     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00070     try:
00071         s.connect(("8.8.8.8", 80))
00072         return s.getsockname()[0]
00073     except Exception:
00074         return "127.0.0.1"
00075     finally:
00076         s.close()
00077
00078
00084 class SettingsDialog(QDialog):
00085
00089     def __init__(self, config, parent=None):
00090         super().__init__(parent)
00091         self.setWindowTitle("Settings")
00092         layout = QFormLayout(self)
00093
00094         self.handle_field = QLineEdit(config["handle"])
00095         self.port_field = QLineEdit(str(config["port"][0]))
00096         self.autoreply_field = QLineEdit(config["autoreply"])
00097         self.imagepath_field = QLineEdit(config["imagepath"])
00098
00099         layout.addRow("Handle:", self.handle_field)
00100         layout.addRow("Port:", self.port_field)
00101         layout.addRow("Autoreply:", self.autoreply_field)
00102         layout.addRow("Image-Ordner:", self.imagepath_field)
00103
00104         save_btn = QPushButton("Save")
00105         save_btn.clicked.connect(self.save)
00106         layout.addWidget(save_btn)
00107
00108         self.config = config
00109
00110
00115     def save(self):
00116         try:
00117             self.config["handle"] = self.handle_field.text()
00118             self.config["port"][0] = int(self.port_field.text())
00119             self.config["autoreply"] = self.autoreply_field.text()
00120             self.config["imagepath"] = self.imagepath_field.text()
00121             os.makedirs(self.config["imagepath"], exist_ok=True)
00122
00123             all_cfg = self.config["__cfg_all"]
00124             index = self.config["__cfg_index"]
00125
00126             clean_config = {
00127                 k: v for k, v in self.config.items()
```

```
00128                 if k not in ("peers", "__cfg_all", "__cfg_index")
00129             }
00130
00131             all_cfg["clients"][index] = clean_config
00132             with open("config.toml", "w") as f:
00133                 toml.dump(all_cfg, f)
00134
00135             QMessageBox.information(self, "Saved", "Configuration saved. Please restart the program.")
00136             self.accept()
00137
00138         except Exception as e:
00139             QMessageBox.warning(self, "Error", f"Invalid input: {e}")
00140
00141
00151 def gui_process(config, to_network, from_network):
00152     handle = config['handle']
00153     img_path = config['imagepath']
00154     os.makedirs(img_path, exist_ok=True)
00155
00156     app = QApplication(sys.argv)
00157     wnd = QWidget()
00158     wnd.setWindowTitle(f"SLCP Chat - {handle}")
00159
00160     # Main layout
00161     vlayout = QVBoxLayout()
00162     chat = QTextEdit(); chat.setReadOnly(True)
00163     vlayout.addWidget(chat)
00164
00165     # Control layout (buttons + inputs)
00166     controls = QHBoxLayout()
00167     dest_input = QLineEdit(); dest_input.setPlaceholderText("Recipient handle")
00168     msg_input = QLineEdit(); msg_input.setPlaceholderText("Message...")
00169     btn_send = QPushButton("Send")
00170     btn_img = QPushButton("Send Image")
00171     btn_clients = QPushButton("Clients")
00172     btn_leave = QPushButton("Leave Chat")
00173     btn_afk = QPushButton("AFK: OFF"); btn_afk.setCheckable(True)
00174     btn_afk.setStyleSheet("background-color: #666; color: white;")
00175     btn_dark = QPushButton("Dark Mode"); btn_dark.setCheckable(True)
00176     btn_settings = QPushButton("Settings")
00177
00178     for w in (dest_input, msg_input, btn_send, btn_img, btn_clients, btn_settings, btn_leave, btn_afk,
      btn_dark):
00179         controls.addWidget(w)
00180     vlayout.addLayout(controls)
00181     wnd.setLayout(vlayout)
00182
00183     local_peers = set()
00184     afk_mode = False
00185
00186
00190     def append(text, color="#010202"):
00191         chat.setTextColor(QColor(color))
00192         chat.append(f"{ts()} {text}")
00193         chat.moveCursor(QTextCursor.End)
00194
00195
00197     def send_message():
00198         dest = dest_input.text().strip()
00199         msg = msg_input.text().strip()
00200         if not dest or not msg:
00201             return
00202         append(f"{handle}: {msg}", "#2A8940")
00203         to_network.send(("MSG", dest, msg))
00204         msg_input.clear()
00205
00206
00208     def send_image():
00209         path, _ = QFileDialog.getOpenFileName(wnd, "Select image", "", "Images (*.png *.jpg *.bmp
      *.webp)")
00210         if not path:
00211             return
00212         dest = dest_input.text().strip()
00213         if not dest:
00214             QMessageBox.warning(wnd, "Error", "Please enter recipient handle!")
00215             return
00216         append(f"{handle} → {dest} [Image]", "#2A8940")
00217         to_network.send(("IMG", dest, path))
00218
00219
00221     def show_clients():
00222         peers = [(h, ip, pt) for (h, ip, pt) in config['peers'] if h != handle]
00223         if not peers:
00224             QMessageBox.information(wnd, "Clients", "No other clients found.")
00225         else:
00226             local_ip = get_local_ip()
00227             local_port = config["port"][0]
```

```
00228             info = "\n".join(f"{h} ({ip}:{pt})" for (h, ip, pt) in peers)
00229             QMessageBox.information(wnd, "Clients", f"You: {handle}
    ({local_ip}:{local_port})\n\nActive clients:\n{info}")
00230
00231     already_closing = False
00232
00233
00235     def leave_chat():
00236         nonlocal already_closing
00237         if already_closing:
00238             return
00239         already_closing = True
00240         to_network.send(("LEAVE", handle, ""))
00241         to_network.send(("EXIT", "", ""))
00242         wnd.close()
00243
00244
00246     def toggle_afk():
00247         nonlocal afk_mode
00248         if btn_afk.isChecked():
00249             btn_afk.setText("AFK: ON")
00250             btn_afk.setStyleSheet("background-color: #cc5500; color: white;")
00251             afk_mode = True
00252             to_network.send(("AFK", handle, "ON"))
00253             append("[System] AFK mode enabled", "#c22809")
00254         else:
00255             btn_afk.setText("AFK: OFF")
00256             btn_afk.setStyleSheet("background-color: #666; color: white;")
00257             afk_mode = False
00258             to_network.send(("AFK", handle, "OFF"))
00259             append("[System] AFK mode disabled", "#31c209")
00260
00261
00263     def toggle_dark():
00264         if btn_dark.isChecked():
00265             app.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
00266             btn_dark.setText("Light Mode")
00267         else:
00268             app.setStyleSheet("")
00269             btn_dark.setText("Dark Mode")
00270
00271
00273     def open_settings():
00274         dlg = SettingsDialog(config)
00275         dlg.exec_()
00276
00277     # Connect GUI controls to functionality
00278     btn_send.clicked.connect(send_message)
00279     msg_input.returnPressed.connect(send_message)
00280     btn_img.clicked.connect(send_image)
00281     btn_clients.clicked.connect(show_clients)
00282     btn_leave.clicked.connect(leave_chat)
00283     btn_afk.clicked.connect(toggle_afk)
00284     btn_dark.clicked.connect(toggle_dark)
00285     btn_settings.clicked.connect(open_settings)
00286
00287     already_left = set()
00288
00289
00291     def poll_network():
00292         current = {h for (h, _, _) in config['peers'] if h != handle}
00293         newcomers = current - local_peers
00294         for h in sorted(newcomers):
00295             append(f"{h} joined the chat.", "#2A8940")
00296         local_peers.update(newcomers)
00297
00298         while from_network.poll():
00299             typ, src, payload = from_network.recv()
00300             if typ == 'MSG':
00301                 append(f"{src}: {payload}", "#204EB4")
00302             elif typ == 'IMG':
00303                 append(f"{src} sent image → {payload}", "#204EB4")
00304                 open_file(payload)
00305             elif typ == 'LEAVE':
00306                 if src in already_left:
00307                     return
00308                 already_left.add(src)
00309                 append(f"WARNING {src} left the chat.", "#D60C0C")
00310                 if src in local_peers:
00311                     local_peers.remove(src)
00312                 config['peers'][:] = [p for p in config['peers'] if p[0] != src]
00313
00314     append(f"Welcome, {handle}!", "#000000")
00315
00316
00319     def on_close_event(event):
00320         nonlocal already_closing
```

```
00321         if not already_closing:
00322             to_network.send(("LEAVE", handle, ""))
00323             to_network.send(("EXIT", "", ""))
00324         already_closing = True
00325         event.accept()
00326
00327     # Use QTimer instead of threads to poll for new data
00328     timer = QTimer()
00329     timer.timeout.connect(poll_network)
00330     timer.start(50)
00331
00332     wnd.show()
00333     app.exec_()
```

8.11 processes/network.py File Reference

Namespaces

- namespace network

Functions

- network.send_image_via_tcp (config, dest_handle, filepath, peer_ip, peer_port)
- network.network_process (config, ui2net, net2ui)

Variables

- int network.MAX_UDP_SIZE = 65507

8.12 network.py

Go to the documentation of this file.

```
00001 """
00002 @file network.py
00003 @brief Handles SLCP networking logic including peer discovery, messaging, AFK handling, and image
      transfer over TCP/UDP.
00004
00005 This module implements the core networking layer of the SLCP protocol. It allows clients to send and
      receive messages and images, manage AFK states, and maintain a list of peers discovered in the
      network. Communication is done using UDP for messages and TCP for binary image transfer.
00006 """
00007
00008 import socket, os, time, threading
00009
00010 MAX_UDP_SIZE = 65507  # Maximum safe UDP packet size
00011
00012 # Sends an image via TCP after notifying the recipient via UDP
00013 #
00014 # @param config       Dictionary containing client configuration.
00015 # @param dest_handle  Handle of the recipient client.
00016 # @param filepath     Path to the image file.
00017 # @param peer_ip      IP address of the peer.
00018 # @param peer_port    UDP port of the peer.
00019 def send_image_via_tcp(config, dest_handle, filepath, peer_ip, peer_port):
00020     handle   = config["handle"]
00021     img_path = config["imagepath"]
00022     data = open(filepath, "rb").read()   # Read image file as bytes
00023     size = len(data)
00024
00025     # Set up temporary TCP server to send image
00026     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
00027     server.bind(("", 0))                  # Bind to a random free port
00028     server.listen(1)
00029     tcp_port = server.getsockname()[1]   # Get the chosen port
00030
00031     # Notify recipient via UDP
00032     udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00033     udp.sendto(f"IMG {handle} {dest_handle} {tcp_port} {size}".encode("utf-8"),
00034                (peer_ip, peer_port))
00035     udp.close()
00036
00037     # Serve the image in a separate thread
00038     def _serve():
00039         conn, _ = server.accept()
00040         conn.sendall(data)   # Send image data
00041         conn.close()
```

```
00042         server.close()
00043
00044     threading.Thread(target=_serve, daemon=True).start()
00045
00046 # Main network process responsible for handling all networking logic
00047 #
00048 # @param config   Client configuration dictionary.
00049 # @param ui2net   Pipe for receiving commands from the UI (CLI or GUI).
00050 # @param net2ui   Pipe for sending events back to the UI.
00051 def network_process(config, ui2net, net2ui):
00052     handle     = config["handle"]
00053     port       = config["port"][0]
00054     whoisport  = config["whoisport"]
00055     peers      = config["peers"]
00056     autoreply  = config["autoreply"]
00057     away       = config.get("away", False)
00058     img_path   = config["imagepath"]
00059     os.makedirs(img_path, exist_ok=True)  # Ensure image directory exists
00060
00061     afk_replied_to = set()  # Tracks who we've already sent AFK autoreplies to
00062
00063     # Create and bind UDP socket
00064     udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
00065     udp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
00066     udp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
00067     udp_sock.bind(("", port))
00068     udp_sock.setblocking(False)
00069
00070     # Periodically broadcast JOIN
00071     def send_periodic_join():
00072         while True:
00073             try:
00074                 msg = f"JOIN {handle} {port}".encode("utf-8")
00075                 udp_sock.sendto(msg, ("255.255.255.255", whoisport))
00076             except Exception as e:
00077                 print(f"[JOIN] Error while sending: {e}")
00078             time.sleep(5)
00079
00080     # Periodically broadcast WHO
00081     def send_periodic_who():
00082         while True:
00083             try:
00084                 udp_sock.sendto(b"WHO", ("255.255.255.255", whoisport))
00085             except Exception as e:
00086                 print(f"[WHO] Error while sending: {e}")
00087             time.sleep(5)
00088
00089     # Start periodic broadcast threads
00090     threading.Thread(target=send_periodic_join, daemon=True).start()
00091     threading.Thread(target=send_periodic_who, daemon=True).start()
00092
00093     # Main event loop
00094     while True:
00095         # Handle UI commands
00096         if ui2net.poll():
00097             cmd, dest, payload = ui2net.recv()
00098
00099             if cmd == "EXIT":
00100                 print("[NETWORK] EXIT received. Notifying peers and shutting down.")
00101                 # Send LEAVE to all known peers before shutdown
00102                 for h, ip, pt in peers:
00103                     try:
00104                         udp_sock.sendto(f"LEAVE {handle}".encode("utf-8"), (ip, pt))
00105                     except Exception as e:
00106                         print(f"[LEAVE] Error notifying {h}: {e}")
00107                 break  # Exit main loop and shut down process
00108
00109             if cmd == "MSG":
00110                 # Standard SLCP message
00111                 header = f"MSG {handle} {dest} {payload}".encode("utf-8")
00112                 for h, ip, pt in peers:
00113                     if h == dest:
00114                         udp_sock.sendto(header, (ip, pt))
00115
00116             elif cmd == "IMG":
00117                 for h, ip, pt in peers:
00118                     if h == dest:
00119                         send_image_via_tcp(config, dest, payload, ip, pt)
00120
00121             elif cmd == "LEAVE":
00122                 for h, ip, pt in peers:
00123                     udp_sock.sendto(f"LEAVE {handle}".encode("utf-8"), (ip, pt))
00124
00125             elif cmd == "AFK":
00126                 # AFK status toggling
00127                 status = payload.strip().upper()
00128                 away = (status == "ON")
```

```
00129                 config["away"] = away
00130                 if not away:
00131                     afk_replied_to.clear()
00132                 print(f"[NETWORK] AFK mode {'enabled' if away else 'disabled'}.")
00133 
00134 
00135         # Handle incoming UDP packets
00136         try:
00137             data, addr = udp_sock.recvfrom(MAX_UDP_SIZE)
00138         except BlockingIOError:
00139             time.sleep(0.01)
00140             continue
00141 
00142         try:
00143             text = data.decode("utf-8").strip()
00144         except UnicodeDecodeError:
00145             continue
00146 
00147         if not text:
00148             continue
00149 
00150         parts = text.split()
00151         cmd = parts[0]
00152 
00153         # Handle incoming chat message
00154         if cmd == "MSG" and len(parts) >= 4:
00155             src, dest = parts[1], parts[2]
00156             msg = ' '.join(parts[3:])
00157             if dest == handle:
00158                 net2ui.send(("MSG", src, msg))
00159 
00160                 # Auto-reply if in AFK mode
00161                 if away and src not in afk_replied_to:
00162                     udp_sock.sendto(
00163                         f"MSG {handle} {src} {autoreply}".encode("utf-8"),
00164                         addr
00165                     )
00166                     afk_replied_to.add(src)
00167 
00168         # Handle incoming image transfer initiation
00169         elif cmd == "IMG" and len(parts) == 5:
00170             src, dest, tcp_port_s, size_s = parts[1], parts[2], parts[3], parts[4]
00171             if dest != handle:
00172                 continue
00173 
00174             tcp_port = int(tcp_port_s)
00175             size     = int(size_s)
00176 
00177             # Connect to sender's temporary TCP server and download image
00178             client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
00179             client.connect((addr[0], tcp_port))
00180 
00181             buf = b""
00182             while len(buf) < size:
00183                 chunk = client.recv(4096)
00184                 if not chunk:
00185                     break
00186                 buf += chunk
00187             client.close()
00188 
00189             # Save image to file and notify UI
00190             fn = os.path.join(img_path, f"{src}_{int(time.time())}.png")
00191             with open(fn, "wb") as f:
00192                 f.write(buf)
00193             net2ui.send(("IMG", src, fn))
00194 
00195         # Handle LEAVE notifications
00196         elif cmd == "LEAVE" and len(parts) == 2:
00197             leaver = parts[1]
00198             net2ui.send(("LEAVE", leaver, ""))
00199             peers[:] = [p for p in peers if p[0] != leaver]
00200 
00201         # Handle KNOWUSERS message to update peer list
00202         elif cmd == "KNOWUSERS":
00203             rest = text[len("KNOWUSERS "):]
00204             for chunk in rest.split(','):
00205                 if not chunk.strip():
00206                     continue
00207                 try:
00208                     h, ip, pt = chunk.strip().split()
00209                     entry = (h, ip, int(pt))
00210                     if h != handle and entry not in peers:
00211                         peers.append(entry)
00212                         print(f"[KNOWUSERS] New peer: {entry}")
00213                 except ValueError:
00214                     continue
00215 
```

```
00216         time.sleep(0.01)
```

8.13 README.md File Reference

Index