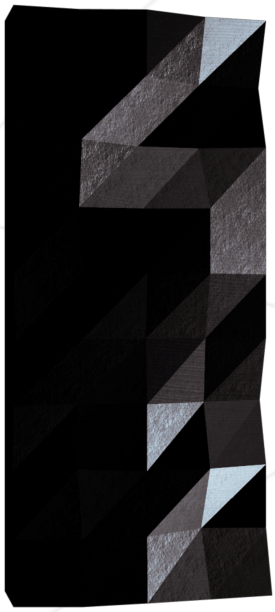

TAP

A shared-world retro text adventure

Summary: Build a small multiplayer text adventure with a TCP server and two clients: one CLI, one GUI. Follow the attached protocol. This is a group project (2 learners).

Version: 1.1

Contents

Chapter I

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter II

Introduction

Before loot boxes and battle passes, adventurers connected to shared worlds made entirely of text. These games were **MUDs** — Multi-User Dungeons — and they ran on university networks and hobbyist servers, often consuming more learner hours than lectures.

In **The Answer Protocol**, your group will create your own small, persistent-feeling world where multiple players can explore rooms, chat, and cooperate in real time. Your server will speak a simple, line-based TCP protocol, and your two clients — one command-line and one graphical — will bring that world to life.

Many gameplay elements are intentionally left to your group's discretion in the server implementation, including combat mechanics, quest systems, and *NPC* behaviors. The attached RFC provides basic command structures but deliberately leaves implementation details unspecified, requiring your group to make design decisions and document them with clear justification. However, *CLI* and *GUI* clients must be interchangeable between different groups working on this project in parallel.

This is a **group project** designed for exactly 2 learners working collaboratively. Consider dividing responsibilities: one person focusing on the server implementation and CLI client, the other on the GUI client and world design.

In 1990, a popular MUD was so addictive that administrators scheduled "MUD blackout hours" to save the campus network. Your job: make something addictive - but maybe not that addictive.

Chapter III

Overview

Your group will deliver:

- A **server** implementing the complete command set from the attached protocol.
- A **CLI client** for text-based interaction.
- A **GUI client** (toolkit of your choice) for a richer interface.
- Static **world data** (YAML or JSON) for rooms, items, and *NPCs*.
- A **building tool** appropriate for your chosen language (e.g., Makefile for C++, Cargo for Rust, Go modules, Maven/Gradle for Java, etc.) with targets/commands to install dependencies, compile, run, lint, and clean. The specific build system and its usage must be documented in your README.

The protocol document (attached) defines:

- All commands (CONNECT, LOOK, MOVE, CHAT, TAKE, DROP, INVENTORY, TALK, ATTACK, STATUS, QUEST, QUESTS, WHO, GROUP, QUIT), arguments, and expected responses.
- The event formats the server must push to clients.
- Error codes and reply structures.

Your implementation must support **every command and event** defined in that document.

`The protocol is formally specified as an RFC-style document (RFC 42TAP) providing precise technical specifications, ABNF syntax definitions, and comprehensive implementation guidelines following Internet standards conventions.`

`The architecture is up to you - use a dispatcher/router or inline handling, as long as the result is correct and maintainable. GUI toolkit choice is entirely free.`

Chapter IV

Global rules

- Language: **C, C++, Rust, Go, or Zig only**. Python is explicitly forbidden for this activity.
- Use appropriate libraries for your chosen language; networking and GUI libraries are permitted.
- Encoding & framing: TCP sockets, UTF-8, one message per line (`\n` terminated).
- **Protocol Compliance:** Your implementation must strictly adhere to RFC 42TAP specifications, including proper ABNF syntax, error codes, and message formats.
- Code must follow your language's standard linting practices and include appropriate type annotations where supported.
- The server must handle disconnects gracefully and remove player state before broadcasting leave events.
- Both clients (CLI and GUI) must remain responsive while receiving asynchronous events.
- No persistence is required; state may reset when the server restarts.

Python is strictly forbidden for this activity. Only C, C++, Rust, Go, or Zig are allowed. Using Python will result in an automatic failure of the peer review. This restriction ensures you work with systems programming languages suitable for network server development and gain experience with lower-level networking concepts.

Any deviation from the protocol must be clearly documented and explained in your README.

Chapter V

Mandatory part

V.1 Server

- Implements all commands and events from the attached protocol.
- Loads static world data from YAML or JSON files.
- Validates that all exits and references in the world are correct.
- Handles malformed commands with protocol-compliant error responses.
- Broadcasts messages without interruption if a client disconnects mid-send.
- **Dynamic Item Management:** implements a proper item system where:
  - Items exist as unique instances in the world
  - Taking an item removes it from the room (no duplication)
  - Dropping an item makes it available to other players
  - Items can be referenced by ID or display name
  - Multi-word item names are fully supported
- **World:** provide a coherent map with the following requirements:
  - **Rooms:** at least 8 interconnected rooms forming one or more *loops*, plus at least one optional branch.
  - **NPCs:** at least 3 distinct *NPC* roles (e.g., dialogue, quest-giver, enemy).
  - **Items:** at least 4 distinct items; at least 2 must be obtainable in-world.
  - **Quests:** at least 2 simple quests (e.g., fetch item, defeat *NPC*, deliver item).
  - **Design Choice:** Your group must design quest progression mechanics, completion validation, and reward systems. Document your implementation approach in the README.
  - **Exploration:** movement must allow a full circuit (no “line-only” maps).
  - **Example format:** see the minimal YAML below. It is intentionally tiny and *does not* meet the required world size; it only illustrates structure.

- **Combat System:** implement basic combat mechanics with the following requirements:
  - Players start with 100 health points (HP) and can be reduced through combat
  - Enemy *NPCs* have varying health points based on their type and difficulty
  - ATTACK command deals damage to enemy *NPCs* and may trigger counter-attacks
  - STATUS command shows current player health and combat status
  - Players with 0 HP should respawn at a safe location with reduced health
  - Combat results must be logged and broadcast to relevant players
  - **Design Choice:** Your group must design and implement turn-based combat mechanics, including damage formulas, initiative order, and additional combat commands (DEFEND, FLEE, etc.). Document and justify your choices in the README.
- **Server Logging:** implement comprehensive logging to monitor server behavior with the following mandatory requirements:
  - Log all client connections and disconnections with timestamps and IP addresses
  - Log every command received from clients with player name and parameters
  - Log all server responses and error codes sent to clients
  - Log world state changes (item movements, *NPC* interactions, combat results)
  - Log quest progress and completion events
  - Use structured logging format (JSON recommended) for easy parsing
  - Include log levels (INFO, WARN, ERROR) for different event types
  - Monitor and log potential abuse patterns (command flooding, rapid connections)
  - All logs must include precise timestamps and be written to appropriate output streams
  - Logging must not significantly impact server performance or responsiveness

V.2 Example World Data (Minimal)

Below is a **tiny, non-compliant** YAML example showing only the basic structure. It is purely illustrative — your real world must be much larger and meet the mandatory requirements.

Listing V.1: Minimal World Example (YAML)

```yaml
world:
  locations:
    start:
      name: "Village Square"
      description: "A bustling square with cobblestone paths."
      exits:
        north: "tavern"
        east: "shop"
      spawns:
        - npc_type: "guard"
          count: 1
    tavern:
      name: "The Prancing Pony"
      description: "A cozy tavern filled with warmth and laughter."
      exits:
        south: "start"
      items: ["ale"]
    shop:
      name: "General Store"
      description: "Shelves lined with various goods and supplies."
      exits:
        west: "start"
      spawns:
        - npc_type: "merchant"
          count: 1

items:
  ale:
    name: "Frothy Ale"
    description: "A refreshing drink that restores energy."
    obtainable: true

npcs:
  guard:
    name: "Village Guard"
    description: "A stern-looking guard watching over the square."
    dialogue: ["Stay safe, traveler.", "The roads can be dangerous."]
    stats:
      hp: 20
  merchant:
    name: "Shop Keeper"
    description: "A friendly merchant eager to trade."
    dialogue: ["Welcome to my shop!", "What can I get for you today?"]
    stats:
      hp: 15
```

Remember: This example is far too small for your submission. Your world must meet the mandatory size requirements listed above.

`The YAML structure, keywords, and organization shown above are not imposed. Each group can design their own data format and structure according to their implementation needs.`

V.3 CLI Client

- Connects to the server and displays incoming messages in real time.
- **Command Interface Choice:** Your group can choose between two approaches: (1) sending user input commands directly to the server using RFC protocol syntax, or (2) implementing a more user-friendly CLI interface that translates user-friendly commands into proper RFC protocol packets behind the scenes. Document your choice and implementation approach in the README.
- Keeps receiving events while waiting for user input.

V.4 GUI Client

- Any graphical toolkit is allowed (`Qt`, `GTK+`, `wxWidgets`, web-based, etc.). Note that `curses` is considered a text-based interface, not a true GUI.
- Displays room details, items, NPCs, and exits with real-time updates.
- Shows player inventory (INVENTORY command) and provides buttons for item management (TAKE, DROP).
- Handles both item IDs and display names for user convenience.
- Updates room view automatically after TAKE/DROP operations to reflect item availability.
- Separates the chat view (*Global*, *Room*, *Group*) from the log view.
- Provides buttons for available actions (LOOK, MOVE, TAKE, DROP, TALK, ATTACK, STATUS, QUEST, QUESTS, WHO, GROUP, QUIT).
- Shows counters for players in the room and on the server.
- Enables NPC interaction through TALK command with dialogue display.

V.5 Example interactions

Basic connection and chat

```
S: OK hello proto=1
C: CONNECT alice
S: OK connected
C: LOOK
S: OK { "room": { ... }, "players":["alice"], "items":[], "npcs":[] }
C: CHAT GLOBAL Hello everyone
S: OK
S: EVT GLOBAL CHAT alice Hello everyone
```

Movement and presence events

```
C: MOVE north
S: OK room=loc.bakery
S(to old room): EVT ROOM PRESENCE LEAVE alice
S(to new room): EVT ROOM PRESENCE ENTER alice
```

Item management and dynamic state

```
C: LOOK
S: OK { "room": { ... }, "items":["item.herbs"], ... }
C: TAKE Herbs
S: OK taken=item.herbs
C: LOOK
S: OK { "room": { ... }, "items":[], ... }
C: DROP item.herbs
S: OK dropped=item.herbs
C: LOOK
S: OK { "room": { ... }, "items":["item.herbs"], ... }
```

NPC interaction and inventory

```
C: TALK guard
S: OK { "npc": "guard", "dialogue": "Stay safe, traveler." }
C: INVENTORY
S: OK ["item.herbs", "item.bread"]
C: WHO
S: OK { "room": ["alice", "bob"], "server": 5 }
```

Chapter VI

Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➠ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.
For this project, the following additional sections are required in your README.md:

- A "**Architecture**" section explaining your server design choices (dispatcher/router vs inline handling, concurrency model, etc.).
- A "**Protocol Implementation**" section documenting any deviations from RFC 42TAP and justifying your choices.
- A "**Combat System**" section describing your turn-based combat mechanics, damage formulas, initiative order, and additional combat commands (DEFEND, FLEE, etc.).
- A "**Quest System**" section explaining your quest progression mechanics, completion validation, and reward systems.

- A "**World Design**" section describing your world layout, room connections, NPC roles, and item distribution.
- A "**Server Logging**" section documenting your logging implementation, including log format, event types, output destinations, and how to monitor server behavior and detect abuse patterns.
- A "**Group Contributions**" section clearly indicating each team member's responsibilities and contributions to different components (server, CLI client, GUI client, world design, etc.).
- A "**Building and Running**" section with detailed instructions for your chosen building tool and how to run each component (server, CLI client, GUI client).
- A "**Testing**" section explaining how to test the multiplayer functionality, combat system, and quest mechanics.

`Your README must be written in English.`

Chapter VII

Submission and peer-evaluation

VII.1 Repository content

- Source files for the server, CLI client, and GUI client.
- Data files for the game world (YAML or JSON).
- A building tool (e.g., `Makefile`, `Cargo.toml` with appropriate scripts, `build.gradle`, etc.) with targets/commands for: install dependencies, run-server, run-client, run-client-gui, lint, clean. The specific build system must match your chosen language.
- A `README.md` explaining how to run each component, architectural choices, and group member contributions.

VII.2 Group work guidelines

- **Team size:** Exactly 2 learners.
- **Collaboration:** Both group members must contribute meaningfully to the codebase.
- **Git practices:** Use proper commit messages and branch management. Each member should have commits in the repository.
- **Documentation:** The README must clearly indicate each member's contributions and responsibilities.
- **Suggested division:** One learner handles server implementation and CLI client, the other handles GUI client and world design, but adapt to your group's strengths.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behavior change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific timeframe is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.