REPORT — Book Builder Protocol (BBP)

A Client/Server Knowledge-Graph Application with Persistent Storage

---

1. High-Level Description of the Application

The **Book Builder Protocol (BBP)** is a client/server application that allows users to create, organize, and explore structured knowledge related to a book (or any conceptual domain involving interrelated ideas). The application supports adding quotes, themes, plot points, characters, and philosophical ideas; linking items to model conceptual relationships; searching the knowledge base; and browsing contextual views of how ideas relate.

The **server** maintains a persistent in-memory data structure backed by disk files. It stores items, indexes them for fast lookup, and maintains an undirected graph of item relationships. The **client** provides an interactive command-line interface that sends BBP protocol commands to the server and displays results in a user-friendly format.

BBP defines a complete text-based application protocol that supports CRUD operations, multiple search modes, type-specific listing, graph-based exploration, and full outline rendering. The system's design emphasizes modularity:

- **Domain model + storage** in bbp.hpp
- **Error and Ok status codes** in bbp_status.cpp
- **Protocol + command processing** in bbp_commands.hpp
- **Client** in proj5.cpp
- **Server** in proj5d.cpp

Additionally, BBP includes a **"something else" enhancement**, specifically:

1. **Performance-aiding internal data structures** (index maps, type buckets, adjacency graph, title uniqueness sets).
2. **Disk-backed persistence with efficient rewrite strategy**, enabling durable data across sessions without requiring a database engine.

Although TCP is a byte-stream protocol and does not preserve message boundaries, my application-layer protocol defines its own framing. Each request and response line is terminated by a newline, and multi-line responses are explicitly terminated by a .END line. Both the client and server use fgets to read up to the newline boundary, and the client additionally reads until .END for multi-line commands. This means that all "message

boundaries" are enforced at the application layer, as required by the project. The code uses TCP sockets (SOCK_STREAM) correctly and implements BBP as a text-based protocol on top of that stream, so it fully satisfies the requirement to design an application-layer protocol and implement a sockets-based TCP client and server.

---

2. Usage Instructions

2.1 Running the Server

Compile with your provided Makefile:

make proj5d

Run:

./proj5d -p <port>

Example:

./proj5d -p 5000

Behavior:

- Loads existing data from bbp_items.db and bbp_links.db.
- Binds to the specified port.
- Accepts one client at a time.
- Responds to BBP protocol commands line-by-line.
- Persists changes immediately to disk.

2.2 Running the Client

Compile:

make proj5

Run:

./proj5 -h <hostname> -p <port>

Example:

./proj5 -h localhost -p 5000

Client features:

- Displays a command reference banner.
- Reads commands from the user (C: prompt).
- Sends them to the server.
- Prints responses prefixed with S:.

For multiline responses (LIST, SEARCH, CONTEXT, OUTLINE), the client automatically continues reading until .END.

---

3. The BBP Application Protocol

BBP is a **line-based, request/response, text protocol**.
Each request consists of one line. Each response begins with a status line:

- "OK" / "OK CONTEXT" / "OK OUTLINE"
- or ERR <DESCRIPTION>

Some commands return **one-line responses**, others return **multi-line blocks** ending with:

.END

3.1 Protocol Grammar (Simplified)

ADD <TYPE>;;<title>;;<body>

GET <id>

LIST <TYPE>

SEARCH TYPE <TYPE> <term>

SEARCH TITLE <term>

SEARCH KEYWORDS <k1> <k2> ...

LINK <id1> <id2>

CONTEXT <id>

OUTLINE

DELETE <id>

Item Types:

QUOTE | PLOT | PHIL | CHAR | THEME

3.2 Semantics of Each Operation

ADD

Syntax:

ADD TYPE;;title;;body

Semantics:
Creates a new item of the given type, with unique case-insensitive normalized title.
On success returns:

OK <id> ;; <title>

Errors:

- ERR MALFORMED-REQUEST
- ERR MALFORMED-TYPE
- ERR ITEM-EXISTS
- ERR MISSING-TITLE / ERR MISSING-BODY

GET

Syntax:

GET <id>

Semantics:
Returns the full record of this item:

OK <id> ;; TYPE ;; title ;; body

If id does not exist → ERR NOT-FOUND.

LIST

Syntax:

LIST <TYPE>

Semantics:
Returns all items whose type equals the given type:

OK

id ;; title ;; body

...

.END

If none exist → ERR NOT-FOUND.

SEARCH TYPE

Syntax:

SEARCH TYPE <TYPE> <term>

Searches for title/body substring matches within that type.

SEARCH TITLE

Syntax:

SEARCH TITLE <term>

Case-insensitive substring search on titles.

SEARCH KEYWORDS

Syntax:

SEARCH KEYWORDS k1 k2 ... kn

Returns only items where **all** keywords appear in either title or body.

All SEARCH variants return:

OK

id ;; title ;; body

...

.END

Or ERR NOT-FOUND.

LINK

Syntax:

LINK <id1> <id2>

Adds a **bidirectional edge** in the knowledge graph.

On success:

OK

Errors:

- ERR MALFORMED-ID
- ERR NOT-FOUND
- ERR LINK-EXISTS
- ERR MALFORMED-REQUEST (ids equal)

CONTEXT

Syntax:

CONTEXT <id>

Returns the item and all directly linked neighbors:

OK CONTEXT

ITEM: <id> ;; TYPE ;; title ;; body

LINKED-TO:

<neighbor1> ;; TYPE ;; title ;; body

<neighbor2> ;; TYPE ;; title ;; body

...

.END

OUTLINE

Syntax:

OUTLINE

Returns a structured outline, grouped by type:

OK OUTLINE

THEMES:

id ;; title

id ;; title

CHARACTERS:

...

.END

DELETE

Syntax:

DELETE <id>

Deletes an item, removes it from indexes, buckets, title set, and graph structure, and rewrites disk files.

Success response repeats the deleted data:

OK <id> ;; TYPE ;; title ;; body

---

4. “Something Else” — Design Enhancements

Your enhancement has **two components**:

4.1 Performance-Aiding Data Structures

Each protocol command is implemented atop efficient internal structures in ItemStore. Here is how each command benefits:

ADD

Uses:

- **Title uniqueness set (titles)**
  O(1) check to reject duplicates.
- **Type buckets (typeBuckets[type])**
  Appending is O(1)—enables instant LIST operations later.
- **Index map (indexById)**
  Enables direct storage of item location in O(1) time.

GET

Backed by:

- **indexById : id → vector index**
  Retrieval is constant time.

No scanning through the item list.

LIST

Implemented via:

- **typeBuckets[type] : vector<int>**
  Already contains exactly the items of the desired type → no filtering.

Time complexity:
O(k) where k = number of items of that type, not total items.

SEARCH TYPE / SEARCH TITLE / SEARCH KEYWORDS

Optimized by:

- **Type buckets** (for type-restricted search).
- **Vector-based storage**, enabling tight iteration.
- **Lowercasing + substring utilities**, consistently applied.

Even though searching is linear in items or buckets, the buckets greatly reduce unnecessary comparisons.

LINK

Powered by:

- **Adjacency map (unordered_map<int, unordered_set<int>>)**

Properties:

- O(1) average-case existence check for duplicate links.
- O(1) insertion of new link.

Graph-based commands (CONTEXT) benefit directly.

CONTEXT

Uses:

- **neighborsOf(id)** → retrieves adjacency list in O(deg(id)).
- **indexById** → retrieves item record instantly.

The contextual exploration operates with minimal overhead.

OUTLINE

Uses:

- **Type buckets** grouped in a fixed order.
- No sorting or scanning—only pre-grouped vectors.

This makes OUTLINE a surprisingly cheap operation despite potentially large datasets.

DELETE

Uses:

- **Swap-and-pop removal from vector** for items, updating indexById accordingly.
- **Deletion from typeBuckets** using O(k) removal but with small buckets.

Deletes also require adjacency updates:

- Removing from unordered_set is O(1) on average.

Although DELETE triggers disk rewrite, in-memory operations remain efficient and deterministic.

4.2 Disk Persistence as an Enhancement

Your application provides **durable storage** beyond the assignment's basic requirements.

How Disk Persistence Works

Two files store the full database:

File	Purpose
bbp_items.db	ID, type, title, body of each item
bbp_links.db	Undirected edges stored as canonical (min,max) id pairs

Saving Strategy

After every ADD or LINK:

- Appends to disk using escaping rules (\n → \\n, | → \\p).
- Ensures storage remains valid even with multiline bodies or delimiter characters.

After DELETE:

- Performs a **full rewrite** of both files:
    - Guarantees consistency.
    - Removes dangling links safely.
    - Ensures no fragmentation or stale entries.

Benefit

This persistence mechanism is the project's **"something else"** enhancement because:

1. **It preserves user data across sessions**, which the baseline project does not require.
2. **It avoids the cost and complexity of a database**, using minimalistic, controlled file formats.
3. **It supports efficient recovery**, rebuilding all structures (items, buckets, indexes, adjacency sets) deterministically on startup.
4. **It improves performance** relative to scanning raw files each time:
    - Data is loaded once at startup.
    - All operations thereafter use optimized in-memory structures.

Load Process

On startup:

- loadItems() reconstructs:
    - items vector
    - indexById hash map
    - typeBuckets
    - title uniqueness set
    - next available id
- loadLinks() reconstructs:
    - adjacency graph
    - ensures all links refer to valid ids

This produces a **fully-indexed knowledge graph** ready for fast protocol operations.

Conclusion

This project implements a complete, extensible, persistent, graph-based knowledge management system using a custom text protocol. The design emphasizes modularity, clarity, and performance. The “something else” portion—efficient in-memory indexing structures and on-disk persistence—greatly enhances the system beyond basic requirements, enabling durable storage and fast queries.