Parking System Project Design Document (V2.0)

1. Introduction

This document outlines the design and implementation details for a comprehensive parking system project. The project is divided into three main parts: a TCP Server capable of handling multiple clients and processing parking data, a Database (DB) for storing pricing information and customer data, and a system integrating STM32 microcontroller and BeagleBone Green (BBG) for emulating GPS coordinates and transmitting them to the server.

2. Part 1: The TCP Server

Objective

Develop a TCP server to support multiple clients, processing incoming GPS coordinates and unique identifiers, and calculating parking fees based on elapsed time and city-specific pricing.

Design and Implementation

Server Architecture:
The server will operate asynchronously to handle multiple client connections simultaneously, processing start and end messages to calculate parking durations.

Data Handling:
Upon receiving a connection, the server will parse the incoming data for GPS coordinates and identifiers, distinguishing between start and end messages to compute the elapsed time.

Good luck

Pricing Calculation:
The server will access a separate pricing file, loading city-specific rates into shared memory for quick retrieval. After calculating the parking duration, it will determine the total fee based on these rates.

Shared Memory Integration:
Use shared memory to exchange pricing information between the server and the DB, ensuring that price updates are immediately accessible.

Configuration

All configurable parameters, such as port number and shared memory size, will be specified in an external CONFIG file to allow for easy adjustments without code modification.

3. Part 2: Database (DB)

Objective

To create a robust database system that stores pricing information and customer parking data. This database will allow for dynamic updates, including the addition or removal of prices.

Design and Implementation

Database Schema

Two tables will be created within the database:

1. Prices Table: Stores city-specific pricing information.
2. Customer Data Table: Records details of parking transactions, including GPS coordinates, customer identifiers, and calculated parking fees.

Good luck

SQLite Database
The system will leverage SQLite for its database needs due to SQLite's lightweight, file-based characteristics, making it suitable for the project's requirements.

Shared Memory Communication
The database will regularly synchronize with shared memory to fetch the latest parking transaction data and pricing updates, ensuring that the most current information is always stored.

Updating Prices

Dynamic Price Modification:
The database must support the dynamic updating, adding, and removing of pricing information. This functionality will be achieved through a specialized mechanism that writes updates to a file. A specific utility or software package, designed as part of the project, will handle these updates.

File-based Updates:
Modifications to the prices will be conducted by writing changes to a designated file. This approach allows for a clear audit trail of price adjustments and a straightforward method for updating the database without direct interaction.

Synchronization with Shared Memory:
After updating the prices file, the changes will be synchronized to shared memory. This ensures that the pricing information used by the TCP server for calculating parking fees is up to date.

Implementation Mechanism:
A detailed specification for the file format and the procedure for updating the database will be included in the project's technical documentation. This will ensure clarity and consistency in how price updates are managed.

Signaling for Update:
Once the price update file has been modified, a signal will be sent to the database to prompt it to read the updated prices from the file. This ensures that the database is always operating with the most current pricing

Good luck

information, and it allows for the updates to be immediately reflected in the parking system's pricing calculations. The technical implementation will include specifying the signal mechanism, ensuring it is reliable and efficient for the system's needs.

4. Part 3: STM32 and BBG System

Objective

Implement a system where the STM32 microcontroller emulates a GPS module, sending data to the BBG, which then communicates with the TCP server over Ethernet.

Design and Implementation

STM32 Functionality:
The STM32 will periodically send GPS coordinates along with unique identifiers and start/end messages to the BBG using the I2C protocol.

BBG Processing:

- The BBG will host two processes:
  - Ethernet Communication: Handles data transmission to the TCP server.
  - I2C Communication: Receives data from the STM32
  - passes it to the Ethernet communication process via IPC PIPE.
- Daemon: Ensure the system's daemon binary is loaded at boot, eliminating the need for manual program starts.

Communication Protocols

I2C:
Used for communication between STM32 and BBG, chosen for its simplicity and efficiency.

Good luck

Ethernet:
Facilitates the transmission of data from BBG to the TCP server, ensuring reliable communication over network infrastructure.

5. Development Standards

Git Usage:
A Git repository will be initialized at the beginning of the project. The repository link will be shared via email to facilitate tracking of development progress and collaboration. It is essential to use clear, concise commit messages and to structure commits logically to reflect the development process.

Documentation:
All code will be thoroughly documented following DOXYGEN standards. This includes comprehensive comments within the codebase to describe the functionality of code segments, input parameters, return values, and any side effects. High-level documentation should provide an overview of the system architecture, data flow, and interaction between components.

Code Efficiency and Cleanliness:
The code should be written with efficiency in mind, avoiding redundancy and ensuring optimal performance. Dead code or code snippets that are not in use should be removed to maintain the cleanliness of the codebase. Functions and classes should be logically organized and focused on a single responsibility to enhance readability and maintainability.

Synchronization and Concurrency Control:
Given the multi-threaded nature of the system and the critical requirement to manage shared resources effectively, appropriate synchronization mechanisms must be implemented. Where necessary, protection mechanisms such as mutexes (for exclusive access control) or semaphores (for limiting concurrent access to a resource) should be added to avoid race conditions. These mechanisms are vital in the sections of code where shared resources, like shared memory segments or file access for updating prices, are accessed or modified by multiple threads or processes concurrently.

Good luck

- Mutexes should be used when mutual exclusion of resource access is required, ensuring that only one thread can access the resource at a time.
- Semaphores can be used for controlling access to a resource pool or for synchronization purposes between threads or processes.
- The choice between mutexes and semaphores should be made based on the specific concurrency control needs of each part of the project. Proper implementation will prevent data corruption and ensure the system's reliability and stability.

Error Handling:
Robust error handling mechanisms must be implemented to ensure the system can gracefully recover from unexpected states or input. This includes checking the return values of functions, especially those that interact with external systems or hardware, and taking appropriate action based on the error conditions.

Logging:
Implement detailed logging for critical system events, errors, and operations. Log files should be structured to provide clear, timestamped entries that detail the context and nature of each logged event. This aids in debugging and monitoring the system's health in production.

6. Logging and Monitoring

- Logging: Each part of the project will generate its own log file, capturing critical operations and errors to aid in debugging and monitoring.
- Log File Management: Log files will be structured to provide clear, timestamped entries detailing system operations without overwhelming detail.

7. Auxiliary Files

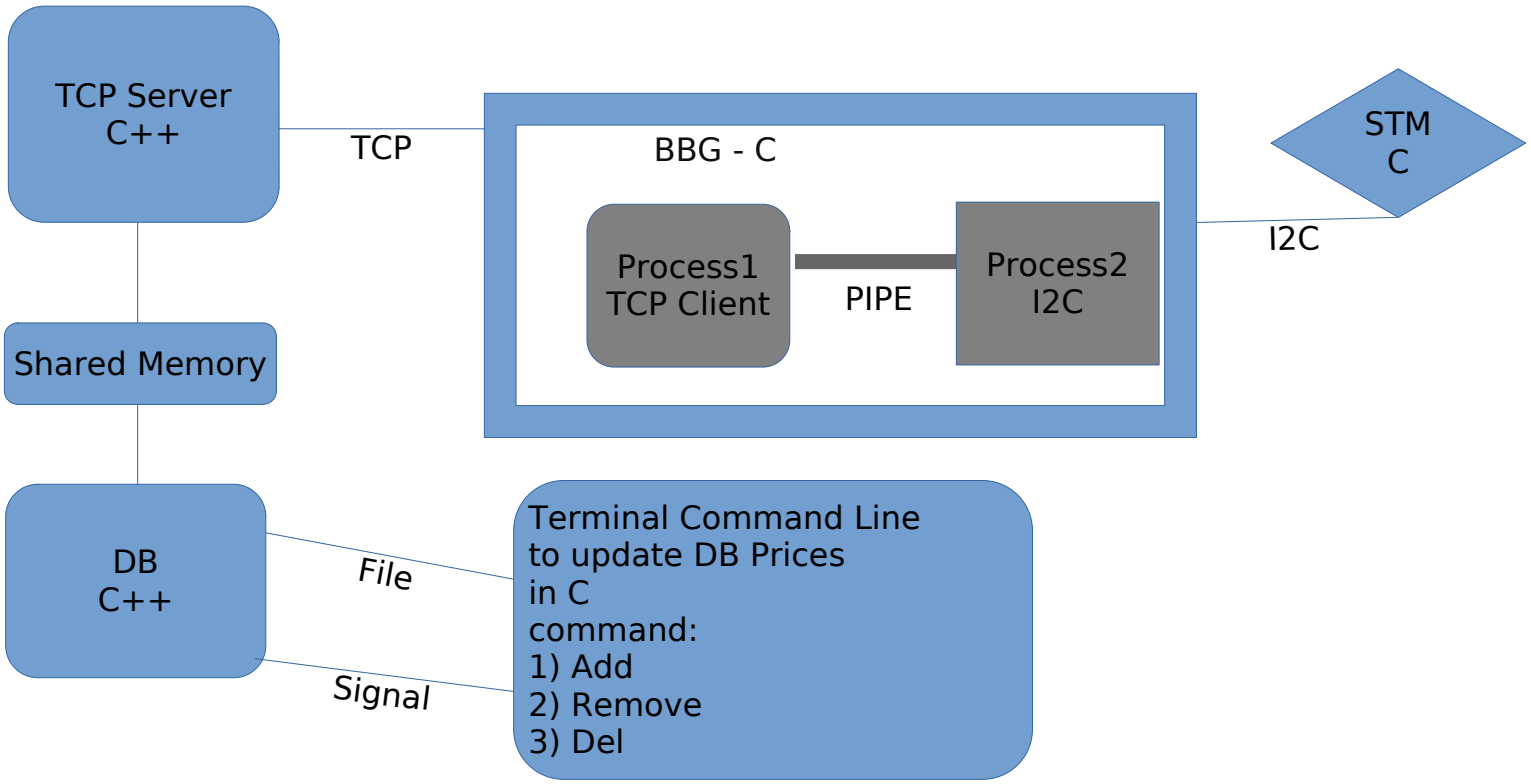
- BASH Scripts: Scripts will be provided to facilitate the starting of parts 1 and 2, ensuring seamless operation and integration.
- MAKEFILEs: Each part of the project will include a MAKEFILE to automate the compilation process, enhancing the development workflow.

Good luck

8. Conclusion

This document has outlined the design and implementation approach for a complex parking system project, covering server-side processing, database management, and hardware integration. By adhering to the specified requirements and utilizing the proposed technologies and protocols, the project aims to deliver a reliable and efficient solution for parking management.

9. Schema



Contact Information

For any inquiries or further assistance, please contact:

Abed - abed@rt-ed.co.il
Eded – eden@rt-ed.co.il

Good luck