Specification Documentation for Communication Layer Between Tasks

This document specifies the design and functionality of a communication layer between tasks for byte-based data transfer. This layer supports inter-process communication, which can occur on the same machine or across different machines.

The communication mechanism uses **byte-oriented circular buffers**, which provides a memory-conscious solution for task communication.

Implemented in Java, this layer functions as a distributed system, facilitating seamless communication across various environments.

1. System Overview

The communication layer facilitates the exchange of bytes between independent tasks via **channels**, which are managed by a **broker**.

These channels can be used to establish a **bidirectional communication** flow between two tasks. This layer allows tasks to communicate without regard to whether they are on the same or different machines.

2. Functional Requirements

This communication layer provides the following core functionalities:

Functional Requirement	Description
Channel Management	Tasks can create or connect to channels using the broker. Channels provide a medium for transmitting byte sequences between tasks.
Efficient Data Transfer	Channels utilize circular buffers to handle byte streams efficiently, with minimal memory overhead.
Concurrency	The system allows multiple tasks to communicate simultaneously. The broker can manage multiple channels, enabling concurrent communication between different tasks without interference or data loss.
Error Handling	Robust error handling, including management of disconnections and buffer overflows.
Safe Disconnection	Supports safe disconnection of the system without data corruption or loss.

3. Components

3.1. Broker

The **Broker** class is a service responsible for managing channels between tasks in a distributed system. A task can request a broker to create a new channel or to connect to an existing one.

Broker Properties and Behavior:

- The broker maintains a list of active channels and allows tasks to either create new channels or connect to existing ones, facilitating communication across different brokers (e.g., on different machines).
- The system supports multiple brokers (one per entity, i.e. client/server) allowing tasks to communicate across different brokers.

The broker interface provides the following key methods:

Method	Description
`Channel accept(int port)`	Listens on the specified port and accepts incoming channel requests. Returns a `Channel` object, which the task can use to send and receive data. The server side will call this method to establish a channel.
`Channel connect(String host, int port)`	Allows a client task to connect to a channel by specifying the server's name and port number. Returns a `Channel` object that the client will use to communicate with the server task. This method is used by clients to initiate communication.

Key considerations

- Each broker should have its **unique name** and **port number**, which can be used to identify and connect to the broker. e.g., we can have :
  - `Broker server = new Broker("server", 8080);`
  - `Broker client = new Broker("client", 8081);`
- The broker is **multi-threaded**, meaning that it can handle multiple tasks concurrently.

3.2. Channel

The **Channel** service provides a medium through which tasks communicate. A channel abstracts the underlying circular buffer and provides methods for reading from and writing to the circular buffer.

Channel Properties and Behavior:

Property	Description
FIFO Lossless	Data is transmitted in the same order it was sent, with no data loss due to the use of TCP connections.
Bidirectional	A channel allows simultaneous two-way communication between tasks, supporting full-duplex operations.

Property	Description
Byte-Oriented	The communication is based on raw byte streams, providing precise control over how data is packed, transmitted, and interpreted. It allows tasks to transmit and receive data incrementally without needing to buffer large chunks of structured data at once.

Channel Interface Methods

The channel interface provides the following key methods:

Method	Description	Return Values	Blocking Operations
`int read(byte[] bytes, int offset, int length)`	Reads bytes from the circular buffer into the provided byte array. Starts reading from the given offset until the specified length.	Number of bytes read, -1 if disconnected.	Yes, waits until data is available to read.
`int write(byte[] bytes, int offset, int length)`	Writes bytes from the provided byte array into the circular buffer, starting from the given offset and writing up to the specified length.	Number of bytes written, -1 if disconnected.	Yes, waits until there is space in the buffer.
`void disconnect()`	Disconnects the channel, preventing further communication.	null	No
`boolean disconnected()`	Checks if the channel has been disconnected, indicating no further data transmission is possible.	`true` if disconnected, `false` otherwise.	No

Note: Read and write operations are ***mutually exclusive***. This means that only one operation (read or write) can happen at a time per channel. Tasks must manage the synchronization of these operations.

3.3. Task

A **Task** is an independent unit of execution that communicates through channels. Tasks are represented as threads, allowing concurrent communication between multiple tasks (we can have n tasks running on 1 broker).

Task Key Methods:

The task interface provides the following key methods:

Method	Description
`Task(Broker b, Runnable r)`	A constructor that initializes a task with a specific `Broker` to manage its communication channels and a `Runnable` to define the work that the thread should execute.

Method	Description
`static Broker getBroker()`	Allows a task to retrieve the `Broker` it uses for managing channels.

3.4. Circular Buffer

The **Circular Buffer** is used within the channels to store byte sequences in a **FIFO** manner.

Circular Buffer Key Methods:

The circular buffer provides the following key methods:

Method	Description
`CircularBuffer(byte[] r)`	Initializes the circular buffer with the provided byte array.
`boolean full()`	Returns `true` if the buffer is full, indicating that no more data can be written.
`boolean empty()`	Returns `true` if the buffer is empty, indicating that no data is available for reading.
`void push(byte b)`	Adds a new element to the buffer. Throws an `IllegalStateException` if the buffer is full.
`byte pull()`	Retrieves and removes the next available byte. Returns the byte pulled from the buffer. Throws an `IllegalStateException` if the buffer is empty.

4. Multi-threading Considerations

Broker Class: The Broker is designed to handle multiple tasks concurrently, meaning it is thread-safe.

Channel Class: The Channel is not multi-threaded. It is up to the tasks to manage synchronization when performing read and write operations.

Task Class: Tasks can run in parallel using threads, but care must be taken when accessing shared resources or channels.

5. Asynchronous Operations (read and write)

Method	Asynchronous Behavior
`connect(String host, int port)`	- Initiates a **non-blocking** connection request to the specified broker. - Allows the task to continue operations while the broker processes the connection in the background. - Returns a `Channel` object upon successful connection; can handle retries or notify of failure asynchronously.

Method	Asynchronous Behavior
`disconnect()`	- Starts a **non-blocking** disconnection process. - Marks the channel as "disconnected" and allows the task to proceed with other operations. - Ongoing read/write operations receive an indication (e.g., return -1) once disconnection is complete.

6. Limitations

Due to design choices, certain concerns and limitations should be considered when using this communication layer:

Limitation	Cause	Solution
Fixed Buffer Size	Circular buffer has a fixed size, which can lead to full or empty states.	Implement buffer size monitoring and handle full/empty states gracefully. Consider dynamic resizing if applicable.
Single-threaded Channels	Read and write operations are not concurrent, limiting throughput.	Use task-level synchronization to manage read/write operations or implement a more complex multi-threaded channel design.
Blocking Operations	`read` and `write` methods are blocking, causing tasks to wait for data availability or buffer space.	Use non-blocking I/O operations or introduce asynchronous methods to improve responsiveness.

7. Conclusion