The Journal of Open Source Software

Netcore: A Flexible Python Framework for Custom Protocol Communication

Yizhou Li [1]

1 Xiangxi Ethnic Middle School, Jishou, Hunan, China

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 24 March 2025
Published: unpublished

Summary

Communication protocols play a critical role in modern software systems. While many mature communication protocols and frameworks exist, custom protocols are often required in specific scenarios. Netcore is a lightweight communication framework implemented in Python, **with its core advantage lying in enabling concurrent message transmission over a single channel**. The framework abstracts away the complexities of protocol implementation, concurrent transmission, and message management by requiring developers to provide only basic I/O functions (send/receive). Through an innovative message chunking and scheduling mechanism, it allows simultaneous transmission and reception of multiple messages without establishing multiple connections, significantly improving the utilization efficiency of limited communication resources.

Statement of need

Developers face several challenges when implementing custom communication systems:

1. Implementing complex protocol details from scratch
2. Managing concurrent message transmission over a single connection
3. Handling message fragmentation and reassembly
4. Organizing code structure for large applications
5. Managing asynchronous operations and scheduled tasks

In domains such as embedded systems, industrial control, and IoT, communication ports (e.g., serial ports, Bluetooth connections) are often scarce and limited resources. Traditional approaches either use a single connection for serial communication or require establishing multiple connections for different services, both of which have significant limitations. **Netcore specifically addresses concurrent communication over a single channel**, filling a gap in existing frameworks.

Existing communication frameworks typically require developers to handle intricate protocol details or are restricted to specific protocols. Netcore resolves these issues through the following approaches:

- Only requires developers to provide basic I/O functions (send/receive)
- Handles all protocol implementation details internally
- Provides data serialization and transmission via `LsoProtocol`
- Abstracts transmission methods through `Pipe`, allowing custom communication implementations
- Supports modular application development via the Blueprint system
- Includes built-in event systems, task schedulers, and caching mechanisms

Architecture

Netcore adopts a three-layer architecture:

1. **Protocol Layer (`LsoProtocol`)**:
   - Implements packet serialization/deserialization
   - Supports message chunking (splitting large messages into smaller fragments)
   - Manages memory and file-based storage
   - **Provides foundational protocol support for concurrent communication over a single channel**
2. **Transport Layer (`Pipe`)**:
   - Requires developers to provide send/receive functions
   - Internally manages message handling
   - Implements message transmission
   - Provides error handling
   - Includes thread-safe mechanisms
   - **Implements chunked message scheduling, enabling multiple message streams to share the same channel**
3. **Application Layer (`Endpoint`)**:
   - Message routing and Blueprint support
   - Request-response management
   - Event system integration
   - Task scheduling and caching
   - Global request context and thread-local storage
   - Multi-threading support with worker thread pools
   - Thread-safe resource access
   - **Provides a transparent concurrent communication interface for applications, hiding underlying complexities**

Comparison with Existing Solutions

Many existing communication frameworks focus on specific protocols (HTTP, WebSocket, etc.) or require developers to implement low-level protocol details. Netcore distinguishes itself through several key features:

1. **Protocol Agnosticism**: Unlike protocol-specific frameworks such as Flask (HTTP) or `websockets` (WebSocket), Netcore works with any communication channel that provides basic I/O capabilities.

2. **Concurrent Transmission over a Single Connection**: This is Netcore's most unique feature. While most frameworks require multiple connections or threads for concurrent communication, Netcore enables multiple data streams over a single I/O connection—particularly useful in scenarios like serial communication where only one connection is allowed. Through its innovative message chunking and scheduling algorithm, multiple logical communication streams share the same physical connection, significantly improving resource utilization.

3. **Minimal Requirements**: Compared to frameworks like ZeroMQ or gRPC, which require specific libraries or implementation details, Netcore only demands that developers provide send/receive functions.

4. **Integrated Features**: Netcore offers a Blueprint system, event handling, and caching as core components of its design.

Multithreading Design

Netcore implements a multithreaded approach for request handling:

1. **Thread Pool**: The `Endpoint` class maintains a configurable worker thread pool to process incoming requests.
2. **Thread-Local Storage**: Request contexts are stored in thread-local storage, ensuring isolated request contexts for each worker thread.
3. **Thread-Safe Mechanisms**:
   - Shared resources are protected by locks
   - Message queues handle inter-thread communication
4. **Request Processing Pipeline**:

   ```
   Main thread: Receives requests → Enqueues
         ↓
   Worker threads: Dequeue → Process → Send responses
   ```
5. **Non-Blocking Operations**: The main thread continues receiving messages while worker threads process requests.

This design complements single-channel concurrent communication, allowing applications to receive new requests while processing current ones, thereby maximizing the use of limited communication bandwidth.

Implementation

A minimal example demonstrating framework usage:

```python
from netcore import Endpoint, Pipe
import serial  # Example using serial communication

# Create a serial connection
device = serial.Serial('/dev/ttyUSB0', 115200)

# Only basic I/O functions are required
pipe = Pipe(device.read, device.write)
endpoint = Endpoint(pipe)

# Register a message handler
@endpoint.request('message')
def handle_message():
    return Response('message', {"status": "received"})

# Start the service
endpoint.start()
```

This example demonstrates: 1. Setup using I/O functions 2. Automatic handling of protocol details 3. Support for message transmission 4. A simple API

With this minimal configuration, an application can handle concurrent messages over a single serial connection without requiring developers to manage concurrency or protocol complexities.

A more comprehensive example showcasing advanced features:

```python
from netcore import Endpoint, Pipe, Blueprint, Response, request
```

```python
# Create a communication pipe with I/O functions
pipe = Pipe(recv_func, send_func)
endpoint = Endpoint(pipe, max_workers=4)  # Configure 4 worker threads

# Create and register a Blueprint
user_bp = Blueprint("users", "/user")

@user_bp.request("/list")
def user_list():
    # Access request data via thread-local context
    page = request.json.get('page', 1)

    # Use the caching system
    cache_key = f"user_list_page_{page}"
    data = endpoint.cache.get(cache_key)
    if data is None:
        data = fetch_users(page)
        endpoint.cache.set(cache_key, data, ttl=300)
    return Response("/user/list", data)

# Add Blueprint middleware
@user_bp.middleware
def auth_middleware(handler):
    def wrapper():
        # Check authorization
        if not is_authorized(request):
            return Response("error", {"status": "unauthorized"})
        return handler()
    return wrapper

endpoint.register_blueprint(user_bp)

# Register event handlers using decorators
@endpoint.event.on('start')
def on_start():
    print("Service started")

# Error handling
@endpoint.error_handle
def handle_error(exception):
    return Response("error", {"error": str(exception)})

# Schedule periodic tasks
def update_cache():
    endpoint.cache.clear()
    print("Cache cleared")

endpoint.scheduler.schedule(update_cache, delay=10, interval=300)

# Start the service with worker threads
endpoint.start()
```

Use Cases

Netcore is suitable for various scenarios, particularly in environments with limited communication channels:

1. **Resource-Constrained Environments**: In embedded systems with restricted connectivity options, Netcore can manage multiple services over a single connection without allocating separate ports for each service.
2. **IoT Applications**: Devices can maintain a single connection to a server while concurrently handling operations such as telemetry uploads, status queries, and configuration updates.
3. **Legacy System Integration**: When integrating with legacy systems offering limited I/O capabilities, Netcore provides a pathway to modern concurrent communication patterns without modifying underlying systems.
4. **Custom Protocol Implementation**: For specialized domains requiring custom protocols, Netcore delivers foundational infrastructure while addressing the complexities of concurrent communication.

Availability

The source code is available at https://github.com/A03HCY/Netcore. Documentation can be found at https://netcore.acdp.top. The software is released under the MIT License.

References