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: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Communication protocols play a crucial role in modern software systems. While many mature communication protocols and frameworks exist, custom communication protocols are often required in specific scenarios. Netcore is a lightweight and extensible communication framework implemented in Python that enables concurrent message transmission over a single connection. Its unique design only requires basic I/O functions (send/receive) from developers, abstracting away all the complexity of protocol implementation, concurrent transmission, and message management. By implementing a message chunking and scheduling mechanism similar to CPU time-slicing, it allows multiple messages to be sent and received simultaneously without establishing multiple connections.

Statement of need

When developing customized communication systems, developers face several challenges:

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

Existing communication frameworks typically require developers to handle complex protocol details or limit them to specific protocols. The Netcore framework addresses these issues through a unique approach:

- Requires only basic I/O functions from developers (send/receive)
- Handles all protocol implementation details internally
- Provides reliable data serialization and concurrent transmission through `LsoProtocol`
- Abstracts transport methods through `Pipe`, allowing any custom communication implementation
- Supports modular application development through the Blueprint system
- Includes built-in event system, task scheduler, and caching mechanisms

Architecture

Netcore employs a three-layer architecture designed for maximum flexibility and minimal developer effort:

1. Protocol Layer (`LsoProtocol`):
   - Implements data packet serialization and deserialization
   - Supports message chunking and concurrent transmission

- Provides data integrity verification
- Handles both memory and file-based storage

2. Transport Layer (`Pipe`):
   - Requires only send/receive functions from developers
   - Handles all message management internally
   - Implements concurrent transmission
   - Provides built-in error handling
3. Application Layer (`Endpoint`):
   - Message routing and blueprint support
   - Request-response management
   - Event system integration
   - Task scheduling and caching
   - Global request context

Features

1. Concurrent Message Transmission
   - Single connection handles multiple messages
   - Automatic message chunking and reassembly
   - Message scheduling and prioritization
   - Non-blocking message transmission
2. Smart Message Management
   - Unique message ID tracking
   - Automatic message queuing
   - Message priority handling
   - Message completion verification
3. Protocol Flexibility
   - Binary protocol with extensible headers
   - Support for JSON and raw data formats
   - Configurable chunk sizes for large data
   - Memory and file-based storage options
4. Advanced Features
   - Blueprint system for modular applications
   - Event system for asynchronous operations
   - Task scheduler for delayed and periodic tasks
   - Cache system with TTL support

Implementation

A minimal example demonstrating the framework's simplicity:

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

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

# Only need to provide basic I/O functions
pipe = Pipe(device.read, device.write)
endpoint = Endpoint(pipe)

# Register message handler
@endpoint.request('message')
def handle_message():
```

```
    return Response('message', {"status": "received"})

# Start service
endpoint.start()
```

This example demonstrates: 1. Minimal setup requirements (only I/O functions needed) 2. Automatic handling of protocol details 3. Built-in support for concurrent transmission 4. Simple and intuitive API

A more comprehensive example showing advanced features:

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

# Create communication pipe with just I/O functions
pipe = Pipe(recv_func, send_func)
endpoint = Endpoint(pipe)

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

@user_bp.request("/list")
def user_list():
    # Use cache system
    data = endpoint.cache.get("user_list")
    if data is None:
        data = fetch_users()
        endpoint.cache.set("user_list", data, ttl=300)
    return Response("/user/list", data)

endpoint.register_blueprint(user_bp)

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

# Schedule periodic task
def update_cache():
    endpoint.cache.delete("user_list")

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

# Start service
endpoint.start()
```

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