Thalamini Hub

A high-performance message routing system written in Go that provides reliable message delivery through a publish-subscribe (pub/sub) pattern with topic-based routing. Measured performance of over 10,000 messages/second with 0.06ms average latency.

> **Note**: This application is designed for Linux systems only. Due to differences in socket communication handling on Windows, the application may experience issues when run on Windows systems.

Features

- High-throughput message processing (>10,000 msg/sec)
- Low latency message delivery (~0.06ms per message)
- Configurable message queues with backpressure handling
- Topic-based message routing with pattern matching
- Automatic client health monitoring (15s ping interval)
- Thread-safe concurrent operations
- TCP-based communication with connection pooling
- Automatic recovery and reconnection
- JSON-based configuration

Installation

```
go get github.com/markoxley/hub
```

Quick Start

The Thalamini system consists of a central hub server that handles message routing, and client libraries for publishing and consuming messages. This guide focuses on implementing publishers and consumers using the client libraries.

Publishing Messages

```
package main

import (
    "log"
    "time"

    "github.com/markoxley/dani/msg"
    "github.com/markoxley/dani/thal"
)

func main() {
    // Configure the publisher
```

```
    cfg := &thal.PublishConfig{
        Address:      "127.0.0.1",  // Thalamini hub address
        Port:        24353,         // Thalamini hub port
        QueueSize:   1000,
        DialTimeout: 1000,  // milliseconds
        MaxRetries:  3,
    }

    // Initialize the publisher
    if err := thal.Init(cfg); err != nil {
        log.Fatal(err)
    }

    // Prepare message data
    data := map[string]interface{}{
        "message": "Hello, World!",
        "timestamp": time.Now(),
        "priority": 1,
    }

    // Publish a message to a specific topic
    err := thal.Publish("notifications", data)
    if err != nil {
        log.Printf("Failed to publish: %v", err)
    }
}
```

Consuming Messages

```
package main

import (
    "log"
    "fmt"

    "github.com/markoxley/dani/msg"
    "github.com/markoxley/dani/thal"
)

// MyConsumer implements the thal.Consumer interface
type MyConsumer struct{}

// Consume processes received messages
func (c *MyConsumer) Consume(m *msg.Message) {
    data, err := m.Data()
    if err != nil {
        log.Printf("Failed to parse message: %v", err)
        return
    }
    log.Printf("Received message: %v", data)
}
```

```go
func main() {
    // Configure the consumer
    cfg := &thal.ConsumerConfig{
        Name:         "myclient",
        HubAddress:   "127.0.0.1",
        HubPort:      24353,
        Address:      "0.0.0.0",
        Port:         8080,
        QueueSize:    1000,
        DialTimeout:  1000,  // milliseconds
        MaxRetries:   3,
        Topics:       []string{"topic1", "topic2"},
    }

    // Create consumer and start listening
    consumer := &MyConsumer{}
    err := thal.Listen(consumer, cfg)
    if err != nil {
        log.Fatalf("Failed to start consumer: %v", err)
    }

    // Keep the main thread running
    select {}
}
```

Configuration

The system uses JSON-based configuration for all components. Configuration files support environment variable expansion.

Hub Configuration

The Thalamini hub server is configured via a `config.json` file. Here's an example configuration with default values:

```jsonc
{
    "ip": "0.0.0.0",          // Required
    "port": 24353,            // Required
    "queueSize": 1000000,
    "workerCount": 100,
    "dialTimeout": 1000,
    "writeTimeout": 2000,
    "readTimeout": 30000,
    "maxRetries": 3,
    "clientQueueSize": 1000,
    "clientWorkerCount": 100,
    "clientDialTimeout": 1000,
    "clientWriteTimeout": 2000,
    "clientMaxRetries": 3
}
```

Parameter	Description	Required	Default
ip	IP address to bind the server to	Yes	"0.0.0.0"
port	Port number to listen on	Yes	24353
queueSize	Size of the message queue buffer	No	1,000,000
workerCount	Number of concurrent message processing workers	No	100
dialTimeout	TCP connection timeout (ms)	No	1000
writeTimeout	Message write timeout (ms)	No	2000
readTimeout	Message read timeout (ms)	No	30000
maxRetries	Maximum message delivery attempts	No	3
clientQueueSize	Size of each client's message queue	No	1000
clientWorkerCount	Workers per client for message processing	No	100
clientDialTimeout	Client TCP connection timeout (ms)	No	1000
clientWriteTimeout	Client message write timeout (ms)	No	2000
clientMaxRetries	Maximum client delivery attempts	No	3

Publisher Configuration

```
{
    "address": "127.0.0.1",    // Required
    "port": 24353,             // Required
    "queueSize": 1000,
    "dialTimeout": 1000,
    "writeTimeout": 2000,
    "maxRetries": 3
}
```

Parameter	Description	Required	Default
address	Hub server address	Yes	"127.0.0.1"
port	Hub server port	Yes	24353
queueSize	Message buffer size	No	1000
dialTimeout	Connection timeout (ms)	No	1000
writeTimeout	Message write timeout (ms)	No	2000
maxRetries	Failed message retry limit	No	3

Consumer Configuration

```
{
    "name": "myconsumer",      // Required
    "hubAddress": "127.0.0.1", // Required
    "hubPort": 24353,          // Required
    "address": "0.0.0.0",      // Required
    "port": 8080,              // Required
    "queueSize": 1000,
    "dialTimeout": 1000,
    "writeTimeout": 2000,
    "maxRetries": 3,
    "topics": ["topic1", "topic2"] // Required
}
```

Parameter	Description	Required	Default
name	Unique consumer identifier	Yes	
hubAddress	Hub server address	Yes	"127.0.0.1"
hubPort	Hub server port	Yes	24353
address	Local binding address	Yes	"0.0.0.0"
port	Local binding port	Yes	
queueSize	Message buffer size	No	1000
dialTimeout	Connection timeout (ms)	No	1000
writeTimeout	Message write timeout (ms)	No	2000
maxRetries	Failed operation retry limit	No	3
topics	Topics to subscribe to	Yes	

Performance Characteristics

Based on performance testing:

- **Throughput**: >10,000 messages/second
- **Latency**: ~0.06ms average per message
- **Queue Size**: 1,000,000 messages default
- **Worker Pool**: 100 workers default
- **Health Check**: 15-second ping interval
- **Connection Pool**: Dynamic with automatic cleanup
- **Memory Usage**: ~1KB per queued message
- **Recovery**: Automatic with exponential backoff

Architecture

The system consists of three main components:

1. **Hub**: Central message router that manages connections and message delivery
    - Worker pool for concurrent message processing
    - Bounded message queue with backpressure
    - Client health monitoring
    - Topic-based routing
2. **Publisher**: Client that sends messages to the hub
    - Asynchronous message queuing
    - Automatic retries with backoff
    - Connection pooling
    - Rate limiting support
3. **Consumer**: Client that receives messages from topics
    - Buffered message processing
    - Automatic reconnection
    - Health monitoring
    - Topic pattern matching

Message Flow

1. Publishers send messages to the hub
2. Messages are queued in bounded buffers
3. Worker pool processes messages concurrently
4. Messages are routed to subscribed consumers
5. Failed deliveries are retried automatically
6. Health is monitored via ping messages

Thread Safety

All operations are thread-safe and support concurrent access:

- Mutex protection for shared resources
- Channel-based message passing
- Atomic operations for counters
- Connection pooling for network I/O
- Goroutine confinement for state

Best Practices

1. **Configuration**:
    - Use JSON configuration files
    - Set appropriate queue sizes for your load
    - Configure timeouts based on network conditions
    - Adjust worker count for your hardware
2. **Performance**:

- Monitor queue lengths for backpressure
- Set appropriate rate limits
- Use topic patterns effectively
- Configure retry counts based on reliability needs

3. **Error Handling**:

- Check all error returns
- Implement graceful shutdown
- Monitor health check failures
- Log connection issues

4. **Topic Design**:

- Use hierarchical topics
- Keep topic names concise
- Document topic patterns
- Consider message routing paths

License