LibPolyCall Trial Documentation

By OBINexusComputing - Nnamdi Michael Okpala

Introduction

Welcome to the LibPolyCall Trial! This document guides you through using LibPolyCall for server communication with both interactive and non-interactive modes. LibPolyCall implements a program-first approach to interface design and business integration, providing a unified architecture for multi-language communication.

What is LibPolyCall?

LibPolyCall is a polymorphic library that enables seamless communication between services written in different programming languages. It uses both direct protocol implementation and stateless architecture to serve clients in a language-agnostic way.

The core philosophy of LibPolyCall is "program-first" rather than "binding-first." Instead of creating separate implementations for each language, LibPolyCall focuses on the protocol and state management, allowing thin language bindings to connect to this unified infrastructure.

Key Features

- **Program-primary interface design**: Core functionality lives in the protocol, not the bindings
- **Stateless communication**: Maintains clean separation between components
- **Flexible client bindings**: Supports multiple languages (Node.js, Python, Java, Go)
- **Strong state management**: Reliable tracking of system state
- **Network transport flexibility**: Works over various transport mechanisms
- **Open source architecture**: Extensible and customizable

Setting Up Your Environment

Basic Setup

1. Create necessary configuration directories:

```
sudo mkdir -p /opt/polycall/services/{node,python,java,go}
```

2. Create the main `config.Polycallfile` in your project root:

```
# PolyCall System Configuration
# Language Server Definitions
server node 8080:8084
server python 3001:8084
server java 3002:8082
server go 3003:8083
```

```
# Network Configuration
network start
network_timeout=5000
max_connections=1000
# Global Settings
log_directory=/var/log/polycall
workspace_root=/opt/polycall
# Service Discovery
auto_discover=true
discovery_interval=60
# Security Configuration
tls_enabled=true
cert_file=/etc/polycall/cert.pem
key_file=/etc/polycall/key.pem
# Resource Limits
max_memory_per_service=1G
max_cpu_per_service=2
# Monitoring
enable_metrics=true
metrics_port=9090
```

3. Create language-specific configurations in their respective directories:

For Node.js binding:

```
# Language Server Configuration
# Port mapping for host:container
port=8080:8084 # Node.js
# Server type specification
server_type=node
# Service specific settings
workspace=/opt/polycall/node
log_level=info
max_connections=100
# Server capabilities
supports_diagnostics=true
supports_completion=true
supports_formatting=true
# Performance settings
max_memory=512M
timeout=30
# Security settings
allow_remote=false
require_auth=true
```

Similar configuration files should be created for Python, Java, and Go services.

Running LibPolyCall

Non-Interactive Mode

This mode runs LibPolyCall as a service without requiring manual interaction.

1. Start the PolyCall service:

```
./bin/polycall -f config.Polycallfile
```

2. Start your language-specific binding server. For Node.js:

```
cd ../bindings/node-polycall/examples
node server.js
```

3. Test the connection with a client:

```
node test_client.js
```

Interactive Mode (REPL)

This mode lets you interact with the LibPolyCall system directly through a command-line interface.

1. Start the PolyCall CLI in interactive mode:

```
./bin/polycall
```

2. You will see a prompt where you can issue commands:

```
PolyCall CLI v1.0.0 - Type 'help' for commands

>
```

3. Common commands include:
   - `start_network`: Start network services
   - `list_endpoints`: List all network endpoints
   - `list_clients`: List connected clients
   - `status`: Show system status
   - `help`: Display available commands
   - `quit`: Exit the program

Testing Communication with LibPolyCall

Python Client Example

The following Python code demonstrates how to communicate with a LibPolyCall server:

```
import json
import http.client

def test_post():
    conn = http.client.HTTPConnection("localhost", 8084)  # Use binding-
specific port
    headers = {'Content-type': 'application/json'}
    post_data = json.dumps({'title': 'Test Book', 'author': 'Test Author'})

    conn.request('POST', '/books', post_data, headers)
    response = conn.getresponse()
    data = response.read().decode()
    print('Created book:', json.loads(data))

    conn.close()
    test_get()

def test_get():
    conn = http.client.HTTPConnection("localhost", 8084)
    conn.request('GET', '/books')
    response = conn.getresponse()
    data = response.read().decode()
    print('Books list:', json.loads(data))

    conn.close()

if __name__ == "__main__":
    test_post()
```

Node.js Client Example

```
const http = require('http');

// Helper function to make HTTP requests
function makeRequest(method, path, data = null) {
    return new Promise((resolve, reject) => {
        const requestOptions = {
            hostname: 'localhost',
            port: 8084,  // Use binding-specific port
            path,
            method,
            headers: {
                'Content-Type': 'application/json'
            }
        };

        const req = http.request(requestOptions, (res) => {
            let data = '';

            res.on('data', chunk => {
                data += chunk;
            });

            res.on('end', () => {
                try {
                    const result = JSON.parse(data);
                    resolve(result);
                } catch (error) {
                    reject(error);
                }
            });
        });

        req.on('error', reject);

        if (data) {
            req.write(JSON.stringify(data));
        }

        req.end();
    });
}

// Test functions
async function testCreateBook() {
    console.log('\nTesting POST /books');
    try {
        const bookData = {
            title: 'Test Book',
            author: 'Test Author'
        };
        const result = await makeRequest('POST', '/books', bookData);
        console.log('Created book:', result);
        return result;
    } catch (error) {
        console.error('Failed to create book:', error.message);
        throw error;
    }
}

async function testGetBooks() {
    console.log('\nTesting GET /books');
    try {
        const result = await makeRequest('GET', '/books');
        console.log('Books list:', result);
        return result;
    } catch (error) {
        console.error('Failed to get books:', error.message);
        throw error;
    }
}

// Run tests
async function runTests() {
    try {
        // First create a book
        await testCreateBook();

        // Then get all books
        await testGetBooks();

    } catch (error) {
        console.error('Test suite failed:', error.message);
    }
}

// Run the tests
runTests();
```

Verifying Your Setup

To verify that your LibPolyCall installation is running correctly:

1. Check the running services:

```
ps aux | grep polycall
```

2. Monitor the port mappings:

```
netstat -tulpn | grep polycall
```

3. Check log files for any errors:

```
tail -f /var/log/polycall/polycall.log
```

Adding New Language Bindings

To integrate a new language binding:

1. Add its configuration to `config.Polycallfile`:

```
server newlang 3004:8084
```

2. Create its service directory and configuration:

```
mkdir -p /opt/polycall/services/newlang
echo "port=3004:8084
```

```
server_type=newlang
workspace=/opt/polycall/services/newlang" >
/opt/polycall/services/newlang/.polycallrc
```

3. Restart the PolyCall service to apply the new configuration.

Important Port Mappings

In the configuration, note the port mapping format: `host_port:container_port`

- `8080:8084`: Node.js service (access on port 8084)
- `3001:8084`: Python service (access on port 8084)
- `3002:8082`: Java service (access on port 8082)
- `3003:8083`: Go service (access on port 8083)

When connecting to a service, always use the container port (the second number in the mapping).

Disclaimer

This is a trial version of LibPolyCall, providing core functionality for evaluation purposes. The full version offers additional features including advanced security controls, failover mechanisms, comprehensive telemetry, and extended language support.

Get the Full Version

The complete LibPolyCall solution is available on payhip.com and piecex.com. Pricing information will be provided shortly.

Support

For questions or assistance, please contact:

- Email: nnamdi@obinexuscomputing.com

Thank you for trying LibPolyCall!

Nnamdi Michael Okpala
OBINexusComputing