Language-Specific Binding in LibPolyCall: Implementation Guide

Introduction to LibPolyCall Binding Architecture

LibPolyCall's architecture is built on the program-first design principle, which emphasizes protocol-level functionality over language-specific implementations. This approach enables consistent behavior across multiple programming languages while allowing each language binding to leverage native idioms and patterns.

This technical guide explains the architecture and implementation of language-specific bindings for LibPolyCall, focusing on protocol integration, state management, and error handling.

Core Architecture Principles

LibPolyCall's binding architecture follows these fundamental principles:

1. **Protocol Centricity**: The core protocol implementation is language-agnostic
2. **Thin Bindings**: Language-specific bindings are lightweight wrappers around the core protocol
3. **State Awareness**: Bindings maintain protocol state for proper request handling
4. **Native Idioms**: Each binding follows language-specific patterns and conventions
5. **Error Propagation**: Errors propagate consistently across language boundaries

Protocol Integration

The first step in implementing a language binding is integrating with the LibPolyCall protocol. This involves:

1. Establishing a network connection
2. Handling protocol messages
3. Managing state transitions
4. Processing responses

Core Protocol Implementation

The protocol is defined in `polycall_protocol_context.h` and follows a state machine model:

c

```c
typedef enum {
    POLYCALL_PROTOCOL_STATE_INIT,
    POLYCALL_PROTOCOL_STATE_HANDSHAKE,
    POLYCALL_PROTOCOL_STATE_AUTH,
    POLYCALL_PROTOCOL_STATE_READY,
    POLYCALL_PROTOCOL_STATE_ERROR,
    POLYCALL_PROTOCOL_STATE_CLOSED
} polycall_protocol_state_t;
```

Message types are structured as:

c

```c
typedef enum {
    POLYCALL_PROTOCOL_MSG_HANDSHAKE,
    POLYCALL_PROTOCOL_MSG_AUTH,
    POLYCALL_PROTOCOL_MSG_COMMAND,
    POLYCALL_PROTOCOL_MSG_ERROR,
    POLYCALL_PROTOCOL_MSG_HEARTBEAT
} polycall_protocol_msg_type_t;
```

Binding Implementation

For each language binding, implement these core protocol functions:

1. **Connection Establishment**:

```javascript
// Node.js Example
class PolyCallClient {
    constructor(options) {
        this.options = options;
        this.state = "INIT";
        this.socket = null;
    }

    async connect() {
        return new Promise((resolve, reject) => {
            this.socket = net.createConnection({
                host: this.options.host || 'localhost',
                port: this.options.port || 8084
            }, () => {
                this.startHandshake()
                    .then(resolve)
                    .catch(reject);
            });

            this.socket.on('error', reject);
            this.setupMessageHandling();
        });
    }
}
```

2. **Protocol Message Handling**:

```javascript
// Node.js Example
setupMessageHandling() {
    this.socket.on('data', (data) => {
        const message = this.parseMessage(data);
        switch(message.type) {
            case MESSAGE_TYPES.HANDSHAKE:
                this.handleHandshakeResponse(message);
                break;
            case MESSAGE_TYPES.AUTH:
                this.handleAuthResponse(message);
                break;
            case MESSAGE_TYPES.COMMAND:
                this.handleCommandResponse(message);
                break;
            case MESSAGE_TYPES.ERROR:
```

```
                this.handleErrorResponse(message);
                break;
            case MESSAGE_TYPES.HEARTBEAT:
                this.handleHeartbeat(message);
                break;
        }
    });
}
```

3. **State Management**:

```javascript
// Node.js Example
async startHandshake() {
    if (this.state !== "INIT") {
        throw new Error("Cannot start handshake: invalid state");
    }

    const handshakeMessage = {
        type: MESSAGE_TYPES.HANDSHAKE,
        version: PROTOCOL_VERSION,
        flags: 0
    };

    await this.sendMessage(handshakeMessage);
    this.state = "HANDSHAKE";
}
```

Language-Specific Implementations

Each language binding should adhere to idiomatic patterns for that language while maintaining protocol compatibility.

Node.js Binding

Node.js bindings leverage promises and async/await for protocol communications:

javascript

```javascript
// Node.js Promise-based API
class PolyCallClient {
    // ... constructor and other methods ...

    async sendCommand(path, data) {
        if (this.state !== "READY") {
            throw new Error("Client not ready");
        }

        return new Promise((resolve, reject) => {
            const commandId = this.generateId();
            this.pendingCommands.set(commandId, { resolve, reject });

            const message = {
                type: MESSAGE_TYPES.COMMAND,
                id: commandId,
                path: path,
                data: data
            };

            this.sendMessage(message).catch(reject);
        });
    }
}

// Usage example
const client = new PolyCallClient({ port: 8084 });
await client.connect();
const result = await client.sendCommand('/books', { title: 'Test Book' });
```

Python Binding

Python bindings can use asyncio for protocol handling:

python

```python
# Python asyncio-based API
class PolyCallClient:
    def __init__(self, host='localhost', port=8084):
        self.host = host
        self.port = port
        self.state = "INIT"
        self.reader = None
        self.writer = None
        self.pending_commands = {}

    async def connect(self):
        self.reader, self.writer = await asyncio.open_connection(
            self.host, self.port)
        await self.start_handshake()
        asyncio.create_task(self.message_loop())

    async def send_command(self, path, data):
        if self.state != "READY":
            raise RuntimeError("Client not ready")

        command_id = self.generate_id()
        future = asyncio.Future()
        self.pending_commands[command_id] = future

        message = {
            "type": MESSAGE_TYPES.COMMAND,
            "id": command_id,
            "path": path,
            "data": data
        }

        await self.send_message(message)
        return await future

# Usage example
client = PolyCallClient(port=8084)
await client.connect()
result = await client.send_command('/books', {'title': 'Test Book'})
```

Java Binding

Java bindings can use CompletableFuture for asynchronous operations:

java

```java
// Java CompletableFuture-based API
public class PolyCallClient implements AutoCloseable {
    private String host;
    private int port;
    private String state = "INIT";
    private Socket socket;
    private Map<String, CompletableFuture<JsonObject>> pendingCommands = new ConcurrentHashMap<

    public PolyCallClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public CompletableFuture<Void> connect() {
        CompletableFuture<Void> result = new CompletableFuture<>();

        try {
            socket = new Socket(host, port);
            startMessageThread();
            startHandshake()
                .thenRun(() -> result.complete(null))
                .exceptionally(ex -> {
                    result.completeExceptionally(ex);
                    return null;
                });
        } catch (IOException ex) {
            result.completeExceptionally(ex);
        }

        return result;
    }

    public CompletableFuture<JsonObject> sendCommand(String path, JsonObject data) {
        if (!"READY".equals(state)) {
            CompletableFuture<JsonObject> future = new CompletableFuture<>();
            future.completeExceptionally(new IllegalStateException("Client not ready"));
            return future;
        }

        String commandId = generateId();
        CompletableFuture<JsonObject> future = new CompletableFuture<>();
        pendingCommands.put(commandId, future);

        JsonObject message = Json.createObjectBuilder()
            .add("type", MESSAGE_TYPES_COMMAND)
```

```
            .add("type", MESSAGE_TYPES.COMMAND)
            .add("id", commandId)
            .add("path", path)
            .add("data", data)
            .build();

        try {
            sendMessage(message);
        } catch (IOException ex) {
            future.completeExceptionally(ex);
            pendingCommands.remove(commandId);
        }

        return future;
    }

    // ... other methods ...
}

// Usage example
PolyCallClient client = new PolyCallClient("localhost", 8084);
client.connect()
    .thenCompose(v -> {
        JsonObject bookData = Json.createObjectBuilder()
            .add("title", "Test Book")
            .build();
        return client.sendCommand("/books", bookData);
    })
    .thenAccept(System.out::println)
    .exceptionally(ex -> {
        ex.printStackTrace();
        return null;
    });
```

Go Binding

Go bindings leverage channels and goroutines:

go

```go
// Go channel-based API
type PolyCallClient struct {
    host           string
    port           int
    state          string
    conn           net.Conn
    pendingCommands map[string]chan interface{}
    errorChan      chan error
}

func NewPolyCallClient(host string, port int) *PolyCallClient {
    return &PolyCallClient{
        host:           host,
        port:           port,
        state:          "INIT",
        pendingCommands: make(map[string]chan interface{}),
        errorChan:      make(chan error, 10),
    }
}

func (c *PolyCallClient) Connect() error {
    var err error
    c.conn, err = net.Dial("tcp", fmt.Sprintf("%s:%d", c.host, c.port))
    if err != nil {
        return err
    }

    go c.messageLoop()

    return c.startHandshake()
}

func (c *PolyCallClient) SendCommand(path string, data interface{}) (interface{}, error) {
    if c.state != "READY" {
        return nil, errors.New("client not ready")
    }

    commandID := c.generateID()
    resultChan := make(chan interface{}, 1)
    c.pendingCommands[commandID] = resultChan

    message := map[string]interface{}{
        "type": MESSAGE_TYPES_COMMAND,
        "id":   commandID,
        "path": path,
```

```
        "path": path,
        "data": data,
    }

    if err := c.sendMessage(message); err != nil {
        delete(c.pendingCommands, commandID)
        return nil, err
    }

    select {
    case result := <-resultChan:
        return result, nil
    case err := <-c.errorChan:
        return nil, err
    case <-time.After(30 * time.Second):
        delete(c.pendingCommands, commandID)
        return nil, errors.New("timeout waiting for response")
    }
}

// Usage example
client := NewPolyCallClient("localhost", 8084)
if err := client.Connect(); err != nil {
    log.Fatal(err)
}

result, err := client.SendCommand("/books", map[string]interface{}{
    "title": "Test Book",
})
if err != nil {
    log.Fatal(err)
}
fmt.Println(result)
```

Error Handling

LibPolyCall employs hierarchical error handling that should be reflected in language bindings:

javascript

```javascript
// Node.js Error Handling
class PolycallError extends Error {
    constructor(source, code, message) {
        super(message);
        this.name = "PolycallError";
        this.source = source;
        this.code = code;
    }
}

class PolyCallClient {
    // ... other methods ...

    handleErrorResponse(message) {
        const commandId = message.id;
        const pendingCommand = this.pendingCommands.get(commandId);

        if (pendingCommand) {
            this.pendingCommands.delete(commandId);
            pendingCommand.reject(new PolycallError(
                message.source,
                message.code,
                message.message
            ));
        }

        // Check if this is a system-level error
        if (message.severity === "FATAL") {
            this.state = "ERROR";
            this.emit('error', new PolycallError(
                message.source,
                message.code,
                message.message
            ));
        }
    }
}
```

Configuration Integration

Bindings should integrate with the configuration system:

javascript

```javascript
// Node.js Configuration Integration
class PolyCallClient {
    constructor(options) {
        this.options = options;

        // Load configuration
        if (options.configFile) {
            this.loadConfigFile(options.configFile);
        }
    }

    loadConfigFile(filename) {
        try {
            const configData = fs.readFileSync(filename, 'utf8');
            const config = JSON.parse(configData);

            // Override options with config file values
            this.options = {
                ...this.options,
                ...config
            };
        } catch (error) {
            throw new Error(`Failed to load config file: ${error.message}`);
        }
    }
}
```

Testing Protocol Compliance

To ensure protocol compatibility, bindings should implement test suites that verify proper protocol behavior:

javascript

```javascript
// Node.js Protocol Compliance Test
describe('Protocol Handshake', () => {
    it('should successfully complete handshake', async () => {
        const client = new PolyCallClient({ port: 8084 });

        // Create a mock server
        const mockServer = net.createServer((socket) => {
            socket.on('data', (data) => {
                const message = JSON.parse(data.toString());

                // Verify handshake message structure
                expect(message.type).to.equal(MESSAGE_TYPES.HANDSHAKE);
                expect(message.version).to.equal(PROTOCOL_VERSION);

                // Send handshake response
                const response = {
                    type: MESSAGE_TYPES.HANDSHAKE,
                    version: PROTOCOL_VERSION,
                    success: true
                };

                socket.write(JSON.stringify(response));
            });
        });

        // Start mock server and connect client
        await new Promise(resolve => mockServer.listen(8084, resolve));
        await client.connect();

        expect(client.state).to.equal("HANDSHAKE");

        mockServer.close();
    });
});
```

Implementing State Machine Integration

LibPolyCall's hierarchical state machine should be reflected in the binding:

javascript

```javascript
// Node.js State Machine Integration
class PolyCallStateMachine {
    constructor() {
        this.states = new Map();
        this.transitions = new Map();
        this.currentState = null;
    }

    addState(name, config) {
        this.states.set(name, {
            name,
            onEnter: config.onEnter,
            onExit: config.onExit,
            parentState: config.parentState || null,
            permissions: config.permissions || []
        });
    }

    addTransition(name, fromState, toState, guard) {
        this.transitions.set(name, {
            name,
            fromState,
            toState,
            guard
        });
    }

    async executeTransition(name) {
        const transition = this.transitions.get(name);
        if (!transition) {
            throw new Error(`Transition not found: ${name}`);
        }

        if (this.currentState !== transition.fromState) {
            throw new Error(`Cannot execute transition: wrong state`);
        }

        // Check guard condition
        if (transition.guard && !await transition.guard()) {
            throw new Error(`Transition guard rejected`);
        }

        // Execute exit action
        const currentStateObj = this.states.get(this.currentState);
        if (currentStateObj.onExit) {
```

```
        if (currentStateObj.onExit) {
            await currentStateObj.onExit();
        }

        // Update state
        this.currentState = transition.toState;

        // Execute enter action
        const newStateObj = this.states.get(this.currentState);
        if (newStateObj.onEnter) {
            await newStateObj.onEnter();
        }

        return true;
    }

    hasPermission(permission) {
        if (!this.currentState) return false;

        const stateObj = this.states.get(this.currentState);
        return stateObj.permissions.includes(permission);
    }
}
```

Router Implementation

The router maps protocol paths to handlers:

javascript

```javascript
// Node.js Router Implementation
class Router {
    constructor() {
        this.routes = new Map();
    }

    addRoute(path, handlers) {
        this.routes.set(path, handlers);
    }

    async handleRequest(path, method, data) {
        const route = this.routes.get(path);

        if (!route) {
            throw new Error(`Route not found: ${path}`);
        }

        const handler = route[method];

        if (!handler) {
            throw new Error(`Method not allowed: ${method}`);
        }

        // Create context for handler
        const context = {
            path,
            method,
            data
        };

        return await handler(context);
    }
}
```

Security Considerations

Language bindings should implement proper security measures:

javascript

```javascript
// Node.js Security Implementation
class PolyCallClient {
    // ... other methods ...

    async authenticate(credentials) {
        if (this.state !== "HANDSHAKE") {
            throw new Error("Cannot authenticate: invalid state");
        }

        // Create secure token
        const token = this.createSecureToken(credentials);

        const authMessage = {
            type: MESSAGE_TYPES.AUTH,
            token: token
        };

        return new Promise((resolve, reject) => {
            const authId = this.generateId();
            this.pendingCommands.set(authId, { resolve, reject });

            authMessage.id = authId;

            this.sendMessage(authMessage).catch(reject);
        });
    }

    createSecureToken(credentials) {
        // Create HMAC-based token
        const timestamp = Date.now();
        const message = `${credentials.username}:${timestamp}`;
        const hmac = crypto.createHmac('sha256', credentials.key)
            .update(message)
            .digest('hex');

        return {
            username: credentials.username,
            timestamp: timestamp,
            signature: hmac
        };
    }
}
```

Event Handling

Bindings should provide event-based notifications:

```javascript
// Node.js Event Handling
class PolyCallClient extends EventEmitter {
    // ... other methods ...

    constructor(options) {
        super();
        this.options = options;
        this.state = "INIT";
        this.socket = null;
    }

    messageLoop() {
        this.socket.on('data', (data) => {
            try {
                const message = this.parseMessage(data);
                this.handleMessage(message);
            } catch (error) {
                this.emit('error', error);
            }
        });

        this.socket.on('close', () => {
            this.state = "CLOSED";
            this.emit('close');
        });

        this.socket.on('error', (error) => {
            this.emit('error', error);
        });
    }
}

// Usage example
const client = new PolyCallClient({ port: 8084 });
client.on('error', (error) => {
    console.error('Client error:', error);
});
client.on('close', () => {
    console.log('Connection closed');
});
```

Conclusion

Implementing language-specific bindings for LibPolyCall requires careful attention to protocol compatibility while embracing language-specific idioms. By focusing on:

1. Clean protocol integration
2. Native language patterns
3. Consistent error handling
4. Proper state management
5. Security implementation

You can create bindings that provide a seamless development experience while maintaining the benefits of LibPolyCall's program-first design.

This approach enables developers to work with familiar patterns in their language of choice while leveraging the powerful cross-language communication capabilities of the underlying protocol.

For additional information on language binding implementation or to contribute bindings for other languages, please refer to the LibPolyCall documentation or contact technical support at nnamdi@obinexuscomputing.com.