MCP Client Implementation Guide

Overview

The Model Context Protocol (MCP) client enables NEXUS to communicate with satellite applications, execute commands, and coordinate workflows across the SFG ecosystem.

Architecture

```
┌─────────────────┐
│   SFG NEXUS     │
│  (MCP Client)   │
└────────┬────────┘
         │
         ├─── MCP ───> Satellite App 1 (MCP Server)
         ├─── MCP ───> Satellite App 2 (MCP Server)
         ├─── MCP ───> Satellite App 3 (MCP Server)
         └─── MCP ───> Satellite App N (MCP Server)
```

Implementation Status

Status: Week 2 Implementation (November 11-17, 2025)

Components

1. MCP Client Library

Location: `/app/lib/mcp-client/`

Core Functions:
- `connect(appId, baseUrl)` - Establish connection to satellite app
- `execute(command, params)` - Execute command on satellite
- `query(resource, filters)` - Query data from satellite
- `subscribe(event, callback)` - Subscribe to satellite events
- `disconnect(appId)` - Close connection

2. Connection Manager

Manages connections to multiple satellite apps concurrently.

```
class MCPConnectionManager {
  private connections: Map<string, MCPConnection>;

  async connect(appId: string): Promise<MCPConnection> {
    // Establish connection to satellite app
  }

  async broadcast(command: string, params: any): Promise<Results[]> {
    // Execute command on all connected apps
  }

  getConnection(appId: string): MCPConnection | null {
    // Get existing connection
  }
}
```

3. Command Router

Routes commands to appropriate satellite apps based on capabilities.

```
class MCPCommandRouter {
  async route(command: string, params: any): Promise<string> {
    // Determine which app can handle this command
    // Return appId of best match
  }

  async executeOnBestApp(command: string, params: any): Promise<Result> {
    const appId = await this.route(command, params);
    return await mcpClient.execute(appId, command, params);
  }
}
```

MCP Protocol Specification

Connection Handshake

```
{
  "type": "connect",
  "client": "sfg-nexus",
  "version": "1.0.0",
  "capabilities": ["execute", "query", "subscribe"]
}
```

Response:

```json
{
  "type": "connected",
  "server": "chronoshift-pro",
  "version": "1.0.0",
  "capabilities": ["execute", "query", "subscribe"],
  "resources": [
    { "name": "schedules", "methods": ["GET", "POST", "PATCH"] },
    { "name": "payroll", "methods": ["GET", "POST"] }
  ]
}
```

Command Execution

```json
{
  "type": "execute",
  "command": "create_schedule",
  "params": {
    "jobId": "18457",
    "teamId": "team-install-1",
    "date": "2025-11-15"
  },
  "requestId": "req-12345"
}
```

Response:

```json
{
  "type": "result",
  "requestId": "req-12345",
  "success": true,
  "data": {
    "scheduleId": "sch-67890",
    "status": "created"
  }
}
```

Query Execution

```json
{
  "type": "query",
  "resource": "schedules",
  "filters": {
    "date": "2025-11-15",
    "status": "SCHEDULED"
  },
  "requestId": "req-12346"
}
```

Response:

```json
{
  "type": "result",
  "requestId": "req-12346",
  "success": true,
  "data": [
    { "id": "sch-67890", "jobId": "18457", "teamId": "team-install-1" }
  ]
}
```

Event Subscription

```json
{
  "type": "subscribe",
  "event": "schedule_updated",
  "filters": {
    "jobId": "18457"
  },
  "requestId": "req-12347"
}
```

Event Notification:

```json
{
  "type": "event",
  "event": "schedule_updated",
  "data": {
    "scheduleId": "sch-67890",
    "changes": {
      "status": "COMPLETED"
    }
  },
  "timestamp": "2025-11-15T14:30:00Z"
}
```

Usage Examples

Connecting to a Satellite App

```javascript
import { MCPClient } from '@/lib/mcp-client';

const client = new MCPClient();

// Connect to ChronoShift Pro
await client.connect('chronoshift-pro', 'https://chronoshift-pro.abacusai.app');

// Verify connection
const isConnected = client.isConnected('chronoshift-pro');
console.log('Connected:', isConnected);
```

Executing a Command

```
// Create a new schedule
const result = await client.execute('chronoshift-pro', 'create_schedule', {
  jobId: '18457',
  teamId: 'team-install-1',
  date: '2025-11-15',
  time: '09:00'
});

console.log('Schedule created:', result.data.scheduleId);
```

Querying Data

```
// Get all schedules for a specific date
const schedules = await client.query('chronoshift-pro', 'schedules', {
  date: '2025-11-15',
  status: 'SCHEDULED'
});

console.log('Schedules:', schedules.data);
```

Subscribing to Events

```
// Subscribe to schedule updates
client.subscribe('chronoshift-pro', 'schedule_updated', (event) => {
  console.log('Schedule updated:', event.data);

  // Handle the event
  if (event.data.status === 'COMPLETED') {
    // Trigger next workflow step
  }
});
```

Broadcasting to Multiple Apps

```
// Execute command on all connected apps
const results = await client.broadcast('sync_data', {
  entity: 'customer',
  customerId: 'cust-123'
});

// Check results
for (const result of results) {
  console.log(`${result.appId}: ${result.success ? 'Success' : 'Failed'}`);
}
```

Workflow Orchestration

Multi-Step Workflow Example

```typescript
// Workflow: Quote → Job → Schedule → Production

async function processQuoteToProduction(quoteId: string) {
  // Step 1: Convert quote to job (NEXUS internal)
  const job = await convertQuoteToJob(quoteId);

  // Step 2: Create schedule (ChronoShift Pro)
  const schedule = await client.execute('chronoshift-pro', 'create_schedule', {
    jobId: job.id,
    installationDate: job.installationDate
  });

  // Step 3: Create production tasks (SFG Vertex)
  const tasks = await client.execute('sfg-vertex', 'create_production_tasks', {
    jobId: job.id,
    specifications: job.specifications
  });

  // Step 4: Sync to ESP (SFG ESP)
  await client.execute('sfg-esp', 'sync_job', {
    jobId: job.id,
    scheduleId: schedule.data.scheduleId
  });

  // Step 5: Notify team (SFGComms Hub)
  await client.execute('sfgcomms-hub', 'send_notification', {
    teamId: schedule.data.teamId,
    message: `New job ${job.id} scheduled for ${job.installationDate}`
  });

  console.log('Workflow completed successfully');
}
```

Error Handling

```
try {
  const result = await client.execute('chronoshift-pro', 'create_schedule', params);
} catch (error) {
  if (error instanceof MCPConnectionError) {
    // Connection failed
    console.error('Connection error:', error.message);
    // Attempt reconnection
    await client.reconnect('chronoshift-pro');
  } else if (error instanceof MCPCommandError) {
    // Command execution failed
    console.error('Command failed:', error.message);
    // Log to persistent memory
    await logDecision({
      title: 'Command Failed',
      description: error.message,
      impact: 'MEDIUM'
    });
  } else {
    // Unknown error
    console.error('Unknown error:', error);
  }
}
```

Testing

Unit Tests

```
describe('MCP Client', () => {
  it('should connect to satellite app', async () => {
    const client = new MCPClient();
    await client.connect('test-app', 'http://localhost:3001');
    expect(client.isConnected('test-app')).toBe(true);
  });

  it('should execute command', async () => {
    const result = await client.execute('test-app', 'test_command', {});
    expect(result.success).toBe(true);
  });

  it('should handle connection errors', async () => {
    await expect(
      client.connect('invalid-app', 'http://invalid')
    ).rejects.toThrow(MCPConnectionError);
  });
});
```

Integration Tests

```
describe('MCP Integration', () => {
  it('should complete end-to-end workflow', async () => {
    // Test complete workflow with real apps
    const result = await processQuoteToProduction('quote-123');
    expect(result.success).toBe(true);
  });
});
```

Performance Optimization

Connection Pooling

Reuse connections instead of creating new ones:

```
const connectionPool = new MCPConnectionPool({
  maxConnections: 10,
  idleTimeout: 60000
});
```

Request Batching

Batch multiple commands into single request:

```
const results = await client.batch([
  { app: 'app1', command: 'cmd1', params: {} },
  { app: 'app2', command: 'cmd2', params: {} }
]);
```

Caching

Cache frequently queried data:

```
const cache = new MCPCache({
  ttl: 300000 // 5 minutes
});

const schedules = await cache.getOrFetch('schedules', async () => {
  return await client.query('chronoshift-pro', 'schedules', {});
});
```

Security

Authentication

All MCP connections require authentication:

```
await client.connect('chronoshift-pro', baseUrl, {
  auth: {
    type: 'bearer',
    token: process.env.MCP_AUTH_TOKEN
  }
});
```

Authorization

Commands are authorized based on capabilities:

```
// App declares capabilities during handshake
{
  "capabilities": {
    "execute": ["create_schedule", "update_schedule"],
    "query": ["schedules", "teams"],
    "subscribe": ["schedule_updated"]
  }
}
```

Encryption

All MCP communication uses TLS:

```
const client = new MCPClient({
  tls: {
    enabled: true,
    rejectUnauthorized: true
  }
});
```

Monitoring

Connection Health

```
// Monitor connection health
client.on('connection_lost', (appId) => {
  console.warn(`Lost connection to ${appId}`);
  // Attempt reconnection
});

client.on('connection_restored', (appId) => {
  console.log(`Restored connection to ${appId}`);
});
```

Performance Metrics

```
// Track command execution time
const metrics = client.getMetrics('chronoshift-pro');
console.log('Average response time:', metrics.avgResponseTime);
console.log('Success rate:', metrics.successRate);
```

Troubleshooting

Connection Fails

1. Check if satellite app is running
2. Verify network connectivity
3. Check firewall rules
4. Verify authentication token

Command Timeout

1. Increase timeout setting
2. Check satellite app performance
3. Verify command parameters
4. Check app logs for errors

Event Not Received

1. Verify subscription was successful
2. Check event filters
3. Verify WebSocket connection
4. Check app logs for event emission

Implementation Timeline: Week 2 (November 11-17, 2025)
Dependencies: App Registry, GitHub App credentials
Priority: HIGH - Enables full orchestration