readme.md for @push.rocks/smartipc

Rock-solid IPC for Node.js with zero dependencies

npm version TypeScript License: MIT

SmartIPC delivers bulletproof Inter-Process Communication for Node.js applications. Built for real-world production use, it handles all the edge cases that make IPC tricky - automatic reconnection, race conditions, heartbeat monitoring, and clean shutdowns. All with **zero external dependencies** and full TypeScript support.

□□ Why SmartIPC?

- **Zero Dependencies** - Pure Node.js implementation using native modules
- **Battle-tested Reliability** - Automatic reconnection, graceful degradation, and timeout handling
- **Type-Safe** - Full TypeScript support with generics for compile-time safety
- **CI/Test Ready** - Built-in helpers and race condition prevention for testing
- **Observable** - Real-time metrics, connection tracking, and health monitoring
- **Multiple Patterns** - Request/Response, Pub/Sub, and Fire-and-Forget messaging
- **Streaming Support** - Efficient, backpressure-aware streaming for large data and files

□□ Installation

```
npm install @push.rocks/smartipc
# or
pnpm add @push.rocks/smartipc
# or
yarn add @push.rocks/smartipc
```

□□ Quick Start

```
import { SmartIpc } from '@push.rocks/smartipc';

// Create a server
const server = SmartIpc.createServer({
  id: 'my-service',
  socketPath: '/tmp/my-service.sock',
  autoCleanupSocketFile: true  // Clean up stale sockets automatically
});

// Handle incoming messages
server.onMessage('greet', async (data, clientId) => {
  console.log(`Client ${clientId} says:`, data.message);
  return { response: `Hello ${data.name}!` };
});

// Start the server
await server.start({ readyWhen: 'accepting' });  // Wait until fully ready
console.log('Server is ready to accept connections! □');

// Create a client
const client = SmartIpc.createClient({
  id: 'my-service',
  socketPath: '/tmp/my-service.sock',
  connectRetry: {
    enabled: true,
    maxAttempts: 10
  }
});

// Connect with automatic retry
await client.connect();

// Send a request and get a response
const response = await client.request('greet', {
  name: 'World',
  message: 'Hi there!'
});
console.log('Server said:', response.response);  // "Hello World!"
```

□□ Core Concepts

Transport Types

SmartIPC supports multiple transport mechanisms, automatically selecting the best one for your platform:

```
// TCP Socket (cross-platform, network-capable)
const tcpServer = SmartIpc.createServer({
  id: 'tcp-service',
  host: 'localhost',
  port: 9876
});

// Unix Domain Socket (Linux/macOS, fastest local IPC)
const unixServer = SmartIpc.createServer({
  id: 'unix-service',
  socketPath: '/tmp/my-app.sock'
});

// Windows Named Pipe (Windows optimal)
// Automatically used on Windows when socketPath is provided
const windowsServer = SmartIpc.createServer({
  id: 'pipe-service',
  socketPath: '\\\\.\\pipe\\my-app-pipe'
});
```

Message Patterns

□□ Fire and Forget

Send messages without waiting for a response:

```
// Server
server.onMessage('log', (data, clientId) => {
  console.log(`[${clientId}] ${data.level}:`, data.message);
  // No return needed
```

```
});

// Client
await client.sendMessage('log', {
  level: 'info',
  message: 'User logged in',
  timestamp: Date.now()
});
```

□□ Request/Response

RPC-style communication with type safety:

```
interface UserRequest {
  userId: string;
  fields?: string[];
}

interface UserResponse {
  id: string;
  name: string;
  email?: string;
  createdAt: number;
}

// Server
server.onMessage<UserRequest, UserResponse>('getUser', async (data) => {
  const user = await db.getUser(data.userId);
  return {
    id: user.id,
    name: user.name,
    email: data.fields?.includes('email') ? user.email : undefined,
    createdAt: user.createdAt
  };
});

// Client - with timeout
const user = await client.request<UserRequest, UserResponse>(
  'getUser',
  { userId: '123', fields: ['email'] },
```

```
  { timeout: 5000 }
);
```

□□ Pub/Sub Pattern

Topic-based message broadcasting:

```
// Subscribers
const subscriber1 = SmartIpc.createClient({
  id: 'events-service',
  socketPath: '/tmp/events.sock'
});

await subscriber1.connect();
await subscriber1.subscribe('user.login', (data) => {
  console.log('User logged in:', data);
});

// Publisher
const publisher = SmartIpc.createClient({
  id: 'events-service',
  socketPath: '/tmp/events.sock'
});

await publisher.connect();
await publisher.publish('user.login', {
  userId: '123',
  ip: '192.168.1.1',
  timestamp: Date.now()
});
```

□□ Advanced Features

□□ Streaming Large Data & Files

SmartIPC supports efficient, backpressure-aware streaming of large payloads using chunked messages. Streams work both directions and emit a high-level `stream` event for consumption.

Client → Server streaming:

```
// Server side: receive stream
server.on('stream', async (info, readable) => {
  if (info.meta?.type === 'file') {
    console.log('Receiving file', info.meta.basename, 'from', info.clientId);
  }
  // Pipe to disk or process chunks
  await SmartIpc.pipeStreamToFile(readable, '/tmp/incoming.bin');
});

// Client side: send a stream
const readable = fs.createReadStream('/path/to/local.bin');
await client.sendStream(readable, {
  meta: { type: 'file', basename: 'local.bin' },
  chunkSize: 64 * 1024 // optional, defaults to 64k
});
```

Server → Client streaming:

```
client.on('stream', async (info, readable) => {
  console.log('Got stream from server', info.meta);
  await SmartIpc.pipeStreamToFile(readable, '/tmp/from-server.bin');
});

await server.sendStreamToClient(client.getClientId(), fs.createReadStream('/path/server.bin'),
{
  meta: { type: 'file', basename: 'server.bin' }
});
```

High-level helpers for files:

```
// Client → Server
await client.sendFile('/path/to/bigfile.iso');

// Server → Client
await server.sendFileToClient(clientId, '/path/to/backup.tar');

// Save an incoming stream to a file (both sides)
server.on('stream', async (info, readable) => {
```

```
  await SmartIpc.pipeStreamToFile(readable, '/data/uploaded/' + info.meta?.basename);
});
```

Events & metadata:

- `channel/server/client` emit `stream` with `(info, readable)`
- `info` contains: `streamId`, `meta` (your metadata, e.g., filename/size), `headers`, and `clientId` (if available)

API summary:

- Client: `sendStream(readable, opts)`, `sendFile(filePath, opts)`, `cancelOutgoingStream(id)`, `cancelIncomingStream(id)`
- Server: `sendStreamToClient(clientId, readable, opts)`, `sendFileToClient(clientId, filePath, opts)`, `cancelIncomingStreamFromClient(clientId, id)`, `cancelOutgoingStreamToClient(clientId, id)`
- Utility: `SmartIpc.pipeStreamToFile(readable, filePath)`

Concurrency and cancelation:

```
// Limit concurrent streams per connection
const server = SmartIpc.createServer({
  id: 'svc', socketPath: '/tmp/svc.sock', maxConcurrentStreams: 2
});

// Cancel a stream from the receiver side
server.on('stream', (info, readable) => {
  if (info.meta?.shouldCancel) {
    (server as any).primaryChannel.cancelIncomingStream(info.streamId, { clientId:
info.clientId });
  }
});
```

Notes:

- Streaming uses chunked messages under the hood and respects socket backpressure.
- Include `meta` to share context like filename/size; it’s delivered with the `stream` event.
- Configure `maxConcurrentStreams` (default: 32) to guard resources.

□□ Server Readiness Detection

Eliminate race conditions in tests and production:

```
const server = SmartIpc.createServer({
  id: 'my-service',
  socketPath: '/tmp/my-service.sock',
  autoCleanupSocketFile: true
});

// Option 1: Wait for full readiness
await server.start({ readyWhen: 'accepting' });
// Server is now FULLY ready to accept connections

// Option 2: Use ready event
server.on('ready', () => {
  console.log('Server is ready!');
  startClients();
});

await server.start();

// Option 3: Check readiness state
if (server.getIsReady()) {
  console.log('Ready to rock! □□');
}
```

□□ Smart Connection Retry

Never lose messages due to temporary connection issues:

```
const client = SmartIpc.createClient({
  id: 'resilient-client',
  socketPath: '/tmp/service.sock',
  connectRetry: {
    enabled: true,
    initialDelay: 100,      // Start with 100ms
    maxDelay: 1500,         // Cap at 1.5 seconds
    maxAttempts: 20,        // Try 20 times
    totalTimeout: 15000     // Give up after 15 seconds total
  },
  registerTimeoutMs: 8000   // Registration handshake timeout
});
```

```
// Will retry automatically if server isn't ready yet
await client.connect({
  waitForReady: true,       // Wait for server to exist
  waitTimeout: 10000        // Wait up to 10 seconds
});
```

□□ Client-Only Mode (No Auto-Start)

In some setups (CLI + long-running daemon), you want clients to fail fast when no server is available, rather than implicitly becoming the server. Enable client-only mode to prevent the “client becomes server” fallback for Unix domain sockets and Windows named pipes.

```
// Strict client that never auto-starts a server on connect failure
const client = SmartIpc.createClient({
  id: 'my-service',
  socketPath: '/tmp/my-service.sock',
  clientId: 'my-cli',
  clientOnly: true,                 // NEW: disable auto-start fallback
  connectRetry: { enabled: false }  // optional: fail fast
});

try {
  await client.connect();
} catch (err) {
  // With clientOnly: true, errors become descriptive
  // e.g. "Server not available (ENOENT); clientOnly prevents auto-start"
  console.error(err.message);
}
```

- Default: `clientOnly` is `false` to preserve backward compatibility.
- Env override: set `SMARTIPC_CLIENT_ONLY=1` to enforce client-only behavior without code changes.
- Note: `SmartIpc.waitForServer()` internally uses `clientOnly: true` for safe probing.

□□ Graceful Heartbeat Monitoring

Keep connections alive without crashing on timeouts:

```
const server = SmartIpc.createServer({
  id: 'monitored-service',
  socketPath: '/tmp/monitored.sock',
  heartbeat: true,
  heartbeatInterval: 3000,
  heartbeatTimeout: 10000,
  heartbeatInitialGracePeriodMs: 5000,    // Grace period for startup
  heartbeatThrowOnTimeout: false           // Emit event instead of throwing
});

server.on('heartbeatTimeout', (clientId) => {
  console.log(`Client ${clientId} heartbeat timeout - will handle gracefully`);
});

// Client configuration
const client = SmartIpc.createClient({
  id: 'monitored-service',
  socketPath: '/tmp/monitored.sock',
  heartbeat: true,
  heartbeatInterval: 3000,
  heartbeatTimeout: 10000,
  heartbeatInitialGracePeriodMs: 5000,
  heartbeatThrowOnTimeout: false
});

client.on('heartbeatTimeout', () => {
  console.log('Heartbeat timeout detected, reconnecting...');
  // Handle reconnection logic
});
```

□□ Automatic Socket Cleanup

Never worry about stale socket files:

```
const server = SmartIpc.createServer({
  id: 'clean-service',
  socketPath: '/tmp/service.sock',
  autoCleanupSocketFile: true,    // Remove stale socket on start
  socketMode: 0o600               // Set socket permissions (Unix only)
```

```
});

// Socket file will be cleaned up automatically on start
await server.start();
```

□□ Real-time Metrics

Monitor your IPC performance:

```
// Server stats
const serverStats = server.getStats();
console.log({
  isRunning: serverStats.isRunning,
  connectedClients: serverStats.connectedClients,
  totalConnections: serverStats.totalConnections,
  metrics: {
    messagesSent: serverStats.metrics.messagesSent,
    messagesReceived: serverStats.metrics.messagesReceived,
    errors: serverStats.metrics.errors
  }
});

// Client stats
const clientStats = client.getStats();
console.log({
  connected: clientStats.connected,
  reconnectAttempts: clientStats.reconnectAttempts,
  metrics: clientStats.metrics
});

// Get specific client info
const clientInfo = server.getClientInfo('client-123');
console.log({
  connectedAt: new Date(clientInfo.connectedAt),
  lastActivity: new Date(clientInfo.lastActivity),
  metadata: clientInfo.metadata
});
```

Broadcasting

Send messages to multiple clients:

```
// Broadcast to all connected clients
await server.broadcast('announcement', {
  message: 'Server will restart in 5 minutes',
  severity: 'warning'
});

// Send to specific clients
await server.broadcastTo(
  ['client-1', 'client-2'],
  'private-message',
  { content: 'This is just for you two' }
);

// Send to one client
await server.sendToClient('client-1', 'direct', {
  data: 'Personal message'
});
```

Testing Utilities

SmartIPC includes powerful helpers for testing:

Wait for Server

```
import { SmartIpc } from '@push.rocks/smartipc';

// Start your server in another process
const serverProcess = spawn('node', ['server.js']);

// Wait for it to be ready
await SmartIpc.waitForServer({
  socketPath: '/tmp/test.sock',
```

```
  timeoutMs: 10000
});

// Now safe to connect clients
const client = SmartIpc.createClient({
  id: 'test-client',
  socketPath: '/tmp/test.sock'
});
await client.connect();
```

Spawn and Connect

```
// Helper that spawns a server and connects a client
const { client, serverProcess } = await SmartIpc.spawnAndConnect({
  serverScript: './server.js',
  socketPath: '/tmp/test.sock',
  clientId: 'test-client',
  connectRetry: {
    enabled: true,
    maxAttempts: 10
  }
});

// Use the client
const response = await client.request('ping', {});

// Cleanup
await client.disconnect();
serverProcess.kill();
```

□□ Event Handling

SmartIPC provides comprehensive event emitters:

```
// Server events
server.on('start', () => console.log('Server started'));
```

```
server.on('ready', () => console.log('Server ready for connections'));
server.on('clientConnect', (clientId, metadata) => {
  console.log(`Client ${clientId} connected with metadata:`, metadata);
});
server.on('clientDisconnect', (clientId) => {
  console.log(`Client ${clientId} disconnected`);
});
server.on('error', (error, clientId) => {
  console.error(`Error from ${clientId}:`, error);
});

// Client events
client.on('connect', () => console.log('Connected to server'));
client.on('disconnect', () => console.log('Disconnected from server'));
client.on('reconnecting', (attempt) => {
  console.log(`Reconnection attempt ${attempt}`);
});
client.on('error', (error) => {
  console.error('Client error:', error);
});
client.on('heartbeatTimeout', (error) => {
  console.warn('Heartbeat timeout:', error);
});
```

□□ Error Handling

Robust error handling with detailed error information:

```
// Client-side error handling
try {
  const response = await client.request('riskyOperation', data, {
    timeout: 5000
  });
} catch (error) {
  if (error.message.includes('timeout')) {
    console.error('Request timed out');
  } else if (error.message.includes('Failed to register')) {
```

```
    console.error('Could not register with server');
  } else {
    console.error('Unknown error:', error);
  }
}

// Server-side error boundaries
server.onMessage('process', async (data, clientId) => {
  try {
    return await riskyProcessing(data);
  } catch (error) {
    console.error(`Processing failed for ${clientId}:`, error);
    throw error;  // Will be sent back to client as error
  }
});
```

□□ Architecture

SmartIPC uses a clean, layered architecture:

```
┌─────────────────────────────────────────────┐
│          Your Application                   │
│        (Business logic)                     │
└─────────────────────────────────────────────┘
                    ↕
┌─────────────────────────────────────────────┐
│         IpcServer / IpcClient               │
│   (High-level API, Message routing)         │
└─────────────────────────────────────────────┘
                    ↕
┌─────────────────────────────────────────────┐
│            IpcChannel                       │
│  (Connection management, Heartbeat,         │
│   Reconnection, Request/Response)           │
└─────────────────────────────────────────────┘
                    ↕
┌─────────────────────────────────────────────┐
```

```
│          Transport Layer                │
│  (TCP, Unix Socket, Named Pipe)         │
│     (Framing, buffering, I/O)           │
└─────────────────────────────────────────┘
```

□□ Common Use Cases

Microservices Communication

```
// API Gateway
const gateway = SmartIpc.createServer({
  id: 'api-gateway',
  socketPath: '/tmp/gateway.sock'
});

// User Service
const userService = SmartIpc.createClient({
  id: 'api-gateway',
  socketPath: '/tmp/gateway.sock',
  clientId: 'user-service'
});

// Order Service
const orderService = SmartIpc.createClient({
  id: 'api-gateway',
  socketPath: '/tmp/gateway.sock',
  clientId: 'order-service'
});
```

Worker Process Management

```
// Main process
const server = SmartIpc.createServer({
  id: 'main',
  socketPath: '/tmp/workers.sock'
```

```
});

server.onMessage('job-complete', (result, workerId) => {
  console.log(`Worker ${workerId} completed job:`, result);
});

// Worker process
const worker = SmartIpc.createClient({
  id: 'main',
  socketPath: '/tmp/workers.sock',
  clientId: `worker-${process.pid}`
});

await worker.sendMessage('job-complete', {
  jobId: '123',
  result: processedData
});
```

Real-time Event Distribution

```
// Event bus
const eventBus = SmartIpc.createServer({
  id: 'event-bus',
  socketPath: '/tmp/events.sock'
});

// Services subscribe to events
const analyticsService = SmartIpc.createClient({
  id: 'event-bus',
  socketPath: '/tmp/events.sock'
});

await analyticsService.subscribe('user.*', (event) => {
  trackEvent(event);
});
```

□□ Performance

SmartIPC is optimized for high throughput and low latency:

Transport	Messages/sec	Avg Latency	Use Case
Unix Socket	150,000+	< 0.1ms	Local high-performance IPC (Linux/macOS)
Named Pipe	120,000+	< 0.15ms	Windows local IPC
TCP (localhost)	100,000+	< 0.2ms	Local network-capable IPC
TCP (network)	50,000+	< 1ms	Distributed systems

- **Memory efficient**: Streaming support for large payloads
- **CPU efficient**: Event-driven, non-blocking I/O

□□ Requirements

- Node.js >= 14.x
- TypeScript >= 4.x (for development)
- Unix-like OS (Linux, macOS) or Windows

License and Legal Information

Trademarks

Company Information

Task Venture Capital GmbH
Registered at District court Bremen HRB 35230 HB, Germany

For any legal inquiries or if you require further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #3
Created 2026-03-28 11:10:43 UTC by foss.global Team
Updated 2026-03-28 12:17:32 UTC by foss.global Team