kotlinx-rpc Internship M. Kozyrev
Design Proposal JetBrains
Category: Informational January 2026

Design Proposal: JSON-RPC 2.0 Protocol for kotlinx-rpc

Abstract

This document specifies a design proposal for implementing the JSON-RPC 2.0 protocol within the kotlinx-rpc library as a part of dedicated JetBrains RPC development research internship. This proposal defines protocol message formats, API structures, error handling mechanisms and integration patterns required to support JSON-RPC 2.0 as a protocol option alongside kRPC and gRPC in kotlinx-rpc library.

Status of This Memo

This document is a design proposal for the kotlinx-rpc library maintained by JetBrains. It is published for review and discussion purposes. Distribution is unlimited.

Copyright Notice

Copyright (c) 2026 JetBrains s.r.o. All rights reserved.

Table of Contents

1. Introduction

The kotlinx-rpc library provides a unified framework for Remote Procedure Call(RPC) implementations in the Kotlin and KMP ecosystem. This document proposes JSON-RPC 2.0 protocol support to complement the existing kRPC and gRPC options.

1.1. Motivation

JSON-RPC 2.0 is a lightweight, transport-agnostic RPC protocol that uses JSON for data encoding. Its widespread adoption makes it essential for many modern use cases, including:

AI Agent Integration:
: Model Context Protocol (MCP), developed by Anthropic and adopted by OpenAI, Google DeepMind, and Microsoft, uses JSON-RPC 2.0 as communication layer. MCP enables AI applications to connect with external tools, data sources, and services. With Kotlin's growing role in Android AI applications and server-side AI infrastructure, native MCP support via JSON-RPC is growing in importance.

Blockchain and Web3:
: Ethereum, Bitcoin, Solana all use JSON-RPC 2.0 as their standard API protocol. The Kotlin ecosystem already has libraries like solana-mobile/rpc-core demonstrating demand for multiplatform JSON-RPC in blockchain contexts.

Language Server Protocol:
: IDE tooling and editor extensions communicate via JSON-RPC. IntelliJ plugins and Kotlin language servers benefit from native JSON-RPC support for LSP implementation.

Web Application Development:
: JavaScript clients communicating with Kotlin backends using human-readable messages for debugging and monitoring.

OpenRPC Specifications:
: API contracts defined in OpenRPC format require JSON-RPC runtime support for service discovery and documentation.

Microservices and Interoperability:
: Polyglot environments where JSON is the common data format benefit from standardized RPC semantics over raw REST.

Serverless Functions:
: AWS Lambda, Google Cloud Functions, and Azure Functions also can expose JSON-RPC endpoints.

1.2. Design Goals

This proposal addresses feedback from the Kotlin community and direct internship requirements listed in the official JetBrains internships webpage <https://internship.jetbrains.com/projects/1718>

1.3. Scope

This proposal covers:

- Complete JSON-RPC 2.0 specification implementation
- Heterogeneous batch request support with type-safe results
- Kotlin-idiomatic API with coroutines and Flow
- Extensible transport abstraction (HTTP, WebSocket, stdio, custom transports for MCP, serverless, etc.)
- Extensible system extension framework (rpc.* methods)
- Full Kotlin Multiplatform (KMP) support

This proposal explicitly excludes:

- Streaming beyond JSON-RPC 2.0 (s. section 21)
- Binary encoding alternatives

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

Client	The origin of Request objects and the handler of Response objects.
Server	The origin of Response objects and the handler of Request objects.
Request	A call to a remote method, expecting a response.
Notification	A call to a remote method without expecting a response.
Batch	An array of Request and/or Notification objects.
Transport	The communication layer (HTTP, WebSocket, stdio etc.)
System Extension	A method with name beginning "rpc." reserved for protocol-level functionality.

4. JSON-RPC 2.0 Protocol Overview

JSON-RPC 2.0 [JSONRPC20] is a stateless, light-weight remote procedure call protocol using JSON [RFC8259] as the data format.

4.1. Request Object

A Request object contains:

jsonrpc MUST be exactly "2.0".

method String name of the method. Names beginning with "rpc." are reserved for system extensions.

params MAY be omitted. If present, MUST be Array (by-position) or Object (by-name). Parameter names are case-sensitive.

id MUST be String, Number, or Null if present. The specification RECOMMENDS against Null or fractional numbers. Implementations MUST accept fractional numbers but SHOULD NOT generate them (truncate to Long).

Example:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

4.2. Response Object

jsonrpc MUST be "2.0".

result REQUIRED on success. MUST NOT exist on error.

error REQUIRED on error. MUST NOT exist on success.

id MUST match the Request id. MUST be Null for parse errors where the Request id cannot be determined.

A Response MUST contain exactly one of result or error.

4.3. Notification

A Notification is a Request without an "id" member.
The Server MUST NOT reply to a notification.

4.4. Batch Requests

Important batch semantics:

- Response Array MAY be returned in any order.
- Server MAY process requests concurrently.
- All-Notification batch MUST NOT return any response (HTTP 204 No Content).
- Empty Array MUST result in Invalid Request error.

4.5. Error Codes

Reserved error codes (-32768 to -32000):

Code	Message	Description
-32700	Parse error	Invalid JSON received
-32600	Invalid Request	Not a valid Request
-32601	Method not found	Method does not exist
-32602	Invalid params	Invalid method parameters
-32603	Internal error	Internal JSON-RPC error
-32099 to -32000	Server error	Implementation-defined server errors in this range

Application error codes MUST be outside the reserved range.

Recommended ranges:

Range	Category
1000-1099	Authentication/Authorization
1100-1199	Validation errors
1200-1299	Resource errors
1300-1399	Business logic errors

5. Comparative Analysis

5.1. Existing Kotlin JSON-RPC Related Libraries

xqt-kotlinx-json-rpc:
A multiplatform JSON-RPC 2.0 library by Reece H. Dunn.
Provides basic protocol support but lacks integration with service interfaces and code generation. Our design builds on similar multiplatform foundations but adds type-safe service stubs.

solana-mobile/rpc-core:
Kotlin multiplatform JSON-RPC for Solana blockchain.
Uses kotlinx.serialization with transport abstraction.
Demonstrates demand for JSON-RPC in cryptocurrency/Web3 contexts.

jsonrpc-kotlin-client (Reedyuk):
Simple KMP client for JSON-RPC. Limited to client-side only.
Suggested design provides both client and server implementations.

5.2. Comparison with Other Technologies

jsonrpc4j (Java):
Uses annotations and manual registration. Our design leverages @Rpc annotation and code generation for less boilerplate.

tRPC (TypeScript):
End-to-end type safety via TypeScript inference. Kotlin achieves similar safety via kotlinx.serialization and generated proxies.

gRPC:
Protocol Buffers with HTTP/2. JSON-RPC trades performance for simplicity and human-readability. Both use interceptors.

5.3. Design Rationale

(1) Sealed class for RpcId: Exhaustive checks, compile-time safety for polymorphic id(it is specified it can be String, Number, Null in JSON-RPC 2.0 specs).

(2) DSL builders: Follows Ktor and kotlinx.html patterns.
See the respective libraries for more details.

(3) Suspend functions only: Avoids blocking, enables structured concurrency. Non-suspend methods are prohibited.

(4) Transport abstraction: HTTP, WebSocket, stdio, serverless handlers, custom implementations etc.

(5) Extensible system extensions: Plugin architecture for rpc.* methods defined as JSON-RPC extensions in JSON-RPC 2.0 specification.

6. Library Architecture

6.1. Module Structure

kotlinx-rpc-jsonrpc-core
Core message types, serializers, multiplatform abstractions
Dependencies: kotlinx-rpc-core, kotlinx-serialization-json

kotlinx-rpc-jsonrpc-client
Client implementation
Dependencies: kotlinx-rpc-jsonrpc-core, kotlinx-coroutines-core

kotlinx-rpc-jsonrpc-server
Server implementation
Dependencies: kotlinx-rpc-jsonrpc-core, kotlinx-coroutines-core

kotlinx-rpc-jsonrpc-transport-http
HTTP transport (framework-agnostic interface)
Dependencies: kotlinx-rpc-jsonrpc-core

kotlinx-rpc-jsonrpc-transport-ktor
Ktor integration for HTTP and WebSocket
Dependencies: kotlinx-rpc-jsonrpc-transport-http, ktor-*

kotlinx-rpc-jsonrpc-transport-stdio
Standard I/O transport for MCP and LSP
Dependencies: kotlinx-rpc-jsonrpc-core

kotlinx-rpc-jsonrpc-extensions
System extension plugins (rpc.discover, etc.)
Dependencies: kotlinx-rpc-jsonrpc-core

6.2. Multiplatform Abstractions

For example a module like ConcurrentHashMap is not available on all platforms.
The library provides a multiplatform ConcurrentMap abstraction:

```
// commonMain
internal expect class ConcurrentMap<K, V>() {
    operator fun get(key: K): V?
    operator fun set(key: K, value: V)
    fun remove(key: K): V?
    fun putIfAbsent(key: K, value: V): V?
    fun computeIfAbsent(key: K, compute: (K) -> V): V
    val keys: Set<K>
    val values: Collection<V>
}

// jvmMain
internal actual class ConcurrentMap<K, V> actual constructor() {
    private val delegate = ConcurrentHashMap<K, V>()
    actual operator fun get(key: K) = delegate[key]
    actual operator fun set(key: K, value: V) {
        delegate[key] = value
```

```
        }
        actual fun remove(key: K) = delegate.remove(key)
        actual fun putIfAbsent(key: K, value: V) =
            delegate.putIfAbsent(key, value)
        actual fun computeIfAbsent(key: K, compute: (K) -> V) =
            delegate.computeIfAbsent(key, compute)
        actual val keys: Set<K> get() = delegate.keys
        actual val values: Collection<V> get() = delegate.values
    }

    // nativeMain / jsMain
    internal actual class ConcurrentMap<K, V> actual constructor() {
        private val delegate = mutableMapOf<K, V>()
        private val mutex = Mutex()
        actual operator fun get(key: K) = delegate[key]
        // synchronization via mutex for thread safety
    }
```

Similar approach MUST be applied everywhere throughout the library.

6.3. Core Abstractions

DSL marker annotation:

```
@DslMarker
@Target(AnnotationTarget.CLASS, AnnotationTarget.TYPE)
public annotation class JsonRpcDsl
```

Value classes for type safety:

```
@JvmInline
public value class MethodName(val value: String) {
    init {
        require(value.isNotBlank()) { "Method name must not be blank" }
    }
    val isSystemExtension: Boolean get() = value.startsWith("rpc.")
}
```

Primary client interface:

```
public interface JsonRpcClient : RpcClient {
    suspend fun <T> call(
        method: String,
        params: Any? = null,
        resultType: KType
    ): T
    suspend fun notify(method: String, params: Any? = null)
    suspend fun batch(
        block: suspend JsonRpcBatchBuilder.() -> Unit
    ): JsonRpcBatchResults
}
public suspend inline fun <reified T> JsonRpcClient.call(
    method: String,
    params: Any? = null
```

```
    ): T = call(method, params, typeOf<T>())
```

6.4. withService Implementation (Suspend-Only Enforcement)

Service interface methods MUST be suspend functions.
This is enforced at proxy creation time to avoid runtime type mismatches:

```
    // JVM implementation
    public inline fun <reified T : Any> JsonRpcClient.withService(): T {
        val serviceClass = T::class
        validateServiceInterface(serviceClass)  // Throws if non-suspend
        return createProxy(serviceClass)
    }

    private fun validateServiceInterface(klass: KClass<*>) {
        klass.memberFunctions
            .filter { !it.isAbstract.not() }
            .forEach { method ->
                require(method.isSuspend) {
                    "Service method " + method.name + " must be suspend. " +
                    "Non-suspend methods are not supported in JSON-RPC " +
                    "services to ensure proper coroutine integration."
                }
            }
    }
```

On non-JVM platforms, withService() is unavailable.
It is necessary to use generated stubs via code
generation instead (see Section 17).

7. Message Model Design

7.1. Request ID Types

The sealed class hierarchy represents polymorphic id types:

```
@Serializable(with = RpcIdSerializer::class)
public sealed class RpcId {
    public data object NullId : RpcId() {
        override fun toString(): String = "null"
    }
    public data class StringId(val value: String) : RpcId() {
        override fun toString(): String = value
    }
    public data class NumberId(val value: Long) : RpcId() {
        override fun toString(): String = value.toString()
    }
    public companion object {
        public fun of(value: String): RpcId = StringId(value)
        public fun of(value: Long): RpcId = NumberId(value)
        public fun of(value: Int): RpcId = NumberId(value.toLong())
        public fun ofNull(): RpcId = NullId
    }
}
```

7.2. Fractional ID Handling

The specification RECOMMENDS against fractional numbers, but some clients may send them(e.g., 1.0 instead of 1). The serializer accepts fractional numbers and truncates to Long:

```
internal object RpcIdSerializer : KSerializer<RpcId> {
    override val descriptor = buildClassSerialDescriptor("RpcId")
    override fun serialize(encoder: Encoder, value: RpcId) {
        val jsonEncoder = encoder as? JsonEncoder
            ?: error("RpcId requires JSON format")
        val element = when (value) {
            is RpcId.NullId -> JsonNull
            is RpcId.StringId -> JsonPrimitive(value.value)
            is RpcId.NumberId -> JsonPrimitive(value.value)
        }
        jsonEncoder.encodeJsonElement(element)
    }
    override fun deserialize(decoder: Decoder): RpcId {
        val jsonDecoder = decoder as? JsonDecoder
            ?: error("RpcId requires JSON format")
        return when (val elem = jsonDecoder.decodeJsonElement()) {
            is JsonNull -> RpcId.NullId
            is JsonPrimitive -> when {
                elem.isString -> RpcId.StringId(elem.content)
                elem.longOrNull != null -> RpcId.NumberId(elem.long)
                elem.doubleOrNull != null -> {
                    // Accept fractional, truncate to Long.
```

```
                        // 1.9 and 1.1 both become 1, which could
                        // cause ID collision if client uses fractional ids.
                        // This is documented behavior.
                        RpcId.NumberId(elem.double.toLong())
                    }
                    else -> error("Invalid RpcId: " + elem)
                }
                else -> error("RpcId must be null or primitive")
            }
        }
    }
```

IMPORTANT: Truncation can cause ID collisions (1.9 and 1.1 both become 1). This is acceptable because the spec recommends against fractional IDs. Clients using fractional IDs do so at their risk.

7.3. Request Representation

```
    @Serializable
    public data class JsonRpcRequest(
        @SerialName("jsonrpc")
        val version: String = JSONRPC_VERSION,
        @SerialName("method")
        val method: String,
        @SerialName("params")
        val params: JsonElement? = null,
        @SerialName("id")
        val id: RpcId? = null
    ) {
        init {
            require(version == JSONRPC_VERSION) {
                "JSON-RPC version must be '2.0'"
            }
            require(method.isNotBlank()) { "Method name must not be blank" }
        }
        val isNotification: Boolean get() = id == null
        val isSystemExtension: Boolean get() = method.startsWith("rpc.")
        public companion object {
            public const val JSONRPC_VERSION: String = "2.0"
        }
    }
```

7.4. Response Representation

```
    @Serializable
    public data class JsonRpcResponse(
        @SerialName("jsonrpc")
        val version: String = JSONRPC_VERSION,
        @SerialName("result")
        val result: JsonElement? = null,
        @SerialName("error")
        val error: JsonRpcErrorObject? = null,
        @SerialName("id")
        val id: RpcId?
```

```
) {
    init {
        require(version == JSONRPC_VERSION)
        require((result != null) xor (error != null)) {
            "Must have result xor error"
        }
    }
    val isSuccess: Boolean get() = error == null
    public companion object {
        public const val JSONRPC_VERSION: String = "2.0"
        public fun success(id: RpcId?, result: JsonElement) =
            JsonRpcResponse(id = id, result = result)
        public fun error(id: RpcId?, error: JsonRpcErrorObject) =
            JsonRpcResponse(id = id, error = error)
        public fun parseError(): JsonRpcResponse =
            JsonRpcResponse(id = null, error = JsonRpcErrorObject.parseError())
    }
}
```

7.5. Error Object Design

```
@Serializable
public data class JsonRpcErrorObject(
    @SerialName("code") val code: Int,
    @SerialName("message") val message: String,
    @SerialName("data") val data: JsonElement? = null
) {
    public companion object {
        public const val PARSE_ERROR: Int = -32700
        public const val INVALID_REQUEST: Int = -32600
        public const val METHOD_NOT_FOUND: Int = -32601
        public const val INVALID_PARAMS: Int = -32602
        public const val INTERNAL_ERROR: Int = -32603

        // Server error range
        public val SERVER_ERROR_RANGE: IntRange = -32099..-32000

        // Standard factory methods
        public fun parseError(data: JsonElement? = null) =
            JsonRpcErrorObject(PARSE_ERROR, "Parse error", data)
        public fun invalidRequest(data: JsonElement? = null) =
            JsonRpcErrorObject(INVALID_REQUEST, "Invalid Request", data)
        public fun methodNotFound() =
            JsonRpcErrorObject(METHOD_NOT_FOUND, "Method not found")
        public fun invalidParams(details: String? = null) =
            JsonRpcErrorObject(INVALID_PARAMS, details ?: "Invalid params")
        public fun internalError(data: JsonElement? = null) =
            JsonRpcErrorObject(INTERNAL_ERROR, "Internal error", data)

        // Extended server errors
        public fun requestCancelled() =
```

```
                JsonRpcErrorObject(-32001, "Request cancelled")
            public fun serverBusy() =
                JsonRpcErrorObject(-32002, "Server busy")
            public fun batchTooLarge(limit: Int) =
                JsonRpcErrorObject(-32003, "Batch too large, limit: " + limit)
            public fun requestTooLarge(limit: Long) =
                JsonRpcErrorObject(-32004, "Request too large, limit: " + limit)
            public fun timeout() =
                JsonRpcErrorObject(-32005, "Request timeout")
        }
    }
```

8. Client API Design

8.1. Client Configuration

```
public fun JsonRpcClient(
    transport: JsonRpcTransport,
    scope: CoroutineScope,
    block: JsonRpcClientConfig.() -> Unit = {}
): JsonRpcClient = DefaultJsonRpcClient(transport, scope, block)

@JsonRpcDsl
public class JsonRpcClientConfig {
    public var idGenerator: RpcIdGenerator = RpcIdGenerator.Sequential()
    public var requestTimeout: Duration = 30.seconds
    public var json: Json = Json {
        ignoreUnknownKeys = true
        encodeDefaults = true
    }

    // Interceptors
    private val interceptors = mutableListOf<JsonRpcInterceptor>()
    public fun install(interceptor: JsonRpcInterceptor) {
        interceptors.add(interceptor)
    }

    // Serialization customization
    public fun serialization(block: JsonRpcSerializationConfig.() -> Unit)
    // Handler for server-initiated requests (bidirectional)
    public var onServerRequest: (suspend (JsonRpcRequest) -> JsonRpcResponse)? =
        null
}
```

8.2. Request Timeout Enforcement

The requestTimeout MUST be applied to every call. This is done internally, not via an optional interceptor:

```
internal class DefaultJsonRpcClient(
    private val transport: JsonRpcTransport,
    private val scope: CoroutineScope,
    block: JsonRpcClientConfig.() -> Unit
) : JsonRpcClient {
    private val config = JsonRpcClientConfig().apply(block)
    private val pending = ConcurrentMap<RpcId, CompletableDeferred<String>>()
    override suspend fun <T> call(
        method: String,
        params: Any?,
        resultType: KType
    ): T {
        val id = config.idGenerator.next()
        val request = JsonRpcRequest(method = method, params = encode(params), id = id)
        val deferred = CompletableDeferred<String>()
        pending[id] = deferred
        try {
```

```
                // TIMEOUT ENFORCEMENT: Always wrap in withTimeout
                return withTimeout(config.requestTimeout) {
                    val responseJson = transport.send(json.encodeToString(request))
                        ?: throw JsonRpcException("No response for request")
                    val response = json.decodeFromString<JsonRpcResponse>(responseJson)
                    if (response.error != null) {
                        throw response.error.toException()
                    }
                    json.decodeFromJsonElement(resultType, response.result!!)
                }
            } catch (e: TimeoutCancellationException) {
                throw JsonRpcTimeoutException(
                    "Request timed out after " + config.requestTimeout,
                    cause = e
                )
            } finally {
                pending.remove(id)
            }
        }
    }
```

8.3. Parse Error Handling (Null ID Responses)

When the server returns a parse error with id=null, the client cannot correlate it to a specific pending request. The library handles this by failing all pending requests:

```
    private suspend fun handleIncomingMessage(text: String) {
        val response = try {
            json.decodeFromString<JsonRpcResponse>(text)
        } catch (e: Exception) {
            // Malformed response - ignore
            return
        }
        // Handle null-id error responses (parse errors)
        if (response.id == null && response.error != null) {
            // Server sent an error but couldn't determine request ID.
            // This typically means our request was malformed.
            // Fail all pending requests with this error.
            val error = response.error.toException()
            pending.values.forEach { it.completeExceptionally(error) }
            pending.clear()
            return
        }

        // Normal response correlation
        val id = response.id ?: return
        val deferred = pending[id]
        if (deferred != null) {
            deferred.complete(text)
        } else {
            // Response for unknown request - possible race or server bug
```

```
            // Log any decided kind of warning but don't fail
        }
    }
```

8.4. Heterogeneous Batch API

The batch API supports calls with different return types using type-safe handles for result retrieval:

```
    public class JsonRpcBatchResults internal constructor(
        private val results: Map<RpcId, JsonRpcBatchResult<*>>
    ) {
        @Suppress("UNCHECKED_CAST")
        public fun <T> get(handle: BatchCallHandle<T>): JsonRpcBatchResult<T> =
            results[handle.id] as? JsonRpcBatchResult<T>
                ?: JsonRpcBatchResult.Error(JsonRpcErrorObject.internalError())
    }

    public sealed class JsonRpcBatchResult<out T> {
        public data class Success<T>(val value: T) : JsonRpcBatchResult<T>()
        public data class Error(val error: JsonRpcErrorObject) :
            JsonRpcBatchResult<Nothing>()
    }

    public class BatchCallHandle<T> internal constructor(
        internal val id: RpcId,
        internal val resultType: KType
    )
```

8.5. Batch Builder with Response Correlation

The batch builder returns handles that can be used to retrieve results, regardless of server response ordering:

```
    @JsonRpcDsl
    public class JsonRpcBatchBuilder internal constructor(
        private val idGenerator: RpcIdGenerator
    ) {
        internal val entries = mutableListOf<BatchEntry>()

        public inline fun <reified T> call(
            method: String,
            params: Any? = null
        ): BatchCallHandle<T> {
            val id = idGenerator.next()
            val entry = BatchEntry.Call(id, method, params, typeOf<T>())
            entries.add(entry)
            return BatchCallHandle(id, typeOf<T>())
        }

        public fun notify(method: String, params: Any? = null) {
            entries.add(BatchEntry.Notification(method, params))
        }
    }
```

Usage with heterogeneous types:

```
val results = client.batch {
    val userHandle = call<User>("getUser", mapOf("id" to 1))
    val countHandle = call<Int>("getUserCount")
    val nameHandle = call<String>("getServerName")
    notify("logAccess", mapOf("action" to "batch"))
}

// Retrieve results by handle, typesafe
val user: JsonRpcBatchResult<User> = results.get(userHandle)
val count: JsonRpcBatchResult<Int> = results.get(countHandle)
val name: JsonRpcBatchResult<String> = results.get(nameHandle)

// Process results
when (user) {
    is JsonRpcBatchResult.Success -> println("User: " + user.value)
    is JsonRpcBatchResult.Error -> println("Error: " + user.error)
}
```

8.6. Client-Side Server Request Handling

For bidirectional transports, the client can handle requests initiated from server. If no handler is configured, client MUST respond with Method not found to prevent the server from waiting:

```
init {
    scope.launch {
        transport.receive().collect { message ->
            handleServerMessage(message)
        }
    }
}
private suspend fun handleServerMessage(text: String) {
    val request = try {
        json.decodeFromString<JsonRpcRequest>(text)
    } catch (e: Exception) {
        return  // Invalid request, ignore
    }

    // It's a request from server, we need to respond
    if (request.id != null) {
        val response = config.onServerRequest?.invoke(request)
            ?: JsonRpcResponse.error(
                request.id,
                JsonRpcErrorObject.methodNotFound()
            )
        transport.send(json.encodeToString(response))
    }

    // If notification, just ignore if no handler
}
```

9. Server API Design

9.1. Server Configuration

```
public fun JsonRpcServer(
    scope: CoroutineScope,
    block: JsonRpcServerConfig.() -> Unit = {}
): JsonRpcServer = DefaultJsonRpcServer(scope, block)
@JsonRpcDsl
public class JsonRpcServerConfig {
    public var maxConcurrency: Int = 64
    public var enableBatch: Boolean = true
    public var maxBatchSize: Int = 100
    public var maxRequestSize: Long = 1_048_576L // 1 MB
    public var errorHandler: JsonRpcErrorHandler = DefaultErrorHandler
    // Method naming strategy
    public var methodNaming: MethodNamingStrategy = MethodNamingStrategy.SIMPLE
    // System extensions
    public var enableSystemExtensions: Boolean = true
    private val extensionProviders = mutableListOf<SystemExtensionProvider>()
    public fun installExtension(provider: SystemExtensionProvider) {
        extensionProviders.add(provider)
    }
    // Interceptors
    private val interceptors = mutableListOf<JsonRpcServerInterceptor>()
    public fun install(interceptor: JsonRpcServerInterceptor) {
        interceptors.add(interceptor)
    }
    public var json: Json = Json {
        ignoreUnknownKeys = true
        encodeDefaults = true
    }
}
```

9.2. Method Naming Strategy

Three naming strategies are supported:

```
public enum class MethodNamingStrategy {
    SIMPLE,           // "methodName" only
    PREFIXED,         // "prefix.methodName" (prefix required at registration)
    FULLY_QUALIFIED   // "ServiceName.methodName" (auto-generated)
}
```

Collision detection:

```
internal class MethodRegistry {
    private val handlers = ConcurrentMap<String, MethodHandler>()
    fun register(methodName: String, handler: MethodHandler) {
        val existing = handlers.putIfAbsent(methodName, handler)
        if (existing != null) {
            throw IllegalStateException(
                "Method '$methodName' is already registered. " +
                "Use distinct prefixes or FULLY_QUALIFIED naming."
            )
```

```
            }
        }
    }
```

9.3. Service Registration

```
    // Simple registration
    server.registerService<CalculatorService> {
        CalculatorServiceImpl()
    }
    // Methods: "add", "subtract", etc.
    // With prefix (PREFIXED strategy or explicit)
    server.registerService<CalculatorService>(prefix = "math") {
        CalculatorServiceImpl()
    }
    // Methods: "math.add", "math.subtract", etc.
    // With FULLY_QUALIFIED strategy configured
    server.registerService<CalculatorService> {
        CalculatorServiceImpl()
    }
    // Methods: "CalculatorService.add", "CalculatorService.subtract", etc.
```

9.4. Request Size Enforcement

The maxRequestSize MUST be enforced before parsing:

```
    suspend fun handle(rawRequest: String): String? {
        // ENFORCE SIZE LIMIT
        if (rawRequest.length > config.maxRequestSize) {
            return json.encodeToString(
                JsonRpcResponse.error(
                    null,
                    JsonRpcErrorObject.requestTooLarge(config.maxRequestSize)
                )
            )
        }
        val element = try {
            json.parseToJsonElement(rawRequest)
        } catch (e: Exception) {
            return json.encodeToString(JsonRpcResponse.parseError())
        }
        return when (element) {
            is JsonArray -> handleBatch(element)
            is JsonObject -> handleSingle(element)
            else -> json.encodeToString(
                JsonRpcResponse.error(null, JsonRpcErrorObject.invalidRequest())
            )
        }
    }
```

9.5. Batch Handling with Concurrency Control

```
    private suspend fun handleBatch(array: JsonArray): String? {
        if (array.isEmpty()) {
            return json.encodeToString(
```

```
                JsonRpcResponse.error(null, JsonRpcErrorObject.invalidRequest())
            )
        }
        if (array.size > config.maxBatchSize) {
            return json.encodeToString(
                JsonRpcResponse.error(
                    null,
                    JsonRpcErrorObject.batchTooLarge(config.maxBatchSize)
                )
            )
        }
        // Process concurrently with semaphore limiting
        val semaphore = Semaphore(config.maxConcurrency)
        val responses = coroutineScope {
            array.map { element ->
                async {
                    semaphore.withPermit {
                        handleSingleElement(element)
                    }
                }
            }.awaitAll().filterNotNull()
        }
        // All notifications means no response
        return if (responses.isEmpty()) null
               else json.encodeToString(responses)
    }
```

9.6. Ktor Integration (example)

Ktor integration is provided as a separate module. Other frameworks can integrate via the core handle() method:

```
    // Ktor server integration
    fun Route.jsonRpc(path: String, server: JsonRpcServer) {
        post(path) {
            val requestBody = call.receiveText()
            val response = server.handle(requestBody)
            if (response != null) {
                call.respondText(response, ContentType.Application.Json)
            } else {
                call.respond(HttpStatusCode.NoContent)
            }
        }
    }

    // Spring WebFlux integration (example)
    @RestController
    class JsonRpcController(private val server: JsonRpcServer) {
        @PostMapping("/rpc")
        suspend fun handle(@RequestBody body: String): ResponseEntity<String> {
            val response = server.handle(body)
            return if (response != null) {
                ResponseEntity.ok()
```

```
                .contentType(MediaType.APPLICATION_JSON)
                .body(response)
        } else {
            ResponseEntity.noContent().build()
        }
    }
}

// AWS Lambda handler (serverless)
class JsonRpcLambdaHandler : RequestHandler<APIGatewayProxyRequestEvent,
                                            APIGatewayProxyResponseEvent> {
    private val server = JsonRpcServer(GlobalScope) { /* config */ }
    override fun handleRequest(
        input: APIGatewayProxyRequestEvent,
        context: Context
    ): APIGatewayProxyResponseEvent {
        val response = runBlocking { server.handle(input.body) }
        return APIGatewayProxyResponseEvent().apply {
            statusCode = if (response != null) 200 else 204
            body = response
            headers = mapOf("Content-Type" to "application/json")
        }
    }
}
```

10. Transport Layer

10.1. Transport Abstraction

The transport interface is minimal to allow maximum flexibility:

```
public interface JsonRpcTransport : Closeable {
    suspend fun send(message: String): String?
    fun receive(): Flow<String>
    val isConnected: Boolean
    override fun close()
}
```

10.2. HTTP Transport

```
public class HttpJsonRpcTransport(
    private val httpClient: HttpClient,
    private val url: String,
    private val headers: Map<String, String> = emptyMap()
) : JsonRpcTransport {
    private var closed = false
    override suspend fun send(message: String): String? {
        check(!closed) { "Transport is closed" }
        val response = httpClient.post(url) {
            contentType("application/json")
            headers.forEach { (key, value) -> header(key, value) }
            body = message
        }
        return when (response.status) {
            204 -> null // no content(all notifications)
            in 200..299 -> response.bodyAsText().takeIf { it.isNotBlank() }
            else -> throw JsonRpcTransportException(
                "HTTP " + response.status
            )
        }
    }
    override fun receive(): Flow<String> = emptyFlow()
    override val isConnected: Boolean get() = !closed
    override fun close() { closed = true; httpClient.close() }
}
```

The HttpClient is injected, allowing any HTTP library (Ktor, OkHttp, HttpURLConnection, etc.):

```
public fun interface HttpClient {
    suspend fun post(
        url: String,
        contentType: String,
        headers: Map<String, String>,
        body: String
    ): HttpResponse
}
public data class HttpResponse(
    val status: Int,
    val body: String
)
```

10.3. WebSocket Transport

```
public class WebSocketJsonRpcTransport(
    private val session: WebSocketSession,
    private val scope: CoroutineScope
) : JsonRpcTransport {
    private val pending = ConcurrentMap<RpcId, CompletableDeferred<String>>()
    private val incoming = MutableSharedFlow<String>()
    private val json = Json { ignoreUnknownKeys = true }
    init {
        scope.launch {
            try {
                for (frame in session.incoming) {
                    if (frame is Frame.Text) {
                        handleIncoming(frame.readText())
                    }
                }
            } finally {
                // Connection closed - fail all pending
                val error = JsonRpcTransportException("Connection closed")
                pending.values.forEach { it.completeExceptionally(error) }
                pending.clear()
            }
        }
    }

    private suspend fun handleIncoming(text: String) {
        val obj = try {
            json.parseToJsonElement(text) as? JsonObject
        } catch (e: Exception) {
            incoming.emit(text)
            return
        }
        val hasResult = obj?.containsKey("result") == true
        val hasError = obj?.containsKey("error") == true
        if (hasResult || hasError) {
            val id = obj?.get("id")?.let {
                json.decodeFromJsonElement<RpcId?>(it)
            }
            val deferred = id?.let { pending.remove(it) }
            if (deferred != null) {
                deferred.complete(text)
                return
            }
        }
        incoming.emit(text)
    }

    override suspend fun send(message: String): String? {
        val request = json.decodeFromString<JsonRpcRequest>(message)
        if (request.isNotification) {
```

```
                session.send(Frame.Text(message))
                return null
            }
            val id = request.id!!
            val deferred = CompletableDeferred<String>()
            pending[id] = deferred
            try {
                session.send(Frame.Text(message))
                return deferred.await()
            } catch (e: Exception) {
                pending.remove(id)
                throw e
            }
        }
        override fun receive(): Flow<String> = incoming.asSharedFlow()
        override val isConnected: Boolean get() = session.isActive
        override fun close() { session.cancel() }
    }
```

10.4. Standard I/O Transport (for MCP and LSP)

MCP and LSP use stdio for local process communication:

```
    public class StdioJsonRpcTransport(
        private val input: InputStream = System.`in`,
        private val output: OutputStream = System.out,
        private val scope: CoroutineScope
    ) : JsonRpcTransport {
        private val incoming = MutableSharedFlow<String>()
        private val reader = input.bufferedReader()
        private val writer = output.bufferedWriter()
        private var closed = false
        init {
            scope.launch(Dispatchers.IO) {
                try {
                    while (!closed) {
                        val message = readMessage() ?: break
                        incoming.emit(message)
                    }
                } catch (e: Exception) {
                    if (!closed) throw e
                }
            }
        }
        private fun readMessage(): String? {
            // Read Content-Length header
            val headers = mutableMapOf<String, String>()
            while (true) {
                val line = reader.readLine() ?: return null
                if (line.isEmpty()) break
                val (key, value) = line.split(": ", limit = 2)
                headers[key] = value
```

```
            }
            val length = headers["Content-Length"]?.toIntOrNull()
                ?: throw JsonRpcTransportException("Missing Content-Length")
            val buffer = CharArray(length)
            var read = 0
            while (read < length) {
                val n = reader.read(buffer, read, length - read)
                if (n < 0) return null
                read += n
            }
            return String(buffer)
        }
        override suspend fun send(message: String): String? {
            withContext(Dispatchers.IO) {
                writer.write("Content-Length: " + message.length + "\r\n")
                writer.write("\r\n")
                writer.write(message)
                writer.flush()
            }
            return null  // Responses come via receive()
        }
        override fun receive(): Flow<String> = incoming.asSharedFlow()
        override val isConnected: Boolean get() = !closed
        override fun close() {
            closed = true
            reader.close()
            writer.close()
        }
    }
```

10.5. Custom Transport Implementation

The transport interface allows arbitrary implementations:

```
    // In-memory transport for testing
    public class InMemoryJsonRpcTransport(
        private val server: JsonRpcServer
    ) : JsonRpcTransport {
        override suspend fun send(message: String): String? =
            server.handle(message)
        override fun receive(): Flow<String> = emptyFlow()
        override val isConnected: Boolean = true
        override fun close() {}
    }
    // Message queue transport (e.g., RabbitMQ, Kafka)
    public class MessageQueueTransport(
        private val requestQueue: MessageQueue,
        private val responseQueue: MessageQueue,
        private val scope: CoroutineScope
    ) : JsonRpcTransport {
        private val pending = ConcurrentMap<RpcId, CompletableDeferred<String>>()
        private val incoming = MutableSharedFlow<String>()
```

```
        init {
            scope.launch {
                responseQueue.consume { message ->
                    // Correlate response to request
                    // ...
                }
            }
        }
        override suspend fun send(message: String): String? {
            requestQueue.publish(message)
            // Await response or return null for notifications
            // ...
        }
        override fun receive(): Flow<String> = incoming.asSharedFlow()
        override val isConnected: Boolean get() = requestQueue.isConnected
        override fun close() {
            requestQueue.close()
            responseQueue.close()
        }
    }
```

10.6. Connection Management

```
    @JsonRpcDsl
    public class ConnectionConfig {
        public var reconnect: ReconnectStrategy = ReconnectStrategy.None
        public var pingInterval: Duration = 30.seconds
        public var pongTimeout: Duration = 10.seconds
        public var connectTimeout: Duration = 10.seconds
    }
    public sealed class ReconnectStrategy {
        public data object None : ReconnectStrategy()
        public data class Fixed(
            val delay: Duration,
            val maxAttempts: Int
        ) : ReconnectStrategy()
        public data class ExponentialBackoff(
            val initialDelay: Duration,
            val maxDelay: Duration,
            val maxAttempts: Int,
            val multiplier: Double = 2.0
        ) : ReconnectStrategy()
    }
```

11. System Extensions

11.1. Extension Architecture

System extensions (rpc.* methods) are implemented via a plugin architecture that allows adding custom extensions beyond the built-in ones:

```
public interface SystemExtensionProvider {
    val supportedMethods: Set<String>
    suspend fun handle(request: JsonRpcRequest): JsonRpcResponse?
}
// Built-in extension provider
internal class BuiltInExtensions(
    private val registry: MethodRegistry
) : SystemExtensionProvider {
    override val supportedMethods = setOf(
        "rpc.discover",
        "rpc.listMethods"
    )
    override suspend fun handle(request: JsonRpcRequest): JsonRpcResponse? {
        return when (request.method) {
            "rpc.discover" -> handleDiscover(request)
            "rpc.listMethods" -> handleListMethods(request)
            else -> null
        }
    }
    private fun handleListMethods(request: JsonRpcRequest): JsonRpcResponse {
        val methods = registry.methodNames.sorted()
        return JsonRpcResponse.success(
            request.id,
            Json.encodeToJsonElement(methods)
        )
    }
    private fun handleDiscover(request: JsonRpcRequest): JsonRpcResponse {
        val spec = OpenRpcSpec(
            openrpc = "1.0.0",
            info = registry.serviceInfo,
            methods = registry.methodDescriptors
        )
        return JsonRpcResponse.success(
            request.id,
            Json.encodeToJsonElement(spec)
        )
    }
}
```

11.2. Custom Extension Example

```
// MCP-specific extensions
class McpExtensionProvider : SystemExtensionProvider {
    override val supportedMethods = setOf(
        "rpc.describe",      // MCP resource description
        "rpc.capabilities"   // MCP capability negotiation
    )
```

```
        override suspend fun handle(request: JsonRpcRequest): JsonRpcResponse? {
            return when (request.method) {
                "rpc.describe" -> handleDescribe(request)
                "rpc.capabilities" -> handleCapabilities(request)
                else -> null
            }
        }
    }
    // Usage
    val server = JsonRpcServer(scope) {
        enableSystemExtensions = true
        installExtension(McpExtensionProvider())
    }
```

11.3. Extension Dispatch

System extensions are checked before application methods:

```
    internal class ExtensionAwareRouter(
        private val config: JsonRpcServerConfig,
        private val registry: MethodRegistry,
        private val extensions: List<SystemExtensionProvider>
    ) : JsonRpcRouter {
        suspend fun route(request: JsonRpcRequest): JsonRpcResponse {
            // 1. Check if system extension
            if (request.isSystemExtension && config.enableSystemExtensions) {
                for (provider in extensions) {
                    if (request.method in provider.supportedMethods) {
                        val response = provider.handle(request)
                        if (response != null) return response
                    }
                }
                // Unknown system extension
                return JsonRpcResponse.error(
                    request.id,
                    JsonRpcErrorObject.methodNotFound()
                )
            }
            // 2. Route to application method
            val handler = registry.get(request.method)
                ?: return JsonRpcResponse.error(
                    request.id,
                    JsonRpcErrorObject.methodNotFound()
                )
            return try {
                val result = handler.invoke(request.params)
                JsonRpcResponse.success(request.id, result)
            } catch (e: Exception) {
                JsonRpcResponse.error(
                    request.id,
                    config.errorHandler.handle(e, request)
                )
            }
```

```
        }
    }
```

11.4. Disabling System Extensions

System extensions can be disabled for security or performance:

```
    val server = JsonRpcServer(scope) {
        enableSystemExtensions = false  // Disable all rpc.* methods
    }
```

When disabled, rpc.* methods are treated as application methods (which will fail with Method not found unless explicitly registered).

12. Serialization Strategy

12.1. kotlinx.serialization Integration

```
@JsonRpcDsl
public class JsonRpcSerializationConfig {
    internal val moduleBuilder = SerializersModuleBuilder()
    public inline fun <reified T> contextual(serializer: KSerializer<T>) {
        moduleBuilder.contextual(T::class, serializer)
    }
    public fun polymorphic(block: PolymorphicModuleBuilder<Any>.() -> Unit) {
        moduleBuilder.polymorphic(Any::class, block)
    }
    internal fun build(): SerializersModule = moduleBuilder.build()
}
```

The json instance uses the configured module:

```
val json = Json {
    ignoreUnknownKeys = true
    encodeDefaults = true
    serializersModule = config.serialization.build()
}
```

12.2. Parameter Encoding

By-position (Array):

```
client.call("add", listOf(5, 3))
// {"jsonrpc":"2.0","method":"add","params":[5,3],"id":1}
```

By-name (Object):

```
@Serializable
data class AddParams(val a: Int, val b: Int)
client.call("add", AddParams(a = 5, b = 3))
// {"jsonrpc":"2.0","method":"add","params":{"a":5,"b":3},"id":1}
```

Parameter names are case-sensitive per specification.

12.3. Custom Type Handling

Users can register custom serializers for application types:

```
val client = JsonRpcClient(transport, scope) {
    serialization {
        contextual(InstantSerializer)
        contextual(UUIDSerializer)
        polymorphic {
            subclass(DogCommand::class, DogCommand.serializer())
            subclass(CatCommand::class, CatCommand.serializer())
        }
    }
}
```

13. Error Handling

13.1. Exception Hierarchy

```
public sealed interface JsonRpcError {
    val code: Int
    val message: String
    val data: JsonElement?
    fun toErrorObject(): JsonRpcErrorObject
}
public sealed class JsonRpcException(
    override val message: String,
    cause: Throwable? = null
) : Exception(message, cause), JsonRpcError {
    override val data: JsonElement? = null
    override fun toErrorObject() = JsonRpcErrorObject(code, message, data)
}
public class ParseErrorException(
    override val data: JsonElement? = null
) : JsonRpcException("Parse error") {
    override val code: Int = -32700
}
public class InvalidRequestException(
    message: String = "Invalid Request",
    override val data: JsonElement? = null
) : JsonRpcException(message) {
    override val code: Int = -32600
}
public class MethodNotFoundException(
    val method: String? = null
) : JsonRpcException("Method not found") {
    override val code: Int = -32601
}
public class InvalidParamsException(
    message: String = "Invalid params",
    override val data: JsonElement? = null
) : JsonRpcException(message) {
    override val code: Int = -32602
}
public class InternalErrorException(
    message: String = "Internal error",
    cause: Throwable? = null,
    override val data: JsonElement? = null
) : JsonRpcException(message, cause) {
    override val code: Int = -32603
}
public class JsonRpcTimeoutException(
    message: String,
    cause: Throwable? = null
) : JsonRpcException(message, cause) {
    override val code: Int = -32005
}
```

13.2. Extended Exception Mapping

The error handler maps exceptions to JSON-RPC errors, including serialization exceptions:

```
public fun interface JsonRpcErrorHandler {
    fun handle(throwable: Throwable, request: JsonRpcRequest): JsonRpcErrorObject
}
public object DefaultErrorHandler : JsonRpcErrorHandler {
    override fun handle(
        throwable: Throwable,
        request: JsonRpcRequest
    ): JsonRpcErrorObject = when (throwable) {
        // JSON-RPC exceptions pass through
        is JsonRpcException -> throwable.toErrorObject()
        // Argument validation
        is IllegalArgumentException ->
            JsonRpcErrorObject.invalidParams(throwable.message)
        // Serialization errors -> Invalid params or Invalid request
        is SerializationException ->
            JsonRpcErrorObject.invalidParams(
                "Serialization error: " + throwable.message
            )
        is JsonDecodingException ->
            JsonRpcErrorObject.invalidParams(
                "JSON decoding error: " + throwable.message
            )
        // Coroutine cancellation
        is CancellationException ->
            JsonRpcErrorObject.requestCancelled()
        // Timeout
        is TimeoutCancellationException ->
            JsonRpcErrorObject.timeout()
        // Everything else -> Internal error (no details exposed)
        else -> {
            // Log for debugging but don't expose to client
            logger.error("Internal error handling " + request.method, throwable)
            JsonRpcErrorObject.internalError()
        }
    }
}
```

13.3. Client-Side Error Handling

```
try {
    val result = client.call<User>("getUser", mapOf("id" to 1))
} catch (e: MethodNotFoundException) {
    println("Method not found")
} catch (e: InvalidParamsException) {
    println("Invalid parameters: " + e.message)
} catch (e: JsonRpcTimeoutException) {
    println("Request timed out")
} catch (e: JsonRpcException) {
    println("JSON-RPC error " + e.code + ": " + e.message)
```

```
    }
```

14. Interceptors

14.1. Interceptor Interface

```
public fun interface JsonRpcInterceptor {
    suspend fun intercept(
        call: JsonRpcCall,
        next: JsonRpcHandler
    ): JsonRpcResponse
}
public data class JsonRpcCall(
    val request: JsonRpcRequest,
    val context: JsonRpcContext
)
```

14.2. Execution Order

Interceptors execute in registration order for requests and reverse order for responses:

```
Request:  Interceptor1 -> Interceptor2 -> Handler
Response: Handler -> Interceptor2 -> Interceptor1
```

14.3. Built-in Interceptors

```
public class LoggingInterceptor(
    private val logger: JsonRpcLogger = DefaultLogger
) : JsonRpcInterceptor {
    override suspend fun intercept(
        call: JsonRpcCall,
        next: JsonRpcHandler
    ): JsonRpcResponse {
        logger.logRequest(call.request, call.context)
        val start = currentTimeMillis()
        val response = next.handle(call)
        val duration = currentTimeMillis() - start
        logger.logResponse(response, duration, call.context)
        return response
    }
}
public class AuthInterceptor(
    private val tokenProvider: suspend () -> String?
) : JsonRpcInterceptor {
    override suspend fun intercept(
        call: JsonRpcCall,
        next: JsonRpcHandler
    ): JsonRpcResponse {
        val token = tokenProvider()
        if (token != null) {
            call.context.attributes[AuthKey] = "Bearer " + token
        }
        return next.handle(call)
    }
}
```

```
public class RateLimitInterceptor(
    private val requestsPerSecond: Int
) : JsonRpcInterceptor {
    private val limiter = RateLimiter(requestsPerSecond)
    override suspend fun intercept(
        call: JsonRpcCall,
        next: JsonRpcHandler
    ): JsonRpcResponse {
        if (!limiter.tryAcquire()) {
            return JsonRpcResponse.error(
                call.request.id,
                JsonRpcErrorObject.serverBusy()
            )
        }
        return next.handle(call)
    }
}
```

15. Cancellation Support

15.1. Client-Side Cancellation

When a client coroutine is cancelled:

(1) Stop waiting for the response immediately.
(2) Optionally send $/cancelRequest notification(MCP/LSP pattern).
(3) Clean up pending request state.

```
@JsonRpcDsl
public class JsonRpcClientConfig {
    public var sendCancelNotification: Boolean = false
}
internal suspend fun <T> callWithCancellation(
    method: String,
    params: Any?,
    resultType: KType
): T = suspendCancellableCoroutine { cont ->
    val id = config.idGenerator.next()
    pending[id] = cont
    cont.invokeOnCancellation {
        pending.remove(id)
        if (config.sendCancelNotification) {
            scope.launch {
                transport.send(json.encodeToString(
                    JsonRpcRequest(
                        method = "$/cancelRequest",
                        params = buildJsonObject { put("id", id.toString()) }
                    )
                ))
            }
        }
    }
    // Send request...
}
```

15.2. Server-Side Cancellation

```
internal class CancellationAwareRouter : JsonRpcRouter {
    private val activeRequests = ConcurrentMap<RpcId, Job>()
    init {
        registerSystemMethod("$/cancelRequest") { params ->
            val idStr = (params as? JsonObject)?.get("id")?.jsonPrimitive?.content
            val idToCancel = idStr?.let { RpcId.of(it) }
            idToCancel?.let { activeRequests[it]?.cancel() }
            null  // Notification, no response
        }
    }
    override suspend fun route(request: JsonRpcRequest): JsonRpcResponse {
        val job = coroutineContext.job
        request.id?.let { activeRequests[it] = job }
        try {
            return handleRequest(request)
```

```
            } finally {
                request.id?.let { activeRequests.remove(it) }
            }
        }
    }
```

16. Kotlin Multiplatform

16.1. Platform Support

Platform	Transport	Proxy Support
JVM	All	Reflection + Generated stubs
Android	All	Reflection + Generated stubs
JS	HTTP, WebSocket	Generated stubs only
iOS	HTTP, WebSocket	Generated stubs only
macOS	All	Generated stubs only
Linux	All	Generated stubs only
Windows	HTTP, WebSocket	Generated stubs only
Wasm	HTTP, WebSocket	Generated stubs only

16.2. Expect/Actual Patterns

```
// commonMain
internal expect fun currentTimeMillis(): Long
internal expect class AtomicLong(initial: Long) {
    fun get(): Long
    fun incrementAndGet(): Long
}
// jvmMain
internal actual fun currentTimeMillis(): Long = System.currentTimeMillis()
internal actual class AtomicLong actual constructor(initial: Long) {
    private val delegate = java.util.concurrent.atomic.AtomicLong(initial)
    actual fun get(): Long = delegate.get()
    actual fun incrementAndGet(): Long = delegate.incrementAndGet()
}
// nativeMain
internal actual fun currentTimeMillis(): Long =
    kotlin.system.getTimeMillis()
internal actual class AtomicLong actual constructor(initial: Long) {
    private val value = kotlin.concurrent.AtomicLong(initial)
    actual fun get(): Long = value.value
    actual fun incrementAndGet(): Long = value.incrementAndGet()
}
```

16.3. JVM-Only Features

withService<T>() using reflection is JVM-only. On other platforms, use generated stubs:

```
// commonMain - interface only
@Rpc
interface CalculatorService {
    suspend fun add(a: Int, b: Int): Int
}
// JVM - can use reflection
val calc = client.withService<CalculatorService>()
// JS/Native - use generated extension
```

```
val calc = client.calculatorService()  // Generated
```

17. Code Generation

17.1. Generated Stubs

```
@Generated("kotlinx-rpc-jsonrpc")
internal class CalculatorService__JsonRpcStub(
    private val client: JsonRpcClient,
    private val prefix: String = ""
) : CalculatorService {
    override suspend fun add(a: Int, b: Int): Int {
        val method = if (prefix.isEmpty()) "add" else prefix + ".add"
        val params = buildJsonArray { add(a); add(b) }
        return client.call(method, params, typeOf<Int>())
    }
    override suspend fun subtract(a: Int, b: Int): Int {
        val method = if (prefix.isEmpty()) "subtract" else prefix + ".subtract"
        val params = buildJsonArray { add(a); add(b) }
        return client.call(method, params, typeOf<Int>())
    }
}
// Generated extension function
public fun JsonRpcClient.calculatorService(prefix: String = ""): CalculatorService =
    CalculatorService__JsonRpcStub(this, prefix)
```

17.2. Build Configuration

```
// Compiler plugin (best performance)
plugins {
    id("org.jetbrains.kotlin.multiplatform")
    id("org.jetbrains.kotlinx.rpc")
}
// Or KSP (wider compatibility)
plugins {
    id("com.google.devtools.ksp")
}
dependencies {
    ksp("org.jetbrains.kotlinx:kotlinx-rpc-jsonrpc-ksp:VERSION")
}
```

17.3. Server-Side Generation

Server stubs for method registration are also generated:

```
@Generated("kotlinx-rpc-jsonrpc")
internal fun JsonRpcServer.registerCalculatorService(
    prefix: String = "",
    provider: () -> CalculatorService
) {
    val service = provider()
    val methodPrefix = if (prefix.isEmpty()) "" else prefix + "."
    registerMethod(methodPrefix + "add") { params ->
        val args = params as JsonArray
        val a = json.decodeFromJsonElement<Int>(args[0])
        val b = json.decodeFromJsonElement<Int>(args[1])
```

```
            json.encodeToJsonElement(service.add(a, b))
        }
        registerMethod(methodPrefix + "subtract") { params ->
            // ...
        }
    }
```

18. Security Considerations

18.1. Transport Security

JSON-RPC messages SHOULD be transmitted over TLS (HTTPS/WSS).

18.2. Authentication

JSON-RPC 2.0 does not define authentication. Use transport-level authentication or an auth interceptor.

18.3. Input Validation

All incoming messages MUST be validated:

- Invalid JSON -> Parse error (id=null)
- Oversized messages -> Request too large (-32004)
- Malformed requests -> Invalid Request (-32600)

18.4. CORS and Origin Validation

For HTTP in browser contexts, configure CORS headers.
For WebSocket, validate the Origin header.

18.5. Information Disclosure

Error messages MUST NOT include:

- Stack traces
- Internal system paths
- Database error details

Method names SHOULD NOT be echoed in error responses for unauthenticated clients to prevent enumeration attacks.

18.6. Rate Limiting

Use RateLimitInterceptor or transport-level rate limiting to prevent denial of service.

18.7. MCP Security Considerations

When implementing MCP servers:

- Require explicit user consent for tool execution
- Implement proper OAuth2 authorization per MCP auth spec
- Validate all tool inputs
- Log all tool invocations for auditing

19. Testing Strategy

19.1. Mock Transport

```
class MockJsonRpcTransport : JsonRpcTransport {
    val sentMessages = mutableListOf<String>()
    private val responseQueue = ArrayDeque<String?>()
    fun queueResponse(response: String?) { responseQueue.add(response) }
    override suspend fun send(message: String): String? {
        sentMessages.add(message)
        return responseQueue.removeFirstOrNull()
    }
    override fun receive(): Flow<String> = emptyFlow()
    override val isConnected: Boolean = true
    override fun close() {}
}
```

19.2. Protocol Compliance Tests

```
@Test
fun emptyBatchReturnsInvalidRequest() = runTest {
    val server = JsonRpcServer(testScope) {}
    val response = server.handle("[]")
    val parsed = json.decodeFromString<JsonRpcResponse>(response!!)
    assertEquals(-32600, parsed.error?.code)
}
@Test
fun allNotificationBatchReturnsNothing() = runTest {
    val batch = json.encodeToString(listOf(
        JsonRpcRequest(method = "notify1"),
        JsonRpcRequest(method = "notify2")
    ))
    val response = server.handle(batch)
    assertNull(response)
}
@Test
fun fractionalIdIsTruncated() = runTest {
    val id = json.decodeFromString<RpcId>("1.9")
    assertEquals(RpcId.NumberId(1), id)
}
```

19.3. Interoperability Tests

Verify against other JSON-RPC implementations:

- jsonrpc4j (Java)
- python-jsonrpc
- MCP TypeScript SDK

20. Observability

20.1. Logging

```
public interface JsonRpcLogger {
    fun logRequest(request: JsonRpcRequest, context: JsonRpcContext)
    fun logResponse(response: JsonRpcResponse, durationMs: Long, context: JsonRpcContext)
    fun logError(error: Throwable, context: JsonRpcContext)
}
```

20.2. Metrics

```
public interface JsonRpcMetrics {
    fun recordRequest(method: String)
    fun recordResponse(method: String, durationMs: Long, success: Boolean)
    fun recordBatchSize(size: Int)
    fun recordError(method: String, code: Int)
}
```

20.3. Tracing

```
public interface JsonRpcTracer {
    fun startSpan(method: String): Span
    fun extractContext(headers: Map<String, String>): TraceContext?
    fun injectContext(context: TraceContext, headers: MutableMap<String, String>)
}
```

20.4. Usage

```
val server = JsonRpcServer(scope) {
    install(LoggingInterceptor(Slf4jLogger))
    install(MetricsInterceptor(MicrometerMetrics))
    install(TracingInterceptor(OpenTelemetryTracer))
}
```

21. Future Considerations

21.1. Streaming Extensions

JSON-RPC 2.0 does not define streaming. Future extensions may support:

- Subscription pattern (client subscribes, server sends notifications)
- Progress notifications for long-running operations
- Server-Sent Events (SSE) integration

21.2. MCP Full Support

Complete MCP SDK implementation including:

- Tools, Resources, and Prompts primitives
- Sampling for nested LLM calls
- Roots for filesystem access
- OAuth2 authorization per MCP spec

21.3. OpenRPC Integration

Full OpenRPC spec generation for service discovery and documentation.

21.4. Binary Encoding

Optional CBOR or MessagePack encoding for performance-critical applications while maintaining JSON-RPC semantics.

22. References

22.1. Normative References

[JSONRPC20] JSON-RPC Working Group, "JSON-RPC 2.0 Specification", March 2010, <https://www.jsonrpc.org/specification>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

22.2. Informative References

[KOTLINX-RPC] JetBrains, "kotlinx-rpc", <https://github.com/Kotlin/kotlinx-rpc>.

[KOTLINX-SERIAL] JetBrains, "kotlinx.serialization", <https://github.com/Kotlin/kotlinx.serialization>.

[KOTLINX-COROUT] JetBrains, "kotlinx.coroutines", <https://github.com/Kotlin/kotlinx.coroutines>.

[MCP] Anthropic, "Model Context Protocol", <https://modelcontextprotocol.io/>.

[LSP] Microsoft, "Language Server Protocol", <https://microsoft.github.io/language-server-protocol/>.

[OPENRPC] OpenRPC, "The OpenRPC Specification", <https://open-rpc.org/>.

[ETH-JSONRPC] Ethereum Foundation, "JSON-RPC API", <https://ethereum.org/developers/docs/apis/json-rpc>.

[XQT-JSONRPC] Dunn, R., "xqt-kotlinx-json-rpc", <https://github.com/rhdunn/xqt-kotlinx-json-rpc>.

[KSP] Google, "Kotlin Symbol Processing", <https://github.com/google/ksp>.

Appendix A. Complete Examples

A.1. Service Definition

```
@Rpc
interface CalculatorService {
    suspend fun add(a: Double, b: Double): Double
    suspend fun subtract(a: Double, b: Double): Double
    suspend fun divide(a: Double, b: Double): Double
    @JsonRpcNotification
    suspend fun logCalculation(operation: String, result: Double)
}
```

A.2. Server Implementation

```
class CalculatorServiceImpl : CalculatorService {
    override suspend fun add(a: Double, b: Double): Double = a + b
    override suspend fun subtract(a: Double, b: Double): Double = a - b
    override suspend fun divide(a: Double, b: Double): Double {
        if (b == 0.0) {
            throw InvalidParamsException("Division by zero")
        }
        return a / b
    }
    override suspend fun logCalculation(operation: String, result: Double) {
        println("Logged: " + operation + " = " + result)
    }
}
```

A.3. Server Setup

```
fun main() {
    val scope = CoroutineScope(Dispatchers.Default + SupervisorJob())
    val server = JsonRpcServer(scope) {
        maxConcurrency = 100
        maxBatchSize = 50
        maxRequestSize = 1_048_576L
        enableSystemExtensions = true
        install(LoggingInterceptor())
    }
    server.registerService<CalculatorService> { CalculatorServiceImpl() }
    // Ktor integration
    embeddedServer(Netty, port = 8080) {
        routing {
            post("/rpc") {
                val body = call.receiveText()
                val response = server.handle(body)
                if (response != null) {
                    call.respondText(response, ContentType.Application.Json)
                } else {
                    call.respond(HttpStatusCode.NoContent)
                }
            }
        }
    }.start(wait = true)
```

```
}
```

A.4. Client Implementation

```
suspend fun main() {
    val scope = CoroutineScope(Dispatchers.Default + SupervisorJob())
    val transport = HttpJsonRpcTransport(
        httpClient = KtorHttpClient(HttpClient()),
        url = "http://localhost:8080/rpc"
    )
    val client = JsonRpcClient(transport, scope) {
        requestTimeout = 30.seconds
        idGenerator = RpcIdGenerator.Sequential()
    }
    // Type-safe invocation (JVM only)
    val calculator = client.withService<CalculatorService>()
    val sum = calculator.add(10.0, 5.0)
    println("10 + 5 = " + sum)
    // Error handling
    try {
        calculator.divide(1.0, 0.0)
    } catch (e: InvalidParamsException) {
        println("Error: " + e.message)
    }
    // Heterogeneous batch
    val results = client.batch {
        val sumHandle = call<Double>("add", listOf(1.0, 2.0))
        val productHandle = call<Double>("multiply", listOf(3.0, 4.0))
        val nameHandle = call<String>("getServerName")
        notify("logCalculation", mapOf("operation" to "batch", "result" to 0.0))
    }
    when (val sum = results.get(sumHandle)) {
        is JsonRpcBatchResult.Success -> println("Sum: " + sum.value)
        is JsonRpcBatchResult.Error -> println("Error: " + sum.error.message)
    }
    client.close()
    scope.cancel()
}
```

A.5. MCP Server Example

```
fun main() {
    val scope = CoroutineScope(Dispatchers.Default + SupervisorJob())
    val transport = StdioJsonRpcTransport(scope = scope)
    val server = JsonRpcServer(scope) {
        enableSystemExtensions = true
        installExtension(McpToolsExtension())
    }
    server.registerMethod("tools/call") { params ->
        val toolName = (params as JsonObject)["name"]?.jsonPrimitive?.content
        val toolArgs = params["arguments"]
        // Execute tool and return result
        executeTool(toolName, toolArgs)
```

```
        }
        // Run MCP server over stdio
        scope.launch {
            transport.receive().collect { message ->
                val response = server.handle(message)
                if (response != null) {
                    transport.send(response)
                }
            }
        }
        // Keep alive
        runBlocking { scope.coroutineContext.job.join() }
    }
```

Appendix B. Error Code Registry

B.1. Protocol Error Codes (Reserved)

Code	Constant	Description
-32700	PARSE_ERROR	Invalid JSON received
-32600	INVALID_REQUEST	Not a valid Request object
-32601	METHOD_NOT_FOUND	Method does not exist
-32602	INVALID_PARAMS	Invalid method parameters
-32603	INTERNAL_ERROR	Internal JSON-RPC error

B.2. Server Error Codes (This Implementation)

Code	Constant	Description
-32001	REQUEST_CANCELLED	Request was cancelled
-32002	SERVER_BUSY	Server overloaded
-32003	BATCH_TOO_LARGE	Batch exceeds maxBatchSize
-32004	REQUEST_TOO_LARGE	Request exceeds maxRequestSize
-32005	TIMEOUT	Request timed out

B.3. Application Error Codes (Recommended Ranges)

Range	Category
1000–1099	Authentication/Authorization
1100–1199	Validation errors
1200–1299	Resource errors (not found, conflict)
1300–1399	Business logic errors
1400–1499	Rate limiting

Author's Address
Mykhailo Kozyrev
Email: smrtkozyrev@gmail.com
URI: <https://github.com/Smikalo>