Network Working Group
Internet-Draft
Intended status: Informational
Expires: 9 July 2026

Y. Shoaib
January 2026

Implementation of JSON-RPC in kotlinx.rpc
draft-youssef-na-jsonrpc-latest

Abstract

This document specifies an implementation of the JSON-RPC 2.0 protocol in the kotlinx.rpc Kotlin Multiplatform library. It provides both a core, transport-agnostic API, as well as Ktor integrations. Further, a quasi-protocol is defined to support bidirectional streaming via Flows over JSON-RPC.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://kyay10.github.io/json-rpc-rfc/draft-youssef-na-jsonrpc.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-youssef-na-jsonrpc/.

Source for this draft and an issue tracker can be found at https://github.com/kyay10/json-rpc-rfc.

Status of This Memo

This Internet-Draft will expire on 5 July 2026.

Copyright Notice

Table of Contents

1. Introduction

JSON-RPC 2.0 (https://www.jsonrpc.org/specification) is a stateless remote procedure call (RPC) protocol based on JavaScript Object Notation (JSON). It has a simple request-response flow and is transport agnostic.

Kotlinx.rpc (https://github.com/Kotlin/kotlinx-rpc) is a Kotlin

Multiplatform (KMP) library that provides RPC abstractions for both clients and servers (hereafter referred to as endpoints).

The library already has support for gRPC (https://grpc.io/) alongside its bespoke krpc protocol. It supports single responses via suspend functions, and bidirectional streaming via flows. It also supports marshalling exceptions.

1.1. Goals

* Provide a full implementation of JSON-RPC 2.0 protocol in kotlinx.rpc
* Support for both suspend functions and flows between Kotlin endpoints
* Simple compatibility for suspend functions with non-Kotlin endpoints
* Maintain backwards compatibility
* Add a minimal API surface (while reusing existing APIs in kotlinx.rpc core)
* Stay transport-agnostic
* Provide Ktor integrations
* No code changes, instead the implementation shall live in new module(s)
* No changes to the compiler plugin

1.2. Non-Goals

* Extend or modify JSON-RPC 2.0 protocol
* Compatibility for flows with non-Kotlin endpoints (since JSON-RPC 2.0 does not provide streaming semantics)

2. Core APIs

2.1. Notifications

JSON-RPC defines notifications as requests that must not be responded to.

Importantly, the server can't repsond to them with errors, either.

While it seems natural to use a Unit return type for notifications, it can be surprising to the user that errors aren't propagated.

To sidestep this problem, a custom NotificationOk return type is provided for notifications:

```
public data object NotificationOk
```

For example:

```
@Rpc interface Foo {
    suspend fun noResponse(): NotificationOk
    suspend fun mightError()
}
```

noResponse is a notification, while mightError is a normal request that can result in an exception.

2.2. Reserved method names

While JSON-RPC reserves method names starting with rpc., the implementation shall make no effort to enforce this restriction.

Instead, we assume that, if the user defines such a method, they know what they are doing.

2.3. Parameter encoding

JSON-RPC allows parameters in a request to be passed by-position (as an array) or by-name (as an object).

Since we'd like to support the full JSON-RPC spec, we provide a ParameterEncoding enum:

```
public enum class ParameterEncoding { ByName, ByPosition }
```

API methods should have ByName as a default where possible, since it's more human-readable.

The choice of parameter encoding is configurable on the client only. The server must support both encodings.

A minor complication arises with the currently-experimental context parameters (https://github.com/Kotlin/KEEP/blob/master/proposals/context-parameters.md), since context parameters may have no name.

In that case, they're assumed to have the special name <contextN>, where N is the index of the context parameter in the parameter list.

An alternative is to simply not support unnamed context parameters with by-name encoding, but that seems less than ideal.

2.4. Ids

Every JSON-RPC request must have an id, except for notifications.

We thus specify that the client must generate numerical ids and ensure their uniqueness.

2.5. Serialization

In the presence of complex data in the parameters or return types of API methods, serialization is necessary.

Since JSON-RPC is JSON-based, we can only support String-based serialization formats.

Hence, we use StringFormat from kotlinx.serialization as the serialization abstraction.

2.6. Errors

JSON-RPC requires an error code in error responses.

Kotlin exceptions don't come with error codes, so, we provide a public JsonRpcException type as follows:

```
public class JsonRpcException(val code: Int, message: String):
IllegalStateException(message)
```

We make no effort to validate error codes, instead leaving it up to the user to provide appropriate codes.

For all other exceptions, we use the error code -32000, which is within the server error range defined by JSON-RPC.

Importantly, JSON-RPC reserves some error codes for special RPC errors, with the relevant ones being (adapted from JSON-RPC 2.0 §5.1):

Code	Message	Meaning

-32700	Parse error	Invalid JSON was received by the server
-32600	Invalid Request	The JSON sent is not a valid Request object
-32601	Method not found	The method does not exist / is not available
-32602	Invalid params	Invalid method parameter(s)

Table 1

These errors must be used by the implementation where appropriate.

The error response's message field must contain the Exception's message.

The error response's data field must contain, at a minimum, the fully-qualified name of the Exception class and the stacktrace.

Other fields may be added as necessary.

Alternatively, SerializedException (https://github.com/Kotlin/kotlinx-rpc/blob/2b895329b779d6560363f4fb79eed0e27ea82e07/krpc/krpc-core/src/commonMain/kotlin/kotlinx/rpc/krpc/internal/SerializedException.kt), which is internally used by Krpc, may be moved into core and used for this purpose.

2.7. Batching

JSON-RPC supports batching of requests into a single request (as an array).

The server implementation must support batching.

Batching on the client is to be done on a best-effort basis, i.e., the client may choose to batch requests or not.

The relaxed constraint is to ensure that the client can make requests in a timely fashion.

2.8. Transport

Since the protocol communicates through JSON, all we require is a

transport that can send and receive strings.

We thus define a simple JsonRpcTransport interface as follows:

```
public interface JsonRpcTransport : CoroutineScope {
    suspend fun send(message: String)
    suspend fun receive(): String
}
```

Importantly, the transport is also a CoroutineScope, to allow for cancellation when needed. IT also thus provides a coroutineContext which determines important coroutine properties such as the dispatcher.

2.9. Entrypoints

Kotlinx.rpc provides RpcClient and RpcServer interfaces as the main entrypoints for clients and servers, respectively.

We thus provide JsonRpcClient and JsonRpcServer implementations of these interfaces:

```
public class JsonRpcClient(
        private val format: StringFormat,
        private val encoding: ParameterEncoding = ParameterEncoding.
ByName,
        private val transport: JsonRpcTransport,
): RpcClient
public class JsonRpcServer(
    private val format: StringFormat,
    private val transport: JsonRpcTransport,
): RpcServer {
    public fun close(message: String? = null) { ... }
    public suspend fun awaitCompletion() { ... }
}
```

3. Ktor

We provide Ktor integrations for both client and server.

3.1. Transport

Our canonical Ktor transport implementation, using Ktor's WebSocketSession, is as follows:

```
@InternalRpcApi
public class KtorTransport(private val webSocketSession:
```

```
WebSocketSession) : JsonRpcTransport, CoroutineScope by webSocketSession {
       override suspend fun send(message: String) = webSocketSession.send(
message)

       override suspend fun receive(): KrpcTransportMessage =
webSocketSession.incoming.receive() as? Frame.Text ?: error("Unsupported
websocket frame type: ${message::class}. Expected Frame.Text")
   }
```

We use @InternalRpcApi from core to denote that this API is not intended for public use.

3.2. Client

Ktor client APIs are provided in Appendix A (Appendix A).

Importantly, we provide an installJsonRpc extension function that takes in a StringFormat and an optional ParameterEncoding, and installs the necessary WebSocket support.

HttpClient.rpc sets up the websocket and allows the user to configure the url and other HttpRequestBuilder properties; the returned JsonRpcClient can then be used to access RPC services

3.3. Server

Ktor server APIs are provided in Appendix B (Appendix B).

Importantly, we provide an installJsonRpc extension function that takes in a StringFormat and installs the necessary WebSocket support.

Route.rpc sets up the websocket route and allows the user to register services inside the block.

4. Flow support

JSON-RPC _technically_ does not prohibit multiple responses to a single request, but it does not define any semantics for it.

We thus provide our own quasi-protocol on top of JSON-RPC to support flows.

Since kotlinx.rpc explicitly supports bidirectional streaming, we'll must have bidirectional communication between endpoints.

Every Flow that's communicated between endpoints is assigned a unique id (unrelated to request/response ids). Communication thus involves

simply sending the id.

Since Flows are cold by default, our quasi-protocol is pull-based, i.e., the receiver requests items from the sender.

We reserve 2 unique method names "kotlinx.rpc.jsonrpc.internal.nextFlowValue" and "kotlinx.rpc.jsonrpc.internal.cancelFlow" for this purpose.

When the receiver wants the next item from the Flow, it sends a request with this method name and the Flow id as the sole parameter.

The sender then responds with the next item from the Flow (and can error if necessary).

Finally, when the Flow is complete, the sender simply returns an error response with code -32001 to indicate completion.

If the flow is cancelled by the client, they must send a request with method cancelFlow and the Flow id as the sole parameter.

For instance, consider the following service:

```
@Rpc interface Numbers {
    fun getNumbers(): Flow<Int>
}
```

When the client calls getNumbers, nothing happens yet, since Flows are cold.

When the client starts collecting the Flow, the client sends a request to the server to invoke getNumbers.

The server responds with a Flow id (say, 42).

Then, the client sends a request with method nextFlowValue and parameter 42.

The server responds with the next number in the Flow.

If the client cancels the Flow, it sends a request with method cancelFlow and parameter 42.

If the server-side Flow completes, it responds to the next nextFlowValue request with an error code -32001.

This does not interfere with normal JSON-RPC requests and responses,

since the quasi-protocol uses reserved method names.

This approach may seem heavyweight since every item in the Flow requires a round-trip. Batching mitigates this somewhat.

Additonally, Flows have excellent buffering operators that can be used to ensure that the receiver always has items to process.

Our quasi-protocol is simple enough to be implemented in non-Kotlin endpoints, if desired.

Appendix A. Appendix A: Ktor Client API

The Ktor client API is as follows:

```
   internal val JsonRpcClientFormatKey = AttributeKey<StringFormat>("
JsonRpcClientFormatKey")
   internal val JsonRpcClientEncodingKey = AttributeKey<ParameterEncoding
>("JsonRpcClientEncodingKey")
   public fun HttpClientConfig<*>.installJsonRpc(format: StringFormat,
encoding: ParameterEncoding = ParameterEncoding.ByName) {
       pluginOrNull(WebSockets) ?: install(WebSockets)
       attributes.put(JsonRpcClientFormatKey, format)
       attributes.put(JsonRpcClientEncodingKey, encoding)
   }

   public fun HttpClient.rpc(
       urlString: String,
           format: StringFormat = attributes[JsonRpcClientFormatKey],
           encoding: ParameterEncoding = attributes[
JsonRpcClientEncodingKey],
       block: HttpRequestBuilder.() → Unit = {},
   ): JsonRpcClient = rpc {
       url(urlString)
       block()
   }

   public fun HttpClient.rpc(
           format: StringFormat = attributes[JsonRpcClientFormatKey],
           encoding: ParameterEncoding = attributes[
JsonRpcClientEncodingKey],
       block: HttpRequestBuilder.() → Unit = {},
   ): JsonRpcClient {
       pluginOrNull(WebSockets)
           ?: error("RPC for client requires $WebSockets plugin to be
installed firstly")
       return JsonRpcClient(format, encoding, KtorTransport(
```

```
webSocketSession(block)))
   }
```

Appendix B. Appendix B: Ktor Server API

The Ktor server API is as follows:

```
   internal val JsonRpcServerFormatKey = AttributeKey<StringFormat>("
JsonRpcServerFormatKey")
   public fun Application.installJsonRpc(format: StringFormat) {
       pluginOrNull(WebSockets) ?: install(WebSockets)
       attributes.put(JsonRpcServerFormatKey, format)
   }
   public class JsonRpcRoute(
       webSocketSession: DefaultWebSocketServerSession,
       rpcServer: RpcServer,
   ) : DefaultWebSocketServerSession by webSocketSession, RpcServer by
rpcServer

   @KtorDsl
   public fun Route.rpc(
       path: String,
       format: StringFormat = attributes[JsonRpcClientFormatKey],
       builder: suspend JsonRpcRoute.() → Unit,
   ) = route(path) { rpc(builder) }

   @KtorDsl
   public fun Route.rpc(
       format: StringFormat = attributes[JsonRpcClientFormatKey],
       builder: suspend JsonRpcRoute.() → Unit,
   ) {
       application.pluginOrNull(WebSockets)
           ?: error("RPC for server requires $WebSockets plugin to be
installed firstly")
       webSocket {
           val server = JsonRpcServer(format, KtorTransport(this))
           builder(JsonRpcRoute(this, server))
           server.awaitCompletion()
       }
   }
```

Acknowledgments

The design approach and implementation has been heavily inspired by the kotlinx.rpc krpc implementation (https://github.com/Kotlin/kotlinx-rpc/tree/1a45abae912a783bf3f610feabaec53d8f7a53e4/krpc).

Author's Address

Youssef Shoaib
Email: canonballt@gmail.com