JSONRPC: Asynchronous RPC

JSON RPC (and other RPC invocations) are synchronous by nature - the caller invokes a remote procedure call, waits for completion and returns the result. These semantics are enforced further by some of the transports which have an explicit FSM in the message layer. For example in ZMQ the transport itself insists that every request has a matching reply.

These semantics introduce a significant performance penalty when interfaced to ODL or any other system built around the concept of asynchronous invocation. While the actual semantics (Java Futures, etc), may differ from language to language they all react badly to forced synchronization. An example of this is the netconf project in ODL which performs a forced synchronization on every operation. Instead of supplying a Future as expected by the caller and setting that Future at a future date it invokes the netconf RPC synchronously and sets the result in the Future immediately upon its creation. Rather unsurprisingly, the performance is quite dire. The performance of the initial version of JSON RPC is not as high as we would have wished it to be for the same reasons.

We intend to work around this limitation in three phases:

- Asynchronous invocation of the RPC and fully leveraging the Futures interface internally in ODL
- Using a RPC extension to convey an asynchronous invocation across JSON RPC
- Support for JSON RPC Batching

Asynchronous Invocation and Full Support for Futures

We cannot use Netty or other mechanisms used elsewhere in ODL to make IO asynchronous as they do not support all the transports we are interested in. We also do not need all of the Netty bells and whistles as our transports have their own pipelines and message framing. In fact, as various experiments on integrating Netty into ZMQ, etc have shown, Netty just gets in the way.

All we need is a simple blocking queue. While we can size the queue dynamically and do various optimizations, we believe that this should be done after the asynchronous support is feature complete and not from the beginning. So, for the time being, a simple ArrayBlockingQueue will suffice.

We separate the actual invocation of the RPC and the invocation of the interface presented by the jsonrpc plugin. The invocation of the RPC now enqueues the request and returns a simple Future. The Future is immediate only if the enqueue fails for some reason. This allows the caller to proceed and do other, more interesting things until it needs the actual result in the Future. An attempt to obtain the result by using get(), checkedGet() or their timeout-enabled versions will block until the actual RPC invocation sets the Future.

The enqueue is now separate from the actual RPC invocation. The RPC is invoked and the result is used to set the Future. Any callers currently blocked on get() will unblock. Any exceptions are tunneled to the caller as needed and presented when the consumer of the Future attempts to perform a get().

This support is now merged.

Conveying Asynchronous Invocation Beyond ODL

We need a JSON RPC 2.0 extension to convey that a call is asynchronous and it should be as transparent as possible to any legacy callers. We will avoid defining the handshake semantics until all the possible implications of taking asynchronous calls beyond ODL has been revealed.

These are our initial assumptions:

- We will leave data and the interaction between asynchronous invocation and transactions until common RPCs are feature complete
- ODL and other callers are assumed to have full support for any asynchronous RPC Extension
- Servers MAY support the asynchronous extension. An async enabled client should not need any additional support for talking to a server that does not have the async support.
- Existing JSON RPC error/result semantics should not be altered.

In order to support asynchronous invocation we introduce an additional property in the JSON RPC 2.0 JSON payload object. We will call this property "metadata". It must be a JSON Object and it is optional. Asynchronous invocation and other extensions developed for ODL and YANG modelled JSON RPC will be conveyed by various properties in the metadata JSON Object. At this time we define only one property called async which is a uuid4 for the asynchronous invocation.

Call flow:

- Client invokes a JSON RPC 2.0 call. Metadata object not present or does not contain the async property.
- Synchronous server will block and will return the result once operation is complete. This is not different from normal JSON RPC.
- Asynchronous extension capable server will return immediately a reply consisting of:
    - null result
    - id
    - metadata object with an async property set to the UUID async handle which uniquely identifies this conversation. The handle must be unique per session.
- The async enabled client receives the reply, extracts the UUID and associates it with the Future used internally to represent the result (when it arrives).
- get() and checkedGet() methods in the Future are overriden. Prior to blocking on the actual result and waiting it to be set(), they enqueue a request into the RPC queue which will produce a JSON RPC request consisting of:
    - null params
    - id
    - metadata object with an async property set to the UUID async handle for the conversation
- The async enabled server can signify that the result is not ready yet by replying with the same object as for the initial handle (null result, id and metadata). If such a result is returned, the request to query the result is re-queued by the JSON RPC plugin. This provides the required poll-for-result semantics.
- The async enabled server signifies completion by replying using a normal JSON RPC result message which does not have a metadata property.

This support is work in progress, initial patches are in ODL gerrit.

Support for Batching

Quite clearly, the introduction of queuing maps very well onto JSON RPC batching semantics (section 6 of the JSON RPC 2.0 Specification). We can introduce batching support before introducing the async extension . We do believe that this will make adding async support more difficult at a later date and have deliberately delayed it until the async extension is complete.