JSON-RPC Transport

Introduction

The "JSON/RPC 2.0 Specification" document defines an RPC mechanism, but leaves a particular transport protocol undefined. This document defines:

- An ASCII transport framing for exchanging JSON/RPC messages
- A subset of JSON/RPC features useful for reliable request/response applications
- A few compatible extensions to JSON/RPC messages:
    - Two extension fields (`string_code` and `details`) to JSON/RPC error message `data` element to improve diagnostics
- A `_Keepalive` method for monitoring connection health
- A `_Error` notification for one-sided error notifications
- A `_Info` notification for one-sided notifications (ignored by recipient)
- A `_CloseReason` notification for providing a connection close reason (error cases)
- Concrete message processing details, especially for handling errors

The resulting protocol is intended to be JSON/RPC 2.0 compatible, so that after a JSON/RPC message has been extracted from the framing, existing JSON and JSON/RPC libraries should be able to handle the messages without errors.

This document defines a transport layer method (`_Keepalive`), which can be implemented at the transport or at the application layer. An application must define other request types, including connection initialization messages. The term "application" is used in this document to refer to the specific messaging which happens over this transport.

The document also defines three transport layer notifications: `_CloseReason`, `_Error`, and `_Info`. An application may define other notification types.

Througout the document the phrase "abort the connection" means that a `_CloseReason` is sent (if possible), followed by closing of the connection.

This document assumes the following (or newer) JSON/RPC specification:

- http://www.jsonrpc.org/specification

JSON is defined by the following:

- http://www.ietf.org/rfc/rfc4627.txt

Transport layer

Connecting

A particular port or protocol is not defined by this document. An application protocol should define relevant port(s) and also define which endpoint initiates connections.

The connection MUST have reliable delivery, which is provided by e.g. TCP and SSL.

Message framing

JSON/RPC messages are framed with the following format (in the following byte-by-byte order):

- 8 bytes: ASCII lowercase hex-encoded length (LEN) of the actual JSON/RPC message (receiver MUST accept both uppercase and lowercase)
- 1 byte: a colon (":", 0x3a), not included in LEN
- LEN bytes: a JSON/RPC message, no leading or trailing whitespace
- 1 byte: a newline (0x0a), not included in LEN

Example:

```
0000000a:{"a":"b!"}\n
```

In exact binary:

```
30 30 30 30 30 30 30 61 3a 7b 22 61 22 3a 22 62 21 22 7d 0a
```

Note that the trailing newline is a part of one framed JSON/RPC message, and must be sent immediately after the message itself.

An implementation may impose a length limit on the largest acceptable incoming message due to internal buffer sizes etc. This message size limit is not indicated to the other party through the protocol, but must be known to both endpoints implicitly. If a message larger than the allowed size is received, an implementation MUST treat it as a framing error and abort the connection.

Maximum message size must be respected for both ordinary (successful) messages and error messages. Error messages and especially error `message` and `details` fields may increase message size by an unknown amount. An implementation must ensure that message size limit is respected by e.g. imposing per-field size limits which guarantees that the overall message remains small enough.

_Error notification

Both endpoints may send informative error notifications using the `_Error` method name at any time. Such notifications are informative only and MUST NOT have a control flow effect. In particular, such notifications MUST NOT directly cause a connection to be disconnected or any active method requests to be disturbed.

The purpose of these notifications is to inform the other endpoint of processing errors which are important, but do not require any action from the other endpoint. The messages are intended for both diagnostics (for understanding better what went wrong from protocol dumps and logs), but endpoints may also e.g. send alert messages from such notifications (these are not considered "control flow" effects).

JSON/RPC notifications are method calls without an `id` field. For the `_Error` notification, the following structure MUST be placed inside the `params` field:

- If the notification is related to some request (a request, or more typically, a result):
    - An `id` field may be present, containing the `id` of the related request or response message.
    - A `method` field may be present, containing the `method` name of the related request message.
    - This is relevant, for instance, when a method call result does not contain fields critical to the application, and there is no other way to inform the other endpoint about errors in the result.
- An `error` field with a JSON object value MUST be present. The `error` object MUST contain related error information in the exactly same format as in error response messages. Here, too, `string_code` and `details` are optional but recommended fields. Note that although exact error information is present, this information MUST NOT be used except for logging (and possibly alerts).

Example #1: minimal error contents:

```
{
    "jsonrpc": "2.0",
    "method": "_Error",
    "params": {
        "error": {
            "code": 1,
            "message": "ExampleMethod result is missing 'example_key'."
        }
    }
}
```

Example #2: full error contents:

```
{
    "jsonrpc": "2.0",
    "method": "_Error",
    "params": {
        "id": "pt-1",
        "method": "ExampleMethod",
        "error": {
            "code": 1,
            "message": "ExampleMethod result is missing 'example_key'.",
            "data": {
                "string_code": "INTERNAL_ERROR",
                "details": "..."
            }
        }
    }
}
```

Even though the error notification contains the original request `id`, it is only intended for diagnostics, and not programmatic use.

_Info notification

Both endpoints may send informative messages using the `_Info` method name at any time. Such notifications are informative only and MUST NOT be processed (other than logging).

The recipient of an `_Info` SHOULD NOT try to parse the notification. As such, the `params` element of the notification may contain any data. For simple textual messages, the following format is recommended:

```
{
    "jsonrpc": "2.0",
```

```
    "method": "_Info",
    "params": {
        "message": "Something interesting happened."
    }
}
```

_CloseReason notification

When a connection is closed forcibly due to some serious error, the closing endpoint MAY send a `_CloseReason` error notification to inform the other side why the connection is being closed. Because a specific notification method is used for this, the close reason can be reliably identified and e.g. used as a "cause" for a connection error.

The `_CloseReason` format is identical to the one specified for `_Error`, except that: (1) the `method` field must (of course) be set to `"_CloseReason"`, and (2) the `id` and `method` fields in the `params` object must not be sent (they are not applicable in this case).

Example with full contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32700,
            "message": "Parse error.",
            "data": {
                "string_code": "JSONRPC_PARSE_ERROR",
                "details": "optional, e.g. error at file.c:123"
            }
        }

```

```
14 |       }
   }
```

If multiple close reasons exist, it is up to the sender implementation to decide whether it sends them all or, for instance, only the first one. Similarly, it is up the receiver implementation to decide which close reason(s) to treat as definitive (for instance, first or last).

Transport layer error handling, connection abort

Errors happening at the transport layer include serious errors such as:

- Invalid framing
- Invalid/malformed JSON data
- Unknown JSON/RPC message (missing fields etc)
- Keepalive errors

The general approach to deal with serious transport layer errors is to abort the connection: write a `_CloseReason` and close the connection. The phrase **"abort the connection"** is used throughout this document for this purpose.

This approach is used instead of e.g. an `_Error` notify because continuing with the connection may cause unpredictable results. For example, if a request cannot be parsed, no reply can be sent, which may cause the peer to wait for a reply indefinitely.

These errors are conceptually implemented at the transport layer, although it is possible to implement at least keepalives in the application layer.

Processing of transport errors

The general response to a transport layer error is to:

- If possible, write a `_CloseReason` notification into the connection. The `params` object `error` field MUST contain a JSON object with the following:
    - Required: `code` set to an appropriate code

    - Required: `message` set to whatever describes the error
    - Optional: `data` (in `error` object) SHOULD contain a `string_code` and a `details` field with further information (e.g. file/line, traceback). Note that it may be difficult to include additional error details in some library implementations.
- However, if writing a `_CloseReason` notification may cause further problems (such as a fatal write error or a write operation which blocks for a considerable time) the notification MUST NOT be written.
- Close the connection afterwards.

The transport layer errors defined below all follow this basic processing. Examples are given with both minimal contents (without optional fields) and suggested contents.

The endpoint which sends the `_CloseReason` is expected to close the underlying connection. In particular, the receiving endpoint SHOULD NOT close the underlying connection as a result of receiving a `_CloseReason` (unless the connection times out otherwise).

Error handling: invalid framing or invalid JSON

If an inbound message does not conform to this framing format (e.g. LEN cannot be parsed, colon or newline does not match), or the JSON data cannot be parsed at the JSON level (e.g. unbalanced braces, missing quotes), the following error notification SHOULD be sent before disconnection.

Minimal contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32700,
            "message": "Parse error."
        }
    }
}
```

Suggested contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32700,
            "message": "Parse error.",
            "data": {
                "string_code": "JSONRPC_PARSE_ERROR",
                "details": "optional, e.g. error at file.c:123"
            }
        }
    }
}
```

Error handling: valid framing and JSON, but unknown message

If an inbound message conforms to the framing format and the JSON data can be parsed, but the message does not match supported message types (request, response, or error), the following error notification SHOULD be sent before disconnection:

Minimal contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32600,
```

```
            "message": "Invalid request."
        }
    }
}
```

Suggested contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32600,
            "message": "Invalid request.",
            "data": {
                "string_code": "JSONRPC_INVALID_REQUEST",
                "details": "optional, e.g. error at file.c:123"
            }
        }
    }
}
```

Error handling: _Keepalive timeout

If transport layer connection monitoring fails, the following error notification SHOULD be sent before disconnection:

Minimal contents:

```
{
    "jsonrpc": "2.0",
```

```
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32000,
            "message": "Keepalive timeout."
        }
    }
}
```

Suggested contents:

```
{
    "jsonrpc": "2.0",
    "method": "_CloseReason",
    "params": {
        "error": {
            "code": -32000,
            "message": "Keepalive timeout.",
            "data": {
                "string_code": "KEEPALIVE",
                "details": "optional, e.g. error at file.c:123"
            }
        }
    }
}
```

Initialization

This document does not specify a particular initialization message, or specify which party (if any) should send an initialization request. The following is, thus, just informative.

Typically some form of initialization is useful for exchanging protocol version, capability, and other similar information. The endpoint that cannot adapt to multiple versions should send the initialization request, so that the more adaptive endpoint can adapt to the required version properly.

Before the initialization message has been successfully exchanged, any other method requests should fail with error code -32601, message "Method not found." (or something similar), `string_code "JSONRPC_METHOD_NOT_FOUND"`. This protects implementations from invoking methods before version and feature negotiation is complete.

Note that `_Keepalive`, `_Info`, `_Error`, and `_CloseReason` messages MUST always be processed. In particular, a `_Keepalive` may arrive before initialization begins or completes, and must be handled normally.

Monitoring with _Keepalive

This specification reserves the method name `_Keepalive` for connection monitoring.

Both endpoints MUST independently monitor the connection status by periodically sending a `_Keepalive` request, and ensuring that a response is received within a reasonable time; if not, the connection should be closed with a `_CloseReason` as described above. The keepalive interval and timeout parameters can be chosen freely by an endpoint, and do not need to match between endpoints. Further, keepalive parameters can be adjusted dynamically by an endpoint, e.g. depending on whether the connection is in active use or not.

The `_Keepalive` request is an empty request (`"params": {}`) with an empty response (`"result": {}`). An example keepalive exchange:

```
--> { "jsonrpc": "2.0", "method": "_Keepalive", "params": {}, "id": "pt-1" }
<-- { "jsonrpc": "2.0", "result": {}, "id": "pt-1" }
```

Although not required, an implementation may also abort a connection from other events (in addition to a `_Keepalive` timeout). For example:

- A frame begins but cannot be completely received within a reasonable time.
- A lower layer indicates the connection has terminated (TCP FIN, RFCOMM close, etc).

Keepalive processing can be implemented at the application layer, because keepalives are simply normal request-response exchanges.

Request timeouts

Because keepalive monitoring ensures that any connection errors are eventually noticed, and because TCP and SSL are reliable transports, per-request timeouts are not strictly necessary unless there is some relevant application layer reason for them. Applications may, of course, still define custom timeouts for each request. Such timeouts SHOULD be longer than the keepalive timeout so that transport errors are detected as keepalive errors rather than an application layer timeout.

Primitive (non-structured) type ranges

Applications using this transport must have a common understanding of supported primitive type ranges. For instance, a specific application might restrict the allowed range for integers to 32 bits, and prohibit non-integer numbers.

If an incoming message contains values outside application imposed ranges, the receiver MUST handle this as a JSON parsing error and abort the connection.

JSON numbers

JSON does not have separate types for integers and decimal / floating point values. As a side-effect, the following are all legal representations of the integer 123, and should be accepted wherever an integer is accepted: "123", "123.00", "12300e-2", "12300E-2", "0.123e3", "0.123E3", "0.123e+3", "0.123E+3". See RFC 4627, Section 2.4 for details.

An implementation may impose constraints over the basic JSON number type, e.g.:

- Restrict numbers to be integers.
- Restrict range of integers and/or floating point values.

Numbers not conforming to application restrictions must cause a JSON parsing error.

Note in particular that if an implementation is expecting an integer, a JSON number MUST NOT be truncated, rounded, or otherwise coerced into an integer; instead, a JSON parsing error should be raised. For instance, "3.0001" MUST NOT be coerced to the integer 3.

JSON strings

An implementation SHOULD validate incoming strings to be valid UTF-8 data.

An application MAY impose additional constraints, e.g. require that a certain field only contain ASCII data.

JSON booleans

No known issues.

JSON nulls

No known issues.

Request/response and notification call styles

JSON/RPC provides both a request/response and a notification call style, the difference being that notification calls do not contain an `id` field, and thus a response is not requested. The request messages contain a `method` field in both styles, and the method names are technically from the same space. Further, in some contexts it might make sense to allow a caller to decide whether to use a request/response or a notification to call some method.

The following constraints are placed in this document for such behavior:

- The `_CloseReason`, `_Error`, and `_Info` methods MUST be called using the notification style (without an `id` field).
- The `_Keepalive` method MUST be called using the request/response style (with an `id` field).
- Behavior for application methods must be defined by the application in question, and may be any of the following:
    - a) request/response only

    - b) notification only
    - c) request/response or notification, with caller's choice

These restrictions may theoretically cause some risk of library incompatibility. If such incompatibilities are identified, the above restrictions (which by themselves are JSON/RPC compatible) can be relaxed.

Required JSON/RPC features

Overview

The following JSON/RPC features are required (more on these below):

- The `method` message (request), with a string-typed `id` field
- The `method` message (notification), without an `id` field, which must be, at minimum, ignored without errors
- The `result` message (response), with a string-typed `id` field
- The `error` message (error result for a request), with a string-typed `id` field

The following features are not required and should not be used:

- Any message with `id` having a type other than a string; in particular, the `null` value in JSON/RPC 2.0 does not need to be supported
- A `method` message with `params` other than a JSON object
- A `result` message with `result` other than a JSON object
- RPC call batches, i.e., top level object is a list of method calls
- JSON/RPC messages from versions earlier than "2.0"

As described above, an implementation may impose additional value range constraints such as rejecting non-integer numbers.

Notification ("method" without "id")

See JSON/RPC specification, example "a Notification". An endpoint MUST, at the very minimum, detect and ignore notification messages, which includes the `_CloseReason`, `_Error`, and `_Info` notification messages. There is no required processing for notifications (unless an application defines additional notifications).

An endpoint SHOULD support at least the `_CloseReason` and `_Error` notification messages, logging the notification contents and optionally sending alert or other diagnostics messages to external systems (where appropriate). An endpoint SHOULD ignore `_Info` messages (except for logging their contents).

An endpoint MUST NOT send any protocol messages (including error notifications) in response to a `_CloseReason`, `_Error`, or `_Info` notification, to avoid any chance of an "error storm".

Request ("method" with "id")

Format

See JSON/RPC specification, example "rpc call with named parameters". The following additional constraints MUST be fulfilled by endpoints:

- The `id` field must be a string. The suggested format is `"x-<num>"` where `x` is the endpoint short name (e.g. `pt` for payment terminal) and `<num>` is a counter starting from 1 and increasing after every method request sent by that party. Example: `"pt-1"`. **An `id` MUST NOT be used more than once per connection!**
- The `params` field must be a JSON object, and present in the message even when no parameters are relevant for the method (JSON/RPC allows omitting this field).
- The `method` field for application methods SHOULD be in CamelCase-format for consistency (e.g. `"ExampleMethod"`). Application method names should not contain underscore to avoid confusion with string codes (e.g. `"EXAMPLE_METHOD"`) and field names (e.g. `"example_method"`).

Example with non-empty parameters:

```
{
    "jsonrpc": "2.0",
```

```
    "method": "ExampleMethod",
    "params": {
        "example_argument": 123
    },
    "id": "pt-1"
}
```

Example with empty parameters:

```
{
    "jsonrpc": "2.0",
    "method": "ExampleMethod",
    "params": {},
    "id": "pt-1"
}
```

Receiver processing

If an incoming message does not fulfill the constraints specified above, the connection MUST be aborted. It's important to abort the connection because a reply cannot be sent, potentially leaving the caller hanging indefinitely.

For application layer errors, such as missing required application layer fields, the normal JSON/RPC `error` response MUST be used.

Response ("result")

Format

See JSON/RPC specification, example "rpc call with named parameters". The following additional constraints MUST be fulfilled by endpoints:

- The `id` field must be a string (which will automatically be the case if the request `id` was a string).

- The `result` field must be a JSON object, and present even when no result values are relevant for this method (JSON/RPC does not allow omitting this field in any case).

Example with non-empty result:

```
{
    "jsonrpc": "2.0",
    "result": {
        "example_result": 321
    },
    "id": "pt-1"
}
```

Example with empty result:

```
{
    "jsonrpc": "2.0",
    "result": {},
    "id": "pt-1"
}
```

Receiver processing

If an incoming message does not fulfill the constraints specified above, the connection MUST be aborted. It's important to abort the connection because a reply cannot be processed, potentially leaving the original request hanging.

Error ("error")

Overview of error messages in JSON/RPC

The JSON/RPC specification requires the `code` and `message` fields for errors. Code is an integer with a few specific error codes, a range for implementation-specific server errors, and a range for application errors. Message is a textual description of the error which is required.

This specification defines, in addition to these, two new error related fields which are placed in the JSON/RPC error `data` object: `string_code` and `details`. String code identifies an error with a machine-processable string identifier, consisting of capital ASCII letters separated by underscores, e.g. `INTERNAL_ERROR`. Details is a human-readable string which contains diagnostics information, such as tracebacks, file/line information, etc. Details can be fairly large (several kilobytes) and is intended to ensure that as much information as possible is obtained for rarely occurring, difficult-to-reproduce errors.

Errors are processed primarily based on `string_code`, which deviates from normal JSON/RPC processing.

The `data` object may also contain additional implementation or application specific values related to the error. For instance, a fictional `AMOUNT_TOO_HIGH` error might contain `requested_amount` and `limit` fields. Such values SHOULD be delivered to the application layer by the transport.

Format

The following additional constraints MUST be fulfilled by endpoints on top of the JSON/RPC error message requirements:

- The `id` field must be a string (which will automatically be the case if the request `id` was a string), and always present. If an incoming request does not have an `id` field or has an `id` field of invalid type, the connection MUST be aborted. Only "normal" error messages (which do not lead to disconnection) are thus considered in this section.
- The `error` field must be a JSON object, containing at least the `code` and `message` fields.
- The `code` field MUST be an integer in the 32-bit signed integer range (`-2^31 ... 2^31-1`).
- The `message` field MUST be a string (may be empty if not description is available).

- The `error` object SHOULD also contain a `data` field. If present, the `data` field MUST be a JSON object value, and contain the following:

    - The `string_code` SHOULD be present. If present, the `string_code` MUST be a string, at most 64 characters long. This field is heavily recommended because error processing is primarily based on the string code of the error. However, an error receiver MUST tolerate a missing "string_code" field.

    - The `details` field MAY be present. If present, the `details` MUST be a string.
- The `data` field (if present) MAY also contain additional implementation or application specific fields. which, if not supported, MUST be ignored without any side-effects.

For application errors, the `code` field SHOULD always be 1, unless a better application specific error code has been defined.

Note that tracebacks or other long diagnostics information in the `details` field may make the message very large (more than 10 kilobytes, for instance). A sender MUST ensure that the entire message never exceeds the message size limit of the receiver. However, if possible, a sender SHOULD include error details (such as tracebacks, file/line information of error, etc) to help debugging. A straightforward way to cap message size is to impose field size limits independently for `string_code` and `message` fields.

Example of a full error with all optional fields:

```
{
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "Parameter X has invalid format (example).",
        "data": {
            "string_code": "PARAMETER_FORMAT",
            "details": "Error occurred in file.c line 123."
        }
    },
    "id": "pt-1"
}
```

Example of a minimal error with no optional fields (not recommended, but allowed):

```
{
    "jsonrpc": "2.0",

```

```
    "error": {
        "code": 1,
        "message": "Parameter X has invalid format (example).",
    },
    "id": "pt-1"
}
```

Example of a full error with all optional fields and application specific additional values:

```
{
    "jsonrpc": "2.0",
    "error": {
        "code": 1,
        "message": "Requested amount is too high.",
        "data": {
            "string_code": "AMOUNT_TOO_HIGH",
            "details": "Error occurred in file.c line 123.",
            "requested_amount": 5000,
            "limit": 1000
        }
    },
    "id": "pt-1"
}
```

Error sender processing

When request processing fails, the error sender (receiver of the original request) needs to determine which error occurred and set the JSON/RPC error message fields appropriately:

- `code`: for application errors, set to 1, unless an application-specific code has been defined. For specific server or implementation level errors, set to one of the pre-defined error codes in the JSON/RPC specification (-32700, -32600, -32601, -32602, or -32603), or the keepalive error code defined in this document (-32000).
- `message`: application dependent human-readable string value. If a value cannot be determined, use the empty string, as `message` is a required field.
- `string_code`: set to some all-caps-and-underscores error identifier dependent on the error type or `"UNKNOWN"` as a fallback. For specific server or implementation level errors (see above), the mapping shown below should be used. Note that this field, if present, is the normative field which governs receiver processing of the error, so it is important to describe errors accurately. In special cases this field MAY be missing.
- `details`: additional human-readable string intended mainly for debugging and diagnostics. This field is intended to contain free-form verbose information such as tracebacks, file and line information, essentially anything useful for debugging. This field is optional and MAY be missing even in normal situations.
- An implementation MAY also provide additional error information in any implementation or application specific format as additional fields in the `data` object. An implementation SHOULD NOT insert additional fields into the `error` object, as this is probably more difficult to interoperate with (at least some) existing libraries.

The `data` field needs to be present if `string_code`, `details`, or any additional implementation or application specific additional error fields are present. The `data` field may or may not be present otherwise.

Error receiver processing

When receiving an error message, the error receiver (typically the sender of the offending request) MUST process it as follows:

- Perform basic validation of fields. If mandatory fields are missing, abort connection.
- If `string_code` is available:
    - Ignore the `code` field, and process the error based on the `string_code` field.
- If `string_code` is not available (`data` not present, or `string_code` missing from the `data` element):
    - Process the error based on the `code` field.

    - Note: an implementation may map the `code` to `string_code` as described below, and then proceed as if `string_code` had been present. This is not required behavior.
- Fields other than `string_code` and `code` MUST be ignored in error processing code (however, other fields may of course be included in logs, error messages, etc).
- Any fields other than `string_code` or `details` in the `data` object MUST be ignored, if not understood by the error receiver. Similarly, any fields other than `code`, `message`, and `data` in the `error` object MUST be ignored if not understood (although some library implementations may not make this easy).

Mapping "code" to "string_code"

The following special codes defined in the JSON/RPC MUST be mapped as follows:

- Code -32700 => `"JSONRPC_PARSE_ERROR"`
- Code -32600 => `"JSONRPC_INVALID_REQUEST"`
- Code -32601 => `"JSONRPC_METHOD_NOT_FOUND"`
- Code -32602 => `"JSONRPC_INVALID_PARAMS"`
- Code -32603 => `"INTERNAL_ERROR"`

The following special code is defined in this specification for keepalive errors:

- Code -32000 => `"KEEPALIVE"`

The following default mapping should be used for all other codes, unless there is an implementation or application specific error code with a more accurate mapping:

- Any other code => `"UNKNOWN"`

Note that this includes implementation-defined server errors, codes reserved for pre-defined errors, and application-defined errors, other than described above.

Deprecated features

- The informative `response_to` field in method responses has been deprecated. The field is non-programmatic, i.e. it must not be processed programmatically.