Independent Submission
Request for Comments: 7047
Category: Informational
ISSN: 2070-1721

B. Pfaff
B. Davie, Ed.
VMware, Inc.

December 2013

The Open vSwitch Database Management Protocol

Abstract

Open vSwitch is an open-source software switch designed to be used as a vswitch (virtual switch) in virtualized server environments. A vswitch forwards traffic between different virtual machines (VMs) on the same physical host and also forwards traffic between VMs and the physical network. Open vSwitch is open to programmatic extension and control using OpenFlow and the OVSDB (Open vSwitch Database) management protocol. This document defines the OVSDB management protocol. The Open vSwitch project includes open-source OVSDB client and server implementations.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7047.

Table of Contents

1. Introduction

In virtualized server environments, it is typically required to use a vswitch (virtual switch) to forward traffic between different virtual machines (VMs) on the same physical host and between VMs and the physical network. Open vSwitch [OVS] is an open-source software switch designed to be used as a vswitch in such environments. Open vSwitch (OVS) is open to programmatic extension and control using OpenFlow [OF-SPEC] and the OVSDB (Open vSwitch Database) management protocol. This document defines the OVSDB management protocol. The Open vSwitch project includes open-source OVSDB client and server implementations.

The OVSDB management protocol uses JSON [RFC4627] for its wire format and is based on JSON-RPC version 1.0 [JSON-RPC].

The schema of the Open vSwitch database is documented in [DB-SCHEMA]. This document specifies the protocol for interacting with that database for the purposes of managing and configuring Open vSwitch instances. The protocol specified in this document also provides means for discovering the schema in use, as described in Section 4.1.2.

The OVSDB management protocol is intended to allow programmatic access to the Open vSwitch database as documented in [DB-SCHEMA]. This database holds the configuration for one Open vSwitch daemon. As currently defined, this information describes the switching behavior of a virtual switch and does not describe the behavior or configuration of a routing system. In the event that the schema is extended in a future release to cover elements of the routing system, implementers and operators need to be aware of the work of the IETF's I2RS working group that specifies protocols and data models for real-time or event driven interaction with the routing system.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.2. Terminology

UUID: Universally Unique Identifier. A 128-bit identifier that is unique in space and time [DCE].

OVS: Open vSwitch. An open-source virtual switch.

OVSDB: The database that is used for the purpose of configuring OVS instances.

JSON: Javascript Object Notation [RFC4627].

JSON-RPC: JSON Remote Procedure Call [JSON-RPC].

Durable: Reliably written to non-volatile storage (e.g., disk). OVSDB supports the option to specify whether or not transactions are durable.

Note that the JSON specification [RFC4627] provides precise definitions of a number of important terms such as JSON values, objects, arrays, numbers, and strings. In all cases, this document uses the definitions from [RFC4627].

2. System Overview

Figure 1 illustrates the main components of Open vSwitch and the interfaces to a control and management cluster. An OVS instance comprises a database server (ovsdb-server), a vswitch daemon (ovs-vswitchd), and, optionally, a module that performs fast-path forwarding. The "management and control cluster" consists of some number of managers and controllers. Managers use the OVSDB management protocol to manage OVS instances. An OVS instance is managed by at least one manager. Controllers use OpenFlow to install flow state in OpenFlow switches. An OVS instance can support multiple logical datapaths, referred to as "bridges". There is at least one controller for each OpenFlow bridge.

The OVSDB management interface is used to perform management and configuration operations on the OVS instance. Compared to OpenFlow, OVSDB management operations occur on a relatively long timescale. Examples of operations that are supported by OVSDB include:

- Creation, modification, and deletion of OpenFlow datapaths (bridges), of which there may be many in a single OVS instance;
- Configuration of the set of controllers to which an OpenFlow datapath should connect;
- Configuration of the set of managers to which the OVSDB server should connect;
- Creation, modification, and deletion of ports on OpenFlow datapaths;

- Creation, modification, and deletion of tunnel interfaces on OpenFlow datapaths;
- Creation, modification, and deletion of queues;
- Configuration of QoS (quality of service) policies and attachment of those policies to queues; and
- Collection of statistics.

OVSDB does not perform per-flow operations, leaving those instead to OpenFlow.

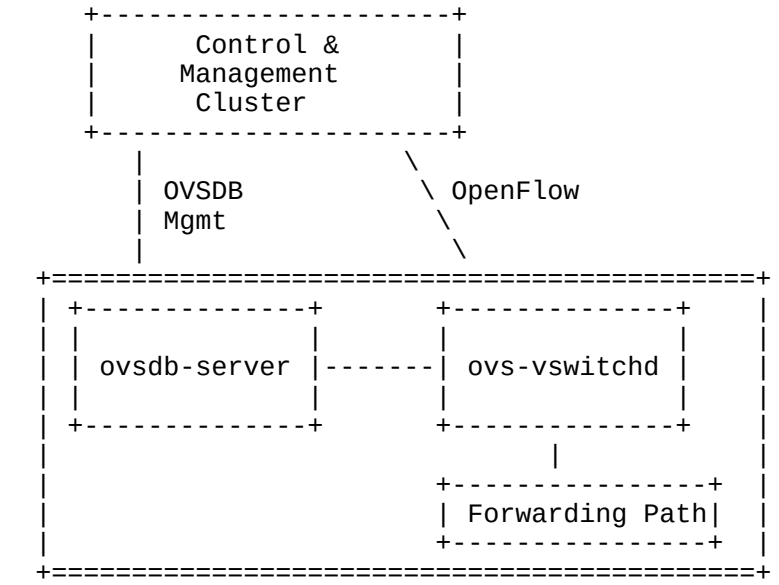

Figure 1: Open vSwitch Interfaces

Further information about the usage of the OVSDB management protocol is provided in [DB-SCHEMA].

3. OVSDB Structure

This section outlines the overall structure of databases in OVSDB. As described here, the database is reasonably generic. For the complete and current description of the database schema as used in OVS, refer to [DB-SCHEMA]. See also Section 4.1.2 for information on how the OVSDB management protocol may be used to discover the schema currently in use.

3.1. JSON Usage

OVSDB uses JSON [RFC4627] for both its schema format and its wire protocol format. The JSON implementation in Open vSwitch has the following limitations:

- Null bytes (\u0000) SHOULD NOT be used in strings.
- Only UTF-8 encoding is supported.

The descriptions below use the following shorthand notations for JSON values. Terminology follows [RFC4627].

<string>
A JSON string. Any Unicode string is allowed. Implementations SHOULD disallow null bytes.

<id>
A JSON string matching [a-zA-Z_][a-zA-Z0-9_]*. <id>s that begin with _ are reserved to the implementation and MUST NOT be used by the user.

<version>
A JSON string that contains a version number that matches [0-9]+\.[0-9]+\.[0-9]+

<boolean>
A JSON true or false value.

<number>
A JSON number.

<integer>
A JSON number with an integer value, within the range -(2**63)...+(2**63)-1.

<json-value>
Any JSON value.

<nonnull-json-value>
Any JSON value except null.

<error>
A JSON object with the following members:

```
"error": <string>          required
"details": <string>        optional
```

The value of the "error" member is a short string, specified in this document, that broadly indicates the class of the error. Most "error" strings are specific to contexts described elsewhere in this document, but the following "error" strings may appear in any context where an <error> is permitted:

"error": "resources exhausted"
The operation requires more resources (memory, disk, CPU, etc.) than are currently available to the database server.

"error": "I/O error"
Problems accessing the disk, network, or other required resources prevented the operation from completing.

Database implementations MAY use "error" strings not specified in this document to indicate errors that do not fit into any of the specified categories. Optionally, an <error> MAY include a "details" member, whose value is a string that describes the error in more detail for the benefit of a human user or administrator. This document does not specify the format or content of the "details" string. An <error> MAY also have other members that describe the error in more detail. This document does not specify the names or values of these members.

3.2. Schema Format

An Open vSwitch configuration database consists of a set of tables, each of which has a number of columns and zero or more rows. A schema for the database is represented by <database-schema>, as described below.

<database-schema>
A JSON object with the following members:

```
"name": <id>                              required
"version": <version>                      required
"cksum": <string>                         optional
"tables": {<id>: <table-schema>, ...}     required
```

The "name" identifies the database as a whole. It must be provided to most JSON-RPC requests to identify the database being operated on.

The "version" reports the version of the database schema. It is REQUIRED to be present. Open vSwitch semantics for "version" are described in [DB-SCHEMA]. Other schemas may use it differently.

The "cksum" optionally reports an implementation-defined checksum for the database schema. Its use is primarily as a tool for schema developers, and clients SHOULD ignore it.

The value of "tables" is a JSON object whose names are table names and whose values are <table-schema>s.

<table-schema>
A JSON object with the following members:

"columns": {<id>: <column-schema>, ...}	required
"maxRows": <integer>	optional
"isRoot": <boolean>	optional
"indexes": [<column-set>*]	optional

The value of "columns" is a JSON object whose names are column names and whose values are <column-schema>s.

Every table has the following columns whose definitions are not included in the schema:

"_uuid": This column, which contains exactly one UUID value, is initialized to a random value by the database engine when it creates a row. It is read-only, and its value never changes during the lifetime of a row.

"_version": Like "_uuid", this column contains exactly one UUID value, initialized to a random value by the database engine when it creates a row, and it is read-only. However, its value changes to a new random value whenever any other field in the row changes. Furthermore, its value is ephemeral: when the database is closed and reopened, or when the database process is stopped and then started again, each "_version" also changes to a new random value.

If "maxRows" is specified, as a positive integer, it limits the maximum number of rows that may be present in the table. This is a "deferred" constraint, enforced only at transaction commit time (see the "transact" request in Section 4.1.3). If "maxRows" is not specified, the size of the table is limited only by the resources available to the database server. "maxRows" constraints are enforced after unreferenced rows are deleted from tables with a false "isRoot".

The "isRoot" boolean is used to determine whether rows in the table require strong references from other rows to avoid garbage collection. (See the discussion of "strong" and "weak" references below in the description of <base-type>.) If "isRoot" is

specified as true, then rows in the table exist independent of any references (they can be thought of as part of the "root set" in a garbage collector). If "isRoot" is omitted or specified as false, then any given row in the table may exist only when there is at least one reference to it, with refType "strong", from a different row (in the same table or a different table). This is a "deferred" action: unreferenced rows in the table are deleted just before transaction commit.

For compatibility with schemas created before "isRoot" was introduced, if "isRoot" is omitted or false in every <table-schema> in a given <database-schema>, then every table is part of the root set.

If "indexes" is specified, it must be an array of zero or more <column-set>s. A <column-set> is an array of one or more strings, each of which names a column. Each <column-set> is a set of columns whose values, taken together within any given row, must be unique within the table. This is a "deferred" constraint, enforced only at transaction commit time, after unreferenced rows are deleted and dangling weak references are removed. Ephemeral columns may not be part of indexes.

<column-schema>
A JSON object with the following members:

"type": <type>	required
"ephemeral": <boolean>	optional
"mutable": <boolean>	optional

The "type" specifies the type of data stored in this column.

If "ephemeral" is specified as true, then this column's values are not guaranteed to be durable; they may be lost when the database restarts. A column whose type (either key or value) is a strong reference to a table that is not part of the root set is always durable, regardless of this value. (Otherwise, restarting the database could lose entire rows.)

If "mutable" is specified as false, then this column's values may not be modified after they are initially set with the "insert" operation.

<type>
The type of a database column. Either an <atomic-type> or a JSON object that describes the type of a database column, with the following members:

```
"key": <base-type>                required
"value": <base-type>              optional
"min": <integer>                  optional
"max": <integer> or "unlimited"   optional
```

If "min" or "max" is not specified, each defaults to 1. If "max" is specified as "unlimited", then there is no specified maximum number of elements, although the implementation will enforce some limit. After considering defaults, "min" must be exactly 0 or exactly 1, "max" must be at least 1, and "max" must be greater than or equal to "min".

If "min" and "max" are both 1 and "value" is not specified, the type is the scalar type specified by "key".

If "min" is not 1 or "max" is not 1, or both, and "value" is not specified, the type is a set of scalar type "key".

If "value" is specified, the type is a map from type "key" to type "value".

<base-type>
The type of a key or value in a database column. Either an <atomic-type> or a JSON object with the following members:

```
"type": <atomic-type>            required
"enum": <value>                  optional
"minInteger": <integer>          optional, integers only
"maxInteger": <integer>          optional, integers only
"minReal": <real>                optional, reals only
"maxReal": <real>                optional, reals only
"minLength": <integer>           optional, strings only
"maxLength": <integer>           optional, strings only
"refTable": <id>                 optional, UUIDs only
"refType": "strong" or "weak"    optional, only with "refTable"
```

An <atomic-type> by itself is equivalent to a JSON object with a single member "type" whose value is the <atomic-type>.

"enum" may be specified as a <value> whose type is a set of one or more values specified for the member "type". If "enum" is specified, then the valid values of the <base-type> are limited to those in the <value>.

"enum" is mutually exclusive with the following constraints:

If "type" is "integer", then "minInteger" or "maxInteger" or both may also be specified, restricting the valid integer range. If both are specified, then "maxInteger" must be greater than or equal to "minInteger".

If "type" is "real", then "minReal" or "maxReal" or both may also be specified, restricting the valid real range. If both are specified, then "maxReal" must be greater than or equal to "minReal".

If "type" is "string", then "minLength" and "maxLength" or both may be specified, restricting the valid length of value strings. If both are specified, then "maxLength" must be greater than or equal to "minLength". String length is measured in characters.

If "type" is "uuid", then "refTable", if present, must be the name of a table within this database. If "refTable" is specified, then "refType" may also be specified. If "refTable" is set, the effect depends on "refType":

+ If "refType" is "strong" or if "refType" is omitted, the allowed UUIDs are limited to UUIDs for rows in the named table.

+ If "refType" is "weak", then any UUIDs are allowed, but UUIDs that do not correspond to rows in the named table will be automatically deleted. When this situation arises in a map, both the key and the value will be deleted from the map.

"refTable" constraints are "deferred" constraints: they are enforced only at transaction commit time (see the "transact" request in Section 4.1.3). The other constraints on <base-type> are "immediate", enforced immediately by each operation.

<atomic-type>
One of the strings "integer", "real", "boolean", "string", or "uuid", representing the specified scalar type.

4. Wire Protocol

The database wire protocol is implemented in JSON-RPC 1.0 [JSON-RPC]. While the JSON-RPC specification allows a range of transports, implementations of this specification SHOULD operate directly over TCP. See Section 6 for discussion of the TCP port.

4.1. RPC Methods

The following subsections describe the RPC methods that are supported. As described in the JSON-RPC 1.0 specification, each request comprises a string containing the name of the method, a (possibly null) array of parameters to pass to the method, and a request ID, which can be used to match the response to the request. Each response comprises a result object (non-null in the event of a successful invocation), an error object (non-null in the event of an error), and the ID of the matching request. More details on each method, its parameters, and its results are described below.

An OVSDB server MUST implement all of the following methods. An OVSDB client MUST implement the "Echo" method and is otherwise free to implement whichever methods suit the implementation’s needs.

The operations that may be performed on the OVS database using these methods (e.g., the "transact" method) are described in Section 5.

4.1.1. List Databases

This operation retrieves an array whose elements are the names of the databases that can be accessed over this management protocol connection.

The request object contains the following members:

- "method": "list_dbs"
- "params": []
- "id": <nonnull-json-value>

The response object contains the following members:

- "result": [<db-name>,...]
- "error": null
- "id": same "id" as request

4.1.2. Get Schema

This operation retrieves a <database-schema> that describes hosted database <db-name>.

The request object contains the following members:

- "method": "get_schema"
- "params": [<db-name>]
- "id": <nonnull-json-value>

The response object contains the following members:

- "result": <database-schema>
- "error": null
- "id": same "id" as request

In the event that the database named in the request does not exist, the server sends a JSON-RPC error response of the following form:

- "result": null
- "error": "unknown database"
- "id": same "id" as request

4.1.3. Transact

This RPC method causes the database server to execute a series of operations in the specified order on a given database.

The request object contains the following members:

- "method": "transact"
- "params": [<db-name>, <operation>*]
- "id": <nonnull-json-value>

The value of "id" MUST be unique among all in-flight transactions within the current JSON-RPC session. Otherwise, the server may return a JSON-RPC error.

The "params" array for this method consists of a <db-name> that identifies the database to which the transaction applies, followed by zero or more JSON objects, each of which represents a single database operation. Section 5 describes the valid operations. The database server executes each of the specified operations in the specified order, except if an operation fails, then the remaining operations are not executed. The set of operations is executed as a single atomic, consistent, isolated transaction. The transaction is committed if and only if every operation succeeds. Durability of the commit is not guaranteed unless the "commit" operation, with "durable" set to true, is included in the operation set. See Section 5 for more discussion of the database operations.

The response object contains the following members:

- "result": [<object>*]
- "error": null
- "id": same "id" as request

Regardless of whether errors occur in the database operations, the response is always a JSON-RPC response with null "error" and a "result" member that is an array with the same number of elements as "params". Each element of the "result" array corresponds to the same element of the "params" array. The "result" array elements may be interpreted as follows:

- A JSON object that does not contain an "error" member indicates that the operation completed successfully. The specific members of the object are specified below in the descriptions of individual operations. Some operations do not produce any results, in which case the object will have no members.
- An <error> indicates that the matching operation completed with an error.
- A JSON null value indicates that the operation was not attempted because a prior operation failed.

In general, "result" contains some number of successful results, possibly followed by an error, in turn followed by enough JSON null values to match the number of elements in "params". There is one exception: if all of the operations succeed, but the results cannot be committed, then "result" will have one more element than "params", with the additional element being an <error>. In this case, the possible "error" strings include the following:

"error": "referential integrity violation"
When the commit was attempted, a column's value referenced the UUID for a row that did not exist in the table named by the column's <base-type> key or value "refTable" that has a "refType" of "strong". (This can be caused by inserting a row that references a nonexistent row, by deleting a row that is still referenced by another row, by specifying the UUID for a row in the wrong table, and other ways.)

"error": "constraint violation"
A number of situations can arise in which the attempted commit would lead to a constraint on the database being violated. (See Section 3.2 for more discussion of constraints.) These situations include:

* The number of rows in a table exceeds the maximum number permitted by the table's "maxRows" value.

* Two or more rows in a table had the same values in the columns that comprise an index.

* A column with a <base-type> key or value "refTable" whose "refType" is "weak" became empty due to deletion(s), and this column is not allowed to be empty because its <type> has a "min" of 1. Such deletions may be the result of rows that it referenced being deleted (or never having existed, if the column's row was inserted within the transaction).

"error": "resources exhausted"
The operation requires more resources (memory, disk, CPU, etc.) than are currently available to the database server.

"error": "I/O error"
Problems accessing the disk, network, or other required resources prevented the operation from completing.

If "params" contains one or more "wait" operations, then the transaction may take an arbitrary amount of time to complete. The database implementation MUST be capable of accepting, executing, and replying to other transactions and other JSON-RPC requests while a transaction or transactions containing "wait" operations are outstanding on the same or different JSON-RPC sessions.

4.1.4. Cancel

The "cancel" method is a JSON-RPC notification, i.e., no matching response is provided. It instructs the database server to immediately complete or cancel the "transact" request whose "id" is the same as the notification's "params" value. The notification object has the following members:

- "method": "cancel"
- "params": [the "id" for an outstanding request]
- "id": null

If the "transact" request can be completed immediately, then the server sends a response in the form described for "transact" (Section 4.1.3). Otherwise, the server sends a JSON-RPC error response of the following form:

- "result": null
- "error": "canceled"
- "id": the "id" member of the canceled request.

The "cancel" notification itself has no reply.

4.1.5. Monitor

The "monitor" request enables a client to replicate tables or subsets of tables within an OVSDB database by requesting notifications of changes to those tables and by receiving the complete initial state of a table or a subset of a table. The request object has the following members:

- "method": "monitor"
- "params": [<db-name>, <json-value>, <monitor-requests>]
- "id": <nonnull-json-value>

The <json-value> parameter is used to match subsequent update notifications (see below) to this request. The <monitor-requests> object maps the name of the table to be monitored to an array of <monitor-request> objects.

Each <monitor-request> is an object with the following members:

```
"columns": [<column>*]            optional
"select": <monitor-select>        optional
```

The columns, if present, define the columns within the table to be monitored. <monitor-select> is an object with the following members:

```
"initial": <boolean>              optional
"insert": <boolean>               optional
"delete": <boolean>               optional
"modify": <boolean>               optional
```

The contents of this object specify how the columns or table are to be monitored, as explained in more detail below.

The response object has the following members:

- "result": <table-updates>
- "error": null
- "id": same "id" as request

The <table-updates> object is described in detail in Section 4.1.6. It contains the contents of the tables for which "initial" rows are selected. If no tables' initial contents are requested, then "result" is an empty object.

Subsequently, when changes to the specified tables are committed, the changes are automatically sent to the client using the "update" monitor notification (see Section 4.1.6). This monitoring persists until the JSON-RPC session terminates or until the client sends a "monitor_cancel" JSON-RPC request.

Each <monitor-request> specifies one or more columns and the manner in which the columns (or the entire table) are to be monitored. The "columns" member specifies the columns whose values are monitored. It MUST NOT contain duplicates. If "columns" is omitted, all columns in the table, except for "_uuid", are monitored. The circumstances in which an "update" notification is sent for a row within the table are determined by <monitor-select>:

- If "initial" is omitted or true, every row in the table is sent as part of the response to the "monitor" request.
- If "insert" is omitted or true, "update" notifications are sent for rows newly inserted into the table.

- If "delete" is omitted or true, "update" notifications are sent for rows deleted from the table.

- If "modify" is omitted or true, "update" notifications are sent whenever a row in the table is modified.

If there is more than one <monitor-request> in an array, then each <monitor-request> in the array should specify both "columns" and "select", and the "columns" MUST be non-overlapping sets.

4.1.6. Update Notification

The "update" notification is sent by the server to the client to report changes in tables that are being monitored following a "monitor" request as described above. The notification has the following members:

- "method": "update"

- "params": [<json-value>, <table-updates>]

- "id": null

The <json-value> in "params" is the same as the value passed as the <json-value> in "params" for the corresponding "monitor" request. <table-updates> is an object that maps from a table name to a <table-update>. A <table-update> is an object that maps from the row's UUID to a <row-update> object. A <row-update> is an object with the following members:

```
 "old": <row>   present for "delete" and "modify" updates
 "new": <row>   present for "initial", "insert", and "modify" updates
```

The format of <row> is described in Section 5.1.

Each table in which one or more rows has changed (or whose initial view is being presented) is represented in <table-updates>. Each row that has changed (or whose initial view is being presented) is represented in its <table-update> as a member with its name taken from the row's "_uuid" member. The corresponding value is a <row-update>:

- The "old" member is present for "delete" and "modify" updates. For "delete" updates, each monitored column is included. For "modify" updates, the prior value of each monitored column whose value has changed is included (monitored columns that have not changed are represented in "new").

- The "new" member is present for "initial", "insert", and "modify" updates. For "initial" and "insert" updates, each monitored column is included. For "modify" updates, the new value of each monitored column is included.

Note that initial views of rows are not presented in update notifications, but in the response object to the monitor request. The formatting of the <table-updates> object, however, is the same in either case.

4.1.7. Monitor Cancellation

The "monitor_cancel" request cancels a previously issued monitor request. The request object members are:

- "method": "monitor_cancel"
- "params": [<json-value>]
- "id": <nonnull-json-value>

The <json-value> in "params" matches the <json-value> in "params" for the ongoing "monitor" request that is to be canceled. No more "update" messages will be sent for this table monitor. The response to this request has the following members:

- "result": {}
- "error": null
- "id": the request "id" member

In the event that a monitor cancellation request refers to an unknown monitor request, an error response with the following members is returned:

- "result": null
- "error": "unknown monitor"
- "id": the request "id" member

4.1.8. Lock Operations

Three RPC methods, "lock", "steal", and "unlock", provide support to clients to perform locking operations on the database. The database server supports an arbitrary number of locks, each of which is identified by a client-defined ID. At any given time, each lock may

have at most one owner. The precise usage of a lock is determined by the client. For example, a set of clients may agree that a certain table can only be written by the owner of a certain lock. OVSDB itself does not enforce any restrictions on how locks are used -- it simply ensures that a lock has at most one owner.

The RPC request objects have the following members:

- "method": "lock", "steal", or "unlock"
- "params": [<id>]
- "id": <nonnull-json-value>

The response depends on the request and has the following members:

- "result": {"locked": boolean} for "lock"
- "result": {"locked": true} for "steal"
- "result": {} for "unlock"
- "error": null
- "id": same "id" as request

The three methods operate as follows:

- "lock": The database will assign this client ownership of the lock as soon as it becomes available. When multiple clients request the same lock, they will receive it in first-come, first-served order.
- "steal": The database immediately assigns this client ownership of the lock. If there is an existing owner, it loses ownership.
- "unlock": If the client owns the lock, this operation releases it. If the client has requested ownership of the lock, this cancels the request.

  (Closing or otherwise disconnecting a database client connection unlocks all of its locks.)

For any given lock, the client MUST alternate "lock" or "steal" operations with "unlock" operations. That is, if the previous operation on a lock was "lock" or "steal", it MUST be followed by an "unlock" operation, and vice versa.

For a "lock" operation, the "locked" member in the response object is true if the lock has already been acquired and false if another client holds the lock and the client's request for it was queued. In the latter case, the client will be notified later with a "locked" message (Section 4.1.9) when acquisition succeeds.

These requests complete and send a response quickly, without waiting. The "locked" and "stolen" notifications (see below) report asynchronous changes to ownership.

Note that the scope of a lock is a database server, not a database hosted by that server. A client may choose to implement a naming convention, such as "<db-name>__<lock-name>", which can effectively limit the scope of a lock to a particular database.

4.1.9. Locked Notification

The "locked" notification is provided to notify a client that it has been granted a lock that it had previously requested with the "lock" method described above. The notification has the following members:

- "method": "locked"
- "params": [<id>]
- "id": null

"Params" contains the name of the lock that was given in the "lock" request. The notified client now owns the lock named in "params".

The database server sends this notification after the reply to the corresponding "lock" request (but only if the "locked" member of the response was false) and before the reply to the client's subsequent "unlock" request.

4.1.10. Stolen Notification

The "stolen" notification is provided to notify a client, which had previously obtained a lock, that another client has stolen ownership of that lock. The notification has the following members:

- "method": "stolen"
- "params": [<id>]
- "id": null

The notified client no longer owns the lock named in "params". The client MUST still issue an "unlock" request before performing any subsequent "lock" or "steal" operation on the lock.

If the client originally obtained the lock through a "lock" request, then it will automatically regain the lock later after the client that stole it releases it. (The database server will send the client a "locked" notification at that point to let it know.)

If the client originally obtained the lock through a "steal" request, the database server won't automatically reassign it ownership of the lock when it later becomes available. To regain ownership, the client must "unlock" and then "lock" or "steal" the lock again.

4.1.11. Echo

The "echo" method can be used by both clients and servers to verify the liveness of a database connection. It MUST be implemented by both clients and servers. The members of the request are:

- "method": "echo"
- "params": JSON array with any contents
- "id": <json-value>

The response object has the following members:

- "result": same as "params"
- "error": null
- "id": the request "id" member

5. Database Operations

This section describes the operations that may be specified in the "transact" method described in Section 4.1.3.

5.1. Notation

We introduce the following notation for the discussion of operations.

<db-name>
: An <id> that names a database. The valid <db-name>s can be obtained using a "list_dbs" request. The <db-name> is taken from the "name" member of <database-schema>.

<table>
An <id> that names a table.

<column>
An <id> that names a table column.

<row>
A JSON object that describes a table row or a subset of a table row. Each member is the name of a table column paired with the <value> of that column.

<value>
A JSON value that represents the value of a column in a table row, one of <atom>, <set>, or <map>.

<atom>
A JSON value that represents a scalar value for a column, one of <string>, <number>, <boolean>, <uuid>, or <named-uuid>.

<set>
Either an <atom>, representing a set with exactly one element, or a 2-element JSON array that represents a database set value. The first element of the array must be the string "set", and the second element must be an array of zero or more <atom>s giving the values in the set. All of the <atom>s must have the same type.

<map>
A 2-element JSON array that represents a database map value. The first element of the array must be the string "map", and the second element must be an array of zero or more <pair>s giving the values in the map. All of the <pair>s must have the same key and value types.

(JSON objects are not used to represent <map> because JSON only allows string names in an object.)

<pair>
A 2-element JSON array that represents a pair within a database map. The first element is an <atom> that represents the key, and the second element is an <atom> that represents the value.

<uuid>
A 2-element JSON array that represents a UUID. The first element of the array must be the string "uuid", and the second element must be a 36-character string giving the UUID in the format described by RFC 4122 [RFC4122]. For example, the following <uuid> represents the UUID 550e8400-e29b-41d4-a716-446655440000:

["uuid", "550e8400-e29b-41d4-a716-446655440000"]

<named-uuid>
A 2-element JSON array that represents the UUID of a row inserted in an "insert" operation within the same transaction. The first element of the array must be the string "named-uuid", and the second element should be the <id> specified as the "uuid-name" for an "insert" operation within the same transaction. For example, if an "insert" operation within this transaction specifies a "uuid-name" of "myrow", the following <named-uuid> represents the UUID created by that operation:

["named-uuid", "myrow"]

A <named-uuid> may be used anywhere a <uuid> is valid. This enables a single transaction to both insert a new row and then refer to that row using the "uuid-name" that was associated with that row when it was inserted. Note that the "uuid-name" is only meaningful within the scope of a single transaction.

<condition>
A 3-element JSON array of the form [<column>, <function>, <value>] that represents a test on a column value. Except as otherwise specified below, <value> MUST have the same type as <column>. The meaning depends on the type of <column>:

integer or real
<function> must be "<", "<=", "==", "!=", ">=", ">", "includes", or "excludes".

The test is true if the column's value satisfies the relation <function> <value>, e.g., if the column has value 1 and <value> is 2, the test is true if <function> is "<", "<=", or "!=", but not otherwise.

"includes" is equivalent to "=="; "excludes" is equivalent to "!=".

boolean or string or uuid
<function> must be "!=", "==", "includes", or "excludes".

If <function> is "==" or "includes", the test is true if the column's value equals <value>. If <function> is "!=" or "excludes", the test is inverted.

set or map
<function> must be "!=", "==", "includes", or "excludes".

If <function> is "==", the test is true if the column's value contains exactly the same values (for sets) or pairs (for maps). If <function> is "!=", the test is inverted.

If <function> is "includes", the test is true if the column's value contains all of the values (for sets) or pairs (for maps) in <value>. The column's value may also contain other values or pairs.

If <function> is "excludes", the test is true if the column's value does not contain any of the values (for sets) or pairs (for maps) in <value>. The column's value may contain other values or pairs not in <value>.

If <function> is "includes" or "excludes", then the required type of <value> is slightly relaxed, in that it may have fewer than the minimum number of elements specified by the column's type. If <function> is "excludes", then the required type is additionally relaxed in that <value> may have more than the maximum number of elements specified by the column's type.

<function>
One of "<", "<=", "==", "!=", ">=", ">", "includes", or "excludes".

<mutation>
A 3-element JSON array of the form [<column>, <mutator>, <value>] that represents a change to a column value. Except as otherwise specified below, <value> must have the same type as <column>. The meaning depends on the type of <column>:

integer or real
<mutator> must be "+=", "-=", "*=", "/=", or (integer only) "%=". The value of <column> is changed to the sum, difference, product, quotient, or remainder, respectively, of <column> and <value>.

Constraints on <column> are ignored when parsing <value>.

boolean, string, or uuid
No valid <mutator>s are currently defined for these types.

set
Any <mutator> valid for the set's element type may be applied to the set, in which case the mutation is applied to each member of the set individually. <value> must be a scalar value of the same type as the set's element type, except that constraints are ignored when parsing <value>.

If <mutator> is "insert", then each of the values in the set in <value> is added to <column> if it is not already present. The required type of <value> is slightly relaxed, in that it may have fewer than the minimum number of elements specified by the column's type.

If <mutator> is "delete", then each of the values in the set in <value> is removed from <column> if it is present there. The required type is slightly relaxed in that <value> may have more or less than the maximum number of elements specified by the column's type.

map
<mutator> must be "insert" or "delete".

If <mutator> is "insert", then each of the key-value pairs in the map in <value> is added to <column> only if its key is not already present. The required type of <value> is slightly relaxed, in that it may have fewer than the minimum number of elements specified by the column's type.

If <mutator> is "delete", then <value> may have the same type as <column> (a map type), or it may be a set whose element type is the same as <column>'s key type:

+ If <value> is a map, the mutation deletes each key-value pair in <column> whose key and value equal one of the key-value pairs in <value>.

+ If <value> is a set, the mutation deletes each key-value pair in <column> whose key equals one of the values in <value>.

For "delete", <value> may have any number of elements, regardless of restrictions on the number of elements in <column>.

<mutator>
One of "+=", "-=", "*=", "/=", "%=", "insert", or "delete".

5.2. Operations

The operations that may be performed as part of a "transact" RPC request (see Section 4.1.3) are described in the following subsections. Each of these operations is a JSON object that may be included as one of the elements of the "params" array that is one of the elements of the "transact" request. The details of each object, its semantics, results, and possible errors are described below.

5.2.1. Insert

The "insert" object contains the following members:

```
"op": "insert"          required
"table": <table>        required
"row": <row>            required
"uuid-name": <id>       optional
```

The corresponding result object contains the following member:

```
"uuid": <uuid>
```

The operation inserts "row" into "table". If "row" does not specify values for all the columns in "table", those columns receive default values. The default value for a column depends on its type. The default for a column whose <type> specifies a "min" of 0 is an empty set or empty map. Otherwise, the default is a single value or a single key-value pair, whose value(s) depend on its <atomic-type>:

- "integer" or "real": 0
- "boolean": false
- "string": "" (the empty string)
- "uuid": 00000000-0000-0000-0000-000000000000

The new row receives a new, randomly generated UUID. If "uuid-name" is supplied, then it is an error if <id> is not unique among the "uuid-name"s supplied on all the "insert" operations within this transaction. The UUID for the new row is returned as the "uuid" member of the result.

The errors that may be returned are as follows:

"error": "duplicate uuid-name"
The same "uuid-name" appears on another "insert" operation within this transaction.

"error": "constraint violation"
One of the values in "row" does not satisfy the immediate constraints for its column's <base-type>. This error will occur for columns that are not explicitly set by "row" if the default value does not satisfy the column's constraints.

5.2.2. Select

The "select" object contains the following members:

```
"op": "select"                required
"table": <table>              required
"where": [<condition>*]       required
"columns": [<column>*]        optional
```

The corresponding result object contains the following member:

```
"rows": [<row>*]
```

The operation searches "table" for rows that match all the conditions specified in "where". If "where" is an empty array, every row in "table" is selected.

The "rows" member of the result is an array of objects. Each object corresponds to a matching row, with each column specified in "columns" as a member, the column's name as the member name, and its value as the member value. If "columns" is not specified, all the table's columns are included (including the internally generated "_uuid" and "_version" columns). If two rows of the result have the same values for all included columns, only one copy of that row is included in "rows". Specifying "_uuid" within "columns" will avoid dropping duplicates, since every row has a unique UUID.

The ordering of rows within "rows" is unspecified.

5.2.3. Update

The "update" object contains the following members:

```
"op": "update"                required
"table": <table>              required
"where": [<condition>*]       required
"row": <row>                  required
```

The corresponding result object contains the following member:

```
"count": <integer>
```

The operation updates rows in a table. It searches "table" for rows that match all the conditions specified in "where". For each matching row, it changes the value of each column specified in "row" to the value for that column specified in "row". The "_uuid" and "_version" columns of a table may not be directly updated with this operation. Columns designated read-only in the schema also may not be updated.

The "count" member of the result specifies the number of rows that matched.

The error that may be returned is:

"error": "constraint violation"
One of the values in "row" does not satisfy the immediate constraints for its column's <base-type>.

5.2.4. Mutate

The "mutate" object contains the following members:

```
"op":  "mutate"               required
"table": <table>              required
"where": [<condition>*]       required
"mutations": [<mutation>*]    required
```

The corresponding result object contains the following member:

```
"count": <integer>
```

The operation mutates rows in a table. It searches "table" for rows that match all the conditions specified in "where". For each matching row, it mutates its columns as specified by each <mutation> in "mutations", in the order specified.

The "_uuid" and "_version" columns of a table may not be directly modified with this operation. Columns designated read-only in the schema also may not be updated.

The "count" member of the result specifies the number of rows that matched.

The errors that may be returned are:

"error": "domain error"
The result of the mutation is not mathematically defined, e.g., division by zero.

"error": "range error"
The result of the mutation is not representable within the database's format, e.g., an integer result outside the range INT64_MIN...INT64_MAX or a real result outside the range -DBL_MAX...DBL_MAX.

"error": "constraint violation"
The mutation caused the column's value to violate a constraint, e.g., it caused a column to have more or fewer values than are allowed, an arithmetic operation caused a set or map to have duplicate elements, or it violated a constraint specified by a column's <base-type>.

5.2.5. Delete

The "delete" object contains the following members:

```
"op": "delete"              required
"table": <table>            required
"where": [<condition>*]     required
```

The corresponding result object contains the following member:

```
"count": <integer>
```

The operation deletes all the rows from "table" that match all the conditions specified in "where". The "count" member of the result specifies the number of deleted rows.

5.2.6. Wait

The "wait" object contains the following members:

"op": "wait"	required
"timeout": <integer>	optional
"table": <table>	required
"where": [<condition>*]	required
"columns": [<column>*]	required
"until": "==" or "!="	required
"rows": [<row>*]	required

There is no corresponding result object.

The operation waits until a condition becomes true.

If "until" is "==", it checks whether the query on "table" specified by "where" and "columns", which is evaluated in the same way as specified for "select", returns the result set specified by "rows". If it does, then the operation completes successfully. Otherwise, the entire transaction rolls back. It is automatically restarted later, after a change in the database makes it possible for the operation to succeed. The client will not receive a response until the operation permanently succeeds or fails.

If "until" is "!=", the sense of the test is negated. That is, as long as the query on "table" specified by "where" and "columns" returns "rows", the transaction will be rolled back and restarted later.

If "timeout" is specified, then the transaction aborts after the specified number of milliseconds. The transaction is guaranteed to be attempted at least once before it aborts. A "timeout" of 0 will abort the transaction on the first mismatch.

The error that may be returned is:

"error": "timed out"
: The "timeout" was reached before the transaction was able to complete.

5.2.7. Commit

The "commit" object contains the following members:

"op": "commit"	required
"durable": <boolean>	required

There is no corresponding result object.

If "durable" is specified as true, then the transaction, if it commits, will be stored durably (to disk) before the reply is sent to the client. This operation with "durable" set to false is effectively a no-op.

The error that may be returned is:

"error": "not supported"
: When "durable" is true, this database implementation does not support durable commits.

5.2.8. Abort

The "abort" object contains the following member:

"op": "abort"	required

There is no corresponding result object (the operation never succeeds).

The operation aborts the entire transaction with an error. This may be useful for testing.

The error that will be returned is:

"error": "aborted"
: This operation always fails with this error.

5.2.9. Comment

The "comment" object contains the following members:

"op": "comment"	required
"comment": <string>	required

There is no corresponding result object.

The operation provides information to a database administrator on the purpose of a transaction. The ovsdb-server implementation, for example, adds comments in transactions that modify the database to the database journal. This can be helpful in debugging, e.g., when there are multiple clients writing to a database. An example of this can be seen in the ovs-vsctl tool, a command line tool that interacts with ovsdb-server. When performing operations on the database, it includes the command that was invoked (e.g., "ovs-vsctl add-br br0") as a comment in the transaction, which can then be seen in the journal alongside the changes that were made to the tables in the database.

5.2.10. Assert

The assert object contains the following members:

"op": "assert"	required
"lock": <id>	required

Result object has no members.

The assert operation causes the transaction to be aborted if the client does not own the lock named <id>.

The error that may be returned is:

"error": "not owner"
: The client does not own the named lock.

6. IANA Considerations

IANA has assigned TCP port 6640 for this protocol. Earlier implementations of OVSDB used another port number, but compliant implementations should use the IANA-assigned number.

IANA has updated the reference for port 6640 to point to this document.

7. Security Considerations

The main security issue that needs to be addressed for the OVSDB protocol is the authentication, integrity, and privacy of communications between a client and server implementing this protocol. To provide such protection, an OVSDB connection SHOULD be secured using Transport Layer Security (TLS) [RFC5246]. The precise details of how clients and servers authenticate each other is highly dependent on the operating environment. It is often the case that

OVSDB clients and servers operate in a tightly controlled environment, e.g., on machines in a single data center where they communicate on an isolated management network.

8. Acknowledgements

Thanks to Jeremy Stribling and Justin Pettit for their helpful input to this document.

9. References

9.1. Normative References

[DCE] "DCE: Remote Procedure Call", Open Group CAE Specification C309, ISBN 1-85912-041-5, August 1994.

[JSON-RPC] "JSON-RPC Specification, Version 1.0", <http://json-rpc.org/wiki/specification>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

9.2. Informative References

[DB-SCHEMA] "Open vSwitch Database Schema", <http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf>.

[OF-SPEC] Open Networking Foundation, "OpenFlow Switch Specification, version 1.3.3", October 2013, <https://www.opennetworking.org>.

[OVS] "Open vSwitch", <http://openvswitch.org/>.

Authors' Addresses

Ben Pfaff
VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
USA

EMail: blp@nicira.com

Bruce Davie (editor)
VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
USA

EMail: bsd@nicira.com