HiveServer2 Thrift API

Introduction

This document is a proposal for a new HiveServer2 Thrift API.

Motivations

Concurrency

Many users have reported that the current HiveServer implementation has concurrency bugs (for example, see HIVE-80). In fact, it's impossible for HiveServer to support concurrent connections using the current Thrift API, a result of the fact that Thrift doesn't provide server-side access to connection handles. Since the current API does not provide explicit support for sessions or connections, HiveServer has no way of mapping incoming requests to client sessions, which makes it impossible for HiveServer to maintain session state in between calls.

HiveServer currently attempts to maintain session state using thread-local variables and relies on Thrift to consistently map the same connection to the same Thrift worker thread, but this isn't a valid assumption to make. For example, if a client executes "set mapred.reduce.tasks=1" followed by "select .....", it is incorrect to assume that both of these statements will be executed by the same worker thread. Furthermore, the Thrift API doesn't provide any mechanism for detecting client disconnects (see THRIFT-1195), which results in incorrect behavior like this:

```
% hive -h localhost -p 10000
[localhost:10000] hive> set x=1;
set x=1;
[localhost:10000] hive> set x;
set x;
x=1
[localhost:10000] hive> quit;
quit;
% hive -h localhost -p 10000
[localhost:10000] hive> set x;
set x;
x=1
[localhost:10000] hive> quit;
quit;
```

In this example the user opened a connection to a HiveServer process, modified the connection's session state by setting x=1, and then closed the connection. The same user then created a new connection and printed the value of x. Since this statement was executed in a new connection/session we expect x to be undefined. However, we see that x actually has the value that was set in the previous session. This incorrect behavior occurs because Thrift has assigned the same worker thread to service the second connection, and since Thrift doesn't provide a mechanism for detecting client disconnects, HiveServer was unable to clear the thread-local session state associated with that worker thread before Thrift reassigned it to the second connection.

While it's tempting to try to solve these problems by modifying Thrift to provide direct access to the connection handle (which would allow HiveServer to map client connection handles to session state), this approach makes it really hard to support HA since it depends on the physical connection lasting as long as the user session, which isn't a fair assumption to make in the context of queries that can take many hours to complete.

Instead, the approach we're taking with HiveServer2 is to provide explicit support for sessions in the client API, e.g every RPC call references a session ID which the server then maps to persistent session state. This makes it possible for any worker thread to service any request from any client connection, and also the avoids the need to tightly couple physical connections to logical sessions.

Improved Support for ODBC/JDBC

It has been a struggle to implement ODBC and JDBC drivers on top of the HiveServer Thrift API. This is largely a result of missing support for sessions, asynchronous query execution, the ability to cancel running queries, and methods for retrieving information about the capabilities of the remote server. We have attempted to address all of these issues with the HiveServer2 API.

Proposed HiveServer2 Thrift API

```
// Licensed to the Apache Software Foundation (ASF) under one
// or more contributor license agreements.  See the NOTICE file
// distributed with this work for additional information
// regarding copyright ownership.  The ASF licenses this file
// to you under the Apache License, Version 2.0 (the
// "License"); you may not use this file except in compliance
// with the License.  You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
```

```
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// required for GetQueryPlan()
include "ql/if/queryplan.thrift"

namespace java org.apache.hive.service
namespace cpp Apache.Hive

// List of protocol versions. A new token should be
// added to the end of this list every time a change is made.
enum ProtocolVersion {
  HIVE_SERVER2_PROTOCOL_V1
}

enum TType {
  BOOLEAN_TYPE,
  TINYINT_TYPE,
  SMALLINT_TYPE,
  INT_TYPE,
  BIGINT_TYPE,
  FLOAT_TYPE,
  DOUBLE_TYPE,
  STRING_TYPE,
  TIMESTAMP_TYPE,
  BINARY_TYPE,
  ARRAY_TYPE,
  MAP_TYPE,
  STRUCT_TYPE,
  UNION_TYPE,
  USER_DEFINED_TYPE
}

const set<TType> PRIMITIVE_TYPES = [
  TType.BOOLEAN_TYPE
  TType.TINYINT_TYPE
  TType.SMALLINT_TYPE
  TType.INT_TYPE
  TType.BIGINT_TYPE
  TType.FLOAT_TYPE
  TType.DOUBLE_TYPE
  TType.STRING_TYPE
  TType.TIMESTAMP_TYPE
  TType.BINARY_TYPE
]

const set<TType> COMPLEX_TYPES = [
  TType.ARRAY_TYPE
  TType.MAP_TYPE
  TType.STRUCT_TYPE
  TType.UNION_TYPE
  TType.USER_DEFINED_TYPE
]

const set<TType> COLLECTION_TYPES = [
  TType.ARRAY_TYPE
  TType.MAP_TYPE
]

const map<TType,string> TYPE_NAMES = {
  TType.BOOLEAN_TYPE: "BOOLEAN",
  TType.TINYINT_TYPE: "TINYINT",
  TType.SMALLINT_TYPE: "SMALLINT",
  TType.INT_TYPE: "INT",
  TType.BIGINT_TYPE: "BIGINT",
  TType.FLOAT_TYPE: "FLOAT",
  TType.DOUBLE_TYPE: "DOUBLE",
  TType.STRING_TYPE: "STRING",
  TType.TIMESTAMP_TYPE: "TIMESTAMP",
  TType.BINARY_TYPE: "BINARY",
```

```
  TType.ARRAY_TYPE: "ARRAY",
  TType.MAP_TYPE: "MAP",
  TType.STRUCT_TYPE: "STRUCT",
  TType.UNION_TYPE: "UNIONTYPE"
}

// Thrift does not support recursively defined types or forward declarations,
// which makes it difficult to represent Hive's nested types.
// To get around these limitations TTypeDesc employs a type list that maps
// integer "pointers" to TTypeEntry objects. The following examples show
// how different types are represented using this scheme:
//
// "INT":
// TTypeDesc {
//   types = [
//     TTypeEntry.primitive_entry {
//       type = INT_TYPE
//     }
//   ]
// }
//
// "ARRAY<INT>":
// TTypeDesc {
//   types = [
//     TTypeEntry.array_entry {
//       object_type_ptr = 1
//     },
//     TTypeEntry.primitive_entry {
//       type = INT_TYPE
//     }
//   ]
// }
//
// "MAP<INT,STRING>":
// TTypeDesc {
//   types = [
//     TTypeEntry.map_entry {
//       key_type_ptr = 1
//       value_type_ptr = 2
//     },
//     TTypeEntry.primitive_entry {
//       type = INT_TYPE
//     },
//     TTypeEntry.primitive_entry {
//       type = STRING_TYPE
//     }
//   ]
// }

typedef i32 TTypeEntryPtr

// Type entry for a primitive type.
struct TPrimitiveTypeEntry {
  // The primitive type token. This must satisfy the condition
  // that type is in the PRIMITIVE_TYPES set.
  1: required TType type
}

// Type entry for an ARRAY type.
struct TArrayTypeEntry {
  1: required TTypeEntryPtr object_type_ptr
}

// Type entry for a MAP type.
struct TMapTypeEntry {
  1: required TTypeEntryPtr key_type_ptr
  2: required TTypeEntryPtr value_type_ptr
}

// Type entry for a STRUCT type.
struct TStructTypeEntry {
```

```
  1: required map<string, TTypeEntryPtr> name_to_type_ptr
}

// Type entry for a UNIONTYPE type.
struct TUnionTypeEntry {
  1: required map<string, TTypeEntryPtr> name_to_type_ptr
}

struct TUserDefinedTypeEntry {
  // The fully qualified name of the class implementing this type.
  1: required string typeClassName
}

// We use a union here since Thrift does not support inheritance.
union TTypeEntry {
  1: TPrimitiveTypeEntry primitive_entry
  2: TArrayTypeEntry array_entry
  3: TMapTypeEntry map_entry
  4: TStructTypeEntry struct_entry
  5: TUnionTypeEntry union_entry
  6: TUserDefinedTypeEntry user_defined_type_entry
}

// Type descriptor for columns.
struct TTypeDesc {
  // The "top" type is always the first element of the list.
  // If the top type is an ARRAY, MAP, STRUCT, or UNIONTYPE
  // type, then subsequent elements represent nested types.
  1: required list<TTypeEntry> types
}

// A result set column descriptor.
struct TColumnDesc {
  // The name of the column
  1: required string col_name

  // The type descriptor for this column
  2: required TTypeDesc type_desc

  // The ordinal position of this column in the schema
  3: required i32 position

  4: optional string comment
}

// Metadata used to describe the schema (column names, types, comments)
// of result sets.
struct TTableSchema {
  1: required list<TColumnDesc> columns
}

// A single column value in a result set.
// Note that Hive's type system is richer than Thrift's,
// so in some cases we have to map multiple Hive types
// to the same Thrift type. On the client-side this is
// disambiguated by looking at the Schema of the
// result set.
union TColumnValue {
  1: bool   bool_val      // BOOLEAN
  2: byte   byte_val      // TINYINT
  3: i16    i16_val       // SMALLINT
  4: i32    i32_val       // INT
  5: i64    i64_val       // BIGINT, TIMESTAMP
  6: double double_val    // FLOAT, DOUBLE
  7: string string_val    // STRING, LIST, MAP, STRUCT, UNIONTYPE, BINARY
}

// Represents a row in a rowset.
struct TRow {
  1: list<TColumnValue> colVals
}
```

```
// Represents a rowset
struct TRowSet {
  // The starting row offset of this rowset.
  1: i64 start_row_offset
  2: list<TRow> rows
}

// The return status code contained in each response.
enum TStatusCode {
  SUCCESS,
  SUCCESS_WITH_INFO,
  SQL_STILL_EXECUTING,
  ERROR,
  INVALID_HANDLE
}

// The return status of a remote request
struct TStatus {
  1: required TStatusCode status_code

  // If status is SUCCESS_WITH_INFO, info_msgs may be populated with
  // additional diagnostic information.
  2: optional list<string> info_msgs

  // If status is ERROR, then the following fields may be set
  3: optional string sql_state  // as defined in the ISO/IEF CLI specification
  4: optional i32 error_code    // internal error code
  5: optional string error_message
}

// The state of an operation (i.e. a query or other
// asynchronous operation that generates a result set)
// on the server.
enum TOperationState {
  // The operation is running. In this state the result
  // set is not available.
  RUNNING,

  // The operation has completed. When an operation is in
  // this state it's result set may be fetched.
  FINISHED,

  // The operation was canceled by a client
  CANCELED,

  // The operation failed due to an error
  ERROR
}


// A string identifier. This is interpreted literally.
typedef string TIdentifier

// A search pattern.
//
// Valid search pattern characters:
// '_': Any single character.
// '%': Any sequence of zero or more characters.
// '\': Escape character used to include special characters,
//      e.g. '_', '%', '\'. If a '\' precedes a non-special
//      character it has no special meaning and is interpreted
//      literally.
typedef string TPattern


// A search pattern or identifier. Used as input
// parameter for many of the catalog functions.
union TPatternOrIdentifier {
  1: TIdentifier identifier
  2: TPattern pattern
```

```
}

struct THandleIdentifier {
  // 16 byte globally unique identifier
  // This is the public ID of the handle and
  // can be used for reporting.
  1: binary guid,

  // 16 byte secret generated by the server
  // and used to verify that the handle is not
  // being hijacked by another user.
  2: binary secret,
}

// Client-side handle to persistent
// session information on the server-side.
struct TSessionHandle {
  1: required THandleIdentifier sess_id
}

// The subtype of an OperationHandle.
enum TOperationType {
  EXECUTE_STATEMENT,
  GET_TYPE_INFO,
  GET_TABLES,
  GET_COLUMNS,
  GET_FUNCTIONS,
}

// Client-side reference to a task running
// asynchronously on the server.
struct TOperationHandle {
  1: required THandleIdentifier op_id
  2: required TOperationType op_type
}


// OpenSession()
//
// Open a session (connection) on the server against
// which operations may be executed.
struct TOpenSessionReq {
  // The version of the HiveServer2 protocol that the client is using.
  1: required ProtocolVersion client_protocol = ProtocolVersion.HIVE_SERVER2_PROTOCOL_V1

  // Username and password for authentication.
  // Depending on the authentication scheme being used,
  // this information may instead be provided by a lower
  // protocol layer, in which case these fields may be
  // left unset.
  2: optional string username
  3: optional string password

  // Configuration overlay which is applied when the session is
  // first created.
  4: optional map<string, string> configuration
}

struct TOpenSessionResp {
  1: required TStatus status

  // The protocol version that the server is using.
  2: ProtocolVersion server_protocol = ProtocolVersion.HIVE_SERVER2_PROTOCOL_V1

  // Session Handle
  3: TSessionHandle session_handle

  // The configuration settings for this session.
  4: map<string, string> configuration
}
```

```
// CloseSession()
//
// Closes the specified session and frees any resources
// currently allocated to that session. Any open
// operations in that session will be canceled.
struct TCloseSessionReq {
  1: required TSessionHandle session_handle
}

struct TCloseSessionResp {
  1: required TStatus status
}


// GetInfo()
//
// This function is based on ODBC's SQLGetInfo() function.
// The function returns general information about the data source
// using the same keys as ODBC.
struct TGetInfoReq {
  // The sesssion to run this request against
  1: required TSessionHandle session_handle

  // List of keys for which info is requested. If unset or empty,
  // the response message will contain the values of all Info
  // properties.
  2: optional list<string> info_keys
}

struct TGetInfoResp {
  1: required TStatus status

  // key/value pairs representing Info values
  2: map<string, string> info
}


// ExecuteStatement()
//
// Execute a statement.
// The returned OperationHandle can be used to check on the
// status of the statement, and to fetch results once the
// statement has finished executing.
struct TExecuteStatementReq {
  // The session to exexcute the statement against
  1: required TSessionHandle session

  // The statement to be executed (DML, DDL, SET, etc)
  2: required string statement

  // Configuration properties that are overlayed on top of the
  // the existing session configuration before this statement
  // is executed. These properties apply to this statement
  // only and will not affect the subsequent state of the Session.
  3: map<string, string> conf_overlay
}

struct TExecuteStatementResp {
  1: required TStatus status
  2: TOperationHandle op_handle
}


// GetTypeInfo()
//
// Get information about types supported by the HiveServer instance.
// The information is returned as a result set which can be fetched
// using the OperationHandle provided in the response.
//
// Refer to the documentation for ODBC's SQLGetTypeInfo function for
```

```
// for the format of the result set.
struct TGetTypeInfoReq {
  // The session to run this request against.
  1: required TSessionHandle session
}

struct TGetTypeInfoResp {
  1: required TStatus status
  2: TOperationHandle op_handle
}


// GetTables()
//
// Returns a list of tables with catalog, schema, and table
// type information. The information is returned as a result
// set which can be fetched using the OperationHandle
// provided in the response.
//
// Result Set Columns:
//
// col1
// name: TABLE_CAT
// type: STRING
// desc: Catalog name. NULL if not applicable.
//
// col2
// name: TABLE_SCHEM
// type: STRING
// desc: Schema name.
//
// col3
// name: TABLE_NAME
// type: STRING
// desc: Table name.
//
// col4
// name: TABLE_TYPE
// type: STRING
// desc: The table type, e.g. "TABLE", "VIEW", etc.
//
// col5
// name: REMARKS
// type: STRING
// desc: Comments about the table
//
struct TGetTablesReq {
  // Session to run this request against
  1: required TSessionHandle session

  // Name of the catalog or a search pattern.
  2: optional TPatternOrIdentifier catalog_name

  // Name of the schema or a search pattern.
  3: required TPatternOrIdentifier schema_name

  // Name of the table or a search pattern.
  4: required TPatternOrIdentifier table_name

  // List of table types to match
  // e.g. "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY",
  // "LOCAL TEMPORARY", "ALIAS", "SYNONYM", etc.
  5: list<string> table_types
}

struct TGetTablesResp {
  1: required TStatus status
  2: TOperationHandle op_handle
}
```

```
// GetColumns()
//
// Returns a list of columns in the specified tables.
// The information is returned as a result set which can be fetched
// using the OperationHandle provided in the response.
//
// Result Set Columns are the same as those for the ODBC SQLColumns
// function.
//
struct TGetColumnsReq {
  // Session to run this request against
  1: required TSessionHandle session

  // Name of the catalog. Must not contain a search pattern.
  2: optional TIdentifier catalog_name

  // Schema name or search pattern
  3: required TPatternOrIdentifier schema_name

  // Table name or search pattern
  4: required TPatternOrIdentifier table_name

  // Column name or search pattern
  5: required TPatternOrIdentifier column_name
}

struct TGetColumnsResp {
  1: required TStatus status
  2: optional TOperationHandle op_handle
}


// GetFunctions()
//
// Returns a list of functions supported by the data source.
//
// Result Set Columns:
//
// col1
// name: FUNCTION_NAME
// type: STRING
// desc: The name of the function.
//
// col2
// name: DESCRIPTION
// type: STRING
// desc: A description of the function.
//
// col3
// name: FUNCTION_TYPE
// type: STRING
// desc: The function type: "UDF", "UDAF", or "UDTF".
//
// col4
// name: FUNCTION_CLASS
// type: STRING
// desc: The fully qualified name of the class
//       that implements the specified function.
//
struct TGetFunctionsReq {
  // Session to run this request against
  1: required TSessionHandle session
}

struct TGetFunctionsResp {
  1: required TStatus status
  2: optional TOperationHandle op_handle
}


// GetOperationStatus()
```

```
//
// Get the status of an operation running on the server.
struct TGetOperationStatusReq {
  // Session to run this request against
  1: required TOperationHandle op_handle
}

struct TGetOperationStatusResp {
  1: required TStatus status
  2: TOperationState op_state
}


// CancelOperation()
//
// Cancels processing on the specified operation handle and
// frees any resources which were allocated.
struct TCancelOperationReq {
  // Operation to cancel
  1: required TOperationHandle op_handle
}

struct TCancelOperationResp {
  1: required TStatus status
}


// GetQueryPlan()
//
// Given an OperationHandle corresponding to an ExecuteStatement
// request, this function returns the queryplan for that
// statement annotated with counter information.
struct TGetQueryPlanReq {
  1: required TOperationHandle op_handle
}

struct TGetQueryPlanResp {
  1: required TStatus status
  2: optional queryplan.QueryPlan query_plan
}


// CloseOperation()
//
// Given an operation in the FINISHED, CANCELED,
// or ERROR states, CloseOperation() will free
// all of the resources which were allocated on
// the server to service the operation.
struct TCloseOperationReq {
  1: required TOperationHandle op_handle
}

struct TCloseOperationResp {
  1: required TStatus status
}


// GetResultSetMetadata()
//
// Retrieves schema information for the specified operation
struct TGetResultSetMetadataReq {
  // Operation for which to fetch result set schema information
  1: required TOperationHandle op_handle
}

struct TGetResultSetMetadataResp {
  1: required TStatus status
  2: optional TTableSchema schema
}
```

```
enum TFetchOrientation {
  // Get the next rowset. The fetch offset is ignored.
  FETCH_NEXT,

  // Get the previous rowset. The fetch offset is ignored.
  // NOT SUPPORTED
  FETCH_PRIOR,

  // Return the rowset at the given fetch offset relative
  // to the curren rowset.
  // NOT SUPPORTED
  FETCH_RELATIVE,

  // Return the rowset at the specified fetch offset.
  // NOT SUPPORTED
  FETCH_ABSOLUTE,

  // Get the first rowset in the result set.
  FETCH_FIRST,

  // Get the last rowset in the result set.
  // NOT SUPPORTED
  FETCH_LAST
}

// FetchResults()
//
// Fetch rows from the server corresponding to
// a particular OperationHandle.
struct TFetchResultsReq {
  // Operation from which to fetch results.
  1: required TOperationHandle op_handle

  // The fetch orientation. For V1 this must be either
  // FETCH_NEXT or FETCH_FIRST. Defaults to FETCH_NEXT.
  2: TFetchOrientation orientation = TFetchOrientation.FETCH_NEXT

  // Max number of rows that should be returned in
  // the rowset.
  3: i64 max_rows
}

struct TFetchResultsResp {
  1: required TStatus status

  // TRUE if there are more rows left to fetch from the server.
  2: bool has_more_rows

  // The rowset. This is optional so that we have the
  // option in the future of adding alternate formats for
  // representing result set data, e.g. delimited strings,
  // binary encoded, etc.
  3: optional TRowSet results
}

service TSQLService {

  TOpenSessionResp OpenSession(1:TOpenSessionReq req);

  TCloseSessionResp CloseSession(1:TCloseSessionReq req);

  TGetInfoResp GetInfo(1:TGetInfoReq req);

  TExecuteStatementResp ExecuteStatement(1:TExecuteStatementReq req);

  TGetTypeInfoResp GetTypeInfo(1:TGetTypeInfoReq req);

  TGetTablesResp GetTables(1:TGetTablesReq req);

  TGetColumnsResp GetColumns(1:TGetColumnsReq req);
```

```
  TGetFunctionsResp GetFunctions(1:TGetFunctionsReq req);

  TGetOperationStatusResp GetOperationStatus(1:TGetOperationStatusReq req);

  TCancelOperationResp CancelOperation(1:TCancelOperationReq req);

  TGetQueryPlanResp GetQueryPlan(1:TGetQueryPlanReq req);

  TGetResultSetMetadataResp GetResultSetMetadata(1:TGetResultSetMetadataReq req);

  TFetchResultsResp FetchResults(1:TFetchResultsReq req);
}
```