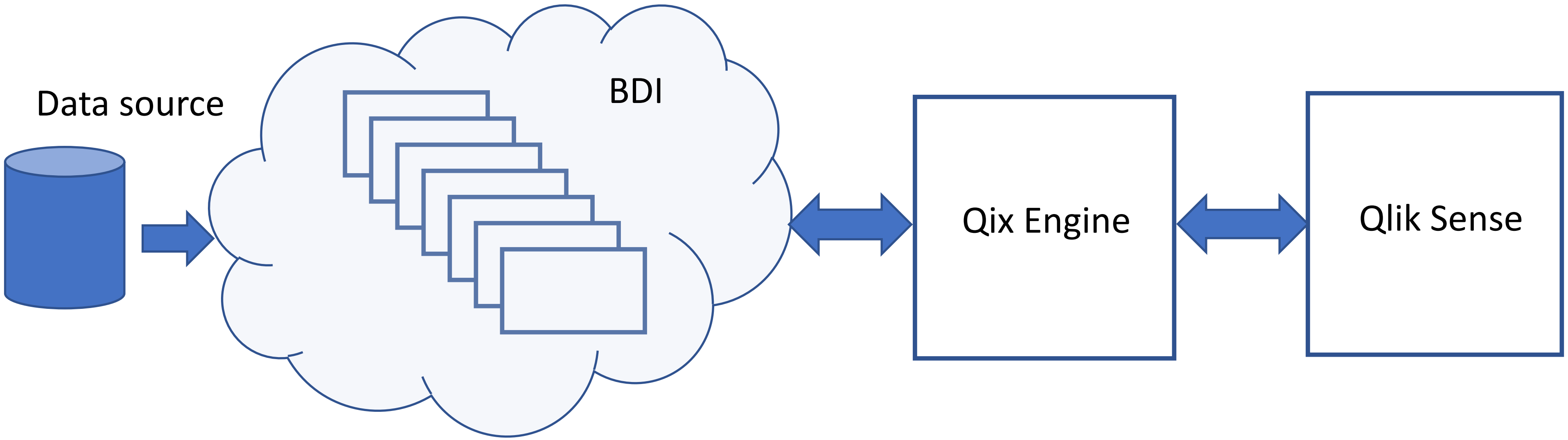
Qlik Associative Big Data Index (BDI)

D. Gueorguiev, December 2018

What is Qlik Associative Big Data Index?

Distributed column store running on AWS or Azure cloud
Connected to the main associative Qlik Engine (a.k.a *Qix Engine*) and serves as a distributed query processor which can process queries over datasets which massive sizes.
The main associative Qlik engine is connected to and serves as an in-memory column store to Qlik Sense which is the main Business Analytcis platform of Qlik Tech.



<u>My Contributions to Qlik Associative Big Data Index project</u>

1. Design and development of algorithm and code for the indexlet versioning, delivery and consumption by the QSL services
2. Design and development of the Index Maintenance Service
These topics are included in my presentation and are covered on slides 19-30.

Slide 20: *New indexlet version notification: the basic sequence of events.*
I have designed and implemented the sequence 1-13.
Slide 21: *Moving new version of appended indexlet: sequence of events inside QSL Manager.* I have designed and implemented the sequence 1-10.
Slide 22: *Moving new version of appended indexlet: sequence of events inside QSL Worker process.* I have designed and implemented the sequence 1-9.
Slides 23 - 25: *The types DatasetsWithIndexlets and ColumnIndex. Consuming the Appended Indexlets with class ReceivedIndexlets and VersionInfo*. I have designed and implemented the data structures for storing and moving the appended indexlets.
Note: I have *not* designed and implemented the structure *ColumnIndex* which is the standard BDI container for storing indexlets
Slides 29 – 35: I have co-designed and implemented the *Index Maintenance Service*.

Qlik Associative Big Data Index (BDI)

<u>Some Terminology</u>:

Datalet: part of the table with 2^24 (~16 million) rows

Global Symbol Table: a per-column distributed collection of unique symbols

Direct index from symbols to records: defined in database theory (abbrev. **s2r**)

Inverted index from records to symbols: defined in database theory (abbrev. **r2s**)

Indexlet: chunk of a column with 2^24 records which is supplied with direct index from symbols to records **s2r** and with inverted index from records to symbols **r2s**.

Inter-table associations: a.k.a. A2A index is built on top of two global symbol tables to enable associations between tables. A2A index is yet another table in which the associations are represented with pair of columns

Qlik Associative Big Data Index (BDI)

Indexation process in BDI:

1. Create the datalets
2. Create the indexlets in parallel
3. Create the Global Symbol tables
4. Create the local symbol maps to global symbols
5. Create the A2A Indexes

The Indexation process in BDI is performed by the BDI Indexing Services which are:

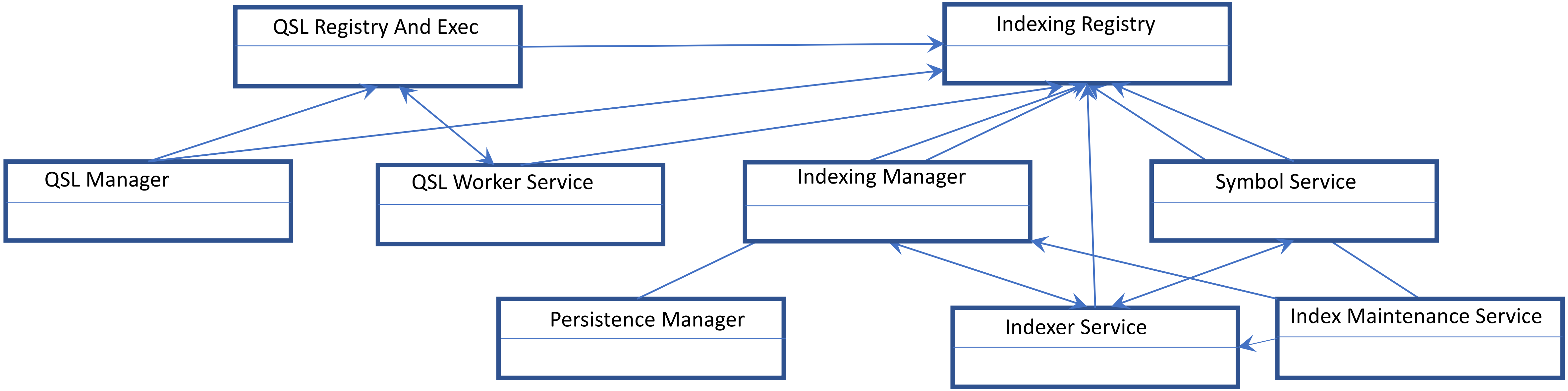
Indexing Registry Service (exactly 1)
Indexing Manager Service (exactly 1)
Symbol Service (1 or more)
Indexer Service (1 or more)
Index Maintenance Service (exactly 1)
Persistence Manager Service (exactly 1 ; > 1 in future impl)

Qlik Associative Big Data Index (BDI)

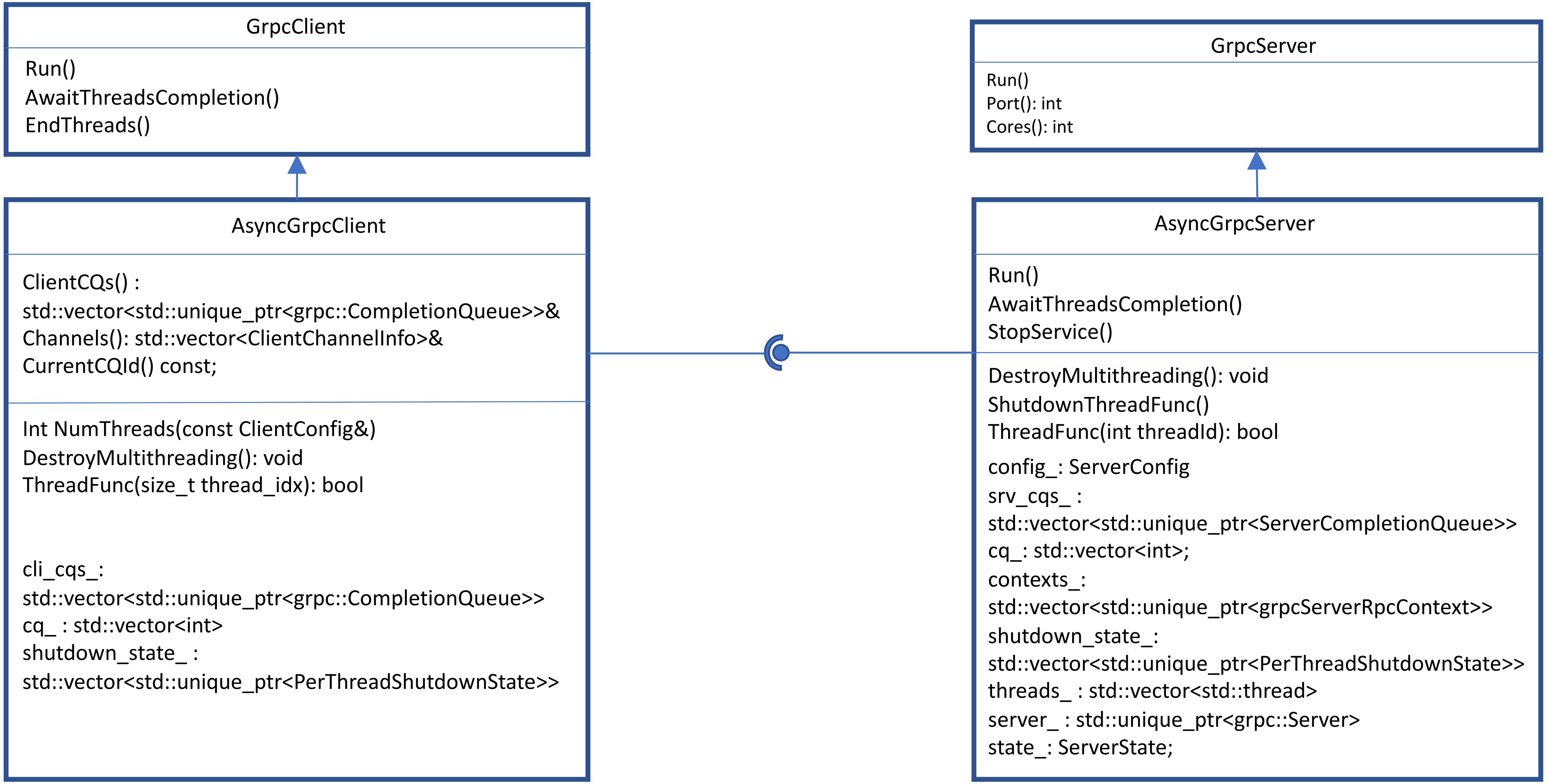
Query parsing, optimization, planning and execution is performed by the QSL Services in BDI which are:

1. QSL Registry And Executor Service (exactly 1)
2. QSL Manager Service (exactly 1 now, more instances will be supported in the future)
3. QSL Worker Service (1 or more)

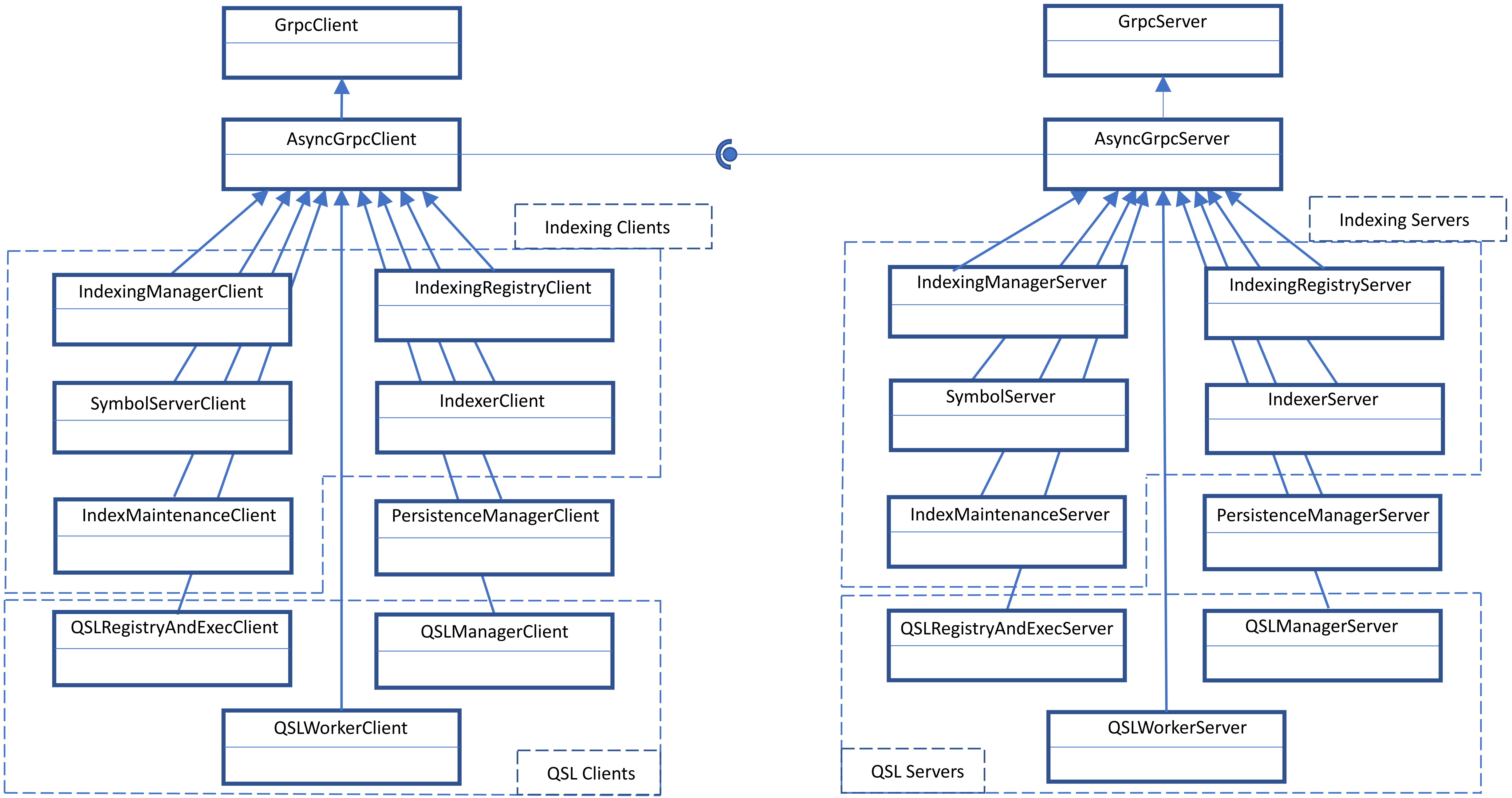
<u>Associations btw the BDI services</u>



The GRPC Async Client-Server design pattern



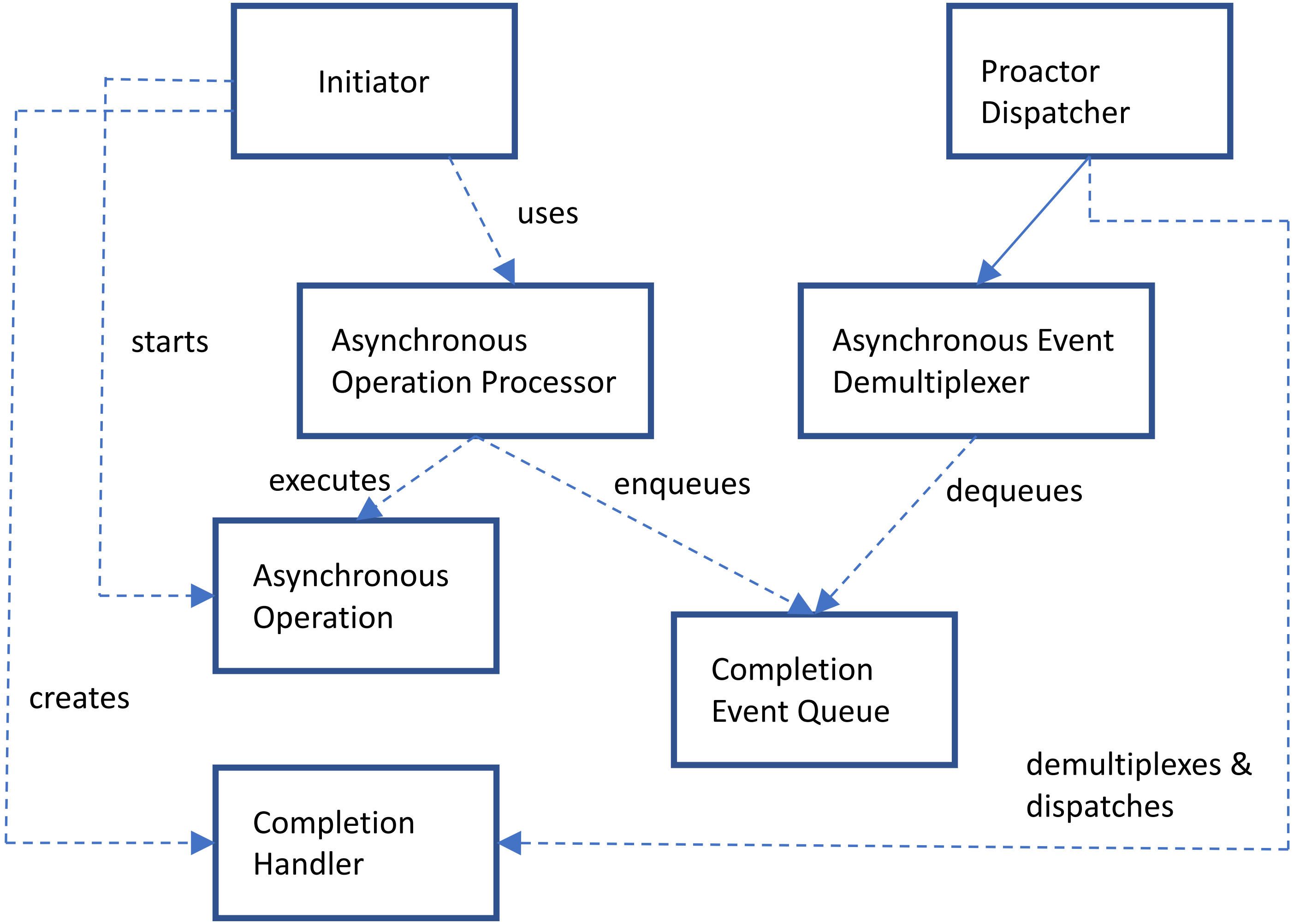
The GRPC Async Client-Server design pattern



Proactor Design Pattern

Characteristics: Efficient use of the asynchronous mechanisms provided by the OS in event driven applications.

Activity Diagram



Asynchronous Operation: defines an operation that is executed asynchronously. For instance read/write IO on a socket

Asynchronous Operation Processor: executes async operations and queues on a completion event queue when operations complete.

Completion Event Queue: Buffers completion events until they are dequeued by async event multiplexer

Completion Handler: Process the result of an asyncrhonous operation. This is a functor (function object)

Asynchronous Event Demultiplexer: Blocks waiting for events to occur on the completion event queue, and returns a completed event to its caller.

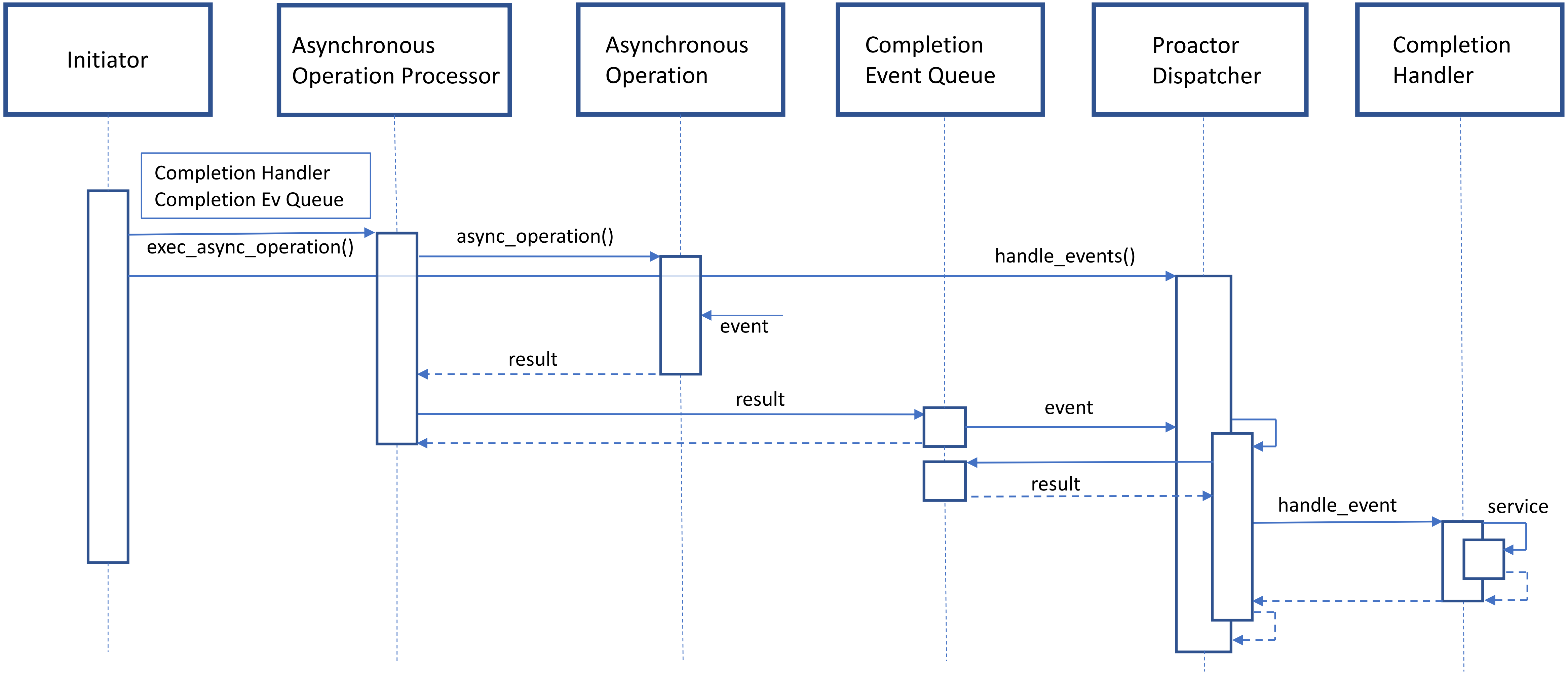
Proactor / Dispatcher: Calls async event demultiplexer to dequeue events and dispatches the completion handler by invoking the function object associated with the event.

Initiator: application-specific code that starts async IO. The Initiator interacts with the async operation processor

boost::asio Impl: https://www.boost.org/doc/libs/1_69_0/doc/html/boost_asio/overview/core/async.html

Proactor Design Pattern

Main Sequence Diagram



Boost Futures and Promises in async gRPC clients and servers

Excerpt from BDI Index Maintenance Service Client

```
// rpc methods
boost::shared_future<Void> Stop(const Void&);

boost::shared_future<MaintainIndexResponse> MaintainIndex(const SchemaSources& request);

boost::shared_future<MaintainedSchemas> GetMaintainedSchemas(const Void&);

boost::shared_future<MaintainIndexResponse> SetMaintenanceState(const SchemaMaintenanceState& request);

boost::shared_future<FileChangeResponse> FileChange(const FileChangeEvents& request);

boost::shared_future<TriggerUpdateTaskResponse> TriggerUpdateTask(const UpdateTaskInfo& request);

boost::shared_future<FileChangeResponse> SingleFileAddition(const SingleFileAddEvent& request);

boost::shared_future<TriggerUpdateTaskResponse> TriggerSingleUpdateTask(const SingleUpdateTaskInfo& request);
```

All Client side RPCs inside the BDI services are using boost::shared_future types in there return values

boost::future defines a value which will be computed/delivered in the future. The member function get() is blocking call and is used to obtain the value. boost::shared_future is the thread-safe version of boost::future. The counterpart of boost::future is boost::promise which is used to set the value which will be delivered asynchronously.

Boost Futures and Promises in async gRPC clients and servers

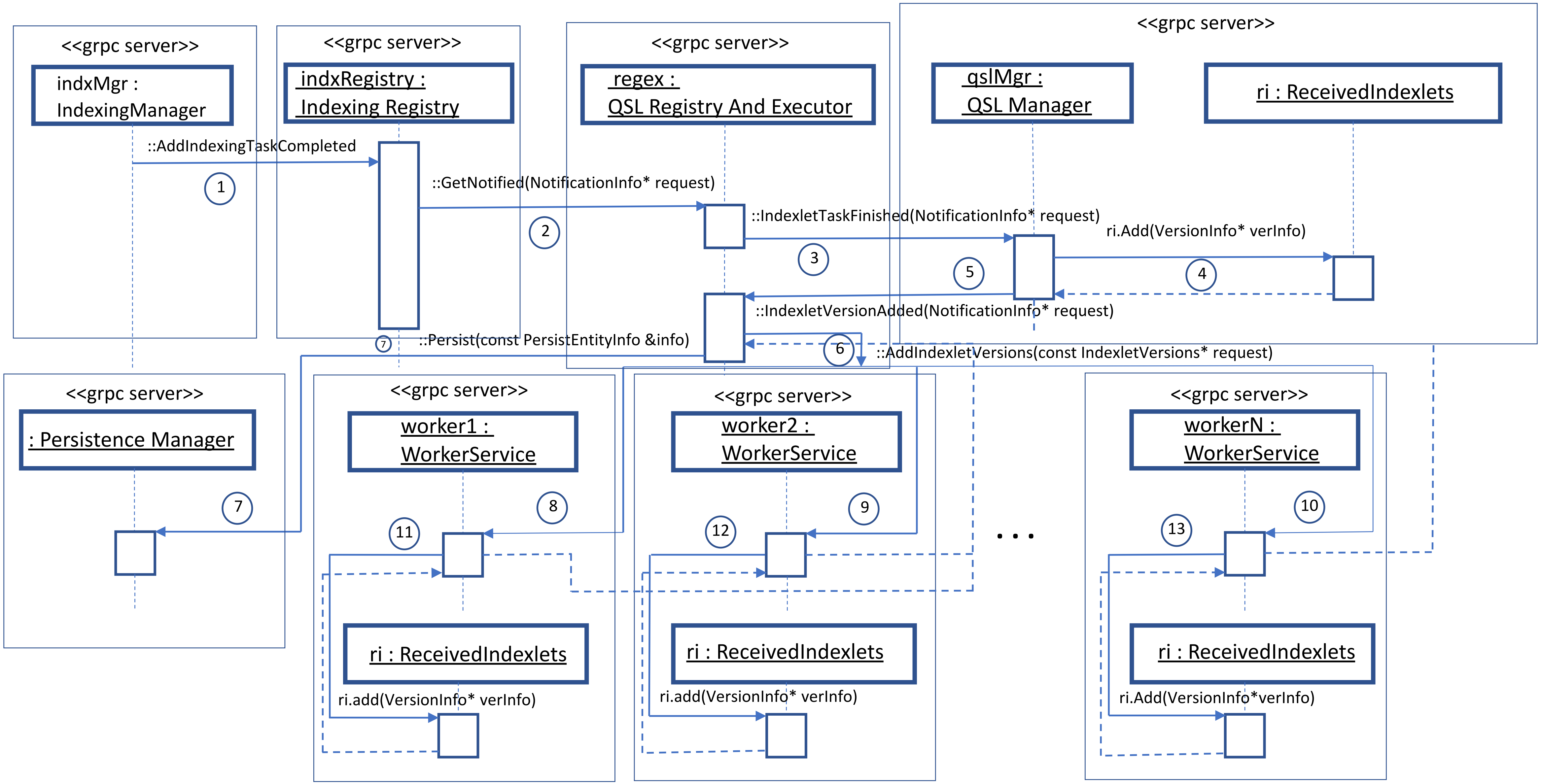
```
#define UNARY_RPC_CLIENT_FUNC(STUB_TYPE, CLIENT_SERVICE, FUNC_NAME,        \
                    REQUEST_TYPE, RESPONSE_TYPE)                            \
  boost::shared_future<RESPONSE_TYPE> CLIENT_SERVICE::FUNC_NAME(            \
    const REQUEST_TYPE &request) {                                          \
    namespace ph = std::placeholders;                                       \
    boost::promise<RESPONSE_TYPE> promiseResp;                              \
    boost::shared_future<RESPONSE_TYPE> fResponse(                          \
      promiseResp.get_future());                                            \
    auto request##FUNC_NAME =                                               \
      std::bind(&CLIENT_SERVICE::Request##FUNC_NAME, this, ph::_1,          \
           ph::_2, ph::_3, ph::_4);                                         \
    auto callback = [](boost::promise<RESPONSE_TYPE> &&prom,                \
              const RESPONSE_TYPE &resp) {                                  \
      prom.set_value(resp);                                                 \
    };                                                                      \
                                                                            \
    . . .                                                                   \
                                                                            \
  }                                                                         \
```

continues on the next page

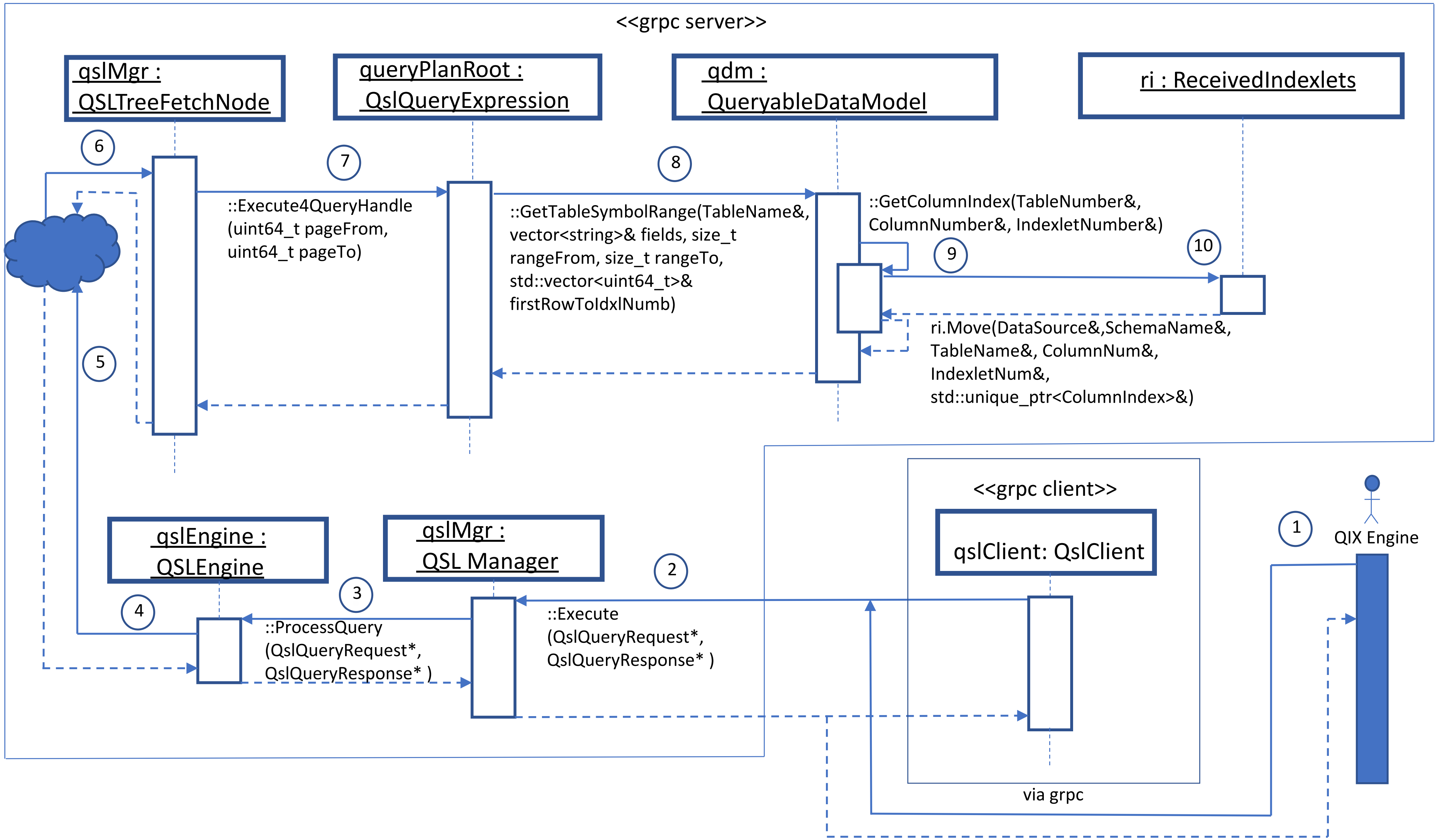
continues from the previous page

```
#define UNARY_RPC_CLIENT_FUNC(STUB_TYPE, CLIENT_SERVICE, FUNC_NAME,      \
                              REQUEST_TYPE, RESPONSE_TYPE)               \
  boost::shared_future<RESPONSE_TYPE> CLIENT_SERVICE::FUNC_NAME(         \
    const REQUEST_TYPE &request) {                                       \
                                                                         \
    . . .                                                                \
                                                                         \
    auto *cq = CurrentClientCQ();                                        \
    auto *ctx =                                                          \
      new ClientRpcContextUnaryImpl<STUB_TYPE, REQUEST_TYPE,             \
                    RESPONSE_TYPE, decltype(callback)>(                  \
        Channel(0).get_stub(), request, request##FUNC_NAME,              \
        std::move(promiseResp), callback);                               \
    ctx->Start(cq, config_);                                             \
    UpdateCurrentCQId();                                                 \
    return fResponse;                                                    \
  }                                                                      \

UNARY_RPC_CLIENT_FUNC(IndexMaintenanceService::Stub, IndexMaintenanceServiceClient,
MaintainIndex, SchemaSources, MaintainIndexResponse);
```

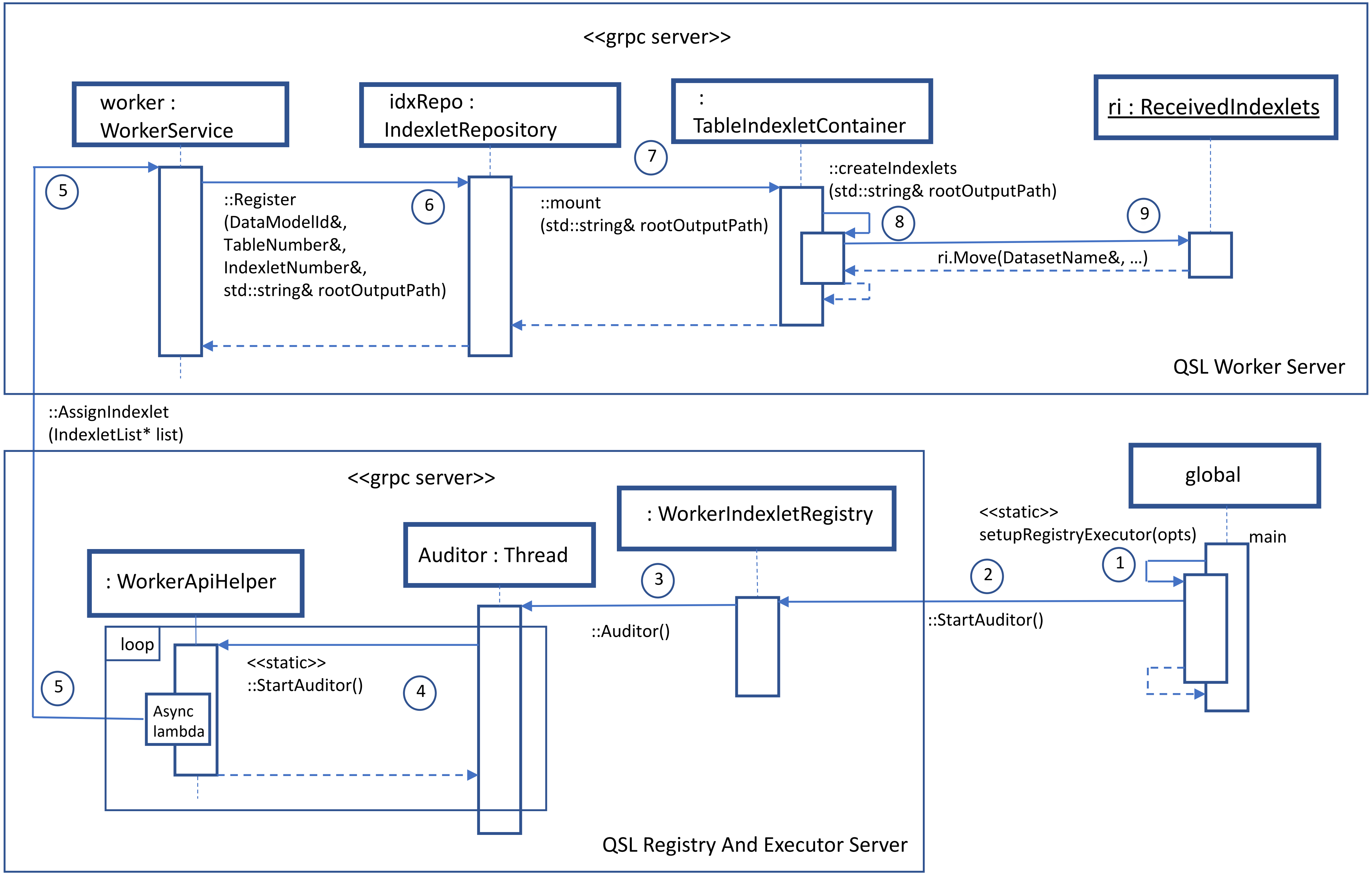
New indexlet version notification: the basic sequence of events



Moving new version of appended indexlet: sequence of events inside QSL Manager



Moving new version of appended indexlet: sequence of events inside QSL Worker process



The types DatasetsWithIndexlets and ColumnIndex

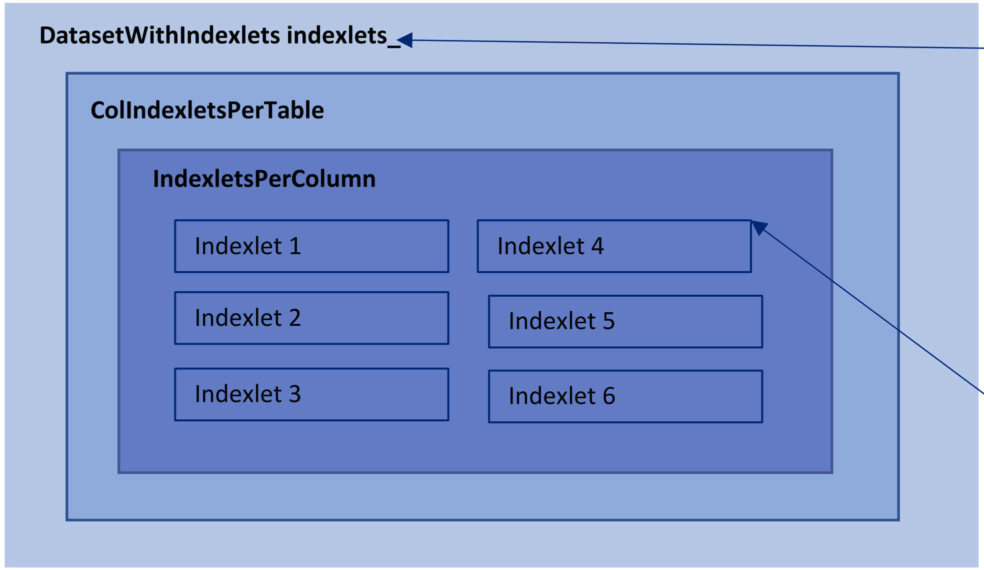
```
typedef ColumnIndex* ColIndexletPtr;
typedef std::vector<ColIndexletPtr> ColIndexletsVect;
typedef std::vector<ColIndexletsVect> IndexletsPerColumn;
typedef std::unordered_map<std::string,IndexletsPerColumn> ColIndexletsPerTable;
typedef std::unordered_map<std::string,ColIndexletsPerTable> DatasetsWithIndexlets;
```

ColumnIndex

```
ColumnIndex(int colNo)
ColumnIndex(ColumnIndex&&)
ColumnIndex& operator=(const ColumnIndex&)
```

```
r2s: std::shared_ptr<BitpackedSequence>
symbolMap: ImmutableRoaring64MapPtr
name: std::string
int colno;
```

The symbol map is stored in memory-mapped file. However, the bitpacked sequence representing the mapping between rows and symbols is all in cache.



Keys in DatasetWithIndexlets instance: datasetName, tableName, columnNumber, indexletNumber. Every indexlet instance stored there can be accessed by those keys

All indexlets stored in the DatasetsWithIndexlets container are instances of indexlet_new::ColumnIndex

Consuming the Appended Indexlets with class ReceivedIndexlets and VersionInfo

VersionInfo

indxletVersion_ : int64_t
rootPath_ : fs:path
table_ : std::string
storagePath_ : fs::path
colId_ : int
idxltId_ : int
In_place_ : bool
ToJson(Json::Value&): void
FromJson(const Json::Value&): void

Every new **VersionInfo** instance has its flag **in_place_** set to false hence <u>it will not be moved to the indexletStorage container immediatelly.</u> The flags **in_place_** of the newly arrived indexlets will be set to true only during quiet period in which no query is running.

ReceivedIndexlets

mu_ : std::timed_mutex_
versions_: std::unordered_map<std::string, std::shared_ptr<VersionInfo>>
indexlets_ : DatasetsWithIndexlets
Add(const fs::path& rootPath, const DatasetName& dsName,
const TableName& tblName, const int colId, const int idxltId,
const fs::path& appendSuffix): bool
Move(const DatasetName& dsName, const TableName& tblName,
const int colId, const int idxltId,
std::unique_ptr<ColumnIndex>& colIdx): bool
Move(const DatasetName& dsName, const TableName& tblName,
const int idxltId, const fs::path& root,
std::vector<std::unique_ptr<ColumnIndex>>& colIndices)
SetInPlaceFlag(): void

CallbackTimer

timerStarted_ : std::atomic<bool>
timerExpired_ : std::atomic<bool>
timerCanceled_ : std::atomic<bool>
intervSecs_ : std::atomic<int>
callback_: void(void)
Reset(int intervSecs = 60): bool
Start(int intervSecs): void
Cancel(): bool

The appended indexlet versions can be moved to the corresponding container in the QSL Manager only in "quiet times". Therefore a callback timer is needed which is started at the end of each query and is reset at the beginning of each query. If the timer expires before it has been reset it invokes a callback which sets to true the **in_place** flag for all received indexlets which have not been consumed.

Consuming the Appended Indexlets with class ReceivedIndexlets and VersionInfo

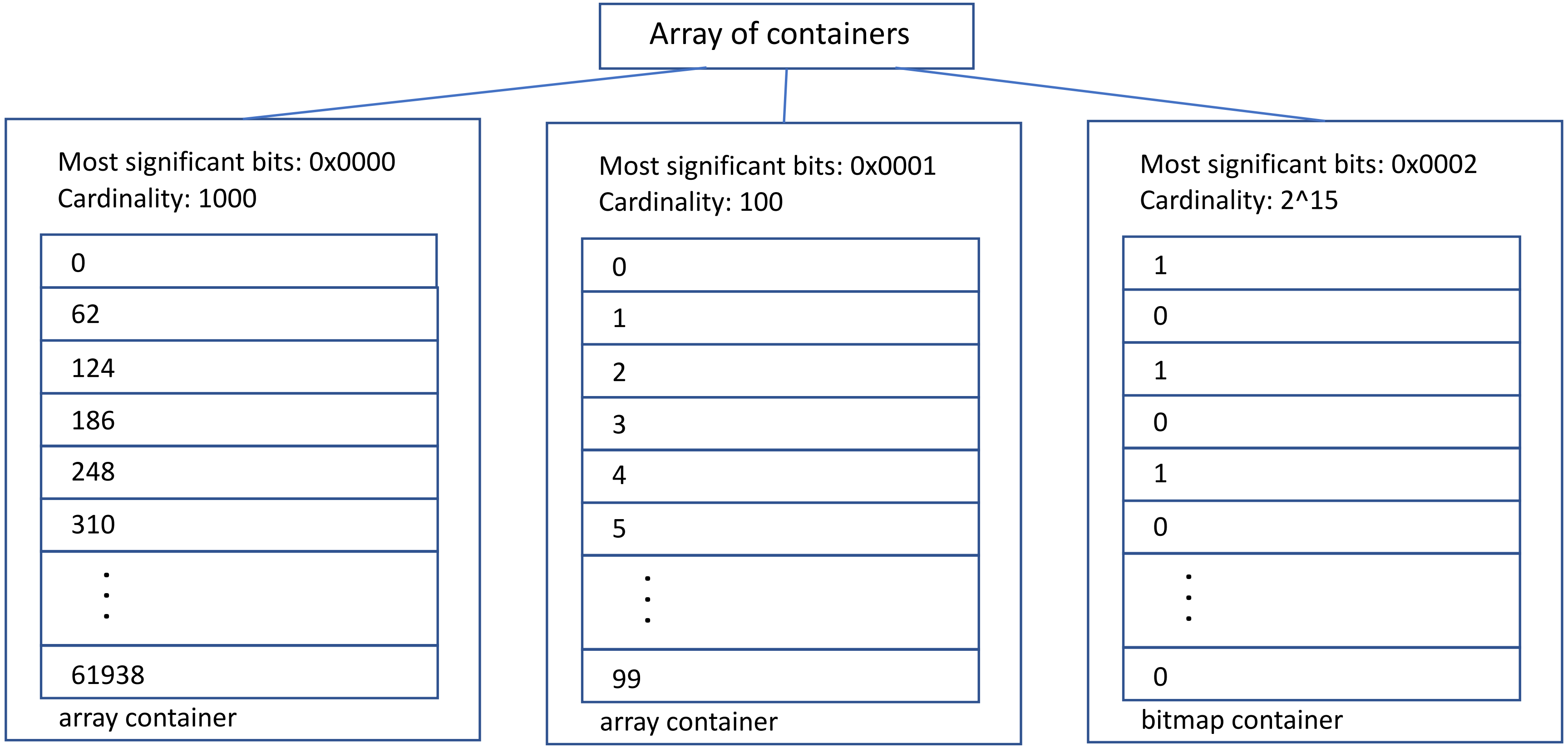
```
grpc::Status QueryExecutorManagerServer::Execute(const QslQueryRequest*  request, QslQueryResponse* response ) {
   QLT_ENTRY(7);

   QLD(5, "{0}", request->msg());

   timer.Cancel();

   std::string queryStr = request->msg();

   bool res = pQSLEngine_->ProcessQuery(request, response);

   timer.Start();

   return grpc::Status::OK;
} // Execute Query RPC
```

Cancel the timer in case it is not already expired

The appended indexlet versions can be moved to the corresponding container in the QSL Manager only in “quiet times”. Therefore a callback timer is needed which is started at the end of each query and is reset at the beginning of each query. If the timer expires before it has been reset it invokes a callback which sets to true the in_place flag for all received indexlets which have not been consumed.

What is Roaring compression and why is being used in BDI?

The Roaring compression is used to store and persist on the file system all symbols for the column indexlets and is popular compression format in DB systems. It is an preferred alternative to the Run-Length Encoding format due to its better performance on bitmap logical operations.
The Roaring algorithm compresses data into an array of containers where each container groups numbers by common higher 16 bits value and stores only the lower 16 bits of each 32 bit number.



Better bitmap performance with Roaring bitmaps, by S Chambi et al, 2014

<u>What I would have done differently?</u>

<u>Note on the container *ColumnIndex*:</u>
Each *ColumnIndex* instance contains *BitpackedSequence* and a pointer of type *ImmutableRoaring64MapPtr*. *ImmutableRoaring64MapPtr* is a pointer to *Roaring64Map* which is a memory-mapped file in *Roaring*-compressed format of all symbols for the current *ColumnIndexlet*.
The *Bitpacked* sequence stores the *row-to-symbol* map in cache while the symbols in the memory-mapped file have very small footprint in cache *initially* before the data has been accessed.

Instead of caching *ColumnIndex* I would have just stored the timestamp for each indexlet version. I thought that loading *ColumIndex* containers from the file system at the time of a running query should not happen because it would be taxing the performance in case we have thousands of newly appended indexlets to load. The current implementation does not cache the *ColumnIndex* instances in global structure but instead stores just the indexlet version and loads the *ColumnIndex* from the file system on demand i.e. in the middle of a running query.
Storing the indexlet timestamp instead of caching the *ColumnIndex* instance may have performance benefit in case the dataset is given with a large set of small files.

What I would have done differently?

I would have avoided using the *NotificationInfo* structs for RPC communication and intra-service communication within the QSL services. *NotificationInfo* structs are heavyweights because they use strings storing messages from the Indexing Services for new versions of appended indexlets. Those strings can get very large in case of datasets with very large number of files. For instance, one of BDI test datasets have more than 3000 files. The problem is that each *NotificationInfo* message contains also a full list of all files of the current dataset. In case of a dataset with more than 3,000 files the message string can become hundreds of Megabytes.

Another implementation would pass just a data structure with timestamps of the latest appended indexlets.

BDI Index Maintenance Service

The purpose of the Index Maintenance Service

To deliver incremental updates of the source dataset in consistent manner while the original dataset is being indexed.

Requirements:

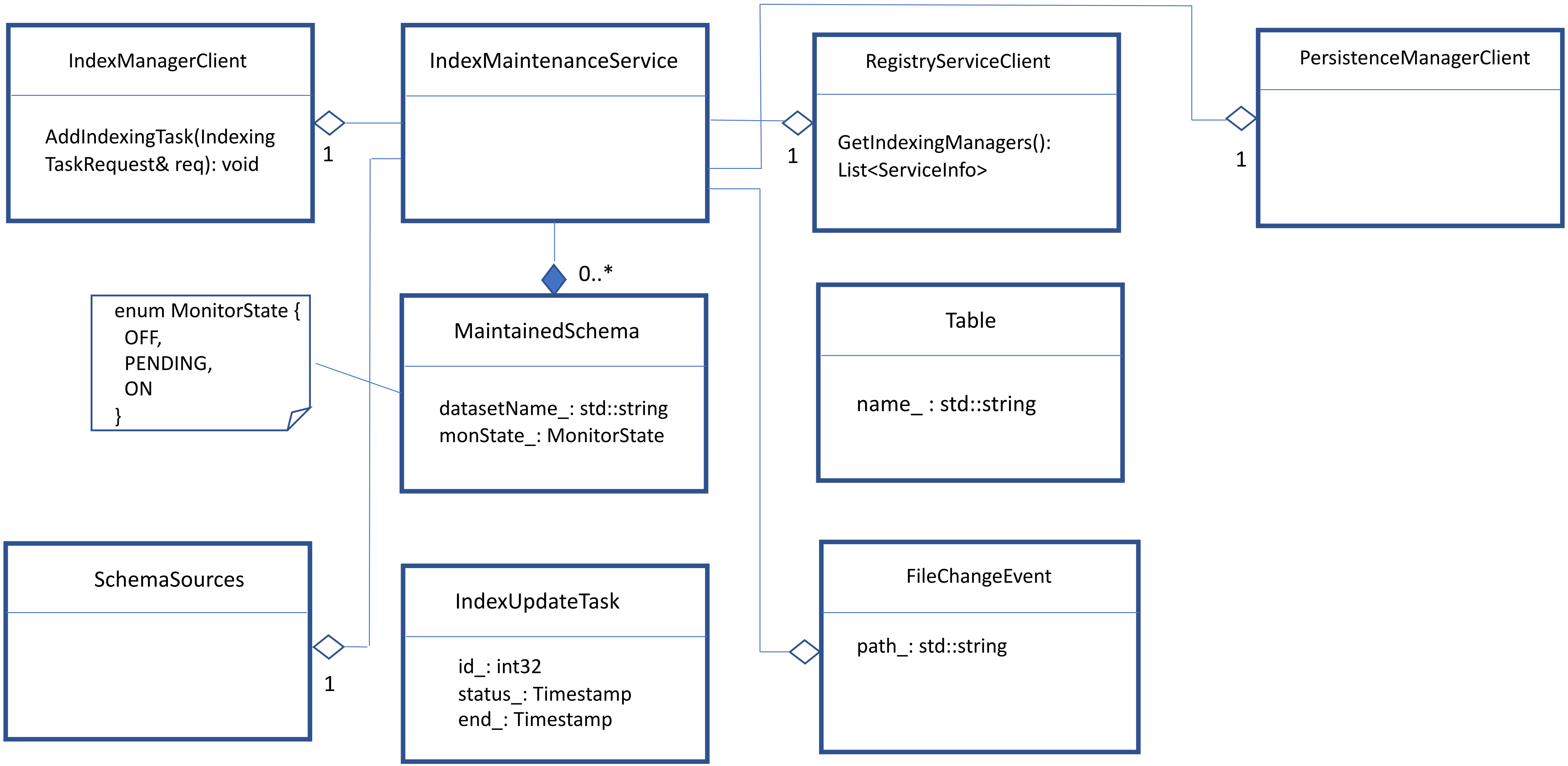
1. Initiate first stage indexing on user-specified data sources
2. Detect new files in source data systems
3. Flexibility on specifying which changes to monitor (e.g. which files, tables, etc.)
4. Low-latency notification of changes (within 10 seconds)
5. Aggregation of multiple changes into composite data updates according to user-specified time windows
6. Initiation of new indexing tasks to index new records
7. Monitoring of indexing tasks
8. Queuing of index tasks

BDI Index Maintenance Service

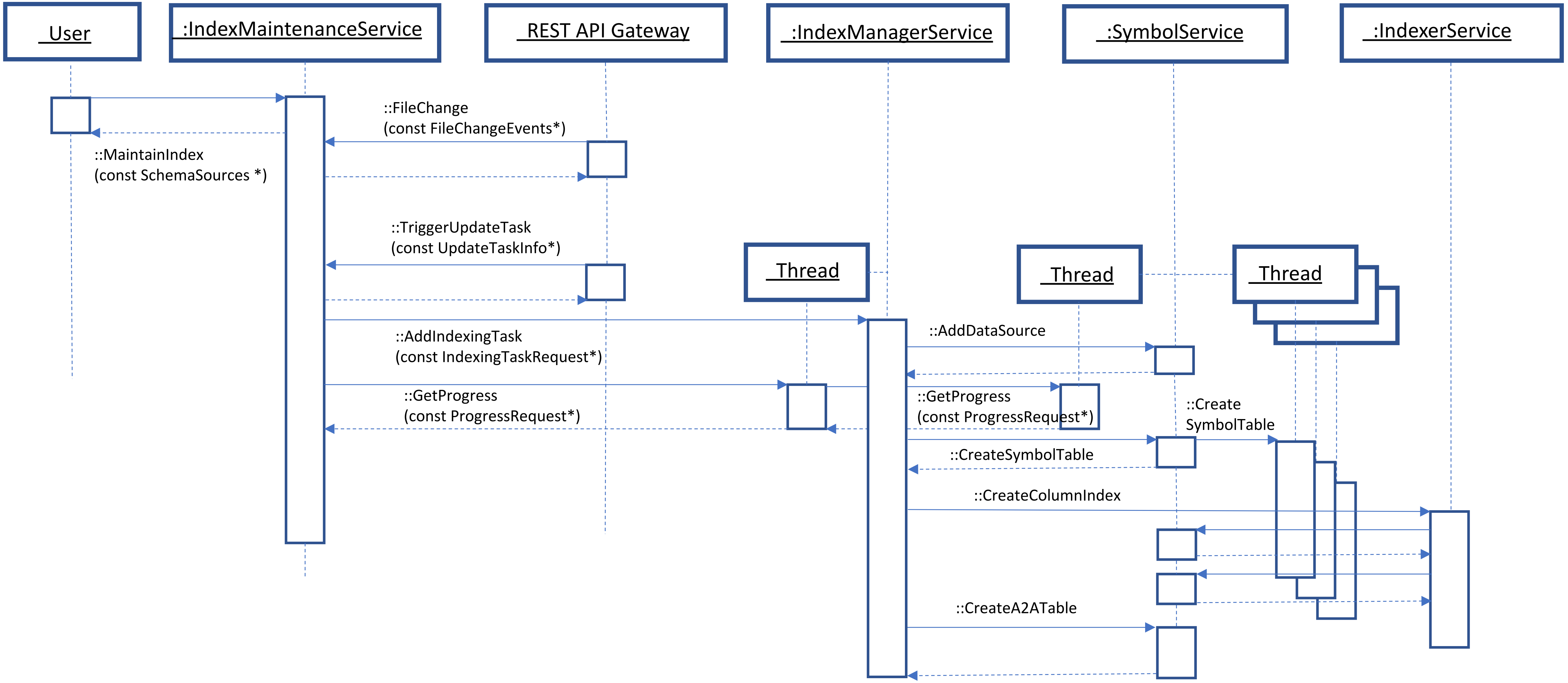
BDI Index Maintenance Service RPCs and REST API interface:

```
// rpc methods
boost::shared_future<Void> Stop(const Void&);

boost::shared_future<MaintainIndexResponse> MaintainIndex(const SchemaSources& request);

boost::shared_future<MaintainedSchemas> GetMaintainedSchemas(const Void&);

boost::shared_future<MaintainIndexResponse> SetMaintenanceState(const SchemaMaintenanceState& request);

boost::shared_future<FileChangeResponse> FileChange(const FileChangeEvents& request);

boost::shared_future<TriggerUpdateTaskResponse> TriggerUpdateTask(const UpdateTaskInfo& request);

boost::shared_future<FileChangeResponse> SingleFileAddition(const SingleFileAddEvent& request);

boost::shared_future<TriggerUpdateTaskResponse> TriggerSingleUpdateTask(const SingleUpdateTaskInfo& request);
```

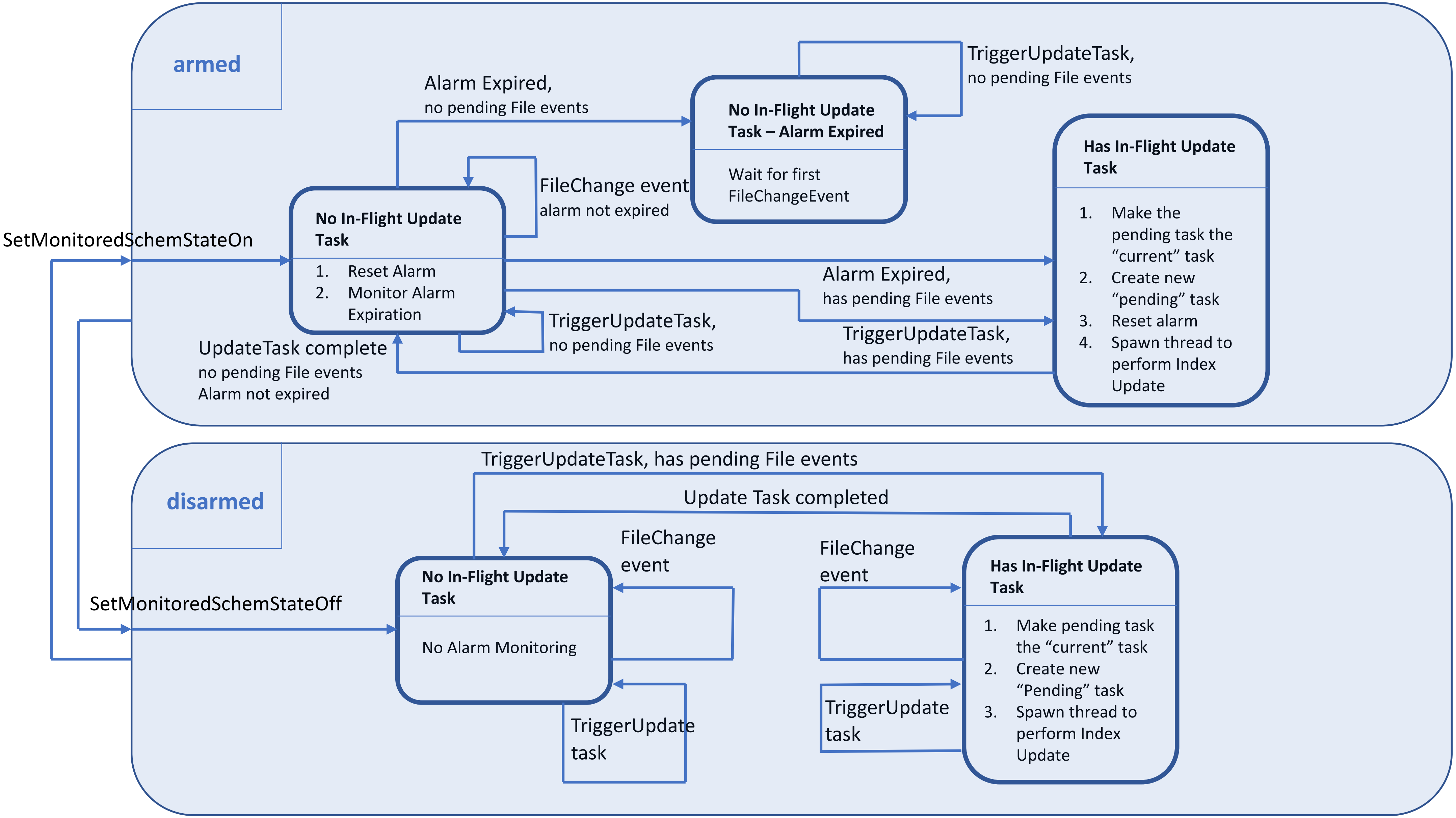
BDI Index Maintenance Service



BDI Index Maintenance Service



BDI Index Maintenance Service
State Transition Diagram for Maintained Schema



Event Loop in BDI Index Maintenance Service

```
void IndexMaintenanceServer::StartMonitoring() {   /* StartMonitoring */
      std::thread([this]() {  /* detached thread lambda */
      int i = 0;
      while (!stopMonitorThread_) {    /* infinite event loop */
         if (i % 60 == 0) {    /* execute once per minute */
            std::lock_guard<std::mutex> lm(schemaMu_);
            for (auto &ms : schemas_) {  /* loop over maintained schemas */

               // see if this schema
               //  a) has no current task
               //  b) has a pending task that is ready for execution and
               //  has events and autocommit is active
               if (ms.currentTask_) {  /* there is queued current task */
                     std::string taskParams;
                     ms.currentTask_->ToTaskRequestString(taskParams);
                     ProgressRequest progressRequest;
                     progressRequest.set_task(taskParams);
```

```
 auto collndxState = TaskStateType::NOT_STARTED;
pIndexManagerClient_->GetProgress(progressRequest)
   .then(boost::launch::deferred,
   [&collndxState](auto fResp) {
         auto resp = fResp.get();
         collndxState = resp.column_index_state();
   })
  .get();
if (collndxState == TaskStateType::FINISHED ||
    collndxState == TaskStateType::NOT_STARTED)
      ms.currentTask_ = nullptr;
}    /* there is queued current task */
```

continues on the next page

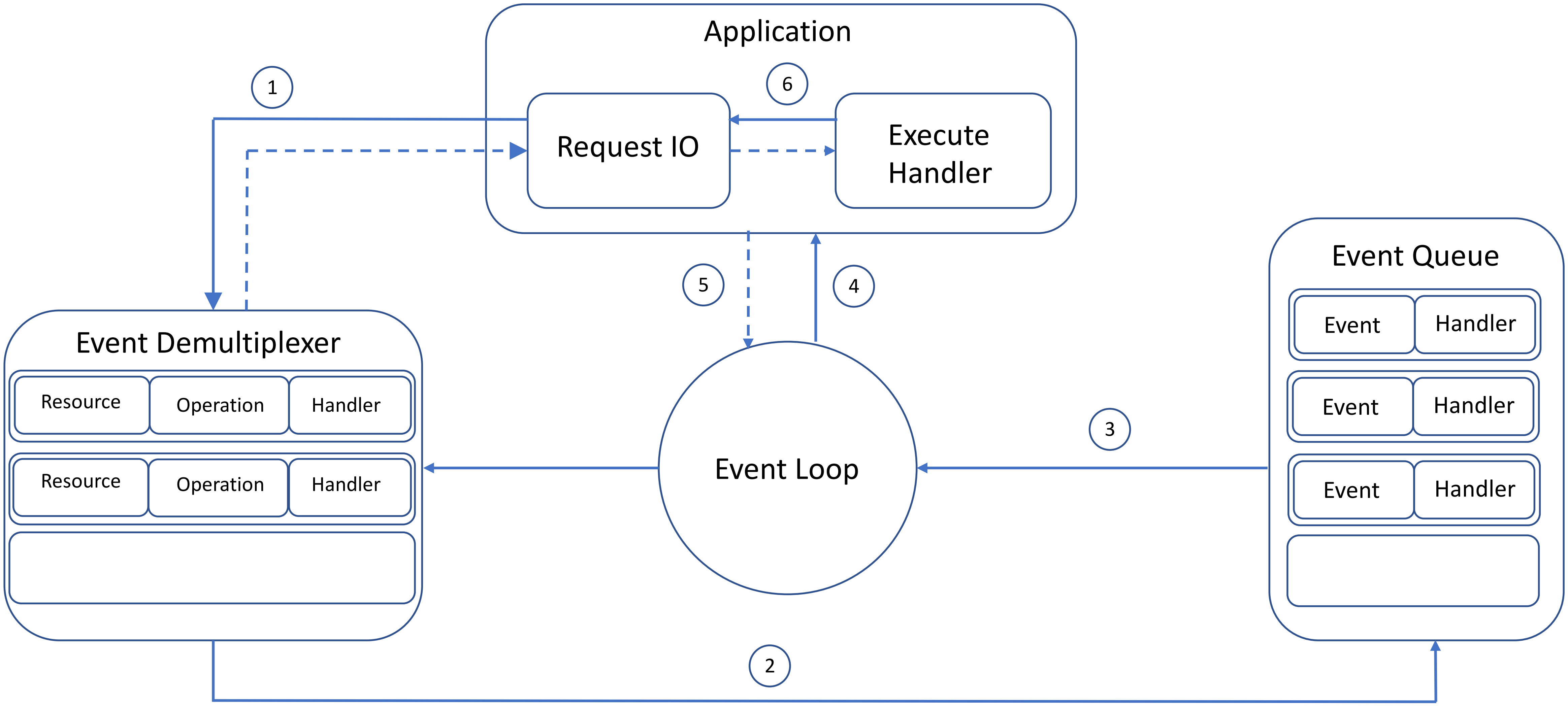
Event Loop in BDI Index Maintenance Service

continues from the previous page

```
 auto curTime = std::chrono::system_clock::now();

 if ( ms.acState_ == AutoCommitState::AC_STATE_ON &&
     !ms.currentTask_ && ms.pendingTask_ &&
     ms.pendingTask_->startOnOrAfter_ <= curTime &&
     ms.pendingTask_->HasEvents()) {
          // make the pending task the current task and create a
         // new pending task to capture all new events
         ms.currentTask_ = std::move(ms.pendingTask_);
         ms.pendingTask_ = std::make_unique<IndexUpdateTask>(
                 ++taskCount_, curTime += ms.GetUpdateInterval(),
                 ms.datasetName(), addIndexingTaskFunc_);

         ms.currentTask_->Start();

         std::this_thread::yield();
     } /* pendingTask_->HasEvents() */
 }   /* currentTask_ is done and it is time to start pendingTask_ /
} /* loop over maintained schemas */
```

```
                    i = 0;

                } /* execute once per minute */
                std::this_thread::sleep_for(1s);
                i++;
            }   /* infinite event loop */
        }).detach();
    }   /* detached thread lambda */
} /* StartMonitoring */
```

Reactive Event Loop Design Pattern



Reactive Event Loop Design Pattern

1. The app generates a new IO operation by submitting a request to the **Event Demultiplexer**. The app also specifies a handler which will be invoked when the operation completes. Submitting a new request to the **Event Demultiplexer** is a non-blocking call and it returns immediatelly control to the app.
2. When a set of IO operations completes, the **Event Demulitplexer** pushes the new events into the **Event Queue**.
3. The **Event Loop** iterates over the items of the **Event Queue**
4. For each **Event**, the associated **Handler** is invoked.
5. The **Handler**, which is part of the application code, will give back the control to the **Event Loop** when its execution completes.
6. New Asynchronous operations may be requested during the execution of the **Handler** (**5**), causing new operations to be inserted in the **Event Demultiplexer** (**1**) before the control is given back to the **Event Loop**.

Example of event loop-based application – Node.js