ZKPROTO v1

(Draft v17)

Table of Contents

Overview

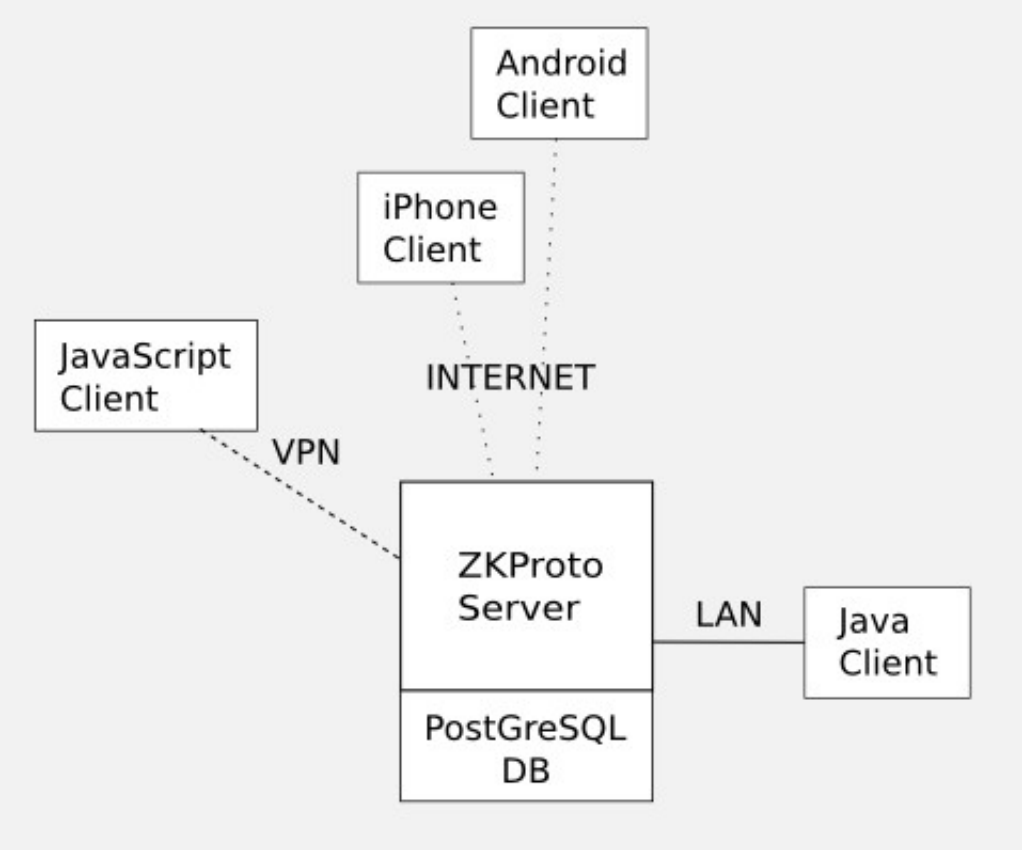
ZKProto aim is providing a modern, fast and reliable server-centralized protocol mainly for DB synchronization (although this is not mandatory) between clients (with/from ZKTeco devices that support it, or for other purposes too). A quick view of ZKProto's characteristics.

Pros	Cons
It allows normal work even when remote connection is not available.	Requires data duplication in all clients and server (this allows clients to work even if server is unreachable)
Read-only queries are performed locally, avoiding DB engine workload and network-bandwidth consumption.	Cannot read any data without getting replicated first *(only when using ZKTeco's Server)*
Database abstraction: each client can freely decide which DB backend to use.	
Support for several network-level protocols: raw TCP, HTTP, JSON...	
Support for several programming languages in both client and server implementations.	
Secure data transmission: RSA+AES encryption	
Versioning: clients/server running different protocol versions can work together.	
No external libraries dependency (this is generally true, but for some target languages some libraries might be required, e.g. Java and SLF4J).	
Multi-platform support (Windows, Linux, Mac, UNIX...).	
Automatic almost real-time database synchronization *(only when using ZKTeco's Server)*	

Introduction

ZKProto is a reliable, fast, modern, secure and flexible communication protocol based on the concepts of *middleware* and *RPC (Remote Procedure Call).* The middleware (in case of ZKProto we use Apache Thrift) takes care of all the raw communication aspects, like sockets and everything related to TCP/IP stack. The RPC part refers to the ability to call remote procedures/methods/functions like they were local.

ZKProto is a centralized protocol as shown in the above diagram, which means the clients never actually connect between themselves but all the communication is done through the server. The sole and unique purpose of the server is to replicate data (if needed) and resolve any data conflicts that might arise.



NOTE: the ZKProto Server depicted here is ZKTeco's implementation (which uses synchronization logic). Any developer can instead develop its own server with its own logic. Also note that server DB is private should never be accessed directly from any source that is not the ZKProto server itself.

ZKProto is not tied to any database structure and thus can be used over any database design, structure and engine. ZKProto does not make any assumptions about the structure/data of the database. It thus can be used with clients that use totally different DB engines. In this schema, clients are responsible to update their DB as they see fit.

ZKProto supports writing clients and servers in several programming languages like Java, C++, Python, PHP, Erlang and many others. Clients and servers can be written in totally different languages. Even each client can be in a different language. For example, client A could be coded in C++ while client B is in Java; and server in PHP. Being language-agnostic also means platform-agnostic, as ZKProto client/server will work as long as the platform supports the language in which it was built. While ZKProto base code is platform-agnostic, the developer must take care of writing portable code if this is needed.

ZKProto supports any transport protocol that can run over TCP, which includes raw binary, HTTP, HTTPS, JSON and several others, even custom ones.

Both ZKProto client and server do not depend on any external library to work (with some exceptions for some languages). A special tool (the Apache Thrift compiler, detailed in Appendix A) will generate the appropriate source code in the targeted programming language to include in your project and have access to ZKProto functionality. ZKProto never depends on any platform-specific external library (DLL, SO, etc...).

ZKProto supports different version in client and server. This means ZKProto will work even if client and server have different versions of the protocol. Even if so, it is highly recommended to have at least latest server version (that is, all your clients should have equal or previous ZKProto version than your server).

ZKProto supports AES encryption for secure data communication. AES is a standard encryption algorithm (ISO/IEC 18033-3) that was designed to be very secure and fast. Currently ZKProto only supports 256-bit key size. Support for other key sizes is planned for future releases. Please make sure your country laws allow the usage of AES 256 before activating it.

ZKProto Servers

ZKProto defines 2 types of server: data server (from now on referred to as *sync server*) and server-control server (from now on referred to as *control server*).

- *Sync server*: controls and performs all DB operations.
- *Control server*: provides information about and allows configuring sync server (i.e. currently connected clients). Clients are always the ones to initiate connection with the server.

ZKProto Sync Basic Operation

Sync server takes care of organizing all data distribution, validation and conflict resolving. Also serves clients any new DB modifications when they request it. Clients pull/push operations always from server (never directly with any other entity). The server offers 3 RPCs for this purpose:

- *open* – First call for a client when it opens a new connection. Sends information about client (i.e. client ID). You need to call this once per connection and before any call to pull/push.
- *push* – Called from client when it has new DB modifications to be sent to the server, or to inform the server it correctly received and processed sent DB modifications (after a *pull*).
- *pull* – Called from client when it wants to retreive any DB modifications that other clients have done. Server can also request client to perform a push through this call.

All calls are blocking (they do not return until server processing is finished). If any error happens, the server will throw a remote exception that extends *TException* (this is a Thrift internal exception). If call returns without exception, it was successful. If an exception is thrown, then the call has failed for the reasons explained in the exception specific subclass and message.

NOTE: there's no multithreading support for the same client, that is, you cannot call the RPCs concurrently.

ZKProto Sync Client Basic Operation Flow

1. *open* to start communication with server. Client sends some information about itself to help prepare the server to communicate with it. Server can reject the open call with a ZKProtoException for various reasons that are server-side implementation-dependent.
2. *pull* operations, store them if necessary and execute them if any. Keep in mind that a *pull* can be multi-part. In this case the client needs to keep pulling until there no more operations before doing any other call (namely *push*).
3. *push* if server orders device to push (through *PUSH* operation), if client decides it has new data to push, or to acknowledge result of fetched operations to the server, or all of the previous. Note that with ACKNOWLEDGE action you tell the server if pulled operations executed successfully or not (see ZKProtoOperation).
4. If you want to close the socket now, you can do it. Otherwise go to step 2 again (for suggestions about pros and cons for keeping socket open or closing it, see the FAQ).

In case opening communication with the server fails, the client should retry connecting after X milliseconds. In case the communication suddenly drops, client will restart communication from step 1.

Here is a simplified example in pseudo-code on how a ZKProto client works:

```
while running
        if connected
                do
                        fetch operations from server through pull
                        process fetched operations if any
                while not last operation batch
                push acknowledge if required (successful or failed) and any local pending DB modifications
        else
                connect
```

How to implement this depends on target programming language and data storage.

ZKProto Sync Server Basic Operation Flow

The server part should have the exact same DB model as the client. Data from a client will be sent to the server's DB, which in turns is sent to all other clients. ZKTeco already provides a ZKProto server, so you usually do not have to implement this part unless you want a different server implementation (features of ZKTeco's ZKProto Server are detailed here).

1. ZKProto server starts listening on configured port (defaults to 4372). The server must accept multiple incoming connections in this same port.

2. Once a device connects, it will call *open*. What the server does with this information is implementation-dependent, as well as what to require from a client (for example, a server can require clients to have the clock correctly set up).

3. The client calls *pull*. We check if we have any pending operations for this device and send them. How to check pending operations is implementation-dependent.

4. The client calls *push* to send new data to the server or to confirm received data. Whether server accepts or not a push call from a client and how it informs the client about this last situation is implementation-dependent.

ZKProtoSyncOpenInfo

ZKProtoOpenInfo is client information sent to server just after connecting (before any pull/push call). This information must be sent each time the client re-connects to server as well.

```
// Synchronization information
struct ZKProtoSyncOpenInfo {
        1: required i32 protocolVersion,      // ZKProto version
        2: required i64 clientId,             // Client UUID
        3: required string customId,          // Customer ID
        4: optional i32 maxOperations,        // Max size of ZKProtoOperations (in bytes)
        5: optional string publicKey,         // Public key for encryption start
        6: required i64 timestamp,            // Current client UTC time
        7: optional ZKProtoSyncTableFilter tableFilter, // Table filters if any
}
```

- *protocolVersion* – client ZKProto version implementation. As of the writing this document, this value has to be 1.

- *clientId* – client UUID. This is the unique identifier for this client. To avoid possible UUID collisions we suggest you generate this number randomly or use a value known to be almost unique (e.g. CPU ID).

- *customId* – client customized ID. This string can be anything and it is not processed by ZKProto. It's for

displaying purposes only.

- *maxOperations* – maximum size in bytes for operations as answer to *pull* call. This defines maximum data a *pull* will return. If not set, there will be virtually no limit. Recommended to set it in low-memory clients (e.g. embedded devices).
- *publicKey* – base16-encoded RSA public key to encrypt AES key with (see Encryption section). If not set, there will be no encryption.
- timestamp – client UTC time as milliseconds since January 1st 1970 00:00. How the server handles this field is implementation-dependent.
- *tableFilter* – a data structure that defines which tables the server should send this client updates. It can be a white or a black list. If not set, server assumes all tables are required.

ZKProtoSyncAckInfo

ZKProtoSyncAckInfo is the data structure returned from *open* call.

```
// Structure returned from open() call
struct ZKProtoSyncAckInfo {
        1: optional string encryptedKey,      // Encrypted key for secure communication
}
```

- *encryptedKey* – base 16-encoded encrypted AES key to be used to encrypt data. This key is encrypted with RSA public key sent by client in *open* call. This key will only encrypt *data* field of the *ZKProtoOperation* structure.

  NOTE: if data is encrypted and compressed, it must be first compressed then encrypted (and the opposite when receiving, that is first decrypted then decompressed).

ZKProtoSyncOperations

ZKProtoSyncOperations is a ZKProtoOperation list wrapper.

```
// Wrapper for a list of operations
struct ZKProtoSyncOperations {
        1: required list<ZKProtoSyncOperation> operations,  // List of operations
        2: required bool last,                              // Is this last part of a multipart?
}
```

- *operations* – list of all operations to execute
- last – if true, server still has operations pending to be pulled by this client, which requires the client to pull again; if false, no more operations are pending.

Here's a pseudo-code example:

```
do
        pull operations
        process operations
while not last
```

ZKProtoSyncOperation

ZKProtoOperation represents one operation to be performed in the client (it can be a DB modification, but not necessarily) or simply it holds data (*NOP* action).

```
// Operation to be performed
struct ZKProtoOperation {
       1: optional i64 id,                  // Operation UUID (-1 if no ID)
       2: required ZKProtoAction action,    // Action performed in operation
       3: optional i64 timestamp,           // UTC timestamp for this operation
       4: optional binary data,             // Data for the action
       5: optional bool compressed,         // Flag if data is compressed
}
```

- *id* – This operation universal 64-bit identifier. This ID should never repeat. Can be ommitted or set to -1 if the operation does not need to be identified or it's not a real operation.

- *action* – ZKProtoAction that defines which action this operation is describing. Current possible actions are:
  - NOP – No Operation, does nothing.
  - INSERT – Informs of new data in DB.
  - UPDATE – Informs of data update in DB.
  - DELETE – Informs of data removal in DB.
  - PUSH – Informs client it has to push all data it has.
  - ACKNOWLEDGE – Informs server about status of client after executing operations from a pull.

- *timestamp* – When this operation was perfomed (UTC as milliseconds since January 1st 1970 00:00)

- *data* – String that represents the data for the operation. The format of this data changes depending on the operation action, but keys and values have always the same format, namely *key=value;.*

  *NOTE: neither key nor value can have any delimiter (**= ; ,**) as part of them. This will be fixed in future versions.*

  - NOP – No data format specified, implementation-dependent.

  - INSERT – *"table=tableName;field1=field1Value;[...]fieldN=fieldNValue;"* where *tableName* is the name of the table to insert into; *fieldN* is the field name; *fieldNValue* is the value to set to *fieldN*. Example: *"table=entity;_id=1245124;enabled=1"* is equivalent to SQL sentence *INSERT INTO entity (_id, enabled) VALUES (1245124, 1).*

  - UPDATE - *"table=tableName;pk=field1=value1,[...]fieldN=valueN,; field1=newField1Value; [...]fieldN=newFieldNValue;"* where *tableName* is the name of the table to update; *fieldN=valueN* are update conditions; *fieldN* is the field name; *newFieldNValue* is the new value to set to *fieldN*. Example: *"table=entity;pk=_id=1245124;enabled=1"* is equivalent to SQL sentence *UPDATE entity SET enabled = 1 WHERE _id = 1245124.*

  - DELETE action - *"table=tableName;[...]fieldN=valueN;"* where *tableName* is the name of the table to delete from; *fieldN=valueN* are delete conditions as in SQL. Example: *"table=entity;_id=1245124 AND _id=44514578"* is equivalent to SQL sentence *DELETE FROM entity WHERE _id = 1245124 AND _id = 44514578*. Note that in this case DELETE can only accept one field for WHERE condition (but it can be as complex as you want). Note that if no field-value condition is specified, the whole table will be deleted.

  - PUSH action – No data field.

  - ACKNOWLEDGE action - *"cause=message;failed=operation"* where *message* is why operation failed, and *operation* is the operation that failed. Both are strings. How server handles failed operations is implementation-dependent.

- *compressed* – Indicates if *data* field is compressed (see Compression section). Note that if *data* is also encrypted, compression is done before encryption (otherwise the compression would be useless, see this discussion about why).

For a complete ZKProto client example in Java, you can check *com.zktechnology.simplezkprotoclient.zkproto.ZKProtoClient* source code.

ZKProto Control Basic Operation

ZKProto control server is meant for special clients not interested about business data but about ZKProto sync server data, and modify server behavior and configuration. It uses a different port and different calls than ZKProto sync server. ZKProto control clients are interested about ZKProto itself: which sync/control clients are actually connected, what's their information and status... The control server offers only two RPCs:

- *open* - First call for a client when it opens a new connection. Sends information about client.
- *execute* – Executes one given control operation and returns the result if any.

ZKProto Control Client Basic Operation Flow

1. *open* to start communication with server
2. *execute* to perform a control operation (i.e. change client zone) or to update current server status (get sync client statuses, get existing zones, etc...).

ZKProto Control Server Basic Operation Flow

The control server does not do much except listen to the client and answer its petitions.

1. ZKProto server starts listening on configured port (defaults to 4373). Each control client will just connect once to this port. Of course the server accepts multiple clients in this port, as with the sync server.
2. Once a device connects, it will call *open*. Here we initialize information about the client.
3. The client eventually calls *execute*. Server checks the requested action, performs it and returns data only if action requires data back. If action does not return data, then empty result means OK. If anything has gone wrong, a *ZKProtoException* will be thrown.

ZKTeco already provides a ZKProto server, so you usually do not have to implement this part unless you want a specific server.

ZKProtoControlOperation

ZKProtoControlOperation is a Thrift structure that defines what a control operation looks like:

```
// Operation to be performed
struct ZKProtoControlOperation {
        1: required ZKProtoControlTarget target,     // Control target
        2: required ZKProtoControlAction action,     // Action performed in operation
        3: optional binary data,                     // Data for the action (if any)
        4: optional bool compressed,                 // Flag if data is compressed
}
```

- *target* – target this action will affect. Each target only accepts a specific subset of actions. Current possible

targets are:

- NONE – when action doesn't need any target.
- SERVER – action refers to the server itself.
- CLIENT – action refers to all or a specific known client.
- ZONE – action refers to all or a specifc zone.

- action – which action to execute for specified target. Current possible actions are, grouped by target:

  NOTE: parameters withing brackets [ ] are optional.
  *NOTE: values cannot have any delimiter (**;**) as part of them. This will be fixed in future versions.*

  - NOP – no action, used to include plain data, for example an answer to a data petition. This can be included in any target
  - SERVER:
    - GET_CONTROL_SETTINGS: returns control server settings. Needs no parameters. Returns *port, debug flag*.
    - SET_CONTROL_SETTINGS: set control server settings. Parameters: *port, debug flag, [login flag], [new password]*. Returns nothing.
    - GET_SYNC_SETTINGS: returns sync server settings. Needs no parameters. Returns *port, debug flag*.
    - SET_SYNC_SETTINGS: set sync server settings. Parameters: *port, debug flag*. Returns nothing.
    - LOGIN_CONTROL: sends login password for control. This action must be sent after calling open to be able to execute further commands, otherwise all commands from this socket are denied until successfully logged in. Parameters: *password*. Returns nothing.
  - CLIENT
    - GET_ALL_CLIENTS: returns list of all sync clients. Parameters: none. Returns list of all clients with each entry as follows: _id, ip_address, code, description, is_replicated, is_connected, is_authorized, policy, id_zone.
    - NEW_CLIENT: creates a new client. Parameters: id, code, description. Returns nothing.
    - EDIT_CLIENT: edits an already existing client. Parameters: id, description, auth on/off, policy. Returns nothing.
    - SET_TO_ZONE: moves a client to a new zone. Parameters: client id, zone id, policy. Returns nothing.
    - REMOVE_CLIENT: permanently deletes a client. Parameters: id. Returns nothing.
  - ZONE
    - GET_ALL_ZONES: returns list of all zones. Parameters: none. Returns list of all zones with each entry as follows: id, name.
    - NEW_ZONE: creates a new zone. Parameters: *name, first script id, second script id... n-th script id*. Returns zone id.
    - GET_ZONE_CLIENT: returns a specific zone cilents. Parameters: zone id. Returns list of all clients with each entry as follows: _id, ip_address, code, description, is_replicated,

is_connected, is_authorized, policy, id_zone.

- REMOVE_ZONE: permanently deletes a zone. Parameters: id. Returns nothing.
- GET_CONFLICTS: get all failed operations and fail reason. Parameters: none. Returns list of conflicts.
- DELETE_CONFLICTS: deletes operation failed logs. Parameters: none. Returns nothing.

Compression

Compression is made using ZLIB compression algorithm and it is only applied to *ZKProtoOperation.data* string field/member. Note that compressed data has no ZLIB header. Most modern programming languages already include classes to handle ZLIB, and older ones have libraries for it.

Encryption

Encryption schema uses 2 different standard encryption algorithms: RSA and AES. These two algorithms are today's standard algorithms for assymmetric and symmetric encryption respectively. Exact RSA and AES algorithms being used (hash and padding) are RSA-ECB-PKCS1 and AES-ECB-PCKCS5 respectively. RSA is used to share the AES key used to actually encrypt data. The process flow is as follows:

1. Client generates an RSA key pair. We will call the public key *CA* and the private key *CB*. On opening communication with server, it sends *CA* to the server (see ZKProtoOpenInfo data structure).

2. Once a client connects, server generates an AES key to use for this client. We will call this key *SK*. The client *CA* key is used to encrypt *SK* and send back to client (see ZKProtoAckInfo data structure).

3. Client uses *CB* to decrypt *SK* from server. Now both sides have the *SK* key and can use this key for secure communciation.

Note that if compression and encryption are both used, you must first compress then encrypt, otherwise the compression will have small to no effect (due to the high entropy from AES encryption).

Please make sure your country laws allow the usage of AES encryption and under which restrictions.

ZKTeco's ZKProto Server

ZKTeco provides a ZKProto server implementation in Java 7 for PC (fully tested in Windows and Linux). This section is going to describe its working details that are not specified in previous parts and how to set it up. This server implementation is also database agnostic but the PostgreSQL server scripts are currently coupled to ZPAD T&A firmware DB structure. Support for any DB structure out-of-the-box is currently planned and in development.

This section refers to ZKProto Server version 0.15.148, which is latest version as of this document writing.

Setting up

- You need the server latest JAR (*zkprotoserver-X.X.X.jar,* where X.X.X is the server version*).*
- You need an up and working PostgreSQL 9.1+ server, configured in the default port (5432) and accessable with the user *postgres* and password *postgres.*

Server rules

In this section we will explain ZKTeco's ZKProto Server internal logic. Please note that this is merely informative. Any ZKProto client implementation should just execute the commands received by the server.

ZKTeco's Server synchronizes data. This means data is replicated between the clients, making all databases of these clients the same. As an abstraction, you can think of the synchronization prodecure like all clients are actually using the same database.

ZKProto server supports *zoning*. This means the server can be configured to group clients so these clients only synchronize data between the members of their group and not with other clients. Each group is called a *zone* in ZKProto terminology. For example, if clients A and B belong to zone 1, and clients C and D belong to zone 2, changes in client A will only be reflected in client B, not C and D.

Open

1. If this client does not exist, create an entry for it to log that this client tried to *open*.
2. If this client already called *open* on this same socket connection, throw an exception.
3. Check if client ZKProto version is equal or lower than server's. If not, throw an exception.
4. Check if client is authorized. If not, throw an exception.
5. Check if client timestamp is in accepted range (current accepted range is 1 minute).
6. Assign this client to its zone. If no zone assigned, assign to default zone (*limbo*).

Pull

1. If this client didn't call *open* on this same socket connection, throw an exception.
2. If this client has been replicated, go to next step. Otherwise prepare to send this client initialization operations (see *Policy* below) and clear any pending operations this client might have. Go to 4.

3. If this client has any pending operations, go to next step. Otherwise check if any new pending changes to send and put them as pending to send.

4. Send to client next pending operations batch.

There are 2 different policies for initializing a client: *RESET* and *PUSH*. *RESET* policy will replicate server DB to client (thus client will lose any data it has previous to this). *PUSH* policy will first instruct the client client to push its data first before getting replicated from server DB (thus no data is lost from this client). *PUSH* policy is much slower and consumes much more bandwidth and traffic than *RESET* policy.

Push

1. If this client didn't call *open* on this same socket connection, throw an exception.

2. If zone has been changed, discard any operations from this client.

3. Process operations pushed (uploaded) by the client in order:

    1. If operation is DB operation (*INSERT*, *UPDATE*, *DELETE*) and client is replicated, execute it. If client is not replicated, throw an exception. If there's an error executing the DB operation, server can determine if this error is harmful to data integrity or not: it can either throw an exception and stop processing, or insert it in a conflict table.

    2. If operation is an *ACKNOWLEDGE*, see if it has failed or succeeded. If failed, set client as not-replicated (see consequences of this on Pull). If success, mark as replicated and clear any pending operations.

For a correct working of the protocol, please only send *ACKNOWLEDGE* with no failed operations from your client when previous *pull* operations were successfully executed. If any error happened, immediately abort operation execution and inform the server through *ACKNOWLEDGE*.

Control server supported features

- Get/set the following control server settings: TCP port, debug logs, authentication, and password. Password is hashed using secure BCrypt hashing over optional -but recommended- AES encryption communication.

- Get/set the following sync server settings: TCP port, and debug logging.

- Get all existing sync zones, and get all clients associated to a given zone.

- Create a new zone, edit and remove an existing one. Zone names are restricted to lowercase english letters, numbers and underscore, and cannot be start with number. There's no limit to number of zones. You can only remove a zone with no clients associated.

- Get all sync clients (all zones mixed).

- Create a new client, edit and remove an existing one. Client names have no restrictions. When creating a client, you can specify its UUID, its code and its description. When editing a client, you can edit its description, its authorization status, and its policy (see next point).

Internals details

In this section I'm going to explain how current ZKTeco's server works internally. This can help you customize the server for your specific business model and needs without having to do any further developing.

Database structure

All tables used by ZKProto Server are stored in *zkproto* schema, and are the following:

- ***client****:* list of all known clients.
  - _id: UUID
  - ip_address: last known socket (stored as IP:port)
  - code: human-readable name
  - description: additional optional description
  - is_replicated: flag that indicates if this client has already successfully replicated data from server.
  - is_authorized: flag that indicates if this client is authorized or not. A client can only send/receive commands if authorized, otherwise the socket is closed after *open* call.
  - policy: which policy to apply when replicating. 0 is NONE, 1 is RESET, 2 is PUSH.
  - last_pull: timestamp in Java format (milliseconds since epoch) of last time this client pulled successfully.
- ***zone:*** list of all zones.
  - _id: zone UUID
  - name: zone name.
- ***client2zone:*** assigns zone to each client.
  - id_zone: assigned zone ID
  - id_client: client ID
- ***operation_log:*** temporary table that holds operations that still need to be delivered to clients. Once the operation has been sent to all clients, it is deleted.
  - _id: operation UUID
  - date: timestamp in Java format (milliseconds since epoch) when this operation happened
  - action: operation action
  - info: operation data
  - id_zone: which zone this operation belongs to
  - id_client: which client sent this operation
- ***script:*** SQL scripts used for initializing a zone's schema initial structure and data.
  - _id: script UUID
  - sql: script SQL code

- ***script2zone:*** assigns initialization scripts to zones. When creating a new schema based on a zone, the SQL scripts specified for this zone will be executed in order.
    - id_zone: zone
    - ordering: order in which the scripts are executed
    - id_script: script to execute
- ***conflict:*** which operations have failed and why.
    - _id: operation ID that failed
    - date: timestamp in Java format (milliseconds since epoch) when this operation happened
    - action: operation action
    - info: operation data
    - id_zone: which zone this operation belongs to
    - id_client: which client sent this operation
    - error: error description
- ***pending_operation:*** which operations are pending for each client. Once client acknowledges this operation, the corresponding entry is deleted.
    - id_client: client ID
    - id_operation: operation ID
- ***setting:*** holds server settings.
    - key: specifies which server the settings are meant for.
    - values: values for settings as ***hstore*** (setting=>value).

Thrid-Party Development using ZKProto

This section explains how should a thrid-party proceed to use ZKProto to get/send data from/to other clients that use ZKProto, or to implant ZKProto as their communication protocol. There are 2 choices for a thrid-party: use ZKTeco's ZKProto Server, or implement their own server. Usually using ZKTeco's server should be the way to go, unless the internal synchronization logic is not suitable to your design/context.

If you have any doubts about which way to go, don't hesitate contacting current ZKProto maintainer so we can advise you better.

Creating your ZKProto client

For this section, we will assume you're using ZKTeco's ZKProto Server, to check if your client can successfully connect and send/receive commands to/from server. This is the most common situation for a developer. In this case the third-party software should simply generate classes for target language using Thrift (as explained previously) and implement a ZKProto sync client, and optionally a ZKProto control client (also as explained previously).

open call handling is trivial and as explained in this document. When called for first time, the server will first send you DELETE operations to clear the DB then all DB data as INSERTs. This way you will have the same data as the server. It's up to your client software to decide what to do with these operations.

To get current DB modifications in other clients (like ZPAD devices for example), you *pull* from the server as explained in this document. Again it's up to your software to decide what to do with this data and how and where to store it (this is outside the scope of ZKProto and thus out of scope of this document).

To send DB modifications to other clients you *push* to the server. Your client is entirely responsible of delivering any pending operations to the server. If you're using different zones remember that you will be sending these modifications only to the clients beloning to the zone your client currently belongs to. Also remember to *ACKNOWLEDGE* any last pulled operations (be it success or failure) first before pushing any operation to the server. Note that you can push *ACKNOWLEDGE* as the first element of an operation list.

In this section, we will guide you through the building of a very simple ZKProto sync client from scratch to test with ZKTeco's ZKProto Server. For this, we will use the following tools:

- Ubuntu 12.04, although we won't be using any Linux-specific tools/methods, so the instructions can be applied to any other OS (Windows, Mac...).
- Java 7 or above installed (we will use OpenJDK 1.7, you're free to use your favorite Java implementation).
- Eclipse 4.3, download the relevant package for your OS).
- Build the Apache Thrift compiler (see Appendix A on how to build it). This step will automatically generate a Java library: *libthrift-0.9.1.jar*.
- ZKProto Apache Thrift Interface file for sync server: *zkprotosync.thrift* (should be included in the ZKProto SDK).

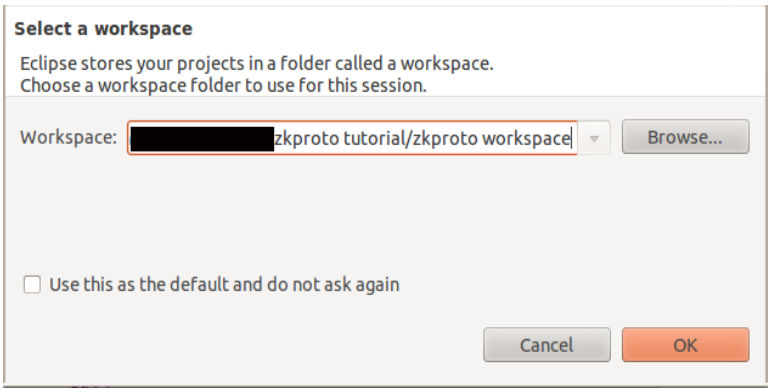
Now that we have the basic tools, let's start with the implementation itself:

1. Make sure your Java JDK is properly installed. Open a console and run *java -version*.

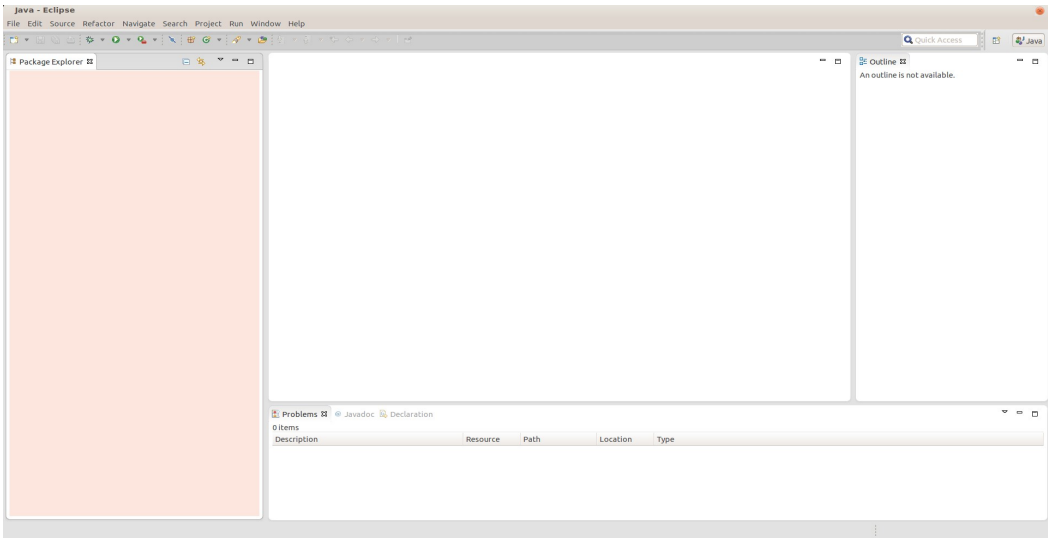
```
java version "1.7.0_55"
OpenJDK Runtime Environment (IcedTea 2.4.7) (7u55-2.4.7-1ubuntu1~0.12.04.2)
OpenJDK 64-Bit Server VM (build 24.51-b03, mixed mode)
```

2. Create a directory called *zkproto tutorial* wherever you feel like. We will be using it as our operations HQ.

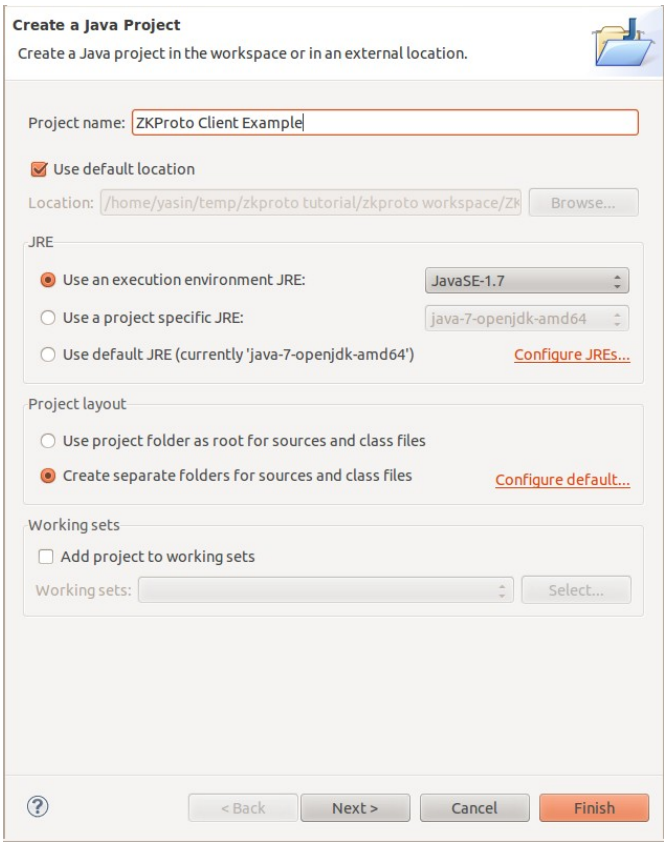
3. Extract Eclipse 4.3. Run *eclipse* executable. It will ask you for a workspace location, we create a new directory under *zkproto tutorial* called *zkproto workspace* to use as Eclipse workspace.



4. Close the Welcome window and you'll be presented with Eclipse's default workspace layout.



5. Create a new Java project: *File* > *New* > *Java Project*. You will be presented with the following window, please fill as indicated and click *Finish.*

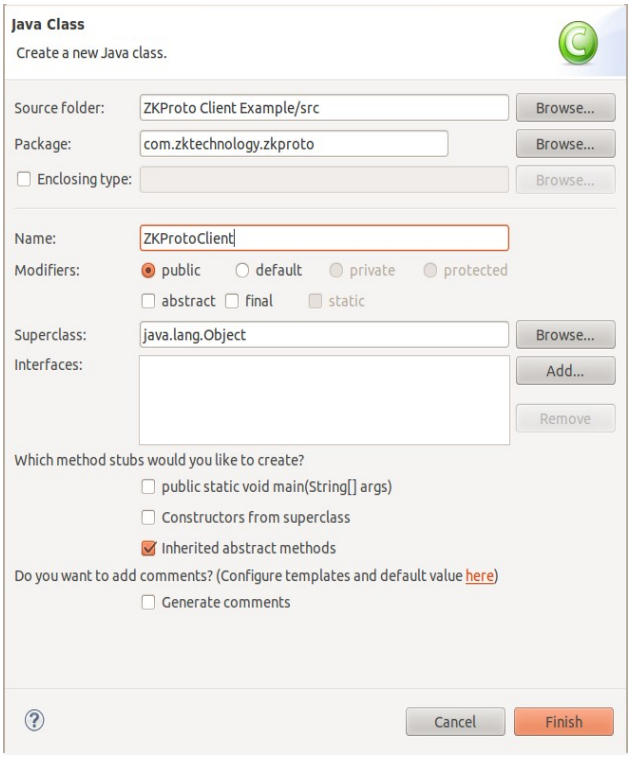


6. The project should appear in the *Package Explorer* window. Create a new folder *libs* inside the project. Right-click on the project: *New* > *Folder* and name it *libs*. Copy *libthrift-0.9.1.jar* to this folder. Include the library into the project build path: right-click on the project > *Build Path* > *Configure Build Path...* > *Libraries* > *Add JARs...* and select *libthrift-0.9.1.jar.* Close OK all windows. You also need to include the SL4FJ logger library (which is almost-standard Java library for logging) with the same procedure as you did with *libthrift-0.9.1.jar.* You can download it from here (the exact JAR to include in the build path is *slf4j-api-X.X.X.jar* (X.X.X being the current version number). This library requirement is exclusively for Java.

7. Copy *zkprotosync.thrift* to *zkproto tutorial directory.* In a console, move (*cd*) to *zkproto tutorial* directory and generate the Java classes and library by executing *thrift -gen java zkprotosync.thrift.* This will create a directory called *gen-java* where you can find the Java classes.

8. Expand the project and right-click on *src* > *New* > *Package.* Name it *com.zktechnology.zkproto* and click *Finish.* Copy the Thrfit-generated Java classes to this package. You can copy them in the your OS file explorer and paste in Eclipse (right-click on the created package and paste).

9. Now we have everything ready to start implementing our ZKProto client. To start, we create a new class called *ZKProtoClient:* right-click on the package > *New* > *Class* and name it *ZKProtoClient.* Click *Finish.*



10. We need an entry point for our example program, so we create a *main* static method in *ZKProtoClient.*

```
package com.zktechnology.zkproto;

public class ZKProtoClient {
        public static void main (final String[] args) {
        }
}
```

11. First thing to do is to open a TCP connection to the ZKProto server (assuming server is running on the same machine on default port 4372), which is as simple as follows:

```
final TTransport transport = new TSocket("localhost", 4372);
transport.open();
```

12. Now we need to identify our client to the server. For this, we must first create a ZKProto sync client based on our previous TCP socket, then identify to the sync server using *open*. This step can be implemented as follows:

```
final Client client = new ZKProtoSyncService.Client(
                new TBinaryProtocol(transport));
final int protocolVersion = 1;
final long clientUUID = 1;
final ZKProtoSyncOpenInfo openInfo = new ZKProtoSyncOpenInfo(
                protocolVersion, clientUUID, "ZKProto test client",
                System.currentTimeMillis());
client.open(openInfo);
```

13. Next, we want to *pull*, which is first thing a client should do after open. Since we already have a ZKProto sync client instance, we will use it to pull. Since this is just an example, we will just output the result of this pull.

```
final ZKProtoSyncOperations operations = client.pull();
System.out.println(operations);
```

14. After pull, we have to inform the server we received (and eventually executed) all operations successfully. For this, we will push an ACKNOWLEDGE operation.

```
final ZKProtoSyncOperation ackOperation = new ZKProtoSyncOperation(-1,
                ZKProtoSyncAction.ACKNOWLEDGE);
final List<ZKProtoSyncOperation> opsToPush = new ArrayList<>();
opsToPush.add(ackOperation);
client.push(opsToPush);
System.out.println("Ack success");
```

15. Finally we close the transport (the TCP socket).

```
transport.close();
```

So, resuming, the final client sample is as follows:

```
package com.zktechnology.zkproto;

import java.util.List;
import java.util.ArrayList;
import org.apache.thrift.TException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;
import com.zktechnology.zkproto.ZKProtoSyncService.Client;

public class ZKProtoClient {

        public static void main(final String[] args) throws ZKProtoException,
                        TException {

                // Open TCP socket connection to server
                final TTransport transport = new TSocket("localhost", 4372);
                transport.open();
                System.out.println("Connection with server established");

                // Create a client and call ZKProto's open
                final Client client = new ZKProtoSyncService.Client(
                                new TBinaryProtocol(transport));
                final int protocolVersion = 1;
                final long clientUUID = 1;
                final ZKProtoSyncOpenInfo openInfo = new ZKProtoSyncOpenInfo(
                                protocolVersion, clientUUID, "ZKProto test client",
                                System.currentTimeMillis());
                System.out.println("Opening ");
                client.open(openInfo);
                System.out.println("Open success");

                // First pull
                final ZKProtoSyncOperations operations = client.pull();
                System.out.println("Pulled operations successfully");
                System.out.println(operations);

                // Push ack
                final ZKProtoSyncOperation ackOperation = new ZKProtoSyncOperation(-1,
                                ZKProtoSyncAction.ACKNOWLEDGE);
                final List<ZKProtoSyncOperation> opsToPush = new ArrayList<>();
                opsToPush.add(ackOperation);
                client.push(opsToPush);
                System.out.println("Ack success");

                // Close connection
                transport.close();
                System.out.println("Finished successfully");
        }
```

```
}
```

Note that if you execute this client against a ZKProto server for the first time, it will raise a *NOT_AUTHORIZED* exception:

```
Exception in thread "main" ZKProtoException(xcause:NOT_AUTHORIZED, data:Client not authorized)
        at com.zktechnology.zkproto.ZKProtoSyncService$open_result$open_resultStandardScheme.read(ZKProtoSyncService.java:1280)
        at com.zktechnology.zkproto.ZKProtoSyncService$open_result$open_resultStandardScheme.read(ZKProtoSyncService.java:1)
        at com.zktechnology.zkproto.ZKProtoSyncService$open_result.read(ZKProtoSyncService.java:1196)
        at org.apache.thrift.TServiceClient.receiveBase(TServiceClient.java:78)
        at com.zktechnology.zkproto.ZKProtoSyncService$Client.recv_open(ZKProtoSyncService.java:93)
        at com.zktechnology.zkproto.ZKProtoSyncService$Client.open(ZKProtoSyncService.java:80)
        at com.zktechnology.zkproto.ZKProtoClient.main(ZKProtoClient.java:30)
```

This is absolutely normal because this client is not authorized and when calling open, the server will refuse to complete this call. After authorizing the client with RESET policy, the output of this client example should be as following:

```
Connection with server established
Opening
Open success
Pulled operations successfully
ZKProtoSyncOperations(operations:[ZKProtoSyncOperation(id:-1, action:DELETE, timestamp:0, data:74 61 62 6C 65
3D 65 6D 70 6C 6F 79 65 65 5F 6C 6F 67 69 6E 5F 63 6F 6D 62 69 6E 61 74 69 6F 6E), ZKProtoSyncOperation(id:-
1, action:INSERT, data:74 61 62 6C 65 3D 65 6D 70 6C 6F 79 65 65 5F 6C 6F 67 69 6E 5F 63 6F 6D 62 69 6E 61 74
69 6F 6E 3B 6C 6F 67 69 6E 5F 63 6F 6D 62 69 6E 61 74 69 6F 6E 3D 33 3B 69 64 5F 65 6D 70 6C 6F 79 65 65 3D
31 3B), ZKProtoSyncOperation(id:-1, action:DELETE, timestamp:0, data:74 61 62 6C 65 3D 72 6F 6C 65 32 65 6E
74 69 74 79), ZKProtoSyncOperation(id:-1, action:INSERT, data:74 61 62 6C 65 3D 72 6F 6C 65 32 65 6E 74 69 74
79 3B 69 64 5F 72 6F 6C 65 3D 31 3B 69 64 5F 65 6E 74 69 74 79 3D 31 3B), ZKProtoSyncOperation(id:-1,
action:INSERT, data:74 61 62 6C 65 3D 72 6F 6C 65 32 65 6E 74 69 74 79 3B 69 64 5F 72 6F 6C 65 3D 32 3B 69 64
5F 65 6E 74 69 74 79 3D 31 3B), ZKProtoSyncOperation(id:-1, action:INSERT, data:74 61 62 6C 65 3D 72 6F 6C 65
32 65 6E 74 69 74 79 3B 69 64 5F 72 6F 6C 65 3D 33 3B 69 64 5F 65 6E 74 69 74 79 3D 31 3B),
ZKProtoSyncOperation(id:-1, action:INSERT, data:74 61 62 6C 65 3D 72 6F 6C 65 32 65 6E 74 69 74 79 3B 69 64
5F 72 6F 6C 65 3D 34 3B 69 64 5F 65 6E 74 69 74 79 3D 31 3B), ZKProtoSyncOperation(id:-1, action:DELETE,
timestamp:0, data:74 61 62 6C 65 3D 72 6F 6C 65), ZKProtoSyncOperation(id:-1, action:INSERT, data:74 61 62 6C
65 3D 72 6F 6C 65 3B 6E 61 6D 65 3D 53 75 70 65 72 20 41 64 6D 69 6E 3B 5F 69 64 3D 31 3B),
ZKProtoSyncOperation(id:-1, action:INSERT, data:74 61 62 6C 65 3D 72 6F 6C 65 3B 6E 61 6D 65 3D 45 6E 72 6F
6C 6C 65 72 3B 5F 69 64 3D 32 3B), [cut for brievity...]], last:true)
Ack success
Finished successfully
```

The server output after running with this sample client should be as following:

```
Wed May 28 10:47:53 CEST 2014 - >================================================<
Wed May 28 10:47:53 CEST 2014 - Starting ZKProto Server 0.16.1
Wed May 28 10:47:53 CEST 2014 - Starting server ZKProtoSync on port 4372
Wed May 28 10:47:53 CEST 2014 - Starting server ZKProtoControl on port 4373
Wed May 28 10:47:56 CEST 2014 - SYNC >>  ZKProto test client: open()
Wed May 28 10:47:56 CEST 2014 - SYNC >>  ZKProto test client: ZKProto test client (UUID: 1, IP:
127.0.0.1:43963, ZKProto version: 1)
Wed May 28 10:47:56 CEST 2014 - SYNC >>  ZKProto test client: Buffer capacity: 2147483647 bytes
Wed May 28 10:47:56 CEST 2014 - SYNC >>  ZKProto test client: Authorized with policy RESET
Wed May 28 10:47:56 CEST 2014 - SYNC >>  ZKProto test client: Policy: RESET
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Zone: limbo
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: NOT replicated
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Doesn't need encryption
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: pull()
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Preparing RESET policy
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Initialization op count: 161
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Pull returning count: 161
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Pull isLast: true
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: push()
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Pushed : 1
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Processing index 0
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Executing pushed operation: id: -1 | action:
ACKNOWLEDGE |
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: Cleaning pending operations for 1
Wed May 28 10:47:58 CEST 2014 - SYNC >>  ZKProto test client: push() success
```

```
Wed May 28 10:48:00 CEST 2014 - >> SYNC PROCESSOR >> : IP 127.0.0.1:43963 has disconnected.
Wed May 28 10:48:00 CEST 2014 - SYNC >>  ZKProto test client: Disconnected.
```

Note that obviously this is not a real working client because operations are not actually executed. How to execute the operations is highly dependent on underlying DB, any ORM layers, language, and other client characteristics. We think that with explanations given in the ZKProtoSyncOperation section it should be relatively simple to execute any operation received from the server.

For the control client, the procedure is very similar except that you need to use *zkprotocontrol.thrift* instead of *zkprotosync.thrift.*

Creating your ZKProto server

In this section, we will guide you through the building of a very simple ZKProto sync server from scratch. Implementing the server part is very similar to the client, except that we need to implement the *open*, *pull* and *push* calls instead of just calling them. We will assume you're using a ZKProto client implementation (which exact implementation is irrelevant for this example). Pre-requisites, tools and Eclipse workspace are same than previous section Creating your ZKProto client.

1. Create a new Java project: *File > New > Java Project* just like with client. Project name will be *ZKProto Server Example.*

2. The project should appear in the *Package Explorer* window. Create a new folder *libs* inside the project. Right-click on the project: *New > Folder* and name it *libs*. Copy *libthrift-0.9.1.jar* to this folder. Include the library into the project build path: right-click on the project > *Build Path > Configure Build Path... > Libraries > Add JARs...* and select *libthrift-0.9.1.jar.* Close OK all windows. You also need to include the SL4FJ logger library (which is almost-standard Java library for logging) with the same procedure as you did with *libthrift-0.9.1.jar.* You can download it from here (the exact JAR to include in the build path is *slf4j-api-X.X.X.jar* (X.X.X being the current version number). This library requirement is exclusively for Java.

3. Reuse the classes generated by Thrift compiler in Creating your ZKProto client section, or re-generate them: copy *zkprotosync.thrift* to *zkproto tutorial* directory. In a console, move (*cd*) to *zkproto tutorial* directory and generate the Java classes and library by executing *thrift -gen java zkprotosync.thrift.* This will create a directory called *gen-java* where you can find the Java classes.

4. Expand the project and right-click on *src > New > Package.* Name it *com.zktechnology.zkproto* and click *Finish.* Copy the Thrfit-generated Java classes to this package. You can copy them in the your OS file explorer and paste in Eclipse (right-click on the created package and paste).

5. Create a new class in this package name *ZKProtoServer*. Create *main* entry method for this class.

```
package com.zktechnology.zkproto;
public class ZKProtoServer {
        public static void main(String[] args) {
        }
}
```

6. Next we will create a new class in *ZKProtoServer.java* which will implement our service RPC calls (*open*, *pull*, and *push*). In Thrift parlance, this is called the *service handler*. For this simple example, the handler will have no logic, we will just print a string with the data passed to the call and immediately return.

```
class ZKProtoServerHandler implements ZKProtoSyncService.Iface {
        @Override
        public ZKProtoSyncAckInfo open(final ZKProtoSyncOpenInfo openInfo)
                        throws ZKProtoException, TException {
                System.out.println("Open called with argument: " + openInfo);
                return new ZKProtoSyncAckInfo();
```

```
        }

        @Override
        public void push(final List<ZKProtoSyncOperation> operations)
                        throws ZKProtoException, TException {
                System.out.println("Push called with argument: " + operations);
        }

        @Override
        public ZKProtoSyncOperations pull() throws ZKProtoException, TException {
                System.out.println("Pull called");
                return new ZKProtoSyncOperations(new ArrayList<ZKProtoSyncOperation>(), true);
        }
}
```

Note that we don't return null in the calls. This is simply a convention to avoid null checks on client side.

7. Now that we have defined our hanlder, we set up the server with it hanlder and start it listening on port 4372 (TCP) as follows:

```
final ZKProtoServerHandler handler = new ZKProtoServerHandler();
final ZKProtoSyncService.Processor<ZKProtoSyncService.Iface>  processor =
        new ZKProtoSyncService.Processor<ZKProtoSyncService.Iface>(handler);
final TThreadPoolServer.Args serverArgs = new TThreadPoolServer.Args(
        new TServerSocket(4372));
serverArgs.processor(processor);
final TServer server = new TThreadPoolServer(serverArgs);
server.serve();
```

8. And that's all. Resuming, the final example class for server implementation looks as follows:

```
package com.zktechnology.zkproto;

import java.util.ArrayList;
import java.util.List;

import org.apache.thrift.TException;
import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TThreadPoolServer;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

class ZKProtoServerHandler implements ZKProtoSyncService.Iface {
        @Override
        public ZKProtoSyncAckInfo open(final ZKProtoSyncOpenInfo openInfo)
                        throws ZKProtoException, TException {
                System.out.println("Open called with argument: " + openInfo);
                return new ZKProtoSyncAckInfo();
        }

        @Override
        public void push(final List<ZKProtoSyncOperation> operations)
                        throws ZKProtoException, TException {
                System.out.println("Push called with argument: " + operations);
        }

        @Override
        public ZKProtoSyncOperations pull() throws ZKProtoException, TException {
                System.out.println("Pull called");
                return new ZKProtoSyncOperations(new ArrayList<ZKProtoSyncOperation>(), true);
        }
}

public class ZKProtoServer {

        public static void main(final String[] args) throws TTransportException {
```

```
                final ZKProtoServerHandler handler = new ZKProtoServerHandler();
                final ZKProtoSyncService.Processor<ZKProtoSyncService.Iface>  processor =
                                new ZKProtoSyncService.Processor<ZKProtoSyncService.Iface>(handler);
                final TThreadPoolServer.Args serverArgs = new TThreadPoolServer.Args(
                                new TServerSocket(4372));
                serverArgs.processor(processor);
                final TServer server = new TThreadPoolServer(serverArgs);
                System.out.println("Starting server");
                server.serve();

        }
}
```

And that's it. If you run the server you will see how it prints a “Starting server” message. Server is now listening on port 4372 in a separate thread. If you connect a client (the example client or a real client implementation, like ZPAD), you will see how the server outputs the calls being made by the client, for example:

```
Starting server
Open called with argument: ZKProtoSyncOpenInfo(protocolVersion:1, clientId:-6112037315054425508,
customId:SimpleJavaClient, maxOperations:2147483647, timestamp:1415378455589,
tableFilter:ZKProtoSyncTableFilter(filter:WHITE_LIST, tableList:[entity, entity_hierarchy, employee, attendance_log,
attendance_event]))
Pull called
Pull called
Pull called
```

Next step should be implementing the server logic on *open*, *pull* and *push* calls, but this part is out of the scope of the current document as there can be different logic models and business rules.

Appendix A: Apache Thrift

ZKProto uses Apache Thrift as the base middleware. We chose Thrift over other middleware solutions (SOAP, CORBA, Google's Protocol Buffers, etc...) because:

- It is licensed under Apache License.
- It has one of the best performances over Java in currently available middlewares.
- Supports several programming languages (C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa (Objective-C), JavaScript, Node.js, Smalltalk, OCaml, Delphi, and many others).
- Supports several transport methods (raw binary, JSON, HTTP...).
- Platform independent.
- It was developped by ex-Google engineers for Facebook, and currently Facebook uses it as middleware.

Please note this is how to install Apache Thrift at the time of writing this document. For latest Thrift and latest instructions, please visit Apache Thrift site. Also note that you need latest ZKProto Thrift interface definition (contact current ZKProto maintainer for access to this file). Take into account that there are 2 of these files: one for sync (*zkserver.thrift*), the other for control (*zkservercontrol.thrift*).

1. Download latest Thrift (0.9.1 as of this writing) source code from here (I suggest first check latest version here).
2. Compile with necessary language support following these instructions. Depending on the target language (Java, PHP, Ruby...), Thrift might generate an additional library to link in your project.
3. Generate the classes/data structures for your target language by feeding the Thrift interface definition to the *thrift* executable. For example:

   thrift -gen target zkserver.thrift

   where target is the target language. Supported targets (at the writing of this document) are: *cpp* (C++), *csharp* (C#), *d* (D), *delphi* (Delphi), *erl* (Erlang), *go* (Go), *hs* (Haskell), *java* (Java), *js* (JavaScript), *ocaml* (OCaml), *perl* (Perl), *php* (PHP), *py.tornado* (Python), *py.twisted* (Python), *py* (Python), *rb* (Ruby).

   Example: *thrift -gen java zkserver.thrift* will generate required language classes for server/client.
4. Use the generated classes (and library if any) to implement ZKProto server/client.

For more information about internal Apache Thrift workings, you can refer to the whitepaper (PDF).

Appendix B: FAQ (Frequently Asked Questions)

What is ZKProto?

ZKProto basically is a server-centralized real-time DB synchronization protocol with client grouping support. Each client keeps a copy of remote DB for local use, this means DB lookups are done in local. Server will keep all clients up-to-date so they have the exact same data for their current group. ZKProto is DB-engine and structure agnostic (it doesn't rely on clients running a given DB engine or a given database design).

What is Apache Thrift?

Apache Thrift is an Apache-licensed Facebook-sponsored middleware protocol built by ex-Google engineers. ZKProto uses Thrift as middleware for communications.

What transport protocol does ZKProto use?

ZKProto uses exclusively TCP as transport protocol. The underlying middleware (Apache Thrift) supports a variety of transport protocols (HTTP, HTTPS...), but all are based on TCP. You can also implement your own transport TCP-based protocol if you need to.

What DB engine should I use to make a ZKProto server/client implementation?

You can use any DB engine you in your server and/or client implementation. ZKProto is independent from any DB technology (for example, you can even use NoSQL if that fits your business model better).

What language can I use to make a ZKProto server/client implementation?

You can use any language supported by Apache Thrift. Most common programming languages are supported: C++, C#, Java, PHP, JavaScript, Objective-C, Ruby, Perl, Python and many others.

What tools do I need to make a ZKProto server/client implementation?

You only need the Apache Thrift compiler and ZKProto Thrift interfaces (although maybe some specific target language might require some external library). If you're going to compile the Thrift compiler from source code, you also need all required dependencies to compile.

Do I need any external library to make a ZKProto server/client?

No, generally there's no external libraries needed, although some languages (e.g. Java) might require an additional library (usually related Apache libraries).

Do I need to write my own server?

As of the writing of this document, ZKTeco offers a Java 7 implementation of the server part that os tested in both Windows and Linux. This server only supports PostGreSQL DB engine for now.

How do I add ZKProto support to my existing software?

Simply write a ZKProto client in your favorite language and integrate with the rest of your software.

How to implement a client that synchronizes into several zones?

You have 2 ways to do this:

1. Use one sync client for each zone (it can be several instances of same client implementation, multithreaded or not).
2. Use one sync client and use a control client to switch to the required zone.

Does ZKProto support compression?

Yes, ZKProto automatically compresses data once it is big enough to be compressed effectively (small data can grow bigger if compressed).

Does ZKProto support encryption?

Yes, ZKProto has optional encryption (enabled by default). ZKProto uses standard encryption algorithms RSA and AES. Only AES-256 is supported for now.

What happens to local data if I change a client's zone?

It's lost because server will erase the DB and refill with this zone's data.

Why there's a sync server and a control server?

Because they don't serve the same purpose: sync clients wants to know what changes happened to the DB. Control clients only care about ZKProto itself and not the business model. Also sync clients are usually not control clients.

Why are most table IDs very big and sometimes also negative?

Because operation IDs are 64-bit numbers generated randomly to avoid ID collisions in case clients are offline when this row is created. The probability of a collision using this system is one in $1.844674407 \times 10^{19}$ or 0.00000000000000001% for each table.

What does server message “Push called from non-replicated client, ignoring...” means?

This message means a client that is not replicated tried to push operations to the server. A non-replicated client is a client has not been successfully initialized.

Should my client open and close the socket after each pull call or keep it open?

Each method has its pros and cons, you should decide which one is better suited to your context:

- *Open and close socket*: costs more time and bandwidth due to to connection establishement: TCP handshake, OS has to allocate resources for the socket, launch a new process/thread, call ZKProto *open*, etc....
- *Keeping socket open*: costs memory and connections due to maintaining the socket resources. If socket is idle it has no bandwidth or CPU cost, but the memory is still reserved.

Some scenarios and our suggestions:

- Limited bandwidth: keep sockets open to avoid connection establishement overhead. An already-open TCP connection has no bandwidth cost at all.
- A lot clients: open and close sockets to free server memory and workload.
- Client pulling interval is short: keep sockets open to avoid opening/closing too much in too short time.

Why control server password is hashed using BCrypt and not a more common hash algorithm like MD5/SHA1?

First, why not MD5/SHA256/SHA1, etc...? These are all *general purpose* hash functions, designed to calculate a digest of huge amounts of data in as short a time as possible. This means they are *fast*, and that is precisely why they're bad: someone who stole your password DB using a modern server can try *every single possible MD5 password hash (assuming passwords are lowercase, alphanumeric, and 6 characters long)* in around **40 seconds**. Now why Bcrypt? Simply because it's slow. It introduces a *work factor*, which allows you to determine how expensive the hash function will be. Because of this, as computers get faster you can increase the work factor and the hash will get slower.

Appendix C: Authors

- Yassine Diouri (ياسين الديوري) - Software architecht, lead developer, tester
- Guillermo Merino – Technical design and support
- Luis Pulido – Technical design and support
- Gonzalo Vázquez – Technical design and support
- Tony Duan (段松林) – Tester
- Ricardo Li (李□□) - Tester