Horizon 2020

Innovative Tools for Cyber-Physical Energy Systems

IDSS - InnoCyPES Data Storage Service

User's Guide

By Massimo Cafaro, Italo Epicoco, Marco Pulimeno and Lunodzo Justine Mwinuka

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956433.

1 Executive summary

This document is related to the release of the InnoCyPES Data Storage Service, from now on called IDSS. It is submitted as Deliverable 2.3. In the InnoCyPES project organisation, Work Package (WP) 2 deals with data accessibility and usability; in particular, this document is concerned with Task 2.2, the implementation of a distributed data storage service. Therefore, this deliverable provides the details related to the release of IDSS. This document provides (i) the information required to compile and install IDSS, (ii) the tool's user's guide, (ii) the description and command-line usage of IDSS associated tools. Two versions of IDSS, based on different technologies and unique combination of peer to peer (P2P) overlay networks and programming languages, are introduced and discussed. The first focuses on distributed management of relational data, whilst the second is concerned with graph-based distributed data management. This document provides a guide on how to run servers and clients, and how to submit queries.

2 Table of Contents

1 Executive summary 1
2 Table of Contents 2
3 InnoCyPES Data Storage Service - IDSS 3
4 IDSS Relational Version 3
4.1 Obtaining, compiling and installing IDSS 3
4.1.1 Installing IDSS dependences 4
4.1.2 Compiling the software 4
4.2 Installing the software 4
4.3 Starting the software 4
4.4 Querying IDSS 8
4.4.1 Queries 8
4.5 Querying IDSS – Distributed query 9
4.6 Local IDSS queries 15
4.7 Local IDSS DELETE, UPDATE and INSERT queries 17
5 IDSS Graph-based Version 18
5.1 Obtaining, compiling and installing IDSS 21
5.1.1 Installing Dependencies 21
5.1.2 Compiling the software 21
5.2 Starting the software 21
5.2.1 Running the servers 21
5.2.2 Running the client 22
5.3 Querying IDSS 22
5.3.1 Types of Queries 22
5.3.2 Querying IDSS 23
5.4 Local IDSS Queries 28
5.4.1 Running local queries 28
5.4.2 Adding node 29
5.4.3 Updating a node 30
5.4.4 Deleting a node 31
6 Conclusion 32

3 InnoCyPES Data Storage Service - IDSS

IDSS is the InnoCyPES Data Storage Service. It is a novel distributed database architecture, designed and implemented to to maximise system throughput and availability, avoiding issues related to a centralised setting and providing support for complex distributed queries, by also allowing simple and efficient data integration. IDSS leverages a P2P (peer-to-peer) architecture and provides support for data handled by either RDBMS (i.e., data handled by a Relational Database Management System) or graph-based DBMS. Chapter 4 describes details related to relational version of the IDSS whilst Chapter 5 describes the graph based IDSS.

It is worth recalling here that IDSS must be considered as a research prototype, even though it can be deployed and used for research purposes. Currently, its Technology Readiness Level (TRL), i.e. the corresponding maturity level is 4 (technology validated in laboratory).

4 IDSS Relational Version

This release of the IDSS leverages a P2P architecture and provides support for relational data (i.e., data handled by a RDBMS – Relational Database Management Systems). Therefore, the data storage service allows dealing with multiple relational sources in a fully decentralised way, on the basis of a common relational schema.

The P2P support is provided by D1HT, a novel distributed one hop hash table which is able to maximize performance with reasonable maintenance traffic overhead even for huge and dynamic P2P systems. D1HT has reasonable maintenance bandwidth requirements even for very large systems, while presenting at least twice less bandwidth overhead than previous single hop distributed hash tables. Support for relational data is provided by SQLite, a small, fast, self-contained, high-reliability, full-featured, SQL database engine. IDSS is a Service Oriented Architecture (SOA), implemented through the use of Web Services (WS). In particular, IDSS leverages the gSOAP Toolkit to provide both the server and its clients, based on SOAP (Simple Object Access Protocol) and XML (eXtensible Markup Language). IDSS makes extensive use of threads, through the pthreads library (POSIX Threads), not only for serving concurrently multiple users, but also internally to manage correctly and as quickly as possible its own modules and functionalities.

4.1 Obtaining, compiling and installing IDSS

The software is freely available on the following GitHub repository: https://github.com/cafaro/IDSS

IDSS is meant to be compiled and installed on linux servers. In the following, we assume that the user is running on linux Ubuntu.

4.1.1 Installing IDSS dependences

Before compiling IDSS it is necessary to install its dependences as follows:

apt-get install libxml2-dev libssl-dev libdb-dev libsqlite3-dev zlib1g-dev lzma-dev

4.1.2 Compiling the software

Once the dependences are installed, IDSS can be compiled. In particular, IDSS requires using the clang C compiler and the clang++ C++ compiler; building requires using the cmake tool as follows:

cmake -S . -B build -DCMAKE_BUILD_TYPE="Release" -D CMAKE_C_COMPILER=clang -D CMAKE_CXX_COMPILER=clang++

It is worth noting here that it is possible to use -DCMAKE_INSTALL_PREFIX in order to change the installation directory, which is by default /usr/local.

Next, the following command must be used to compile IDSS:

cmake --build build -v

4.2 Installing the software

Once the software has been compiled, it can be installed as follows:

cmake --install build -v

To cleanup, simply remove the build directory:

rm -fr build

4.3 Starting the software

In the following, we assume that the IDSS server has been installed in /home/cafaro/idss-cmake-based/simulation/idss. The LD_LIBRARY_PATH must be updated as follows:

export LD_LIBRARY_PATH=/home/cafaro/idss-cmake-based/simulation/idss/lib:$LD_LIBRARY_PATH

Before starting the software:

1) the following configuration file

/home/cafaro/idss-cmake-based/simulation/idss/etc/idss_conf.xml

must be modified replacing “ubuntu” with the fully qualified hostname (or IP address) of the machine hosting the IDSS server. Note that this operation needs to be performed only once, before starting IDSS for the first time. In the following example, we substitute the 127.0.0.1 IP address:

sed -i -e "3s/ubuntu/127.0.0.1/" /home/cafaro/idss-cmake-based/simulation/idss-q/etc/idss_conf.xml

2) in order to enable TLS/SSL security, X509v3 digital certificates must be generated and copied to the corresponding IDSS installation directories. In particular, two certificates are required: one identifying the Certificate Authority (CA) used to issue the certificates and one identifying the IDSS server. Finally, a certificate must be issued to each user willing to use the IDSS server; these user certificates will be used to identify clients of the IDSS server. In the following example, we have three certificates: cacert.pem (CA certificate), server.pem (IDSS certificate) and client.pem (a user certificate). Note that the certificates must be created only the first time, and must be renewed before they expire, in order to allow correct authentication among clients and IDSS servers available in the P2P overlay network.

   OpenSSL can be used on both linux and macOS to create a CA and to issue the required certificates; in the following, we provide some examples.

Creating the Certificate Authority's Certificate and Keys

1. Generate a private key for the CA:
   openssl genrsa 2048 > ca-key.pem
2. Generate the X509 certificate for the CA:
   openssl req -new -x509 -nodes -days 365000 \
   -key ca-key.pem \
   -out cacert.pem

Creating the Server's Certificate and Keys

1. Generate the private key and certificate request:
   openssl req -newkey rsa:2048 -nodes -days 3650 \
   -keyout server-key.pem \
   -out server-req.pem

2. Generate the X509 certificate for the server:
   openssl x509 -req -days 365 -set_serial 01 \
   -in server-req.pem \
   -out server.pem \
   -CA cacert.pem \
   -CAkey ca-key.pem

Creating the Client's Certificate and Keys

1. Generate the private key and certificate request:

   *openssl req -newkey rsa:2048 -nodes -days 365 *

   *-keyout client-key.pem *

   -out client-req.pem

2. Generate the X509 certificate for the client:

   *openssl x509 -req -days 365-set_serial 01 *

   *-in client-req.pem *

   *-out client.pem *

   *-CA cacert.pem *

   -CAkey ca-key.pem

Verifying the Certificates

1. Verify the server certificate:

   *openssl verify -verbose -show_chain -CAfile cacert.pem *

   *cacert.pem *

   server.pem

2. Verify the client certificate:

   *openssl verify -verbose -show_chain -CAfile cacert.pem *

   *cacert.pem *

   client.pem

Alternatively, there are several applications that allow creating a CA and managing/issuing certificates. On macOS, one can use the system's application "Keychain Access". Alternatively, xca can be freely downloaded from https://hohnstaedt.de/xca/index.php and works on both linux and macOS. EJBCA (https://www.ejbca.org) is platform independent; the open source community edition can be downloaded from https://github.com/Keyfactor/ejbca-ce

Once created, the certificates must be made available to the IDSS server as follows:

cp ./server.pem /home/cafaro/idss-cmake-based/simulation/idss/server
cp ./cacert.pem /home/cafaro/idss-cmake-based/simulation/idss/server
cp ./client.pem /home/cafaro/idss-cmake-based/simulation/idss/bin
cp ./cacert.pem /home/cafaro/idss-cmake-based/simulation/idss/bin

3) In order to use the IDSS server, a SQL database must be prepared and installed on each IDSS server available in the P2P overlay network. This database must be based on a relational schema that must be compliant with SQLite. Assuming that the relational database schema is available in

the file /home/cafaro/idss-cmake-based/simulation/create_schema_p2p.sql, the corresponding database can be obtained as follows:

/usr/bin/sqlite3 /home/cafaro/idss-cmake-based/simulation/idss_db < /home/cafaro/idss-cmake-based/simulation/create_schema_p2p.sql

Once created, the database file must be put in place as follows:

cp /home/cafaro/idss-cmake-based/simulation/idss /home/cafaro/idss-cmake-based/simulation/idss/server/idss_db

4) The IDSS command-line arguments are the following:

-p <port number> sets the IDSS service port number
-f <file> sets the iDSS configuration file pathname
-d <level> sets the debug level (0-2)
-v prints the IDSS version
-e <file> sets the pathname of file where standard error will be redirected
-l <log file> sets the pathname of file where IDSS run-time information will be logged
-o <port number> sets the P2P overlay node port number
-a <IP address> specify the node IP address
-r <reduction factor> specify the reduction factor for TTL (Time To Live)
-j <host:port> sets the bootstrap node host and port

In order to start the IDSS server, the following considerations must be taken into account. An IDSS server can be used standalone, or as a peer belonging to an overlay network. In the former case, IDSS can be started as follows (assuming the IP address 192.168.1.10):

/home/cafaro/idss-cmake-based/simulation/idss/server/idss -p 19000 \
-a 192.168.1.10 \
-f /home/cafaro/idss-cmake-based/simulation/idss/etc/idss_conf.xml \
-d 2 \
-e /home/cafaro/idss-cmake-based/simulation/idss/server/stderr \
-l /home/cafaro/idss-cmake-based/simulation/idss/server/log \
-o 19001 \
-r 0.75

In the latter case, the first instance of IDSS being started shall play the role of the “overlay bootstrap node”, and the instances started after the bootstrap node shall join the overlay by contacting the bootstrap node. As a minimal example, consider an overlay with a bootstrap node and a single peer. The bootstrap node is started as shown before (i.e., like a standalone server). The peer node is started as follows (assuming it has been deployed on /usr/local/idss on a machine with IP address 150.190.1.15):

```
/usr/local/idss /server/idss -p 19000  \
  -a 150.190.1.15 \
  -f /usr/local/idss/etc/idss_conf.xml \
  -d 2 \
  -e /usr/local/idss/server/stderr \
  -l /usr/local/idss/server/log \
  -o 19001 \
  -r 0.75 \
  -j 192.168.1.10:19001
```

4.4 Querying IDSS

In the following we will make use of the example relational schema developed for testing purposes. The generic idss_test_client can be used in order to issue a distributed query. Its arguments are the following ones:

-h <hostname or IP address> (sets the IDSS hostname to be queried)

-p <port number> (sets the IDSS port number on which the IDSS server is listening for incoming connections)

-v (sets verbose mode)

-t <time to live, in seconds> (sets the maximum time allowed to process a query)

-q <SQL query> (sets the SQL query to be issued)

To use the IDSS client tools, update the LD_LIBRARY_PATH environmental variable:

export LD_LIBRARY_PATH=/home/cafaro/idss-cmake-based/simulation/idss/lib:$LD_LIBRARY_PATH

4.4.1 Queries

Here are some examples of different queries, related to the relational schema developed for testing purposes.

4.4.1.2 Simple queries

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -v -t 2 -q "SELECT * FROM tb_user;"*

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT * FROM tb_consumption;"*

4.4.1.3 Queries with aggregation operators

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT COUNT(user_id) FROM tb_user;

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT MIN(Power), MAX(Power), AVG(Power) FROM tb_user;"

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT MIN(measurement), MAX(measurement), AVG(measurement) FROM tb_consumption;"

4.4.1.4 Queries with where clause

*home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT * FROM tb_user WHERE Power > 3.5;"*

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT * FROM tb_consumption WHERE measurement >= 3.48;"*

4.4.1.5 Nested queries

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT * FROM tb_user WHERE user_id in (SELECT user_id FROM tb_consumption WHERE measurement >= 3.48);"*

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -t 2 -q "SELECT * FROM tb_consumption WHERE user_id in (SELECT user_id FROM tb_consumption WHERE measurement >= 3.48);"*

4.5 Querying IDSS – Distributed query

Here is an example. We start 100 IDSS servers (on the same local network), each storing 10 records, then we issue the following query:

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_test_client -h 127.0.0.1 -p 19000 -v -t 2 -q "SELECT * FROM tb_user;"*

The IDSS server provides the following reply:

Contacting the web service listening on... https://127.0.0.1:19000

Entering verify_callback...

The following certificate has been successfully verified:

issuer = /C=IT/ST=Italy/L=Lecce/O=InnoCyPES/OU=University of Salento/CN=InnoCyPES Certification Authority

subject = /C=IT/ST=Italy/L=Lecce/O=InnoCyPES/OU=University of Salento/CN=InnoCyPES Certification Authority

Exiting verify_callback...

Entering verify_callback...

The following certificate has been successfully verified:

issuer = /C=IT/ST=Italy/L=Lecce/O=InnoCyPES/OU=University of Salento/CN=InnoCyPES Certification Authority

subject = /C=IT/ST=Italy/L=Lecce/O=InnoCyPES/OU=University of Salento/CN=127.0.0.1

Exiting verify_callback...

UQI: 1740675925116283

A shown, both IDSS and the client perform mutual authentication using the issued X509V3 digital certificates. Upon successful certificate verification, the query is actually submitted from the client to the IDSS server, which replies with a UQI (Unique Query Identifier). The UQI can be used later, after the TTL has expired, to retrieve the query's output using the idss_get_file_client, whose arguments are:

-h <hostname or IP address> (sets the IDSS hostname to be queried)

-q <UQI> (sets the query UQI)

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_get_file_client -h 127.0.0.1 -q 1740675925116283

The IDSS server replies as follows:

Filename: 1740675925116283.xml

This file contains the actual records extracted from the P2P overlay network. As shown, 140 records have been retrieved from the P2P overlay network (<RECS N="140">) by allowing 2 seconds to complete the distributed query (-t 2):

```
<?xml version="1.0" encoding="UTF-8"?>
<RECORDSET>
  <SCH>
    <ATT TYPE="string">Name</ATT>
    <ATT TYPE="string">Surname</ATT>
    <ATT TYPE="integer">user_id</ATT>
    <ATT TYPE="integer">Contract_Number</ATT>
    <ATT TYPE="float">Power</ATT>
  </SCH>
  <RECS N="140">
    <R>
      <F>Michael</F>
      <F>Williams</F>
      <F>531</F>
      <F>470</F>
      <F>6.000000</F>
    </R>
    <R>
      <F>Anthony</F>
      <F>Singleton</F>
      <F>532</F>
      <F>469</F>
      <F>6.000000</F>
    </R>
    …
  </RECS>
</RECORDSET>
```

In general, by allowing more time to complete the distributed query we get back more records. For instance, issuing again the same query, this time using -t 5 we get back all of the 1000 records stored into the distributed overlay (only the first few records retrieved are shown):

```
<?xml version="1.0" encoding="UTF-8"?>
<RECORDSET>
 <SCH>
  <ATT TYPE="string">Name</ATT>
  <ATT TYPE="string">Surname</ATT>
  <ATT TYPE="integer">user_id</ATT>
  <ATT TYPE="integer">Contract_Number</ATT>
  <ATT TYPE="float">Power</ATT>
 </SCH>
 <RECS N="1000">
  <R>
   <F>Dawn</F>
   <F>Jones</F>
   <F>391</F>
   <F>610</F>
   <F>2.000000</F>
  </R>
  <R>
   <F>Kevin</F>
   <F>Lambert</F>
   <F>392</F>
   <F>609</F>
   <F>5.000000</F>
  </R>
…
</RECS>
</RECORDSET>
```

This behaviour is consistent with the IDSS specifications. A distributed query is performed “best effort” and strictly depends on the TTL (Time To Live) allowed. Indeed, assuming a very large, distributed collection of IDSS servers, each one storing a huge number of records, it is, in practice, impossible to retrieve all of the data. This would require, depending on the actual number of records, months or years. It is worth noting here that retrieving the data is only the beginning of the story. To be useful, those data would need to be read – requiring again months or years – and processed (the time for this is at least O(n) where n is the total number of records in the recordset, assuming the processing a record can be done in O(1) time).

However, assuming a manageable size of records stored within a server, IDSS can easily retrieve all of them in a feasible amount of time or provide a high-quality, best effort approximation of the data.

In the following example, we retrieve from the distributed overlay the minimum, maximum and average power by using -t 3 (SELECT MIN(Power), MAX(Power), AVG(Power) FROM tb_user;):

```
<?xml version="1.0" encoding="UTF-8"?>
<RECORDSET>
 <SCH>
  <ATT TYPE="float">MIN(Power)</ATT>
  <ATT TYPE="float">MAX(Power)</ATT>
  <ATT TYPE="float">AVG(Power)</ATT>
 </SCH>
 <RECS N="1">
  <R>
   <F>2.000000</F>
   <F>9.000000</F>
   <F>5.583000</F>
  </R>
 </RECS>
</RECORDSET>
```

Note that there is only 1 record retrieved, containing the required information. Next, we issue the same query by allowing -t 6 and we get the same results (not shown). This confirms that the results are related to the full set of records, since the results are unchanged. As shown, this kind of query is very fast: less information must be retrieved, and processed in a distributed fashion.

Next, we shown an example of the following nested query: “SELECT * FROM tb_user WHERE user_id in (SELECT user_id FROM tb_consumption WHERE measurement >= 3.48);” in which we use -t 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<RECORDSET>
 <SCH>
  <ATT TYPE="string">Name</ATT>
  <ATT TYPE="string">Surname</ATT>
  <ATT TYPE="integer">user_id</ATT>
  <ATT TYPE="integer">Contract_Number</ATT>
  <ATT TYPE="float">Power</ATT>
 </SCH>
 <RECS N="38">
  <R>
   <F>Eric</F>
```

<F>Martin</F>
<F>817</F>
<F>184</F>
<F>5.000000</F>
</R>
<R>
<F>David</F>
<F>Johnson</F>
<F>820</F>
<F>181</F>
<F>6.000000</F>
</R>
…
</RECS>
</RECORDSET>

The same query, this time using -t 6, retrieves 331 records instead of the previous 38.

<?xml version="1.0" encoding="UTF-8"?>
<RECORDSET>
<SCH>
<ATT TYPE="string">Name</ATT>
<ATT TYPE="string">Surname</ATT>
<ATT TYPE="integer">user_id</ATT>
<ATT TYPE="integer">Contract_Number</ATT>
<ATT TYPE="float">Power</ATT>
</SCH>
<RECS N="331">
<R>
<F>Megan</F>
<F>Williams</F>
<F>223</F>
<F>778</F>
<F>8.000000</F>
</R>
<R>

<F>Carol</F>

<F>Perry</F>

<F>225</F>

<F>776</F>

<F>8.000000</F>

</R>

…

</RECS>

</RECORDSET>

4.6 Local IDSS queries

In the following we will make use of the example relational schema developed for testing purposes. The generic idss_search_client_sql can be used in order to issue a local query (i.e., a query targeting only one specific IDSS server). Its arguments are the following ones:

-h <hostname or IP address> (sets the IDSS hostname to be queried)

-p <port number> (sets the IDSS port number on which the IDSS server is listening for incoming connections)

-v (sets verbose mode)

-q <SQL query> (sets the SQL query to be issued)

To use the IDSS client tools, update the LD_LIBRARY_PATH environmental variable:

export LD_LIBRARY_PATH=/home/cafaro/idss-cmake-based/simulation/idss/lib:$LD_LIBRARY_PATH

Here is an example:

*/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_search_client_sql -h 127.0.0.1 -p 19000 -v -q "SELECT * FROM tb_user;"*

<?xml version="1.0"?>

<Recordset xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://www.gridlab.org/WorkPackages/wp-10/Software/idss_recordset.xsd">

<Schema>

<field name="Name" id="0" type="text"/>

<field name="Surname" id="1" type="text"/>

```
  <field name="user_id" id="2" type="bigint"/>
  <field name="Contract_Number" id="3" type="bigint"/>
  <field name="Power" id="4" type="float"/>
</Schema>
<Records numrecord="10">
  <record field="0" value="Lauren"/>
  <record field="1" value="Patterson"/>
  <record field="2" value="1"/>
  <record field="3" value="100"/>
  <record field="4" value="2.000000"/>
  <record field="0" value="Gina"/>
  <record field="1" value="Ryan"/>
  <record field="2" value="2"/>
  <record field="3" value="99"/>
  <record field="4" value="6.000000"/>
  <record field="0" value="Natasha"/>
  <record field="1" value="Howard"/>
  <record field="2" value="3"/>
  <record field="3" value="98"/>
  <record field="4" value="2.000000"/>
  <record field="0" value="Glenn"/>
  <record field="1" value="Pratt"/>
  <record field="2" value="4"/>
  <record field="3" value="97"/>
  <record field="4" value="4.000000"/>
  <record field="0" value="Alyssa"/>
  <record field="1" value="Ramirez"/>
  <record field="2" value="5"/>
  <record field="3" value="96"/>
  <record field="4" value="5.000000"/>
  <record field="0" value="Deborah"/>
  <record field="1" value="Mack"/>
  <record field="2" value="6"/>
  <record field="3" value="95"/>
  <record field="4" value="8.000000"/>
  <record field="0" value="Robin"/>
```

```
    <record field="1" value="Parrish"/>
    <record field="2" value="7"/>
    <record field="3" value="94"/>
    <record field="4" value="5.000000"/>
    <record field="0" value="Tina"/>
    <record field="1" value="Ramirez"/>
    <record field="2" value="8"/>
    <record field="3" value="93"/>
    <record field="4" value="5.000000"/>
    <record field="0" value="Caleb"/>
    <record field="1" value="Porter"/>
    <record field="2" value="9"/>
    <record field="3" value="92"/>
    <record field="4" value="8.000000"/>
    <record field="0" value="Kimberly"/>
    <record field="1" value="Allen"/>
    <record field="2" value="10"/>
    <record field="3" value="91"/>
    <record field="4" value="2.000000"/>
  </Records>
</Recordset>
```


4.7 Local IDSS DELETE, UPDATE and INSERT queries

In the following we will make use of the example relational schema developed for testing purposes. The generic idss_query_test_client too can be used in order to issue a local DELETE, UPDATE and INSERT query. Its arguments are the following ones:

-h <hostname or IP address> (sets the IDSS hostname to be queried)

-p <port number> (sets the IDSS port number on which the IDSS server is listening for incoming connections)

-v (sets verbose mode)

-q <SQL query> (sets the SQL query to be issued)

To use the IDSS client tools, update the LD_LIBRARY_PATH environmental variable:

export LD_LIBRARY_PATH=/home/cafaro/idss-cmake-based/simulation/idss/lib:$LD_LIBRARY_PATH

Here are some examples.

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_query_test_client -h 127.0.0.1 -p 19000 -v -q "DELETE FROM tb_user WHERE Surname='Williams';"

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_query_test_client -h 127.0.0.1 -p 19000 -v -q "UPDATE tb_user SET Name='Kim' WHERE Surname='Black';"

/home/cafaro/idss-cmake-based/simulation/idss/bin/idss_query_test_client -h 127.0.0.1 -p 19000 -v -q "INSERT INTO tb_user (Name, Surname, user_id, Contract_number, Power) VALUES ('Gina', 'Davis', 372, 98367, 3.5);"

5 IDSS Graph-based Version

The graph-based IDSS maintains the original design of the IDSS, with slight differences in implementation that were mostly guided by the chosen technologies. This version utilises a structured P2P overlay network built using libp2p[1] (library peer-to-peer) to enable decentralised communication and data distribution. Libp2p is open-source full stack protocol for P2P systems that brings a simplified approach of implementing large-scale P2P systems that has always been complex and challenging for many years. It is built on top of InterPlanetary File System (IPFS)[2] giving new applications state-of-the-art P2P capabilities like flexibility to choose specific network features, interoperability, and enabling decentralised infrastructure even to non-distributed projects. With libp2p, each node in the system initialises as a libp2p host that generates a unique cryptographic identity and dynamically allocates network addresses (multiaddress) giving a unique identification of each peer in an overlay. A node's multiaddress[3] is formed by several segments which when concatenated, represent a self-describing hierarchical address format enabling peers to establish connections among each other. These addresses are defined using the following format **`/ip4/127.0.0.1/tcp/4000/p2p/QmXyz123Abc456…`**, described below:

- ***/ip4/127.0.0.1*** - This part specifies the network protocol (IPv4) and the IP address of the node. It tells you which protocol and corresponding address the node is reachable on. Other supported alternatives include (IPv6)/ip6, /dns/, /dns4/, or /dns6/, which allow to resolve hostnames to their respective IP addresses;
- ***/tcp/4000*** - This indicates that the node is using the TCP transport protocol and is listening on port 4000. The transport layer ensures reliable, ordered delivery of messages between peers. Other alternatives include UDP (/udp/), which is useful for lower latency but may be less reliable and QUIC (/quic/) that provide connection-oriented reliability on top of UDP. For browser or web-based applications, WebSockets (/ws/) and secure WebSockets (/wss/) are also supported;
- ***/p2p/QmXyz123Abc456…*** - The `/p2p` (formerly /ipfs in earlier versions) prefix identifies that the following string is the unique Peer ID of the node. This identifier is generated using cryptographic keys and distinguishes the node within the P2P overlay network.

[1] https://libp2p.io/
[2] https://ipfs.tech/
[3] A Multiaddress (multiaddr) in LibP2P is a self-describing network address format that allows applications to define how to reach a given node using various transport and protocol combinations.

The overlay leverages the Kademlia DHT[4] (a libp2p's custom DHT implementation) for efficient node discovery and content routing. The IDSS leverages DHT to facilitate efficient key-based routing and peer discovery, with additional local discovery enabled by multicast DNS (mDNS). This provides IDSS with the ability to run both locally (using local and custom bootstrap nodes) and globally (using default bootstrap peers offered by libp2p). To control propagation of messages in an overlay, libp2p supports an addition of a specific protocol prefix in the protocol ID. The IDSS applies its own "`/idss`" prefix. For instance, the IDSS's protocol used across an overlay becomes **`/idss/kad/1.0.0`** for global connection and **`/idss/lan/kad/1.0.0`** for local connection instead of the generalised protocol IDs **`/kad/1.0.0`** and **`/lan/kad/1.0.0`**. This approach allows for dynamic peer connectivity and controlled population of the DHT. In this regard, only peers that recognise and support this protocol, can handle the messages. This avoids conflicts with other protocols operating over the same network, helps in organising and routing messages correctly, and facilitates future upgrades or changes without interfering with other network communications. Hence, key libp2p features implemented in the IDSS can be summarised as follows:

- *Multiaddress support and peer identification* - each node is assigned a unique cryptographic identity during initialisation, ensuring that every peer can be uniquely and securely recognised within the network. This is accompanied by a self-describing multiaddress format allowing identification of various network protocols;
- *Secure communication* - libp2p integrates both the TLS and Noise security protocols to encrypt communications between peers. This guarantees confidentiality and integrity of the transmitted data;
- *Stream multiplexing* - multiple logical streams can be carried over a single physical connection. This feature allows concurrent communication channels between peers, improving overall network efficiency;
- *Local peer discovery* - mDNS enables automatic discovery of peers on local networks without the need for centralised DNS servers, which facilitates rapid bootstrapping of nodes. Additionally, support is also available for the bootstrap and random walk protocols;
- *Efficient peer discovery and bootstrapping* - the Kademlia DHT facilitates finding and connecting to other peers through a decentralised routing mechanism. It supports bootstrapping through a list of known peers and continuously discovers new joining nodes;
- *Key-based routing* - messages are routed based on keys in a deterministic manner. This ensures that any message addressed to a given key is incrementally forwarded toward the responsible node;
- *Structured overlay* - by organising peers into a structured overlay, the DHT supports scalable and efficient lookups. The routing table is maintained in a way that balances load and minimises latency;
- *Periodic Routing Table refresh* - the DHT periodically refreshes its routing table to account for peer churning (nodes joining or leaving), which ensures that the routing information remains current and reliable.

The IDSS integrates EliasDB[5], a graph-based database to manage data and query information. EliasDB is a lightweight solution for projects having data that demands graph structure and provide most functionalities without the need for a third-party application. It is a disk-based graph database that is employed for storing both application data and metadata related to queries. This approach facilitates the representation of complex relationships by modelling data as nodes and edges. EliasDB defines a storage file in a disk which can contain a fixed size of records, each with a unique identifier. On disk the logical Storage File is split into multiple files each with a maximum file size of 10 Gigabyte.

Like most graph databases in the domain, EliasDB stores data as nodes, maps relationships through edges, and can be sharded using partitions. Data can be stored and retrieved using the GraphQL[6]

[4] https://pl-launchpad.io/curriculum/libp2p/dht/
[5] https://github.com/krotik/eliasdb
[6] https://github.com/krotik/eliasdb/blob/master/graphql.md

interface and EliasDB's own query language, EQL[7], for more complex queries with an SQL-like syntax. For robust applications, the embedded EliasDB supports transactions with rollbacks, iteration of data and rule-based consistency management. To benefit from the rich set of features offered by EliasDB, IDSS utilised EQL for data querying while utilising specific functions of ECAL[8] for node/edge storage, update, removal and retrieval.

The graph database is initialised during peer startup (with its sample data generated via an external Python script following a predefined graph data structure defined in a **`json`** file). The format allows for easy mapping of nodes and edges with their associated properties. The examples in this report will follow a snapshot of data schema described in Figure 1. During program execution, the number of nodes and edges can be defined in the python script.

```
{
    "nodes": [
        {
            "kind": "Client",
            "key": 1,
            "name": "William Rich",
            "contract_number": 1186887,
            "power": 22
        },
{
            "kind": "Consumption",
            "key": 1,
            "timestamp": 1567413200.7773664,
            "measurement": 619219
        }
],
"edges": [
        {
            "key": "e1",
            "kind": "belongs_to",
            "end1key": 1,
            "end1kind": "Client",
            "end1role": "owner",
            "end1cascading": true,
            "end2key": 14,
            "end2kind": "Consumption",
            "end2role": "usage",
            "end2cascading": true
        },
{
            "key": "e100",
            "kind": "belongs_to",
            "end1key": 10,
            "end1kind": "Client",
            "end1role": "owner",
            "end1cascading": false,
            "end2key": 60,
            "end2kind": "Consumption",
            "end2role": "usage",
            "end2cascading": true
        }
    ]
}
```

Figure 1: Graph nodes and edges structure

[7] https://github.com/krotik/eliasdb/blob/master/eql.md
[8] https://github.com/krotik/eliasdb/blob/master/ecal.md

5.1 Obtaining, compiling and installing IDSS

The software is freely available on the following GitHub repository: https://github.com/cafaro/IDSS

The current release of the graph-based IDSS is designed to compile and run in Linux environments with golang installed in them. In the rest of this document, we assume the use of an Ubuntu based linux system.

5.1.1 Installing Dependencies

To successfully run IDSS, the following must be installed:

- Golang 1.23.1 or above is recommended (IDSS code base);
- Python 3 to automate the generation of synthetic data;
- Git to enable initial cloning of the project. This is not needed if the project is directly downloaded from other sources.
-

To install these packages, run

```
sudo apt install golang git python3
```

5.1.2 Compiling the software

Before compiling ensure that all Go modules required to run IDSS are installed by running the following command in the IDSS root directory:

```
go mod tidy
```

The IDSS is compiled using the command

```
go build -o idss_server
```

5.2 Starting the software

5.2.1 Running the servers

To run IDSS servers navigate to the server repository in the IDSS repository. Each peer requires a separate terminal to launch. To form an overlay, you need to launch at least two peer nodes. For a better impression of decentralised data management, distributed queries, DHT population and other lipp2p functionalities, you may need to launch at least 10 or more peer nodes in an overlay. We have made this simple by preparing a bash script (start_peers.sh) which will automate the process of launching peers and

data generation. Each peer operates as an independent daemon, maintaining its own graph database instance and communicating securely over the P2P overlay.

- To run individual peers, run the main package with **"`go run .`"** command or use compiled code **`./idss_server`** (each in a separate terminal);
- To automate peer launching, run the bash script with the command **"`./start_peers.sh <number_of_peers>`**". We recommend this approach for a simulation.

After launch, each node will generate its own directory for graph data in **`idss_graph_db`** directory and will generate its own log file in the **`log`** directory, with names following peer numbers, i.e., *peer1, peer2*, …, *peerN*.

Sample graph nodes and edges based on synthetic data are generated using **`generate_data.py`** and the data model is mapped by the generated **`.json`** file (as shown by Figure 1) during peer launching.

Note: Each peer server prints its multiaddress in its specific terminal to allow you identifying and connecting to them. In case you run the servers with an automated script, the multiaddress will be printed in a specific log file to give you the freedom to choose which server you would want to connect to and run the queries. With the automated script, we print the multiaddress of the first server to be launched on the terminal to which the script was initiated. But still, the rest of details can be found in the appropriate log file.

5.2.2 Running the client

All queries are submitted by the client. To launch a client, navigate to the "client" repository and run the command indicating the peer multiaddress of one of the running servers (found in server logs):

```
go run . -s <server_address>
```

A client should be able to connect. Proceed with query submission. Use sample queries described in section 5.3.2. A query must be in the following format:

```
get Client, 3 // This will run for 3 seconds. A number after the comma indicates the TTL
```

5.3 Querying IDSS

The implementation of data querying in the IDSS follows the features supported by EQL, which provides an SQL-like syntax allowing for smooth data retrieval from the graph. This section describes the supported types of queries and demonstrate actual query submission and results retrieval:

5.3.1 Types of Queries

5.3.1.1 Simple Queries

The IDSS implements all basic queries supported by EQL's syntax. Since their retrieval patterns involves fetching nodes under conditions like "**`=,!=, <, >, contains,`** arithmetic operators**`, like`**" and others, running such queries in distributed environments, requires data aggregation methods among peers.

Example queries may include:

- `get Client`
- `get Consumption traverse ::: where name = "Alice"`
- `lookup Client '3' traverse :::`
- `get Client traverse owner:belongs_to:usage:Consumption`
- `get Client show name`

The traversal expressions indicated by ":::" provides a way to traverse the edges to fetch nodes and edges that are connected to each other (an equivalence to relationships in relational databases). The traversal statements may be followed by conditional statements to filter data nodes/edges that comply to a condition given:

- `get Client traverse owner:belongs_to:usage:Consumption where measurement >= 300 and measurement <= 800`

The EQL comes with the built in `COUNT` function, which can be useful in data querying. For instance, the `COUNT` function can be used to fetch nodes which have a certain number of connections (relationships). In this regard when the count number is set to greater than zero (0), the query would will all the nodes in the graph which are connected to each other and have at least one connection, so any connected node will be returned. Possible queries can include:

- `get Client where @count(owner:belongs_to:usage:Consumption) > 2`
- `get Client where @count(owner:belongs_to:usage:Consumption) > 4`
- `get Consumption where @count(:::) > 0`
- `get Client where Power > 150 and @count(owner:belongs_to:usage:Consumption) > 5`

5.3.1.2 Distributed Data Querying

The IDSS supports distributed sorting of results using built-in EQL capabilities. To sort data EQL uses the `WITH` clause, followed by the "`ordering`" command, for which a query has to specify an `ascending` or `descending` directive.

- `get Client with ordering(ascending Client:name)`
- `get Client show name with ordering(ascending name)`

5.3.2 Querying IDSS

We start IDSS with 50 peers, each having 10 clients, 100 consumption records, and 5000 edges (each client having 100 edges - direct connection to consumption record). IDSS can be started using the prepared bash script called start_peers.sh as follows

```
./start_peers.sh 50

Build completed. The binary is named idss_server.
Peer 1 is running at address:
/ip4/127.0.0.1/tcp/40847/p2p/QmeVHAZvpEbTBJj4TsPf9znJp2WvuF7g4U666v8EDXAmTk
Peer 1 Launched with ID QmR6bdzNrUYojZMF3Qja9AC1FN8cz3kBdkDzrco3iJ12J2
Peer 2 Launched with ID QmXbTUaWwHVfapSFnbTRyLwLw6zKPNrZNpm8CFba8UM713
Peer 3 Launched with ID QmQy9uPgXdDRNQ6UVtv5br1oHumRD8HFydZRYgX54ZB6bK
Peer 4 Launched with ID QmTjLH88hc7yuZstv7YDVL7SWC8qtDadV9cYfoLNdtQMGx
Peer 5 Launched with ID QmSKSQ4kSxPsTmnaRrYTB6m59Ln5ctFURk3Wykq44kwsXG
```

```
…
Peer 48 Launched with ID QmUc61qpCqS2FsiKkh7ShmpjSten6ga6zXADfBVcKxa5cb
Peer 49 Launched with ID QmTsRSdDgAHSES3DCjTKdtB8A47FuGuVaQ6nT3eedzP3sp
Peer 50 Launched with ID QmT5oTnd5gQdxWubnDqJrbtMU2HDYe9JpgLtWbqaZWWQMC
Peers have joined the overlay.
```

In another client terminal (it should be in the ~idss_graphdb/client directory) A client connects to the given address, and the following feedback is given:

```
go run . -s /ip4/127.0.0.1/tcp/40847/p2p/QmeVHAZvpEbTBJj4TsPf9znJp2WvuF7g4U666v8EDXAmTk
INFO[0000] Client listening on:
/ip4/127.0.0.1/tcp/38095/p2p/QmYuB1woqVedJBiEvHTCPgieemMv1XJfbW1petQ8JGichW
INFO[0000] Client is discovering and connecting to server...
INFO[0000] Connected to server: QmeVHAZvpEbTBJj4TsPf9znJp2WvuF7g4U666v8EDXAmTk
INFO[0000] Protocol: /lan/kad/1.0.0
Enter your query and TTL or 'exit' to quit:
Use comma to separate query and TTL
->
```

We launch a query to fetch Client nodes, with TTL 3

```
get Client, 3
```

IDSS Response logs show as follows

```
-> get Client, 3
INFO[0015] Stream opened.
INFO[0015] Query sent to server.
INFO[0015] UQI:  QmYuB1woqVedJBiEvHTCPgieemMv1XJfbW1petQ8JGichW-1741614084225714275
INFO[0018]
----------- Response -----------
INFO[0018] Got 330 records
INFO[0018] Time spent: 2.61577838s
INFO[0018] Response saved to file: results/QmYuB1woqVedJBiEvHTCPgieemMv1XJfbW1petQ8JGichW-
1741614084225714275.xml
INFO[0018] Results will be deleted when the client exits.
```

QmYuB1woqVedJBiEvHTCPgieemMv1XJfbW1petQ8JGichW-1741614084225714275.xml content

```
<response>
    <resultCount>330</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power</data>
    </result>
    <result>
        <data>10,7.978654e+06,Gina Nguyen,1581</data>
    </result>
    <result>
        <data>4,7.34502e+06,Matthew Nixon,841</data>
    </result>
    <result>
        <data>5,4.279658e+06,Shannon Edwards,1343</data>
    </result>
    <result>
        <data>3,4.635064e+06,Charles Rivera,578</data>
    </result>
    …
    <result>
        <data>4,6.694122e+06,Amanda Adams,1571</data>
    </result>
</response>
```

We then increase the TTL to 6

```
-> get Client, 6
INFO[0175] Stream opened.
INFO[0175] Query sent to server.
INFO[0175] UQI:  Qmd5rdCmhY1JeGHnrqQxnet5m3LQ9PEHzb4JH6VTsAbV3C-1741616244185203056
INFO[0180]
----------- Response -----------
INFO[0180] Got 500 records
INFO[0180] Time spent: 4.594162527s
INFO[0180] Response saved to file:
results/Qmd5rdCmhY1JeGHnrqQxnet5m3LQ9PEHzb4JH6VTsAbV3C-1741616244185203056.xml
INFO[0180] Results will be deleted when the client exits.
```

Qmd5rdCmhY1JeGHnrqQxnet5m3LQ9PEHzb4JH6VTsAbV3C-1741616244185203056.xml content:

```
<response>
    <resultCount>500</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power</data>
    </result>
    <result>
        <data>10,6.736268e+06,Thomas Lewis,366</data>
    </result>
    <result>
        <data>3,3.893242e+06,Jessica Sanchez,36</data>
    </result>
    <result>
        <data>6,990816,Sara Stein,201</data>
    </result>
    <result>
        <data>9,9.618244e+06,Kenneth Sheppard,1623</data>
    </result>
    <result>
        <data>1,1.486434e+06,Anne Richard,47</data>
    </result>
    ...
    <result>
        <data>3,4.772766e+06,Darius Wallace,1032</data>
    </result>
</response>
```

Now all records were fetched. As explained in the relational IDSS, fetching of results is based on “best effort”, considering a potentially huge number of distributed servers. It should be noted that querying can also be affected by several other factors including network latency, so querying with TTL 3 for instance, will not always return the same number of records.

Query traversal can also be run in the IDSS specific to search targeted records. For instance, the following query traversal retrieves all client records with their associated consumption records. i.e., these are two nodes connected through an edge with a connection pattern described as *owner:belongs_to:usage:Consumption*

get Client traverse owner:belongs_to:usage:Consumption, 7

Results

```
<response>
    <resultCount>500</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power,Consumption Key,Measurement,Timestamp</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65,70,488659,6.036404018596651e+07</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65,58,7.004064e+06,5.168698156886269e+08</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65,96,2.978145e+06,1.076991537583794e+09</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65,25,4.919746e+06,4.517733891643259e+08</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65,67,2.352883e+06,7.513685369231331e+07</data>
    </result>
    …
    <result>
        <data>9,9.124043e+06,Michael Ward,235,12,2.429503e+06,6.888155371613868e+08</data>
    </result>
    <result>
        <data>9,9.124043e+06,Michael Ward,235,72,4.930209e+06,1.688180374202979 8e+09</data>
    </result>
</response>
```

IDSS can run queries with the COUNT function. The following query returns client records having more than 4 consumption records

```
get Client where @count(owner:belongs_to:usage:Consumption) > 4
```

Results

```
<response>
    <resultCount>200</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power</data>
    </result>
    <result>
        <data>10,9.538066e+06,Janice Nelson,65</data>
    </result>
    <result>
        <data>7,2.136687e+06,Troy Howard,1727</data>
    </result>
    <result>
        <data>5,1.61167e+06,Sandra Larson,599</data>
    </result>
    <result>
        <data>9,5.587508e+06,Nicholas Tran,1773</data>
    </result>
    …
    <result>
        <data>1,4.534641e+06,Clayton Barnes,1568</data>
    </result>
    <result>
        <data>4,6.7361e+06,Lindsey Baker,1488</data>
    </result>
</response>
```

You may recall that each client has 10 consumption records, hence, running the following query should return zero records

```
get Client where @count(owner:belongs_to:usage:Consumption) > 10, 7
```

Results

```
<response>
    <resultCount>0</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power</data>
    </result>
</response>
```

We can also use a `WITH` clause, to sort distributed data in either ascending or descending order. The following query sorts the client records in ascending order using a “name” property.

```
get Client with ordering(ascending Client:name), 7
```

Results:

```
<response>
    <resultCount>500</resultCount>
    <result>
        <data>Client Key,Contract Number,Client Name,Power</data>
    </result>
    <result>
        <data>10,583870,Adam Miller,1529</data>
    </result>
    <result>
        <data>3,9.990314e+06,Aimee Griffith,517</data>
    </result>
    <result>
        <data>7,8.592712e+06,Alan Gray,1226</data>
    </result>
    <result>
        <data>6,1.464627e+06,Albert Singh,268</data>
    </result>
    <result>
        <data>9,2.066003e+06,Alec Lloyd,1433</data>
    </result>
    …
   <result>
        <data>6,2.728136e+06,Zachary Barajas,1179</data>
    </result>
    <result>
        <data>7,9.54653e+06,Zachary Dominguez,942</data>
    </result>
    <result>
        <data>8,5.825457e+06,Zachary Meyers,485</data>
    </result>
    <result>
        <data>6,1.194729e+06,Zachary Moreno,511</data>
    </result>
</response>
```

5.4 Local IDSS Queries

IDSS also supports running local queries - queries that are not meant to be broadcasted in an overlay. This allows fetching data exclusively from an individual peer, without necessarily fetching data from other peers. Also, all operations that require modifications of the graph data are limited to local data repositories, because it would not be reasonable to allow a peer to modify other peers' data items.

5.4.1 Running local queries

To run local queries a client has to submit a query with either *-l* or *-local flag*. With the current release, we do not necessarily need to submit a TTL since the TTL is meant to control query propagation and dictating how long a client can wait before getting the results. Here are an example query and client logs:

```
-> get Client -l
INFO[0384] Stream opened.
INFO[0384] Query sent to server.
INFO[0384] UQI:  QmaJzD85QH18P3a4DAYoZENjctosbr9E2F9j5G93kizui3-1742315611285564800
INFO[0384]
----------- Response -----------
INFO[0384] Got 10 records
INFO[0384] Time spent: 26.559245ms
INFO[0384] Response saved to file:
results/QmaJzD85QH18P3a4DAYoZENjctosbr9E2F9j5G93kizui3-1742315611285564800.xml
```

Results

```
<response>
    <resultCount>10</resultCount>
    <result>
        <data>Client Key,Client Name,Contract Number,Power</data>
    </result>
    <result>
        <data>10,Tonya Giles,2643,1706</data>
    </result>
    <result>
        <data>5,Alexandra Li,1021,1133</data>
    </result>
    <result>
        <data>8,Duane Russell,4169,380</data>
    </result>
    <result>
        <data>3,Julia Wright,9689,903</data>
    </result>
    <result>
        <data>7,Taylor Benton,3943,1726</data>
    </result>
    <result>
        <data>1,Austin Alvarez,2738,1770</data>
    </result>
    <result>
        <data>4,Jessica Allen,4769,7</data>
    </result>
    <result>
        <data>2,Mr. David Garrett,6347,302</data>
    </result>
```

```
    <result>
        <data>9,Edward Thomas,5361,1136</data>
    </result>
    <result>
        <data>6,Tamara Phillips,2521,1660</data>
    </result>
</response>
```

EQL does not inherently support data modification queries, but *EliasDB* has built-in ECAL functions to do data modifications which can be run as *graph_manager* operation or *transactions*. These functions are `StoreNode()` for adding a node, `RemoveNode()` for removing a node, and `UpdateNode()` for updating an existing node. The following subsections demonstrate how to modify graph data in IDSS.

5.4.2 Adding node

The IDSS uses a keyword "`add`" to refer to a query that requires an addition of a node. The query must adhere to the "`add <kind> <key> [attributes]`" syntax. This can be run as follows:

```
add Client 14 client_name="Lunodzo Mwinuka" contract_number=7437643 power=7575
```

Now, if we run a query to fetch local results, we get the following:-

```
<response>
    <resultCount>11</resultCount>
    <result>
        <data>Client Key,Client Name,Contract Number,Power</data>
    </result>
    <result>
        <data>10,Tonya Giles,2643,1706</data>
    </result>
    <result>
        <data>14,Lunodzo Mwinuka,7437643,7575</data>
    </result>
    <result>
        <data>5,Alexandra Li,1021,1133</data>
    </result>
    <result>
        <data>8,Duane Russell,4169,380</data>
    </result>
    <result>
        <data>3,Julia Wright,9689,903</data>
    </result>
    <result>
        <data>7,Taylor Benton,3943,1726</data>
    </result>
    <result>
        <data>1,Austin Alvarez,2738,1770</data>
    </result>
    <result>
```

```
        <data>4,Jessica Allen,4769,7</data>
    </result>
    <result>
        <data>2,Mr. David Garrett,6347,302</data>
    </result>
    <result>
        <data>9,Edward Thomas,5361,1136</data>
    </result>
    <result>
        <data>6,Tamara Phillips,2521,1660</data>
    </result>
</response>
```

5.4.3 Updating a node

The IDSS uses the "update" keyword to change the values of an attribute/property in a graph. In this example, we want to update the *Client 14*, to a different name. So, we run the following query:

```
update Client 14 client_name="Massimo Mwinuka"
```

Fetching the data now we get the following results:

```
<response>
    <resultCount>11</resultCount>
    <result>
        <data>Client Key,Client Name,Contract Number,Power</data>
    </result>
    <result>
        <data>10,Tonya Giles,2643,1706</data>
    </result>
    <result>
        <data>14,Massimo Mwinuka,7437643,7575</data>
    </result>
    <result>
        <data>5,Alexandra Li,1021,1133</data>
    </result>
    <result>
        <data>8,Duane Russell,4169,380</data>
    </result>
    <result>
        <data>3,Julia Wright,9689,903</data>
    </result>
    <result>
        <data>7,Taylor Benton,3943,1726</data>
    </result>
    <result>
        <data>1,Austin Alvarez,2738,1770</data>
    </result>
    <result>
        <data>4,Jessica Allen,4769,7</data>
    </result>
    <result>
        <data>2,Mr. David Garrett,6347,302</data>
    </result>
    <result>
        <data>9,Edward Thomas,5361,1136</data>
```

```
    </result>
    <result>
        <data>6,Tamara Phillips,2521,1660</data>
    </result>
</response>
```

5.4.4 Deleting a node

Similarly, to delete a graph node, a client must use a "`delete`" keyword, followed by a graph *kind* and *key* in the following format: "`delete <kind> <key>`". In this example, we delete some data that we have just added, so, the query is formulated as follows:

```
delete Client 14
```

Now, the new results after deleting are the following:

```
<response>
    <resultCount>10</resultCount>
    <result>
        <data>Client Key,Client Name,Contract Number,Power</data>
    </result>
    <result>
        <data>10,Tonya Giles,2643,1706</data>
    </result>
    <result>
        <data>5,Alexandra Li,1021,1133</data>
    </result>
    <result>
        <data>8,Duane Russell,4169,380</data>
    </result>
    <result>
        <data>3,Julia Wright,9689,903</data>
    </result>
    <result>
        <data>7,Taylor Benton,3943,1726</data>
    </result>
    <result>
        <data>1,Austin Alvarez,2738,1770</data>
    </result>
    <result>
        <data>4,Jessica Allen,4769,7</data>
    </result>
    <result>
        <data>2,Mr. David Garrett,6347,302</data>
    </result>
    <result>
        <data>9,Edward Thomas,5361,1136</data>
    </result>
    <result>
        <data>6,Tamara Phillips,2521,1660</data>
    </result>
</response>
```

Note: In adding and updating graph node, queries may include attributes/properties that were not initially available in the graph data. This will imply that on fetching the whole dataset, some graph node will be *nil* in some of its properties, unless data is universally updated.

6 Conclusion

This document describes two different releases of the IDSS. We have implemented a relational database version of IDSS with D1HT as the supporting P2P overlay. The programming language of this version is C/C++ with GSOAP Toolkit to provide web services based protocols. This version relies on the use of the SQLite embedded database engine. The graph-based IDSS uses libp2p's Kademlia DHT as the supporting P2P overlay. The programming language of this version is Go, with EliasDB as an embedded graph database engine. On the basis of a quick comparison between the two releases, the IDSS relational version performs way faster than its graph-based counterpart. This is because of two major reasons, first, the relational version is implemented using the C/C++ programming languages which are inherently faster than Go. Second, the relational version used D1HT for lookup, query broadcast and results merging, whilst a graph based IDSS uses Kademlia DHT. The two uses different mechanisms for lookup, theoretically contributing highly to performance variation. D1HT is a single-hop DHT, in the sense that each node maintains complete routing information, hence enabling direct communication to any other peer with just one-hop. This makes D1HT achieve lookup with extremely low lookup latency; however it incurs high overhead and limited scalability in extremely large networks. In contrast, Kademlia DHT, is a multi-hop DHT protocol that organises nodes using an XOR-based metric in a binary-tree-like structure. Nodes recursively route queries though several intermediate peers (multi-hop) to reach destination node, resulting to logarithmic complexity for lookups. However, it can scale efficiently and remains robust in a large, dynamic environment, under the expense of high latency compared to D1HT. Thus, D1HT is preferable for small networks where ultra-low latency is critical, whereas Kademlia suits larger, more dynamic networks prioritising scalability and resilience over raw lookup speed.

On the other hand, whilst the relational IDSS offers support for complex SQL queries, the query formulation and the approach to fetch and aggregate data in a distributed manner is a complex task. The graph-based IDSS follows a simple syntax based on the EQL language which basically fetches and aggregates data easily when compared to its relational counterpart. The choice between the two would vary greatly based on data requirements of stakeholders.