DirectDemocracyP2P Architecture

Marius C. Silaghi, Khalid Alhamed, Osamah Dhanoon, Hang Dong,
Song Qin, Rahul Vishen, Ryan Knowles, Yi Yang, Yog Lok Seo
... and hopefully you!

March 11, 2015

Chapter 1

Introduction

1.1 This document

The latest version of this document is on `dev/MSilaghi/Doc/GUIDE.pdf` in my SVN, or on the `https://github.com/ddp2p/DDP2P/tree/master/DOC/GUIDE.pdf` (from now on this site is simply referred to as the `github`).

1.2 Release History

The technical details and intentions of this project were first lined out in 2004 in the FIT technical report `CS-2004-07`, that had also been submitted in 2003 as an NSF funding request. The funding was not grated, and after repeated requests to different governmental agencies (including a 2005 EU proposal with Markus Zanker from Austria and 10 EU institutions), in 2007 we started to develop the system as a volunteer work. A first version of the system was developed as a web-application prototype (`http://debatedecide.fit.edu`). It was not P2P and its intentions was a proof of concept, and potential demo for further funding requests. By 2009 it was nevertheless pretty complete and is still used to manage some classes at FIT. Its weaknesses are:

- difficulties with the java applets plugins frequently not well supported by browsers (but needed to create and handle certificates).
- the system administrator has too much power as he can apply censorship and delete items.

In 2010 we started the P2P implementation of the system. First we still thought of a P2P prototype, in python. After a few months developing an interface with python, we felt that it was less easy to build and maintain then an actual full system in Java.

The actual implementation of the open-source P2P Java program started in spring 2011 with the ASN1 packages and STUN-like NAT piercing mechanism. The GUI development started in summer 2011, with the tree view for

constituents and neighborhoods (which helped revise the final structure of the corresponding objects. Song also started working that summer on the Census panel. The swing GUI centered on developing autonomous widgets for various functions. Their assembly into an ergonomic GUI was left for later (and not yet tackled by 2014). Currently widgets are just chained in a pretty long tab.... (apologize to those that do not like it, as you are welcome to change it to an ergonomic GUI). The year 2012 was spent designing and implementing the various objects, integrating an ad-hoc network module (with Osamah Dhanoonn's master thesis), and integrating an automatic update mechanism (with the PhD work of Khalid Alhamed).

The first public release of the system was in summer 2013, at the *Open Peer-Reviewed Workshop on Decentralized Coordination* that we organized at Florid Tech in April 7. There we presented the articles detailing the technical and scientific contributions of the first version. A demo was made at the P2P 2013 conference. The version 1 is pretty stable but slow, since the only synchronization point of the various modules is the database on the disk. The database access is slow, which slows down the whole system. This is why in September 2013 we started working on a new version. The version 1 was left untouched except for a few very minor fixes (versions 0.9.49 to 0.9.55), some proving more bothering then worth. The version is available on github in the `src` folder.

While the second version (which uses a cache of objects as point of synchronization) is pretty finalized, there are still a couple of bugs (see the Bugs section), and was not yet released as an installation package until now (Oct 2014). You can download the code from the `src_version2` folder and compile it yourself.

Chapter 2

Development

2.1 Generating New Releases

To generate a new release (or a new tester recommendation) first one needs a secret key that I typically store in a file "Trusted64.sk". This file consists of two lines: the first line contains the secret key and the second line contains the public key. One can generate such a file from the software with the method **DD**.`createTrustedSKFile()` in package `config`. It can also be generate from the Swing GUI using the button **Sign Updates** found under the tab **Updates**, under the tab **Settings**, when one specifies an unexisting file to the popup requesting a file with a Trusted key. Note that to arrive there one has to pass an updates file (e.g., as one generated with the following scripts described below).

By default, the trusted key is RSA with 4k bits. That can be changed by changing the implementation of the method **DD**.`createTrustedSKFile()`.

Further, one can generate a release using the scripts in the folder **installers/DD_P2P** on the github repository. The sample script **./installers/DD_P2P/mk_dist_full_DD.sh** must be executed giving it as parameter the curent version of the release, which has to be identical with the String in **DD**.`VERSION`. That script contains in fact an example of parameters for executing **./installers/DD_P2P/mk_dist.sh**. A release generation should specify the parameters for qualities of test and amount of test that the author of the release evaluates, rather than the values hardcoded in the script **./installers/DD_P2P/mk_dist_full_DD.sh**.

The file "Trusted64.sk" must be placed in the same folder with these scripts, prior to running them. Also, the updated "createEmptyDelib.sqlite" file with the DDL of the database must be placed in the subfolder **0.0.00**, while the latest "DD.jar" file obtained by compiling the project must be placed in **0.0.00/jars/**. Any other updated file must be placed in the relative folder position where it is expected to appear in the installation/update package.

After all files are in placed according to these recommendations, the script

mk_dist.sh is executed as per the above recommendation, and the system will generate a folder with the name of the release version, and two archives with the name composed of the prefix **ddp2p_** and followed by the version number. The folder contains the package for automatic updates, while the archives contains the release file for first installations.

The content of the created folder can be uploaded on mirror used for automatic updates, while the file **DD_Updates.signed** found in that folder has to be copied outside that folder at the URL that is specified as mirror inside the DDP2P system.

Mirrors can compose the updates folders coming from different testers by merging the tester part of the text files, as long as the binary files are identical.

2.2 Overall Structure

Instructions to work with the code in Eclipse are available in the file: `instructionsEclipseSetUp.pdf` on github, prepared by Song Qin.

The code consists of:

- a main engine that was tested on Android, Linux, Windows, MacOS.
- GUI interfaces: we have developed an interface in Java Swing (under package `widgets`), and a beginning of an interface on Android (development led by Dong Hang). GUI interfaces can start the engine and then be attached to it by registering an implementation of the interface `config.Vendor_GUI_Dialogs`.
- Database modules: currently we have three database modules supporting sqlite-jdbc, sqlite-4-java (needed for upgrades on MacOS), and an interface to the Android sqlite.
- Plugins: we currently have two plugins for PC (a chat application, and a game) and a plugin for Android. The game plugin works with the version 1 of the DDP2P, and has to be recompiled to work with version 2 (no big changes should be needed, but menus now subclass a different type).

  The plugins can be loaded dynamically from any `.jar` file found in the `plugins` folder, but can also be linked in the code, by calling a static method `loadPlugin` in the class `PluginRegistration` of package `plugin_data`, namely:
  `plugin_data.PluginRegistration.loadPlugin(Class<?> plugin, String peer_GID, String peer_name).`

  The dynamic loading is performed at startup (or when called from GUI), and happens in:
  `plugin_data.PluginRegistration.loadPlugins(String peer_GID, String peer_name).`

In these methods, the `peer_GID` and `peer_name` are the `GID` and name for the current peer. Simple educational plugins are in the `Hello` examples in folder `plugins`, as well as in the package `AndroidChat`.

- Installers and tools: we have some scripts to create release packages (`installers/DD_P2P`), merge databases of existing installations `merge_databases/Seo`, etc.

2.3 Debugging

Almost each class has the constants `DEBUG` and `_DEBUG`. Typically `DEBUG` is false and `_DEBUG` is true. Printing run-time information is done with `if (DEBUG) System.out.println("CLASS_NAME: METHOD_NAME: message")`.

Sometimes for debugging a class I either:

- set its `DEBUG` to true
- set some of its `if (DEBUG)` into `if (_DEBUG)`.
- a local variable is declared in the debugged method `boolean DEBUG = true`

2.4 Known Bugs

It seems existing `D_Witness` objects signature fails. Have to check if it is due to old bugs when they were made, or something newer.

Motions are not synchronized. Have to check the corresponding class in the streaming `package`, see if they are correctly queried.

Chapter 3

Architecture

3.1 Structure of Main Engine

The engine is composed of a set of libraries (packages) for maintaining data structures for the managed items, as well as servers and clients for exchanging this data. This engine is independent of database and of GUI. These libraries are compiled into DD_Android.jar.

Using Databases, Email and GUI To be independent of database, this whole code uses an abstract database interface defined in `util`.**DBInterface**, which itself uses an abstract class `config`.**Vendor_DB_Email**. Applications using a database or email must instantiate `config`.**Application_GUI**.`dbmail` with an instance of this class. For example on Linux/Mac/Windows we use `util.db`.**Vendor_JDBC_EMAIL_DB** which is based on jdbc for sqlite, javax.mail.jar, sqlite-jdbc-3.7.2.jar. On Android we use (see android code)...

To be independent of GUI, the warnings and notifications of data arrivals to GUI are done via an abstract GUI interface class `config`.**Vendor_GUI_Dialogs**. Applications using some GUI for notifications must instantiate `config`.**Application_GUI**.`gui` with an instance of this class. On standard Oracle java systems we use: `widgets.components`.**GUI_Swing**, whose static method `initSwingGUI` does the job.

Main Application and Tools The entry points for various tools and GUIs are in the subpackages:

- `widgets` : if they use GUI
- `util`.tools : if they use databases but no GUI
- `tools` : if they need no database and no GUI (e.g. tools related to digital signatures and keys). Here there still are some old tools that need

databases and GUI and need to be moved to the appropriate package (and to be changed by adding the aforementioned code for initializing the database and GUI drivers).

Initialization of Identity The communication servers need to know what is the identity of the current peer (to sign sent messages). This identity is loaded from a database using any one of the static methods:

- `config.`**`Identity`**`.getCurrentPeerIdentity_QuitOnFailure()` which may try to use GUI to create or load the peer from a file if the database does not specify the current peer, and it exits with `System.exit()` if no peer is established.
- `config.`**`Identity`**`.getCurrentPeerIdentity_NoQuitOnFailure()` which may try to use GUI if the database does not specify the current peer.
- `config.`**`Identity`**`.init_Identity(boolean quit_on_failure, boolean set_peer_myself, boolean announce_dirs)`.
  - When setting the parameter `quit_on_failure` to true it will try to use GUI if the database does not specify a peer, to create or load the current peer from a file.
  - When setting the parameter `set_peer_myself` to true, this will also initialize a `data.`**`HandlingMyself_Peer`**`._myself` peer (the one used for signing messages), otherwise it just loads the data from the database with no other attempt to create the peer structure or to query the user with GUI for a peer when this is not found in the database.
  - When setting the parameter `announce_dirs` to true, the obtained peer and its addresses are sent to any directory in the list of directories loaded from the database.
- `data.`**`HandlingMyself_Peer`**`.loadIdentity(null)`. Same as the above call with all parameters set to false.
- `data.`**`HandlingMyself_Peer`**`.get_myself_or_null()`. Returns the currently set peer identity, or null if none was set.
- `data.`**`HandlingMyself_Peer`**`.get_myself_with_wait()`. Returns the currently set peer identity, or on null it waits indefinitely for one to be set (e.g. by the GUI interaction n the aforementioned methods).

Initialization of Directories To enable communication to roaming peers and peers behind NATs, such peers need help from *access points* (aka *directory* or *supernode*). A peer with a static IP does not need such a directory! In fact it can volunteer to act as an access point for others (and may get incentives for it, see the IEEE-P2P14 article). Each peer can set-up such access points or

select from existing ones and present it as its address. From that moment the peer would need to keep announcing its current roaming position (IP address) to that access point.

Currently selected access points can be loaded from the database using the static method: `config.`**DD**`.load_listing_directories()` which can generate exceptions, or from `config.`**DD**`.load_listing_directories_noexception()`. They are made available to the code via the static data structures `config.`**Identity**`.listing_directories_xxx` where `xxx` is `inet`, `addr` or `string` which contain the data as `hds.`**Address** structures, as resolved internet addresses for making sockets ready for communication, or as strings ready for pretty printing.

3.2 Connections

The communication in DDP2P can be based on any of the Internet technologies called TCP and UDP. The communication code is found in the package `hds`, while the code to build messages and to integrate incoming messages is found in the package `streaming`.

A module exists for broadcasting over Adhoc WIFI using UDP broadcast (see packages `wireless`, `widgets.wireless`, and `handling_wb`). The module was built under the master thesis of Osamah Dhanoonn and during the last experiments it had some useful general parts commented out for faster hard-coded configuration of some experiments. Somebody should take time to reverse that process and replace the hard-coded shortcuts with the original general procedures.

Hopefully that should be done when the broadcasting module is tested for Android, or when it is extended to Bluetooth communication.

3.3 Streaming

3.3.1 ASN1 Tags

There are various types of messages and the server distinguish them based on ASN1 tag. Each message in DDP2P is encapsulated as an ASN1 object using our own implementation of ASN1 standard found in package `ASN1`. This implementation was not tested to be compatible with the standard but was tested only for the correct invertibility of the encoding and decoding, as well as for the DER minimality and determinism of the encoding (for usage with digital signatures). Implementation of explicit tags was not done, even if it is relatively trivial from the available primitives.

ASN1 Encoder In principle, a container sequence is created with `initSequence()` on an empty ASN1 **Encoder** object. Elements are added to this sequence using `addToEncoder()` functions. There exist `Encoder`

constructors for most primitive types: **boolean, int, byte, BigInteger, String, byte[], double**. A parameter **Calendar** is for obtaining an ASN1 GeneralizedTime. The **Encoder** class provides the static method `getGeneralizedTime(Calendar)` to convert between types Calendar and String in the GeneralizedTime format. Use `getNullEncoder()` or `new Encoder().setNull()` to create a NULL tag.

A tag for the Encoded data can be set with the various `setASN1Type` methods.

To get an Encoder for ASN1 BITSTRING use the method `get_BIT_STRING(byte[])` since the constructor with parameter byte[] is used for OCTET STRING. The constructors Encoder(BigInteger[]) and Encoder(int[]) build an encoding of an ASN1 OID. To encode arrays of integers or arrays of BigInteger you must used the functions described below!

The `getEncoder()` static functions are used to create homogeneous sequences from arrays or ArrayLists. When the elements of these vectors are not primitive, then they must be extending the abstract class **ASNObjArray**, and must be implementing the optional `instance()` method of that abstract class. For primitive elements of the arrays one can use `getBNsEncoder(), getStringsEncoder(), getStringEncoder(), getEncoderArray()`, handling BigInteger[], ArrayList<String>, String[], int[] or float[]

One can encode hashtables Hashtable<String,String> using `getKeysStringEncoder()` or `getHashStringEncoder()`. The second variant encodes both the key and the value, sorted by the key. The first variant encodes just the keys, placing them at positions defined by a function based on the value.

To get the array of bytes serialization from an ASN1 **Encoder** one calls its method `"getBytes()"`. Directly from a class implementing the ASNObj abstract class, the DER serialization is obtained using the method `encode()`, or the calls `getEncoder().getBytes()`.

ASN1 Decoder Our ASN1 **Decoder** works similarly. A **Decoder** object is constructed based on an array of bytes. From a constructed object (SEQUENCE) one removes the outer envelop with the method `getContent()`. Further the **Decoder** of each element is obtained with `getFirstObject(extract)`. The parameter must be "true" for the object to be extracted from the stream (such that we get the next element at the next call). Here there are also separate member function to extract primitives from the current **Decoder** (if it is for a primitive element).

- `getInteger()` returns a BigInteger
- `getString()` returns a String
- `getBytes()` returns byte[]
- `getOID`, `getBNOID()` returns an OID as int[], or BigInteger[], respectively

- `getBoolean()` returns boolean
- `getGeneralizedTime(), getGeneralizedTimeCalender()` returns a generalized time as String or Calender, respectively.
- `getReal(), getFloatsArray(), getIntsArray, getBNIntsArray(), getAny()` returns double, float[], int[], BigInteger[], byte[], respectively. The encoding of reals/floats is not standard!

Each of these functions throws an exception **ASNLenRuntimeException** if the encountered byte type is not the same as the native one for the corresponding primitive. Optionally they can get a parameter of type byte which specifies the expected type byte for the data to be decoded. Some have a discouraged extra version that accepts to decode data of any type without error (and typically are recognized by the suffix AnyType added to the aforementioned method names). A similar type byte can be passed as a parameter of getFirstObject

Homogeneous arrays are obtained with factories `getSequenceOf[AL|BN]` that receive as parameter an instance of the array and/or an instance of the ASN1 object type to be decoded in the array.

```
// The next class implements the following ASN1 definition
// MyASNObj := SEQUENCE {
//     val [0] IMPLICIT INTEGER
// }
class MyASNObj extends ASNObjArrayable {
  int val = 1;
 MyASNObj instance() {return new MyASNObj();}
 @Override
 Encoder getEncoder() {
  Encoder enc = new Encoder().initSequence(); // creates SEQUENCE
  enc.add(new Encoder(val).setASN1Type(DD.TAG_APO)); // IMPLICIT [0] INTEGER
  return enc.setASN1Type(getASN1Type);
 }
 @Override
 MyASNObj decode(Decoder dec) {
   Decoder decoder = dec.getContent; // remove the SEQUENCE envelope
   val = decoder.getFirstObject(true).getInteger(DD.TAG_APO).intValue();
 }
}

// The next class implements the message definition
// Message := SEQUENCE {
//   obj [1] SEQUENCE OF MyASNObj
// }
class Message extends ASNObj {
 MyASNObj obj[];
 @Override
```

```
Encoder getEncoder() {
 Encoder enc = new Encoder().initSequence(); // creates SEQUENCE
                           // obj IMPLICIT [1] SEQUENCE OF MyASNObj
 enc.add(Encoder.getEncoder(obj).setASN1Type(DD.TAG_AC1));
 return enc.setASN1Type(getASN1Type);
}
@Override
Message  decode (Decoder decoder) {
 Decoder content = decoder.getContent();
 obj = content.getFirstObject(true, DD.TAG_AC1)
  .getSequenceOf (
    MyASNObj.getASN1Type(),
    new MyASNObj[0],
    new MyASNObj);
  return this;
 }

 // The next method is an example of usage of the class to decode a message
 static Message extract(byte[] message) {
 return new Message().decode(new Decoder(message));
}
byte[] serialize() {
 return encode(); // or return getEncoder().getBytes();
}
```

Objects of type Hashtable<String,String> are obtained with factory `getSequenceOfHSS` that receives as parameter a type of each element. These are encoded sorted by the key.

Non-primitive objects that extend ASNObj are typically decoded by creating an empty instance with an empty constructor or factory, and then calling their member method `decode()` giving as parameter the decoder of the structure obtained with one of the aforementioned methods (constructor from array of bytes or extraction from a decoder with `getContent()` or `getFirstObject()`.

The type/tag of the object that will be extracted next can be queried on a **Decoder** object using the methods:

- `isFirstObjectTagByte(tag)`. Which returns true if the next object has the type byte equals to the parameter.
- `getTypeByte()`, `type()` Return the type byte.
- `getTagValueBN()`, `tagVal()` returns the tag value as a BigInteger, or an int strictly smaller than 31, respectively.
- `typeClass()`, `typePC()` returns the class and class type values as `int`s.

3.3.2 Streaming Threads

The streaming of data between DDP2P peers is managed by the following threads (classes in package `hds`):

- **Connections**. This thread manages a data structure with all the known addresses of all the peers ("safes") that we are currently polling (i.e., "used" peers).
- **Client2**. This thread uses the data structure maintained by **Connections** to poll peers in a loop based on TCP or UDP. This is a new version replacing **Client1** which was not using a **Connections** data structures, but rebuilt it herself at each polling round.
- **UDPServer**. This is the UDP server thread that listens for connections on a port. To send messages one uses the same UDP port (socket).
- **UDPServerThread**. This is a thread launched from **UDPServer** to handle a given incoming datagram. Since datagrams are limited in size. If incoming messages are of type **UDPFragment**, then they are assembled. Only when the whole message is assembled, then it is interpreted further and handled.

3.3.3 Handling Fragments

To send large messages one can use the method `hds`.**UDPServer**.sentLargeMessage(), which breaks them into blocks of size MTU and encapsulates each of them in a `hds`.**UDPFragment** and in a UDPMessage.

Ar the receiving end, fragments are assembled in the **UDPServer**.`recvMessages` list of **UDPMessage**. The lists of received fragments are reevaluated in each loop of the **UDPServer** thread, with the method `sendFragmentReclaim()` which builds a list of messages expired and reclaimes them.

They are reevaluated also at each loop of the UDPServer-Thread when in `handleRequest()` the peer is checked with **UDPServer**.`transferringPeerAnswerMessage()` for the existence of other ongoing messages towards the same destination (to avoid sending parallel answers to **SyncRequest**, which would be redundant).

Both sending and receiving messages are reevaluated from **UDPServer**.`transferringPeerMessage()` which is used from **UDPServer-Thread**.`handleSTUNfromPeer` to check for duplicates before sending messages SyncRequest.

A peer send only a limited number of fragments at a time (`DD.FRAGMENTS_WINDOW`=10). More fragments are sent when **UDPFragmentAck** objects are received witnessing the array of fragments delivered so far (and which are sent on receiving the fragments or on reclaim). When the acknowledgements testify that all fragments were received we free the buffer.

Under the tag of a reclaim, `DD.TAG_AC16`, the peers also send a **UDPFragmentAck** after an expiration of waiting time. If the message reclaimed is found to be a zombie (a message believed sent and removed from the system) then a NACK is sent to the remote peer. Zombies may exist because either an ack was duplicated and when the latter duplicates arrive the message's arrival was already known, or because some fragment was duplicated and the recipient started to store it as a new message whose other fragments never arrive (having been acknowledges and received once).

Chapter 4

Connection Data Structure

In a diagram, the main hierarchy of data structures explained below could be represented as:

Connections
`used_peers_xxx, myselfPeer_HT_IPPORT_CPD` →
Connection_Peer
`instances` →
| **Connection_Instance**
| `peer_directories` →
| | **Connections_Peer_Directory**
| | `supernode_addr, reported_peer_addr` →
| | **Address_SocketResolved_TCP**
| `peer_sockets, peer_sockets_transient` →
| **Connections_Peer_Socket**
| `addr` →
| **Address_SocketResolved**
`shared_peer_directories` →
Connections_Peer_Directory
`supernode_addr, reported_peer_addr` →
Address_SocketResolved_TCP

Status of Polled Peers The `hds`.**Connections** class keeps in static members the data describing the addresses and resolved sockets of the currently polled remote peers. The corresponding data structure elements are:

- `myselfPeer_HT_IPPORT_CPD`. This is a hashtable mapping the string obtained by concatenating the IP and PORT of each of my listing directories to an object of type **Connections_Peer_Directory** for that directory. The object is placed in this hashtable when an answer from a directory comes containing my own peerGID and instance name.

- `my_directories_AL`. This is an **ArrayList<My_Directory>** for all my listing directories. Its updating and usage not yet implemented as of 12/28/2014.
- `used_peers_AL_CP`. Is an **ArrayList<Connection_Peer>**, one for each of the peers being currently polled.
- `used_peers_GID_CP`. Is an **Hashtable<String,Connection_Peer>**, with an entry for each of the peers being currently polled. It maps the global identifier (GID) of the peer to the connection data for that peer.
- `used_peers_GIDH_CP`. Is an **Hashtable<String,Connection_Peer>**, with an entry for each of the peers being currently polled. It maps the hash of the global identifier (GIDH) of the peer to the connection data for that peer.
- `peersAvailable` simply caches the number of peers in `used_peers_AL_CP`.

Status of the Data Manager For managing the updates needed to be performed to the above structures, we use the following:

- `update_dirs` tells that some modifications were made to my listing directories.
- `update_peers` tells that some modifications were made in my database to the peers that we are polling, and the connection structures need to be updated.
- `update_needing_peer_connections` is a list of objects of type **Connection_Peer** that were inaccessible at the last poll attempt, and therefore whose directories should be queried for updates. Its modifications is synchronized on `lock_update_needing_peer_connections`.
- `lock_used_structures` is a monitor for synchronizing changes to the `used_peers_xxx` data structures.
- `monitor_integrateDirAddresses` is a monitor for synchronizing changes due to incoming data from directories. I believe it should be replaced with `lock_used_structures`, where there is time to test such changes!
- `monitor_wait_obj` is a monitor on which the connections maintaining thread is waiting to be notified of changes or timeout. Timeouts may not be really needed since any change or failure is found separately.

When the connection manager thread updates its data structures, it does not do so concurrently with the **Client2** thread that uses this structures, since both processes could be long and synchronization between them could be slow or complex. Therefore we have opted for implementing it by having the manager create the update into a set of temporary structures, which are later switched with the ones actually accessed by end-user processes.

Working Data Structures Therefore the **Client2** thread accesses the above data structures while the **Connections** creates the update in the set of similar structures:

- `tmp_my_directories_AL`
- `tmp_used_peers_AL_CP`.
- `tmp_used_peers_GID_CP`
- `tmp_used_peers_GIDH_CP`
- `tmp_peersAvailable`

At the end of the update, the data structures are swapped, ad the is the only time when the **Connections** thread needs to hold the lock of `lock_used_structures`. The user thread **Client2** should create its copy of this set of structures at the beginning of each loop, which would also reduce its need of synchronization to a short time (we have to check that we are doing so).

While the **Connections** is creating the new data structure with fields updated from the database changes, various status flags and logs may be changed inside the currently used data structure based on the events on the network (e.g., flags about the connection attempts and their success, as well as new socket and NAT addresses obtained from directories). To correctly integrate these flags into the new data structures build built, the updatable structures are grouped in objects that can be directly linked from the new data. This way the updates happen simultaneously in the old and the new structures.

New uncertified sockets and new instances received from the directories (that are not yet on the disk) have to be linked into the new data structure. Currently new instances received from directories are directly stored in database (table "peer_instance") but that may not be needed and in fact may be counterproductive since it is hard to avoid attacks with spam without requiring signatures certifying each directory information (which would slow down the system). Either that saving should no longer be done until the peer instance contacts us itself with a digitally signed information, or saving should also store the directory server announcing the instance to help trace attacks.

4.1 Data Types for Connections

4.1.1 Address

An **Address** defines away to contact a peer or a directory, and consists of:

- `domain`: host or IP address
- `tcp_port`, `udp_port`: the ports for TCP and UDP servers
- `pure_protocol`, `branch`, `agent_version`: together these define the type of the messages understood by the peer

- `instance`, `certified`, `name`: with local informative value. Each address may be associated with a given 'instance' (rather than being a generic directory for the peer, or a static address of a peer with a single instance). Only the certified addresses are to be stored in database table 'peer_address'. The name is typically used to easily identify directories.
- `active` used for my listing directories to tell whether I am currently announcing myself to this address.
- `priority` used to deterministically order the addresses in ASN1 encoded sequences for correct verification of digital signatures. It could also be used by clients as an order of testing addresses.
- `inetSockAddr` used with my listing directory servers (where the tcp and udp ports are the same), to cache their socket. It is initialized in `config`.**DD**.`load_listing_directories()`.

Address_SocketResolved_TCP

This type of object hold:

- `addr`: an address of type **Address**
- `isa_tcp`, `isa_udp`: cached socket addresses for TCP and UDP, respectively

Address_SocketResolved

This type of object hold:

- `addr`: an address of type **Address**
- `isa_tcp`, `isa_udp`: cached InetSocketAddresses for TCP and UDP, respectively
- `ia`: cached InetAddress

4.1.2 My_Directory

This structure (visibly not yet used) stores, for a directory listing me:

- `supernode_addr`: the **Address_SocketResolved_TCP** socket address of the directory. Initalized in **Connections**.`init_my_active_directories_listings` but never used
- `reported_my_addr`: my (NAT?) **Address_SocketResolved_TCP** socket address seen by the directory
- `last_contact`, `contacted_since_start`, `last_contact_successful`: date and flags of the last contact

4.1.3 Connection_Peer

All the connection data about a given polled peer is stored in an object of this type:

- `peer`: the **D_Peer** object whose connections are described here.
- `shared_peer_directories`: the generic directories listed in peer, of type **ArrayList<Connections_Peer_Directory>**, one for each shared address of the peer that is of type DIR.
- `shared_peer_sockets`: the generic socket addresses listed in peer, of type **ArrayList<Connections_Peer_Socket>**, one for each shared address of the peer that is of type SOCKET.
- `instances_AL`, `instances_HT`: for each known instance of the peer there is a **Connection_Instance** in the array list, and Hashtable, respectively. In the Hashtable, the instances are placed using as key the name of the instance made non-null using the method **Util**.`getStringNonNullUnique()` (which is invertible with `getStringNullUnique`). The instances in the arrayList are sorted descending based on the number of objects exchanged so far.
- `status`: a structure (passed on update from old structures to new ones) containing
  - `contacted_since_start`, `last_contact_successful`: flags to tell the status of the last contact. These flags are encapsulated in the status structure to easily pass them to the next updated version of the structure!
  - `justRequestedSupernodesAddresses`: flag set to tell that directories were just asked for addresses by UDP, and should not send a new request (until **Client2** finds that existing addresses are not yet working, i.e. each second attempt)

    The flag tells that the peer's directories were just asked by UDP for addresses in the previous iteration, and for at least one iteration of usage one can have patience for the UDP answer to arrive.

4.1.4 Connection_Peer_Instance

This structure stores the connection information related to a give clone (aka instance):

- `dpi`: the database/directory information about this clone, of type **D_PeerInstance**.
- `peer_directories`: the directories listed in peer, of type **ArrayList<Connections_Peer_Directory>**, one for each instance address of the peer that is of type DIR.

- `peer_sockets`: the socket addresses listed in peer, of type **ArrayList<Connections_Peer_Socket>**, one for each instance address of the peer that is of type SOCKET.

- `peer_sockets_transient`: the socket addresses received from directories for this instance, of type **ArrayList<Connections_Peer_Socket>**, one for each volatile/transient instance address of the peer that is of type SOCKET.

  Currently transient directories are not stored in separate lists (keeping an old implementation) and therefore there is a risk they can be lost when updating the structures (but we assume that transient directories will likely not exist and therefore we do not take time now to implement a similar `peer_directories_transient` member).

- `status`: a structure (passed on update from old structures to new ones) containing

  - `contacted_since_start_TCP`: set if there was a success of the TCP to this instance since start
  - `last_contact_successful_TCP`: flags to tell the status of the last TCP contact attempt with this instance.
  - `ping_pending_UDP`: flag to tell whether there was a UDP ping sent to this instance without answer so far
  - `last_contact_date_UDP`: flag to tell the last date when aping reply came from this instance

4.1.5 Connections_Peer_Socket

This structure stores all the information needed to keep track of the status of the connection with a given socket address (typically for an instance, but generic ones may exist if the peer has a single instance). It holds:

- `addr`: is an object of type **Address_SocketResolved** containing an address and its resolved sockets.

- `address_LID`: the local pseudokey of the address in the "peer_address" table. A negative value testifies that this is a transient address. received from a directory but not certified.

- `behind_NAT`: true with transient addresses if there was a NAT address reported with this socket

- `_last_contact_date_TCP`,

- `last_contact_successful_TCP`, `contacted_since_start_TCP`: date and flags with the status of the TCP connection to this address

- `_last_ping_sent_date`, `_last_ping_received_date`: dates of the UDP connection to this address (when UDP connection was last attempted, reply was last received).
- `replied_since_start_UDP`: This flag is redundant since it results from the existence of a value to `_last_ping_received_date`.
- `last_contact_pending_UDP`: This flag is set iff any UDP ping was sent since the last UDP ping reply was received.

4.1.6 Connections_Peer_Directory

This structure stores all the information needed to keep track of the status of the connection with a given listing directory of a given peer/instance. It holds:

- `supernode_addr`: on abject of type **Address_SocketResolved_TCP** that contains the resolved address of the directory server.
- data structures concerning the connection via NAT addresses reported by this directory.
  - `_reported_peer_addr`: a hashtable mapping "instance" names for a peer into objects of type **Address_SocketResolved_TCP** that contains the resolved address of the NAT entry to that "instance", as reported by this directory. Instance names are translated in non-null versions using Util.getStringNonNullUnique().
  - `_reported_last_contact_date`: a hashtable mapping "instance" names for a peer into objects of type String holding the GeneralizedTime of the last time a ping reply was received for the NAT address reported for that instance by this directory.
  - `_reported_last_ping_pending`: a hashtable mapping "instance" names for a peer into objects of type **Boolean** which are set to TRUE if a ping request was sent for the NAT address reported by this directory since the last reply, and to FALSE or null otherwise.
- `_last_contact_TCP`, `contacted_since_start_TCP`, `last_contact_successful_TCP`: date and flags for the connection status via TCP to this directory when querying addresses of the current peer.
- `_last_contact_UDP`, `contacted_since_start_UDP`, `last_directory_address_request_pending`: date and flags for the connection status via UDP to this directory when querying addresses of the current peer.
- `address_LID`: the local pseudokey in the database table "peer_address" for the data associated to this directory. Set to -1 if this is a transient directory, i.e., received from a directory.

- `reportedAddressesUDP`: an object of type **ArrayList<Address>** with the complete list of addresses received over UDP from this directory for the current instance. Used in displays, in the widget **DirectoryPing** with status of directories, and as temporary answer from this peer when doing a UDP request.

- `lastAnswer`: this is the whole last answer received from this directory over UDP. It is used only for shared directories to retrieve the last know list of addresses from an instance as temporary answer to an UDP request or for display in the **DirectoryPing**.

4.2 Connections Update Process

The simplified diagram of this process is shown in Figure 4.1. The con-

Connections.`__run`
init
wait_updates (locking lock_updates_pc, check changes in dirs, peers, update_pc)
updates
init `tmp_xxx` & switch
| `init_used_peers`
| for all `used_peers`
| `loadAddresses_to_PeerConnection`
| links transient instances from old and, for all instances:
| `loadInstanceAddresses` and links transient instances from old
| `loadInstanceAddresses_certified` and link transient addresses
| `locatePS` from old or resolve addresses
for all `update_needing_peer_connections`
`update_supernodeaddress()`
`update_supernodeaddress_instance()`
`update_supernodeaddress_instance_dir()`
`getDirAddress()`
`getDirAddressUDP()` only on the failure of TCP
`getKnownDirectoryAddresses()` use old UDP addresses while expecting new ones

Figure 4.1: **Connections** maintenance

nections update process consists of a loop. An initial initialization in `Connections.init()` has as purpose is to make the structures non-null, and loaded from the database.

The loop waits on the monitor `monitor_wait_obj` until a change is signaled in the database, a connection fails, or until a timeout of amount

`CONNECTIONS_UPDATE_TIMEOUT_MSEC` (currently set to 3 minutes). After each wake up, the method `updates` is called.

The method `updates` initializes the `tmp_xxx` data structures in **Connections** with the data in the database (as well as the data in the currently used data structures) and then uses them to replace the currently used data structures.

Further, if there are peer connections that did not work recently, these are updated by contacting to their directories and asking new transient addresses, to be integrated in the data structures.

4.3 Client Process

```
Client2.__run
 loop for each peer, until turnOff is set
  try_wait
  handlePeer
   handlePeerRecent()
   handlePeerNotRecentlyContacted()
    trySocketsListTCP()
    | try_TCP_connection()
    | Client2.transfer_TCP
    |  ClientSync.buildRequest()
    |  Client2.integrateUpdate()
    |   UpdateMessages.integrateUpdate()
    trySocketsListUDP()
    | try_UDP_connection_socket()
    try_UDP_connection_directory()
```

Figure 4.2: **Client2** and connections structures

The **Client2** loops over the available connections and for each of them attempts TCP and UDP connections, and is shown in Figure 4.2.

By calling method `Client2.try_wait()` we make sure that each peer is handled only when the load of the system in terms of number of **UDPServerThread**s (based on `UDPServer.getThreads()`) is below `UDPServer.MAX_THREADS`/2 (currently 3=6/2). Otherwise a delay is set to `ClientSync.PAUSE`. At the loop immediately after a wakeup request, the value of `Client2.recentlyTouched` is true. The client thread also delays the same amount at the end of each loop if the `Client2.recentlyTouched` is not set. The end of the loop is detected by comparison of the current peer index `peersToGo` with `Connections.peersAvailable`.

Once it is decided to poll some peer, this is obtained with `Connections.getConnectionAtIdx(peersToGo)`, and handled with

`handlePeer`. Further each peer is handled in this method separately based on whether it was recently reached or not.

The methods `handlePeerNotRecentlyContacted()`, and `handlePeerRecent()` work by calling `trySocketsListTCP()` (if `DD.ClientTCP` is set) and `trySocketsListUDP()` (if `DD.ClientUDP`) is set. These methods attempt connections separately for each listed socket. If a TCP connection is successful, further connections are not tried for that peer.

In `handlePeerRecent()` the TCP sockets are tried in the reversed order of the date of last connections. Both methods try TCP sockets in instances giving priority to the ones with which we exchanged so far the largest number of items.

If trying UDP connections for an instance, the methods call `try_UDP_connection_directory` to also send pings using directories of the instance as well as the shared directories, as STUN servers.

4.4 UDPServerThread Process

UDPServerThread.`__run`
 Connections.`acknowledgeReply()` on ping reply
 Connections.`registerIncomingDirectoryAnswer()` on answer from server
  `Connections.getConnectionSharedPeerDirectory()`
  for each instance
   may update `Connections.myselfPeer_HT_IPPORT_CPD`
   `Connections.getConnectionInstancePeerDirectory()`
   `Connections.getConnectionPeer()`
   build visualisation data in `D_Peer.peer_contacts`
   `getSocketAddresses_for_peerContacts_widget()`
   | build visualisation data in `ClientSync.peer_contacted_addresses`
   `Connections.integrateDirAddresses()`
    `Connections.locatePS()`
    `Connections.locatePD()`

Figure 4.3: **UDPServerThread** and connections structures

This process retrieves incoming datagram packets, and integrates them based on their type. The algorithm is given in Figure 4.3. If they are UDP ping sent by another initiator, then just a reply ping is sent.

When a reply is received to a ping sent my this agent as an initiator, then a **SyncRequest** is sent back and also the connections structure is updated announcing of the success of the connection.

When an answer is received from a directory with the list of addresses of a peer, new transient addresses are added to the corresponding place in the data structure. Also, NAT addresses reported are stored in the corresponding directory structure. The directory sending the message is identified among the

directories serving the peer mentioned in the reply, based on the IP and port thereof, using methods `Connections.getConnectionSharedPeerDirectory()` and `Connections.getConnectionInstancePeerDirectory`. This can be improved in the future by adding the `address_LID` of the directory in the exchange...

The **UDPServerThread** then calls **Connections**.`registerIncomingDirectoryAnswer()`. This method iterates over all in instances in the answer. If any reports the instance of the agent receiving the message, the result is stored in `Connections.myselfPeer_HT_IPPORT_CPD`.

The results for each instance are stored in the corresponding **Connection_Peer** and **Connections_Peer_Instance**s obtained using method `Connections.getConnectionPeer()` (old versions have used `Connections.getConnectionPeerInstance()`. Significant code there deals with building the data structure used for visualization in `D_Peer.peer_contacts`, and `ClientSync.peer_contacted_addresses`.

The actual integration is done using method `Connections.integrateDirAddresses()`, and is followed by waking up the client using method `Client2.touchClient()`.

The method `Connections.integrateDirAddresses()` iterates over all the instances in the obtained structure and tries to locate previous status objects using locatePS and locatePD. New addresses are stored in `peer_sockets_transient` and `peer_directories`. It also marks instances found, or no longer found, behind NAT.

4.5 Starting the Connection Processes

Each of the three main processes (UDPServerThread, Client2, Connections) can have only one instance running at a time in the system. Those instances can be accessed (as long as they are running) using static members, and the starting/stopping of the processes should be preferably done using given static methods:

- **Client2**: The static reference to the unique instance of **Client2** is in **Application**.`g_PollingStreamingClient` which is of type **IClient** since at certain moments we supported in parallel several implementations of the client (**Client1** and **Client2**). To start or stop the client use the static method `config`.**DD**.`startClient(boolean on)`. To wake up the client that sleeps between two rounds of polling, use **DD**.`touchClient()`.
- **Connections**: A static reference to this process is found in **Client2**.`g_Connections`. It is started with **Client2**.`startConnections()`.
- **UDPServer**: This is started with **DD**.`startUServer(boolean on,` **Identity** `peer_id)`. The unique instance in the system is linked into the static `config`.**Application**.`g_UDPServer` (where one can also find **Application**.`g_TCPServer` and **Application**.`g_DirectoryServer`).

Chapter 5

Data Objects

The data exchanged between peers is organized into semantically independent units, each of them of manageable size. Ideally this size should fit a small number of UDP datagrams, exchangeable in one round of communications between two peers found in adhoc wifi contact between two cars running in opposite directions on a highway. In fact, an encounter may fit one or more such data items.

The data items exchangeable in DDP2P are:

- peer (aka safe)
- organization of type authoritarian
- organization of type grassroot
- active constituent
- external constituent
- neighborhood
- witness stance
- motion
- justification
- signature
- news item
- translation item
- tester recommendation

Each of these items is identified by a unique global identifier (GID) whose construction algorithm depends on the type of item described.

Some items may have multiple interchangeable GIDs such as a public key and its secure hash with an agreed digest algorithm such as SHA1 (e.g., peers, authoritarian organizations, active constituents, tester items).

Agents may locally store only the GIDs of the data units of interest (or an indication of their existence, such as a hash of a bundle of GIDs), while the actual data items may be stored on cloud and retrieved on need based on these GIDs.

The GID of some items may have a scope, being unique only in the context of some other object (e.g., active constituent GIDs are unique only in the context of a given organization GID).

5.1 Object Caches

5.1.1 Motivation and Status

A caching mechanism was developped to speed up access to frequently used objects such as: `peers, organizations, constituents, neighborhoods, motions, justifications, directory entries, and recommendations of testers`. Ideally all shared objects should be made cachable, mechanism that would also permit easy implementation of a cloud support. In particular, if we ensure that any access to the database is made via a small interface in the corresponding objects (**D_Peer**, ...), then it is sufficient to change that interface for supporting a cloud.

We have not yet implemented cache for some important shared objects, such as: `news, votes and witness`. The reason these were left behind, besides the lack of time, is that they are never needed except for:

- shipping them to other peers, or
- for statistics,

which is done acceptably well by database queries, when not the whole database can fit in memory.

TODO: A type of object that needs urgent cache support is the secret key. The system has in its database a table for secret keys and this table is accessed each time signatures are generated or local ownership of a key is verified. Ideally, if the set of known secret keys is small, then it should be fully loaded into a cache, with a flag telling that there is nothing else on disk. A simple cache is implemented by having peers, organizations and constituents optionally cache their secret keys, but this is less efficient then if a specialized cache would be implemented.

To implement caching we use a store consisting of a data structure based on a doubly linked list and a set of hashtables. Each type of cashed object has its own such store, methods, and a thread that handles data in that store.

5.1.2 The Doubly Linked List

We need a data structure with efficient (constant):

- access and removal of not recently used elements,
- insertion of new elements as the most recently used,
- tagging existing elements as recently used,
- locate an item based on a key: local pseudokey (LID) or global identifier (GID).

The doubly linked list is a data structure that offers the first three properties, and can be combined with hashtables to obtained the fourth property.

Java does not have an implementation of a standard doubly linked list. Therefore we provide our own implementation based on the classes:

- `util`.**DDP2P_DoubleLinkedList<T>** which encapsulates a circulat doubly linked list with at least one element (`head`) that has an empty payload, and the `counter` of the number of used nodes (all nodes except for the head must have a payload).

  Insertions are done in `head.next`, and removals of unused is done in `head.previous`. Any used element is moved at retrieval after the `head`, to be marked as recent. It is expected that the type **T** implements the interface `util`.**DDP2P_DoubleLinkedList_Node_Payload<T>**. Insertions are done with `offerFirst()`, marking as recent with `moveToFront()`, and removal with `removeTail()` and with `remove()`.

- `util`.**DDP2P_DoubleLinkedList_Node<T>** is a generic class defining the type of a node in the linked list. Besides the fields `next` and `previous`, it also has the field `payload` of type **T**. It is expected that the type **T** implements the interface `util`.**DDP2P_DoubleLinkedList_Node_Payload<T>**.

- `util`.**DDP2P_DoubleLinkedList_Node_Payload<T>** is an interface for payloads of the doubly linked list. The interface requires two methods whic are used to set and retrieve a pointer from a payload to the instance of `util`.**DDP2P_DoubleLinkedList_Node<T>** that holds it. This pointer is used to locate the node of a used object, such that the node could be relocated to the head of the list.

5.1.3 Payloads of Doubly Linked Lists

Typically each of the cached object types such as **D_Peer**, **D_Organization**, **D_Constituent** .... implements the interface pkgutil.**DDP2P_DoubleLinkedList_Node_Payload<T>**.

These classes have to implement the two methods of the interface pkgutil.**DDP2P_DoubleLinkedList_Node_Payload<T>**. To group

most of the the members of the payload that are related to caching, a static class member is defined in each of the cached objects. For example we have:

- **D_Peer.D_Peer_Node**,
- **D_Organization.D_Organization_Node**,
- **D_Constituent.D_Constituent_Node**,
- **D_Neighborhood.D_Neighborhood_Node**,
- **D_Motion.D_Motion_Node**,
- **D_Justification.D_Justification_Node**,
- **D_RecommendationOfTester.D_RecommendationOfTester_Node**,
- **DirectoryServerCache.D_Directory_Storage**.

The typical members of each of these are:

- `loaded_object` is a static member holding the cache, i.e. the doubly linked list instance of **DDP2P_DoubleLinkedList<T>**.
- `current_space`, is a static member holding the estimated RAM size of the current cache.
- `MAX_TRIES` is a constant with the number of attempts to remove unused elements from the cache doubly linked list when the size has grown more than the maximum allowed for this cache. At each attempt at most one element is removed from the cache, if it is not locked for writing. Those locked for writing are moved to the front of the doubly linked list of the cache.
- `my_node_in_loaded` holds the pointer to the instance of `util`.**DDP2P_DoubleLinkedList_Node<T>** in the doubly linked list of this cache.
- `message` is an ASN1 encoding of the cached object, to help having it ready for sending in messages. TODO: when calling `getEncoder()` on a cached object one should send this value instead of reencoding the object.
- `loaded_By_LocalID` is a **Hashtable<Long,T>** mapping pseudokey identifiers of type **Long** into cached objects found in the doubly linked list.
- `loaded_By_GID`, `loaded_By_GIDH` are **Hashtable<String,T>** mapping global identifiers (GID) and their hashes (GIDH) of type **String** into cached objects found in the doubly linked list.

- `loaded_By_GID_ORG`, `loaded_By_GIDH_ORG`, `loaded_By_ORG_GID`, `loaded_By_ORG_GIDH` are alternatives to the data structures mmbloaded_By_GID and `loaded_By_GIDH`, that are needed for caches where the GIDs may repeat in different organizations. This is the case of **D_Constituent**, where the GID is a public key that can be reused in different organizations.

  Currently such hashtables are not needed for **D_Neighborhood**, **D_Motion** and **D_Justification**, where the GIDs are currently computed by hashing the data of the object including the GID of the organization for which the object is relevant. However these four structures are used in these cached types since we want to make it easy to experiment in the future with implementations where objects are shared between organizations.

  These alternative hashtables are of type **Hashtable<String,Hashtable<Long,T»** or **Hashtable<Long,Hashtable<String,T»** mapping global identifiers (GID) and their hashes (GIDH) of type **String** into tables mapping LIDs of organizations into cached objects found in the doubly linked list.

5.1.4 Locking Objects in Cache

To avoid that an object is dropped from cached while being modified and not yet saved, a locking mechanism is implemented. Each cached object has a member `status_lock_write` that is incremented by any thread that plans to modify the object and is decremented after the modifications were finalized (and potentially a storing activity to permanent storage was scheduled). This is accessed via `get_StatusLockWrite()`, `inc_StatusLockWrite()` and `dec_StatusLockWrite()`.

For **D_Peer**, **D_Organization** and **D_Constituent** objects there is an additinal locking flag that avoids removing them from the cache as long as they are delected as the *current* context for a GUI. This flag is `status_references` and is managed via `get_StatusReferences()`, `inc_StatusReferences()` and `dec_StatusReferences()`.

As long as one of these counters is positive, a cache does not drop the object, preventing conflics that would lead in duplicate objects being modified concurrently and inconsistently for the same GID.

To increment the lock `status_lock_write` of an object, it has to be done when the object is queried from the cache (or added to the cache) via a factory of the object (generally the parameter `keep` of the factory). This lock is released by users based on calling `releaseReference()`. The `status_references` lock can be raised while the `status_lock_write` lock is raised.

If two different threads attempt to raise simultaneously the `status_lock_write` lock (raising it to values above 1), it signals a potential bug. Always the first thread locking the object has its **StackTraceElement**[] data stored in the field `lastPath`, such that it can be displayed for debugging

if a second thread tries to lock the same object concurrently. These tests are done under the synchronization of `monitor_reserve`.

TODO: Some of the caches may inverse the usage of the two locking means or use the name `status_references` instead of `status_lock_write`. This should be checked and the names should be standardized!

5.1.5 Hashtables of Cached Objects

The hashables shown in the previous section are used to efficiently retrieve cached objects both by their LID and by their GID. Whenever an object is created or read from the disk and registered in the cache, it is also linked into the corresponding hashtables.

Some objects are recently received or created and have GIDs but were not yet stored on the disk and therefore have no LID pseudokey at the moment of their registration in the cache. Nevertheless they need to be registered in cache before saving them, to make sure that we can lock them to avoid that two objects with the same GID are saved in parallel leading to inconsistent databases or errors.

When registering an object that does not yet has a LID, that object is not linked into the `loaded_By_LocalID` hashtable of the cache. As soon as the object is saved (while being locked), the method **T**.**T__Node**.`register_newLID_ifLoaded()` is called via the method `setLID_AndLink` of the cached payloads, and it registers the LID in the hashtable, while checking for potential bugs from illegal usage that would have resulted in duplications of cache for the same GID or LID.

Similarly, some objects are created and while they are being edited they do not yet have a GID (e.g., organizations, motions, justifications). These *temporary* objects when they are registered in cache (to enable locking them for writing), they are not locked in the corresponding hashtables for the GIDs. When the object editing is finalized by generating a GID, the object is linked into the apropriate hashtables via the method **T**.**T__Node**.`register_newGID_ifLoaded()`, called from **T**.`setGID_AndLink()`.

5.1.6 Factories of Cached Objects

Users are expected to obtained cached objects not via the private constructors that take as parameters LIDs and GIDs, but via *factory* static methods. This is required in order to avoid duplicate instances for the same object, which could lead to inconsistencies, and to ensure that cached versions are found and given priority to reloading from disk/cloud.

An empty constructor, available via a factory called `empty()` is made available to be used when decoding ASN1 messages, or when building new objects from scratch in the editor. Objects created in this way and that do not yet have a GID because they are temporary, are stored using the method

`storeSynchronouslyNoException()`, and can later be loaded into the cache using the factory methods `getXXXByLID(LID, ...)`

Objects created with empty constructors or factories and that have a GID are then registered in the cache and saved by following the next steps (sometimes implemented in a method called `storeRemote()`):

1. Query the cache by the GID using a factory with `create` and `keep` flags set, and when available in the factory, with the new object in the `storage`. If nothing was known about the requested object:

   - if a no-null object is provided in parameter `storage`, then this object is linked in the cache, filled with the right GID, marked dirty, and returned.
   - if no storage parameter is provided by the called, them a new object is created, with the GID set to the provided value, and marked dirty and temporary

   Some factories can take as parameter both GIDs and GIDHs. Their types (GID vs GIDH) can be detected dynamically and GIDHs can be computed by the factory from a provided GID.

2. If the object returned was different from the one we need to save, then the content of the one we need to store is copied into the one returned by the factory, e.g. using methods called `loadRemote()`.

   Typically `loadRemote()` also checks first whether the new object is really newer or more complete then the previous one by either checking if the old one was temporary or by comparing creation dates, quitting with false if parameter is not newer. If the object is filed with the parameter, then some dirty flags are set and the temporary flag is also cleared.

   The method `loadRemote()` can detect when unknown objects are being referred (e.g. a justification regering the GID of an unknown motion or constituent), and those GIDs are stored in the parameter `missing_rq` bag, to be requested from peers. Temporary objects with those GIDs are also created using apropriate factories. GIDHs of newly obtained objects are returned in the parameter `sol_rq` bag.

3. If a dirty flag was set, then the cache saving threads are notified by calling `storeRequest()` on the object.

4. The object's lock is released using `releaseReference()`. If this step is forgotten, then when a new operation tries to lock the object, the cache with dump debugging information with the trace of the location of the first locking operation to help in debugging. Also subsequent accesses are delayed for 5 seconds to try to avoid accidental concurency.

For example, the code for storing newly arrived justifications in `streaming.`**`WB_Messages`**`.store()` is:

```
if (! j.isGIDValidAndNotBlocked()) continue;
// preparing management of missing/obtained GIDH
rq = missing_sr.get(j.getOrgGIDH());
if (rq == null) rq = new RequestData();
sol_rq = new RequestData();
new_rq = new RequestData();

// actual integration of a decoded justification "j"
// assumed verified for signatures and blocking
D_Justification jus =
       D_Justification.getJustByGID(
                 j.getGID(), true, true, true, peer,
                 p_oLID, p_mLID, j);
if (jus == j) {
    jus.fillLocals(sol_rq, new_rq, peer);
    config.Application_GUI.inform_arrival(jus, peer);
    jus.storeRequest();
} else
  if (jus.loadRemote(j, sol_rq, new_rq, peer)) {
    config.Application_GUI.inform_arrival(jus, peer);
    jus.storeRequest();
  }
jus.releaseReference();

// management of new and missing GIDH
rq.update(sol_rq, new_rq);
missing_sr.put(j.getOrgGIDH(), rq);
```

For only reading the content of an object, it will be loaded by calling a factory based on the known LID or GID, and without the `keep` parameter set.

For reading and writing the contect of an object, it will be loaded by calling a factory based on the known LID or GID, and with the `keep` parameter set. After modification the user should set appropriate dirty flags, based on the parts of the object that were modified, and should call the `storeRequest()` method, followed by the `releaseReference()`. The `storeRequest()` method should not be called if no change was made, but the `releaseReference()` method must necessarily be called to unlock the object.

If we already have a reference to an object before we decide that we need to lock it for writing, that can be achieved using factories of the type `getXXXByXXX_Keep()` which try the cache all LID, GID and GIDH of the object passed in parameter, before registering a new one. The actual such factories are called: `getPeerByPeer_Keep()`, `getOrgByOrg_Keep()`, ..., `getJustByJust_Keep()`

TODO: Not all the pieces of code in `streaming.`**WB__Messages**`.store()` are so standardized as the one shown here, and the remaining ones should be

brought to such an optimized shape. Also, not all are yet tested for blocking and signatures or GID correctness. With improvements, communication should be tested to see if synchronization work well after each change!

5.1.7 Constants to Control Caching

There are two Thread classes used for each cache type XXX: **D_XXX_SaverThread** and **D_XXX_SaverThreadWorker**. The first type of thread has a single running instance that wakes up when notified by `storeRequest()` or on timeouts and, when there are not too many running threads, will create an instance of the thread of the second type which will store a dirty object from the cache. Dirty objects are found in **HashSet**s `_need_saving_obj` and `_need_saving` where they are placed by `storeRequest()`. Historically, when the object has a GID/LID is stored in the second of those hashsets and otherwise in the first.

TODO: However not the distinction is would need to be reverified.

A set of constants are used to controll the behavior of the two threads that control the storing of each cache to the permanent support (disk/cloud). These constants are in class **SaverThreadsConstants**. These constants control:

- `MAX_LOADED_XXX`, `MAX_XXX_RAM` are user modifiable variables controlling the sizes of the caches. There is also a constant `MIN_LOADED_XXX` since it referes to the count of objects undre which the user-modifiable limit in `MAX_LOADED_XXX` is not considered.
- `MAX_NUMBER_CONCURRENT_SAVING_THREADS`: the maximum number of running saver threads instances of **D_XXX_SaverThreadWorker**, over which new ones are not started.
- `SAVER_SLEEP_BETWEEN_OBJECTS_MSEC`: delay between launching threads when there are many waiting to be saved.
- `SAVER_SLEEP_WAITING_OBJECTS_MSEC`: delay between launching threads when there is none waiting to be saved.
- `threads_xxx` number of threads for each given type xxx of cached objects. The total count is given by `getNumberRunningSaverThreads()`.

Chapter 6

Streaming Logic

Since each peer's database may be voluminous, we do not intend to have complete databases synchronized at each interaction. Rather we want at each TCP or UDP encounter to only exchange a token amount of information. For this purpose a total order is defined on all semantically independent items of information of the database. This order is given by the `arrival_date` in the database, namely the date when the latest version of the item was locally defined (either by arrival from another peer, or by local construction). The resolution of this date in is terms of milliseconds. Due to limited precision of the local clock, multiple items may have arrived at the same declared arrival time and therefore they would have to be shipped together. To avoid large such clusters, we tend to artificially delay saving arrival objects with delays of a millisecond. To force different items to have different arrival timestamps, the arrival timestamp of each item is generated using the static method: `data.`**`ArrivalDateStream`**`.getNextArrivalDate()`, which sleeps 1ms when there is a conflict. This is currently not done consistently, and a mechanism to ensure it across the whole system may be designed using a global timestamping manager to be queries by all item saving processes (e.g., intercepting `Util.CalendarGetInstance()` or `Util.getGeneralizedTime()`).

6.1 Incremental Synchronization

Each agent Alice keeps for each polled peer Bob the latest arrival time of that peer, $arrival\_time_{Bob}$, up to each it has obtained all the data. At each poll round, Alice will send to Bob the value it has for $arrival\_time_{Bob}$ (`lastSnapshot` in the request) and Bob will reply back with bag of GIDs of sequential items (in the order given by arrival times), and starting immediately after the arrival time, as well as the arrival time of the last of them to be used as the new $arrival\_time_{Bob}$ of Alice.

Alice also send in each poll a bag of GIDs it has obtained from Bob and which are for items that she does not have yet. Therefore Bob gives priority in

each reply to sending some of the objects requested by Alice in the poll, rather then sending new GIDs. First Bob ads to the reply message requested items until it reaches an overall size of 3/4 of a datagram. New GIDs are sent only if after adding all requested data, the overall size of the obtained message is smaller than the maximum size of a datagram.

Bob has to send back the list of GIDs that he does not have, such that Alice avoids asking them again from him (this is not implemented yet!).

6.2 Building SyncRequests

The elements of a sync request, defined in class `hds`.**ASNSyncRequest**, are:

- `version`: the current version of the request structure is "2". It is not used since a best effort is done now at decoding.
- `lastSnapshot`: the arrival time of the last items etrieved from the destination. We want items with larger arrival times. A null value signifies that we want a block of items with the smallest arrival times. This time is set to null when we first query a peer or when we want to restart retrieving from beginnig (e.g., because we were announced by that peer that they have changed the policy by which it disseminates its older data). Users can also request to reset this `lastSnapshot` via a GUI interface (r.g. peers and peer-instances widgets). The agent is storing the next `lastSnapshot` for each instance in the **peer_instance** table, based on the answered received from that peer.
- `randomID` this stores a random string to make each request different to avoid replay attacks, as well as to avoid replying twice to the same request that happens to arrvive simultaneously on two different paths.
- `tableNames`: is a set of names of tables that the peer supports (e.g., peers and news). In original versions of the streamming, peers where shiping data as tables. This is currently scrapped since it is not sufficiently flexible to ship data of various versions, and this field can be deprecated and left empty.
- `orgFilter`: is a set of filters that tell which organization and which motions/constituents in that organzation are of interest to this agent. The feature was implemented but was not tested in recent versions. It is currently sent empty to tell the peer that no filter is installed.
- `address`: the **D_Peer** object of the peer sending the request, to be used for validating the request (and its signature) at the destination.
- `request`: the set of GIDHs that this user is interested in getting from the remote peer. It is an instance of `streaming`.**SpecificRequest**, where GIDHs are groupped by organization, with one entry per organization in the array `rd` of `streaming`.**RequestData**. The **SpecificRequest** also

contains lists of GIDs for global `news` and translations (`tran`) as well as a hashtable `peers` that maps peer GIDHs into their minimal creation dates requested.

- `plugin_msg` contains a set of messages sent to the plugins of the remote peer and generated by the plugins of the sender, tagged with the plugin `GID`. These messages are typically saved by plugins in the database of the peer, and therefore they are retrived in this database when the request is built.

- `plugin_info` is a set of descriptions of plugins registered with the sender agent, to let the receiver peer know about the possibility to communicate with them.

- `pushChanges` has the structure of a **ASNSyncPayload** response to a request and allows the sender of the request to push the data it desires to the remote peer. Typically we place in it objects (motions,votes) recently created with the GUI editors of the sender agent.

  This is calculated with `hds.`**ClientSync**`.getSyncReqPayload` which assembles the component `advertised` of `pushChanges` (of type **SpecificRequest**) by merging the data in **ClientSync**.`_payload_fix` describing items manually registered by the advertiser, with the data in **ClientSync**.`_payload_recent` describing data recently created locally.

  The registration of recent data is made with: `hds.`**ClientSync**`.addToPayloadAdvertisements()`.

- `signature`: is a signature of the whole request message, signed with the key of the agent mentioned in the field `address`.

- `dpi`: is the information of type **D_PeerInstance** about the instance of the sender peer describing the particular agent sending the message. It has its own signature.

The ASN1 description of the SyncRequest message is:

```
TableName := IMPLICIT [PRIVATE 0] UTF8String
NULLOCTETSTRING := CHOICE {
OCTET STRING,
NULL
}
ASNSyncRequest := IMPLICIT [APPLICATION 7] SEQUENCE {
version UTF8String, -- currently 2
lastSnapshot GeneralizedTime OPTIONAL,
signature [APPLICATION 8] IMPLICIT NULLOCTETSTRING OPTIONAL,
tableNames [APPLICATION C0] IMPLICIT SEQUENCE OF [PRIVATE 0] IMPLICIT TableName OPTIONAL,
orgFilter [APPLICATION C1] IMPLICIT SEQUENCE OF OrgFilter OPTIONAL,
address [APPLICATION C2] IMPLICIT D_PeerAddress OPTIONAL,
request [APPLICATION C3] IMPLICIT SpecificRequest OPTIONAL,
```

```
plugin_msg [APPLICATION C4] IMPLICIT D_PluginData OPTIONAL,
plugin_info [APPLICATION C6] IMPLICIT SEQUENCE OF ASNPluginInfo OPTIONAL,
pushChanges ASNSyncPayload OPTIONAL,
signature NULLOCTETSTRING, -- prior to version 2 it was [APPLICATION 5]
dpi [APPLICATION C7] Implicit D_PeerInstance OPTIONAL
}
```

The request is commonly prepared in `hds.`**ClientSync**`.buildRequest()` and is handled by `hds.`**UDPServerThread**`.handleRequest()`.

6.3 Replying SyncRequests

A limit is preestablished on the number of objects we want to ship in the response of one request. First a number of objects is sent to the destination based on the specific request of GIDHs received in the SyncRequest.

Function of the remaining space in the answer message, the remaining part of the answer is built in two passages. First the database is scanned separately for each type of object (peer, prganization ...) for data newer than the value of `lastSnapshot` in the request. At each query of such data, a limit is set to the maximum number of retrieved objects, and tha maximum date is computed as the date of the most recent object withing that limit. The output of this first passage is a date: the date until which data is retrieved.

In the second passage we gather GIDHs for all the data older than `lastSnapshot` and the at most as new as the date obtained in the first pass. GIDHs harvested in this passes (filtered to only contained broascasted objects that are not temporary and have GIDHs), are mailed to the requesting peer.

TODO: If a request comes for an object that this agent does not have ready, the peer has to be informed to avoid requesting that object again. Otherwise we risk that future requests could be cluttered with GIDHs that I do not have and a practical DoS is acieved, where I can no longer send anything to this peer. An attacker can create it by somehow convincinf a peer that I have these GIDHs. It is therefore essential to warn the peer about the fact that they are not available at me. Recent changes were needed in the data structure `streaming.`**OrgPeerDatHashes** used to store items to request. However, these changes were not yet fully implemented in the management of this data! See the MS thesis if Hussein Ihsan that has a discussion on the topic but unfortunately was not yet implemented.

6.4 Integrating Data

Chapter 7

AdHoc Synchronization