

Technical Design

Deliverable #1.3



Version: 1.0

Date: 1/10/2024

Authors:

Fintan McGee

Version History

Version	Date	Author	Description
V0.1	30/01/2023	Fintan McGee	Initial version
v.02	1/4/2023/	Fintan McGee	Added architecture information and twitter info
V0.3	1/10/2024	Fintan McGee	Updated document to reflect the current application state, clarified requirements in a separate table
V1.0	1/10/2024	Fintan McGee	Updated to version 1.0 for Deliverable (originally due at M18) document is a work in progress still and will be revised further.

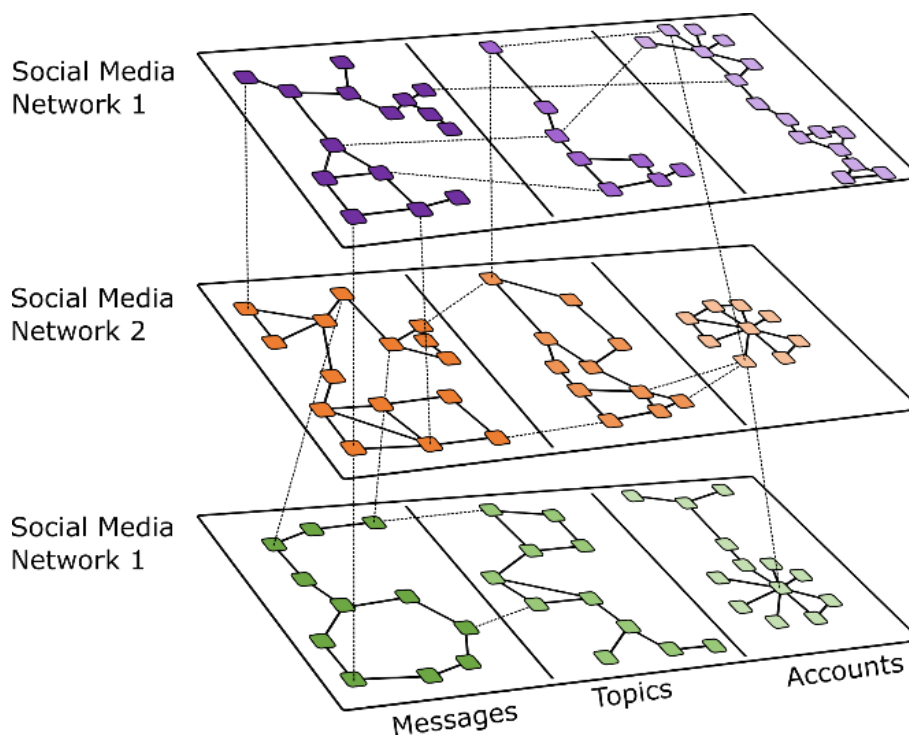
Contents

Version History.....	2
1 Introduction	3
2 Architecture and Technology overview	4
2.1 Angular Web front end.....	5
2.2 Node JS server Web front end.....	5
2.3 Graph Processing Python Server	5
2.4 Application data base	5
2.4.1 Third level subsection section heading.....	Error! Bookmark not defined.
3 Multilayer Network Definition and Implementation.....	6
3.1 Multilayer Network model Overview.	6
3.1.1 Nodes and Links	6
3.1.2 Class MultilayerNetwork.....	7
Appendix A: Appendix Heading	Error! Bookmark not defined.

1 Introduction

This document describes the design of the CON-NET visual analytics platform. Its goal is to describe the overall system architecture, a description of all technologies used and also to provide detail about low level implementation features.

A key aim of CON-NET is to use the multilayer network framework (Kivelä *et al.*, 2014) to integrate content and network structure in online social networks in order to understand, detect, and mitigate misbehaviour in these networks. Online misbehaviour is predominantly understood on a single social network. A multilayer approach is needed to understand the complex patterns of misbehaviour and their consequences with meaningful impact, e.g., orchestrated manipulation campaigns can be only fully understood in the context of different social and knowledge networks



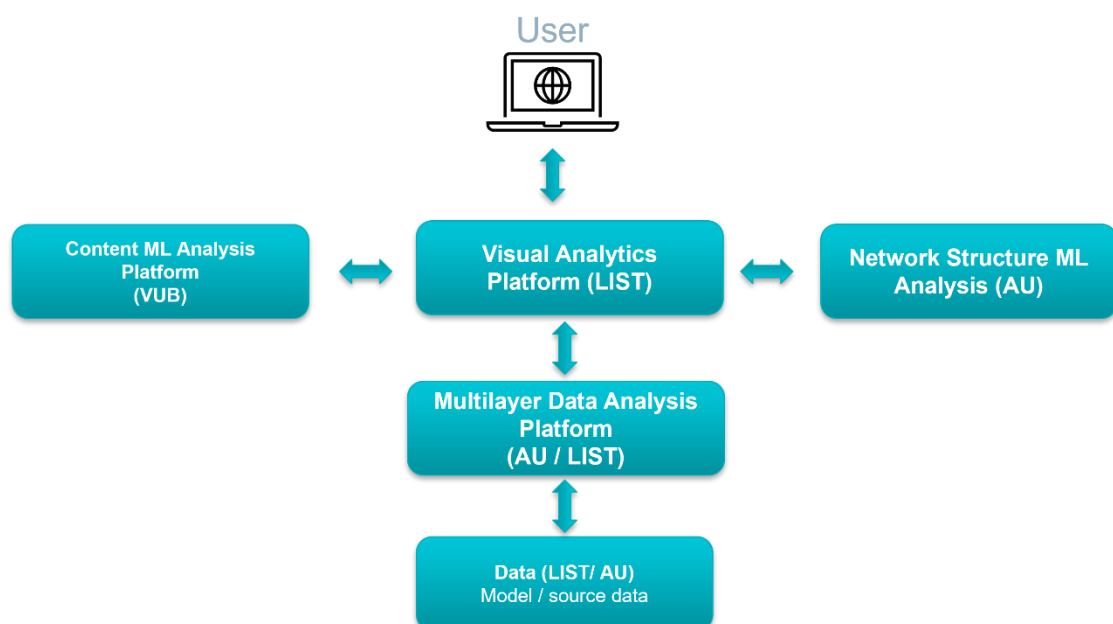
2 Requirements

The application is designed to analyse large social networks using a multilayered approaches. The network will be augmented with Machine Learning outputs, indicating the bot score of nodes supplied by Sabanci, as well as the level of misinformation of n content supplied by VUB.

Requirements#	Functional Requirement	Technical Implementation
1	The systems should be accessible by users easily without special software installations	The system will be developed as a web-based platform
2	The system should be capable of visualising and processing large data sets in the order of millions of nodes	The system should use a data base approach suitable for managing large data sets
3	The system should be responsive even when visualizing large data sets	The system will leverage performant web technologies such as webGPU
4	As the system will contain personal data such as tweet content, it must be secure	The system must require authentication and encrypted data for transmission to users.
5	The system should be able to analyse the network data, providing standard centralities and information.	The system will integrate with an analytics server that will process graph data with common algorithms (e.g. python's networkX)
6	The system must be able to integrate enrichment data from the systems project partners	The system will be able to import enrichment information generated by project partners

3 Architecture and Technology overview

The list application will integrate ML learning outputs from the parent organisation (Sabanci and VUB) into network visualizations server through a web front end



18

Figure 1 Initial overview of the system. Originally https connections were foreseen between LIST VA platform and the ML processing platforms, but that has been replaced by file imports.

The list visualization application consists of an angular a web application served by a node JS server. The application web server stores data in a Cassandra data bases and leverage a python serve for graph processing

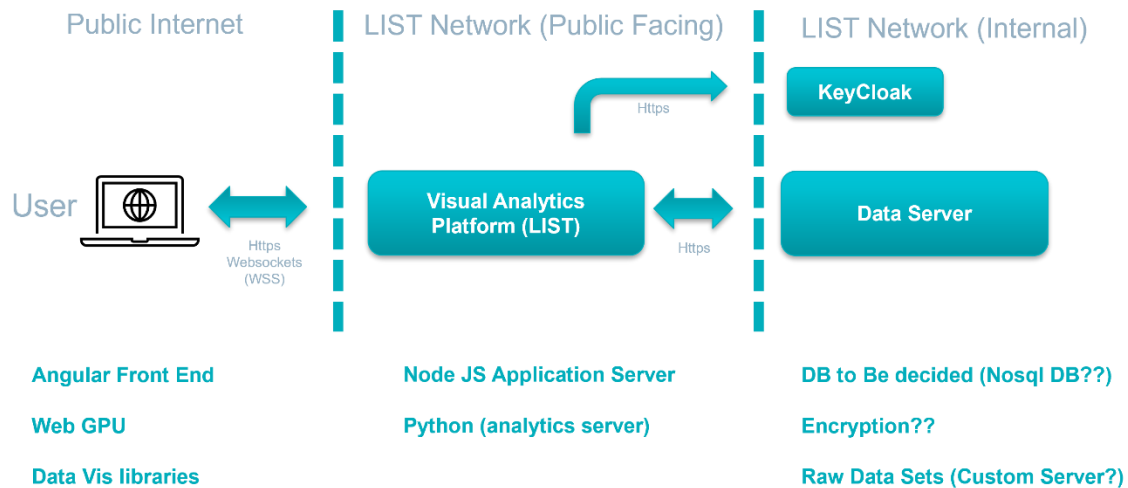


Figure 2 Over view of the VA tools architecture

3.1 Angular Web front end

The angular web front end used (we propose v 13) served by the node.js server. This allows users to log on to the application from standard equipment (see requirement 1). The angular front end allows for a fast responsive user interface and the implementation supports graphs being visualized using WebGPU, allowing interactivity to be maintained for large graphs (requirements 3).

3.2 Node JS server Web front end

This will serve the angular front end, store the graph being visualized by currently logged in users (requirement 1), fetch and store data in the data base, do some local graph processing (requirement 5) and connect to the graph processing server. Server access is controlled using a key cloak instance to manage usernames and passwords, and web pages are served using HTTPS encryption (requirement 4).

3.3 Graph Processing Python Server

The python graph processing server calculates standard centralises (degree, betweenness eigenvector etc) as well as clustering's and calculating layouts, using standard libraries such as Network and SNAP. (Requirement 5)

3.4 Application data base

User graphs will be stored in a CASSANDRA data base a fast scalable solution used on many large platforms and websites (see requirement 2). Application Data

There are two types of data live social media data connected to twitter, and previously downloaded data sets (e.g., the data set provided by Finland) which can be loaded into the application. The live data will be used for experiments with real users and data

3.5 Twitter integration

In early versions of the application twitter integration was a key feature. Due to the cessation of Academic access this is no longer an option, and an alternative is being sought.

3.6 Machine Learning integration

In early versions of the application, it was expected to integrate directly with the ML processing servers (requirement 6) , however a simpler approach of importing enrichment data from text files on the server has been adopted. This will require preprocessing of some data a by project partners for experiments.

4 Multilayer Network Definition and Implementation

The multilayer network calls is a core piece of functionality in the application.

It is defined as a JavaScript class (declared with the keyword class, as part of the ECMAScript 6 specification), that is part of the node JS server The class is named *MultilayerNetwork*. In the front, end application, it is accessed via interfaces defined via typescript. The full multilayer instance is not stored at the front end, only individual layers for display.

4.1 Multilayer Network model Overview.

The base layer of a multilayer network is our model is the root layer. It is the union of all nodes and edges that exist in any layer of the multilayer network.

The root contains an array of all other layers in the multilayer network. Each layer ins an Instance of the multilayer network class. Only the root layer instance of the multilayer network class has these values populated.

On every layer, an array of nodes and links can be found. Nodes and links are defined by a uid. This is an identifier for that entity (represented by a node) or relationship (represented by a link) no matter the layer. It is unique for the entity or relationship being modelled, not for the node or link instance. Each layer in a multilayer network has its own set of nodes and links. The same entities may be modelled in multiple layers by different nodes instances, all with the same uid.

4.1.1 Nodes and Links

On the application server nodes and links are modelled by the MLN_Node and MLN_Link classes

The MLN_node class represents a modelled instance of an entity in a specific layer. In other works if the same real-world entity appears in multiple layers, each appearance is a new instance of the MLN_node class. Similarly, an instance of MLN_Link models a relationship in a specific layer. The same relationship can have multiple instances of MLN link in across different layers. Instances of nodes and Links only belong to a single specific layer.

Thre are 3 different types of data stored related to nodes and Links: properties, centralities and attributes.

4.1.1.1 Properties

In the context the CON-NET definition, Properties are data items that are specific to an instance in a layer. They can be considered as relating to the mathematical graph object.

Each node has a fixed set of properties, such as position (x and y), weight, label, color, clusterId etc.. New properties cannot be added a runtime. The properties are the list of items needed to model that specific node instance and display it at the front end. Similarly for links. The values of the properties can be changed (e.g. x, y, label, color, etc) as needed.

4.1.1.2 Centralities

In the context of CON-NET, centralities are metrics calculated for a node or a link, in a given layer.

There are a wide range of centralities, and the list is not fixed. Therefore all centralities calculated are stored in a look up table in the node's layer, with the node UID as key. The centralities can be accessed via a node, which calls this look up table. However, a single current centrality value may be stored on a node in a special field, to alter the appearance of the node to help with filtering in the front end.

4.1.1.3 *Attributes*

Attributes are data items associates with the real-world entity but are not specific to a node in a specific layer. They are stored at the root layer in a look up table (with the node UID as key)

Multilayer Network Implementation

The Multilayer network definition is stored in the server in the file "*MultilayerNetwork.cjs*". It is a common JavaScript file, not typescript.

4.1.2 *Class MultilayerNetwork*

The class multilayer network is used to represent all layer in the network. For a given MLN there is a root layer, which contains an array of all other layers. For everz node and link instance that apperars in another layers, there should be a corresponding node and link instance with the same uUIDid in the root layer. Only the root layer instance contains other layers. Each instance of class contains its own array of nodes and links

Instances are crated by passing an array ob objects to become nodes and objects to become links to the constructor.

4.1.2.1 *Member variables*

#nodes

An array of all nodes stored in the layer.

#node

A look up table (JavaScript object) that maps from node uid to a node instance.

#links

An array of all nodes stored in the layer.

#link

A look up table (JavaScript object) that maps from link uid to a node instance.

#multilayerNetworkName;

A string indicating the name of the entire multilayer network.

#rootLayer

A reference to the root layer of this MLN. If this is the root layer, it refers to this layer itself.

#layers

An array of reference to the layers of the Multilayer Network. It is only populated for root layers.

#layerName

A string indicating the name of the entire multilayer network. For a root layer this must be “root” (which is also the default for a new MLN).

#layerSchemaName

A string indicating the approach to layer that is applied to this network.

#aspects

Aspects are sets of layers with common characterizations, This JS object would be a mapping from aspect name to a set of layer. An example would be “Time”, that aspect would map to a reference to all layers. Aspects are not currently integrated into the application as this functionality is still under development.

#nodeLinkMap

a JS object that maps node UIDs to all links incident on that node (in or out).

#colorCategoryField;

The name of the field that has been used to determine color category (sometimes multiple fields can be combined).

#colorCategories

array of all color categories currently in graph,

#nodeAttributes

A JS object look up table that maps the node UID to a JS object containing name value pairs of attributes. This is only populated on the root layer, as attributes should not differ across layers. For nodes on layers that are not the root, attributes are retrieved via the layers reference to the root layer.

#linkAttributes

A JS object look up table that maps the node UID to a JS object containing name value pairs of attributes. This is only populated on the root layer, as attributes should not differ across layers. For nodes on layers that are not the root, attributes are retrieved via the layers reference to the root layer.

#visualAttributes

A set of attributes that describe the layer and how it should be rendered visually, stored here so they are not recalculated, and persist across the layer no matter where it is displayed. Eg, nodeMinSize: 10, nodeMaxSize: 10, nodeSizeAttribute: 'x'

LinkMaps

if there are more than 8.3 million nodes, multiple maps are needed due to JS limits

#nodeMaps

if there are more than 8.3 million links , multiple maps are needed due to JS limits, This is used to store the multiple maps required

`#nodeAttributeMaps`

if there are more than 8.3 million links , multiple maps are needed due to JS limits, This is used to store the multiple maps required

`#linkAttributeMaps`

if there are more than 8.3 million links , multiple maps are needed due to JS limits, This is used to store the multiple maps required.

`#nodeCentralityNames = {};`

A JS object containing the names (as keys) of all node centralities that exist for this layer.

`#linkCentralityNames = {};`

A JS object containing the names (as keys) of all node centralities that exist for this layer.

`#nodeDatePropertyName`

The name of the date attribute that is stored in the node date property on the layer nodes

`#linkDatePropertyName`

The name of the date attribute that is stored in the node date property on the layer nodes

`#connectedComponents ;`

an object that maps a component ID to nodes id arrays, specifying the nodes in a connected component

`#nodeCentralities;`

A JS object look up table that maps the node UID to an JS object containing name value pairs of attributes. This is populated for each layer root layer, as centralities differ across layers. Centralities can be retrieved for a node, which then refers to this look up via its link reference,

`#linkCentralities`

A JS object look up table that maps the link UID to an JS object containing name value pairs of attributes. This is populated for each layer root layer, as centralities differ across layers. Centralities can be retrieved for a node, which then refers to this look up via its link reference.