Dynamic webpage design for user interfaces in a Virtual Reality data exploration platform/frontend

Josef Hackl

Contents

1 Introduction / Task description

VRNetzer is a Virtual Reality Framework for Network Visualization and Analytics. It can be used for the visual, interactive exploration of large networks.[1] Currently, it consists of the following components:

The VR Module uses the Unreal 4 Engine and is responsible for rendering the network in 3D. For data science tasks the analytics module is reponsible. The input data is retrieved from a database (currently, MySQL). The UI module processes the input of the user, for instance selecting nodes of the network. It has to be able to send and receive calls from the VR module.

In the project presented in the following, the possibilities for a new user interface have been explored. The modified frontend should be based on python to allow easy customization of new user interfaces. However, the crucial goal is to add multiplayer functionality, i.e. it should be possible to share data between the clients. As an example in this specific context, a user should be able to share specific elements, e.g. selected nodes of the network, with other users.

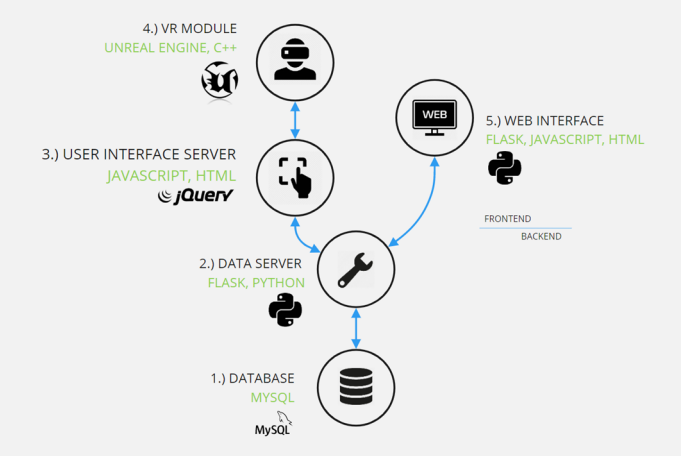


Figure 1: VRNetzer components[1]

2 Current user interface

The implementation of a new user interface could also modify the components set-up: If both analysis and user interaction tasks use python, it could be possible to merge this module an run all these tasks on one server.

The current user interface relies on websites using javascript. Different layouts can be selected and several 2D projections of the network can be visualized.

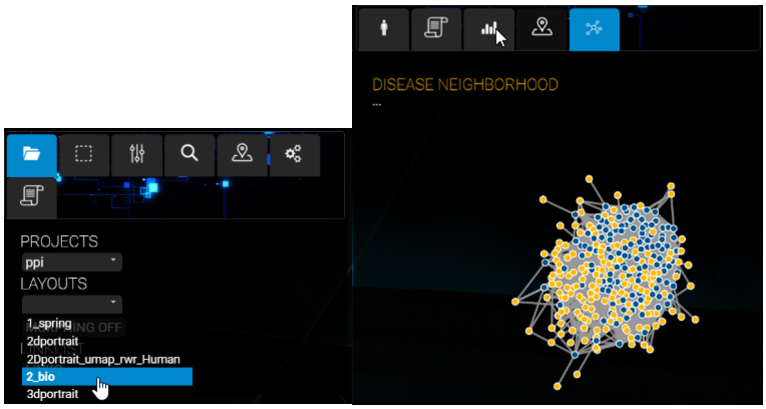


Figure 2: Current user interface [2]

3 Used libraries / frameworks

3.1 Flask

Currently, the data server runs on Flask, a micro web framework written in Python. This could also be the basis for the new UI module.

For implementing bidirectional communication, which is necessary for multiplayer-like tasks, libraries like Websockets or SocketIO will be necessary. In contrast to usual HTTP-queries, websockets-bassed communication can be initiated like Post-Requests from client, but the connection stays open and multiple clients can connect to one websocket. The server itself can send messages to some or all clients too.

For a Flask server, the usage of socketio is straightforward via Flask-SocketIO.

3.2 Dash

Dash is a Python framework which works on top of flask. Internally, it uses javascript (for the drawing of elements the react and plotly libraries are used) , the layout of the dash application relies on HTML.

However, for the creation of a new application no javascript is necessary (examples are shown below), so it should be relatively easy to create new user interfaces.

Dash offers various elements which are of interest for data and network analysis:

For instance, data tables (dash_table) can be used to present computed values like centrality measures etc. Other elements are sliders (dcc.slider) and text elements (e.g. dcc.textarea).

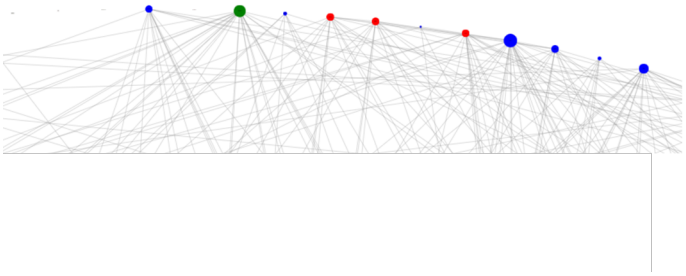
Typical HTML-elements (Button, Table, div,...) are also directly available.

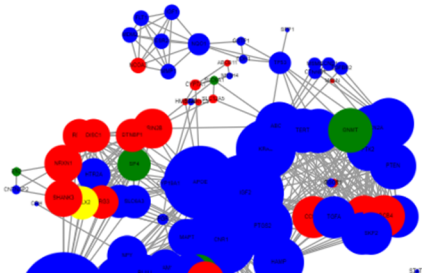
For multiplayer-related tasks, the dcc.store-element can be of interest: Whereas all clients have access to global variables - and therefore modification on one

client modifies the variable on all clients - data stored in the dcc.store is specific for each client.

Furthermore, it has full plotly-functionality. This makes the import and plotting of networkx-graphs or similar objects easily possible. Also histograms or scatterplots can easily be drawn.

Of special interest for the task here discussed is the cyto.Cytoscape-package which offers the possibility to draw interactive (positions of nodes can be changed, etc.) cytoscape graphs within the dash framework. Various layouts, for instance force-directed ones, are available for visualization.





3.3 Dash_devices

A different, comfortable way of allowing multitasking is the usage of the package dash_devices. It has a similar functionality as Dash (in fact, all dash-elements are supported), but uses quart, a flask reimplementation based on asyncio, instead of flask as underlying server.

It has built-in WebSocket support. Therefore, multicasting is possible without using any additional socketio-library.

These functions facilitates sharing of specific, even more complex elements drastically, e.g. show to some users statistics of a node when selected. Even customizable sharing of tabs is easily possible : If one user changes the tab, the other users see this recently chosen tab (or not, depending on the settings of the client stored in the dcc.store element)

Two additional functions are especially important: Besides usual callbacks, via callback_shared shared callbacks can be triggered. These callbacks work the same way as the dash callbacks, however the output is updated on all clients. They have the same mechanism like a regular callback, but the output is transferred to all clients via Websockets.

```
@app.callback_shared(Output('shared_slider_output', 'children'), [Input('shared_slider', 'value')])
def func(value):
    return value

@app.callback(Output('regular_slider_output', 'children'), [Input('regular_slider', 'value')])
def func(value):
    return value
```

Figure 3: Shared callbacks (dash_devices)

The function push_mods can be used to send the current state of an element is sent to all clients (outside of the concrete callback) or to a specific client. Modifications of the element can trigger a new callback, i.e. the clients are updated accordingly.

3.4 Custom elements

Dash also supports the creation of new modules, called custom elements. Similar to HTML custom elements, one can write own dash component using javascript, specifically react.js. Dash converts the React components into python code. After creating a new package, the model can be imported into the dash-app. In principle, such custom elements can be combined with socket-io or a similar library for communication between the clients and the server.

4 Example implementation

An example application exploring the functionality of flask and dash for network analysis and visualization has been implemented. This implementation was further modified to compare the various possibilities of introducing multitasking functionality.

Examples are the sharing of slider or of the status of a graph, e.g. one user selects a node which will be highlighted for all users.

4.1 Example input

As example data, a data set from DisGeNET has been used. DisGeNET collects data about associations between genes and diseases.[3, 4, 5]

Using this data, the example app visualizes the disease network, consisting of all diseases with common genes, the disease gene network of all genes which cause the same diseases, and the bipartite graph which connects diseases and associated genes, i.e. the diseasome [6],

Fig. 4 shows a selection of the entries after uploading to the dash app:

In this data set, genes are describes by 'geneSymbol' and 'geneId', diseases by 'diseaseName' and 'diseaseId'. The class of the disease and the information source for the association are also given.

DSI indicates the Disease Specificity Index. A gene which is only associated to one disease has a DSI of 1, whereas a gene which is associated to many diseases has a low DSI.

The Disease Pleiotropy Index (DPI) is also a number between 0 and 1 indicating the number of disease classes a gene is associated to. If a gene is associated with many different disease classes, it has a higher DPI.

The score-column shows the DisGeNET score which is higher for gene-disease-associations with many sources confirming its presence.

geneId	geneSymbol	DSI	DPI	diseaseId	diseaseName	diseaseType	diseaseClass	diseaseSemanticType	score	EI	YearInitial	YearFinal	NofPmids	NofSnps	source
102	ADAM10	489.0	846.0	C0002395	Alzheimer's Disease	disease	C10;F03	Disease or Syndrome	0.7	986.0	2000.0	2019.0	1	1	CTD_human
111	ADCY5	617.0	577.0	C0011860	Diabetes Mellitus, Non-Insulin-Dependent	disease	C18;C19	Disease or Syndrome	0.7	947.0	2002.0	2019.0	2	3	CTD_human
181	AGRP	488.0	769.0	C0028754	Obesity	disease	C23;C18	Disease or Syndrome	0.7	938.0	1997.0	2019.0	0	0	CTD_human
324	APC	373.0	962.0	C0376358	Malignant neoplasm of prostate	disease	C04;C12	Neoplastic Process	0.7	0.96	1994.0	2019.0	2	0	CTD_human
324	APC	373.0	962.0	C2239176	Liver carcinoma	disease	C06;C04	Neoplastic Process	0.7	963.0	1993.0	2019.0	0	11	CTD_human
338	APOB	453.0	808.0	C0015695	Fatty Liver	disease	C06	Disease or Syndrome	0.7	1.0	2001.0	2018.0	2	0	CTD_human
348	APOE	338.0	962.0	C0002395	Alzheimer's Disease	disease	C10;F03	Disease or Syndrome	0.7	946.0	1991.0	2020.0	9	5	CTD_human
348	APOE	338.0	962.0	C0010068	Coronary heart disease	disease	C14	Disease or Syndrome	0.7	966.0	1983.0	2019.0	1	1	CTD_human
348	APOE	338.0	962.0	C0020443	Hypercholesterolemia	disease	C18	Disease or Syndrome	0.7	957.0	1985.0	2019.0	4	0	CTD_human
596	BCL2	291.0	885.0	C0011860	Diabetes Mellitus, Non-Insulin-Dependent	disease	C18;C19	Disease or Syndrome	0.67	1.0	2006.0	2020.0	1	1	CTD_human

Figure 4: DisGeNET data

4.2 Dash usage

One can either start a flask server and subsequently add dash (fig. 5) or simply instantiate the dash app and, if desired, afterward access the created flask server .

```
app = Flask(__name__)
dash_app = Dash(__name__, server = app, routes_pathname_prefix='/dashboard/' )
```

Figure 5: Flask initialization

There is also the possibility of merging multiple applications using the DispatcherMiddleware of the werkzeug-library (fig. 6).

In one of the example implementations, the Flask application was used as entry point from which the additional Dash application can be entered.

```
#creating wsgi_app containing flask and dash
from werkzeug.exceptions import NotFound
app.wsgi_app = DispatcherMiddleware(app.wsgi_app, {
    "/prefix": app,
     '/dash1': dash_app.server,
    # '/dash2': dash_app2.server
})
```

Figure 6: Combination of applications

Setting up socket-io is straightforward (fig. 7).

```
socketio = SocketIO(app, manage_session=False)
if __name__ == '__main__':
    # socketio.run(app, host="127.0.0.1", port=5000)
     socketio.run(app)
```

Figure 7: socket-io initialization

4.2.1 Layout

The central element of every dash-app is the layout (app.layout). It consists of all elements which appear on the web site.

```
app.layout = html.Div([
    dcc.Slider(id="scorethresholdslider", min=0, max=1, step = 0.05, value=0.6,
               marks={0: '0', 1: '1'}),
    html.Div(id='thresh'),
    dcc.RadioItems(
    id='cytographlayout',
    options=[

        {'label': 'preset', 'value': 'preset'},
        {'label': 'circle', 'value': 'circle'},
        {'label': 'cose', 'value': 'cose'},
    ],
    value='preset'
    ),
    html.Button('Share Graph 1', id='sharecyto', n_clicks_timestamp=0),
    html.Button('Share Graph 2', id='sharecyto2', n_clicks_timestamp=0),
    cyto.Cytoscape(
        id='cytoscapenetw',
        style={'width': '60%', 'height': '600px'},
        elements=elements,
        layout={
            'name': 'circle'
        }
    ),
     html.P(id='placeholder1'),
     html.Div(id='placeholder2'),
    dcc.Store(id='cytostore', data = json.dumps(elements)),
    dcc.Textarea(
        id='cytostoretext',
        value="start",
        style={'width': '100%', 'height': 100},
    ),
])
```

Figure 8: Dash Layout

If one wants to create a multi-tab app, the layout itself contains the present tabs, whereas the content of the tabs is stored in individual elements.

```
dash_app.layout = html.Div([
    sockettest.sockettest(),
    html.H1('Dash Graph multiple tabs', className="app-headerf" ),
    dcc.Store(id='memory-tab2'),
    dcc.Tabs(className="app-header", id="tabs-example", value='tab-1-example', children=[
        dcc.Tab(id="tab-1", label='Data upload', value='tab-1-example'),
        dcc.Tab(id="tab-2", label='Gene Network', value='tab-2-example'),
        dcc.Tab(id="tab-3", label='Bipartite network', value='tab-3-example'),
        dcc.Tab(id="tab-4", label='Disease network', value='tab-4-example'),
        dcc.Tab(id="tab-5", label='Statistics / pick node before', value='tab-5-example'),
        dcc.Tab(id="tab-6", label='Diversity-ubiquity plot', value='tab-6-example'),
    ]),
    html.Div(
    id='tabs-content-example',
             children = tab1,className="app-headerf2")
],className="app-headerf",)
```

Figure 9: Dash Layout Multi-tabs

4.2.2 Callbacks

In Dash, the contents are exclusively updated via callback functions. If an input property changes, the callback function is triggered. Subsequently, the changed output of one callback can be input for another callback

4.2.3 Dash_devices

As already mentioned, dash_devices offers the possibility to use shared callbacks which allow to update the output of the callback on all clients (without using global variables).

A similar behaviour outside a specific callback can be triggered via push_mods. This function can also be used for sharing tab content, as shown in fig. 12.

```
@app.callback([Output('placeholder2', 'children'),
               Output('cytostore', 'data')],
                 [Input('sharecyto2', 'n_clicks_timestamp')],
                 [State('cytoscapenetw', 'elements'),
                 State('cytoscapenetw', 'tapNode')]
                 )
def save_graph_state(sharebutton, elementsn, tapnode):
```

Figure 10: Dash Callback

Figure 11: Dash Tabs

```python
#callback for switching tabs
@dash_app.callback(Output('tabs-content-example', 'children'),
              [Input('tabs-example', 'value')],
              [State('sharetabsstore', 'data')]
              )
def render_content(tab, sharetabs):

    if (sharetabs == "Y"):
        if tab == 'tab-1-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab1}})
            return tab1
        elif tab == 'tab-2-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab2}})
            return tab2
        elif tab == 'tab-3-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab3}})
            return tab3
        elif tab == 'tab-4-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab4}})
            return tab4
        elif tab == 'tab-5-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab5}})
            print("return tab5")
            return tab5
        elif tab == 'tab-6-example':
            dash_app.push_mods({'tabs-content-example': {'children': tab6}})
            return tab6
    else:
        if tab == 'tab-1-example':
            return tab1
        elif tab == 'tab-2-example':
            return tab2
```

Figure 12: Dash_devices Tab Multicasting

4.3 Different implementations

Several versions of the app have been implemented.

In one version, a Flask server with login and chat modules is set up. The dash up can be accessed via a button on the chat interface.

Multicasting possibilities, especially sharing a slider and highlighting a node for all users, using directly socket-io, a custom websockets-component and dash_devices have been tried out.

Basic usage of SocketIO is also in Dash easily possible (fig.13). However, one problem occurs: How to trigger a new callback if data is updated (without polling)? If only socket-io is used, the appearance of the elements has to be changed by accessing the underlying HTML structure via javascript (fig. 14).

Other techniques for sharing are to implement socketIO via a dash custom element or use dash_devices.

```python
from flask_socketio import SocketIO
app = dash.Dash(__name__)
server = app.server
server.debug = False
socketio = SocketIO(server)
```

Figure 13: Dash SocketIO initialization

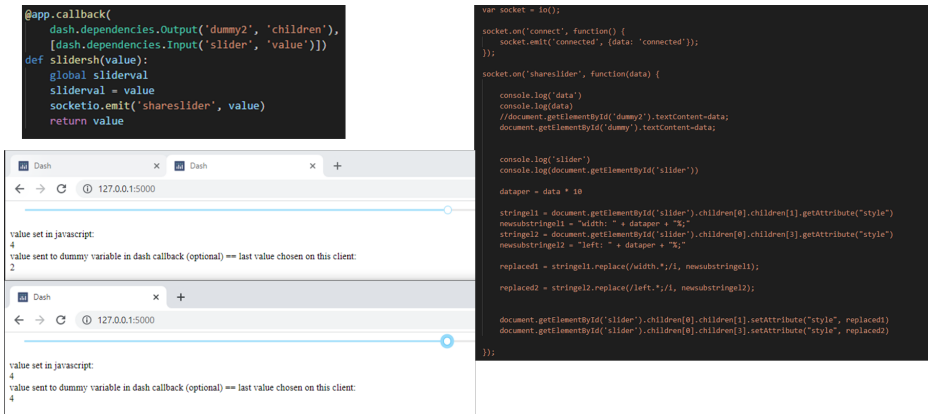


Figure 14: SocketIO slider

4.4 Individual tabs

4.4.1 Data upload

Data can be uploaded via a dash-upload-module. If one wants to use the DisGeNET-data, the diseasenet-option has to be selected (by clicking on the corresponding radio-items-button). Using a slider component, one can set a threshold for the minimum score of the imported gene-disease-association. After upload, the data is shown in a table-element.

4.4.2 Gene network / Disease network

These tabs show a representation of the gene and disease network, respectively. The graphs are cytoscape elements from dash. Fig. 15 shows example graphs. In the gene network, the size of the nodes indicate its degree, the color the DPI of the gene.

One of the example implementations based on dash_devices uses push_mods to highlight a selected node for all users (fig. 16).

4.4.3 Bipartite network

Gene/Protein and disease data can be combined to a bipartite plot showing the associations between genes and diseases.

In this case, a networkx-plot is created and plotted as a dash-graph object using plotly.

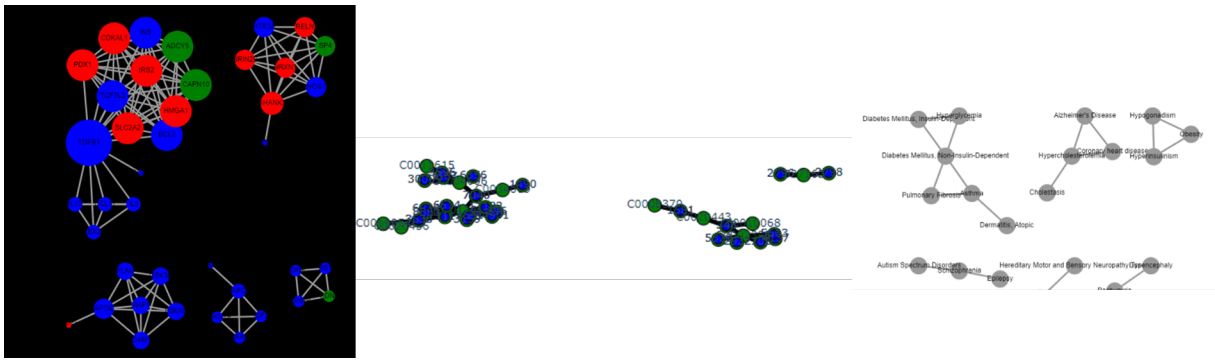


Figure 15: left: gene network / Cytoscape; middle: bipartite graph / networkx; right: disease network / Cytoscape

```
@self.app.callback(None, [Input('cytoscapenetw', 'tapNode')],[State('cytoscapenetw', 'elements')])
def funcgraph(tapdata, elementsn):
    for el in elementsn:
        if "color" in el['data']:
            if el['data']['id'] == tapdata['data']['id']:
                if el['data']['color'] == 'red':
                    el['data']['color'] = 'green'
                elif el['data']['color'] == 'green':
                    el['data']['color'] = 'yellow'
                elif el['data']['color'] == 'yellow':
                    el['data']['color'] = 'red'
    self.app.push_mods({
        'cytoscapenetw': {'elements': elementsn}
    })
    return None
```

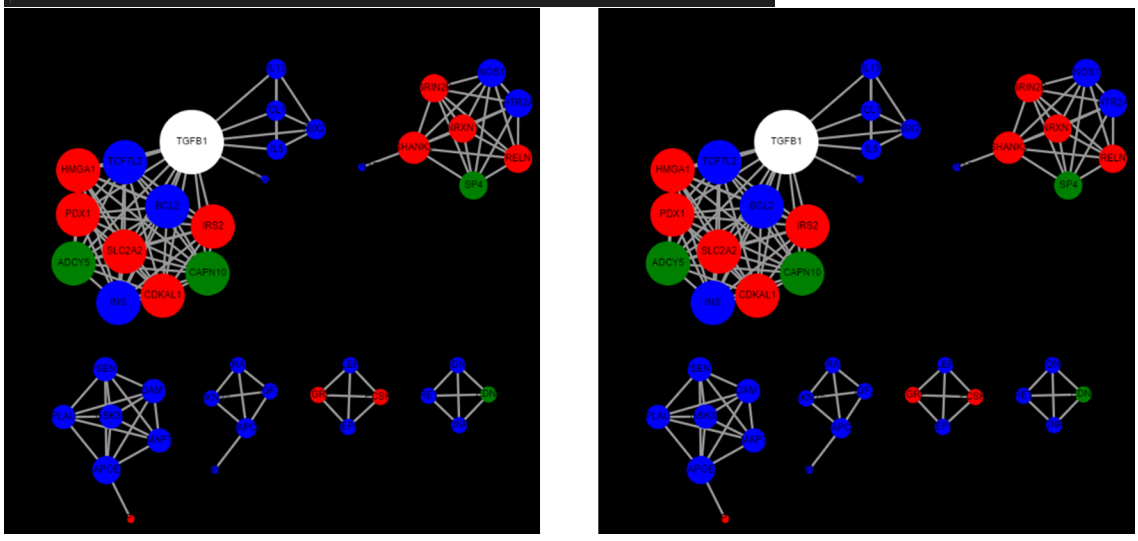


Figure 16: multicasting for node selection

4.4.4 Statistics

The 'statistics'-tab can be accessed after a node in the plotly-graphs of the disease- or the gene-network has been picked.

It offers additional information about the selected node. Its id and name is shown. Furthermore, a dash-datatable is created which contains calculations of statistical properties of the node in the gene- or disease-projection as well as in the bipartite gene-disease-graph (fig.17).

The degree-column simply states the number of nodes a selected node is connected to, i.e. the number of edges. The first row has the values for the projection, e.g. for a disease 'degree' gives the number of diseases with which it is connected, i.e. the number of disease which are caused by the same genes.

The centrality measures are computed using networkx.

In networkx, degree centrality is calculated as the fraction of nodes a specific node is connected to.(so the value is between 0 and 1).

Closeness centrality calculates the reciprocal value of the average lengths of the shortest paths between the selected node and all other nodes. A higher value indicates that the average shortest path distance is lower, therefore the node is closer to all other nodes.

Betweenness centrality indicates how many shortest paths between other nodes pass through the selected node.

Finally, eigenvector centrality determines the centrality of a node based on the centrality of the nodes in its neighbourhood by solving an according eigenvalue problem.[7]

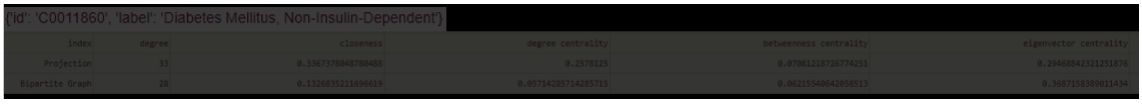


Figure 17: statistics of a selected disease

Histograms visualize the distribution of genes and diseases. The number of bins can be adjusted (fig. 18).

```python
#callback for creating a degree distribution histogram
@dash_app.callback_shared(
    Output("histo1", "figure"),
    [Input("binnumber", "value"),
     Input("histoselectgraph", "value")])
def display_histogram(binnumber, histoselectgraph):
    global disdisdegrees
    global genegenedegrees

    if histoselectgraph == 'D':
        disdisdegreedata = []
        for elem in disdisdegrees:
            disdisdegreedata.append(elem[1])

        fig = px.histogram(disdisdegreedata, nbins = binnumber)
    else:
        genegenedegreedata = []
        for elem in genegenedegrees:
            genegenedegreedata.append(elem[1])

        fig = px.histogram(genegenedegreedata, nbins = binnumber)

    return fig
```

Figure 18: Histograms of the degree distribution

4.4.5 Diversity-ubiquity plot

As an example of a scatterplot, a diversity-ubiquity plot showing the relation between the number of diseases a specific gene causes and the average number of genes which also cause this disease was implemented. Its creation can be triggered by clicking on a button (fig. 19).

```python
#create diversity-ubiquity plot
@dash_app.callback(
    Output('ubiqugraph', 'figure'),
    Input('showubiqu', 'n_clicks_timestamp'))
def update_stat_table(click):
    bipartdegrees = nx.degree(nxgraph1)
    #number of diseases each gene causes
    bipartdegreesgene = { genekey: bipartdegrees[genekey] for genekey in nxgraphdisease.nodes }
    bipartdegreesgdis = { diskey: bipartdegrees[diskey] for diskey in nxgraphdisease.nodes }
    plotxy = []

    #calculate ubiquity for genes (i.e. average number of genes that also cause the disease)
    for elem in nxgraphdisease.nodes:
        disnumbers = []
        for dis in nxgraph1[elem]:
            disnumber = len(nxgraph1[dis])
            disnumbers.append(disnumber)
        avrgdis = np.mean(disnumbers)
        plotxy.append([elem, bipartdegreesgene[elem], avrgdis])
    plotxydf = pd.DataFrame(plotxy, columns=['elem', 'diversity', 'ubiquity'])
    fig = px.scatter(plotxydf, x="diversity", y="ubiquity", hover_data=['elem'], title="diversity-ubiquity plot", labels = {"diversity": "diversi
    return fig
```

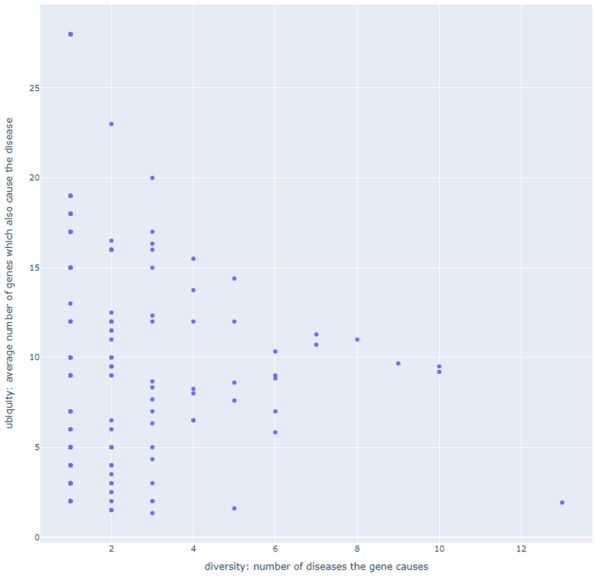


Figure 19: diversity-ubiquity plot

5 Conclusion

The various ways of implementing multiplayer functionality have different advantages; basically, the Flask-SocketIO-functionality allows all forms of multicasting on a Flask server. However, the features of Dash can drastically facilitate the creation of new, interactive user interfaces and are especially well-suited for network analysis and visualization tasks.

The best solution certainly depends on what has to be shared: only variable, sliders, or even graphs with interactive elements (for instance, selecting a node,...).

For Dash, if only data has to be shared, usage of global variables in combination with dcc.store-elements seems to be sufficient. For more complex cases, socketIO works fine, but dash_devices provide the most comfortable and effective ways of interactive, multicasted modification of elements (dash_devices is intended for exactly this use case). However, one has to rely on a not so well-known, perhaps in the future less supported package (but, on the other hand, the quart-asyncio architecture is more recent and possibly more efficient).

The other factor is the intricacy and difficulty of creating new interfaces or elements: Dash custom elements offer a highly customizable way to create new elements with socketio-support, but its implementation is rather complicated, whereas dash_devices do not cause much additional complexity compared to usual, non-shared dash callbacks.

App availability

The code for the app mentioned in the report can be found in the following github-repositories: https://github.com/jalhackl/websockets_plus_dash_disgenet , https://github.com/jalhackl/sockettest_plus_dash_disgenet , https://github.com/jalhackl/quart_disgenet.

Furthermore, some 'minimal examples' for the usage of socketio and dash_devices have been created.

References

[1] Pirch, S., Müller, F., Iofinova, E. et al.: The VRNetzer platform enables interactive network analysis in Virtual Reality. Nat Commun 12, 2432 (2021).

[2] https://github.com/menchelab/VRNetzer

[3] Piñero, J., Queralt-Rosinach, N., Bravo, À., Deu-Pons, J., Bauer-Mehren, A., Baron, M., Sanz, F., & Furlong, L. I. (2015): DisGeNET: a discovery platform for the dynamical exploration of human diseases and their genes. Database : the journal of biological databases and curation, 2015.

[4] Piñero, J., Bravo À., Queralt-Rosinach, N., Gutiérrez-Sacristán, A., Deu-Pons, J., Centeno, E., García-García, J., Sanz, F., Furlong, L.I.: DisGeNET: a comprehensive platform integrating information on human disease-associated genes and variants, Nucleic Acids Research, Vol. 45, Issue D1, 2017, D833–D839

[5] Piñero, J., Ramírez-Anguita, J.M., Saüch-Pitarch, J., Ronzano, F., Centeno, E., Sanz, F., Furlong, L.I.: The DisGeNET knowledge platform for disease genomics: 2019 update, Nucleic Acids Research, Vol. 48, Issue D1, 2020, D845–D85

[6] Goh, K., Cusick, M.E., Valle, D., Childs, B., Vidal, M., Barabási, A.-L.: The human disease network. Proceedings of the National Academy of Sciences May 2007, 104 (21)

[7] Newman, Mark. Networks: An Introduction. Oxford: Oxford University Press, 2010. Oxford Scholarship Online, 2010.