Source code explanation:

On a day-to-day basis, device monitoring is a key element in IT operations. It is important to be able to know when device connections have an outage, as well as the frequency and duration of these outages.

The goal in this project is to have a visualization application that gives the flexibility to view this outage data and create reports based on data filtering – we need to generate graphs that display the counts of all outage events (when a link goes down or when it is restored) based on any desired timeframe and device links that we'd like to view.

The front-end of this project uses Bokeh, a python library, to create an interactive plot that visualizes the data we want to see. The foundation for our plot relies on a data structure unique to Bokeh: the Column Data Source (CDS). This object can be considered a collection of sequences of data that each have their own individual column names.

The backend of this project is dependent on Pandas, another python library that is used for data manipulation and analysis. All our data will revolve around and be contained in a particular Pandas data structure called a DataFrame.

Required files

We begin by parsing and preparing the data in a way that is digestible for our Bokeh application in our parse_data.py script. Before this, we need to retrieve the information we need from two files, titled data.csv and dictionary.txt.

data.csv

This file should consist of all event data. Each line in this file must consist of the format:

Device,Port,YYYY/MM/DD,hh:mm:ss,Value of Event (shown as 0 or 1)

Note: the value of event represents whether the device link went down (signified by a 0) or went back to normal (signified by a 1). These both correlate to the same outage but are considered individual events.

** if you are using Windows OS, opening this file by default will change the format of the data (Microsoft Excel is the default program that .csv files are opened with and it likes to modify the port column to a date) – beware of this if you'd like to view or modify the data.csv file.

dictionary.txt

Each line in this file must consist of the format:

Device,port,Label for Link

Note: These are essentially key-value pairs (in the sense that [Device,port] is a key to find the link label) and is used to label device links for a more user-friendly name.

Once the data we need is available, we continue to the parse_data.py script to parse and format our data, creating a DataFrame that will eventually be used by the Bokeh application.

Parsing data

Parse_data.py

The goal of this script is to take all the information from the previous files and return a DataFrame in this format:

Note that the leftmost column is a datetime index.

	Link 1	Link 2	Link 3	Link 4
YYYY-MM-DD	Count of events at this timestamp	Count of events at this timestamp	Count of events at this timestamp	Count of events at this timestamp
YYYY-MM-DD	" "	" "	" "	" "
YYYY-MM-DD	" "	" "	" "	" "
YYYY-MM-DD	" "	" "	" "	" "

Ultimately, there are 6 steps taken to create this output (with diagrams for illustrative purposes):

1. Read data from data.csv and parse them into a DataFrame using Pandas – the default columns will be 'Timestamp', 'Device', 'Port', and 'Event'

Index (default)	Timestamp	Device	Port	Event
0	YYYY-MM-DD hh:mm:ss	Device-Name	#/#	0 (or 1)
1	' '	' '	' '	' '
2	' '	' '	' '	' '

2. Concatenate the Port column into the Device column and remove the entire Port column (as it's no longer needed)

Index (default)	Timestamp	Device	Event
0	YYYY-MM-DD hh:mm:ss	Device-Name,#/#	0 (or 1)
1	' '	' '	' '
2	' '	' '	' '

3. Count duplicates of Device values in the same timestamp and put that summation into a new column titled 'Count'

Index (default)	**Timestamp**	**Device**	**Count**
0	YYYY-MM-DD hh:mm:ss	Device-Name,#/#	Duplicate count
1	‘ ‘	‘ ‘	‘ ‘
2	‘ ‘	‘ ‘	‘ ‘

4. Pivot the DataFrame so that the Device column values become their own column headers and the Timestamp column becomes the DataFrame’s new index

Index (used to be Timestamps column)	**Device-Name,#/#**	**Device-Name,#/#**	**Device-Name,#/#**
YYYY-MM-DD hh:mm:ss	Count of events at this timestamp	Count of events at this timestamp	Count of events at this timestamp
‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘ ‘	‘ ‘	‘ ‘

5. Count all events that occur for device links based on day*

Index (used to be Timestamps column)	**Device-Name,#/#**	**Device-Name,#/#**	**Device-Name,#/#**
YYYY-MM-DD	Count of events at this day	Count of events at this day	Count of events at this day
‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘ ‘	‘ ‘	‘ ‘

6. Rename the column headers based on the labels we received from dictionary.txt

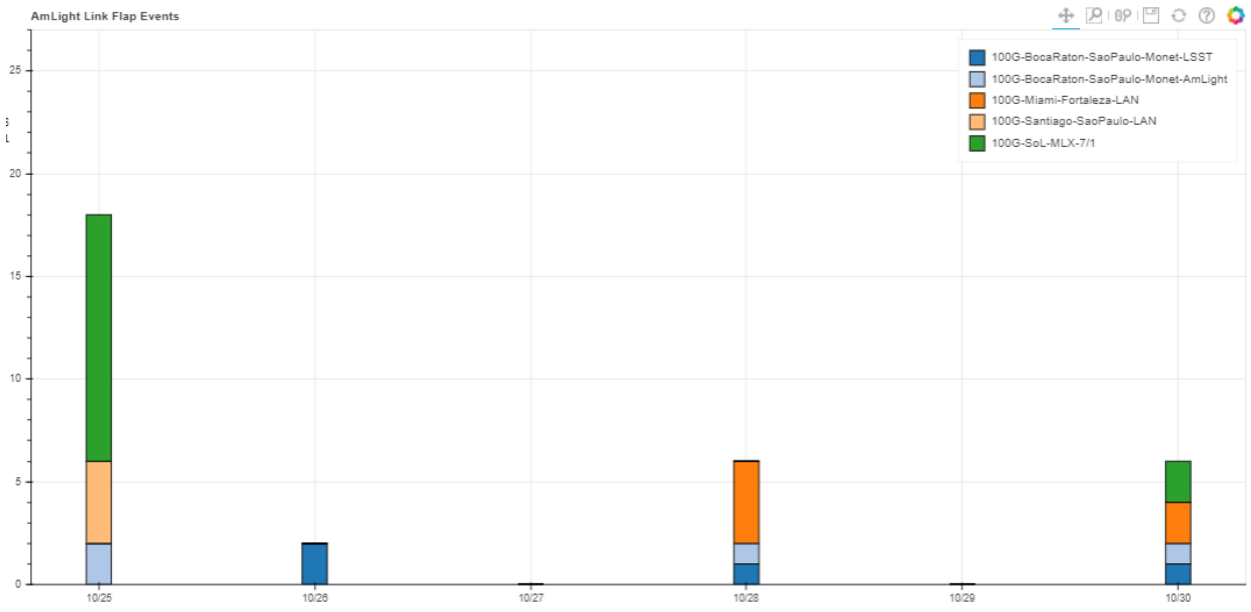
Index (used to be Timestamps column)	**Link label**	**Link label**	**Link label**
YYYY-MM-DD	Count of events at this day	Count of events at this day	Count of events at this day
‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘ ‘	‘ ‘	‘ ‘

* As mentioned in Step 5, all the event data is grouped by the day they occurred. Later on the desired grouping may be based on hour or even minute, so it’s not inherently necessary to group by day here. This grouping can be done in this script, or it can be done separately in the visualizer.py application – ultimately, that decision is left to the developer.

Now that the DataFrame is created, it is called by and passed to visualizer.py, which is what creates our Bokeh application.

Bokeh application

The entirety of the Bokeh server application is created by visualizer.py. It takes the DataFrame that was created in parse_data.py and creates a Column Data Source from that DataFrame. After the plot and widgets are created, a callback is made to update the data that is displayed on the plot upon user interaction with the widgets. For the widgets to update dynamically, the application runs on a Bokeh server, which connects frontend events to the running python code. The graph that is created is a vertical stacked bar chart, where each link is considered a 'stacker' on each bar. To illustrate, here is a screenshot from the running application:



As shown in the legend, each link that has events on a particular day is stacked upon each other. Note that on the y-axis we have the total counts of events that occurred, and the x-axis displays the day that the events occurred.

In Bokeh's ColumnDataSource, each column is considered an individual "stack", and the index (which would be a series of timestamp values in this case) is automatically made to be the plot's x-axis. Note that in the application, the plot is explicitly given a datetime axis to accept the timestamps in the DataFrame's index.

For another example of a stacked bar chart in Bokeh, refer to this example shown on the Bokeh website.

Filtering the data

Calls to filter the data will be made via widgets on the application. Currently, there are three widgets: the DatePicker, CheckBoxGroup, and Button widgets. Their purpose (and names) are as follows:

- DatePicker: There are currently two in use, named daterange_start and daterange_end.
    - Upon clicking, the user can choose any specific date range of the data they'd like to see. Daterange_start and daterange_end will choose the start and end dates, respectively.

- CheckBoxGroup: Only one exists and is named link_checkbox.
    - This widget consists of all links in the data. Clicking one of the checkboxes will either add or filter out its link from the graph. All checkboxes are enabled by default.
- Button: One named select_all.
    - This button enables all checkboxes in the CheckBoxGroup widget at once.

Upon interaction with these widgets, an update function is called that takes all current values of the widgets to filter data out all at once.

Update approach

To filter the data, a new DataFrame (named temp_df in the code) is created from a subset of the main DataFrame – first, it takes all columns that are considered 'active' (or checked) in the CheckBoxGroup widget as well as their values and indices. Then, the values from both DatePicker widgets are used to filter temp_df – made possible by the DataFrame's datetime index. Then, the ColumnDataSource is updated.

To update the CDS using a new DataFrame, the following approach was used:

1. A "refresh" of the current columns in the original CDS, providing the new columns that are given from the CheckBoxGroup
2. A new CDS is created from temp_df
3. The original CDS is updated directly from the data property of the new CDS

The reasoning behind this approach is that, while Bokeh allows the creation of multiple, interactive plots, these plots are only able to use the CDS that it was created with at initialization. The best way to update the data would be to modify the CDS that is in use.

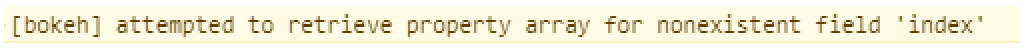
Issues/reservations with the current code

Axis label bug

There is a bug where the tick labels for both the x and y axis sometimes disappear after selecting a filter from one of the widgets on the left column. These labels return once one clicks on the 'reset plot' tool in the upper right, or by panning the plot with the mouse (using the move tool, which is enabled by default). The labels do not disappear again unless the data is filtered after a plot reset – so the best case would be to simply pan the graph once before interacting with the widgets.

Potential reason:

When the bug occurs, the console outputs an error:



```
[bokeh] attempted to retrieve property array for nonexistent field 'index'
```

This shows that the field 'index' does not exist in the column data source once the data in the column data source is changed. This may be an issue with updating the ColumnDataSource, which updates as follows (snippet from visualizer.py):

```
new_source = ColumnDataSource(new_dataframe)
source.data.update(new_source.data)
```

Considering we are updating an existing ColumnDataSource, perhaps the bug may be since the original CDS is receiving modified version of a data frame (which implies it to have a different number of rows than the original).

A potential fix may lie with how the CDS reads this update.

Choosing Date Ranges

Originally, the intended widget to choose a date range for data filtering purposes was more similar to a calendar widget, shown below:

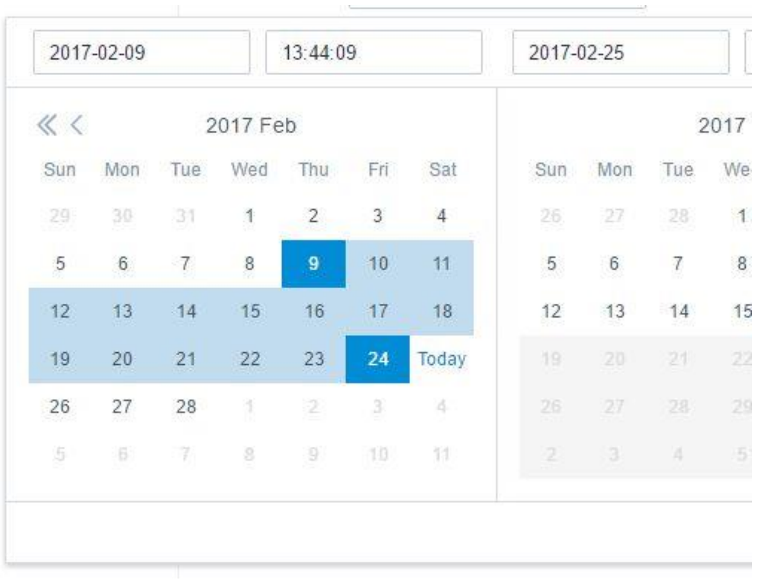


Image taken from https://github.com/bokeh/bokeh/issues/8441

It'd be more convenient for the user to have the ability to choose the entire range at-a-glance, rather than having to click between two widgets. Implementing two DatePicker widgets was a quick, simple way to have this option but there may be other methods to investigate.

Updating the Legend

Originally, the intention was to update the plot on the graph by clicking the links on the legend rather than using a CheckBoxGroup – however the only functionality that is supported on Bokeh at the moment is the option to hide glyphs. This causes an issue with the plot as the glyphs for the stackers don't automatically adjust to those that were hidden and it ultimately results in missing gaps in the middle of the bars in the graph.

Also, the current method to filter data cannot update the legend as well, as the legend is generated with the plot and cannot be modified. This leads to an approach that may be worth investigating, discussed in the next section.

Regenerating plots

Considering that the application is a Bokeh server, it may be worthwhile to look into multithreading in order to regenerate the plot. This can be an alternative to the current approach of updating the plot's ColumnDataSource and may resolve both the axis label and legend issues.

Resources

For tutorials and example programs, refer to Bokeh's user guide.

For examples of Bokeh applications, refer to the Bokeh gallery.

For discussions and announcements for the Bokeh project, refer to its discourse page.

For more information on the functions used with Pandas, refer to its API reference.

For miscellaneous questions regarding Bokeh and Pandas, the community at StackOverflow.com helped a lot with the creation and debugging of this project.