Task 8.2C - SIT225

Truong Khang Thinh Nguyen

September 12, 2024

1 Introduction

Updating continuous data on a dashboard for live monitoring presents a significant challenge. This process involves two key tasks: efficiently receiving data into a buffer and selectively updating the dashboard graphics with minimal changes. The goal is to provide a seamless experience just like streaming a movie, where a sequence of image frames creates a smooth sense of continuity. To address these challenges, I employ an alternative approach known as Partial Property Update. This method optimizes the way data is managed and displayed, ensuring that only the relevant portions of the dashboard are updated, thereby maintaining a fluid and responsive user experience while minimizing unnecessary data processing and visual disruptions.

2 Server Side Callback

The original method that used in the task 8.1P is the Server Side Callback. It includes sending a request from the client to the server to perform an operation or retrieve data. The server processes this request and sends back the updated data or a complete component, which then replaces or updates the existing content on the client side. Basically, it involves re-rendering or replacing an entire data structure or user interface component whenever an update is needed. This means that every time new data or changes are available, the entire component or page is reloaded or refreshed from the server or data source.
But this approach has some flaws

1. **Performance Overhead**: Full page or component reloads can be resource-intensive and may result in higher latency, as the entire content must be processed and sent from the server to the client.

2. **User Experience**: Frequent or large-scale updates can lead to visible disruptions, such as flickering or slow loading times, which can negatively impact the user experience.

3. **Increase Bandwidth**: Sending complete updates or full page reloads can consume more bandwidth compared to incremental updates, which might be an issue for users with limited or slow connections.

4. **Scalability Issues**: As the number of users or the volume of data increases, the server may face scalability challenges, leading to potential performance bottlenecks.

For those mentioned disadvantages, to address those problems, we will use Partial Property Update method.

3 Partial Property Update

The Partial Property Update method using the `patch()` function is a technique for updating only specific parts of a data structure or user interface component, rather than re-rendering the entire component, hence, save a lot of time in terms of updating the graph. This approach is particularly useful in scenarios involving live data streaming or real-time updates, as it allows for more efficient and responsive updates.

1. **Change Detection**: Identify which specific properties or elements of the component or data structure have changed since the last update. This involves tracking changes at a granular level, rather than at the component or page level.

2. **Apply Updates**: Use method `patch()` to apply only the identified changes. This function updates only the parts of the component or data structure that need to be refreshed, leaving the rest unchanged.

3. **Efficient Rendering**: By updating only the necessary parts, the Partial Property Update method minimizes the amount of rendering or data processing required. This leads to faster updates and a smoother user experience.

4. **Seamless Integration**: The incremental updates are integrated seamlessly into the existing component or data structure, maintaining continuity and reducing visual disruptions.

Let's look at the performance of those two approaches to verify whether the Partial Property Update surpasses the Full Refresh Approach

Approach	Time	Packet Size
Server Side Callback	50 - 67 ms	8.8 KB
Partial Property Update	2 - 4 ms	353 - 1.1 KB

Table 1: Comparison of Server-Side Callback and Partial Property Update in terms of performance

After running Python script using two different approaches, we can see that the performance of the Partial Property Update significantly surpasses the original one. Thus, we can firmly verify our enhanced alternative method in terms of making smooth changes.

4 Partial Property Update code

```
# Initialize Dash app
app = dash.Dash(__name__)

# Example initial plot
fig = go.Figure()
fig.add_trace(go.Scatter(x=[], y=[], mode='lines', name='X'))
fig.add_trace(go.Scatter(x=[], y=[], mode='lines', name='Y'))
fig.add_trace(go.Scatter(x=[], y=[], mode='lines', name='Z'))

app.layout = html.Div([
    dcc.Graph(figure=fig, id='live-update-graph'),
    dcc.Interval(id='interval-component', interval=1000, n_intervals=0)  # Update every second
])

```

```
@app.callback(
    Output('live-update-graph', 'figure'),
    Input('interval-component', 'n_intervals')
)
def update_graph(n_intervals):
    global plot_data
    if plot_data:
        # Convert the plot_data to structured format for updating the graph
        reformatted_data = {
            'timestamp': [row[1] for row in plot_data],
            'x': [row[2] for row in plot_data],
            'y': [row[3] for row in plot_data],
            'z': [row[4] for row in plot_data]
        }
        # Update the graph with the new data
        patch = Patch()
        patch["data"] = [
            {"x": reformatted_data['timestamp'], "y": reformatted_data['x']},
            {"x": reformatted_data['timestamp'], "y": reformatted_data['y']},
            {"x": reformatted_data['timestamp'], "y": reformatted_data['z']}
        ]
        return patch
    return go.Figure()
```

5 API Function

```
# API Function
def update_smoothly(data=None, x_name=None, y_name=None, dash_app=None,
                    graph_id=None, interval_id=None):
    """
    Updates the graph in a Dash application smoothly using incremental
       updates.

    Parameters:
    - data (list of lists or DataFrame): Data to be used for updating the
       graph.
    - x_name (str): The column name in `data` to be used for the x-axis.
    - y_name (list of str): List of column names in `data` to be used for the
        y-axis.
    - dash_app (Dash): The Dash application instance where the callback will
       be registered.
    - graph_id (str): The ID of the graph component to be updated.
    - interval_id (str): The ID of the interval component triggering the
       updates.

    Raises:
    - ValueError: If any of the provided arguments are None.
    """

    # Define a dictionary of arguments for validation
    arguments = {
        "data": data,
        "x_name": x_name,
        "y_name": y_name,
        "dash_app": dash_app,
        "graph_id": graph_id,
        "interval_id": interval_id
    }

```

```
    # Validate that none of the arguments are None
    for arg_name, arg_value in arguments.items():
        if arg_value is None:
            raise ValueError(f"{arg_name} mustn't be None.")

    @dash_app.callback(
        Output(graph_id, 'figure'),
        Input(interval_id, 'n_intervals')
    )
    def update_graph(n_intervals):
        """
        Callback function to update the graph with new data.

        Parameters:
        - n_intervals (int): The number of intervals since the last update (
           used to trigger the update).

        Returns:
        - dict: A dictionary representing the updated figure with new data.
        """
        # Convert data to a DataFrame for easier manipulation
        df = pd.DataFrame(data, columns=['Index', 'Timestamp', 'X', 'Y', 'Z'
           ])

        # Initialize a Patch object to handle incremental updates
        patch = Patch()

        # Update the graph data with the provided y_names
        for i in range(len(y_name)):
            patch["data"][i] = {"x": df[x_name], "y": df[y_name[i]]}

        return patch
```

Documentation

```
def update_smoothly(data=None, x_name=None, y_name=None, dash_app=None,
                    graph_id=None, interval_id=None):
```

Description

The `update_smoothly` function updates a graph in a Dash application with smooth, incremental updates. This function utilizes client-side callbacks to efficiently handle updates by minimizing the amount of data sent between the client and server. It updates the graph dynamically using data stored in a provided structure.

Arguments

- **data** (`list of lists` or `DataFrame`): The data to be used for updating the graph. This should include values for both the x and y axes.
- **x_name** (`str`): The name of the column in `data` to be used for the x-axis. This is a required parameter.
- **y_name** (`list of str`): A list of column names in `data` to be used for the y-axis. Each element corresponds to a separate trace on the graph. This is a required parameter.
- **dash_app** (`Dash`): The Dash application instance where the callback will be registered. This is a required parameter.

- **graph_id** (`str`): The ID of the `dcc.Graph` component to be updated with the new data. This is a required parameter.
- **interval_id** (`str`): The ID of the `dcc.Interval` component that triggers the graph updates at regular intervals. This is a required parameter.

Returns

- `dict`: A dictionary representing the updated figure with new data. This dictionary includes the updated data for each trace in the graph.

Raises

- `ValueError`: If any of the required arguments (`data`, `x_name`, `y_name`, `dash_app`, `graph_id`, `interval_id`) are `None`. This ensures all necessary parameters are provided before attempting to update the graph.

Callback Function

The `update_graph` callback function is invoked periodically based on the `interval_id` component. It processes the provided `data` to update the graph, applying incremental updates to enhance performance and user experience.

Example Usage

Here is an example of how to use the `update_smoothly` function:

```
# Example usage of the update_smoothly function
update_smoothly(
    data=your_data,
    x_name='Timestamp',
    y_name=['X', 'Y', 'Z'],
    dash_app=app,
    graph_id='live-update-graph',
    interval_id='interval-component'
)
```

6 Conclusion

To conclude, the Partial Property Update approach outperforms the Server Side Callback method by efficiently managing incremental updates. By updating only the altered areas of the graph, it reduces data transfer and processing overhead, resulting in speedier response times and a more pleasant user experience. This strategy ensures that changes are applied effortlessly, similar to streaming media, where continuous data flows without noticeable interruptions. The ability to focus updates on specific graph components rather than the entire graph significantly improves the performance of real-time applications, making Partial Property Update an excellent choice for dynamic and interactive data visualizations in Dash apps.