Elaboratore Grafici

A cura di:

ANDREA IMPARATO

Indice:

1. FUNZIONALITÀ APPLICAZIONE

Elaboratore grafici è un software sperimentale il cui scopo ultimo è permettere la visualizzazione di dati numerici estratti da un file di input in forma di grafico. Le principali funzionalità che offre al momento sono:

- **Selezione di un file da cui caricare i dati**, tramite apposita opzione nel menù principale. Al momento della selezione il caricamento inizia in modo automatico, ma se risultano già presenti dei dati in memoria viene prima chiesto all'utente cosa fare tramite prompt (Annullare l'operazione, Sostituire i dati correnti, Caricare i nuovi dati in una nuova finestra). I formati accettati nella versione corrente sono:
  - **txt**
  - **csv**
  - **xls**
  - **xlsx**
- **Creazione grafico**, tramite cui viene creato un grafico a partire dai dati caricati. Questa funzionalità non necessita di input da parte dell'utente in quanto viene innescata in modo automatico non appena dei dati sono presenti in memoria. Viene altresì innescata in modo automatico tutte le volte che si verifcano cambiamenti significativi (come ad esempio un cambio nei dati da studiare o nelle dimensioni della finestra), in modo da mantenere il grafico sempre aggiornato.
- **Scelta di un nuovo tipo di grafico**, tramite un apposito menù a tendina messo a disposizione nella pulsantiera dedicata alle impostazioni del grafico. Nella versione corrente i grafici che è possibile creare sono di tre tipi:
  - **Linea**
  - **Istogramma**
  - **Scatterplot**
- **Selezione di una colonna specifica come input per il grafico**, da scegliere tra quelle rilevate nel data set caricato tramite apposito menù a tendina messo a disposizione sempre nella pulsantiera dedicata alle impostazioni del grafico. Sulla base del tipo di grafico le colonne usate sono una (Istogramma), due (Scatterplot) oppure in numero variabile (Linea). La versione attuale accetta solo dati di tipo numerico.

- **Regolazione della finestra per le medie mobili**, funzionalità disponibile solo quando il tipo di grafico è impostato su "Linea": in grafici di questo tipo è possibile mostrare per ognuna delle linee disegnate una linea ausiliaria che ne rappresenta la media mobile, la cui finestra può essere resa più grande o più piccola tramite uno slider (che per grafici di questo tipo è sempre disponibile nella pulsantiera delle impostazioni). Nella versione attuale tutte le medie mobili devono avere finestre di uguali dimensioni, e vengono disegnate non appena l'utente sceglie un valore per le finestre diverso da "1" (che è quello di default e determina la disattivazione delle medie mobili). La dimensione che è possibile scegliere va da un minimo di 1 a un massimo impostato dinamicamente in modo da essere pari a circa 1/10 del numero di record rilevati nei dati di input (questo valore non può comunque mai superare la soglia massima di 100).
- **Scelta del numero di linee**, altra funzionalità messa a disposizione solo quando il tipo del grafico è impostato su "Linea". Essa permette (sempre tramite menù a tendina) di regolare il numero di linee che si vogliono mostrare in contemporanea nel grafico. Quando vengono aggiunte nuove linee vengono anche aggiunti alla pulsantiera dei nuovi menù a tendina per scegliere le colonne che rappresentano (e viceversa).
- **Regolazione della visibilità di una linea**, altra funzionalità disponibile solo quando il tipo del grafico è impostato su "Linea". Essa permette di nascondere una linea in modo da evitare che il grafico diventi troppo affollato (o mostrarla di nuovo, in caso di secondo click). Viene innescata facendo click sul nome della linea che si vuole nascondere/mostrare nella legenda in alto, legenda che viene generata automaticamente alla creazione del grafico. Se si sceglie di nascondere una linea, viene nascosta automaticamente anche la linea ausiliaria che ne rappresenta le medie mobili qualora fosse presente.
- **Scelta del numero di classi**, funzionalità messa a disposizione solo quando il tipo di grafico è impostato su "Istogramma". La scelta viene operata tramite uno slider messo automaticamente a disposizione nella pulsantiera e permette di regolarne il numero di classi (da un minimo di 1 a un massimo di 150).
- **Modalità a schermo intero**, con cui l'utente può estendere il grafico in maniera tale da massimizzarne la dimensione. Per innescare questa funzionalità viene messo a disposizione un apposito tasto in alto a destra in ogni grafico (per tornare alle dimensioni normali è sufficiente premere di nuovo il tasto o cliccare il tasto Esc).

- **Ingrandimento di una porzione di grafico**, funzionalità messa a disposizione solo per grafici di tipo "Linea" o "Scatterplot". Tramite essa l'utente può evidenziare una porzione di grafico affinché venga ingrandita, nello specifico tenendo premuto il tasto sinistro del mouse e trascinando un rettangolo di selezione sull'area interessata. Una volta ingrandita l'area, l'utente può effettuare un nuovo ingrandimento con il tasto sinistro oppure ripristinare il grafico originale tramite un singolo click con il tasto destro.
- **Salvataggio del grafico come immagine**, tramite cui l'utente può salvare su file il grafico visualizzato in un determinato momento. Questa funzionalità non è disponibile al momento del lancio dell'applicazione ma viene automaticamente abilitata non appena viene completato il disegno del primo grafico. Per accedervi è sufficiente cliccare sull'apposito tasto nel menù principale in alto: a quel punto sarà necessario specificare (tramite prompt) il nome del file da creare, la sua estensione e la directory in cui metterlo. In fase di salvataggio la dimensione del grafico viene automaticamente regolata in modo da creare un'immagine il più grande possibile ma il contenuto non ne viene alterato: se ad esempio l'utente ha nascosto una linea in un grafico di tipo "Linea" ed effettuato uno zoom su una porzione ben precisa, l'immagine salvata riguarderà solo la porzione ingrandita e non includerà la linea esclusa. Nella versione attuale, i formati dell'immagine creata possono essere:
  - **jpg**
  - **png**
  - **svg**

2. TECNOLOGIE ADOTTATE

2.1 Linguaggi e moduli

Il Back End dell'applicazione è implementato primariamente tramite **node.js** e fa uso dei seguenti moduli:

- **electron**, che ne costituisce il pilastro fondamentale (giacché fornisce l'intelaiatura di base che regola il funzionamento sia del back end che dei processi che gestiscono le finestre di front end).
- **fs**, usata nelle situazioni in cui l'applicazione deve interagire con il file system per leggere o scrivere su file.
- **path**, per gestire in modo più comodo i path delle risorse con cui interagisce l'applicazione (questo modulo non necessita di installazione tramite npm).
- **child-process**, usato per lanciare come sotto-processi alcuni script ausiliari usati per rispondere a delle richieste specifiche (anche questo modulo non necessita di installazione tramite npm).

Gli script ausiliari lanciati come sotto-processi sono scritti in **python** e fanno uso dei seguenti moduli:

- **pandas**, per gestire il caricamento dei dati in memoria a partire da un file di input
- **openpyxl**, per gestire correttamente alcuni formati dei file di input
- **sys**, per interagire con i parametri passati agli script da linea di comando (questo modulo non necessita di installazione tramite pip).
- **json**, per scrivere i dati in output in formato JSON (anche questo modulo non necessita di installazione tramite pip).
- **numpy**, per gestire alcune operazioni di calcolo (anche questo modulo non necessita di installazione tramite pip).

Le finestre di front end sono scritte in html, css e javascript e sono costituite per la maggior parte da codice scritto ad hoc. Alcune funzionalità si appoggiano però a librerie javascript già pronte (a volte addirittura importate come script all'interno del documento html). In particolare:

- **File System API** per interagire con il file system in fase di salvataggio del grafico come immagine (non richiede import)
- **D3** per disegnare i grafici (i file di cui si compone la libreria sono stati innestati nella cartella degli script in modo da poterla includere direttamente tramite tag <script> senza doverla importare come modulo).
- **dom-to-image** per convertire il tag <svg> che incapsula il grafico in un blob salvabile su file .jpg o .png (anche questa libreria è stata innestata negli script in modo da poterla importare direttamente tramite tag <script>).

L'applicazione è stata inoltre concepita affinché fosse possibile impacchettare il codice sorgente creando un file eseguibile con cui lanciarla. Questa scelta ha determinato la presenza di alcune dipendenze aggiuntive che non sono però legate in senso stretto alla logica dell'applicazione. Esse sono:

- **electron-forge** per quanto riguarda node.js
- **Git** per quanto riguarda il sistema usato dall'utente (giacché electron-forge richiede la presenza di git per funzionare)
- **pyinstaller** per quanto riguarda python

2.2 Installazione

Alla luce delle dipendenze illustrate nella sezione precedente, per poter lanciare l'applicazione è in ultima analisi necessario installare sia node.js che python oltre ai moduli che non sono inclusi in automatico con le due tecnologie. Nel caso di node.js questi moduli sono elencati nel file "package.json" incluso nella directory "SourceCode" (in cui risiede tutto il codice sorgente). Nel caso di python questi moduli sono invece elencati nel file "requirements.txt" incluso nella sottodirectory "python" (a sua volta reperibile all'interno della directory "SourceCode"). Supponendo che l'utente si trovi nella directory "SourceCode", per poter includere correttamente i moduli mancanti è sufficiente lanciare i seguenti comandi:

```
npm install
pip install -r ".\python\requirements.txt"
```

Il primo si occupa di installare i moduli mancanti di node.js (creando così la cartella "node-modules") a partire dal file "package.json", mentre il secondo installa i moduli necessari a python usando il file "requirements.txt". Affinché sia possibile lanciare l'applicazione è però necessario che risulti installato anche Git sul sistema in uso dall'utente. Supponendo che lo sia (in caso contrario è sufficiente installarlo a parte), l'applicazione potrà a quel punto essere lanciata (sempre dalla cartella "SourceCode") tramite il comando:

```
npm start
```

2.3 Creazione di un singolo eseguibile

L'applicazione può essere impacchettata in un singolo eseguibile tramite electron-forge, ma prima di farlo è necessario convertire preventivamente a eseguibili anche gli script python ausiliari che vengono lanciati dal processo principale. Per prima cosa si suppongono essere stati già seguiti tutti i passi illustrati nella sezione precedente (incluso l'aver aggiornato la directory corrente a "SourceCode"). Per poter creare un file .exe a partire dal codice sorgente vanno quindi lanciati in ordine i seguenti comandi:

```
pyinstaller --distpath "./python" --specpath "./python" --workpath "./python/build" -F "./python/DataLoader.py"
pyinstaller --distpath "./python" --specpath "./python" --workpath "./python/build" -F "./python/DataProcessor.py"
npm run make
```

Una volta terminato, l'ultimo comando produrrà nella cartella del codice sorgente una cartella "out" che contiene al suo interno una cartella con un eseguibile per l'installazione e una cartella con l'applicazione impacchettata (l'eseguibile per l'installazione si limita a ricreare quest'ultima all'interno della directory AppData dell'utente che lo lancia). La cartella con l'applicazione può essere spostata a piacimento fintanto che i file al suo interno non vengono separati. Per far partire l'applicazione sarà a quel punto sufficiente lanciare l'eseguibile .exe (trattandosi di un prototipo è necessario fornire i permessi di amministratore).

3. ARCHITETTURA SOFTWARE

3.1 Struttura di un'applicazione electron

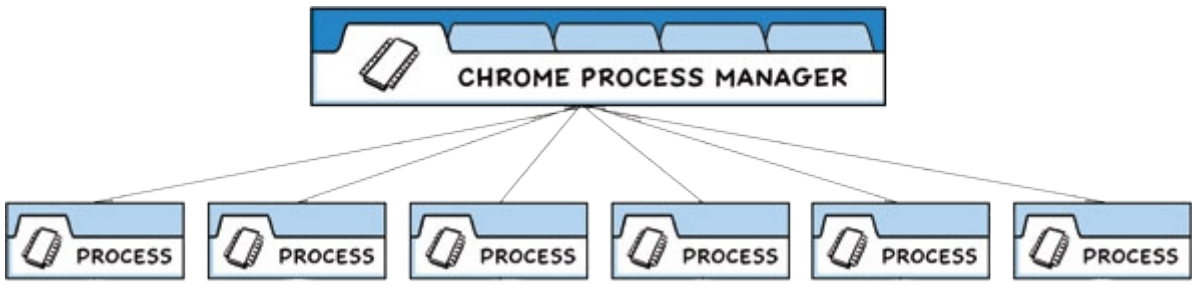
La struttura basilare di una generica applicazione electron è descritta in dettaglio nei seguenti link:

https://www.electronjs.org/docs/latest/tutorial/quick-start
https://www.electronjs.org/docs/latest/tutorial/process-model

Volendo sintetizzare il più possibile, i componenti che entrano in gioco sono fondamentalmente quattro:

- Un singolo processo **Main**
- I processi **Renderer**, ognuno dei quali gestisce una delle finestre aperte
- Lo script di **Preload**
- La comunicazione bidirezionale tra questi processi (**Ipc**)

L'utilizzo di più processi (in luogo di un singolo processo che si occupa di tutto) è di fatto un tentativo di replicare il modello già usato in Google Chrome in cui un processo madre in background gestisce un singolo processo per ogni tab aperto, come illustrato nella seguente figura:



In un'app electron il processo madre consiste nel processo **Main**, il quale è il primo a venir lanciato e costituisce il cuore del back end dell'applicazione. Di fatto, altro non è che uno script JavaScript che viene fatto girare su node.js automaticamente all'avvio dell'app (va indicato esplicitamente nel file package.json, alla voce "main"). In questo script è possibile

fare riferimento all'applicazione stessa tramite la variabile 'app' importata dal modulo 'electron': tramite essa se ne possono definire i comportamenti sfruttando dei listener che reagiscono ai cambiamenti di stato che avvengono durante il suo ciclo di vita. Un comportamento molto basilare può essere ad esempio:

```
const { app, BrowserWindow } = require('electron');

app.whenReady().then( ( ) => {
        createWindow( )
}) ;

function createWindow( ) {
        const win = new BrowserWindow({
        width: 800,
        height: 600,
        webPreferences: {
                preload: path.join(__dirname, 'preload.js')
                }
        });
        win.loadFile('index.html');
}
```

Quando l'app ha raggiunto lo stato 'ready' viene innescata la funzione 'createWindow' che si fa carico del lanciare un nuovo processo **Renderer** tramite chiamata a BrowserWindow. BrowserWindow è un costruttore fornito sempre dal modulo 'electron' che richiede in input un parametro con alcuni dati di configurazione (tra cui altezza e larghezza iniziali della finestra da istanziare) e restituisce in output un riferimento al nuovo processo che gestisce la finestra. Il metodo loadFile chiamato su quel riferimento provvede a riempire la finestra con l'html specificato nel file 'index.html'. Nell'oggetto di input da passare a BrowserWindow figura anche un oggetto WebPreferences in cui va specificato un sottoinsieme di parametri specifici, tra cui il path dello script di **Preload**. Questo script è importantissimo perché se da un lato le singole finestre aperte dall'app principale possono caricare i loro script come sempre, dall'altro non possono mai accedere ai moduli di node per motivi di sicurezza, vedendosi quindi precluse moltissime funzionalità (ad esempio l'accesso al file system tramite modulo

'fs'). Lo script di preload è l'unico ad avere accesso ai moduli di node.js pur essendo caricato dalle finestre front end, risultando di fatto in una specie di "limbo" che gli permette di fare da ponte tra i due contesti: l'utilità di ciò sta nel fatto che grazie alla variabile 'contextBridge' (sempre importata dal modulo 'electron', e da cui si può invocare il metodo 'exposeInMainWorld') diventa possibile esportare presso le finestre front end alcune funzionalità che sono normalmente riservate al back end (vedi codice qui sotto).

```
const { contextBridge, ipcRenderer } = require("electron");

contextBridge.exposeInMainWorld(
	"api", {
		send: (channel, data) => {
			ipcRenderer.send(channel, data);
		},
		receive: (channel, func) => {
			ipcRenderer.on(channel, (event, args) => func( args ) );
		}
	}
);
```

In questo breve esempio le chiamate a ipcRenderer vengono incapsulate come metodi "send" e "receive" di un oggetto "api" che viene reso disponibile alla finestra di front end come proprietà della window. Per usare **ipcRenderer.send(channel , data)** il front end dovrà quindi scrivere **window.api.send(channel , data)**. Una descrizione più dettagliata di questa architettura è resa disponibile ai seguenti link:

https://www.electronjs.org/docs/latest/tutorial/ipc
https://stackoverflow.com/questions/44391448/electron-require-is-not-defined

Essa è fondamentale anche e soprattutto perché costituisce l'unico modo per rendere disponibile al front end le funzionalità di **Ipc**, il principale componente di riferimento messo a disposizione da electron per la comunicazione tra processo main e processi renderer (da cui il nome Ipc). Complessivamente Ipc permette di realizzare una comunicazione bidirezionale, ma ognuna delle sue singole procedure agisce sempre in riferimento a una sola direzione (dal front end al back end o viceversa) e a un canale specifico (che è di fatto un modo per permettere il multiplexing). Le procedure principali che entrano in gioco sono quattro, e sono tutte event-based. Per illustrarle, la via più semplice è far vedere il modo in cui collaborano tra loro:

```
ipcMain.on( 'front_to_back_1' , ( event, data ) => {
        //Fai qualcosa con i dati ricevuti, rappresentati dal parametro "data"
});
```

Tramite questo primo metodo il processo main può predisporre un comportamento che si attiva quando arriva una comunicazione sul canale "front_to_back_1", rappresentata dall'evento "event" e contenente le informazioni "data".

```
ipcRenderer.send( 'front_to_back_1', data );
```

Tramite questo secondo metodo, il processo renderer può innescare l'evento di comunicazione sul canale 'front_to_back_1', allegando come informazioni trasmesse il contenuto della variabile "data". (affinché la comunicazione riesca il main deve ovviamente essersi messo preventivamente in ascolto sul canale 'front_to_back_1' con il metodo subito sopra)

```
ipcRenderer.on( 'back_to_front_1' , ( event, data ) => {
        //Fai qualcosa con i dati ricevuti, rappresentati dal parametro "data"
});
```

Tramite questo metodo, il processo renderer può predisporre un comportamento che si attiva quando arriva una comunicazione sul canale "back_to_front_1" (è di fatto il metodo "speculare" rispetto al primo).

```
win.send( 'back_to_front_1', data );
```

Tramite questo metodo, il processo main può innescare l'evento di comunicazione sul canale "back_to_front_1" con allegate le informazioni contenute nella variabile "data". Se le finestre di front end non devono specificare un destinatario per le comunicazioni in uscita (giacché il processo main è sempre uno solo), lo stesso non vale per il main (che potrebbe trovarsi con più finestre front end aperte in contemporanea): il metodo va quindi chiamato a partire da un riferimento "win" che rappresenta il processo BrowserWindow a cui inviare la comunicazione.

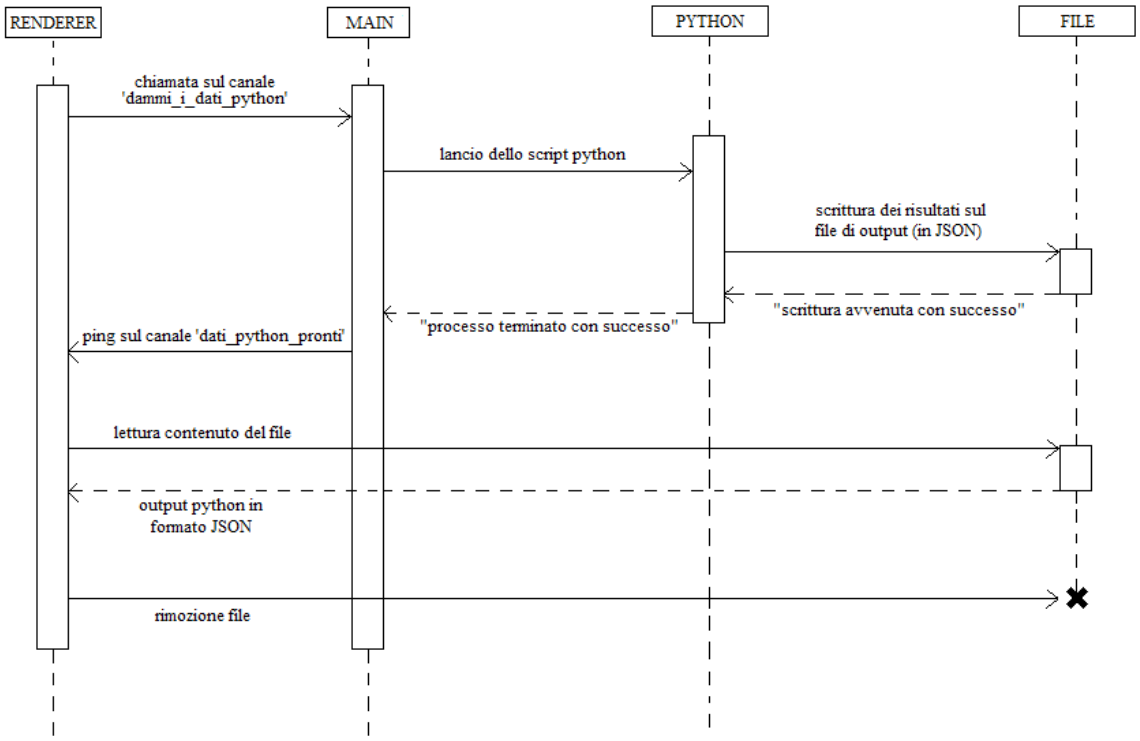
3.2 Terzo layer (Python), limiti di IPC, pipe "artigianali"

I componenti illustrati fino a questo punto costituiscono dei pilastri fondamentali che è facile trovare in quasi tutte le app electron. Rispetto a questa architettura di base, l'Elaboratore Grafici presenta però anche un livello aggiuntivo che consiste in due script python normalmente inattivi che vengono eseguiti quando arrivano determinate richieste dalle finestre di front end. Più nello specifico, uno di questi script (**dataLoader.py**) si occupa di leggere e processare il contenuto del file contenente il data set di cui si vogliono creare i grafici mentre l'altro (**dataProcessor.py**) si occupa di svolgere calcoli aggiuntivi su porzioni specifiche di un data set su cui si sta già lavorando (al momento l'unica opzione di calcolo disponibile è la media mobile). Questi due script possono essere lanciati solo su ordine del main come processi figli (tramite il metodo "spawn" del modulo "child-process") e non ricevono comunicazioni se non i parametri di input passati da linea di comando al momento del lancio (sempre dal main). Ha quindi senso che il main stesso faccia da middle-man tra gli script python da un lato e le finestre di front end dall'altro (che ne devono ricevere i risultati in output). Facendo riferimento a quanto detto nella sezione precedente, l'architettura che idealmente vorremmo realizzare tramite Ipc prevederebbe l'incapsulamento da parte del Main all'interno di una stessa funzione tanto della chiamata di avvio dello script python quanto dell'inivio al front end dei dati prodotti in output, la quale funzione si attiva non appena il Main riceve la richiesta (come illustrato nel seguente pseudo-codice)

```
ipcMain.on( 'dammi_i_dati_python' , ( event, python_args ) => {
        win = finestra chiamante
        risultato = runPythonScript( python_args );
        win.send( 'tieni_i_dati_python' , risultato )
});
```

Sfortunatamente, un'architettura del genere si è rivelata incapace di gestire grandi quantità di dati quando questi vengono fatti transitare dal back end al front end, rendendo necessaria l'introduzione di un escamotage alternativo per garantire l'arrivo dei dati al front end. Dopo vari tentativi la soluzione più semplice si è rivelata essere la lettura/scrittura da file secondo il modello produttore/consumatore (come illustrato nel seguente diagramma di sequenza)

Diagramma di sequenza che illustra la comunicazione
(front end, back end e script pyton)



I passi che avvengono sono in ordine i seguenti:

- Il main riceve la richiesta dalla finestra front end
- Il main attiva lo script python associato alla richiesta ricevuta
- Lo script esegue le sue operazioni e scrive il suo output (in formato JSON) su un file
- Quando lo script python termina, il main intercetta l'evento e manda un ping alla finestra di front end per segnalare che i dati richiesti sono pronti e possono essere raccolti
- La finestra front end apre il file prodotto da python e ne legge il contenuto
- Una volta letto il contenuto, la finestra front end cancella il file.

Il funzionamento di questa comunicazione è a conti fatti molto simile a quello delle pipe messe a disposizione dai vari sistemi operativi, con l'importante differenza che nel nostro caso non vengono usate system call di basso livello ma semplici chiamate al modulo 'fs' di node.js (in modo da semplificare l'implementazione e preservare la portabilità). Nel caso specifico dello script dataProcessor.py c'è inoltre un passaggio in più: la preparazione dei dati su cui lo

script deve eseguire i suoi calcoli, i quali vengono scritti dalla finestra front end su un file temporaneo in formato JSON molto simile a quello usato per ricevere i risultati (in altre parole in questo caso il produttore è il front end e il consumatore lo script python). Ne consegue che complessivamente ogni finestra potrebbe trovarsi tanto a dover inviare dei dati da elaborare quanto a dover raccogliere i risultati di un'elaborazione conclusa. E' quindi bene che per ogni finestra attiva siano sempre riservati sia un file su cui scrivere le informazioni da inviare, sia uno da cui leggere le informazioni da ricevere. Per raggiungere questo scopo, i nomi dei file che fanno da ponte hanno sempre la stessa struttura: un identificatore numerico che è unico per ogni finestra seguito dal suffisso "To.txt" oppure "From.txt". Pertanto, la finestra numero 1 userà il file "1To.txt" per ricevere dati e il file "1From.txt" per inviare dati. Tutti i file di questo tipo risiedono nella directory "bridge" (la quale viene pulita a ogni nuovo avvio e chiusura dell'app come misura di sicurezza aggiuntiva).

3.3 Gestione dei dati in memoria, ciclo di vita del grafico

L'app è stata concepita con in mente l'idea di limitare il più possibile le chiamate che le finestre devono fare al back end. Ne consegue che quando una finestra ottiene un data set di cui ha chiesto il caricamento essa provvede a conservarlo in una variabile globale dedicata (chiamata "informazioni") il cui contenuto rimane inalterato fino alla chiusura della finestra o al caricamento di un nuovo data set che sostituisca il precedente. Una volta completato il caricamento viene subito avviata la creazione di un primo grafico e messa a disposizione la pulsantiera tramite cui l'utente può cambiarne i parametri (tipo di grafico, variabili studiate, etc.). La creazione di un nuovo grafico comincia sempre con l'inizializzazione di una variabile globale "buffer" in cui viene copiato il sottoinsieme di colonne specifiche di cui verrà disegnato il grafico. La motivazione principale dietro questa operazione è che per disegnare il grafico i dati devono essere in un formato leggermente diverso rispetto a quello usato in "informazioni" (il cui contenuto però non deve mai venire alterato). La variabile globale "buffer" conserva però anche altre informazioni importanti per la creazione del grafico, come ad esempio le sue dimensioni: come principio generale, le varie funzioni legate al ciclo di vita del grafico interagiscono tutte con questa variabile (a seconda dei casi o sfruttandone il contenuto per disegnare oppure alterandone il contenuto per modificare alcuni aspetti fondamentali del grafico). In virtù di questo le varie azioni intraprese dall'utente innescano

delle operazioni di ricalcolo che possono essere più o meno estese a seconda dell'azione specifica ma rimangono sempre limitate al front-end (con l'importante eccezione del ricalcolo delle medie mobili per i grafici di tipo "Linea", nel qual caso è inevitabile una chiamata al back end). In generale queste operazioni consistono in:

- Aggiornare la cornice del grafico (newFrame( ))
- Aggiornare una parte dei dati del grafico (aggiornaDatiX( ) , aggiornaDatiY(count))
- Reinizializzare completamente i dati del grafico (inizializzaDati( ))
- Aggiornare gli assi (aggiornaAsse(axis), ricalcoloEstremiY( ))
- Ridisegnare il grafico (reDraw( ))

L'approccio modulare permette di far sì che di volta in volta vengano invocate solo le operazioni realmente necessarie per far fronte alle richieste dell'utente. In riferimento alle funzionalità illustrate in precedenza:

- Se l'utente carica un nuovo data set o sceglie di cambiare il tipo di grafico vengono eseguite tutte le operazioni: vengono inizializzati i dati, inizializzate le dimensioni della cornice del grafico, aggiornati gli assi inizializzandone sia la dimensione (calcolata con la nuova cornice) che la scala (calcolata con i nuovi dati appena inizializzati) e infine viene disegnato il grafico.
- Se la finestra front end viene ridimensionata dall'utente, viene aggiornata la cornice del grafico, vengono aggiornati gli assi (per cambiarne la lunghezza in modo da riflettere la nuova cornice) e viene infine ridisegnato il grafico. Non vengono però alterati i dati tramite cui viene fatto il disegno, né la scala degli assi.
- Se l'utente sceglie una nuova colonna da studiare vengono modificati parzialmente i dati e aggiornati gli assi (giacché la scala potrebbe cambiare in virtù dei nuovi dati), ma non vengono modificate le dimensioni della cornice. Viene quindi ridisegnato il grafico.
- Se l'utente cambia il numero di linee in un grafico di tipo "Linea" vengono aggiornati i dati per aggiungere o togliere le colonne interessate, aggiornati gli assi (nuova scala) e viene ridisegnato il grafico. Anche in questo caso non vengono modificate le dimensioni della cornice.
- Se l'utente cambia il numero di classi in un grafico di tipo "Istogramma" il grafico

viene semplicemente ridisegnato senza bisogno di alterare le dimensioni della cornice o i dati. Anche gli assi non vengono modificati perché nel caso particolare dell'istogramma la scala dell'asse verticale viene automaticamente ricalcolata ogni volta che il grafico viene ridisegnato (giacché dipende dalle frequenze assolute delle classi).

- Se l'utente attiva la modalità schermo intero vengono innescate le stesse operazioni che hanno luogo quando avviene un generico ridimensionamento della finestra: aggiornamento delle dimensioni della cornice, aggiornamento della lunghezza degli assi, nuovo disegno del grafico. Anche in questo caso non vengono modificati i dati.
- Se l'utente effettua uno zoom su una specifica porzione di grafico vengono aggiornate delle apposite proprietà nel buffer in cui risiedono i dati: esse servono a memorizzare gli estremi che delimitano la porzione interessata dall'ingrandimento. Questi estremi vengono poi usati per aggiornare la scala degli assi (il cui minimo e massimo viene fatto corrispondere proprio a essi) e il grafico viene infine ridisegnato. Non vengono alterati né i dati di partenza né la dimensione della cornice. Operazioni simili avvengono se l'utente ripristina il grafico al suo stato originale.

Le azioni di salvataggio del grafico come immagine e preparazione delle medie mobili costituiscono due casi particolari. Nel caso del salvataggio vengono cambiate le dimensioni della cornice estendendole fino alla dimensione dello schermo usato dall'utente (in modo da garantire un'immagine della massima dimensione) e poi il grafico viene ridisegnato. Se il formato dell'immagine da salvare è svg, al grafico viene anche aggiunta una legenda ad hoc perché la legenda ordinaria non viene inclusa (essendo scritta in html e non in svg). Dopo il salvataggio, vengono ripristinate le dimensioni originali della cornice e il grafico viene ridisegnato una seconda volta per riportarlo al suo stato originale. Nel caso della preparazione delle medie mobili vengono invece effettuate operazioni diverse a seconda della situazione. Se la preparazione termina subito senza l'invio di alcuna richiesta al back end (ad esempio se l'utente riporta a "1" la dimensione della finestra della media, disattivando così le medie mobili) allora il grafico viene semplicemente ridisegnato. Se invece viene effettivamente inoltrata una richiesta al back end (ad esempio quando l'utente imposta la dimensione della finestra della media a un qualsiasi altro valore diverso da "1") non viene fatto nulla e il ridisegno del grafico viene innescato direttamente dalla funzione che processa i dati in arrivo dal back end (in modo da disegnare solo quando questi sono stati effettivamente ricevuti).