Challenge CINI: Sistema di Monitoraggio e Controllo dell'illuminazione pubblica per smart city

M. Calzetta
Università Roma "Tor Vergata"
m_calzetta@live.com

V. Di Tinno
Università Roma "Tor Vergata"
valerio.ditinno@gmail.com

O. Morga
Università Roma "Tor Vergata"
morga.olga@gmail.com

M. Vittorini
Università Roma "Tor Vergata"
vittorini.maurizio@gmail.com

ABSTRACT

Lo scopo di questo progetto è quello di realizzare un sistema di monitoraggio e controllo dell'illuminazione pubblica in grado di adattare dinamicamente il grado di intensità luminosa dei lampioni disseminati sul territorio urbano in base alle condizioni ambientali e di traffico. Il nostro sistema permette di migliorare l'efficienza energetica e di ridurre l'inquinamento luminoso garantendo performance e scalabilità grazie l'utilizzo delle più moderne tecnologie per il data stream processing.

KEYWORDS

datastream processing, distributed system, smart city, cloud, big data, Apache Flink, Apache Kafka

1 INTRODUZIONE

Consumare meno energia è da anni la prerogativa di qualsiasi ente pubblico e privato. In particolare è nell'ambito della gestione dell'illuminazione pubblica, che si è concentrata maggiormente l'attenzione e per comprendere il perché basta osservare alcuni dati. Una stima dei costi affrontati dalle città italiane per l'illuminazione pubblica arriva dall'Enea, che in [1], evidenzia come, ogni comune spende in media 103,3 euro per ciascun punto luce e 817,7 euro per kW, che si traducono in una spesa media di quasi 19 euro all'anno per abitante (18,7 euro). Considerando ad esempio che nel 2012 a Torino i punti luce installati erano 96000; a Milano 138364 ed a Roma 181991, ci si rende conto che si sta parlando di diverse centinaia di milioni di euro che gravano sui conti pubblici.

Numerose sono le aziende che hanno proposto soluzioni al problema, come la Philips che con il suo progetto CityTouch [?] diventato realtà in oltre 30 paesi ha ottenuto risultati sorprendenti come nel caso della cittadina spagnola di Salobre in cui l'introduzione combinata di luci LED e del proprio software di gestione hanno portato una riduzione del consumo di energia di oltre il 70% ed una riduzione dell'emissione di CO2 di circa 29 tonnellate l'anno.

I risultati ottenuti da Philips seppur ottimistici, fanno riflettere su quale possa essere l'impatto di una soluzione che miri a ridurre gli sprechi dell'illuminazione pubblica.

L'obiettivo di questo progetto, realizzato nell'ambito del Challenge CINI[2] per smart city, è proprio quello di produrre un sistema di monitoraggio e controllo dell'illuminazione pubblica capace di rilevare lo stato di funzionamento di ogni lampione in real-time e di controllare l'intensità luminosa dei lampioni della rete, in base al livello di luce naturale e al livello di traffico al fine di:

- agevolare la manutenzione e ridurre i costi di gestione;
- permettere l'adattamento dell'illuminazione ad eventi e condizioni atmosferiche;
- aumentare l'efficienza energetica riducendo l'impatto ambientale;
- aumentare la sicurezza delle città, assicurando che il livello di luminosità non sia inferiore ad una soglia critica.

2 ARCHITETTURA DEL SISTEMA

Il sistema, progettato per riceve in input dati da lampioni di ultima generazione capaci di collegarsi in rete e dotati di sensori di luminosità, è stato ideato per lavorare secondo un approccio data-stream processing e suddiviso in sottosistemi indipendenti tra i quali troviamo:

- il sistema di monitoraggio, che permette di monitorare lo stato della rete, produrre statistiche aggiornate sul consumo e rilevare guasti;
- la dashboard, che permette di visualizzare le statistiche provenienti dal sistema di monitoraggio ed effettuare operazioni CRUD sui lampioni;
- il sistema di controllo, che permette di aggiustare il livello di luminosità dei lampioni in base all'intensità di luce naturale e l'intensità di traffico;
- il controllore locale, che riceve gli aggiustamenti di

intensità luminosa provenienti dal sistema di controllo e le inoltra ai specifici lampioni

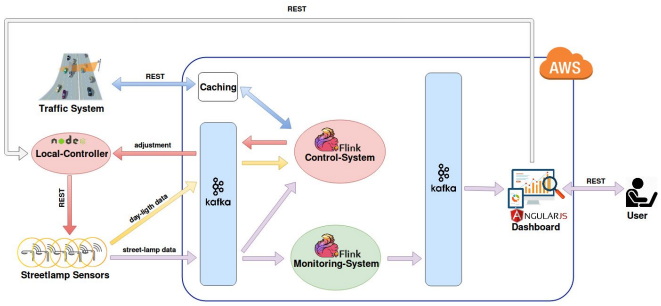


Figura 1: architettura del sistema.

I sottosistemi organizzati a micro-servizi comunicano, a seconda delle necessità, tramite API REST o tramite il sistema publish-subscribe Apache Kafka[3]: il quale risulta essere l'ideale per applicazioni di data-stream processing, in quanto garantisce, una semantica di comunicazione di tipo at-least once, inoltre è scalabile orizzontalmente per cui regala robustezza al nostro sistema potendo replicare le partizioni dei topics su più broker.
Per quanto riguarda i sistemi di monitoraggio e controllo abbiamo scelto di realizzarli tramite il framework Apache Flink[4] poiché è risultato essere quello in grado di offrire le migliori garanzie dal punto di vista prestazionale, garantendo alto throughput e bassa latenza, oltre a garantire una ricca API che ci ha agevolato nello sviluppo del progetto.
Entriamo ora in maggior dettaglio nella descrizione dei vari sottosistemi e delle loro funzionalità.

2.1 Monitoring-System

Il sistema di monitoraggio riceve in input i dati da Kafka, provenienti dai sensori posizionati sui lampioni, e li processa al fine di produrre statistiche aggiornate sul consumo e rilevare guasti. Successivamente le statistiche prodotte e i warning vengono pubblicati su topic Kafka, dove saranno consumati dalla Dashboard che li mostrerà in tempo reale all'utente. In particolare il sistema di monitoraggio permette di:

- **Query 1**: rilevare e notificare la presenza di lampioni non funzionanti, dove per non funzionanti si intendono lampioni con dati medi sul consumo anomali rispetto a dati medi aggregati (strade, città) o che presentano anomalie sul loro stato interno di accensione;
- **Query 2**: calcolare la classifica dei X-lampioni le cui lampade abbiano superato un prefissato tempo di vita, con X parametro configurabile compreso tra 3 e 10;
- **Query 3**: calcolare le statistiche sul consumo del singolo lampione o di aggregati di lampioni(lampioni lungo una singola strada, tutta la città) per fasce orarie(nell'ultima ora, nelle ultime 24 ore e nell'ultima settimana);
- **Query 4**: calcolare la percentuale di lampioni in ogni strada aventi la mediana(i.e. 50-percentile) del consumo calcolata nell'ultima ora maggiore della mediana globale sempre calcolata nell'ultima ora. L'output deve essere prodotto solo quando cambia il valore della percentuale per una strada.

2.2 Dashboard

La dashboard permette agli addetti alla manutenzione e all'installazione dei lampioni di visualizzare in real-time le statistiche provenienti dal sistema di monitoraggio tramite grafici e tabelle. Realizzato in AngularJS[10], è stato ideato per essere di semplice utilizzo ma allo stesso tempo accattivante per competere con le altre WebApp del Challenge che hanno puntato molto su questo aspetto. Presenta un Layout responsive per poter accedere al sistema di controllo e monitoraggio anche da dispositivi mobili. Permette di effettuare operazioni CRUD al fine di inserire, modificare o rimuovere un lampione e le informazioni relative. Inoltre permette di tener traccia degli aggiustamenti del sistema effettuati dal sistema controllo.

2.3 Control-System

Il sistema di controllo, prende in input, oltre ai dati provenienti dai lampioni, quelli prodotti dai sensori di intensità di luce naturale co-locati sui lampioni che vengono emessi come un flusso continuo di dati. Il sistema legge i dati tramite Kafka ed utilizza tali informazioni per aggiustare in real-time il livello di intensità luminosa. Il sistema di controllo nel modificare il livello di intensità luminosa dei singoli lampioni tiene conto dell'intensità di traffico veicolare presente sulla strada dove si trova il lampione abbassandolo in caso di scarso traffico e aumentandoli altrimenti. Le informazioni sul traffico vengono ottenute interrogando un sistema di terze parti per il monitoraggio del traffico tramite un'interfaccia REST. Il sistema produce in output gli aggiustamenti del livello di in intensità luminosa per ogni lampione e lo pubblica su topic Kafka affinché il controllore che si occupa del dato lampione possa leggere l'informazione ed inviare l'aggiustamento al singolo lampione. In particolare il sistema di controllo permette di:

- **Query 5**: adattare il livello di intensità luminosa del lampione in base all'intensità di luce naturale in modo

da adattarsi automaticamente al variare del livello di luce naturale al fine di garantire un livello ottimale di illuminazione;

- **Query 6**: adattare il livello di intensità luminosa in base al traffico veicolare presente sulla strada dove è collocato il lampione, abbassandolo in caso di scarso traffico e aumentandolo altrimenti, in modo da ottimizzare il consumo energetico senza degradare i livelli minimi di condizioni di sicurezza;

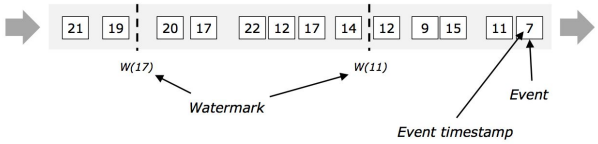
2.4 Local controller

Il controllore locale dei lampioni riceve gli aggiustamenti del livello di intensità luminosa provenienti dal sistema di controllo tramite topic Kafka e li inoltra allo specifico lampione a cui è indirizzato tramite protocollo HTTP. Ogni controllore gestisce un sottoinsieme dei lampioni presenti nella città, fornendo uno strato di indirezione tra il sistema di controllo centrale ed i lampioni, che in questo modo viene sollevato dal problema di conoscere l'identità dei singoli lampioni. Come ipotesi esemplificativa il sistema è stato realizzato per funzionare con un unico controllore locale, poiché nella simulazione dovrà gestire un centinaio di strade e qualche migliaia di lampioni: possibili sviluppi futuri per gestire la variante con più local-controller verrà esposta nella sezioni conclusive del progetto, ma ci teniamo a precisare che il sistema è stato realizzato per essere semplicemente estendibile. Inoltre il controllore locale espone interfaccia REST per le operazioni CRUD provenienti dalla Dashboard. Il controllore locale è stato realizzato in NodeJS, poiché per le sue caratteristiche risulta ideale per essere eseguito su microprocessori: ipotizziamo che questo componente potrebbe essere installato su delle torrette in prossimità dei lampioni che controlla in modo da ridurre il consumo per la comunicazione e garantire un minimo di disponibilità di servizio anche in assenza di connettività di rete .

3 IMPLEMENTAZIONE

In questa sezione verranno esposte le scelte implementative che hanno caratterizzato lo sviluppo del nostro sistema.

Aver scelto Apache Flink ci ha dato il grande vantaggio di poter aggregare Stream di dati in finestre temporali, infatti il framework mette a disposizione nativamente diverse interfacce per effettuare operazioni su tali aggregazioni. Inoltre Flink permette nativamente la gestione del tempo in diverse modalità[19]. In particolar modo è risultata indispensabile la gestione del tempo in modalità Event Time, dove il tempo dell'evento è il tempo in cui ogni singolo evento si è verificato sul suo dispositivo di produzione. Questo valore è tipicamente incorporato all'interno dei record prima di entrare in Flink e viene estratto in ingresso al sistema. Ad esempio una finestra ad Event Time di un'ora contiene tutti i record aventi timestamp compresi in quell'ora a prescindere dal momento e dall'ordine di arrivo degli eventi. Un elaboratore di stream che supporta la modalità Event Time ha bisogno di un modo per misurare lo scorrere del tempo degli eventi, e questo è realizzato in Flink grazie al meccanismo dei watermark[17]. I watermark sono particolari tuple inserite nel flusso dati che quando raggiungono un operatore, esso può avanzare il suo Event Time Clock interno al valore del timestamp associato al watermark. Un watermark al tempo t dichiara che è stato raggiunto quel particolare evento t e che non ci dovrebbero essere più elementi nel flusso con un timestamp t' <= t.



Il particolare tipo di finestre scelte per l'implementazione dei sistemi di Controllo e Monitoraggio sono:

- EventTimeTumblingWindow: prevede la specificazione di un solo parametro, ossia la durata della finestra.
- EventTimeSlidingWindow: a differenza della finestra precedente consente la definizione di un secondo parametro, ossia il parametro di slide. Tale parametro definisce il tempo di emissione di una nuova finestra.

L'utilizzo di finestre di tipo Sliding consente di ottenere risultati in tempi minori, rispetto a quanto previsto dalla lunghezza della finestra.

Dalle diverse possibilità in cui è possibile operare su dati aggregati[20], per ottimizzare sia le prestazioni che l'occupazione di memoria si è scelto di utilizzare aggregazioni incrementali nelle finestre. L'utilizzo di una semplice WindowFunction per effettuare operazioni sullo stream, prevede che la computazione avvenga solo dopo lo scadere dell'arco temporale prefissato dalla finestra. Durante tale periodo lo Stream viene tenuto in memoria in attesa del processamento. Per evitare questo accumulo di dati, dato dalla necessità di mantenere per lunghi lassi temporali informazioni sullo stream, la WindowFunction viene combinata con una FoldFunction il che consente di effettuare la computazione su ogni tupla in ingresso alla finestra, al fine di mantenere in memoria un unico valore, risultato delle operazioni

effettuate sulle tuple fino a quel momento, che verrà restituito allo scadere del periodo temporale indicato dalla finestra.

3.1 Monitoring-System

Il sistema di monitoraggio come già accennato computa statistiche e rileva malfunzionamenti dei lampioni. Le tuple in ingresso vengono immesse nel sistema da Kafka in formato JSON. In particolare i dati verranno emessi secondo il seguente formato:

```
{
        "lampId": "long",        "address":"string",
        "latitude":"string",
        "longitude":"string",
        "model":"string",
        "consumption":"double",
        "stateOn":"bool",
        "lightIntensity":"(0,1) double",
        "lastSubstitutionDate":"long",
        "residualLifetime":"long",
        "timestamp":"u_int",
        "city":"string"
},
```

Tali tuple corrispondono all'oggetto Lamp definito nel nostro sistema e un apposito filtro a monte dei diversi flussi computazionali garantisce l'immissione di tuple ben formate. La figura 2 riproduce lo schema generale del sistema e si può notare come al termine di ciascun flusso i risultati delle operazioni vengono inseriti in appositi topic Kafka, sempre in formato JSON.

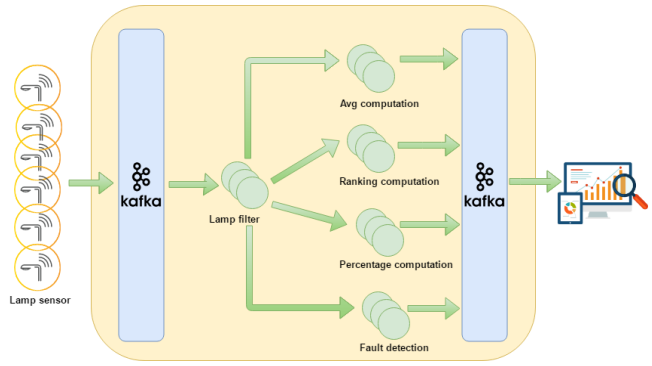


Figura 2: architettura del sistema di monitoraggio.

Passiamo ora ad esaminare nel dettaglio i flussi.

3.1.1 *Lifetime ranking*

Questo flusso è stato implementato per rispondere alla query 1 utilizzando la struttura dati TreeSet che consente nativamente di mantenere collezioni di oggetti ordinati secondo uno specifico attributo. Come anticipato la prima operazione che viene effettuata è un'operazione di filtraggio delle tuple che non hanno superato un prefissato tempo di vita: come ipotesi semplificativa abbiamo assunto che il tempo di vita possa essere calcolato in base al valore del timestamp meno il valore relativo alla data di ultima sostituzione. Invece di calcolare un'unica classifica per tutti i dati in ingresso, il flusso viene suddiviso in diversi operatori che parallelamente elaborano una loro classifica parziale per poi convergere in un operatore che potrà più facilmente calcolare una classifica complessiva. Per garantire la coerenza del calcolo parziale ad ogni operatore vengono assegnate le tuple dello stesso set di lampioni raggruppando per id lo stream di dati processato.

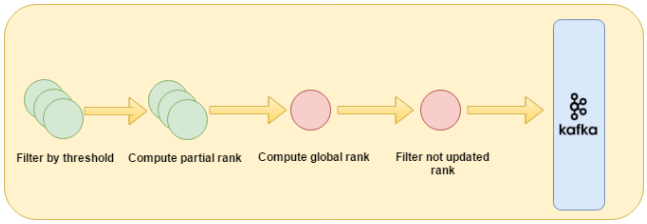


Figura 3: flusso flink calcolo della classifica.

La classifica finale verrà emessa su un topic Kafka solo nel caso in cui sia differente dalla classifica precedentemente emessa. Nella figura 3 si può notare il differente grado di parallelismo dei vari operatori.

3.1.2 *Average Consumption Computation and Fault Detection*

Questo flusso ci ha permesso di rispondere alle query 1 e 3. Il calcolo della media ha portato a molteplici studi su come venivano elaborati i dati da Flink, in quanto avevamo l'esigenza di calcolare i dati aggregati per lampione, strada e città in diversi archi temporali. Le considerazioni precedenti sulle accortezze nell'utilizzo di SlidingWindow e di FoldFunction a supporto delle WindowFunction ci hanno permesso rispettivamente di non attendere il periodo di tempo richiesto all'elaborazione del risultato (ore, giorni, settimane) e di evitare di accumulare dati per tali archi temporali che non avrebbe permesso al sistema di essere "scarico" e performante.

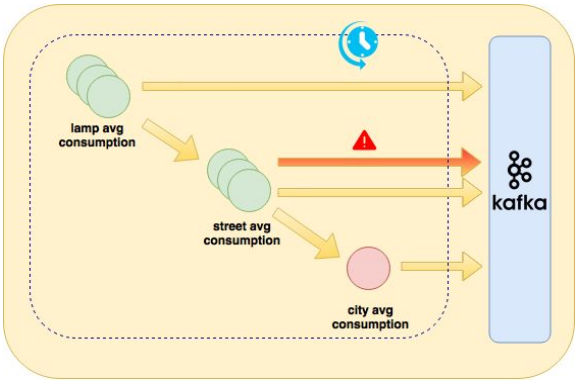


Figura 4: flusso flink calcolo della media.

Dalla figura 4 si evince come il calcolo del consumo medio per lampione o per strada venga replicato su più operatori: flink mette a disposizione l'operazione di keyBy che permette di redirezionare lo stream di dati in base a una certa chiave, in modo che ciascun operatore sia l'unico a ricevere i dati provenienti da un certo lampione o da una certa strada per evitare che il calcolo risulti inconsistente. Il calcolo della media per tutta la città converge invece in un unico operatore che effettua il calcolo per i dati già aggregati per strada. Lo schema degli operatori proposto in figura è ripetuto per le 3 fasce orarie di interesse ma per evitare di processare ripetutamente gli stessi dati il calcolo della statistica per lampione viene effettuato sfruttando la gerarchia temporale: ciascun flusso prende in input i risultati prodotti dall'aggregato temporale più piccolo. Ogni risultato delle operazioni precedenti, in base al periodo di tempo, viene utilizzato per calcolare il consumo medio di potenza per i dati aggregati su strada e città, come mostrato in figura 4. Per il calcolo della media viene utilizzata la formula che segue che viene eseguita nella FoldFunction per ogni tupla in ingresso mantenendo di volta in volta il valore aggiornato della media [20].

$$\Delta = x_i - \overline{x_{n-1}}$$

$$\overline{x_n} = \overline{x_{n-1}} + \frac{\Delta}{n}$$

Durante la computazione del calcolo dei valori delle medie per ogni strada è possibile che vengano sollevate delle eccezioni. Di volta in volta utilizzando la formula precedente l'operatore mantiene il valore del consumo medio per la strada e si vede arrivare in ingresso un consumo relativo ad un singolo lampione di quella strada. Se questo supera o è più piccolo di un certo numero di volte quello della media per la strada viene sollevato un warning. Il parametro per la generazione di warning è configurabile ed il controllo viene effettuato come segue:

$$\left(\frac{l_i}{s_{n-1}} > WARN\right) \vee \left(\frac{l_i}{s_{n-1}} < \frac{1}{WARN}\right)$$

con l_i indica il valore del consumo del lampione rapportato al valore del consumo medio della strada a cui appartene.

3.1.3 *Computation of the percentage*

Questo flusso ci permette di soddisfare la query 4 ossia di trovare la percentuale dei lampioni per strada aventi la mediana dei consumi superiore alla mediana globale. Per rispondere a questa necessità abbiamo diviso la computazione in due step: il calcolo della mediana del consumo per singolo lampione e il calcolo della mediana del consumo di tutti i lampioni. Gli operatori che effettuano tale operazione sui singoli lampioni sono replicabili con la solita accortezza di raggruppare opportunamente lo stream per id.
Questi due flussi convergono in un operatore di join che li unisce agganciando ad ogni tupla di un lampione il valore della mediana globale di tutti i lampioni. In questo modo l'operatore seguente ricevendo lo stream, raggruppato per strada, può valutare rapidamente il numero di lampioni per strada che superano il valore della mediana del consumo globale nell'arco temporale prestabilito restituendo un valore in percentuale.

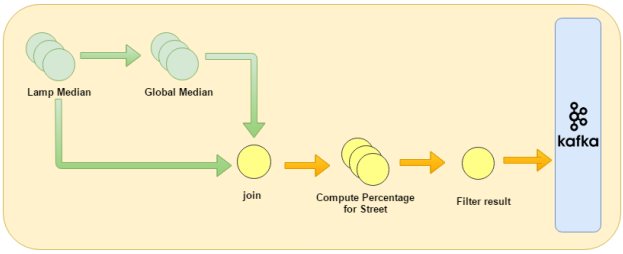


Figura 5: flusso flink calcolo della percentuale

Per calcolare la mediana abbiamo utilizzato la libreria T-Digest [6] che consente di effettuare il calcolo dei percentili on-line in maniera veloce, mantenendo un alto tasso di accuratezza ma soprattutto senza necessità di memorizzare grandi quantità di dati grazie all'utilizzo dell'algoritmo esposto in [7]. Infine il dato finale verrà emesso su un topic Kafka solo nel caso in cui sia differente dal dato precedente.

3.1.4 *StateOn warning*

Questo flusso permette di rispondere alla query 1. Tra le informazioni contenute nell'oggetto Lamp è presente anche un attributo booleano, stateOn, indicante lo stato di accensione del lampione. Con tale attributo è possibile rilevare malfunzionamenti considerando il consumo di potenza: se il lampione risulta essere

spento ci si attende un consumo nullo di potenza. Nel caso in cui questo risulti positivo viene sollevato un warning per il lampione corrispondente.

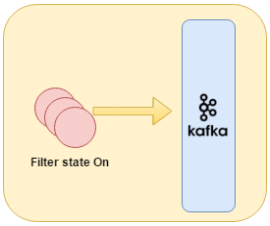


Figura 6: flusso flink rilevazione degli errori

3.2 Control-System

Il sistema di controllo, come descritto precedentemente ha il compito di verificare se l'intensità di un lampione è adatta per consentire un'ottima visibilità in base al traffico veicolare e all'intensità di luce naturale presenti. Il sistema riceve da Kafka le tuple contenenti informazioni sullo stato dei lampioni e dell'intensità di luce naturale, ed in base alla percentuale di traffico sulla strada, ottenuto interrogando il sistema del traffico tramite API REST, calcolerà degli aggiustamenti da inoltrare ai lampioni nuovamente tramite topic Kafka.

L'oggetto considerato dal sistema di Controllo per quanto riguarda le informazioni sui lampioni rimane lo stesso presentato nel sistema di monitoraggio, invece le tuple contenenti informazioni derivanti dai sensori di luce naturale si presenteranno anch'esse come oggetti JSON aventi la seguente composizione:

```
{
        "lightSensorId": "long",
        "address":"string",
        "lightIntensity":"(0,1) double",
        "timestamp":"u_int",
},
```

Assumiamo che l'id del sensore di luce naturale sia lo stesso del lampione sul quale è collocato. Anche l'aggiustamento verrà scritto su kafka come oggetto JSON:

```
{
        "lampId": "long",
        "lightIntensityAdjustment":"(0,1) double",
},
```

Al local-controller sarà sufficiente ricevere solamente l'id del lampione per poter dirottare il segnale di aggiustamento verso il corretto lampione, avendo mappato al proprio interno la corrispondenza tra id del lampione e corrispondente indirizzo ip.

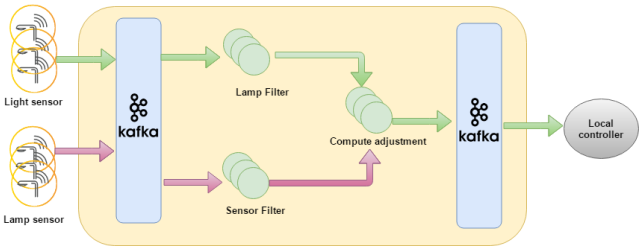


Figura 7: architettura del sistema di controllo.

Dalla figura è possibile notare che anche in questo sistema, sono presenti delle operazioni di filtraggio per poter eliminare dal sistema tuple che non siano ben formate, e che, come per il sistema di monitoraggio, è presente in ingresso ed in uscita un interfacciamento con Kafka.

3.2.1 *Computation of light adjustment*

Questo flusso risponde alle query 5 e 6, il sistema di controllo prende in ingresso due flussi differenti, quello della luce naturale e quello dell'intensità di luce generata dai lampioni. Non potendo assumere che le frequenze di questi dati in ingresso siano uguali, viene presa in considerazione la mediana dei valori, calcolata come nel flusso del sistema di Monitoraggio, anch'essa valutata in un intervallo temporale configurabile. In questo modo sarà possibile disaccoppiare la computazione dalla frequenza dei dati in input al sistema. Durante l'operazione di join in cui vengono convogliati i flussi, per ogni tupla viene letto da un sistema di cache, implementato come una HashMap, il valore relativo alla percentuale di traffico sulla strada dove il lampione è locato. La cache viene riempita a mano a mano che vengono processate tuple appartenenti a strade diverse. Inoltre un thread periodicamente si occuperà di effettuare richieste http GET in modo da mantenere costantemente aggiornati i valori relativi alla percentuale di traffico.

Gli aggiustamenti vengono effettuati garantendo sempre un minimo di visibilità sulle strade: se la luce naturale esterna supera il valore del 50%, i lampioni verranno sempre spenti, mentre in caso contrario ci siamo avvalsi della seguente formula per il calcolo dell'aggiustamento:

$$L + S = T,$$
$$T > S$$

dove L rappresenta la luce dei lampioni, S la luce naturale e T la percentuale di traffico, con ciò significa che l'intensità di luce totale, sia naturale che del lampione, sia pari al valore indicante la percentuale di traffico.

Ad esempio se $S = 20\%$ e abbiamo una percentuale di traffico $T = 70\%$, il lampione dovrà avere un'intensità luminosa del 50%.

Qual'ora l'intensità di un lampione non soddisfi questa equazione, verrà inviato l'aggiustamento corrispondente. Questa equazione non viene utilizzata nel caso in cui la percentuale di traffico sia minore del minimo della soglia di luminosità che dobbiamo garantire, in tal caso verrà seguita la seguente uguaglianza:

$$L + S = MIN\_PRC,$$
$$MIN\_PRC > S$$

dove MIN_PRC è appunto il minimo di percentuale luminosa che dobbiamo garantire. Ad esempio, qual ora T sia del 10% e MIN_PRC uguale al 40% (mantenendo S=20%), allora l'aggiustamento verrà inviato in modo da impostare l'intensità luminosa del lampione all 20%. Nel caso non valgano le condizioni delle equazioni presentate i lampioni riceveranno un segnale di spegnimento.

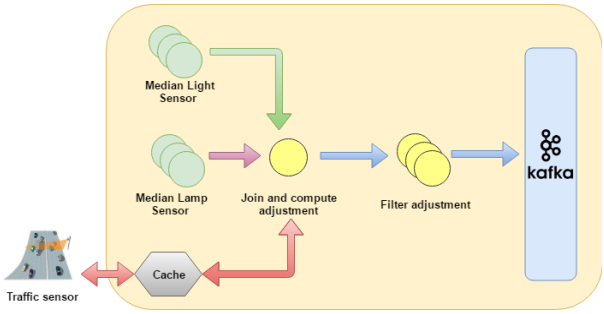


Figura 8 : flusso flink calcolo dell'aggiustamento.

Gli aggiustamenti calcolati nell'operatore di join verranno pubblicati su Kafka solo in caso in cui siano diversi da 0 altrimenti verranno scartati dal filtro a valle del sistema di controllo.

3.3 Local controller

Il Local Controller è il componente incaricato della lettura dei segnali correttivi inviati dal sistema di controllo tramite Kafka e dirottarli tramite interfaccia REST al giusto sensore al fine di poter correggere l'intensità luminosa del lampione a lui associato.
Tale processo di dispatching, viene eseguito anche per quanto riguarda le operazioni di inserimento e cancellazione di un lampione dal sistema, in quanto tale componente riceve tramite interfaccia REST il comando da eseguire su un lampione e provvede a contattare il sistema di sensori per simulare, l'inserimento o la rimozione di un lampione. In un sistema reale questo ultimo passaggio per la rimozione o inserimento di un lampione, dovrebbe essere rappresentato dall'operazione fisica sul lampione e sul sensore.
Tale componente è stato implementato tramite il framework Express[8] per applicazioni web NodeJS[9], tecnologia scelte per la loro facilità di utilizzo nella creazione di Servizi Web ed utilizza il modulo di supporto kafka-node[13] per interfacciarsi con Kafka.
Inoltre grazie a tale piattaforma è possibile rispettare il formato nativo dei dati JSON, previsto nell'architettura REST, evitando così operazioni di conversione in formati di dati differenti.

3.4 Dashboard

Come componente di data-presentation abbiamo previsto una web application per poter facilmente visualizzare i risultati prodotti dai Sistemi di Controllo e Monitoraggio. L'applicazione è strutturata come un sistema client-server, fornisce inoltre un layout responsive per l'accesso da dispositivi mobili.
Il lato server è stato implementato tramite il framework Express[8] per la piattaforma NodeJS[9], mentre il lato-client è realizzato mediante il framework AngularJS[10] per lo sviluppo di applicazioni web Single Page.
Il server è incaricato della lettura da topic Kafka degli output prodotti dai Sistemi di Controllo e Monitoraggio, e inviarli direttamente al client per la loro visualizzazione, inoltre espone alcune API REST per l'interazione con il database a supporto del sistema, in particolare per poter leggere informazioni sui lampioni ed effettuare le operazioni di inserimento e cancellazione di un lampione. Il database scelto è il database NoSQL Cloudant[11], in quanto è un database distribuito basato sul progetto CouchDB, e che rispetta il formato di dati JSON utilizzato ampiamente nell'intero sistema. Inoltre è stato scelto per la rapidità di utilizzo essendo un servizio esposto pubblicamente sulla piattaforma Cloud IBM Bluemix[12].
Sono state adottate come supporto i moduli kafka-node[13] per interfacciarsi con Kafka, socket.io[14] per l'hand off dei messaggi ricevuti da topic Kafka dal server verso il client e cloudant[15] per l'utilizzo delle API REST verso il database.
Il client, come detto, è realizzato mediante il framework AngularJS ed altre tecnologie tipiche per lo sviluppo del frontend di Web Application, come Bootstrap, HTML5 e SASS.
Nella dashboard è prevista quindi la visualizzazione di ogni tipo di output generato dai Sistemi di Controllo e di Monitoraggio, inoltre è prevista una view per la visualizzazione di tutti i lampioni monitorati dal sistema e che offre la possibilità di effettuare le operazioni di inserimento e cancellazione di un lampione.

3.5 Sistemi di supporto testing

Oltre ai sistemi sopra dettagliati, abbiamo avuto l'esigenza di

crearne altri due di supporto. Quest'ultimi ci hanno permesso di fare test prestazionali, e verificare la validità del nostro sistema a regime, soprattutto in condizioni realistiche che sarebbe stato impossibile riprodurre altrimenti.

3.5.1 Sensors system

Questo sistema simula i data stream provenienti dai lampioni e dai sensori di luce naturale. Sviluppato avvalendosi del framework Spring e utilizzando MongoDB per la persistenza: il sistema basa il suo comportamento sull'ipotesi che i lampioni siano in grado di scrivere su un topic Kafka e che ognuno di essi esponga un interfaccia REST che permetta di ricevere dal local controller le operazioni CRUD e gli aggiustamenti di intensità luminosa. Abbiamo scelto di strutturare il sistema in maniera multi-threading in modo da poter associare ad uno stesso thread un numero configurabile di sensori al fine di ottimizzare le prestazioni del sistema e poter creare un carico maggiore di dati in ingresso. I sensori vengono inizializzati a partire da un file di configurazione contenente una tupla per ciascun sensore che ne specifica i parametri rilevanti. Ogni sensore della luce esterna varia la propria intensità luminosa in base alla fascia oraria. Lo stesso comportamento è stato implementato per sensori dei lampioni con la differenza che quest'ultimi variano l'intensità luminosa anche in base agli aggiustamenti ricevuti. Inoltre ciascun lampione avrà associato un consumo di potenza in base al particolare modello di lampada. Al fine di evidenziare comportamenti erronei e analizzare il comportamento dei nostri sistemi in casi limite, sono stati creati dati incoerenti o malformati.

3.5.2 Sistemi di monitoraggio del traffico

Il traffic-system consiste in un server che simula il funzionamento di un sistema di monitoraggio del traffico veicolare. Implementato tramite Express[8] e NodeJS[9], è utilizzato per testare il pieno funzionamento del sistema di controllo il quale prevede l'interfacciamento con tale sistema tramite API REST. Il sistema invia una percentuale di traffico prefissata in base alla fascia oraria.

4 TEST

Si è scelto di effettuare due tipi di test sul sistema realizzato. Un primo test effettuato in locale solo sui sistemi di controllo e monitoraggio, per valutare le performance dei due sistemi con vari gradi di parallelismo, e con un numero prefissato di tuple. Il secondo test, è un test a regime, con le varie componenti dell'architettura caricate sulle piattaforme Cloud, fatta eccezione per il sistema di controllo e di monitoraggio, per monitorare l'occupazione di memoria.

4.1 Test locali dei sistemi Flink-based

Essendo gli output dei sistemi realizzati in Flink temporizzati dalla dimensione delle finestre, più precisamente dal parametro di slide, effettuare misurazioni su latenze e throughput delle tuple in ingresso e in uscita dal sistema sarebbe risultato poco significativo. D'altro canto avevamo necessità di capire in che modo impostare diversi parametri legati al grado di replicazione degli operatori logicamente implementati come funzionalmente replicabili e alla quantità di memoria utilizzabile dal framework Flink per garantire le migliori prestazioni.

Come anticipato, dato il particolare scorrere del tempo se un sistema Flink caratterizzato da Event Time viene "bombardato" di tuple anche aventi timestamp consecutivi distanti ore o giorni queste verrebbero processate rapidamente venendo inseriti nel flusso in ingresso watermark in base ai timestamp che Flink si vede arrivare dalle tuple in ingresso.
Da queste osservazioni si è deciso di generare da un dataset iniziale composto da dati di 100 strade ognuna avente 10 lampioni, 1000 tuple, un susseguirsi di re-inoltri di tali tuple al sistema facendo variare il timestamp di 10 secondi simulando il contatto continuo di 1000 lampioni. Il numero di contatti è stato invece impostato a 1000 in modo che i sistemi elaborassero 1 milione di tuple al variare del grado di parallelismo degli operatori

Abbiamo deciso di utilizzare un dataset di test composto da tuple provenienti da 100 strade ognuna avente 10 lampioni, per un totale di 1000 tuple: al fine di osservare le performance al variare del grado di parallelismo degli operatori ogni lampione invia dati ogni 10 secondi per 1000 volte, per un totale di 1 milione di tuple.

Questo test è stato eseguito sia per il sistema di monitoraggio che per il sistema di controllo con una variante, avendo per quest'ultimo in ingresso due diversi stream di dati, le tuple totali in ingresso risultano essere il doppio. Inoltre per il sistema di monitoraggio avente più flussi e potenzialmente più a rischio di occupazione di memoria si è scelto di monitorare il tempo di elaborazione anche al variare di due parametri di Flink:

- jobmanager.heap.mb: dimensione del heap per il JobManager in MB;

- taskmanager.heap.mb: dimensione del heap per il TaskManager in MB.

4.1.1 Test Monitoring-System

Nel grafico che segue emerge come all'aumentare del grado di parallelismo degli operatori migliorino i tempi di elaborazione delle tuple in ingresso rappresentate nel grafico come istogrammi la cui altezza indica il tempo di elaborazione in secondi. Il miglioramento sembra non essere così significativo passando da parallelismo 3 a 4 nonostante aumenti di 1 il parallelismo di tutti gli operatori logicamente replicabili. Per tali ragioni si è scelto di settare come parallelismo di default il valore 3 per tutti gli operatori del sistema di monitoraggio.

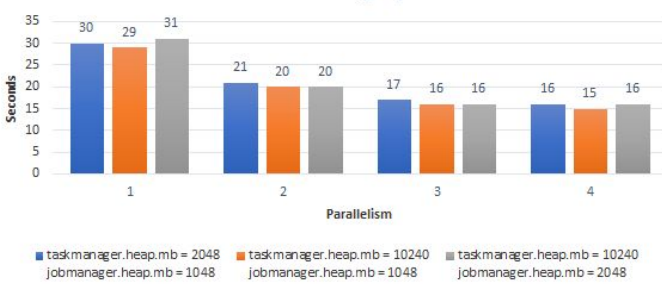


Figura 9 : Test sistema di monitoraggio

Per quanto riguarda i diversi parametri di Flink, è invece evidente come la memoria a disposizione non infici la velocità di elaborazione, visto che un aumento di memoria del TaskManager da 2048 MB a 10240 MB non altera il tempo complessivo, che risulta confrontabile per tutti i gradi di parallelismo. Lo stesso vale per il JobManager che si occupa di gestire lo stato dei diversi operatori in esecuzione.
Nonostante questa evidenza abbiamo scelto di impostare per i test a regime, le caratteristiche associate all'ultima configurazione dando la maggior quantità di memoria possibile al TaskManager e lasciando una parte di memoria esclusivamente all'utilizzo del SO come indicano le best practices di Flink [18].
I test sul monitoring sono stati effettuati con una macchina avete CPU Intel Core i7 (I7-6700HQ) 2.6 GHz - Quad Core e RAM 16 GB LPDDR3 2133MHz

4.1.2 Test Control-System

Da quanto emerso dal test sulle performance del sistema di monitoraggio, per il sistema di controllo i parametri relativi alla memoria assegnata al TaskManager ed al JobManager di Flink sono stati assegnati in accordo con l'ultima configurazione. Valutando il parallelismo degli operatori del sistema di controllo, è possibile notare che anche in questo componente l'aumentare del grado di parallelismo migliora le prestazioni.

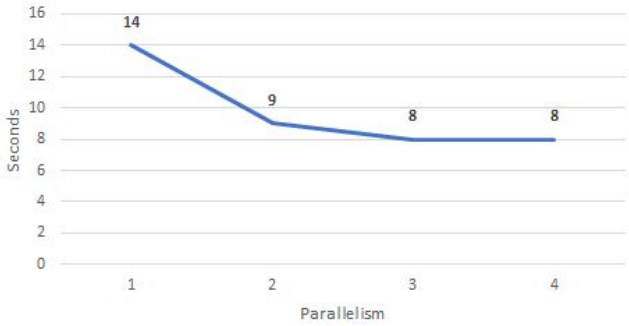


Figura 10 : Test sistema di controllo

Dal grafico riportato in figura è possibile analizzare le prestazioni con vari gradi di parallelismo. Per le stesse considerazioni fatte per il sistema di monitoraggio, anche in questo caso, è stato scelto il grado di parallelismo 3 per effettuare il test a regime. I test del sistema di controllo sono stati effettuati con una macchina avente CPU Intel Core i7 (I7-4770HQ) 2.2 GHz - Quad Core e RAM 16 GB DDR3 1600MHz.

4.2 Test a regime

Come anticipato, al fine di poter monitorare lo stato della memoria che il sistema utilizza per processare grandi quantità di dati, abbiamo scelto di effettuare un test a regime di lunga durata.
A tal fine è stato necessario deployare i componenti realizzati sulle piattaforme Cloud AWS[5] e Bluemix[12] come riportato nella tabella che segue:

System	Deploy
sensors-system	EC2 - AMI Linux t2.medium
kafka-input	EC2 - AMI Linux t2.medium
kafka-output	EC2 - AMI Linux t2.medium
traffic-system	Applicazione Cloud Foundry - SDK for Node.js
local-controller	Applicazione Cloud Foundry - SDK for Node.js
dashboard	Applicazione Cloud Foundry - SDK for Node.js

I sistemi di controllo e monitoraggio invece, solo durante la fase di test, sono eseguiti su macchine locali, per avere maggiore controllo sulla macchina fisica su cui sono eseguiti i sistemi e per

mantenere inalterate le configurazioni previste dal test sulle performance, in quanto il test a regime, serve proprio a confermare la bontà delle scelte fatte sul grado di parallelismo, anche in termini di utilizzo di memoria. Tuttavia occorre precisare, che per avere il sistema up and running, i sistemi di Controllo e Monitoraggio, sono comunque caricati su istanze EC2 - AMI Linux t2.xlarge.

Seguono le informazioni relative all'occupazione di memoria da parte delle applicazioni Flink-based raccolte tramite il supporto del tool VisualVM [21], su un test della durata di 2 ore con tuple relative a 1000 lampioni, inviate ogni 10 secondi.

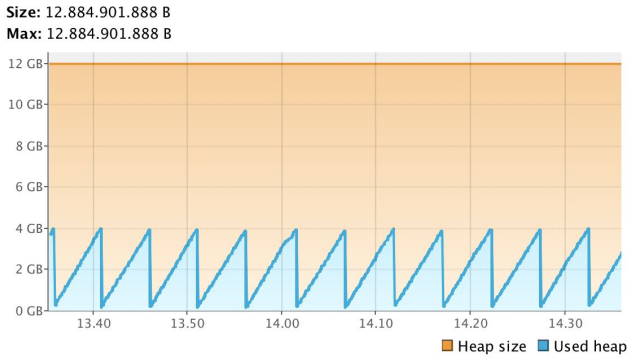


Figura 11 : Test sistema di monitoraggio a regime

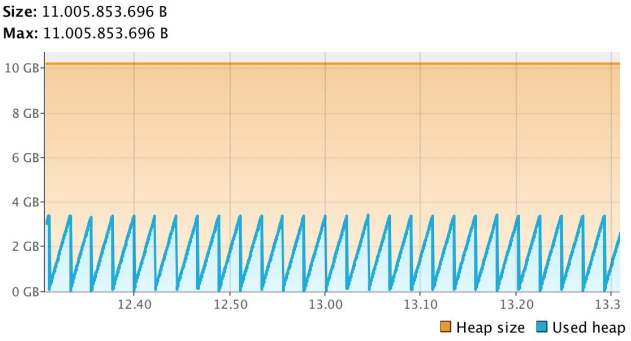


Figura 12 : Test sistema di controllo a regime

I grafici mostrano l'andamento della memoria nella seconda ora di monitoraggio; per entrambi il grafico della memoria utilizzata corrisponde ad un susseguirsi di rampe. Questo tipo di comportamento è giustificato dalla continua creazione di oggetti per l'ingresso dei dati ai sistemi fino all'intervento del Garbage Collector che libera la memoria dalle risorse non più utilizzate. La frequenza delle rampe all'interno dello stesso intervallo temporale nel sistema di controllo è molto più elevata, quasi doppia, rispetto la frequenza del sistema di monitoraggio; tale crescita di memoria è dovuta all'ingresso non solo dello stream di dati relativi alle rilevazioni sui lampioni ma anche a quelle relative alla luce naturale.

Ciò che si voleva evidenziare è che entrambi i sistemi riescano a mantenere, grazie agli accorgimenti presi sull'utilizzo di Flink, un utilizzo controllato di memoria visto le periodicità riscontrate nei grafici.

5 INSTALLAZIONE E CONFIGURAZIONE

I requisiti minimi per l'installazione e l'esecuzione del sistema sono l'aver installato:

- JDK: 1.8.0_121
- Apache Flink: 1.2.0
- Apache Kafka: 2.10-0.10.2.0

Per maggiori informazioni riguardo l'installazione, la configurazione e le esecuzioni dei vari sottosistemi rimandiamo alle procedure descritte nei file README di ogni applicazione, reperibili nelle rispettive repository github mostrate in tabella.

monitoring-system	github.com/calmar3/monitoring-system
control-system	github.com/calmar3/control-system
sensors-system	github.com/calmar3/sensors-system
local-controller	github.com/valerioditinno/local-controller
dashboard	github.com/calmar3/dashboard
dashboard-backend	github.com/calmar3/dashboard-backend
traffic-system	github.com/calmar3/traffic-system

Inoltre per maggiori dettagli relativi alle operazioni di deploy, nelle configurazioni standard utilizzate per il progetto, consultare il file *util.txt* incluso nella repository principale del progetto.

6 CONCLUSIONI, LIMITAZIONI RISCONTRATE E SVILUPPI FUTURI

Lo scopo di questo progetto è stato quello di realizzare un sistema di monitoraggio e controllo dell'illuminazione pubblica in grado di adattare dinamicamente l'intensità luminosa dei lampioni sul territorio urbano in base alle condizioni ambientali e al traffico, al fine di migliorare l'efficienza energetica e di ridurre l'inquinamento luminoso garantendo performance e scalabilità grazie l'utilizzo delle più moderne tecnologie per il data stream processing.

Tra le limitazioni riscontrate troviamo il fatto che Apache Kafka, utilizzato nella configurazione attuale in input al sistema di monitoraggio e controllo rappresenta il bottleneck delle prestazioni poiché non è in grado di trasferire le tuple che arrivano ai sensori con la stessa frequenza con cui arrivano soprattutto al crescere del numero di tuple in ingresso. Come possibile sviluppo futuro si potrebbe pensare di replicare l'istanza di Kafka in ingresso al sistema in modo da alleggerire il carico sulla singola replica, suddividendo i lampioni tra le varie repliche. In tal modo si garantirebbe un throughput maggiore in ingresso ai sistemi di monitoraggio e controllo che risulta necessario per garantire al sistema la possibilità di gestire una città di grandi dimensioni con centinaia di migliaia di lampioni.

Possibili scenari di sviluppi futuri, potrebbero portare il sistema ad avere un grado di replicazione diverso da quello attuale.
Per quanto riguarda i sistemi Flink-based la loro replicazione introdurrebbe la necessità di un secondo sistema di data stream processing a valle dei componenti replicati per svolgere il compito di unire i risultati delle computazioni dei diversi sistemi. Questa suddivisione logica in due livelli consentirebbe di distribuire il traffico in ingresso in base a dei parametri significativi, come potrebbe esserlo un insieme di strade o un quartiere, per poi avere in uscita comunque un unico risultato.

Un altro possibile scenario di sviluppo futuro, potrebbe garantire al sistema la fault tolerance. Questo sarebbe possibile tramite un meccanismo nativo di Flink, il meccanismo dei Checkpoints. Tale meccanismo consente appunto di creare dei salvataggi su storage esterni dello stato del sistema, al fine di poterlo ripristinare in caso di guasti.

Infine si potrebbe pensare di sostituire l'hashmap, che attualmente fa da cache tra il sistema di controllo e quello di traffico con un qualcosa di più performante come potrebbe essere il sistema di storage in-memory Redis.

REFERENCES

[1] M. Annunziato, G. Giuliani, N. Gozo, C. Honorati Consonni, A. Frascone F. Bucci, B. Lo Bue. 2011. *Progetto Lumière: Efficienza Energetica nell'Illuminazione Pubblica. Percorso e metodologia di sviluppo*

[2] https://www.consorzio-cini.it/index.php/it/

[3] https://kafka.apache.org/

[4] https://flink.apache.org/

[5] https://aws.amazon.com/it/

[6] RAJ JAIN and IIMRICH CHLAMTAC.*The Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations.*

[7] https://github.com/tdunning/t-digest

[8] http://expressjs.com/it/

[9] https://nodejs.org/en/

[10] https://angularjs.org

[11] https://cloudant.com

[12] https://www.ibm.com/cloud-computing/bluemix/it

[13] https://www.npmjs.com/package/kafka-node

[14] https://socket.io

[15] https://www.npmjs.com/package/cloudant

[16] https://ci.apache.org/projects/flink/flink-docs-release-1.2/

[17] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html

[18] https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/config.html

[19] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html

[20] Donald Knuth, Art of Computer Programming, Vol 2, "Seminumerical Algorithms", section 4.2.2

[21] https://visualvm.github.io