Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Un'infrastruttura distribuita per l'acquisizione e la visualizzazione interattiva dei dati dell'LHC

Relatore:
Chiar.mo Prof.
Panzieri Fabio

Correlatore:
Dott.
Copy Brice

Presentata da:
Piccinelli Flavio

Sessione II
2014-2015

Indice

Introduzione

Il progetto descritto in questo elaborato è lo sviluppo di un sistema distribuito di acquisizione e visualizzazione interattiva di dati. Tale sistema è utilizzato al CERN[1] al fine di raccogliere i dati relativi al funzionamento dell'LHC[2] e renderli disponibili al pubblico in tempo reale[3], tramite una dashboard web *user-friendly*.

L'infrastruttura sviluppata è basata su di un prototipo progettato ed implementato al CERN nel 2013 [1]. Come descritto nell'articolo di presentazione di questo prototipo, negli ultimi anni il CERN è diventato sempre più popolare presso il grande pubblico. Per questo si è sentita la necessità di rendere disponibili in tempo reale, ad un numero sempre maggiore di utenti esterni allo staff tecnico-scientifico, i dati relativi agli esperimenti effettuati e all'andamento dell'LHC.

Le problematiche da affrontare per realizzare ciò riguardano sia i produttori dei dati, ovvero i dispositivi dell'LHC, sia i consumatori degli stessi, ovvero i client che vogliono accedere ai dati. Da un lato, i dispositivi di cui vogliamo esporre i dati sono sistemi critici che non devono essere sovraccaricati di richieste, che risiedono in una rete protetta ad accesso limitato ed utilizzano protocolli di comunicazione e formati dati eterogenei. Dall'altro lato, è necessario che l'accesso ai dati da parte degli utenti possa av-

[1] Organizzazione Europea per la Ricerca Nucleare, con sede a Ginevra.

[2] Acronimo di Large Hadron Collider, infrastruttura ove avvengono la maggior parte degli esperimenti condotti al CERN.

[3] Per questo progetto con "tempo reale" si intende *soft real time*, ovvero sono accettabili ritardi nell'ordine del minuto.

venire tramite un'interfaccia web (o dashboard web) ricca, interattiva, ma contemporaneamente semplice e leggera, fruibile anche da dispositivi mobili.

La soluzione iniziale a questi problemi è stata di pubblicare i dati provenienti dall'acceleratore sotto forma di immagini statiche. Più precisamente, queste immagini sono *screenshot* di varie schermate di comuni sistemi SCADA[4], contenenti diagrammi e grafici, aggiornate più volte al minuto e pubblicate su un server web.

Questa prima soluzione, però, come vedremo più in dettaglio in seguito, non soddisfa alcuni dei requisiti cercati. Innanzitutto un'infrastruttura di questo tipo non è scalabile. Inoltre la presentazione dei dati è estremamente statica e non interattiva. I dati distribuiti in questa maniera non sono in formato cosiddetto *machine readable*, per questo è impossibile modificare la visualizzazione dei dati, se non andando a riconfigurare gli applicativi che generano le immagini. Inoltre, in questo modo, l'utente non ha la possibilità di utilizzare questi dati in maniera diversa dalla mera visualizzazione. Esiste, ad esempio, un progetto che utilizza i dati prodotti dall'acceleratore e da noi diffusi in modo non convenzionale, ovvero per creare opere di arte digitale[5].

Per colmare le lacune che questa soluzione presenta, come anticipato, nel 2013 è stata progettata un'infrastruttura composta da tre *layer*. Il primo è costituito da *agenti*, ovvero applicativi Java che si interfacciano con i diversi dispositivi dell'LHC e raccolgono i dati prodotti. Il secondo strato è un *web server* a cui gli agenti inviano i dati e a cui gli utenti possono richiederli. Infine abbiamo il layer di presentazione, ovvero una *dashboard* in cui i dati vengono visualizzati tramite grafici e diagrammi.

Il lavoro qui presentato si pone come obiettivo la riprogettazione di alcuni elementi di questo prototipo al fine di creare un'infrastruttura che soddisfi nuovi requisiti.

[4]Acronimo di Supervisory Control and Data Acquisition, sistemi utilizzati per monitorare e controllare infrastruttura industriali.

[5]Si veda http://www.lechantdesparticules.net/.

Per quanto riguarda il front-end, l'obiettivo è di avere un'interfaccia web (o dashboard) *mobile friendly*, facilmente configurabile e composta da *widget* (componenti ove i dati sono presentati agli utenti) anch'essi configurabili, riutilizzabili ed interattivi. Si vuole dare la possibilità, ad utenti esperti, di configurare, tramite semplici file JSON, le pagine della dashboard ed i widget presenti in ognuna di esse. Oltre a ciò, è opportuno che questi widget siano esportabili e facilmente integrabili in altre pagine web. Inoltre si vuole dare la possibilità agli utenti di interagire con i widget, richiedendo dati passati, cambiando la finestra temporale o la risoluzione dei dati visualizzati. Gli utenti devono anche poter richiedere al sistema i dati "grezzi" e non solo elaborati e presentati tramite grafici e diagrammi. Infine, il sistema di comunicazione tra web server e client dovrebbe essere leggero e permettere di inviare i dati in streaming, in modo da non obbligare il client ad effettuare frequenti richieste, spesso inutili, che appesantiscono l'intero sistema. Per questi motivi, il lato client, rispetto al suddetto prototipo, è stato interamente riprogettato.

Per quanto riguarda il back-end, è necessario avere un'infrastruttura modulare, scalabile orizzontalmente e che distribuisca in maniera efficiente i dati raccolti dalle diverse sorgenti ai web server. Innanzitutto, come abbiamo detto, i produttori dei dati sono i dispositivi dell'LHC, che utilizzano protocolli di comunicazione diversi e formati dati eterogenei. Questi dati devono quindi essere raccolti da connettori specializzati, ognuno dei quali si interfaccia con una sorgente dei dati. Quest'ultimi devono poi essere resi il più possibile omogenei e trasmessi ad uno o più server web. Vediamo in seguito le caratteristiche che il back-end di comunicazione dell'infrastruttura deve avere per i motivi suddetti. Innanzitutto deve permettere di creare un cluster, dal quale sia possibile rimuovere o aggiungere nodi in maniera trasparente al sistema, il quale sarà composto da uno o più connettori e da uno o più web server. Inoltre dovrebbe implementare il pattern di comunicazione *publish/subscribe*, in modo da distribuire in maniera efficiente i dati da un nodo del cluster (un connettore) a tutti quelli che ne fanno richiesta (i web server). Infine, per

rendere il sistema più reattivo, è necessario che il back-end di comunicazione permetta di fare il caching dei dati trasmessi in maniera più efficiente e trasparente possibile.

Data la natura del progetto, abbiamo quindi adottato un processo di sviluppo software di tipo *incrementale*. In particolare abbiamo utilizzato il framework di sviluppo *scrum*. Abbiamo adottato questo framework, afferente alla classe di processi software *agile*, principalmente per tre motivi.

Innanzitutto, essendo il team di sviluppo molto piccolo, la coordinazione tra i membri del team è molto semplice e non necessita di particolari accorgimenti. Un processo *agile* si presta bene ad essere utilizzato in queste circostanze.

Inoltre, essendo questo un progetto innovativo, si è capito fin da subito che sarebbe stato difficile per i clienti definire, in maniera precisa, dettagliata e formale, tutti i requisiti che il sistema avrebbe dovuto soddisfare. Per questo è utile procedere nello sviluppo del progetto in maniera iterativa ed incrementale, definendo cicli di breve durata che permettano, di volta in volta, di verificare con i clienti quanto è appena stato implementato e qual è la prossima priorità. Nella terminologia *scrum*, questi cicli hanno il nome di *sprint*.

Infine, dato che si vuole mantenere l'architettura del sistema sostanzialmente invariata rispetto al prototipo di partenza, si devono re-implementare gradualmente parti di un'infrastruttura già esistente. Anche questo porta alla necessità di adottare una metodologia di sviluppo che permetta di definire cicli di progettazione ed implementazione, ognuno dei quali porti al miglioramento di una parte dell'infrastruttura immediatamente integrabile con il sistema esistente. In particolare, abbiamo proceduto alternando *sprint* di perfezionamento del front-end e del back-end; questo perché, se da un lato è necessario che i due lati dell'infrastruttura siano il più possibile disaccoppiati, dall'altro è fondamentale mantenere, ad ogni passo dello sviluppo del sistema, la sua coerenza.

Come vedremo più in dettaglio nel secondo capitolo, infatti, abbiamo inizialmente pensato a come soddisfare i requisiti client. Al fine di avere una dashboard modulare e widget con le caratteristiche suddette, abbiamo deciso di utilizzare le API OpenSocial. Questo framework, in breve, permette di realizzare pagine web (i widget), incapsulate in *iframe*, a cui è possibile passare parametri definendoli nella URL dell'iframe stesso. I parametri, e conseguentemente la URL, possono essere impostati da un container OpenSocial, in questo caso la dashboard. D'altra parte, una volta creata la URL, è possibile integrarli in altre pagine web come qualsiasi iframe.

In merito al protocollo di comunicazione tra client e server, abbiamo deciso di utilizzare, come nel prototipo di partenza, il framework Atmosphere, integrandolo in maniera migliore nel sistema. Esso, composto da un modulo client e uno server, implementa il pattern di comunicazione *publish/subscribe* utilizzando i protocolli REST o WebSocket ed incapsulandoli in un unica interfaccia. Questo ci dà la possibilità, da un lato, di servire client datati che non supportano i WebSocket, dall'altro, di poter utilizzare quest'ultimi al fine di avere una comunicazione leggera, bidirezionale e che supporti notifiche push.

Per quanto riguarda il lato server, è stato ereditato dal precedente progetto il concetto di agenti specializzati, ovvero abbiamo un diverso applicativo per ogni protocollo e formato dati. In altre parole, l'infrastruttura ha tanti agenti quante sono le diverse sorgenti da cui colleziona i dati. Per contro, è stato modificato il protocollo di comunicazione tra agenti e server web. Al fine di avere una comunicazione efficiente tra agenti e web server, con cui si possa fare il caching dei dati inviati e che al contempo permetta di creare un'infrastruttura facilmente scalabile, abbiamo adottato Hazelcast. Esso è un *in-memory data grid*, ovvero un sistema che permette di creare un database non persistente distribuito in un cluster di macchine. Hazelcast implementa, come richiesto, il pattern di comunicazione *publish/subscribe*, ovvero permette di inviare dati in multicast da un nodo del cluster agli altri in maniera efficiente.

Infine, al fine di supportare l'interattività di cui abbiamo parlato sopra, è nata la necessità di conservare i dati collezionati dai diversi dispositivi in un database a rapido accesso e su cui si possano fare semplici calcoli in tempo reale. Per far fronte a questa necessità, è stato adottato Elasticsearch, un potente database distribuito che offre avanzate funzionalità per la ricerca, aggregazione e filtraggio dei dati.

Nel presente elaborato approfondiremo tutti gli argomenti di cui abbiamo parlato. Inizieremo illustrando brevemente l'ambiente in cui si colloca il nostro progetto, ovvero il CERN ed in particolare l'LHC; in seguito descriveremo i sistemi SCADA, in cosa il nostro sistema differisce da quelli presenti sul mercato e quindi cosa ci ha spinti a creare un nuovo prodotto anziché adottarne uno già sviluppato. Nel secondo capitolo riprenderemo più dettagliatamente i requisiti che il sistema cercato deve soddisfare, quali sono state le soluzioni precedenti ed in quali aspetti non soddisfano i nostri obbiettivi. Il terzo capitolo tratterà della progettazione del nuovo sistema. Vedremo il processo di sviluppo adottato, l'architettura e le tecnologie utilizzate per realizzare il progetto. In particolare riprenderemo i tool sopra citati, descrivendo il loro funzionamento e il loro ruolo all'interno dell'infrastruttura. Nel capitolo successivo vedremo poi come queste tecnologie sono utilizzate ed integrate per implementare il nostro sistema. A questo punto, nel quinto capitolo delineeremo ed analizzeremo i test effettuati per validare l'infrastruttura. Infine, nell'ultimo capitolo, discuteremo delle performance del sistema e dei molteplici sviluppi possibili, concentrandoci sui punti di forza e sulle lacune ancora presenti nell'infrastruttura realizzata.

Capitolo 1

Il CERN e i sistemi SCADA

Iniziamo questo elaborato andando a presentare lo scenario in cui il nostro progetto si va ad inserire, ovvero il CERN, l'LHC ed i sistemi SCADA già in uso al fine di monitorarlo e gestire i suoi dispositivi.

1.1 Il CERN

Il CERN, *Organizzazione Europea per la Ricerca Nucleare*, è il principale centro di ricerca mondiale sulla fisica delle particelle ed in particolare sulla fisica delle alte energie. L'organizzazione, fondata nel 1954 da 12 paesi europei, è sempre stata all'avanguardia nella ricerca scientifica di base nel campo della fisica nucleare. Da qualche anno, grazie alle sofisticate apparecchiature presenti, sono condotti progetti di ricerca ed esperimenti in altri campi della scienza, come ad esempio la medicina, la meteorologia, etc[1]. Si ricorda inoltre che, proprio al CERN, grazie al progetto "WorldWideWeb", è stato inventato il WEB come lo conosciamo ora [2].

Nel corso degli anni sono state costruite infrastrutture sempre più innovative e performanti per effettuare svariati esperimenti nel campo della fisica

[1]Si veda il sito http://knowledgetransfer.web.cern.ch/life-sciences/from-physics-to-medicine per le applicazioni mediche della fisica e il sito http://cloud.web.cern.ch/ per i progetto di ricerca sulle interazioni tra raggi cosmici e clima

delle alte energie. In particolare, a partire dal 1957, sono stati costruiti diversi acceleratori sempre più potenti ed evoluti, ognuno dei quali utilizza i precedenti come iniettori delle particelle. L'ultimo di questi acceleratori è appunto l'LHC, oggi utilizzato per effettuare i principali esperimenti condotti al CERN. Per informazioni più approfondite sulla storia e la struttura del CERN si può far riferimento ai siti web ufficiali `http://home.cern/about` e `http://press.web.cern.ch/`.

1.2 Il Large Hadron Collider

L'LHC, acronimo di *Large Hadron Collider*, è il più grande acceleratore di particelle finora realizzato. Come abbiamo detto, è attualmente utilizzato per condurre gli esperimenti principali sulla fisica delle particelle condotti al CERN. In figura 1.1 è schematizzato il complesso degli acceleratori.

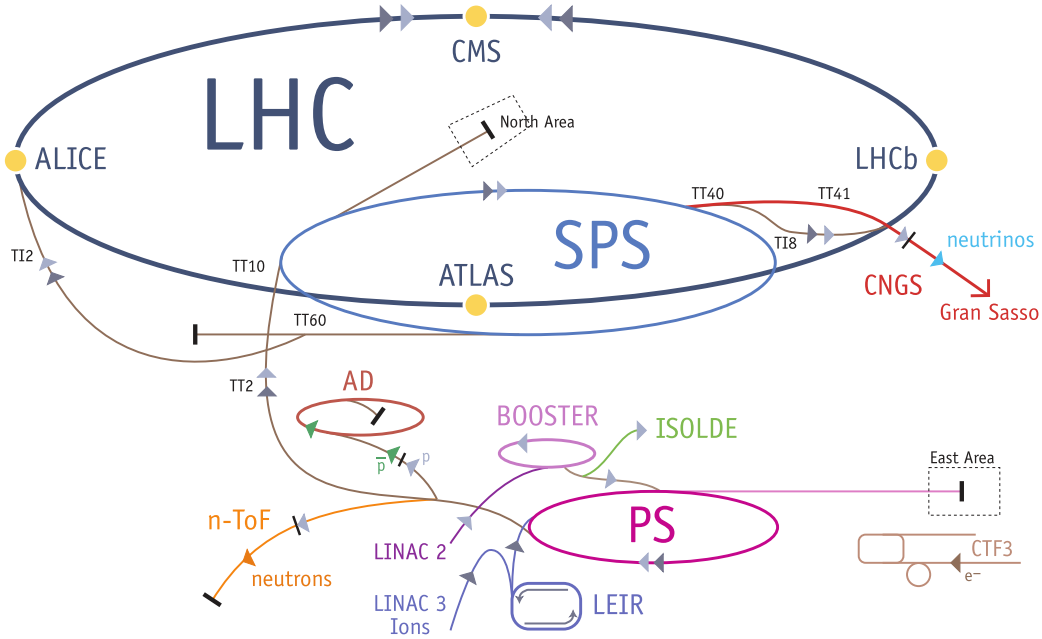


Figura 1.1: Schema del complesso degli acceleratori. Il più grande e nuovo è l'LHC. (fonte: CERN)

L'acceleratore è alloggiato in una cavità sotterranea ad anello lunga 27 chilometri. Qui, grazie alla complessa infrastruttura composta dai precedenti acceleratori, due fasci di *adroni*[2] vengono iniettati all'interno di due tubi circolari ed accelerati a velocità prossime a quelle della luce. I due fasci, una volta accelerati, vengono fatti collidere in quattro punti predeterminati dell'anello. In questi punti si trovano i rivelatori dei quattro esperimenti principali condotti grazie all'LHC. Per accelerare le particelle, nell'LHC sono presenti più di 1600 magneti, i quali, per creare il campo magnetico idoneo ad accelerarle, devono assumere la caratteristica di superconduttori. Per essere superconduttori, i magneti devono essere raffreddati e mantenuti a temperature prossime allo zero assoluto, ovvero al di sotto dei due gradi kelvin. Questo risultato si ottiene grazie ad un complesso sistema idraulico in cui, tramite diverse tecniche, l'elio liquido viene raffreddato a quelle temperature.

Già da questa breve descrizione, è evidente che l'infrastruttura dell'acceleratore, anche senza contare i rivelatori dei quattro esperimenti, è piuttosto complessa, composta da numerosi impianti, a loro volta costituiti di svariati dispositivi interoperanti. Le infrastrutture da monitorare, gestire e delle quali vogliamo raccogliere e distribuire i dati tramite il sistema presentato in questo elaborato, sono molteplici. Vediamo quelle principali. Innanzitutto bisogna monitorare i fasci di particelle, il loro stato e se stanno avvenendo o meno collisioni. Abbiamo poi l'insieme dei magneti utilizzati, come abbiamo detto, per accelerare le particelle. Di questi è necessario monitorare temperatura e allineamento. C'è poi l'impianto di criogenia, utilizzato per raffreddare i magneti. Fondamentale è l'impianto delle pompe a vuoto, preposto a mantenere il vuoto all'interno dei tubi in cui circolano le particelle ed evitare che avvengano collisioni indesiderate. Infine abbiamo l'impianto di alimentazione elettrico, utilizzato per alimentare l'intero sistema.

Come vedremo nella prossima sezione, l'acquisizione ed elaborazione dei dati di questi impianti, viene attualmente effettuata tramite sistemi di controllo industriale o sistemi SCADA.

[2]Particelle sub-atomiche, nel caso dell'LHC sono protoni e ioni pesanti.

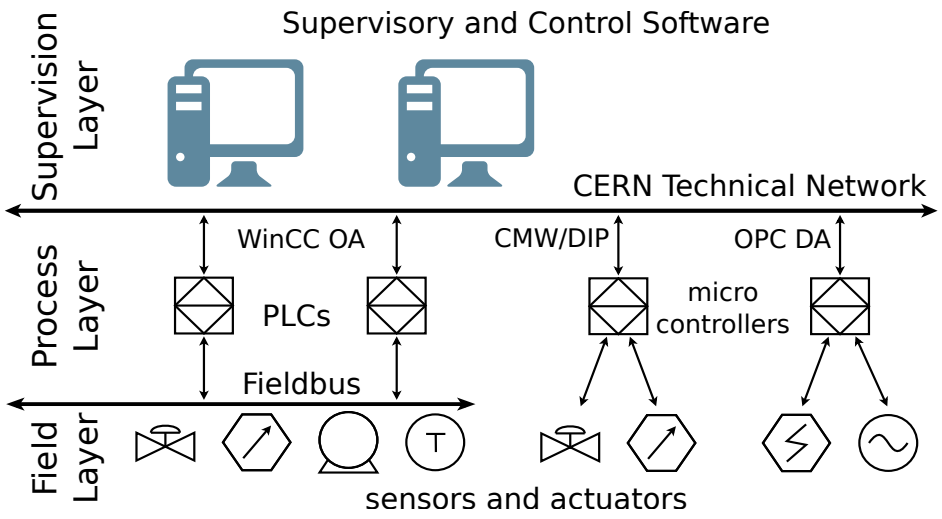


Figura 1.2: Architettura dei sistemi SCADA al CERN

1.3 I sistemi SCADA

Con il termine SCADA, acronimo di *Supervisory Control and Data Acquisition*, si indicano sistemi distribuiti, utilizzati al fine di raccogliere, analizzare e visualizzare in tempo reale i dati di infrastrutture e dispositivi industriali remoti. Grazie a questi dati, gli applicativi di controllo possono notificare agli operatori malfunzionamenti dei sistemi monitorati ed eventualmente intervenire sui dispositivi stessi grazie all'azione di microcontrollori.

1.3.1 Architettura di un sistema SCADA

In generale un'infrastruttura SCADA è composta da molteplici componenti. In figura 1.2 è schematizzata la tipica architettura a tre *layer* di un sistema SCADA.

Innanzitutto, il livello più basso è costituito da uno o più sensori che effettuano misurazioni fisiche su ciò che si vuole monitorare ed attuatori utilizzati per interagire con i vari macchinari. Nel caso dell'LHC, per fare un esempio, ci sono sensori per la misurazione della temperatura e della

pressione dei liquidi criogenici e valvole, pompe o altro per controllare il flusso e la pressione di questi liquidi.

In seguito è presente il middleware che mette in comunicazione i dispositivi elettronici dell'acceleratore con gli applicativi di controllo. Questo layer è composto da microcontrollori o comunque PC industriali, preposti a raccogliere i dati dai dispositivi a cui sono connessi tramite *bus* di comunicazione di basso livello e a renderli disponibili alle applicazioni di acquisizione ed analisi dati tramite la rete, solitamente una LAN, grazie a protocolli di alto livello basati su TCP. Sono inoltre utilizzati per interagire con i dispositivi a cui sono connessi; per esempio tramite essi si possono controllare interruttori, pompe, valvole, etc. Solitamente tutti questi dispositivi, essendo sistemi critici, risiedono in una rete schermata e controllata, in cui il traffico dati è predicibile. Un microcontrollore, ad esempio, può gestire non più di dieci connessioni contemporaneamente.

Infine l'ultimo strato di un'infrastruttura SCADA è composto dai supervisori. Questi sono tipicamente software incaricati di raccogliere ed analizzare i dati provenienti da più microcontrollori. I dati sono poi presentati agli operatori che, in base all'andamento del sistema, decidono le azioni da intraprendere.

1.3.2 Protocolli utilizzati

Nonostante la presenza del middleware intermedio, in un complesso di infrastrutture esteso come l'LHC, alcune provenienti da produttori industriali, altre costruite internamente al CERN, è evidente che i protocolli di comunicazione di alto livello utilizzati sono molteplici.

I microcontrollori, o PLCs, fabbricati da Siemens, comunicano utilizzando protocolli proprietari, sviluppati da Siemens stessa, che è possibile utilizzare tramite i driver di Siemens WinCC OA (Windows Control Center, Open Architecture). D'altra parte, molti sensori dell'acceleratore sono connessi a computer Linux embedded, i quali comunicano utilizzando protocolli

sviluppati internamente al CERN nell'ambito del progetto CMW (Controls Middleware) come il protocollo RDA (Remote Device Access). Un altro protocollo utilizzato per comunicare con i dispositivi dell'acceleratore, in particolare quelli relativi ai sistemi elettrici, è il protocollo OPC DA (Open Platform Communication, Data Access). Ultimo protocollo di nostro interesse è DIP, utilizzato per inviare e ricevere note informative sull'andamento dell'acceleratore.

Tutti i protocolli sopra elencati, funzionano utilizzando un'architettura client-server. Nello specifico, i client sono le applicazioni di supervisione di un sistema SCADA, e i server, nel caso di WinCC OA e OPC DA sono i microcontrollori stessi, nel caso di RDA sono sistemi Linux embedded connessi, tramite un bus di comunicazione industriale, a più sensori ed attuatori dell'acceleratore.

1.3.3 Supervisory and Control software

I software di supervisione e controllo, che costituiscono il livello più alto di un sistema SCADA, sono applicativi che, come abbiamo detto, servono a raccogliere, analizzare e presentare i dati dei dispositivi di un'infrastruttura industriale. Esistono svariati prodotti di questo tipo, quello maggiormente utilizzato al CERN è *UNICOS* (UNified Industrial Control System). Questo software, sviluppato internamente al CERN, è un framework per la progettazione di sistemi e processi di controllo industriale. Un altro software largamente utilizzato è *LabVIEW*, sviluppato da National Instruments. Entrambi permettono di creare sistemi di supervisione molto complessi, completi e facilmente estendibili con connettori per svariati protocolli di comunicazione [3, 4].

Questi software, pur adempiendo alla funzione principale, ovvero visualizzare in tempo reale dati provenienti da diversi dispositivi industriali, non sono adatti ai nostri scopi per un motivo cruciale: sono applicazioni desktop. Noi vogliamo avere interfacce web standard, accessibili da un qualsiasi web

browser senza dover costringere gli utenti ad installare software sul proprio PC. Inoltre presentano molte funzionalità per noi superflue, come un'avanzata analisi dei dati o la possibilità di controllare ed agire sui dispositivi dell'LHC.

Negli ultimi anni, comunque, le maggiori industrie del settore hanno iniziato a sviluppare sistemi di supervisione basati su interfacce web. Siemens, per esempio, ha sviluppato un web server WinCC OA, che mette a disposizione un'interfaccia web configurabile tramite la quale è possibile interagire con tutte le risorse gestibili da un client WinCC OA classico. Una soluzione di questo tipo, però, seppur si avvicina alle nostre necessità, non è comunque soddisfacente. Innanzitutto, essendo soluzioni proprietarie, sono poco modulari e difficilmente integrabili con altri protocolli di comunicazione. Inoltre, anche se fossero facilmente estendibili, questi sistemi sono pensati per consentire l'accesso remoto ai dispositivi industriali da parte di utenti esperti del settore. Questo presupposto porta ad un problema nell'adottarli per i nostri scopi: questi sistemi, essendo pensati per essere utilizzati da un numero relativamente ristretto di utenti, sono poco scalabili. Il loro target, in termini di numero di utenti da servire, è nell'ordine delle centinaia. Il nostro, volendo servire il grande pubblico come un qualsiasi sito web, è nell'ordine delle decine di migliaia. La scarsa scalabilità è data dal fatto che, come abbiamo visto, i dispositivi a cui questi sistemi si connettono possono gestire poche connessioni contemporanee. Per distribuire i dati ad un grande numero di utenti, è quindi necessario avere un modo per replicare i dati e farne il broadcast. I software SCADA ne sono attualmente sprovvisti.

Va menzionato *ELVis*, un software di analisi e visualizzazione dati in via di sviluppo da parte di Siemens. Questo sistema si avvicina molto alle nostre necessità di scalabilità e configurabilità e fornisce inoltre avanzate funzionalità per l'analisi off-line e on-line dei dati. Tuttavia, oltre ad essere un software non ancora rilasciato in versione stabile, esso può utilizzare solamente protocolli di comunicazione sviluppati da Siemens, e non è possibile aggiungere connettori per altri protocolli.

Per questi motivi si è preferito non adottare una soluzione proprietaria, o comunque già presente sul mercato, e sviluppare un sistema internamente al CERN: la “LHC Dashboard”.

Capitolo 2

La "LHC Dashboard"

Dopo aver descritto lo scenario di applicazione del sistema e vista l'impossibilità di utilizzare prodotti già sviluppati, introduciamo formalmente la piattaforma che vogliamo realizzare. Iniziamo definendo meglio i requisiti, citati nell'introduzione, che il sistema deve soddisfare. Dopo aver chiarito gli obiettivi, descriviamo brevemente le precedenti soluzioni proposte, analizzando le lacune che presentano. Nel capitolo successivo passeremo alla descrizione del sistema da noi sviluppato.

2.1 Requisiti

Riprendiamo quindi, in maniera più dettagliata e formale, i requisiti che il sistema deve rispettare, andandoli a suddividere, come di consueto, in requisiti funzionali e non funzionali. Va detto che, essendo l'infrastruttura realizzata un miglioramento di sistemi precedenti, i requisiti erano già stati definiti. Noi ci siamo limitati a riprenderli e, per meglio delineare il progetto, abbiamo elencato i requisiti non funzionali.

2.1.1 Funzionali

Vediamo qui di seguito la lista di requisiti funzionali che il sistema deve soddisfare.

1. Visualizzare in tempo reale i dati raccolti dall'LHC tramite una dashboard web, sotto forma di grafici, diagrammi o immagini statiche.
2. Diffondere in tempo reale i dati raccolti dall'LHC in formato *machine readable*.
3. L'interfaccia web deve essere *mobile friendly*, ovvero deve essere fruibile anche da dispositivi mobili.
4. I grafici e diagrammi con cui sono presentati i dati devono essere esportabili ed integrabili in pagine web esterne al CERN.
5. I grafici e diagrammi devono essere interattivi. Es.: nei grafici deve essere possibile zoomare in un'area a scelta.
6. Deve essere possibile modificare agevolmente la struttura dell'interfaccia web e dei grafici o diagrammi che contiene mediante semplici file di configurazione.

Casi d'uso

Dopo aver elencato schematicamente i requisiti funzionali generali, andiamo a descrivere, aiutandoci con il diagramma UML in fig. 2.1, quali sono gli attori e come devono poter interagire col sistema.

Abbiamo due tipologie di attori: utenti generici e utenti esperti. I primi devono poter accedere alla dashboard e visualizzare i dati. Essi possono anche richiedere al sistema i dati grezzi, ovvero in formato *machine readable*. La seconda tipologia di utenti è preposta alla configurazione della dashboard. In particolare deve impostare la struttura della dashboard con grafici e diagrammi e i dati visualizzati in ognuno di essi.

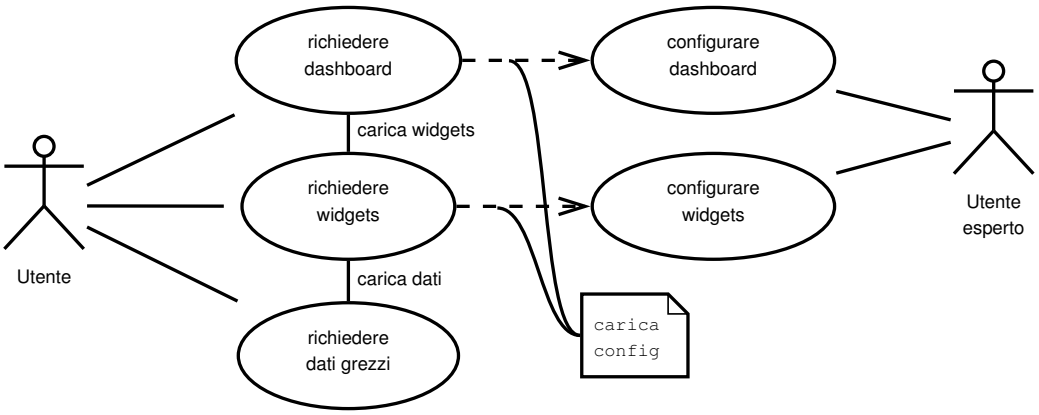


Figura 2.1: Diagramma dei casi d'uso del sistema

2.1.2 Non funzionali

Stabiliti i requisiti funzionali ed i casi d'uso, vediamo quali requisiti non funzionali sono stati definiti. Per maggiore chiarezza li suddividiamo in requisiti del front-end e del back-end.

Front-end

1. L'interfaccia deve essere sviluppata utilizzando tecnologie web standard e sviluppata utilizzando un *responsive design*.

2. Per essere fruibile da dispositivi mobili, i dati visualizzati in ogni pagina devono essere pochi, se necessario devono essere filtrati ed aggregati (dal back-end).

3. Grafici e diagrammi devono essere incapsulati in widget configurabili, riutilizzabili ed esportabili come comuni iframe.

4. La configurazione delle pagine, dei widget contenuti in esse, la loro disposizione e la configurazione dei widget stessi (sia in termini di dati visualizzati sia in termini estetici) deve avvenire tramite uno o più file in formato JSON presenti nel DFS (Distributed File System) del CERN.

5. La configurazione della struttura della dashboard (ovvero la gerarchia tra pagine) deve riflettere la gerarchia del DFS (a partire da una data cartella), ove risiedono anche i file di configurazione suddetti.

Back-end

6. L’infrastruttura deve essere scalabile orizzontalmente, ovvero deve essere semplice aggiungere o rimuovere un server web.

7. La trasmissione dei dati al client deve avvenire in maniera push, in modo da inviare nuovi dati il più velocemente possibile.

8. È necessario avere un modo per immagazzinare i dati raccolti dai diversi dispositivi.

9. Bisogna avere un modo per filtrare ed aggregare in tempo reale i dati richiesti dal client, in modo da inviargli una quantità di dati significativa ma non eccessiva. Es.: in una serie temporale si vogliono avere al massimo 500 punti, sia che si stia visualizzando i dati di un giorno o di un mese.

10. I dati devono essere raccolti da diverse risorse, interfacciandosi con diversi protocolli. È opportuno avere un modo semplice per estendere l’infrastruttura con connettori per nuovi protocolli.

2.2 Le soluzioni precedenti

Ora, avendo chiari gli obiettivi ed i requisiti che il sistema deve soddisfare, andiamo a riprendere più dettagliatamente le soluzioni che sono state sviluppate per rispondere a queste necessità.

Figura 2.2: Home screen di Vistar

2.2.1 Vistar e altri tool

La soluzione iniziale, ed ancora largamente utilizzata, è Vistar[1]. Questo sistema è costituito da un web server e da un'applicazione SC (Supervisione e Controllo) come quelle precedentemente nominate. Questa applicazione invia al web server, ad intervalli regolari, immagini contenenti i dati relativi al funzionamento dell'LHC. Più precisamente, queste immagini sono *screenshot* delle schermate dell'applicazione stessa, ognuna delle quali contenente diagrammi, grafici e tabelle tramite cui sono presentati i dati. Inoltre, il web server mette a disposizione degli utenti un'interfaccia web generata tramite uno script PHP. Questa interfaccia permette di scegliere quale schermata visualizzare e si fa carico di ricaricare periodicamente l'immagine visualizzata in modo da presentare dati sempre aggiornati. In figura 2.2 è riportata la schermata iniziale di Vistar.

Anche se nel complesso questo approccio soddisfa il primo requisito, ovvero visualizzare i dati dell'LHC in tempo reale, il limite maggiore di un sistema

[1]Si veda il sito web https://op-webtools.web.cern.ch/op-webtools/Vistar/vistars.php

così concepito è la sua architettura "monolitica". Essa infatti è costituita da un solo layer, in cui raccolta, analisi e presentazione dei dati avvengono nello stesso posto, ovvero nell'applicativo SCADA che sta alla base del sistema. Il web server è utilizzato semplicemente per rendere accessibile dal web parte dell'applicazione.

Com'è facile intuire, quindi, questo approccio non soddisfa molti dei requisiti sopra elencati. Innanzitutto questa soluzione non è scalabile orizzontalmente, ovvero non è semplice aggiungere web server al sistema in caso bisogno. Per fare ciò bisognerebbe istruire l'applicazione che genera le immagini di pubblicarle su diversi web server. Oltre a questo, tale sistema non permette di diffondere i dati in formato *machine readable*, limitandone l'uso alla mera visualizzazione predefinita dall'utente che ha configurato l'applicazione SCADA. Dato che diversi grafici e diagrammi sono visualizzati nella stessa immagine, non è inoltre possibile esportarli separatamente ed integrarli in altre pagine web. Oltre a ciò, essendo immagini statiche, è evidente che non siano interattive. Infine, abbiamo il problema della difficoltà di configurazione. Infatti, per modificare la presentazione dei dati, ovvero la disposizione dei grafici, i dati visualizzati e l'estetica di grafici e diagrammi, è necessario andare a configurare l'applicativo che genera le immagini, cosa non banale e per la quale sono autorizzati un numero limitato di utenti.

Un altro tool del tutto simile, ovvero basato su un applicativo SC (in questo caso LabVIEW) che invia le immagini delle sue schermate ad un web server, è quello raggiungibile all'indirizzo `http://lhcdashboard.web.cern.ch`. Questo è un primo tentativo di avere un sistema interattivo. Infatti, sebbene anche qui siano visualizzate solamente immagini statiche, è possibile cliccare su un grafico per zoomare su di esso e, tramite alcuni bottoni, è anche possibile modificare la finestra temporale dei dati visualizzati. Questo tipo di interattività è comunque raggiunta creando molteplici schermate nell'applicativo SCADA, in cui i dati sono presentati in maniera differente, e pubblicando gli screenshot sul web server.

È importante sottolineare che, come diremo anche in seguito, nella nostra

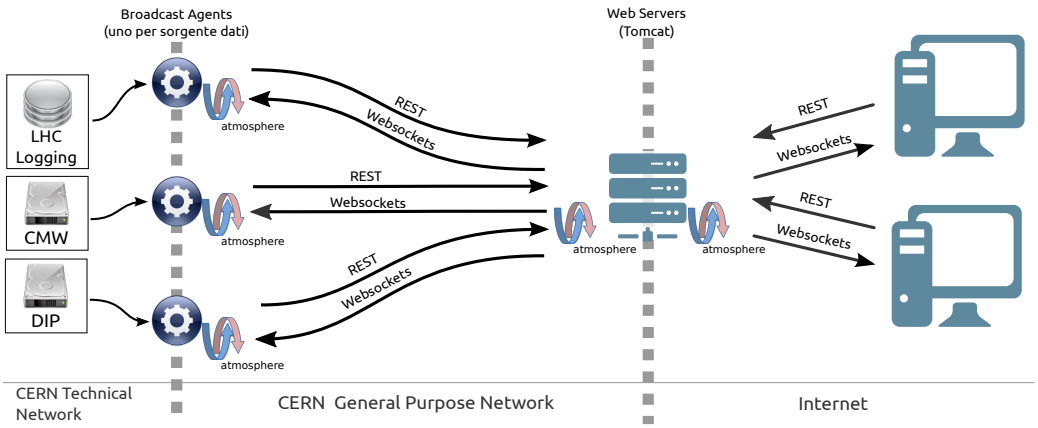


Figura 2.3: Schema della precedente infrastruttura

infrastruttura utilizziamo per convenienza alcune delle immagini e parte dei dati prodotti dall'applicativo SCADA che sta alla base di questo sistema. Facciamo così in via temporanea perché è necessario filtrare i dati provenienti da alcuni dispositivi in maniera particolare e, fintanto che non sarà aggiunto al nostro sistema un modo per filtrare opportunamente i dati in tempo reale, utilizziamo quelli già puliti dall'applicazione suddetta.

2.2.2 La prima "LHC Dashboard"

La seconda soluzione proposta, ovvero il prototipo da cui siamo partiti per sviluppare il nostro sistema, getta le basi per un'infrastruttura modulare e configurabile [1]. Questo sistema, infatti, presenta un'architettura a tre livelli: uno di acquisizione, uno di ed elaborazione ed uno di presentazione dei dati. In questo modo, com'è naturale, le diverse parti del sistema sono meno accoppiate e, nel caso una di queste non fosse più idonea (ad esempio è nata una tecnologia migliore per una determinata funzione), è possibile modificarla o sostituirla con poco sforzo.

L'architettura di questo sistema, schematizzata in figura 2.3, introduce il concetto di "agente". Questa entità altro non è che un connettore specializzato per raccogliere dati da una determinata sorgente, interfacciandosi con essa tramite protocolli specifici (ci riferiamo ai protocolli descritti nella

sezione 1.3.2). Questi agenti inviano i dati raccolti ad un web server via HTTP, il quale li inoltra ai client che ne hanno fatto richiesta. Dal momento che questi dati devono essere inviati ai client in tempo reale, si utilizza, se possibile, i WebSocket.

La modularità suddetta è raggiunta facendo in modo che agenti e web server, per interfacciarsi con il back-end di comunicazione, utilizzino interfacce Java che definiscono i metodi da invocare al fine di inviare richieste e dati. Queste interfacce sono poi realizzate da classi che implementano il meccanismo di comunicazione vero e proprio. Nel caso di questa infrastruttura queste classi utilizzano Atmosphere per gestire la comunicazione. Anche la comunicazione client-server è basata su Atmosphere. In questo caso, però, il lato client non fornisce nessun tipo di astrazione del metodo di comunicazione e, dato che i grafici utilizzano direttamente le API di Atmosphere, il lato client non ha la modularità vista per il back-end.

Per quanto riguarda la trasmissione dei dati, ed in particolare il loro formato, va detto che un difetto di questo sistema è che gli agenti non uniformano i dati ricevuti dai diversi dispositivi, lasciando l’onere ai client di farne il parsing ed omogeneizzarli per visualizzarli coerentemente. Ad esempio gli agenti RDA e DIP incapsulano i dati raccolti in oggetti JSON, mentre l’agente preposto ad interfacciarsi con l’LHC Logging estrae ed invia ai client i dati in formato CSV.

Anche questa soluzione non soddisfa diversi requisiti. Iniziando dal front end, vediamo come l’interfaccia grafica non è configurabile. Infatti, sebbene i dati siano presentati tramite grafici e diagrammi in maniera del tutto simile al sistema da noi sviluppato, l’interfaccia è interamente generata da uno script PHP preposto alla configurazione delle diverse pagine della dashboard, dei dati visualizzati e della loro presentazione estetica. Per modificare qualcosa è necessario riscrivere parti dello script stesso. Questo previene inoltre la possibilità di esportare i dati, sia presentati graficamente sia in formato *machine readable*. Le lacune principali però, risiedono nei back-end di comunicazione tra client e web server e tra quest’ultimi e gli agenti. In entrambi

i casi è utilizzato il framework Atmosphere, il quale permette di instaurare canali di comunicazione WEB bidirezionali utilizzando, quando possibile, i WebSocket. Questo framework è però utilizzato in maniera parziale, costituendo canali di comunicazione monodirezionali, e utilizzando una comune interfaccia REST nell'altro senso. Questo riduce drasticamente le potenzialità di Atmosphere. Più precisamente dal lato client non è possibile effettuare più di una richiesta di dati per ogni connessione, rendendo complicata la realizzazione di widget interattivi. Dal lato server, dato che i messaggi sono inviati al web server tramite HTTP, la frequenza e la quantità di dati inviata è limitata. Infine creare un'infrastruttura distribuita modulare e scalabile basando la comunicazione sul protocollo HTTP è decisamente complicato.

Partendo da queste analisi, abbiamo cercato di migliorare il sistema andando a riprogettare alcune componenti.

Capitolo 3

Progettazione del nuovo sistema

Passiamo ora alla progettazione dell'infrastruttura da noi realizzata. In questo capitolo vedremo innanzitutto com'è stato organizzato il progetto e quale processo di sviluppo software abbiamo adottato ed, in seguito, sarà presentata l'architettura del sistema e analizzata nelle sue componenti. Di volta in volta descriveremo i tool ed i software adottati ad ogni fase dello sviluppo. Nel prossimo capitolo ci concentreremo su come il sistema è stato implementato.

Iniziamo sottolineando che, data la necessità di avere un'infrastruttura *cross platform*, abbiamo deciso, in linea con le scelte effettuate per la vecchia dashboard, di utilizzare il linguaggio Java per realizzare tutte le componenti del back-end del nostro sistema. Da questa decisione deriva inoltre la possibilità di usare le librerie o software di seguito presentati, tutti implementati in Java o che, quanto meno, espongono API per il linguaggio Java.

3.1 Il processo di sviluppo

Vediamo quindi come è stato progettato lo sviluppo del software. Innanzitutto, dato che questo progetto si basa su un sistema preesistente, è ne-

cessario stabilire quali porzioni dell'infrastruttura precedente possono essere riutilizzate e quali, non essendo idonee, vanno re-implementate. In seguito è necessario scegliere le tecnologie da utilizzare per implementare le nuove componenti con cui sostituire le vecchie. Infine bisogna procedere alla progettazione dell'implementazione di ogni nuova porzione e alla sua integrazione col sistema esistente.

Per queste ragioni, abbiamo quindi deciso di adottare un processo di sviluppo software della classe *agile* ed in particolare di tipo *incrementale*. Nello specifico abbiamo utilizzato il framework di sviluppo software *scrum*. Grazie a questo framework è possibile definire cicli di progettazione, implementazione e deploy, detti *sprint*, ognuno dei quali porti al miglioramento di una componente dell'infrastruttura immediatamente integrabile con il sistema esistente. È evidente che, data la natura del progetto, per noi è molto conveniente alternare *sprint* di perfezionamento del front-end e del back-end, in modo da re-implementare le parti necessarie integrandole opportunamente con l'infrastruttura precedente. Risulta molto utile procedere nello sviluppo del progetto in maniera iterativa ed incrementale poiché, essendo questo un progetto innovativo, è difficile specificare anticipatamente, per ogni componente del sistema, tutti i requisiti specifici, i quali possono anche cambiare nel corso dello sviluppo. Gli *sprint* sono infatti cicli di breve durata che permettono, di volta in volta, di verificare con i clienti quanto è appena stato implementato e definire qual è la prossima priorità.

Infine, un processo di sviluppo di tipo *agile* risulta adatto in una situazione in cui, date le dimensioni ristrette del team di sviluppo, la coordinazione tra i membri del team è molto semplice e non necessita di particolari accorgimenti. Un processo di sviluppo più strutturato, probabilmente, introdurrebbe complicazioni superflue.

Descriviamo ora brevemente due tool fondamentali utilizzati nel processo di sviluppo del sistema, ovvero JIRA per quanto riguarda la gestione del processo di sviluppo e Jenkins, una piattaforma utilizzata per gestire il testing ed il deploy del sistema.

3.1.1 JIRA - il processo software

JIRA è un *project management software* largamente utilizzato al CERN. Questo software gestionale, grazie ad una semplice interfaccia web, permette di definire, organizzare e tenere traccia dello stato di avanzamento di *ticket*, ovvero descrizioni di problemi o richieste di nuove funzionalità nell'ambito di un processo industriale. JIRA permette inoltre di organizzare questi ticket all'interno di un processo di sviluppo software *scrum*, permettendo di definire *sprint* che contengono diversi *ticket*. Come previsto da *scrum*, ad ogni ticket è possibile assegnare una stima della difficoltà del problema da risolvere o della nuova funzionalità da implementare. Questo porta ad avere una stima ragionevole del tempo che si impiegherà a completare uno *sprint* e di conseguenza permette di gestire efficientemente il processo di sviluppo software[5, 6].

Oltre a ciò JIRA è una piattaforma open source ed estendibile. Grazie alle API REST e Java, è infatti possibile arricchire il software con diversi plugin[7]. Questo sistema di plugin può essere utilizzato dagli sviluppatori per aggiungere svariate funzionalità, ad esempio al CERN è stato sviluppato un mail handler per gestire i ticket via e-mail[1].

Infine, è molto conveniente utilizzare JIRA come software per la gestione del processo di sviluppo, perché è ben integrato con Confluence, una versatile piattaforma di collaborazione. Questo software, sviluppato dalla stessa azienda che ha creato JIRA, è molto utilizzato al CERN per creare e gestire i wiki dei progetti supportati internamente.

Per un maggior grado di approfondimento, rimandiamo i lettori interessati ad alcune letture[8, 9].

3.1.2 Jenkins - l'integrazione continua

Jenkins è un tool per *l'integrazione continua*, ovvero aiuta a gestire in maniera semi-automatica il ciclo di compilazione, testing e rilascio di un software, per-

[1]Home page del progetto: https://wikis.web.cern.ch/wikis/display/JMH/Home

mettendo di rilasciare nuove versioni frequentemente, assicurando la corretta integrazione di ogni componente del software.

Questo tool è estremamente utile nel gestire lo sviluppo di un'infrastruttura come quella qui descritta. Innanzitutto, dato che, come vedremo nella prossima sezione, il sistema è composto da diversi moduli interconnessi, è necessario accertarsi che, ad ogni fase dello sviluppo di una componente, essa sia compatibile ed integrabile con il resto dell'infrastruttura. Questo deve essere fatto in due fasi. Innanzitutto in fase di compilazione del modulo, verificando che il modulo stesso o le sue dipendenze non vadano in conflitto con il resto del sistema. In seguito, prima della fase di rilascio del nuovo sistema, è fondamentale la fase di testing, eseguendo degli *unit test* ed *integration test* che, se ben concepiti, assicurano che il nuovo componente sia integrato correttamente col sistema.

Jenkins, grazie al supporto nativo dei progetti Maven, permette di configurare in maniera semplice la risoluzione e gestione automatica molto elaborata delle dipendenze del progetto. Oltre a ciò, Jenkins permette di eseguire automaticamente i test e di ottenere un report dettagliato del risultato. Questo è di fondamentale aiuto nell'individuazione degli errori a *run time*.

Comunque, anche qui rimandiamo i lettori interessati ad un approfondimento su Jenkins e Maven ad alcune letture[10, 11].

3.2 L'architettura

Dopo aver visto come si è scelto di procedere nella progettazione e sviluppo del sistema, da un punto di vista organizzativo, descriviamo ora l'architettura sottostante e l'interazione, ad alto livello, tra le varie componenti del sistema.

Innanzitutto, data la necessità di mantenere attivo il sistema, si è deciso di mantenere l'architettura sostanzialmente invariata rispetto all'infrastruttura da aggiornare, re-implementando di volta in volta diverse componenti. Come si può vedere in figura 3.1, abbiamo mantenuto il concetto di *agente* come

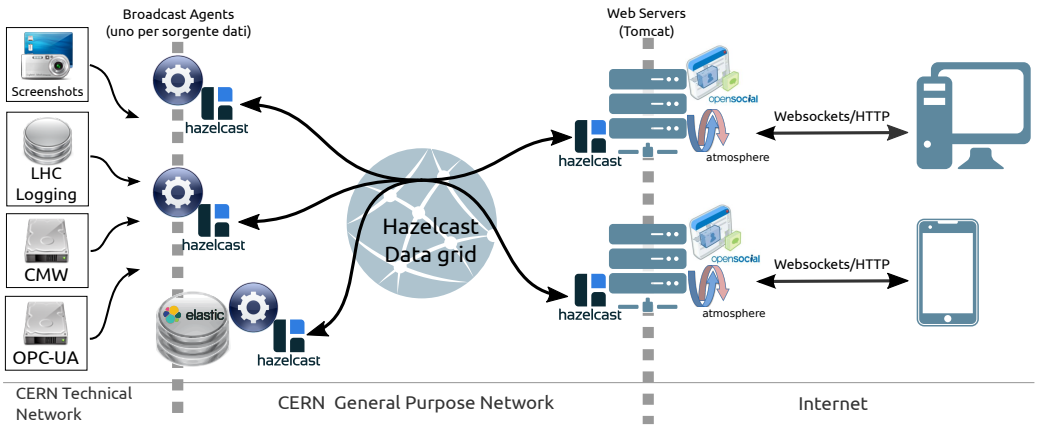


Figura 3.1: Schema dell'architettura del nuovo sistema

connettore specializzato per una sorgente dati e come ponte tra le due sottoreti del CERN che ci concernono. Abbiamo poi i web server, utilizzati per gestire le richieste dei client ed inoltrare loro i dati ricevuti dagli agenti. Infine vi sono i client web, ovvero i consumatori dei dati prodotti dall'LHC. A livello architetturale è stato aggiunto, come definito nei requisiti, un database in cui immagazzinare i dati raccolti e inviarli, opportunamente filtrati ed aggregati, ai client che ne fanno richiesta.

Un'altra fase fondamentale nello sviluppo del sistema è la progettazione delle modalità con cui le varie componenti, vecchie e nuove, devono interagire tra loro. Come è riportato nel diagramma in figura 3.2, l'azione, come di consueto, è iniziata dal client. Esso, una volta connesso a uno dei web server, richiede la configurazione della dashboard; a questo punto carica i widget contenuti nella pagina che sta visualizzando, infine ogni widget apre una connessione con il web server e si sottoscrive ad uno stream dati. Ora il ruolo attivo passa al back-end. Il web server invia la richiesta di sottoscrizione all'agente appropriato. Quest'ultimo inizia a collezionare i dati richiesti dalla sorgente dati. A questo punto, ogni volta che sono disponibili nuovi dati, vengono re-direzionati verso il (o i) client che ne hanno fatto richiesta.

Per avere una più chiara visione delle scelte progettuali effettuate nello sviluppo del sistema, vediamo più dettagliatamente le problematiche da af-

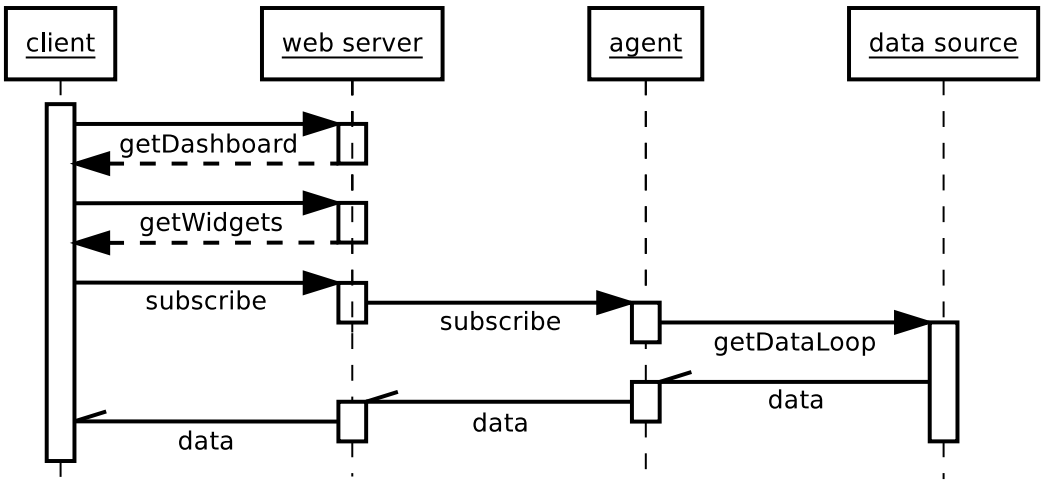


Figura 3.2: Diagramma delle interazioni

frontare, citando le tecnologie adottate per ovviare a questi problemi. Nella sezione successiva descriveremo meglio ciascun tool impiegato.

3.2.1 Front-end

Nel caso del front-end, le difficoltà da superare sono principalmente relative alla configurazione della dashboard, alla trasmissione e alla presentazione dei dati.

Per quanto riguarda il primo problema, abbiamo sfruttato la possibilità di mappare una cartella del DFS (il file system distribuito del CERN) su una URL ed eseguirvi script PHP. In questo modo è possibile creare uno script che scansioni la gerarchia del file system a partire da una determinata cartella, legga i file di configurazione presenti in essa e che, in base a questi, crei un oggetto JSON che rappresenta la configurazione della dashboard, in termini di pagine, sotto-pagine, widget e parametri dei widget stessi. Più precisamente, quindi, una volta che il client si connette al web server, richiede allo script PHP la configurazione della dashboard. A questo punto crea l'interfaccia e carica i widget da visualizzare, i quali si connettono al server e richiedono i dati in base alla loro configurazione.

Inoltre, dato che vogliamo realizzare un'interfaccia multipiattaforma, è necessario avere un modo per configurare in maniera semplice la disposizione dei vari widget nella pagina che al tempo stesso si adatti alle dimensioni della finestra contenente la dashboard, in modo da essere visualizzata in maniera ottimale sia sul grande schermo di un PC, sia su quello di uno smartphone. Questo problema può essere risolto grazie a *Bootstrap*, una libreria che permette di realizzare, senza sforzi eccessivi, il *responsive web design*.

Il secondo problema, relativo alla presentazione dei dati, sottende due sfide. Innanzitutto è necessario definire cosa siano questi widget e come sia possibile configurarli. Per affrontare il problema abbiamo deciso di utilizzare le API *OpenSocial*, le quali permettono di creare pagine web, incapsulate in iframe, a cui è possibile passare parametri in maniera standard. Nel nostro caso tramite questi parametri è possibile specificare quali dati devono essere visualizzati e come. Il punto di forza di questi widget è l'essere componenti riutilizzabili ed integrabili in altre pagine web come qualsiasi iframe. La seconda problematica relativa alla presentazione dei dati è data dalla loro alta risoluzione. Una serie temporale, in media, ha un valore ogni dieci secondi, quindi, nel caso un utente volesse visualizzare i dati dell'ultimo mese, nel grafico dovrebbero essere visualizzati quasi 300.000 punti per ogni serie. Questa risoluzione, oltre che inutile, rende la visualizzazione impossibile. Per contro, nel caso l'utente voglia visualizzare i dati relativi ad un intervallo di tempo di qualche minuto, la risoluzione nell'ordine dei secondi è appropriata. Abbiamo deciso di spostare questo problema al lato server. Il client, in questo modo, richiede (e riceve) i dati già opportunamente filtrati ed aggregati.

3.2.2 Back-end

Iniziamo dall'ultima problematica presentata per il client, ovvero il problema dell'analisi dei dati. Il back-end deve avere un modo di immagazzinare i dati raccolti dalle varie sorgenti ed analizzarli in tempo reale su richiesta dei client. Per soddisfare questa necessità ci siamo orientati su *Elasticsearch*,

un database distribuito con elementari funzionalità per l'analisi dei dati. In concomitanza abbiamo anche sviluppato un agente che si occupi di estrarre, dalla pubblicazione a cui i client si sottoscrivono per ottenere i dati, i parametri definiti dal client e mapparli nel linguaggio di query di Elasticsearch. I parametri definiscono il dispositivo di cui si vogliono i dati, la finestra temporale e il tipo di aggregazione da effettuare.

Per rimanere in tema di interazione client-server, è necessario riprogettare parzialmente la metodologia di comunicazione tra web server e client. Questa necessità è data dal fatto che ora, avendo aggiunto il supporto per widget interattivi, l'utente può richiedere stream di dati diversi tramite lo stesso widget. Precedentemente, ogni widget (o diagramma/grafico) corrispondeva ad uno stream dati diverso. Per far fronte a questo problema abbiamo fatto sì che, sia i dati verso i client, sia le richieste dei client, fossero gestiti dal framework *Atmosphere*. Grazie a questa piccola modifica è possibile tenere traccia nel web server, dei client connessi e delle loro richieste. In questo modo è possibile permettere al client di sottoscriversi o annullare la sottoscrizione a diversi stream dati utilizzando la stessa connessione.

La sfida più importante che presenta il back-end, però, è data dal problema di raccogliere dati da diverse risorse eterogenee, filtrarli, renderli il più omogenei possibile e distribuirli ad un grande numero di utenti. Inizialmente si era pensato di riprogettare completamente il back-end, sostituendo gli agenti con un *ESB* (Enterprise Service Bus) distribuito. Questo tipo di tool, permette, grazie alla combinazione di diversi moduli, di integrare in maniera omogenea connettori per svariati protocolli, nascondendo la loro eterogeneità dietro ad un unico linguaggio di query. In questo modo i diversi agenti diventerebbero moduli di un unico ESB. Una volta estratti i dati dai diversi connettori è necessario filtrarli *on-line* opportunamente, archiviarli nel database Elasticsearch e, nel caso gli utenti li richiedano, inoltrarli anche ai web server. Il filtraggio dei dati in tempo reale è necessario perché i valori ricevuti dai dispositivi non sono sempre accurati, talvolta presentano delle aberrazioni dovute a rilevazioni inesatte per motivi fisici. In questo progetto

di architettura, i web server interrogano direttamente l'ESB, il quale può interrogare Elasticsearch per avere dati che non ha nella cache ed instradare opportunamente verso i web server i dati filtrati provenienti dai diversi dispositivi per avere aggiornamenti in tempo reale.

Abbiamo però valutato che un cambiamento del genere avrebbe potuto dilatare eccessivamente i tempi di sviluppo del sistema. Questo perché avremmo dovuto confrontarci con tecnologie completamente nuove, ovvero un ESB come Apache ServiceMix ed un sistema di analisi dati *on-line* come ad esempio Apache Storm; per non parlare del fatto che avremmo potuto riscontrare difficoltà nell'ottenere una sufficiente capacità computazionale per implementare tutto questo. Abbiamo quindi optato di non modificare radicalmente la vecchia architettura. Ciò che è stato modificato è il back-end di comunicazione tra agenti e web server. Al fine di avere un'infrastruttura modulare e scalabile, abbiamo deciso di adottare *Hazelcast*, un database distribuito in memoria che implementa il pattern di comunicazione *publish/subscribe*, ottimo tool che permette di creare un *cluster* all'interno del quale è possibile distribuire e fare *caching* dei dati in maniera efficiente. Grazie ad Hazelcast è inoltre possibile realizzare alcuni meccanismi utili per ottimizzare l'uso delle risorse e per gestire il crash di alcuni nodi del cluster. Come vedremo meglio nel capitolo relativo all'implementazione del sistema, grazie alle funzionalità offerte da Hazelcast, possiamo far sì che, in caso di crash di un agente, tutte le pubblicazioni da esso servite siano trasferite ad un altro agente capace di gestirle, o fare in modo che l'agente, una volta ripristinato, se ne faccia nuovamente carico.

Come soluzione temporanea al problema dei valori da filtrare, abbiamo optato per collezionare i dati da una delle applicazioni SCADA che generano le immagini per le vecchie dashboard di cui abbiamo parlato nel capitolo precedente, la quale, ovviamente, esegue già il filtraggio dei dati.

L'idea che sta alla base del sistema, quindi, è che i web server inoltrano agli agenti appropriati le richieste ricevute dai clienti. A questo punto i diversi agenti, quando viene loro richiesto, si fanno carico di estrarre i dati

dalle diverse fonti, li incapsulano in un formato dati omogeneo e li inviano ai web server tramite Hazelcast. A questo punto i web server inoltrano i dati ai client che ne hanno fatto richiesta tramite Atmosphere. Si noti come il formato delle richieste, ovvero delle sottoscrizioni, sia noto solo ai client (ai widget) e agli agenti. I web server, utilizzando Atmosphere ed Hazelcast, sono preposti ad instradare opportunamente le richieste ed i dati, trattando le pubblicazioni come comuni stringhe con cui etichettare i diversi canali di comunicazione.

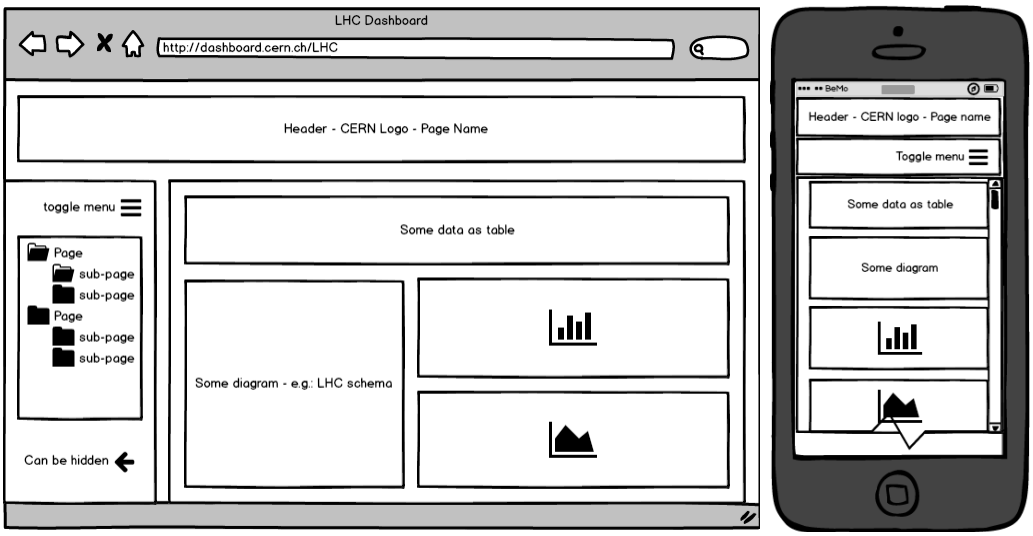
Un'ultima componente molto importante da progettare è il monitoring. Un metodo molto semplice per realizzarlo è utilizzare il framework Spring per esporre, tramite JMX (Java Management Extensions), metriche utili per avere un'idea della salute di un applicativo Java, nel nostro caso web server ed agenti. Grazie al software Jolokia è possibile accedere a queste metriche tramite un'interfaccia REST, in modo da permettere il monitoring via HTTP standard [12]. Infine, tramite Jenkins, è poi possibile raccogliere periodicamente queste metriche, confrontarle con parametri di riferimento ed essere notificati, ad esempio via mail, in caso di problemi.

3.3 Le componenti

Dopo aver descritto l'infrastruttura ad alto livello, al fine di comprendere meglio la funzione delle varie componenti e l'interazione tra esse, vediamo più nello specifico i tool sopra citati. I primi due sono utilizzati dal lato client, gli ultimi due dal lato server, ed Atmosphere da entrambi i lati.

3.3.1 Bootstrap

Vediamo innanzitutto in cosa consiste questa libreria e come può essere utilizzata per creare la nostra dashboard rispettando il responsive design. Bootstrap, inizialmente sviluppato come framework per lo sviluppo di interfacce



(a) Versione desktop (b) Versione mobile

Figura 3.3: Wireframe dell'interfaccia web (generata con balsamiq)

interno a Twitter, mette a disposizione una collezione di tool, basati su tecnologie standard come *CSS3* o *HTML5*, pensati per realizzare velocemente una potente interfaccia web *cross-platform*, fruibile sia da classici PC sia da dispositivi mobili.

Questo framework è particolarmente adatto per creare un'interfaccia come quella desiderata, schematizzata in figura 3.3. La nostra interfaccia, infatti, deve essere composta da un header, un menu laterale e un contenitore centrale in cui sono disposti diversi widget tramite cui visualizzare i dati. Grazie a Bootstrap è possibile definire, per il contenitore centrale, un layout a griglia, in cui i diversi widget si riorganizzino automaticamente; ciò che su un grande schermo viene visualizzato come in figura 3.3a, su un dispositivo mobile sarà visualizzato come in figura 3.3b.

Questo può essere ottenuto aggiungendo agli elementi HTML le classi Bootstrap proposte a questo scopo. Il framework, infatti, suddivide la larghezza di un elemento HTML in 12 segmenti. Applicando ad esempio la classe `col-md-6` ad un `<div>`, esso sarà largo $\frac{6}{12}$, ovvero la metà dell'elemento contenitore negli schermi grandi, ma occuperà l'intera larghezza sugli

schermi piccoli. Per una più completa descrizione del framework si può far riferimento alla pagina web del progetto `http://getbootstrap.com/`.

3.3.2 OpenSocial

OpenSocial è un framework open source che permette di creare applicazioni web, dette gadget, configurabili ed integrabili in un qualsiasi contenitore OpenSocial, incoraggiando lo sviluppo di componenti riutilizzabili. OpenSocial definisce un ambiente contenitore ed una serie di API per creare applicazioni web configurabili dal contenitore. Più precisamente, le API permettono di creare un descrittore dell'applicazione web (un file .xml) che dà al contenitore informazioni su quali parametri è possibile passare all'applicazione o come visualizzarla. Il contenitore, invece è preposto a renderizzare l'applicazione web, configurandola ed incapsulandola in un iframe. Grazie al contenitore OpenSocial, i gadget sono integrabili anche in siti web "non OpenSocial" come tradizionali iframe.

Detto ciò, è evidente che questo meccanismo ci fornisce le funzionalità di cui abbiamo bisogno per avere un'interfaccia web modulare e configurabile. Infatti, è sufficiente implementare un solo gadget per una determinata funzione e, configurando opportunamente la dashboard come descritto in sezione 3.2.1, è possibile creare più istanze dello stesso gadget, configurate per visualizzare dati diversi o in modi diversi. Pensiamo ad esempio all'evoluzione nel tempo della temperatura dei magneti. È possibile creare un solo gadget contenente il codice necessario per visualizzare serie temporali ed istanziarlo diverse volte, configurando diversamente lo stream di dati da plottare o l'estetica del grafico grazie alle API OpenSocial.

Questo, associato alla possibilità di esportare i widget in una pagina web classica (non OpenSocial), permette anche di creare widget che fungano da gateway tra la rete del CERN ed internet. Questo dà la possibilità di esportare i dati sia in formato grafico sia in formato *machine readable*, aiutando la diffusione degli stessi in ambienti scientifici o meno ed aumentando le possibilità di analisi sui dati [13].

3.3.3 Atmosphere

Vediamo ora l'attore principale nella comunicazione client-server. Atmosphere è un framework open source pensato per implementare un meccanismo di comunicazione web secondo il pattern *publish/subscribe*. Grazie all'uso congiunto di un modulo server ed uno client, è infatti possibile creare un'infrastruttura in cui il server può diffondere in multicast una cospicua quantità di dati a molteplici client.

Per quanto riguarda il nostro progetto, abbiamo aggiunto, rispetto al sistema precedente, la possibilità di avere sottoscrizioni dinamiche, necessarie per supportare l'interattività di cui abbiamo parlato. Nello specifico, abbiamo riprogettato i moduli client e server relativi ad Atmosphere (già utilizzato nell'infrastruttura da cui siamo partiti), in modo da poter cambiare stream di dati a cui si è sottoscritti, mantenendo la stessa connessione, ovvero senza dover chiudere e riaprire un nuovo WebSocket.

Una funzionalità che rende Atmosphere un framework molto versatile è il fatto che provvede alla gestione del protocollo di comunicazione sottostante. Atmosphere, infatti, utilizza in maniera trasparente diversi protocolli per veicolare i dati. Tra gli altri, può utilizzare HTTP classico o i WebSocket in base alle capacità del server e del client. In questo modo è possibile creare applicazioni che richiedono la trasmissione di dati real-time, capaci di supportare client datati usando HTTP puro e, con i client più recenti, di sfruttare la potenza dei WebSocket, necessari per implementare al meglio un'applicazione di questo tipo [14, 15].

3.3.4 Hazelcast

Hazelcast è un *in-memory data grid* open source. Esso permette di creare un cluster di macchine, in cui mantenere un database in memoria centrale, partizionato e replicato sui nodi del cluster, in modo da essere resistente in caso di malfunzionamento di uno o più nodi. Hazelcast implementa anche il

modello di comunicazione *publish/subscribe*, grazie al quale è possibile inviare dati da un nodo agli altri.

Per queste caratteristiche abbiamo scelto di adottare Hazelcast come back-end di comunicazione tra agenti e web server. Dal punto di vista della nostra infrastruttura, l'insieme degli agenti e dei web server è visto come un cluster Hazelcast. Il punto di forza chiave è la capacità del cluster di riorganizzarsi automaticamente in caso di entrata o uscita di un nodo, sia in termini di distribuzione del database, sia per quanto riguarda i canali di comunicazione basati sul metodo *publish/subscribe*. Questo ci permette, da un lato di creare un cluster resistente al crash di alcuni nodi, dall'altro ci permette, nel caso fosse necessario, di scalare orizzontalmente l'infrastruttura in maniera trasparente, ad esempio aggiungendo uno o più web server in caso di un numero elevato di richieste da parte dei client.

Inoltre, grazie alle funzionalità di comunicazione basate sul metodo *publish/subscribe*, le richieste di nuovi dati fluiscono dai web server agli agenti, e i dati stessi vengono trasmessi in direzione opposta. Grazie alla possibilità di avere un database distribuito, è inoltre possibile fare il caching dei dati al fine di recuperarli più velocemente in caso di richieste multiple delle stesse informazioni.

3.3.5 Elasticsearch

Vediamo infine l'ultimo tool aggiunto al sistema, ovvero il database in cui archiviare i dati prodotti dall'LHC ed estrarli su richiesta dei client. Elasticsearch è un database NoSQL distribuito che mette a disposizione degli utenti avanzate funzionalità di ricerca *full-text* e semplici query per l'analisi dei dati numerici, come filtri e calcolo di metriche su dati aggregati. Grazie al suo design distribuito, al partizionamento dei dati e all'esecuzione distribuita di interrogazioni, Elasticsearch ha un'ottima capacità di scalare orizzontalmente, sia in termini di disponibilità dei dati (sono partizionati e replicati nel cluster) sia in termini di velocità nell'esecuzione delle query [16].

Abbiamo quindi deciso di adottare questo database perché, nel caso i dati richiesti dal client non siano già presenti nella cache di Hazelcast, ci permette di estrarli filtrandoli ed aggregandoli in base alle richieste dei client in pochi millisecondi. Un utente, ad esempio, potrebbe richiedere i dati appartenenti ad una determinata finestra temporale e con una risoluzione minore di quella di acquisizione. In questi casi possiamo istruire Elasticsearch ad estrarre i dati aggregandoli per un dato intervallo e per ognuno calcolare valore minimo, medio, massimo, varianza, etc.

Oltre a questo, abbiamo adottato Elasticsearch come database interno al sistema perché è già utilizzato internamente al CERN e vi è un progetto per attivare un servizio di storage basato su Elasticsearch. Questo ci darebbe la possibilità di usarlo senza doverci curare delle problematiche dovute al deploy e manutenzione di un database, attività di cui si occuperebbe la sezione del CERN preposta alla gestione del servizio.

Capitolo 4

Implementazione del nuovo sistema

Proseguendo nella trattazione dello sviluppo della nuova "LHC Dashboard", passiamo ora alla fase di implementazione. In questo capitolo vedremo nello specifico come le problematiche descritte nel capitolo precedente sono state affrontate utilizzando i tool citati e come ogni nuova componente del sistema è stata realizzata ed integrata con il resto dell'infrastruttura.

Inizieremo dal front-end, ovvero dall'interfaccia utente, per proseguire con la descrizione delle comunicazioni client-web server e web server-agenti. Vedremo poi come abbiamo utilizzato Elasticsearch per archiviare e analizzare i dati. Infine illustreremo le tecniche utilizzate per il monitoring del sistema.

4.1 Interfaccia utente

Vediamo quindi com'è stata realizzata l'interfaccia utente utilizzando le tecniche ed adottando i tool descritti nel capitolo precedente. Per avere una visione più chiara di quanto diremo, corrediamo la descrizione con un esempio. Nelle prossime sezioni prenderemo come riferimento l'interfaccia in figura 4.1.

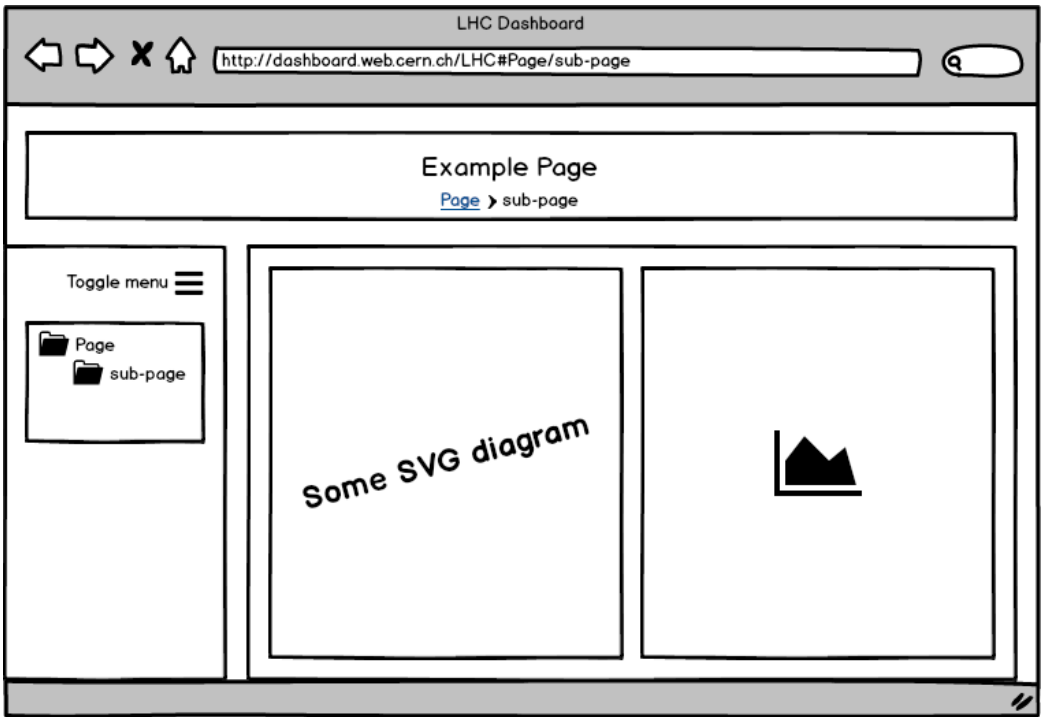


Figura 4.1: Interfaccia di esempio da realizzare (generata con balsamiq)

4.1.1 Configurazione

Il contenuto della dashboard è configurato in base alla gerarchia del DFS e ad alcuni file di configurazione che definiscono il layout di ogni pagina, la disposizione dei widget in essa e i parametri da passare ai widget stessi.

Innanzitutto va specificato che l'indirizzo web corrispondente alla cartella del DFS che è "radice" della gerarchia della configurazione è *hard coded* all'interno della servlet a cui il client si connette.

Vediamo esattamente quali sono le azioni che avvengono al fine di configurare la dashboard. Quando il client si connette, riceve lo scheletro della dashboard, a questo punto uno script JavaScript richiede la configurazione della dashboard ad uno script PHP. Quest'ultimo, scansionando la gerarchia del DFS, partendo da una data cartella e leggendo i file di configurazione in essa contenuti, genera un oggetto JSON, il quale rappresenta la configurazione della dashboard. Lo script JavaScript, una volta decodificato que-

st'oggetto, imposta appropriatamente il menu laterale e, quando richiesto, renderizza i widget, configurandoli grazie alle API OpenSocial e disponendoli opportunamente grazie a Bootstrap.

Per comprendere meglio quanto suddetto vediamo un esempio del processo che porta alla configurazione della dashboard. Pensiamo ad esempio di voler ottenere una dashboard come quella in figura 4.1. Innanzitutto definiamo l'ipotetica cartella radice nel DFS:

```
/websites/lhcdashboard
```

ora, avendo la dashboard corrente il path `/LHC`, la servlet inietta nello script JavaScript da inviare al client la URL che corrisponde alla radice della dashboard, ovvero un indirizzo del tipo:

```
http://lhcdashboard.web.cern.ch/LHC
```

Va detto che l'host name (`lhcdashboard.web.cern.ch`) è preimpostato nel file di configurazione della servlet (`web.xml`). Dato che vogliamo avere una pagina "Page" ed una sotto-pagina "sub-page", la cartella suddetta deve contenere la seguente gerarchia:

```
/websites/lhcdashboard/LHC/content/Page/sub-page
```

Passiamo ora ai file di configurazione necessari. Il primo, contenuto nella cartella `content/`, è il catalogo dei widget OpenSocial disponibili, ovvero contiene i riferimenti ai descrittori dei gadget (file .xml). Si veda il sorgente 1 come esempio. In questo caso sono disponibili due gadget, uno con ID "gadget-svg" e l'altro "gadget-chart". Gli altri file di configurazione sono utilizzati per definire il layout ed il contenuto di ciascuna pagina. Questi file si trovano nelle cartelle relative alle pagine e definiscono quali gadget visualizzare in ogni pagina, i parametri da passare a ciascun gadget e la disposizione dei gadget stessi in termini di classi Bootstrap. Seguendo il nostro esempio, la cartella "sub-page" contiene il file "screen1.page", il contenuto del file è riportato nel sorgete 2. Questo JSON definisce il titolo della pagina,

i due gadget da visualizzare, la loro disposizione ed i parametri da passare ai gadget stessi.

Per motivi di sicurezza la gerarchia del file system e i file di configurazione non sono accessibili direttamente dal web. Per questo il client, come abbiamo detto, richiede ad uno script PHP, contenuto nella cartella `LHC/php`, di generare la configurazione della dashboard. Questo script è accessibile dal web tramite un indirizzo del tipo:

`http://lhcdashboard.web.cern.ch/LHC/php/config.php`

4.1.2 Navigazione

Passiamo ora al problema della navigazione tra le pagine della dashboard. L'obiettivo finale è di avere un modo per mantenere la logica della navigazione a lato client e allo stesso tempo poter referenziare tramite un URL univoco ogni singola pagina. Per fare ciò, la navigazione è implementata utilizzando il *fragment identifier*, ovvero la stringa che sta alla destra del cancelletto (o *hash*, `#`) in un URL. Nella pratica, ad ogni pagina è associato un *fragment identifier*, generato in base al suo posto nella gerarchia delle pagine e in base ai nomi delle pagine stesse. Sempre seguendo l'esempio precedente, la pagina "sub-page" è referenziata tramite lo URL:

`http://lhcdashboard.web.cern.ch/LHC#Page/sub-page`

Grazie a questa URL, uno script client recupera, dal JSON di configurazione di cui abbiamo parlato prima, le caratteristiche della pagina "sub-page". A questo punto carica, dispone e configura opportunamente i widget richiesti e imposta opportunamente la pagina, evidenziando nel menu laterale la pagina visualizzata ed impostando il titolo della pagina stessa nell'header.

A questo punto è chiaro come avviene il processo di navigazione. In fase di generazione del menu laterale, uno script client, dopo aver richiesto allo script PHP suddetto la configurazione della dashboard, imposta nell'attributo HTML `href` di ogni elemento del menu il *fragment identifier* opportuno.

Per tornare all'esempio precedente, l'elemento del menu relativo a "sub-page" è nella forma:

```
<a href="Page/sub-page">sub-page</a>
```

Quando l'utente clicca sull'elemento, il browser intercetta l'evento della modifica dell'hash e innesca il processo suddetto per la configurazione della pagina.

4.1.3 Visualizzazione

La visualizzazione dei dati, come abbiamo già detto in precedenza, avviene tramite grafici e diagrammi renderizzati come gadget OpenSocial. Grazie alle API di questo framework abbiamo creato quattro tipi di widget configurabili, tre per visualizzare i dati ed uno per esportarli in formato *machine readable*. Più precisamente un widget renderizza immagini PNG ricevute in formato *base64*, uno renderizza i dati sotto forma di diagrammi ed immagini dinamiche in formato SVG e l'ultimo visualizza i dati sotto forma di grafici.

Per quanto riguarda la configurazione dei gadget, com'è possibile vedere dal file di configurazione di esempio nel sorgente 2, è costituita da diversi file, o per meglio dire si tratta di referenze a diversi file che il contenitore OpenSocial traduce in URL. Questi file contengono altri file di configurazione specifici per ciascun widget. In particolare è necessario impostare il parametro "sbsList", ovvero la lista di sottoscrizioni da attivare. Per i diagrammi SVG, è obbligatorio impostare anche il parametro "svgUrl", ovvero la referenza al file SVG da renderizzare. Il parametro "styleUrl" definisce l'estetica interna al widget; per le immagini questo è un file contenente codice *CSS*, per i grafici è un file di configurazione specifico per la libreria grafica; nel nostro caso, dato che utilizziamo Highcharts, il file contiene un oggetto JSON nel formato richiesto dalla libreria. Un ultimo parametro molto importante è "jscbUrl", tramite il quale è possibile ridefinire le funzioni richiamate in fase di inizializzazione del widget, di ricezione di nuovi dati o in altre fasi specifiche di ogni

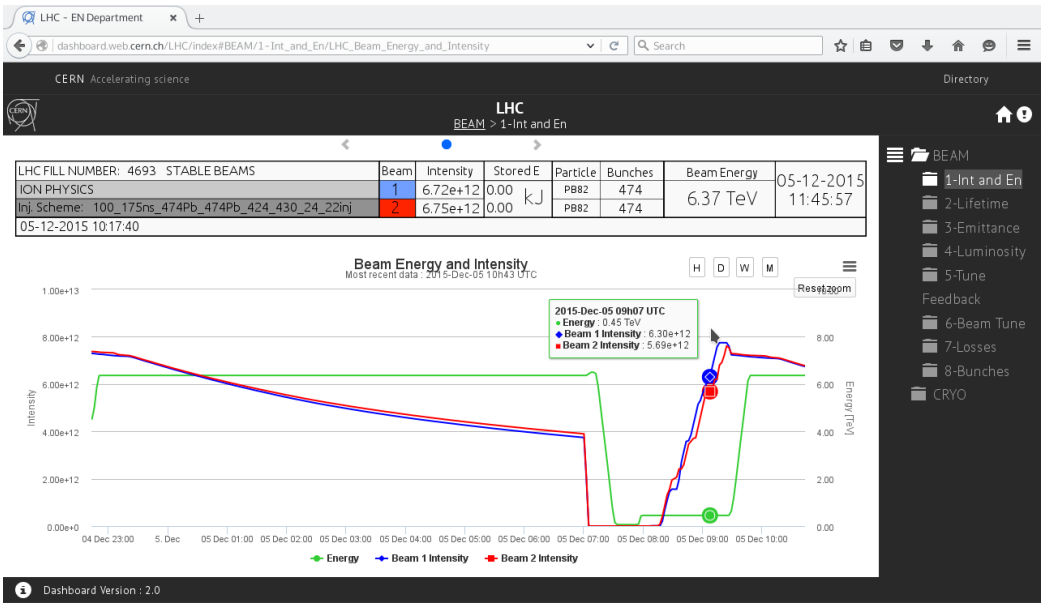


Figura 4.2: Schermata della dashboard

widget. Con questo parametro, quindi, è possibile modificare comodamente il comportamento di default del widget. Questo è spesso necessario perché, anche se gli agenti rendono strutturalmente omogenei i dati provenienti dai diversi dispositivi dell'LHC, per poterli visualizzare opportunamente bisogna sapere cosa rappresentano semanticamente questi dati.

L'interattività di cui abbiamo parlato in precedenza si traduce essenzialmente nella possibilità di cambiare i dati visualizzati in un grafico come quello rappresentato in figura 4.2, cambiando finestra temporale e risoluzione dei dati. Per fare ciò, come vedremo meglio nelle prossime sezioni, i widget, ad ogni interazione, richiedono dati diversi al web server, cambiando i parametri delle pubblicazioni a cui sono sottoscritti.

Sempre in figura 4.2 è rappresentata una videata della dashboard che mostra due gadget, di cui il primo è un'immagine SVG in cui sono visualizzati in tempo reale i dati dei due fasci di adroni. Nel secondo gadget, invece, sono visualizzati tramite un grafico Highcharts i dati relativi all'energia ed intensità dei fasci nelle ultime sei ore. Grazie ai suddetti file di configurazione è possibile controllare l'aspetto dei widget stessi o i dati visualizzati.

Grazie alle funzionalità di interattività, invece, è possibile modificare la finestra temporale da monitorare, ad esempio facendo sì che venga mostrato l'andamento dei valori nell'ultimo mese.

4.2 Comunicazione client-server

Continuiamo la descrizione dell'implementazione del sistema descrivendo come è realizzata la comunicazione tra client e web server. Come abbiamo già detto abbiamo utilizzato il framework Atmosphere. Esso presenta un modulo client, costituito da una libreria JavaScript, ed un modulo server, ovvero una libreria Java, tramite i quali abbiamo realizzato un semplice protocollo applicativo per permettere ai client di richiedere e ricevere i dati dal web server. Per quanto riguarda il client abbiamo realizzato una piccola "classe" JavaScript che rappresenta una connessione Atmosphere, utilizzata dai diversi widget per connettersi al sistema, richiedere i dati ed essere notificati quando sono disponibili aggiornamenti, senza doversi curare delle API Atmosphere. Dal lato server, abbiamo realizzato un meccanismo per decodificare le richieste dei client, tenerne traccia, inoltrarle agli agenti e, una volta ricevuti dati dagli agenti, inoltrarli ai client che ne hanno fatto richiesta.

Vediamo più nello specifico il lato client. Il sorgente 3 mostra uno stralcio della classe JavaScript suddetta. Com'è possibile vedere, una volta istanziata la classe *BroadcastConnection*, i gadget possono invocare alcuni metodi per comunicare col server. Innanzitutto possono aprire una connessione con il server e registrare callback da eseguire quando arrivano nuovi dati, possono inoltre richiedere uno o più stream dati, specificando la lista di sottoscrizioni da attivare. Si noti che queste sottoscrizioni devono avere un formato riconosciuto da uno degli agenti disponibili. In altre parole, al fine di inoltrare più facilmente le richieste dei client all'agente opportuno e fornire agli utenti esperti che configurano i widget un sistema di query a loro familiare, il formato delle sottoscrizioni è strettamente legato alla sorgente da cui provengono i dati, o, per meglio dire, al protocollo utilizzato dal produttore dei dati.

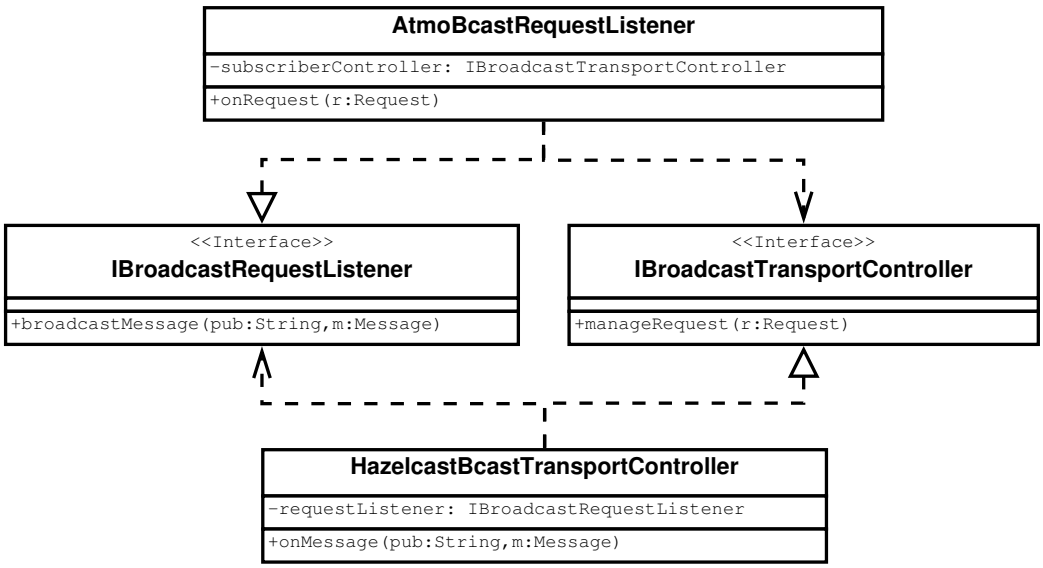


Figura 4.3: Diagramma delle classi principali del web server

Prendiamo ad esempio la sottoscrizione seguente:

```
rda://rda/cryo.S12.CryoStart/MSR1
```

`rda` è il protocollo usato dalla sorgente dati e `cryo.S12.CryoStart/MSR1` rappresenta il dispositivo e l'attributo di cui si richiedono i dati. In questo caso è richiesto lo stato dell'attivazione del supporto criogenico nel settore 12 dell'LHC ed in particolare del dispositivo "MSR1". Queste sottoscrizioni sono inviate al server incapsulate in una struttura JSON che Atmosphere de-serializza a lato server in un oggetto della classe Java opportuna. Nel sorgente 7 è mostrato un esempio di una richiesta client in formato JSON e il sorgente 8 rappresenta l'oggetto Java corrispondente.

Passiamo ora al lato server. Nel diagramma in figura 4.3 è rappresentato lo schema delle classi e delle interfacce che compongono il web server. In questa sezione ci concentriamo sulla classe *AtmoBcastRequestListener*, il cui pseudo codice è riportato nel sorgente 4. Grazie ad alcune annotazioni Atmosphere, è possibile registrare diversi metodi per gestire le fasi di connessione e disconnessione di un client e la ricezione di un messaggio. Più precisamente questa classe ha quattro metodi, due per gestire la connessione

e disconnessione di un nuovo client, uno per gestire le richieste pervenute dai client e l'ultimo preposto ad inviare un messaggio ai client sottoscritti ad una data pubblicazione. Internamente Atmosphere utilizza una classe chiamata *Broadcaster* che rappresenta una pubblicazione. Quando un client invia al server la richiesta di sottoscrizione, istanziamo (se non è già attivo) un broadcaster per quella sottoscrizione ed aggiungiamo ad esso una referenza del client che ne ha fatto la richiesta. A questo punto, quando il modulo del web server preposto alla comunicazione con gli agenti (di cui parleremo nella prossima sezione) riceve dati richiesti da uno o più client, invoca il metodo `broadcastMessage()` sull'interfaccia *IBroadcastRequestListener*. Essendo implementato dalla classe Atmosphere, questo metodo invia il messaggio, opportunamente serializzato come stringa JSON, utilizzando il *Broadcaster* appropriato.

Si noti infine come, utilizzando la classe JavaScript *BroadcastConnection* per il lato client ed implementando l'interfaccia Java *IBroadcastRequestListener* per il server, si introduce uno strato intermedio che permette di avere un sistema più modulare. Questo può risultare molto utile se si ritiene necessario sostituire la libreria su cui è basata la comunicazione client-server, dovendo modificare solamente le classi preposte ad interfacciarsi con il mezzo di comunicazione.

4.3 Comunicazione server-agente

Presentiamo ora la realizzazione del meccanismo di trasmissione e caching dei dati tra web server e agenti che, come abbiamo già visto, è basato su Hazelcast. Facendo riferimento ai diagrammi in figura 4.3 e 4.4, le classi preposte ad interfacciarsi con Hazelcast sono *HazelcastBcastTransportController* e *HazelcastAgentTransportController*.

Iniziamo dall'ostacolo più insidioso da superare nell'implementare questa porzione dell'infrastruttura, ovvero l'instradamento delle richieste dei client all'agente corretto. In altre parole, per non sovraccaricare inutilmente alcuni

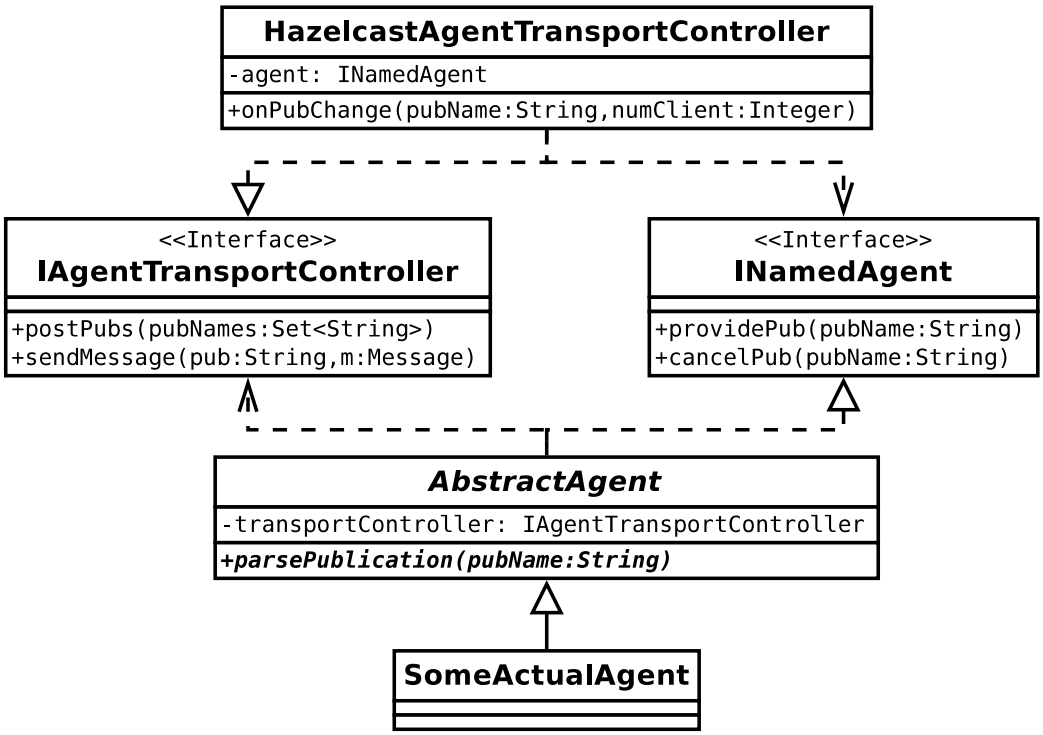


Figura 4.4: Diagramma delle classi principali di un agente

produttori di dati ed il cluster Hazelcast con dati inutili, gli agenti devono inviare i dati solamente se e quando richiesto dai client. Per avere questo comportamento è quindi necessario inoltrare le richieste di sottoscrizione dei client a tutti e soli gli agenti che possono soddisfarle, ovvero a nodi specifici del cluster Hazelcast. Per far fronte a questo problema, abbiamo sfruttato le mappe Java distribuite messe a disposizione da Hazelcast e la possibilità di registrare listener su una mappa in modo da essere notificati quando viene modificata.

Aiutandoci con il diagramma in figura 4.5 vediamo come sono inoltrate agli agenti le richieste di sottoscrizione ricevute dai client. Innanzitutto va detto che un agente, quando si unisce al sistema, pubblica la lista dei protocolli o delle pubblicazioni che può gestire, aggiungendole ad un *set* distribuito, il cui nome è conosciuto da tutti i nodi del cluster. A questo punto crea tante mappe distribuite quanti sono i protocolli o le pubblicazioni

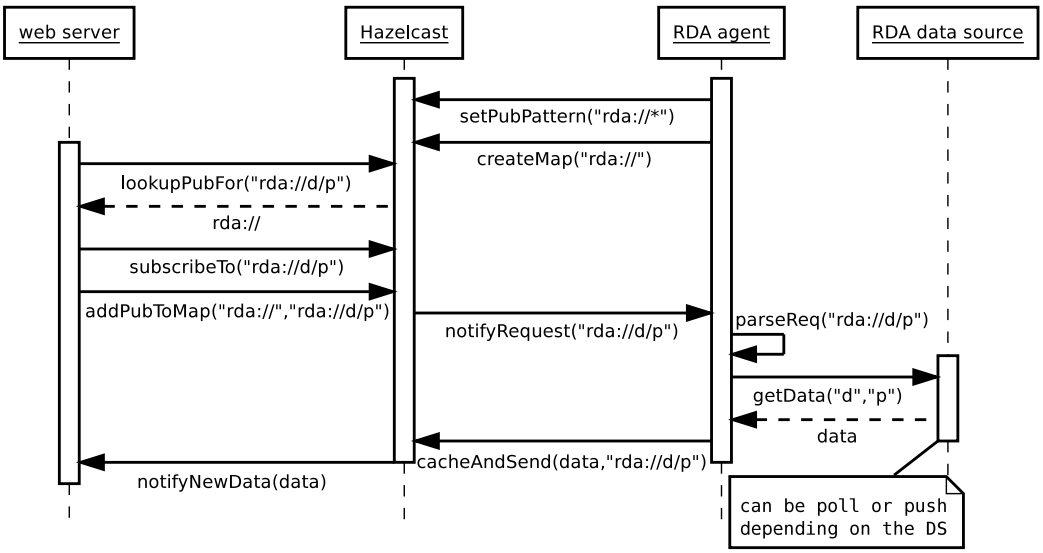


Figura 4.5: Esempio di richiesta e trasmissione dati per il protocollo RDA

a cui può provvedere e su ognuna registra un *listener* per essere notificato della loro modifica. Per capire meglio questo passaggio, pensiamo ad un agente preposto alla gestione di tutte le richieste dirette alle sorgenti dati che utilizzano il protocollo RDA. Questo agente aggiunge il pattern `rda://*` al set distribuito chiamato "pubPatterns" e, se non è presente nel cluster, crea una mappa distribuita chiamata "rda://", su cui registra un listener. A questo punto, un web server, quando riceve la richiesta di sottoscrizione alla pubblicazione `rda://device/property`, cerca il pattern a cui la pubblicazione corrisponde e una volta trovato, aggiunge alla mappa "rda://" la *entry* `<rda://device/property, 1>`, indicando che un client è sottoscritto a quella pubblicazione[1]. L'agente riceve la notifica che la mappa è stata modificata ed inizia ad estrarre i dati corrispondenti alla pubblicazione. Oltre a ciò l'agente acquisisce anche un lock distribuito con lo stesso nome della pubblicazione in oggetto. Questo serve ad evitare che una stessa pubblicazione sia lavorata più volte, sia dallo stesso agente sia da agenti diversi. Va detto che, nel caso il web server non trovi nessun pattern corrispondente alla pub-

[1]Nel caso in cui `N` client sono già sottoscritti alla pubblicazione, la entry diventerebbe `<rda://device/property, N+1>`

blicazione richiesta, la suddetta entry è aggiunta ad una mappa contenente le richieste in sospeso chiamata "pendingReq".

Evidentemente, quando un client si disconnette o cancella una sottoscrizione, in maniera analoga a quanto detto prima, il web server a cui fa riferimento modifica la entry suddetta così: `<rda://device/property, N-1>`. Quando il contatore degli agenti sottoscritti alla pubblicazione diventa 0, l'agente interrompe lo stream di dati.

A questo punto passiamo alla trasmissione e al caching dei dati, problema molto più semplice da risolvere, la cui sequenza di eventi è schematizzata sempre in figura 4.5. Quando nuovi dati sono disponibili[2], l'agente li invia al web server sfruttando le funzionalità *pub/sub* offerte da Hazelcast, ovvero trasmette i dati sul canale etichettato `rda://device/property` a cui il web server si è precedentemente sottoscritto. A questo punto il messaggio è inoltrato ai client che ne hanno fatto richiesta come descritto nella sezione precedente. Il caching dei dati è implementato grazie ad una mappa distribuita preposta a fungere da cache. In questa mappa le entry, aggiornate dagli agenti quando sono disponibili nuovi dati sono nella forma `<{publication}, {lastdata}>`. Quando il web server riceve una richiesta di sottoscrizione da parte di un client, controlla se questa mappa contiene un valore corrispondente alla pubblicazione richiesta; in attesa di nuovi messaggi dagli agenti il dato, se presente in cache, è immediatamente inviato al client.

Vediamo ora gli accorgimenti implementati al fine di far fronte al fallimento di uno o più nodi del cluster. Uno dei problemi da gestire è il crash di un agente. In questo caso è necessario trasferire le pubblicazioni da esso gestite ad un altro agente capace di farsene carico. Nel caso non sia presente nel sistema un agente idoneo, bisogna far in modo che, una volta ripristinato l'agente precedente, esso provveda nuovamente a tutte le pubblicazioni che può gestire. Per realizzare ciò, la classe *HazelcastAgentTransportController* contiene un metodo grazie al quale è possibile verificare, in maniera rapida

[2]La notifica di aggiornamenti dipende dalle funzionalità offerte dalla sorgente dati: l'agente potrebbe ricevere notifiche push o dover fare polling della risorsa.

ed efficiente, se la mappa delle richieste in sospeso (pendingReq), o una delle mappe create dall'agente (ad esempio "rda://"), contengono entry che rappresentano pubblicazioni che l'agente può gestire. Se così è, l'agente si fa carico di fornire i dati per queste pubblicazioni. Questo metodo è invocato da ogni agente in fase di inizializzazione o in seguito alla notifica dell'uscita dal cluster di un nodo.

Un'altra difficoltà da affrontare è rappresentata dal crash di un web server. In questo caso si può perdere traccia del numero esatto di client sottoscritti ad una data pubblicazione. Questo può potenzialmente portare a ritenere attive pubblicazioni che in realtà non lo sono, facendo sì che gli agenti estraggano ed inviino dati inutilmente. Per risolvere questo problema abbiamo sfruttato la possibilità di assegnare alle entry di una mappa un *time to live* o TTL. In questo modo ogni web server reimposta ad intervalli regolari il TTL delle entry che ha gestito. Quando i client cancellano una sottoscrizione, o nel caso in cui il web server abbia dei problemi, il TTL di quella pubblicazione non è più reimpostato e la entry corrispondente viene rimossa. L'agente preposto alla gestione della pubblicazione è notificato di questa rimozione ed interrompe la trasmissione dei dati.

Se da un lato si potrebbe obiettare che questi approcci non sono particolarmente efficienti, vogliamo sottolineare che il cluster è ideato per essere composto da un numero limitato di nodi ed essere abbastanza stabile. I suddetti meccanismi dovrebbero quindi attivarsi solamente in rare occasioni.

Anche qui, come per la comunicazione client-server, l'utilizzo di interfacce Java permette di avere un sistema modulare, grazie al quale la modifica del back-end di comunicazione è trasparente alla logica interna di web server ed agenti.

4.4 Agenti e storage

Passiamo ora allo strato più basso dell'architettura, ovvero quello composto dagli agenti e dal database Elasticsearch. Non è nostra intenzione dilungarci

nella descrizione del funzionamento degli agenti, diciamo solamente che essi sono preposti a fare il parsing delle richieste dei client e mappare queste richieste nel linguaggio di query della sorgente dati con cui si interfacciano. A questo punto, ogni volta che ricevono nuovi dati li inviano ai web server utilizzando il sistema di comunicazione sottostante. Quando tutti i client sottoscritti ad una pubblicazione non sono più interessati a quei dati, gli agenti smettono di inviarli al web server e, in alcuni casi, di estrarli dalla sorgente dati. Com'è possibile vedere dal diagramma in figura 4.4, gli agenti utilizzano l'interfaccia del trasporto e il trasporto utilizza l'interfaccia dell'agente. Questo fa sì che la sostituzione del back-end di comunicazione sia trasparente agli agenti e viceversa rendendo, anche in questo caso, più facili gli sviluppi futuri dell'infrastruttura.

Vediamo ora com'è utilizzato esattamente Elasticsearch. Come abbiamo detto nel capitolo relativo alla progettazione del back-end, alcuni agenti ricevono dati continuativamente dai dispositivi a cui si interfacciano e li indicizzano nel database Elasticsearch. Gli agenti che fanno ciò sono quelli che si interfacciano con sorgenti che contengono dati di cui si vuole visualizzare un'evoluzione nel tempo, come i valori relativi alla temperatura o pressione degli apparati di criogenia. D'altra parte i client, se desiderano visualizzare serie temporali, è necessario che specifichino il protocollo ANX, grazie al quale i web server instradano le richieste all'agente (o agli agenti) che si interfaccia con Elasticsearch. In questo tipo di richieste è possibile specificare diversi parametri, tra cui la finestra temporale di dati da visualizzare, il numero massimo di punti, etc. L'agente opportuno traduce queste richieste in query Elasticsearch. Vediamo un semplice esempio di quanto suddetto. Una pubblicazione a cui un client desidera sottoscriversi può essere la seguente:

```
anx://es/esindex/datatype/dataId?startDate=now-6h&points=500
```

In questo caso il client vuole ricevere l'evoluzione dell'andamento del dispositivo "dataId" (il suo identificativo ha lo stesso formato usato dal protocollo utilizzato dal dispositivo stesso, es.: RDA) nelle ultime sei ore e desidera

ricevere al massimo 500 punti. L'agente che riceve la richiesta estrae i dati richiesti con le modalità desiderate e, dato che Elasticsearch non permette di ricevere notifiche in caso di nuovi dati, verifica ad intervalli regolari se sono presenti nuovi dati per la query suddetta. L'agente smette di interrogare il database quando nessun client è più sottoscritto a quella pubblicazione.

4.5 Monitoring

Come ultimo argomento della sezione riguardante l'implementazione del sistema, illustriamo quali tecniche abbiamo adottato per monitorare l'infrastruttura e identificare eventuali problemi. Come descritto in precedenza, il monitoring è realizzato raccogliendo metriche che indicano lo stato di salute del sistema, confrontandole con parametri prestabiliti e inviando email di allerta in caso di problemi.

Abbiamo utilizzato le funzionalità offerte dal framework Spring per esporre in maniera semplice e conveniente, tramite JMX, alcune metriche specifiche del nostro software. I web server espongono il numero di utenti connessi, di pubblicazioni gestite ed il numero di messaggi inviati ai client e agli agenti. D'altra parte quest'ultimi espongono le informazioni relative alle richieste ricevute dai clienti, ovvero quante e a quali dati corrispondono, e le informazioni relative ai dati ricevuti dai dispositivi o database a cui si interfacciano in termini di query effettuate.

Dato che vogliamo collezionare questi dati tramite protocollo HTTP, utilizziamo il software Jolokia, il quale crea un piccolo web server che permette di accedere agli attributi JMX del programma per cui è configurato tramite un'interfaccia REST. A questo punto è possibile utilizzare Jenkins per eseguire periodicamente uno script Groovy (i cui tratti salienti sono riportati nel sorgente 11) che rileva i valori delle metriche suddette, li confronta e, se lo script rileva un qualche malfunzionamento, Jenkins informa chi di dovere tramite email. I malfunzionamenti di cui parliamo possono essere relativi all'applicazione stessa, oppure possono coinvolgere l'intero sistema.

Nel primo caso può essere rilevato un eccessivo uso di RAM o CPU, nel secondo potrebbe essere rilevato un numero di richieste gestite dagli agenti inferiore al numero di richieste pervenute ai web server. In questa maniera monitoriamo anche lo stato di Elasticsearch, il quale espone nativamente un'interfaccia REST. In questo caso ci assicuriamo solamente che almeno un server Elasticsearch sia in esecuzione e che non segnali errori interni.

Infine va sottolineato che questo metodo di monitoring, attualmente non implementa gli accorgimenti che sarebbero necessari per avere un'esatta rappresentazione dello stato interno di un'infrastruttura distribuita. Quando rileva i valori relativi allo stato di ogni componente non si preoccupa di effettuare un taglio coerente del sistema, ma tiene conto del possibile errore che ne deriva. Dato che non è un sistema critico, abbiamo reputato inutile lo sforzo richiesto per implementare un sistema di monitoring più preciso.

Capitolo 5

Test, validazione e deploy

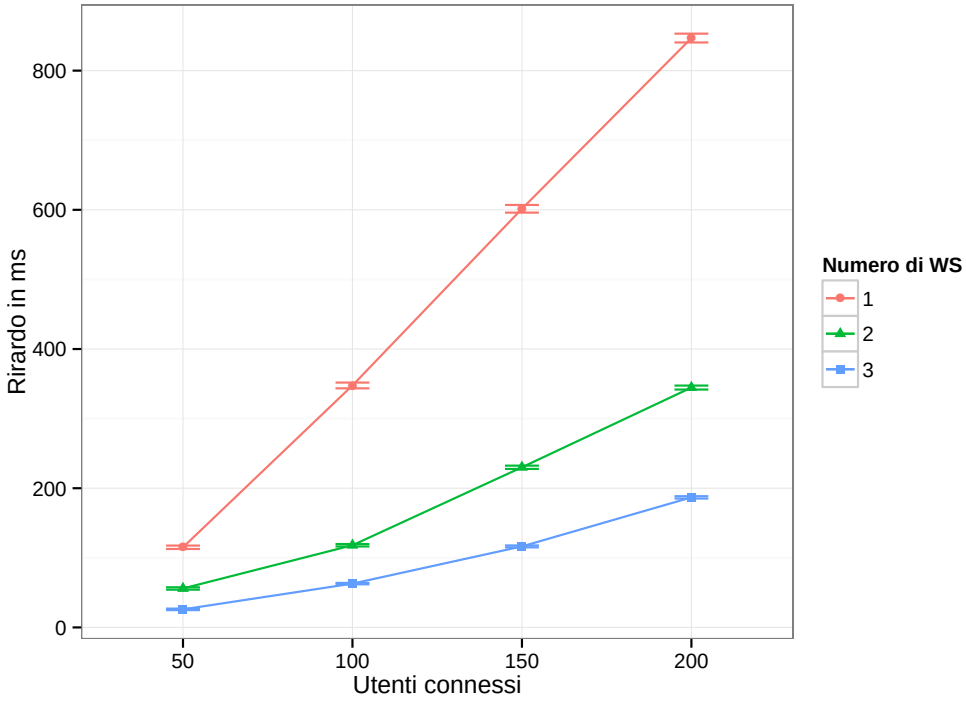
Andiamo ora a trattare le ultime fasi dello sviluppo del sistema, ovvero quelle di testing, validazione e deploy dell'infrastruttura. Innanzitutto va detto che abbiamo creato due infrastrutture separate, una per testare il software nuovo e una per il deploy del software stabile. Questa separazione ci permette di testare le nuove componenti in un ambiente (quasi) completo, grazie al quale si possono effettuare test più completi, ma senza disturbare l'attività dell'infrastruttura stabile. Per ogni iterazione del processo di sviluppo, il nuovo modulo è quindi testato e validato nell'infrastruttura di test ed in seguito integrato nell'infrastruttura di produzione. Dal punto di vista pratico, le fasi di test e deploy sono state coadiuvate dall'uso di Jenkins, il quale può caricare i sorgenti da un sistema di controllo versione (SVN nel nostro caso) e, grazie a Maven, compilare e testare automaticamente il software generando un riepilogo consuntivo della fase di test. In caso di assenza di errori è possibile istruire Jenkins a fare il deploy del sistema.

Per quanto riguarda l'interfaccia utente, abbiamo adottato un approccio di validazione informale, ovvero, al termine dell'implementazione di ogni nuova funzionalità o caratteristica, rilasciamo il software aggiornato nell'ambiente di testing e collezioniamo i feedback dagli utenti. In caso di inadeguatezze, si esegue nuovamente l'iterazione *progettazione-implementazione-validazione*, tenendo conto delle osservazioni fatte dagli utenti.

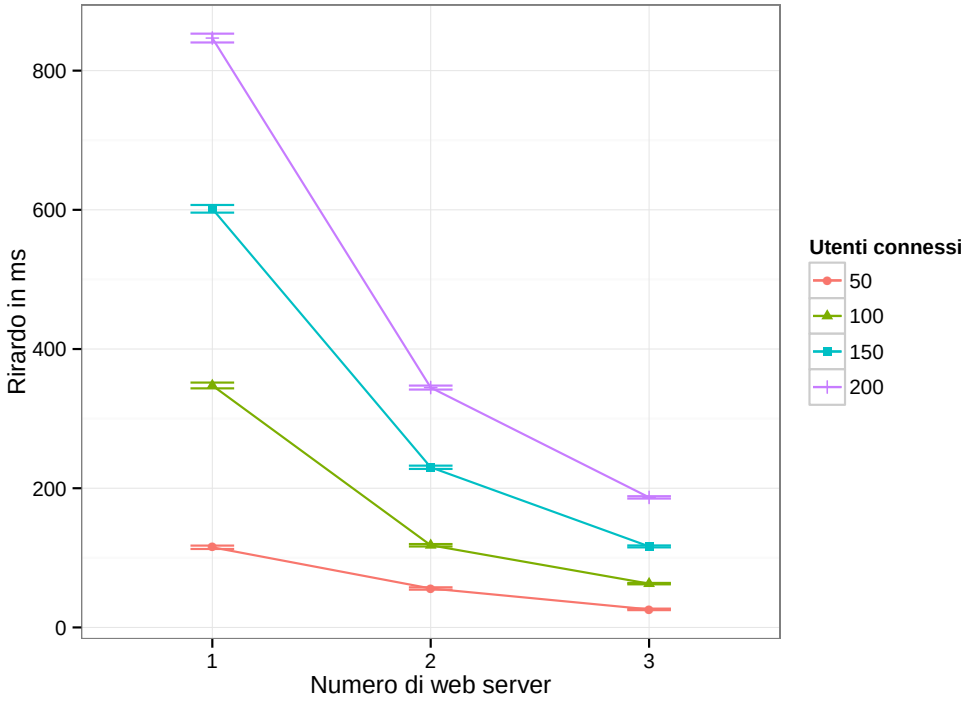
Restando in ambiente client vediamo come, per testare i moduli JavaScript, abbiamo adottato un approccio più formale, utilizzando la libreria di test JavaScript "QUnit" per definire diversi *test case*, ognuno dei quali rappresenta una funzionalità del codice da testare. In particolare la porzione di codice testata in maniera più rigorosa è quella contenente la classe *BroadcastConnection*, ovvero la classe utilizzata dai widget per interagire con i web server.

Analogamente a quanto fatto per la parte JavaScript, al lato server abbiamo creato un *test set* per ognuna delle classi Java contenenti la logica principale del back-end. Per fare ciò abbiamo utilizzato "JUnit". Grazie a questo framework abbiamo creato diversi *unit test* per verificare la correttezza di ogni classe come unità a se stante. In questo modo, ad esempio, abbiamo testato i diversi metodi tramite cui agenti e web server si interfacciano con Hazelcast. Gli *unit test* non sono però sufficienti per testare l'infrastruttura opportunamente. Per verificare se il sistema nel suo complesso rispetta i requisiti desiderati, è necessario creare degli *integration test*. Utilizzando JUnit abbiamo creato diversi casi di test che, utilizzando client ed agenti simulati, verificano se le componenti sono integrate correttamente. In questo caso abbiamo utilizzato agenti e client simulati, perché siamo interessati a testare il funzionamento delle componenti del sistema che si interfacciano con Hazelcast ed Atmosphere.

Al termine dello sviluppo dell'intero progetto, abbiamo inoltre creato uno *stress test* ed un *load test*. Questi test, che possono essere considerati una sottoclasse degli *integration test*, sono molto utili per analizzare alcune funzionalità del sistema. Il test di stress è utilizzato per appurare se l'infrastruttura è resistente al fallimento di uno o più nodi del cluster. In questo test, infatti, abbiamo creato un cluster Hazelcast con due nodi, un agente ed un web server. Mentre essi gestiscono richieste fittizie, uno dei due viene terminato improvvisamente e poi riavviato. Il test, che ha avuto esito positivo, ha appurato che il sistema si riconfigura e, una volta tornati attivi i nodi del cluster, riesce a riprendere la gestione delle richieste.



(a) Ritardo di trasmissione a parità di web server



(b) Ritardo di trasmissione a parità di utenti connessi

Figura 5.1: Risultati del test di carico

Vediamo infine il test di carico, il più importante e per il quale presentiamo e discutiamo i risultati ottenuti. Tramite questo test è possibile verificare che il sistema supporti un adeguato numero di richieste client e che il sistema scali adeguatamente aggiungendo dei web server all'infrastruttura. Per effettuare questo test, abbiamo realizzato un'infrastruttura composta da un agente, tre web server e un numero variabile di client. I web server sono eseguiti su tre macchine virtuali (in esecuzione sullo stesso host) con ridotte capacità, ovvero sono macchine VirtualBox che hanno una CPU da un core a 1,5GHz di clock e 512MB di RAM. L'agente ed i client sono invece eseguiti su una macchina fisica collegata alle tre suddette tramite LAN 100Mb/s. Abbiamo scelto di usare come web server delle macchine con basse prestazioni al fine di sovraccaricare più velocemente il sistema e ottenere dati più significativi.

Descriviamo nello specifico com'è stato condotto il test (i cui risultati sono riportati in figura 5.1. Innanzitutto va tenuto a mente che ogni web server, date le esigue capacità, può accettare un massimo di 200 connessioni. Abbiamo testato tre configurazioni del suddetto sistema, partendo con un solo web server ed aggiungendone uno per ogni configurazione. Abbiamo poi impostato quattro scenari diversi per ogni configurazione del sistema, modificando il numero di utenti connessi, rispettivamente 50, 100, 150, 200. Per ognuno dei 12 test abbiamo fatto sì che i client presenti richiedessero tutti uno stesso stream di dati. A questo punto l'agente è configurato per inviare 50 messaggi con una frequenza di 5 al secondo. Ciò che abbiamo misurato è il ritardo medio che intercorre tra l'invio di un messaggio da parte dell'agente e la sua ricezione da parte di ciascun client.

Al fine di mettere in luce diversi aspetti del test effettuato, abbiamo pensato di esporre i dati in due maniere diverse. In figura 5.1a è possibile vedere quanto aumenta il ritardo in base al numero di utenti connessi a parità di web server nel sistema. D'altra parte, nel grafico in figura 5.1b è ben evidente il fattore di riduzione del ritardo di trasmissione quando si aumenta il numero di web server nel sistema. Il primo grafico mette in risalto come, all'aumentare di utenti connessi, il ritardo di trasmissione aumenti di

un fattore lineare e che questo fattore è minore tanti più sono i web server nel sistema. Il secondo grafico, invece, mette in risalto la scalabilità del sistema, ovvero, aumentando i web server, il ritardo di trasmissione si riduce. Com'è possibile vedere, l'aumento delle performance al crescere dei web server è sublineare. Questo è probabilmente dovuto al fatto che le configurazioni con 2 e 3 web server non sono mai completamente cariche (esse possono sopportare fino a 400 e 600 client rispettivamente), quindi il miglioramento delle prestazioni quando si passa da 2 a 3 web server risulta inferiore rispetto quello riscontrato tra le configurazioni con 1 e 2 web server. Infine, da questo test è possibile stimare immediatamente il carico massimo sopportabile dal sistema in termini di utenti gestibili e frequenza dei messaggi sopportata. Ad esempio si può vedere come l'infrastruttura che presenta un solo web server può gestire 200 client solamente se la frequenza dei messaggi da inoltrare loro è inferiore a circa un messaggio ogni 850ms. Per contro, il sistema con 3 web server può sopportare 200 utenti ed una frequenza di aggiornamento fino a 5 messaggi al secondo. Nel caso questi sistemi vengano sovraccaricati al di sopra delle loro capacità, il buffer dei messaggi da inviare ai client si colmerebbe a tal punto che alcuni di essi verrebbero scartati e persi.

Ciò che si evince dai test suddetti è che il sistema funziona correttamente e rispetta i requisiti richiesti, sia in termini di funzionalità sia in termini di performance. A questo proposito sarebbe però opportuno eseguire *stress* e *load test* più stringenti (con un numero maggiore di utenti e web server) al fine di indagare a fondo le potenzialità dell'infrastruttura.

Conclusioni e sviluppi futuri

Alla luce di quanto visto in questa trattazione, discutiamo infine di come sono stati raggiunti gli obiettivi prefissati e quali lacune del sistema sono ancora da colmare. Per fare ciò ripercorriamo brevemente le problematiche a cui l'infrastruttura deve far fronte e, di volta in volta, vediamo come sono state affrontate, come il sistema soddisfa i requisiti fissati e in cosa consistono i miglioramenti rispetto alle precedenti piattaforme.

Iniziamo dall'interfaccia grafica. Grazie all'utilizzo di OpenSocial, siamo riusciti a soddisfare alcuni dei requisiti principali dell'interfaccia utente. Innanzitutto, tramite questo framework, abbiamo realizzato alcuni gadget all'interno dei quali è possibile la visualizzazione dei dati tramite grafici, diagrammi o immagini. L'utilizzo organico di questi gadget ci ha permesso di realizzare una dashboard modulare per la visualizzazione dei dati dell'LHC e al contempo è possibile esportare i dati in formato grafico integrandoli in altre pagine web. Abbiamo inoltre realizzato un gadget OpenSocial che funge da *bridge* tra la rete interna del CERN e la rete internet, grazie al quale è possibile esportare i dati grezzi in formato *machine readable*. Ripensando alle soluzioni precedenti, ovvero Vistar e la prima LHC Dashboard, risultano evidenti i miglioramenti. Rispetto alle immagini statiche di Vistar, ora abbiamo grafici e diagrammi riutilizzabili. Inoltre, a differenza delle soluzioni precedenti, è ora possibile esportare non solo i dati presentati graficamente, ma anche i dati numerici, ovvero in formato machine readable. Ad ogni modo, in questo caso, uno dei possibili sviluppi è la sostituzione di OpenSocial con i Web Component, ovvero un insieme di standard *W3C* in via di svi-

luppo, che permette di creare contenuti riusabili in maniera più efficiente e standard rispetto ad OpenSocial.

Inoltre, combinando la modularità offerta dai gadget con il framework Bootstrap, abbiamo gettato le basi per permettere di realizzare un'interfaccia *responsive*, ovvero fruibile, da un punto di vista grafico, da diverse piattaforme. Abbiamo utilizzato l'espressione "gettato le basi" perché, se da un lato l'interfaccia si adatta alla dimensione dello schermo, è necessario un maggiore lavoro per far sì che la dashboard sia ben fruibile da dispositivi mobili. Ad esempio bisogna migliorare la visualizzazione dei grafici su schermi piccoli, dove risultano troppo compatti e possono presentare sovrapposizioni di scritte e pulsanti. Anche l'esperienza utente con dispositivi *touch* può essere migliorata, rendendo più efficiente e coerente l'interazione con i diversi widget.

Per quanto riguarda il requisito secondo il quale l'interfaccia deve essere facilmente configurabile, possiamo affermare di essere riusciti a soddisfarlo completamente. Infatti, come abbiamo visto, se nei precedenti sistemi la configurabilità dell'interfaccia utente era molto complicata, nella nostra soluzione essa ed i widget di cui è composta sono configurabili in ogni parte semplicemente modificando alcuni file JSON.

Per ottenere l'interattività dei grafici e l'aggiornamento in tempo reale dei dati, funzionalità completamente assente in Vistar e nella prima LHC Dashboard, è stata di fondamentale importanza la re-implementazione sia dal lato client sia dal lato server delle componenti preposte ad interfacciarsi con Atmosphere, ovvero il framework utilizzato per realizzare la comunicazione client-server. Ancora più importante è però stata l'introduzione di un database Elasticsearch all'interno dell'infrastruttura. Questo infatti permette di archiviare i dati necessari con la massima risoluzione possibile ed inviarli ai client, qualora lo richiedano, con la definizione più opportuna per non sovraccaricarli.

Passiamo ora ai requisiti che il back-end deve soddisfare. L'obbiettivo principale è raccogliere dati da diversi dispositivi eterogenei ed inviarli

ai web server affinché siano inoltrati ai client che ne hanno fatto richiesta. Questo obbiettivo era già stato parzialmente raggiunto nella vecchia LHC Dashboard, in cui era stato introdotto il concetto di agente specializzato per interfacciarsi con una determinata sorgente dati. Nel sistema qui descritto, pur mantenendo lo stesso approccio, abbiamo completamente riprogettato la comunicazione tra agenti e web server al fine di affrontare la sfida maggiore, ovvero realizzare un'infrastruttura modulare e scalabile orizzontalmente, (composta da agenti e web server) in cui sia possibile aggiungere o rimuovere un nodo in maniera trasparente al sistema. Per realizzare ciò abbiamo utilizzato Hazelcast, grazie al quale si può creare un cluster di macchine che all'uscita o entrata di nodi si riconfigura autonomamente, permettendo di scalare orizzontalmente in maniera naturale e di poter garantire la *high avaliability* del sistema. Grazie ad Hazelcast, abbiamo inoltre realizzato un metodo per fare il caching distribuito dei dati ed instradare in maniera efficiente sia le richieste dai web server agli agenti sia i dati in direzione opposta.

Come abbiamo visto e come risulta dai test effettuati e descritti nel capitolo precedente, siamo riusciti a soddisfare tutti i requisiti definiti per il back-end. In questo caso gli sviluppi futuri riguardano il miglioramento dell'efficienza e dell'eleganza del sistema. Innanzitutto, utilizzando al meglio le funzionalità di *data grid* e *computing grid* offerte da Hazelcast, è possibile migliorare l'efficienza della comunicazione tra agenti e web server e l'efficacia della cache. Più precisamente, nell'implementazione attuale, ogni aggiornamento inviato dagli agenti e messo in cache è una stringa JSON che contiene tutti i dati relativi ad una pubblicazione, non solo il valore nuovo. In questo modo, oltre a trasmettere molti dati ogni volta, può capitare che gli stessi siano ridondanti. Se si mantenessero i dati in cache in formato binario sarebbe possibile scaricare su Hazelcast l'onere di interrogare la cache, filtrando ed aggregando i dati opportunamente. In questo modo la richiesta e trasmissione dei dati risulterebbe più immediata.

Per quanto riguarda l'eleganza dell'infrastruttura, si potrebbe indagare a fondo la possibilità di sostituire gli agenti, Hazelcast finanche i web server

con un ESB distribuito. Grazie ad un software come Apache ServiceMix è possibile interfacciarsi con svariati protocolli di comunicazione, inclusi HTTP e WebSocket. Inoltre grazie ad Hazelcast è possibile rendere gli ESB predisposti distribuiti, in modo tale da non perdere scalabilità e *high availability* dell'infrastruttura.

Sarebbe inoltre interessante studiare la possibilità di aggiungere all'infrastruttura un modulo preposto all'analisi in tempo reale dei dati, come ad esempio Apache Storm, così da non dover dipendere da un servizio esterno al sistema. Ciò renderebbe la piattaforma un sistema distribuito completo per l'acquisizione e analisi dati e la supervisione di dispositivi industriali.

In conclusione, ciò che emerge da questa trattazione è che il nuovo sistema da noi realizzato migliora e colma molte delle lacune presenti nelle soluzioni precedenti. Rispetto a Vistar e sistemi similari, esso risolve i problemi legati alla staticità della presentazione dei dati e alla possibilità di interagire con essi. Introduce inoltre il concetto di widget riutilizzabili ed esportabili, funzionalità fondamentale per la distribuzione presso il grande pubblico dei dati dell'LHC. Infine, grazie ad Hazelcast, è stata realizzata un'architettura scalabile e robusta che, a differenza di quella della prima LHC Dashboard, può resistere al crash di alcune sue componenti e a modifiche a run time della sua struttura.

Oltre a queste nuove funzionalità e possibilità, il nostro sistema, come abbiamo visto e com'è naturale per un progetto di questo tipo, fornisce molteplici spunti per interessanti sviluppi futuri.

Appendice A

Tabelle dei test

N. web server	Ritardo medio	CI 95%
1	115.15 ms	±2.43 ms
2	55.96 ms	±1.69 ms
3	25.94 ms	±1.10 ms

Tabella A.1: Ritardo medio di trasmissione con 50 client

N. web server	Ritardo medio	CI 95%
1	347.65 ms	±4.14 ms
2	118.06 ms	±1.74 ms
3	62.93 ms	±1.02 ms

Tabella A.2: Ritardo medio di trasmissione con 100 client

N. web server	Ritardo medio	CI 95%
1	601.48 ms	±5.49 ms
2	230.08 ms	±2.37 ms
3	116.40 ms	±1.39 ms

Tabella A.3: Ritardo medio di trasmissione con 150 client

N. web server	Ritardo medio	CI 95%
1	846.80 ms	±6.33 ms
2	344.64 ms	±2.84 ms
3	186.84 ms	±1.68 ms

Tabella A.4: Ritardo medio di trasmissione con 200 client

Appendice B

Sorgenti

```
{
  "gadgets": {
    "gadget-svg": {
      "gadgetSource":
       ↪  "http://lhcdashboard.web.cern.ch/gadget-svg/gadget.xml"
    },
    "gadget-chart": {
      "gadgetSource":
       ↪  "http://lhcdashboard.web.cern.ch/gadget-chart/gadget.xml"
    }
}
```

Sorgente 1: Esempio di catalog.root

```
{
  "name"   : "Example Page",
  "gadgets": [
    {
      "id"    : "gadget-svg",
      "layout": ".col-md-6",
      "params": {
        "sbsList" : "diagram-subs.js",
        "svgUrl"  : "diagram.svg",
        "styleUrl": "diagram.css",
        "jscbUrl" : "diagram-callbacks.js"
      }
    }, {
      "id"    : "gadget-chart",
      "layout": ".col-md-6",
      "params": {
        "sbsList" : "chart-subs.js",
        "styleUrl": "chart-hcSettings.js",
        "jscbUrl" : "chart-callbacks.js"
      }
    }
  ]
}
```

Sorgente 2: Esempio di file di configurazione di una pagina

```
function BroadcastConnection(serverUrl) {
  this.serviceUrl = serverUrl;
  this.socket = null;
  this.activeSubscriptions = [],
}
BroadcastConnection.prototype.openConnection =
 ↪  function(dataCallback) {
  var request = new atmosphere.AtmosphereRequest();
  // ... set request options like host address and protocols to use
  request.onMessage = function(response) {
    // ... response check
    data = JSON.parse(response.responseBody);
    // ... data check
    dataCallback(data); // callback defined by the gadget
  };
  // ... setup other callbacks
  // open connection with server and get a reference
  this.socket = atmosphere.subscribe(request);
}
BroadcastConnection.prototype.closeConnection = function() { /*...
 ↪  close connection with server */ }
BroadcastConnection.prototype.subscribeTo = function(subscriptions)
 ↪  { /*... send subscribe request */
  this.socket.push(JSON.stringify({
    subscriptionsToAdd : subscriptions
  }));
}
BroadcastConnection.prototype.getData = function(subscriptions) {
 ↪  /*... send a transient subscription request */ }
BroadcastConnection.prototype.unsubscribeFrom =
 ↪  function(subscriptions) { /*... send an unsubscription request
 ↪  */ }
```

Sorgente 3: Pseudocodice della classe JavaScript “BroadcastConnection”

```
@ManagedService(path = "/atmo/broadcast")
public class AtmoBcastRequestListener implements
    IBroadcastRequestListener {
  @Inject
  IBroadcastTransportController transportController;
  BroadcasterFactory broadcasterFactory;
  @Ready
  public void onConnect(final AtmosphereResource r) {
    // setup variables for new client connected
  }
  @Disconnect
  public void onDisconnect(AtmosphereResourceEvent e) {
    // cleanup variables for client disconnected
  }
  @Message(decoders = { RequestDecoder.class })
  public void onRequest(AtmosphereResource client, Request r) {
    Collection<Message> m = transportController.manageRequest(r);
    if(m != null) {
      // send cached values if needed
      sendMessages(m, client);
    }
  }
  @Override
  public void broadcastMessage(String pub, Message m) {
    // send message to all client subscribed to the publication
    broadcasterFactory.get(pub).broadcast(m);
  }
}
```

Sorgente 4: Pseudocodice della classe Java “AtmoBcastRequestListener”

```
public class HazelcastBcastTransportController implements
 ↪  IBroadcastTransportController {
  private HazelcastInstance hz;
  private IBroadcastRequestListener requestListener;
  public Collection<Message> mangeRequest(Request r) {
    Collection<Message> cache;
    for (String pubName: r.subscriptionsToAdd) {
       cache.add(hz.getMap("cache").get(pubName)); // get cache
       String pubMapName = lookupMapName(pubName);
       // add listener to be notified for updates
       hz.getTopic(pubName).addListener(new MessageHandler());
       // this is actually done atomically
       hz.getMap(pubMapName).set(pubName,
        ↪  hz.getMap(pubMapName).getOrDefault(pubName, 0) + 1);
    }
    for (String pubName: r.subscriptionsToRemove) {
       String pubMapName = lookupMapName(pubName);
       hz.getTopic(pubName).removeListener();
       // this is actually done atomically
       hz.getMap(pubMapName).set(pubName,
        ↪  hz.getMap(pubMapName).getOrDefault(pubName, 1) - 1);
    }
    return cache;
  }
}
public class MessageHandler implements MessageListener<Message> {
  private IBroadcastRequestListener requestListener;
  public void onMessage(Message m){
    requestListener.broadcastMessage(m.publication, m);
  }
}
```

Sorgente 5: Pseudocodice della classe “HazelcastBcastTransportController”

```
public class HazelcastAgentTransportController implements
    IAgentTransportController {
  private HazelcastInstance hz;
  private INamedAgent agent;
  public void postPubs(Set<String> pubNames) {
    for(String pubName : pubNames) {
      hz.getSet("publications").add(pubName);
      hz.getMap(pubName).addEntryListener(new RequestHandler);
    }
  }
  public void sendMessage(String pubName, Message m) {
    hz.getMap("cache").set(pubName, m); // cache message
    hz.getTopic(pubName).send(m); // send message
  }
}
public class RequestHandler implements EntryListener<String,
 ↪  Integer> {
  private INamedAgent agent;
  @Override
  public void entryUpdated(EntryEvent<String, Integer> entry) {
    Integer clients = entry.getValue();
    if(clients > 0 /* and no one is handling this pub */) {
      agent.providePub(entry.getKey());
    } else {
      agent.removePub(entry.getKey());
    }
  }
  @Override // entry removed after TTL expiration
  public void entryEvicted(EntryEvent<String, Integer> entry) {
      agent.removePub(entry.getKey());
  }
}
```

Sorgente 6: Pseudocodice della classe “HazelcastAgentTransportController”

```
{
  "transient" : false,
  "subscriptionsToAdd" : [ "pubName" ]
}
```

Sorgente 7: Esempio di richiesta di sottoscrizione in formato JSON

```
public class Request {
  Boolean transient = false;
  Set<String> subscriptionsToAdd = /* <"pubName"> */;
  Set<String> subscriptionsToRemove = null;
}
```

Sorgente 8: Esempio di classe rappresentante una richiesta di sottoscrizione

```
{
  "timestamp" : 0,
  "publication" : "pubName",
  "values" : [ "someData" ]
}
```

Sorgente 9: Esempio di messaggio in formato JSON

```
public class Message {
  Long timestamp = 0;
  String publication = "pubName";
  Collection<Object> values = /* <"someData"> */;
}
```

Sorgente 10: Esempio di classe rappresentante un messaggio

```
def servletAttributes = [
  "requestCount",
  "errorCount",
  "maxTime",
  "processingTime",
  "available",
  "stateName",
  "clientRequestsCount",
  "messagesCount"
]
def agentAttributes = [
  "activePublicationsCount",
  "messagesCount"
]
def servletEvaluator = {
  /* check server health */
}
def agentEvaluator = {
  /* check agent health */
}
def elasticsearchEvaluator = {
  /* check elasticsearch health */
}
/* check system health and report errors */
```

Sorgente 11: Pseudocodice del file monitoring.groovy

Bibliografia

[1] B. Copy, R. Nielser, F. Tilaro, and M. Labrenz, "Mass-accessible controls data for web consumers," in *ICALEPCS2013*, pp. 449–452, 2013.

[2] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret, "The world-wide web," *Commun. ACM*, vol. 37, pp. 76–82, Aug. 1994.

[3] M. Arruat, L. Fernandez, S. Jackson, F. Locci, J.-L. Nougaret, M. Peryt, A. Radeva, M. Sobczak, and M. V. Eynden, "Front-end software architecture," in *ICALEPCS2007*, pp. 310–312, 2007.

[4] O. Andreassen, D. Kudryavtsev, A. Raimondo, and A. Rijllart, "The labview rade framework distributed architecture," in *ICALEPCS2011*, pp. 658–661, 2011.

[5] C. R. Prause, M. Scholten, A. Zimmermann, R. Reiners, and M. Eisenhauer, "Managing the iterative requirements process in a multi-national project using an issue tracker," in *Proceedings of the 2008 IEEE International Conference on Global Software Engineering*, ICGSE '08, (Washington, DC, USA), pp. 151–159, IEEE Computer Society, 2008.

[6] J. Malik, *Agile Project Management with GreenHopper 6 Blueprints*. Packt Publishing, 2013.

[7] M. B. Doar, *Practical JIRA Plugins*. O'Reilly Media, Inc., 2012.

[8] P. Li, *JIRA Essentials, 3rd Edition*. Packt Publishing, 2015.

[9] P. Li, *JIRA Agile Essentials*. Packt Publishing, 2015.

[10] J. Ferguson Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[11] K. Bowersox, *Learning Apache Maven*. O'Reilly Media, Inc., 2015.

[12] B. Copy, M. Zimny, and H. Milcent, "Standards-based open-source plc diagnostics monitoring," in *ICALEPCS2015*, 2015.

[13] Z. Guo, R. Singh, M. Pierce, and Y. Liu, "Investigating the use of gadgets, widgets, and opensocial to build science gateways," in *Proceedings of the 2011 IEEE Seventh International Conference on eScience*, ESCIENCE '11, (Washington, DC, USA), pp. 31–38, IEEE Computer Society, 2011.

[14] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, pp. 45–53, July 2012.

[15] J. Lengstorf and P. Leggetter, *Realtime Web Apps: With HTML5 WebSocket, PHP, and jQuery*. Berkely, CA, USA: Apress, 1st ed., 2013.

[16] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey, "Mining modern repositories with elasticsearch," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 328–331, ACM, 2014.