ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un sistema per gestire e partizionare segnali neurali

Tesi di laurea in
Object oriented programming

Relatore
Prof. Alessandro Ricci

Candidato
Michele Faedi

Correlatore
Dott. Marco Renzi
Dott. Luca Talevi

Anno Accademico 2019-2020

Sommario

La Brain Computer Interface, con l'acronimo BCI, è un mezzo per la comunicazione tra cervello e macchina. La comunicazione si basa sulla emanazione di segnali elettrici del cervello che vengono rilevati da un dispositivo, il quale invia i segnali digitalizzati ad un elaboratore.

I segnali elettrici, chiamati EEG, permettono al cervello di regolare la comunicazione tra le diverse cellule neurali. La BCI intercetta questi segnali e, previa elaborazione, permette di ottenere diversi diagrammi, detti metriche, per poter misurare, sotto svariati aspetti, il funzionamento del cervello. Le ricerche scientifiche sulle EEG hanno rilevato una correlazione tra i segnali elettrici nel cervello di un soggetto con il suo livello di performance e stato emotivo. È quindi possibile comprendere, tramite una serie di parametri, come la mente dei soggetti reagisce a stimoli esterni di svariata tipologia durante lo svolgimento di un'attività.

L'elaboratore, che riceve il segnale dalla BCI, è il componente che si occupa di trasformare i segnali elettrici, generati dal cervello e digitalizzati, in risultati facilmente interpretabili dall'utente. Elaborare i segnali EEG in tempo reale porta a dover utilizzare algoritmi creati appositamente per questo scopo e specifici per le metriche preposte.

Lo scopo di questa tesi è quello di presentare un progetto sullo sviluppo della fase di smistamento dei dati ricevuti dall'elaboratore. Nel contempo si fornirà una conoscenza scientifica minima per comprendere le scelte fatte. Tale progetto è stato reso possibile dalla collaborazione con l'azienda Vibre, che si dedica allo sviluppo di un sistema comprendente BCI ed elaboratore.

Ringraziamenti

Desidero ringraziare i correlatori che sono stati molto disponibili a fornirmi l'aiuto necessario per sviluppare il progetto e in particolare Marco Renzi che mi ha aiutato nella stesura di questo documento.

Ringrazio Giulia Faedi per l'aiuto dato nella revisione del documento.

Indice

Elenco delle figure

Elenco dei listati

Introduzione

Atlas è una piattaforma in costruzione che permette di effettuare analisi neurali per ottenere risultati riguardanti lo stato mentale dei soggetti studiati. Questo documento raccoglie il processo di sviluppo di un componente di *Atlas*, delegato a fare da tramite tra il flusso di dati neurali provenienti dal soggetto e il componente che esegue i calcoli necessari per fornire i risultati richiesti. Il progetto si basa sulla implementazione di un componente chiamato *Gateway*. Il componente che esegue i calcoli, chiamato *NeuroServer*, ha la necessità di utilizzare porzioni di registrazioni parzialmente sovrapposte. Sarà quindi dovere del *Gateway* fare questa sovrapposizione seguendo un preciso ordine di calcoli, chiamato flusso, da affidare al *NeuroServer*.

Il documento è strutturato in modo tale che ogni capitolo approfondisce sempre di più gli argomenti di contorno al *Gateway*, fino ad arrivare al Capitolo 4 in cui si esamina in dettaglio l'implementazione e, nel capitolo successivo, si mostrano le tecniche utilizzate per verificare il corretto adempimento dei requisiti.

Capitolo 1: introduce *Atlas* come piattaforma per elaborare i segnali neurali. Spiega come la BCI viene utilizzata per ricevere i dati sugli impulsi EEG e spiega che tipo di dati possono essere elaborati, fornendo un piccolo quadro sulle ricerche fatte, sui valori calcolabili e sulle loro possibili applicazioni.

Capitolo 2: approfondisce la piattaforma *Atlas* mostrando i vari componenti. Si introducono i requisiti del progetto che verranno approfonditi nei capitoli seguenti.

Capitolo 3: studia più approfonditamente *NeuroFrame* come implementazione di *Atlas*, mostrando tutti i componenti, loro interazioni e parte della struttura interna.

Capitolo 4: spiega il processo di sviluppo del progetto, dando prima una introduzione sulle tecnologie utilizzate, per poi soffermarsi sulla architettura del software prodotto.

Capitolo 5: mostra il processo di verifica del corretto funzionamento del software creato.

Capitolo 6: conclude la tesi riassumendo il contributo dato e le prospettive future.

capitolo 7: informazioni aggiuntive sugli endpoint utilizzati da *Db controller* e *NeuroServer*.

Capitolo 1

Contesto

In questo capitolo si farà l'introduzione alla piattaforma *Atlas*, che permette di analizzare i segnali neurali di un soggetto e di estrapolare metriche, cioè tipologie di risultati, riguardanti lo stato mentale dell'utente. Successivamente verranno spiegate le operazioni che possono essere fatte utilizzando i segnali neurali, anche detti EEG. Verrà spiegato cosa sia una BCI monodirezionale e bidirezionale e cosa misura. Successivamente verrà fatta una introduzione ad alcune tipologie di metriche e come esse sono correlate ai cambiamenti nel comportamento del soggetto studiato; in particolare, verrà spiegato cosa sia il carico cognitivo, come causa la *mental fatigue* e come essa è influenzata dal livello di sonno.

1.1 La piattaforma Atlas e ambiti

I dati, resi disponibili dalla elaborazione, risultano utili quando si ha la necessità di sapere, in maniera oggettiva, quanto il cervello sia sotto sforzo e quanto ancora sia in grado di funzionare a pieno regime. Dagli studi fatti sui possibili utilizzi del segnale EEG, Vibre ha prodotto diversi algoritmi per l'analisi e la gestione dei dati e ha prodotto uno studio sugli utilizzi concreti che possono essere effettuati. Le metriche sono diverse, ma raggruppandole per obbiettivi comuni sono state create 3 categorie:

- Riconoscimento dei progressi fatti in una certa attività.
- Riconoscimento dei sentimenti provati a seguito dell'utilizzo di un prodotto o dal suo design.
- Riconoscimento della difficoltà soggettiva nel compiere un lavoro.

La piattaforma *Atlas* è stata creata e viene utilizzata per fornire diversi tipi di prodotti per l'elaborazione dei segnali neurali: NeuroPerform, NeuroDesign,

NeuroFrame. Ognuno dei quali ha le sue peculiarità sia come obbiettivo e sia come implementazione. Ognuno di questi prodotti ha un'architettura software dedicata e indipendente dalle altre.

1.1.1 NeuroPerform

"Migliora le performance di uno sportivo"

Se qualcosa può essere misurato, può anche essere migliorato. NeuroPerform prevede una misura oggettiva delle performance mentali durante sessioni di allenamento e quantifica i miglioramenti raggiunti.
Ha tre obbiettivi:

Identificazione di flow Essere in grado di capire e talvolta prevedere se il soggetto è in stato di flow, condizione nella quale è in grado di ottenere i risultati migliori dall'allenamento

Identificazione dei punti critici Identifica in quali sport il soggetto si trova ad avere più problemi ad eseguire le singole attività e in cui ha bisogno di più allenamento.

Quantificazione performance Quantifica i miglioramenti raggiunti durante le sessioni di allenamento successive.

1.1.2 NeuroDesign

"Entra nella mente del cliente"

NeuroDesign misura in tempo reale l'apprezzamento di cosa l'utente sta utilizzando e misura le reazioni nel subconscio che influenzano al 95% la scelta nel comprare un prodotto.
Ha tre obbiettivi:

Analisi del prodotto e del packaging Analizza la reazione dell'utente per capire se le qualità di un prodotto, sia dal lato utilitario sia dal lato di design, nella sua fase prototipale sono di interesse per l'utente. Aiuta ad anticipare le reazioni del mercato riguardo ad un nuovo prodotto e valuta la percezione del tuo bene anche in mercati stranieri.

Analisi di campagna commerciale Valuta il coinvolgimento e l'attrattiva di un video commerciale e la sua campagna pubblicitaria. Quantifica il coinvolgimento e lo stress percepito durante la visione di ogni frame in tempo reale.

Valutazione di user experience e user interface Valuta l'usabilità e l'efficienza di e-commerce, sito web o applicazione per aumentare il parco utenti, aumentando il ritorno economico di un investimento.

1.1.3 NeuroFrame

"Riduci il burnout e migliora la qualità del lavoro"

NeuroFrame misura lo stato mentale di uno o più utenti in tempo reale, anticipando i momenti di distrazione, situazioni stressanti e momenti di troppo lavoro.
Ha tre obbiettivi:

Prevenzione del burnout Misura il livello di attenzione e stress di un utente durante il suo lavoro, per prevedere eventi di burnout.

Valutazione del carico di lavoro Valuta il carico di lavoro per rilevare i lavori eccessivi e la consumazione di risorse mentali.

Misuratore di fatica Valuta la fatica fatta alla fine della sessione di lavoro, comparandola con valori di riferimento.

1.2 Interfacce neurali

Il cervello per effettuare la trasmissione di informazioni tra le cellule neurali utilizza segnali elettrici. Questi segnali indicano che una certa area del cervello è in utilizzo. Solitamente tutte le sinapsi hanno un'attività minima, tuttavia alcune aree, in determinate circostanze, sono più utilizzate di altre. Analizzando la differenza di intensità dei campi elettrici, nelle diverse aree del cervello, si individuano le aree più utilizzate. Sapendo quali aree sono utilizzate e sapendo il loro scopo si possono calcolare diverse metriche.

Il rilevatore del segnale elettrico del cervello è chiamato elettroencefalografo. È un macchinario che prevede l'utilizzo di elettrodi, cioè sensori di campo elettrico, sul capo. È possibile posizionare gli elettrodi solamente nelle aree del cervello che si vogliono studiare (solitamente viene fatto per motivi estetici o pratici nei contesti in cui non è richiesto la misurazione dell'intero cervello). Si è osservato che i segnali elettrici hanno un andamento ritmico, e producono le "onde cerebrali"[14] cioè segnali a frequenze ben definite che indicano lo stato del cervello.
Le onde cerebrali a seconda della frequenza si dividono in:

Delta Sono caratterizzate da una frequenza che va da 0,1 a 3,9 hertz. Sono le onde che caratterizzano gli stadi di sonno profondo.

Theta Vanno dai 4 ai 7,9 hertz, caratterizzano gli stadi 1 e 2 del sonno NREM e il sonno REM.

Alfa Sono caratterizzate da una frequenza che va dagli 8 ai 13,9 hertz, sono tipiche della veglia ad occhi chiusi e degli istanti precedenti l'addormentamento.

Beta Vanno dai 14 ai 30 hertz, si registrano in un soggetto in stato di veglia, nel corso di una intensa attività mentale (ad es. durante calcoli matematici) e soprattutto da aree cerebrali frontali.

Gamma Vanno dai 30 ai 42 hertz, caratterizzano gli stati di particolare tensione.

La BCI (brain-computer interface) può essere sia monodirezionale che bidirezionale.

monodirezionale: l'unica interazione è la lettura dei segnali elettrici del sistema nervoso centrale, essi vengono inviati ad un dispositivo che li elabora. Un esempio è il controllo del cursore di un computer per soggetti con disabilità motoria [11].

bidirezionale: l'interazione è sia dal cervello, con segnali elettrici, sia verso il cervello, utilizzando dispositivi appositi. In questo caso si parla di "neuroprotesi", sono dispositivi in grado di sostituire o migliore specifiche funzioni del sistema nervoso. Un esempio è l'impianto cocleare che ripristina la percezione uditiva nelle persona con sordità profonda [18].

Non è argomento di questa tesi discutere le scelte fatte sul raccoglimento e l'elaborazione dei dati in quanto sono campi di ricerca che esulano dall'obbiettivo. Basti sapere che i dati ottenibili da EEG sono diversi, alcuni possono essere ottenuti solo in determinati contesti e possono richiedere la conoscenza di dati pregressi.

Come BCI viene utilizzato un dispositivo chiamato "Muse" [5]. Questo dispositivo è equipaggiato con 4 elettrodi e un'interfaccia di comunicazione Bluetooth. Il "Muse" deve essere accompagnato da un altro dispositivo che si occupa, oltre al collegamento con il "Muse", di fornire un'interfaccia utente e una connessione con l'elaboratore che elabora i dati e li rende utilizzabili.

1.3 Knowledge-workers, carico cognitivo e mental fatigue

I *Knowledge-workers* sono un gruppo crescente di lavoratori sia in economie emergenti che avanzate. La letteratura sui *Knowledge-workers* offre differenti definizioni e classificazioni di chi siano ma in questo contesto gli attribuiamo la connotazione

Figura 1.1: Muse 2 fornito da InteraXon Inc

di chi crea, applica e distribuisce conoscenza[9]. In senso più ampio sono coloro che usano risorse mentali anziché fisiche per raggiungere un obbiettivo. Tali persone sono soggette ad una nuova serie di problematiche che fino a pochi anni fa, e in alcuni luoghi ancora oggi, vengono ignorate o sottostimate.

Quando una persona viene sottoposta ad un problema che richiede l'impiego di risorse mentali, viene sottoposta ad un carico cognitivo, utilizzato comunemente come *cognitive workload*, che causa un impegno mentale per risolvere un problema. Si è osservato che il *cognitive workload* è associato direttamente alla memoria a lungo termine. L'impegno che serve per portare a termine un compito risulta essere correlato alla quantità di dati da ricercare nella propria memoria. Le prestazioni della memoria dipendono quindi non soltanto da quanto complessi siano i ricordi e quanto sono utilizzati (i ricordi più utili vengono ottimizzati per essere ricercati più velocemente), ma anche da quanto tempo la memoria è in uso [11]. Il *cognitive workload* può essere misurato riferendolo a quanto un lavoro è impegnativo, ma anche riferendolo alla resilienza del cervello sottoposto ad un carico mentale. Il carico cognitivo è un fattore molto importante per la creazione di interfacce utente, in questo contesto ad ogni operazione è associato un peso, più è basso e meglio è. Potrebbe non essere semplice individuare le operazioni complesse con un carico cognitivo troppo elevato [3], analizzandolo mediante rilevamenti oggettivi si acquistano dati aggiuntivi per stimare più correttamente il carico. Inizialmente il carico cognitivo veniva misurato mediante sondaggi ad una popolazione che svolgeva lo stesso compito ripetutamente, tuttavia ciò non può essere sufficiente. I sondaggi danno risultati soggettivi e non permettono analisi in tempo reale, tuttavia pos-

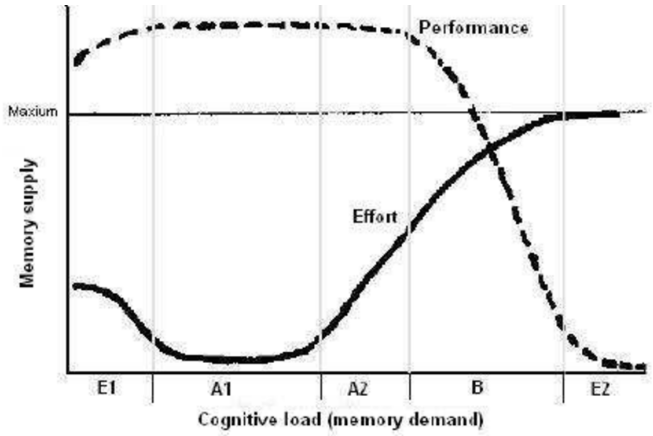

Figura 1.2: Grafico sul degrado delle prestazioni in caso di troppo carico

sono essere comunque utili per analizzare l'andamento delle risposte con sondaggi successivi ed è possibile fare domande mirate su argomenti ritenuti critici.

Il troppo carico cognitivo può causare la *mental fatigue*, cioè un degrado delle performance mentali dovuto all'affaticamento. La persona che si ritrova a dover elaborare troppe informazioni, nella condizione di *mental fatigue*, è portata ad utilizzare ulteriori risorse per concentrarsi. Questo tende a diventare un circolo vizioso che causa l'abbassamento delle prestazioni esponenzialmente con il superamento del carico massimo [4]. In fig. 1.2 si mostra appunto questo evento.

La *mental fatigue* è una conseguenza che dovrebbe essere evitata: essa è una delle cause principali dello stress dovuto all'eccessivo carico di lavoro.

A seguito dello studio sulla *mental fatigue*, sono stati scoperti ulteriori valori misurabili tramite *EEG*, come il livello di sonno che se alto causa diversi problemi.

Sonno

Il sonno è un processo comune a tutti gli animali e serve per riorganizzare il cervello, riposare il fisico e renderli pronti per la nuova giornata. La privazione del sonno è un problema sia dal punto di vista psichico sia da quello fisico, in particolare se la mente non è riposata è in grado di rispondere ad un carico cognitivo molto minore e quindi non è in grado di lavorare a pieno regime [7].

Prontezza

La principale abilità che si perde con la privazione del sonno è la prontezza, cioè l'abilità di rispondere a stimoli esterni in maniera veloce ed efficace. Questo è stato dimostrato dal *test psicomotorio di vigilanza* o PVT: questo test misura il tempo

che impiega il soggetto a rispondere ad uno stimolo visivo [7]. I soggetti sottoposti a questo test mostrano punteggi sensibili al sonno e dimostrano miglioramenti dovuti al ciclo circadiano, come mostrato in fig. 1.3.

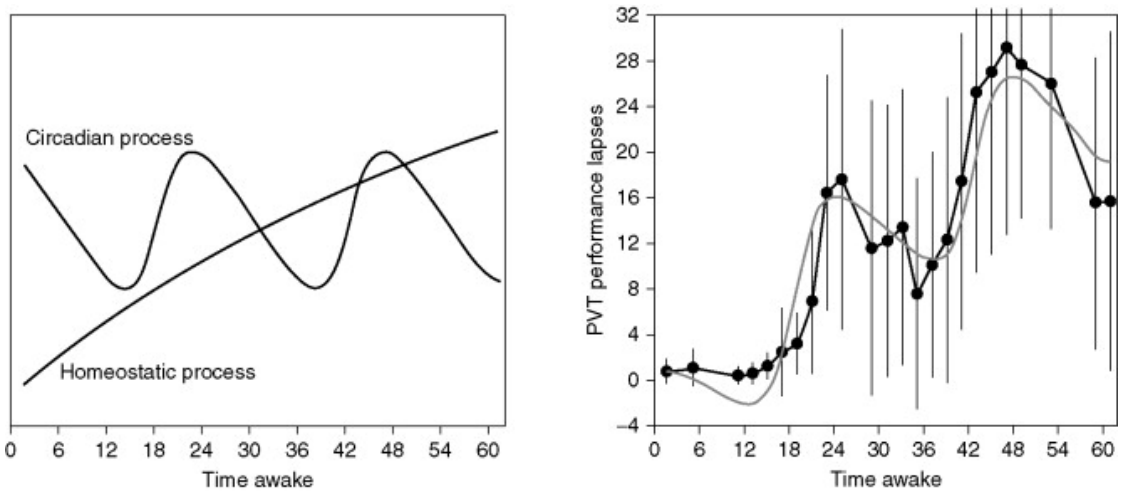

Figura 1.3: Risultati del PVT

Flow

In psicologia positiva [1][15] lo stato di *flow* è uno stato mentale nel quale una persona esegue un'attività totalmente immersa e concentrata. È lo stato di massima performance, nel quale la mente è utilizzata nel pieno del suo potenziale. Lo stato di flow permette di svolgere un'attività in stato di totale immersione e concentrazione, è lo stato in cui un soggetto rende al massimo delle proprie potenzialità riuscendo ad ignorare gli stimoli interni ed esterni di intralcio. È diversa dallo stato di iperconcentrazione, che ha una accezione negativa, in cui la mente concentra il flusso di pensiero in futilità distraendosi dal lavoro assegnatogli.

[1]il termine psicologia positiva designa una prospettiva teorica ed applicativa della psicologia che si occupa dello studio del benessere personale, costrutto al centro della qualità della vita.

Capitolo 2

Studio del problema

In questo capitolo verrà analizzato più in dettaglio *Atlas* e la sua organizzazione senza entrare in dettagli implementativi, ma dando una visione più approfondita dei suoi componenti e come sono interconnessi. Successivamente verranno introdotti i diversi requisiti del progetto e, con l'aiuto di diagrammi, l'analisi dei requisiti soffermandosi sulle problematiche che essi comportano.

2.1 Atlas

Vibre offre servizi che fanno affidamento sulla registrazione, elaborazione e visualizzazione in tempo reale di segnali neurali, utilizzando una sistema di *NeuroServices* interconnesso. Un *NeuroService* è un servizio software che utilizza metriche neurali, proprietà numeriche che descrivono lo stato mentale e/o comportamenti dell'utente, risultanti dall'elaborazione di segnali neurali, nello specifico di elettroencefalogrammi.

Questa piattaforma è composta da tre componenti, che non sono "hardware" o "software", ma concetti più ad alto livello, che possono essere implementati in diversi modi. In fig. 2.1 è sono mostrati esempi di implementazione dell'*Atlas Framework*, con l'indicazione di *NeuroFrame*, perché è l'area di interesse di questo progetto che verrà approfondita in seguito.
I componenti sono:

Source I segnali neurali e il contesto sono raccolti dalla *Source* configurata per il *NeuroService* specifico. È il componente con cui interagisce l'utente.

NeuroService I *NeuroService* utilizzano il *Connector* per connettere la *Source* con i servizi da essa utilizzati. Gestisce la sessione di registrazione: un colloquio tra due entità con l'invio e ricezione di dati durante tutta la durata

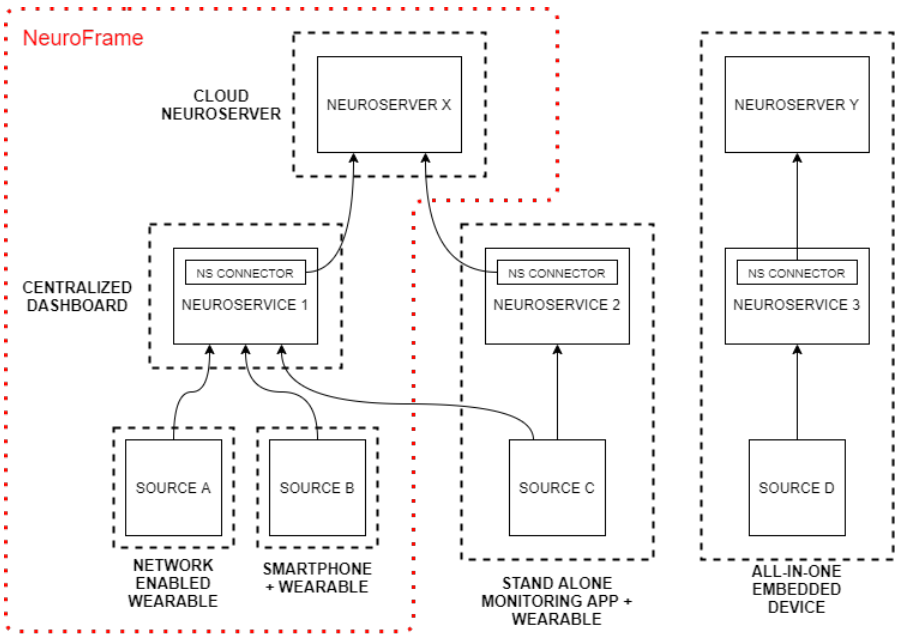

Figura 2.1: Esempio di *Atlas Framework*

del collegamento. Il *NeuroService* si occupa anche di mostrare i dati elaborati possibilmente in tempo reale.

NeuroServer Il *NeuroServer* gestisce tutti i calcoli richiesti ricevuti dal *NeuroService* e ne restituisce il risultato. Non gestisce delle sessioni perciò è *stateless* [1].

2.1.1 Source

La *Source* è il componente della piattaforma lato utente, possiede quindi una interfaccia grafica e gestisce il raccoglimento e l'invio dei dati indicando il contesto in cui sono registrati. Per contesto si intende ogni informazione utile alla comprensione dei dati come, ad esempio, se fatta ad occhi chiusi o aperti, o se svolta durante un lavoro o a riposo. In fig. 2.2 è mostrato un esempio completo di Source.

Ogni *Source* è composta da

- Un dispositivo in grado di leggere i segnali neurali

[1] Un *stateless protocol* è un protocollo di comunicazione in cui nessuna variabile di sessione viene utilizzata dal ricevente, di solito un server. I dati rilevanti vengono inviati in modo tale che ogni pacchetto può essere interpretato singolarmente.

- Un soggetto
- Un ambiente e un contesto in cui questi dati sono rilevati
- Un apparato di invio, software e hardware connesso o integrato al dispositivo, capace di inviare i dati registrati al *NeuroService* utilizzando un protocollo ben definito chiamato *Atlas Comunication Protocol*.

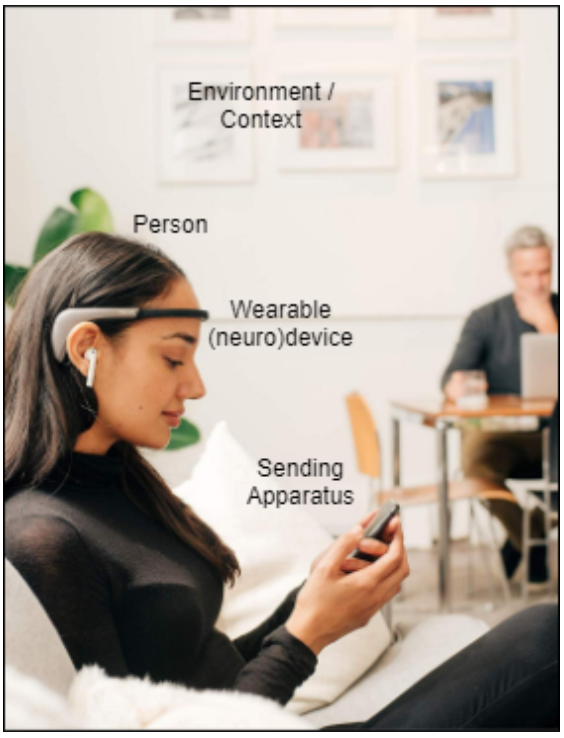

Figura 2.2: `Esempio di Source`

2.1.2 NeuroServer

Il *NeuroServer* è il componente che racchiude tutti gli algoritmi per il calcolo dei segnali neurali. Esso distribuisce i *modelli neurali*[2] che utilizzano segnali neurali e il loro contesto per misurare lo stato mentale degli utenti. La parola server nel nome non è da intendere come software che comunica tramite rete internet, ma come **servitore** di modelli. Nella fig. 2.1 si nota che il *NeuroServer* può essere integrato in un unico dispositivo.

[2]Con modello neurale si intende una serie di algoritmi che producono un risultato partendo da dati neurali

Riassumendo il *NeuroServer* è il componente che esegue i calcoli dei segnali neurali.

Mind#: MindPulse, MindFeel e MindPrint

È stato già preannunciato come con l'EEG sia possibile raggiungere molti risultati interessanti. Per questo Vibre ha scelto di dividere la gamma di algoritmi sviluppati in tre pacchetti:

MindPulse È un insieme di algoritmi per eseguire analisi delle performance mentali: (i) *Cognitive workload* (ii) *Mental fatigue* (iii) *Attentional effort* (iv) *Flow likelihood*.

MindFeel Contiene modelli riguardanti lo stato emotivo come stato di relax e soddisfazione.

MindPrint È un progetto sperimentale che punta a fornire un metodo di autenticazione utilizzando l'impronta neurale unica del soggetto.

2.1.3 NeuroService

Con *NeuroService* si intende un servizio come *NeuroFrame*, *NeuroDesign* o *NeuroPerform*. Essi possono essere implementati in diversi modi che possono non essere compatibili, tuttavia rimane fissa la necessità della *Source* di dover comunicare con il *NeuroService* scelto: per questo è stato creato l'*Atlas (NeuroService) Connector*, che ha appunto il ruolo specifico di permettere alla *Source* di interagire con il resto dei componenti di *Atlas*. I ruoli del *NeuroService* non ricoperti dal *Connector* sono:

- Esporre i risultati delle elaborazioni in modo adeguato
- Salvare i dati per la gestione degli utenti
- Salvare i dati neurali delle diverse sessioni di misurazioni

Atlas (NeuroService) Connector

L'*Atlas (NeuroService) Connector* è la parte del *NeuroService* che ha il compito di gestire le comunicazioni tra le diverse *Source* e il *NeuroServer*. In generale il suo compito è:

- Fornire la configurazione alla *Source* in modo da fargli adottare il contesto e l'ambiente richiesti dal *NeuroService* scelto.

- Definire e mantenere il concetto di sessione aperta tra il *NeuroService* e la *Source* al quale è connesso.
- Gestire il dimensionamento e partizionamento dei segnali neurali in modo che possano essere accettati dagli altri componenti del framework.
- Mantenere l'*Atlas Comunication Protocol* tra i *NeuroService*.
- Stabilire il flusso di elaborazioni necessarie per l'elaborazione delle metriche appartenenti al *NeuroService*.

2.1.4 NeuroFrame

Ponendo particolare attenzione al *NeuroFrame* nella sua implementazione il *NeuroService* è stato diviso in 3 parti indipendenti:

Gateway: alias di *Connector*, svolge lo stesso ruolo.

Db controller: componente delegato alla memorizzazione dei dati sia degli utenti e sia delle loro sessioni di misurazione.

Dashboard: componente delegato all'esposizione dei risultati in modo esaustivo.

La *Source* e *NeuroServer* di *NeuroFrame* non hanno subito modifiche strutturali, cioè il loro ruolo non è stato diviso in sottoparti indipendenti.

2.2 Analisi dei requisiti di sistema

Vibre usa la metodologia Agile e la sua teoria è basata sulla modellazione dei requisiti attraverso user story, cioè i requisiti che ci si aspetta che il progetto abbia; tuttavia, non tutti i requisiti hanno la stessa importanza e alcuni, che sono il cuore del progetto, devono essere sviluppati prima di altri per poter avere un software utilizzabile il più presto possibile, almeno nella versione base.

In questo elaborato è presentato lo sviluppo del *Gateway* e del suo modulo per il dimensionamento e partizionamento dei dati chiamato *ACP windower*, che è stato trattato come progetto a sé stante e parzialmente indipendente data la sua dimensione.

2.2.1 Prioritizzazione tramite metodo MoSCoW

Il metodo MoSCoW è una tecnica di priorizzazione usata per raggiungere una comune comprensione con gli stakeholder riguardo l'importanza di ogni requisito[12]. In questa tecnica ci sono 4 livelli di priorità, a scendere i meno importanti: *Must Have*, *Should Have*, *Could Have* e *Won't Have*.

2.2.2 Must Have

I requisiti denominati *Must Have* hanno l'importanza massima. MUST può essere considerato acronimo di **M**inimum **U**sable **S**ubse**T**, cioè i requisiti minimi che rendono un software utilizzabile.

Gateway

- *Deve essere possibile seguire la misurazione*

  I dati devono essere processati seguendo delle trasformazioni specifiche. Questo verrà spiegato in seguito nella sezione 3.4.3.

- *Deve essere possibile seguire il flusso di calibrazione*

  Per l'elaborazione dei segnali sono richiesti diversi valori pregressi, chiamati di calibrazione, che sono valori che permettono agli algoritmi di adattarsi alle caratteristiche uniche del soggetto. Questi valori devono essere ottenuti utilizzando un flusso di registrazione apposito e i valori ottenuti devono essere utilizzati in tutte le versioni di *Atlas*, perciò il *Gateway* deve rendere possibile calcolare anche metriche non utilizzate da *NeuroFrame*. Questo verrà spiegato successivamente nella sezione 3.4.2

- *Deve essere possibile salvare i dati elaborati*

  Sia i dati di calibrazione, sia i dati di monitoraggio devono essere salvati su database, ma di questo si occupa il *Db controller*, tuttavia è necessario utilizzare il formato richiesto per salvare i dati.

- *Deve essere possibile fornire i dati alla* Source *per eseguire la calibrazione*

  Nella calibrazione ci sono 2 fasi, una a occhi aperti e una a occhi chiusi, perciò bisogna fare in modo che la *Source* si adatti a questi cambiamenti, le fasi sono configurabili e i parametri devono essere forniti dal *NeuroService*.

- *Deve essere in grado di gestire la sessione utente*

  La sessione utente è intesa come mantenimento della connessione per l'invio da parte della *Source* dei dati registrati, per fare ciò ci deve essere l'apertura e la chiusura della sessione mantenendola indipendente da altre precedentemente e successivamente create.

ACP Windower

- *Deve essere possibile memorizzare dati relativi a un soggetto durante una sessione*

  Deve essere in grado di memorizzare i dati consegnatogli.

- *Deve essere possibile leggere i dati salvati*

  I dati salvati devono poter essere letti.

- *Deve essere possibile comporre finestre di segnale formattandole secondo il protocollo ACP*

  Deve gestire i dati con lo schema di *Chunk*, definito nel listato 3.1, perciò con regole ben definite per quanto riguarda la divisione e l'unione dei segnali neurali. Questo requisito verrà approfondito nella sezione 2.5.1.

2.2.3 Should Have

I requisiti denominati *Should Have* sono importanti, ma non necessari per la consegna della prima versione utilizzabile. Mentre questi requisiti possono essere importanti quanto quelli *Must Have*, non sono critici sulle tempistiche o possono esserci dei workaround[3] in modo da svilupparli correttamente successivamente.

Gateway

- *Dovrebbe essere in grado di gestire diverse sessioni utente contemporaneamente*

  Deve essere possibile che più di un utente alla volta sia in grado di seguire il flusso di calibrazione o monitoraggio, quindi senza attese troppo lunghe evitando il mescolamento dei dati.

- *Dovrebbe essere configurabile nei suoi parametri*

  Deve essere possibile configurare i parametri utilizzati per gestire i flussi, i dati di configurazione per la *Source* e tutti i valori utilizzati internamente che potrebbero cambiare nel tempo.

- *Dovrebbe essere semplice aggiornare i flussi per eseguire calibrazione e misurazione*

  Il modo in cui i flussi vengono gestiti deve rendere agile un suo aggiornamento. Questo è un requisito non funzionale e non c'è un obbiettivo definito da raggiungere, quindi è a discrezione del progettista ragionare una struttura adatta.

[3]Risoluzione di problemi temporanea che implica che una soluzione migliore esiste e che verrà sviluppata in future release.

ACP Windower

- *I dati salvati devono essere temporizzati con un TTL*[4].

  I dati già utilizzati non dovrebbero restare memorizzati inutilmente perché occuperebbero memoria senza una reale necessità.

- *Dovrebbe essere riusabile / modulare.*

  Il *ACP windower* è stato pensato per operare all'interno di *Gateway* ma ciò non deve impedirne il riutilizzo in altri contesti.

- *Dovrebbe essere configurabile nei suoi parametri*

  Deve essere possibile configurare i parametri per il dimensionamento e sovrapposizione dei segnali.

- *Dovrebbe gestire diversi utenti contemporaneamente*

  La gestione dei dati in ingresso deve essere fatta in modo tale da non non unire i dati di utenti diversi e si deve garantire la provenienza dei dati dall'utente corretto.

2.2.4 Could Have

I requisiti *Could Have* sono desiderabili ma non necessari e potrebbero migliorare il prodotto con poco sforzo. Sono da includere se tempo e risorse lo permettono.

Per i due progetti non sono stati scelti requisiti con questa priorità.

2.2.5 Won't Have

I requisiti *Won't Have* sono quelli, a detta degli stakeholder, meno importanti o quelli non appropriati in questo momento. Come risultato questi requisiti non sono pianificati e solitamente sono o scartati o vengono considerati come possibili aggiunte a future release.

Gateway

- *Dovrebbe validare i dati in ingresso secondo l'*Atlas Comunication Protocol

  Il *Gateway* è un server che si interfaccia con l'esterno della rete, perciò i dati potrebbero essere inviati da malintenzionati o da applicazioni mal funzionanti, di conseguenza bisogna assicurarsi che i dati in ingresso siano corretti e

[4]Time to live, tempo in cui i dati vengono mantenuti trascorso il quale non ne è garantita l'esistenza

che rispettino l'*Atlas Comunication Protocol*. Questo requisito non è ritenuto prioritario perché una validazione dei dati viene fatta dal *NeuroServer*.

- *Dovrebbe essere in grado mantenere i dati degli utenti anche a seguito di una chiusura inaspettata.*

  Questo requisito modella la qualità di *robustezza*, ossia il comportamento del sistema in caso di errori imprevisti. In questo contesto se il server ha un crash deve poter recuperare i dati della sessione degli utenti in modo tale da recuperare il suo stato senza perdita di dati.

ACP Windower

- *Dovrebbe validare i dati in ingresso secondo il protocollo ACP*

  In questo contesto la validazione dei dati ha un ruolo di minore importanza rispetto a quella del *Gateway* perché questo componente è integrato in altri quindi la correttezza dei dati può essere gestita dall'utilizzatore.

2.3 Atlas Comunication Protocol

Atlas è un sistema distribuito su diversi componenti indipendenti, quindi è importante stabilire un protocollo di comunicazione condiviso per poter facilmente integrare nuove tecnologie mantenendo la compatibilità con quelle già esistenti.

L'*Atlas Comunication Protocol*, chiamato brevemente *ACP*, è l'insieme di schemi, interfacce e metodi di comunicazione usati all'interno dell'*Atlas Framework* solo per i messaggi contenenti segnali neurali.
L'*ACP* ha i seguenti obbiettivi:

- Definire e rafforzare una struttura per tutti i tipi di segnali e metriche che viaggiano nell'ecosistema dei *NeuroServices*.
- Deve rendere facile la sua divisione, unione, memorizzazione e lettura.
- Essere indipendente dal *NeuroServices*
- Essere facile da comprendere, implementare ed estendere in tutte le sue parti.

Sono supportati tre formati:

Chunk Utilizzato tra *Source* e *NeuroService* per contenere i dati neurali e le sue elaborazioni. Un *Chunk* ha una lunghezza espressa come numero di campioni ma viene spesso usata la durata in secondi di registrazione.

User Utilizzato da *Db controller* per i dati degli utenti.

MOC Utilizzato tra *NeuroService* e *NeuroServer*, al suo interno contiene un *Chunk* e i parametri per eseguire l'elaborazione richiesta.

2.4 Modellazione dei requisiti di sistema

Per il progetto è stato scelto di modellare i requisiti con 4 diagrammi:

- Casi d'uso
- Sequenza per sessione con la *Source*
- Classi del Gateway
- Stati per la sessione con i vari flussi di calibrazione e monitoraggio.

2.4.1 Casi d'uso

Nel diagramma 2.3 si nota che tutte le interazioni partono dalla *Source*. Nella fase di invio non sempre si hanno dei dati da elaborare e non sempre si ottengono i dati finali. La fine della sessione non provoca interazioni nel *Db controller* e nel *NeuroServer*, infatti la sessione è un fattore che viene gestito dal *Gateway* e i possibili dati rimasti non ancora elaborati andranno persi. Il diagramma è abbastanza esplicativo, si nota che tutte le interazioni vengono causate dall'interazione con la *Source*.

2.4.2 Sequenza

Nel diagramma 2.4 è stato rappresentato la sequenza di operazioni da fare per eseguire una sessione di elaborazione, che può essere di calibrazione o monitoraggio.

Non è mostrato che il ciclo di invio di *Chunk* viene interrotto a discrezione della *Source*, mentre quello della calibrazione è di durata fissa stabilita dal *Gateway*.

2.4.3 Analisi orientata agli oggetti

Il diagramma a classi 2.5 rappresenta la struttura generale del *Gateway*.
Le classi hanno i seguenti ruoli:

Router Classe che gestisce le varie richieste provenienti dall'esterno e le ridirige alla classe appropriata in base all'endpoint che ha ricevuto il messaggio.

Configuration Classe che viene utilizzata per fornire i dati di configurazione alla *Source*.

Login Classe che esegue il login e se ha avuto successo salva l'utente in *Active_users*.

Active_users Classe che mantiene salvati i dati degli utenti e gestisce lo scadere della sessione dopo un certo lasso di tempo dall'ultima interazione.

Chunk Classe che elabora i *Chunk* ricevuti.

Operation Classe che incapsula il concetto di sessione di registrazione.

Windower Modulo *ACP windower*.

Db_talker Classe con l'obbiettivo di gestire la comunicazione con il *Db controller*.

2.4.4 Analisi orientata agli stati

Il Diagramma degli stati 2.6 rappresenta come deve essere gestita la sessione per i flussi di calibrazione e monitoraggio e di come essi devono gestire i dati dalla *Source*. Poi questi stati modellano il loro flusso (fig. 3.4 e fig. 3.5).

Quando la sessione di registrazione viene creata si attende di ricevere i dati EEG dalla *Source*. Quando viene ricevuto un *Chunk*, al suo interno è contenuto un messaggio con l'intenzione di effettuare un monitoraggio o calibrazione, in base a quel messaggio si entra nello stato opportuno. Quando la calibrazione è finita si salva il risultato e si entra nella fase di monitoraggio. Se durante una di queste fasi la *Source* non invia più i dati si ritorna allo stato iniziale.

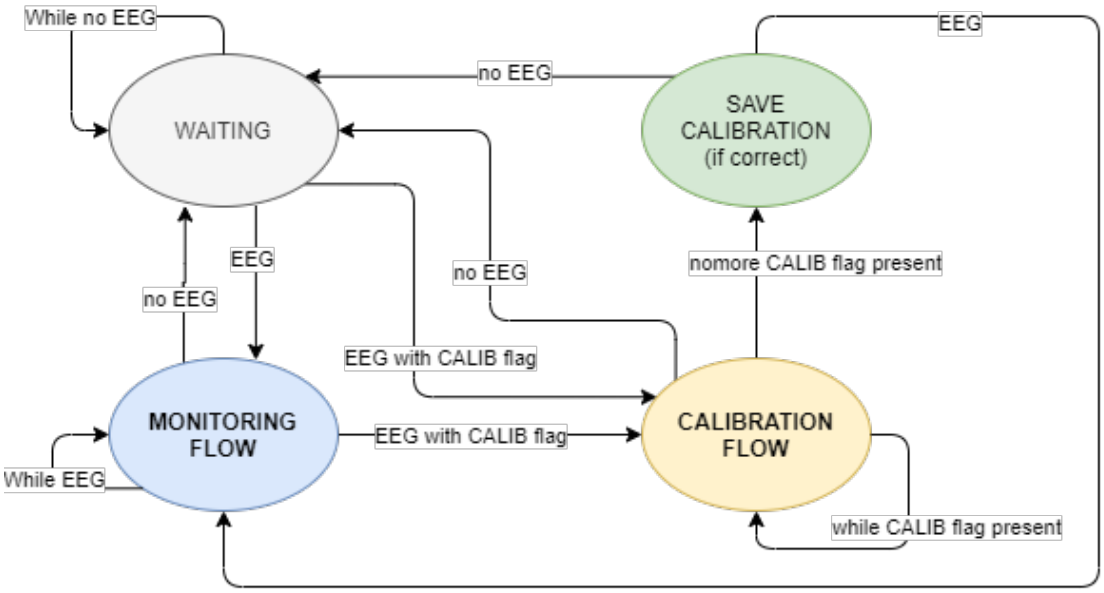

Figura 2.6: Diagramma a stati per la gestione della sessione e dei vari flussi

2.5 Aspetti critici del modulo ACP windower

Il modulo windower ha un ruolo molto importante per la gestione dei *chunk* di dati, esso infatti ha l'obbiettivo, oltre che salvare i dati, di ridimensionarli per averli della lunghezza e caratteristiche richieste dal *NeuroServer*.

2.5.1 Finestratura

Uno dei requisiti *Must Have* del modulo *ACP Windower* è: "*comporre finestre di segnale formattandole secondo il protocollo ACP*". Questo requisito necessita di spiegazione, in particolare il termine finestra di segnale.

Gli algoritmi implementati sul *NeuroServer* richiedono in ingresso delle finestre di segnale sovrapposte. Con finestra si intende una porzione della registrazione effettuata dalla *Source*. Il partizionamento è necessario perché un requisito del *NeuroServer* è la divisione di una registrazione in più parti, le quali devono avere una parziale sovrapposizione; inoltre gli algoritmi, per essere applicati, richiedono finestre di dimensione fissa per l'algoritmo scelto, diversi algoritmi necessitano di finestre di diverse dimensioni. Questo requisito deriva dal fatto che è necessaria una certa continuità nei risultati e per questo le finestre vengono in parte sovrapposte. Ricorda una tecnica di interpolazione chiamata spline interpolation [17], in cui vengono usati dei polinomi di basso grado per interpolare molti punti.

Il *NeuroServer* restituisce un singolo valore scalare. A seguito di questa caratteristica diventa rilevante individuare la dimensione giusta della finestra, poiché se corta il risultato non è molto preciso, se lunga aumenta il tempo richiesto per mostrare e aggiornare i dati all'utente.

Questo problema è stato risolto applicando un algoritmo che viene chiamato *finestratura*, che permette di creare le finestre aggiungendo la porzione finale della precedente con quella iniziale della successiva. La logica verrà approfondita nella sezione 3.2.1 in cui si faranno esempi per aiutare la comprensione.

2.5.2 Gestione dei dati con cache

L'*ACP Windower* è utilizzato per gestire, quindi salvare, i dati voluminosi che devono essere salvati e letti con rapidità.

Una sessione può registrare anche per un'ora o più. Di seguito è mostrato un calcolo con il volume dati richiesto per un'ora di registrazione. Un *Chunk* di *EEG* di 1 secondo occupa 17.648 byte. Quindi il volume di dati per un'ora di registrazione è:

$$17.648_{\text{byte}} * 3600_{\text{second/hours}} = 63.5_{\text{MB/hour}}$$

Non è un traffico dati che può essere gestito senza utilizzare strutture dati appropriate, soprattutto considerando l'evenienza di avere più utenti attivi contemporaneamente. Data questa problematica si è pensato di utilizzare un servizio esterno per gestire i dati che renda il più rapido possibile la loro scrittura e lettura.

2.5.3 Diversi flussi di dati

MindPulse necessita di diverse elaborazioni intermedie per poter arrivare ad elaborare i valori richiesti, queste diverse elaborazioni vengono chiamate flussi. Come spiegato nel sezione 2.5.1 ogni algoritmo ha una lunghezza da utilizzare, quindi per ogni flusso vengono create finestre con caratteristiche diverse.

Quando una finestra è pronta, cioè ha raggiunto la dimensione giusta per essere elaborata, deve essere inviata al *NeuroServer* e la risposta ricevuta deve essere aggiunta al *ACP windower*, la quale può provocare la creazione di una ulteriore finestra. Quindi i passaggi da fare, ricevuto un *Chunk* dalla *Source*, possono causare molte interazioni con il *NeuroServer*. Quindi per una singola sessione bisogna gestire non solamente i dati ricevuti dalla *Source* ma anche i dati intermedi da inviare e ricevere dal *NeuroServer*. Ciò comporta per ogni utente una gestione di dati con significato diverso, che devono quindi essere salvati in modi diversi.

2.5.4 Operazioni con bassa latenza

Come sopracitato l'invio di dati dalla *Source* può causare numerose comunicazioni tra *Gateway* e *NeuroServer*. Uno dei requisiti di *Atlas* è quello di fornire i risultati in tempo reale, il processo di gestione della finestra deve essere fatto possibilmente senza interruzioni o ritardi.

Ragionando sui tre componenti di *NeuroFrame* sia il *Connector* che il *NeuroServer* sono entrambi lato back-end, questo vuol dire che sono entrambi server in loco e perciò il tempo di comunicazione, o RTT, è basso o quasi nullo. Questo verrà discusso più approfonditamente nei capitoli seguenti.

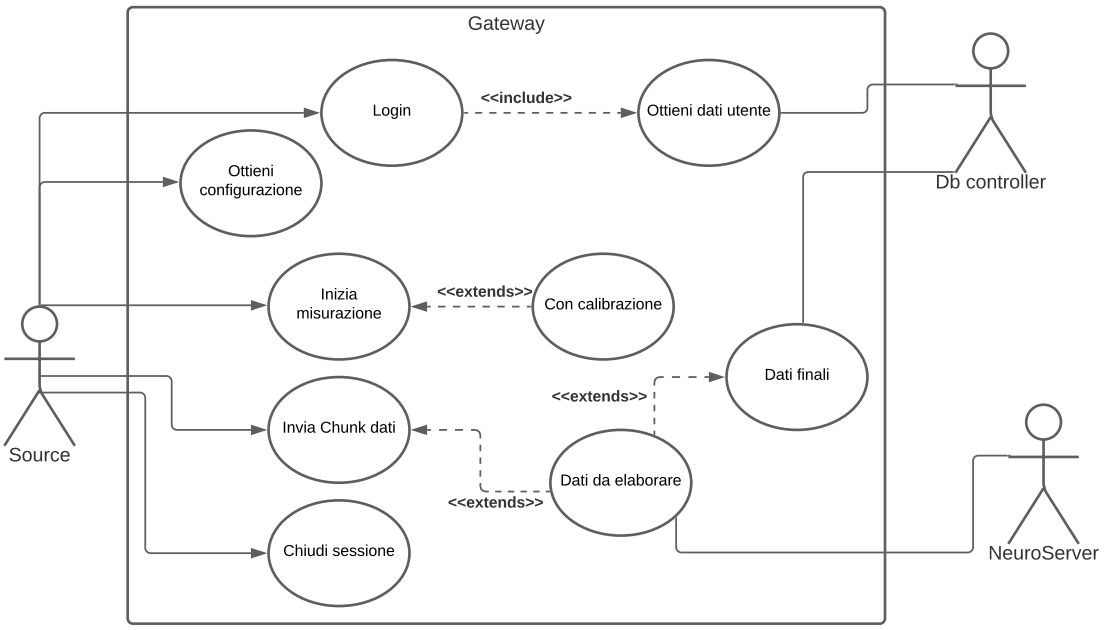

Figura 2.3: Casi d'uso Gateway

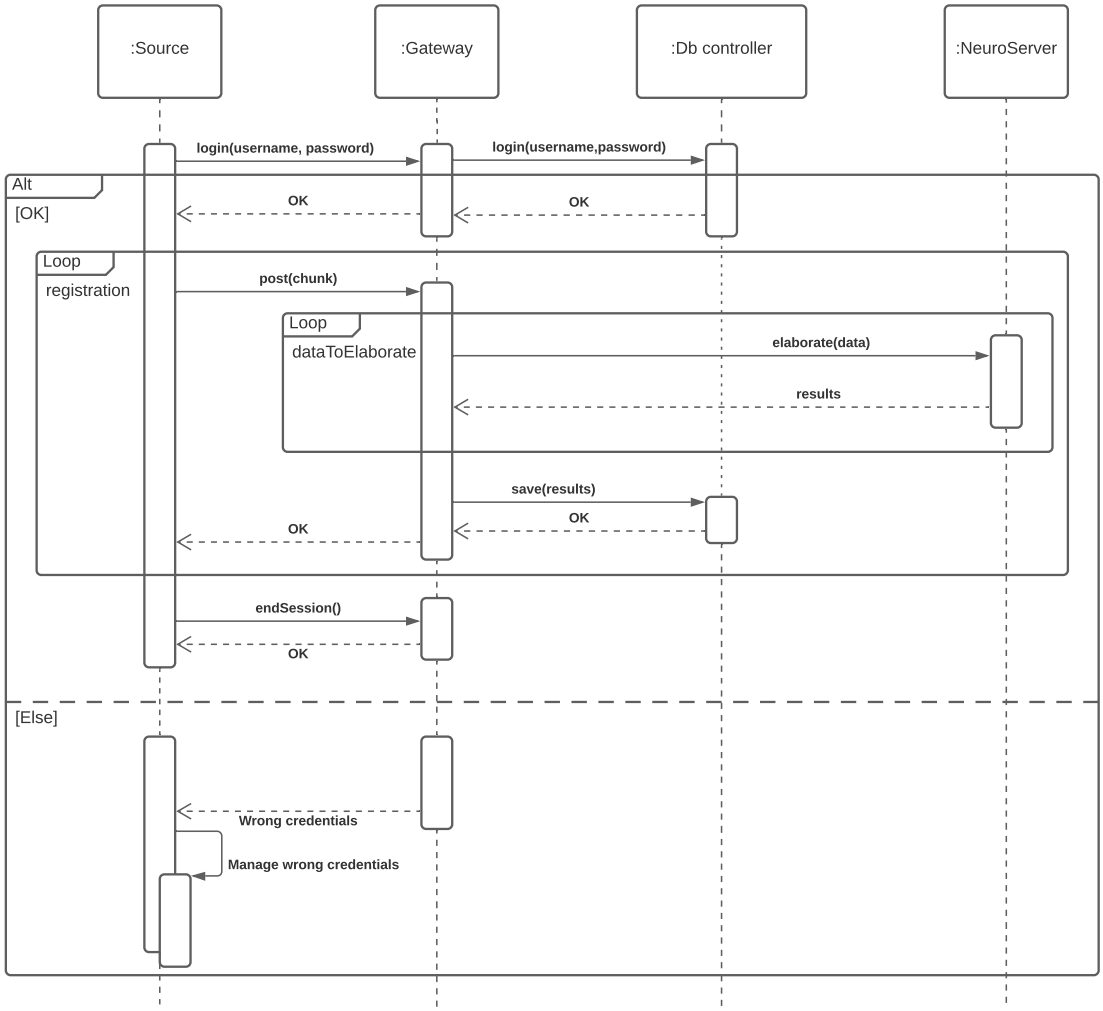

Figura 2.4: Diagramma di sequenza tra Gateway e Source

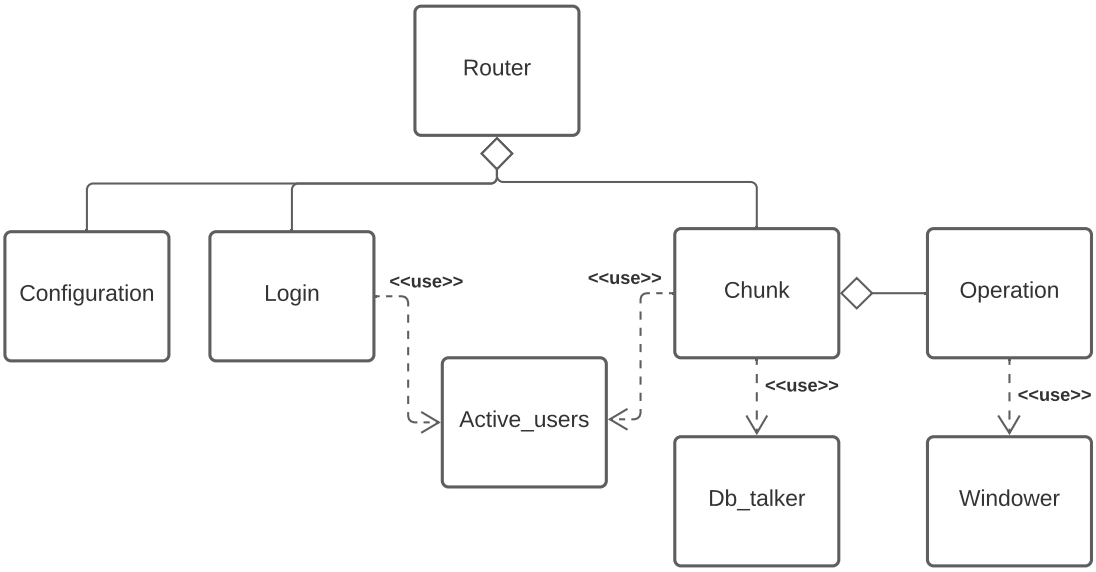

Figura 2.5: Diagramma classi gateway

Capitolo 3

Progettazione del sistema

In questo capitolo verrà presentato il modello organizzativo Agile, utilizzato durante la realizzazione del progetto. Successivamente verranno introdotti i componenti di *NeuroFrame* esterni al *Gateway* che interagiscono con esso, presentando inoltre l'*Atlas Comunication Protocol*. Verranno inoltre illustrate le metriche dell'algoritmo *MindPulse*.

3.1 Sviluppo attraverso la metodologia Agile

La metodologia Agile è stata precedentemente scelta da Vibre e, per mantenere una continuità con le sue metodologie, viene mantenuta anche in questo progetto. La si può definire come: "un insieme di valori, principi e pratiche per modellare il software, che può essere applicato ad un processo di sviluppo in modo efficace e veloce".

Essa è stata creata per risolvere alcuni problemi delle metodologie precedenti ed è caratterizzata da:

- Consegna incrementale software
- Team di progettazione compatti e motivati
- Metodi informali
- Livello minimo di documenti di ingegneria del software
- Comunicazione continua tra sviluppatori e clienti

Il team di sviluppo è composto da tre elementi, di cui uno si è occupato dello sviluppo del *Gateway*, mentre i restanti si sono occupati di validare e verificare il funzionamento di tutti gli altri componenti, cioè *NeuroServer*, *Db controller* e

Source. *NeuroFrame* non è stato commissionato da un cliente esterno ma da Vibre stessa, che lo utilizza per fornire i suoi servizi; quindi, i requisiti sono scelti e ragionati dal team stesso, ciò ha dato più libertà sulle tempistiche permettendo di posticiparle se necessario.

La consegna incrementale del software non viene applicata perché non c'è un cliente a cui sottoporla, tuttavia il processo di sviluppo ha previsto la realizzazione di diverse versioni del software utilizzabile anche senza una reale necessità.

La metodologia Agile prevede che ogni settimana, massimo due, ci sia una nuova *Sprint*. La *Sprint* è un insieme di compiti, che spesso corrispondono a user story, che devono essere svolti nell'arco di tempo assegnato. Il vantaggio di avere scadenze brevi è quello di incentrare l'impegno degli sviluppatori sugli elementi essenziali dando loro un senso di urgenza nello sviluppo delle nuove feature.

Nel capitolo precedente i diagrammi presentati hanno mostrato in maniera superficiale la struttura del *Gateway*, perché sono utilizzati solo come guida generale, la reale documentazione è stata composta durante le varie *Sprint*. I diagrammi che rimangono validi per più tempo sono quelli dinamici, perché la struttura logica di *NeuroFrame* e le interazioni all'interno del suo ecosistema sono state già definite e non sono soggette a cambiamenti, almeno nel medio periodo.

3.1.1 Applicativi utilizzati

La metodologia Agile prevede molta interazione tra i vari stakeholder[1] perché la scarsa presenza di documenti d'ingegneria del software provoca la necessità di fare molti colloqui per stabilire le nuove strutture software da adottare. Per questo sono particolarmente comodi gli applicativi per la comunicazione da remoto e per la gestione dei progetti.

Zenkit

È una piattaforma per la gestione dei progetti e dei team di sviluppo: permette di creare e gestire tutti i progetti dell'azienda potendo assegnare le scadenze, le milestone e gli obbiettivi di ogni *Sprint*.

Attraverso Zenkit è possibile rappresentare i requisiti attraverso il modello *MoSCoW*, presentato in precedenza, per gestire la priorità dei requisiti di un progetto. Ad ogni requisito viene associato il progetto di appartenenza, la priorità, il membro del team a cui è assegnato e la fase in cui risiede. La fase può essere "archivio" per i requisiti implementati, "in corso" se ancora da sviluppare e in "*Sprint*" per quelli in corso.

[1] Gli stakeholder sono i soggetti direttamente o indirettamente coinvolti in un progetto o nell'attività di un'azienda.

Mattermost

È il servizio di chat aziendale: permette il raggruppamento dei vari team di sviluppo in gruppi e la creazione di canali per le comunicazioni di servizio. L'aspetto più interessante è che i messaggi inviati vengono compilati in markdown[2] e questo permette di inviare codice molto leggibile.

Vibre Library

È la pagina in cui viene salvata tutta la documentazione dei vari progetti, le ricerche o alcuni approfondimenti. Utilizza *BookStack*, una piattaforma semplice e gratuita, che può facilmente essere installata e gestita tramite docker.

Prevede la creazione di categorie e sottocategorie chiamate libri e capitoli, al cui interno ci sono le varie pagine. Ogni pagina è un documento in markdown al quale un utente ha il permesso di lettura e di scrittura solo se autorizzato dal creatore. È molto comoda per la documentazione perché il markdown rende semplice e veloce scrivere i documenti e li formatta rendendoli semplici da leggere.

Miro

È un applicativo che fornisce lavagne interattive in cui è possibile creare schizzi, incollare immagini, utilizzare link e molto altro in modalità condivisa. È stato utilizzato per organizzarsi e dialogare in smartworking fornendo grafici e schemi in maniera facile e veloce.

3.2 Feature base del sistema

Il progetto è stato diviso in due parti distinte: *ACP windower* e *Gateway*.

3.2.1 ACP windower

Il progetto è stato inizialmente svolto completando il modulo *ACP windower* perché senza di esso il *Gateway* non può raggiungere una funzionalità di base. È stato chiaro fin da subito che l'*ACP windower* necessita di uno studio approfondito sulle strategie da utilizzare per rendere efficiente il suo utilizzo.

È stato deciso di implementare subito tutti i requisiti *Must Have* e *Should Have*, almeno nella creazione delle strutture dati, perché doverle aggiornare per ogni *Sprint* porterebbe ad inutili dispendi di tempo nel creare funzioni base funzionanti ma non utilizzabili per la mancanza del *Gateway*.

[2]Il markdown è un linguaggio di markup che prevede la traduzione in codice html rendendo la formattazione di documenti molto semplice e veloce.

Finestratura

Come già introdotto i diversi algoritmi implementati nel *NeuroServer* richiedono in ingresso dati di lunghezza precisa, detto in altri termini una metrica richiede in input una "*finestra di segnale lunga x*" (con x che dipende dall'algoritmo richiesto).

Con *finestra di segnale lunga x* si intende un *Chunk* che corrisponde ad una registrazione di x secondi. Per poter elaborare la registrazione in tempo reale è necessario che ogni finestra sia parzialmente sovrapposta con quella precedente: scelta l'area di sovrapposizione si prende la parte finale della finestra precedente e si concatena all'inizio della successiva. Questo porta a definire una ulteriore variabile chiamata *step* che indica quanti dati vengono aggiunti alle nuove finestre.

In fig. 3.1 è rappresentato il concetto di finestre di segnale, di step e sovrapposizione.

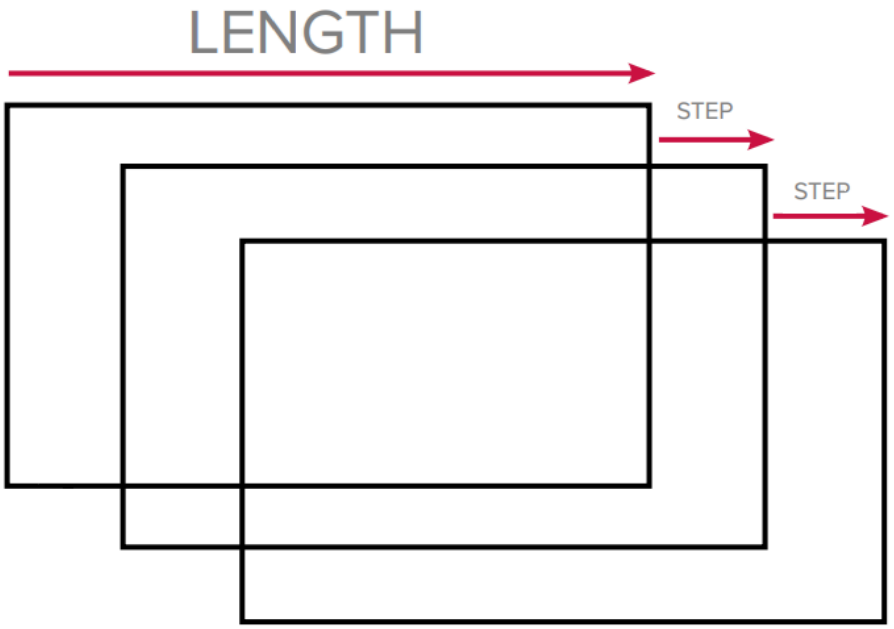

Figura 3.1: Finestratura di segnale neurale

Per spiegare questo concetto che sta alla base dell'*ACP windower* è proposto l'esempio per il calcolo di *Alpha Prevalence*.

Calcolo di Alpha Prevalence

In questo esempio si osserva come in input si ricevano finestre di EEG lunghe 5 secondi. La dimensione della finestra richiesta dal *NeuroServer* è di 10 secondi con step di 5 secondi, quindi saranno necessari due *Chunk* per la creazione del risultato. Vedi fig. 3.2 per la prima fase.

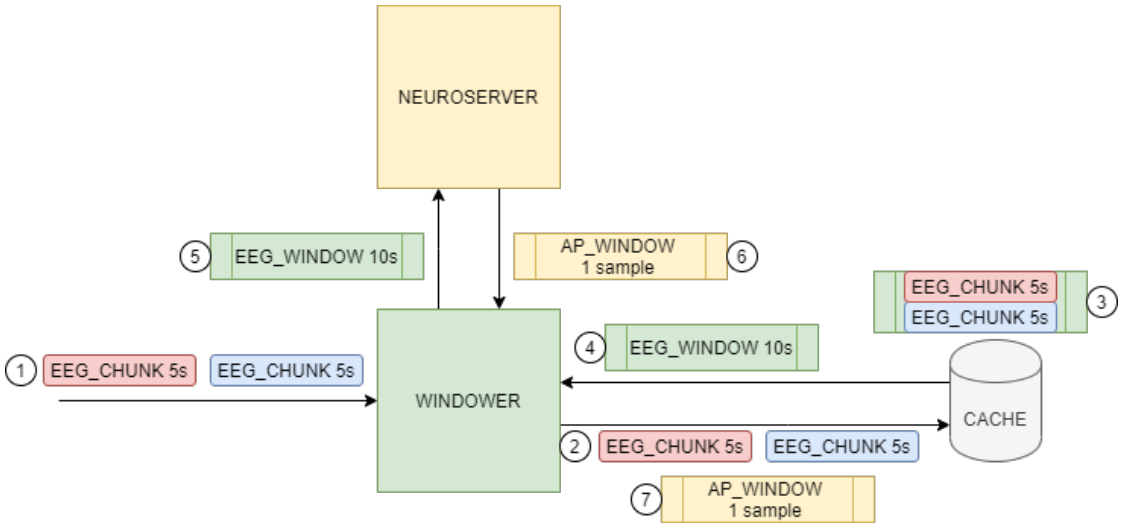

Figura 3.2: Fase 1 per calcolo di Alpha Prevalence

Una volta creata la prima finestra si può iniziare a crearne di nuove quando nella cache si aggiunge uno step, in questo caso di 5 secondi come il *Chunk* in ingresso. Quindi a ogni nuovo *Chunk* verranno create nuove finestre perché la cache è già piena e basta fornire i nuovi dati. Vedi fig. 3.3 per la seconda fase.

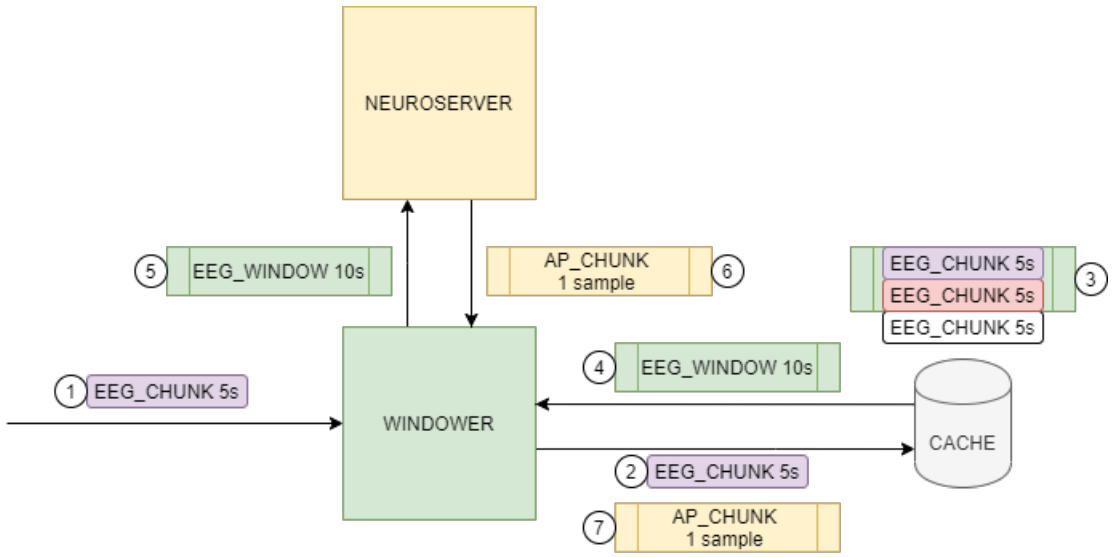

Figura 3.3: Fase 2 per calcolo di Alpha Prevalence

Cache

Come mostrato nelle figure 3.2 e 3.3 i *Chunk* vengono salvati in una *cache* che viene rappresentata come dispositivo esterno. Il *Gateway* è spesso soggetto ad un inteso traffico di dati, diventa quindi ragionevole pensare che i dati debbano essere consegnati ad una entità esterna per poterli utilizzare solo quando richiesti, estraendo così dall'*ACP windower* la funzione di database. Salvare i dati in un dispositivo esterno rende possibile anche recuperare la sessione di registrazione interrotta a seguito di un crash imprevisto.

La *cache* svolge quindi questo ruolo: salvare i dati utilizzando strutture dati appropriate per aumentarne l'efficienza di utilizzo e rende possibile recuperare la sessione in caso di crash.

3.2.2 Gateway

Una volta che l'*ACP windower* è stato completato si può iniziare con il ciclo di sviluppo del *Gateway*. Il *Gateway* ha avuto una rapida implementazione dei requisiti *Must Have*, ma durante lo sviluppo dei requisiti *Should Have* c'è stato un rallentamento dovuto al requisito sulla estensibilità a nuovi flussi perché ha richiesto lo sviluppo di strutture dati ad hoc per renderle leggibili e manipolabili. La soluzione trovata per l'estensibilità verrà spiegata nel sezione 4.4.9.

3.3 NeuroFrame come sistema distribuito

Con *sistema distribuito* si intende una tipologia di sistema informatico costituito da un insieme di processi interconnessi tra di loro in cui le comunicazioni avvengono con l'invio di messaggi [1].

Esistono diverse definizioni che rendono più chiaro il concetto:

- "Un sistema distribuito è una porzione di software che assicura che un insieme di calcolatori appaiano come un unico sistema coerente agli utenti del sistema stesso" [8]
- "Un sistema distribuito consiste di un insieme di calcolatori autonomi, connessi fra loro tramite una rete e un middleware di distribuzione, che permette ai computer di coordinare le loro attività e di condividere le risorse del sistema, in modo che gli utenti percepiscano il sistema come un unico servizio integrato di calcolo" [2]

Questo tipo di architettura si presta molto per fornire servizi molto complessi perché delega la gestione delle varie parti a sistemi indipendenti. Il vantaggio più importante è che alcuni componenti possono essere riutilizzati da nuovi sistemi

aumentando così la riusabilità del software. È importante quindi utilizzare delle logiche di comunicazione standardizzate, per questo si è deciso di utilizzare *ACP* come protocollo standard per la comunicazione di segnali neurali tra i componenti di *Atlas*. La comunicazione interna dei componenti, che possono anche essere più sottosistemi, non è stata definita perché può variare a seconda dell'architettura scelta.

Software as a service

Software as a service, chiamato SaaS, è un modello di distribuzione del software dove il produttore sviluppa e gestisce un'applicazione web, mettendola a disposizione dei clienti via internet, spesso utilizzando un servizio di cloud computing.

Questo tipo di distribuzione prevede di affidare la manutenzione dei server a terzi, delegando così la gestione della macchina fisica, la connessione internet e la corrente elettrica.

Il mercato delle SaaS offre diversi vantaggi e svantaggi:

Vantaggi

- È possibile fornire il servizio in abbonamento dando la possibilità così di utilizzarlo solo per un tempo limitato, annullando inoltre il mercato delle versioni pirata.
- Offre soluzioni molto flessibili che possono essere create ad hoc per specifici utenti.
- Utilizzo di dispositivi a basso costo che fanno solo da intermediari tra l'utente e il servizio, che può richiedere calcoli molto esosi in termini di risorse hardware.

Svantaggi

- I dati del cliente risiedono nei server del fornitore e quindi bisogna operare il trattamento dei dati per la gestione della privacy con misure di sicurezza adeguate.
- Non c'è più un solo sistema che si occupa di fornire il servizio ma almeno due: uno dal lato cliente, tipicamente pagina web; e uno lato server per fornire il servizio. Di conseguenza è necessario il mantenimento due software indipendenti.
- I problemi di connessione internet possono precludere l'utilizzo del servizio. Questo problema è delegato all'internet service provider del cliente, che può non agire tempestivamente per risolvere il problema.

3.3.1 Componenti NeuroService di NeuroFrame

NeuroFrame segue la struttura generale di *Atlas*. In questa implementazione il *NeuroService* svolge un ruolo molto complesso, per questo motivo è stato diviso in tre sistemi indipendenti:

Gateway Server che ha il ruolo di *Connector* come spiegato in sezione 2.1.3

Db controller Svolge il ruolo di database in cui salvare i dati degli utenti, compresi i dati di calibrazione e le loro misurazioni.

Dashboard Pagina web che espone i risultati delle elaborazioni in tempo reale.

Originariamente *Atlas* non ha questa divisione in sottosistemi, perciò le comunicazioni tra essi possono non utilizzare lo standard *ACP*.

3.3.2 Atlas Comunication Protocol

L'*Atlas Comunication Protocol* è stato introdotto, nel capitolo precedente, come protocollo utilizzato per la comunicazione tra i vari componenti di *Atlas* solo per i messaggi contenenti segnali neurali.

La comunicazione tra i componenti è fatta principalmente tramite protocolli di rete, di conseguenza serve un formato in cui trasferire i dati e lo standard de facto è il *JSON* [10]. Il *JSON* offre una struttura leggibile e molto espandibile, per questo *Atlas Comunication Protocol* lo utilizza come formato per la comunicazione.
Per seguire gli scopi del *ACP*, già elencati nella sezione 2.3, sono stati creati 3 diversi tipi di schemi: *Chunk*, *User*, *MOC*.

Chunk

Nel listato 3.1 è mostrato lo schema completo del *Chunk* e nel listato 3.2 è mostrato un suo esempio. I campi più importanti sono *signal*, *ts* e *flags*.

In *ts* sono contenuti i timestamp di ogni *sample*, il *sample* è il gruppo di valori riferiti ad uno stesso istante di tempo, nell'esempio `[5.0,7.0]` è un *sample*. In *signal* è contenuto il vettore di *sample*, che è il valore che occupa più spazio del messaggio.

Il campo *flags* è utilizzato per esprimere commenti o altri comandi. È una coppia chiave valore, la chiave è una stringa, mentre il valore è una stringa seguita da un vettore che contiene, nel caso di *ts_list*, i timestamp, oppure, nel caso di *ts_tange*, un range delineato dai due estremi (-1 è utilizzato per indicare che l'estremo non è contenuto nel vettore di *ts*).

```
{
    "vibre_id": string,
    "session_id": string,
    "channels": string[],
    "signal": number[][],
    "ts": number[],
    "flags": Map<string, {value?: string,
        ts_range?:[number, number], ts_list?:number[] }>,
    "start": Date,
    "stop": Date,
    "signal_type": string
}
```

Listato 3.1: *Chunk datatype*

```
{
    "vibre_id":"utente1",
    "session_id":"a6f08d87f08a91eb39faa09e",
    "channels":["TP8", "AF9", "TP9", "AF10" ],
    "signal":[[5.0, 7.0, 1.0, 4.2],[3.0, 4.0, 0.2, 1.3]],
    "ts":[100, 300],
    "flags":{
      "eyes_closed":{ "ts_range":[100,200] }
    },
    "start":"2020-10-02 16:09:36.394495",
    "stop":"2020-10-02 16:09:36.394495",
    "signal_type":"eeg"
}
```

Listato 3.2: esempio di *Chunk*

User

Nel listato 3.3 è mostrato lo schema completo dello *User* e nel listato 3.4 è mostrato un suo esempio. I campi importanti sono i *biometrics* che sono i valori di calibrazione. Se un soggetto non ha ancora effettuato la calibrazione vengono utilizzati dei valori di default, che se utilizzati per il monitoraggio possono portare a risultati non corretti.

```
{
    "vibre_id": string,
    "email": string,
    "biometrics":{
        "alpha_cog": number,
        "zeroload": number,
        "resting_asymmetry": number
    }
}
```

Listato 3.3: *User datatype*

```
{
    "vibre_id":"utente1",
    "email":"mario.bianchi@gmail.com",
    "biometrics":{
        "alpha_cog":9.8,
        "zeroload":25.0,
        "resting_asymmetry":0.0
    }
}
```

Listato 3.4: esempio di *User datatype*

Model Options Chunk - MOC

Il *Model Options Chunk*, è utilizzato solo tra *Connector* e *NeuroServer* per richiedere una elaborazione di una metrica specifica. Il campo *model* viene utilizzato per indicare che tipo di elaborazione eseguire. Nel campo *options* vengono utilizzati valori aggiuntivi per l'operazione come, ad esempio, i dati di calibrazione del soggetto o alcune opzioni per l'elaborazione come, ad esempio, il *threshold*[3]. Il campo *Chunk* è un valore nell'omonimo formato, mostrato nella sezione 3.3.2.

Nel listato 3.5 è mostrata la sua versione completa meno quella del *Chunk*, e nel listato 3.6 un suo esempio.

[3]Valore utilizzato per indicare il limite, superato il quale i dati sono considerati inutilizzabili per il troppo rumore.

```
{
    "model": string,
    "options": Map<string, any>,
    "chunk": "Chunk_Datatype"
}
```

Listato 3.5: *MOC datatype*

```
{
    "model":"cognitive_workload",
    "options":{
        "zeroload": 25.0
    },
    "chunk": {...}
}
```

Listato 3.6: esempio di *MOC datatype*

3.4 MindPulse

MindPulse è un insieme di algoritmi per l'analisi delle performance mentali. Di seguito sono elencati i vari algoritmi con i loro requisiti e contesti di utilizzo.

3.4.1 Algoritmi MindPulse

Come convenzione di nomi si è scelto di usare la stessa sia per l'algoritmo sia per il *signal_type* prodotto (vedi il listato 3.1). Siccome questi algoritmi sono implementati nel *NeuroServer* il protocollo di comunicazione per richiederne l'elaborazione è il *MOC*. Diversi algoritmi richiedono opzioni, cioè valori aggiuntivi, che devono essere contenuti nel campo *options*.

Alpha Center of Gravity

Stima della frequenza alfa principale del soggetto.

- **Identificatori:** *alpha_cog*, *acog*
- **Opzioni utilizzabili:**
  - *derivation*
  - *threshold*
  - *fs*
  - *bandpass_limits*
  - *cog_search_limits*
  - *debug*
- **Requisiti:** *signal_type*="EEG"
- **Contesto:** calibrazione occhi chiusi

Alpha Prevalence

Misura la prevalenza dell'onda alpha di un soggetto, con un range da 0 a 100(%).

- **Identificatori:** *alpha_prevalence*, *ap*
- **Opzioni utilizzabili:**
  - *derivation*
  - *threshold*
  - *fs*
  - *alpha_cog*
  - *theta_band_ratios*
  - *alpha_band_ratios*
  - *beta_band_ratios*
  - *debug*
- **Requisiti:** *signal_type*="EEG", *alpha_cog* nelle *options*
- **Contesto:** qualunque

Alpha Prevalence Asymmetry

Misura l'asimmetria del lobo fronto-temporale attraverso l'*alpha prevalence*. In un range da -100 a + 100.

- **Identificatori:** *alpha_prevalence_asymmetry*, *apas*
- **Opzioni utilizzabili:**
  - *derivation*
  - *threshold*
  - *fs*
  - *alpha_cog*
  - *theta_band_ratios*
  - *alpha_band_ratios*
  - *beta_band_ratios*
  - *debug*
- **Requisiti:** *signal_type*="EEG", *alpha_cog* nelle *options*
- **Contesto:** qualunque

Cognitive Workload

Stima il carico cognitivo di un soggetto, con un range da -1 (a riposo) a +1 (carico cognitivo alto).

- **Identificatori:** *cognitive_workload*, *cw*
- **Opzioni utilizzabili:**
  - *zeroload*
  - *ap_fs*
  - *cw_version*
  - *debug*
- **Requisiti:** *signal_type*="alpha_prevalence", *zeroload* nelle *options*
- **Contesto:** qualunque

Attentional Effort

Stima la probabilità di una completa attenzione del soggetto, con un range da 0 (estremamente improbabile) a 1 (molto certo).

- **Identificatori:** *attentional_effort*, *ae*
- **Opzioni utilizzabili:**
  - *zeroload*
  - *ap_fs*
  - *debug*
- **Requisiti:** *signal_type*="alpha_prevalence", *zeroload* nelle *options*
- **Contesto:** qualunque

Mental Fatigue

Stima la *mental fatigue* di un soggetto, con un range da 0 (non stanco) a $+\infty$ (molto stanco).

- **Identificatori:** *mental_fatigue*, *mf*
- **Opzioni utilizzabili:**
  - cw_fs
  - debug
- **Requisiti:** *signal_type*="cognitive_workload"
- **Contesto:** qualunque

Nota importante: questo algoritmo richiede tutti i valori di cognitive_workload utilizzati durante tutta la sessione. Quindi la dimensione della finestra da utilizzare non rimane fissa ma deve aumentare.

Flow Likelihood

Stima la probabilità che un utente entri nello stato di *Flow*. Questa metrica ha significato solo calcolata mentre l'utente continua a fare la stessa cosa conosciuta e definita. Ha un range da 0(poco probabile) a +1(molto probabile).

- **Identificatori:** *flow_likelihood*, *fl*
- **Opzioni utilizzabili:**
  - *zeroload*
  - *ap_fs*
  - *optimal_band*
  - *debug*
- **Requisiti:** *signal_type*="alpha_prevalence", *zeroload* nelle *options*
- **Contesto:** set fisso di operazioni conosciute.

Mean

Calcola la media di qualunque segnale.

- **Identificatori:** mean
- **Opzioni utilizzabili:** nessuna
- **Requisiti:** nessun requisito
- **Contesto:** qualunque

3.4.2 Calibrazione

Per poter calcolare alcune metriche è necessario effettuare la calibrazione. La fase di calibrazione prevede due fasi: una ad occhi aperti e una ad occhi chiusi. La calibrazione prevede che la *Source* si adatti al flusso in corso, cosa che può richiedere del tempo, perciò è necessario gestire anche i momenti di riposo durante la registrazione.

In fig. 3.4 vengono mostrate le varie fasi, da notare il calcolo della media in cui sono richiesti diversi valori dello step precedente per essere calcolati. Le frecce indicano che ogni valore per essere calcolato richiede i risultati dello step precedente.

I valori finali sono: *alpha_cog, eyes_closed, zeroload* e *resting_asymmetry*. Una volta calcolati devono essere inviati al *Db controller* per poter essere memorizzati per l'utente specifico.

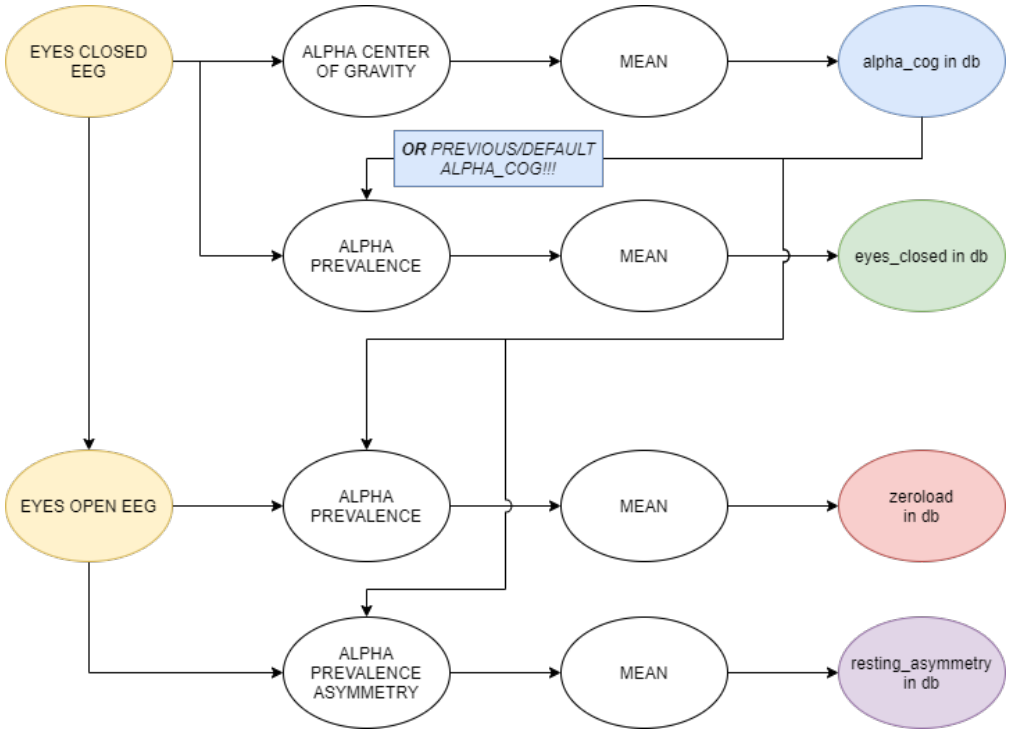

Figura 3.4: Flusso di calibrazione

3.4.3 Monitoraggio

Una volta finita la fase di calibrazione si può effettuare il monitoraggio per poter calcolare le performance mentali. In fig. 3.5 è mostrato questo flusso. Si nota che non tutti i valori calcolati nella fase di calibrazione vengono utilizzati: quei valori possono essere utilizzati per successive osservazioni manuali sulla qualità dei dati oppure da altri servizi di *Atlas* come *NeuroPerform* o *NeuroDesign*.

I dati che verranno inviati al *Db controller* sono: *Cognitive workload, Mental fatigue, Attentional Effort, Flow likelihood*. Questi dati vengono inviati in blocco, quindi è necessario attendere che siano stati tutti elaborati, inclusa la *Mental fatigue* che viene calcolata successivamente, prima di poterli inviare. Al blocco dati è richiesto anche un timestamp e un valore che indica la qualità della misurazione, chiamato *signal_quality*. La *signal_quality* viene calcolata dalla *Source* tramite la deviazione standard della registrazione: se è troppo alta vuol dire che il segnale ha molto rumore e quindi ha una qualità molto bassa.

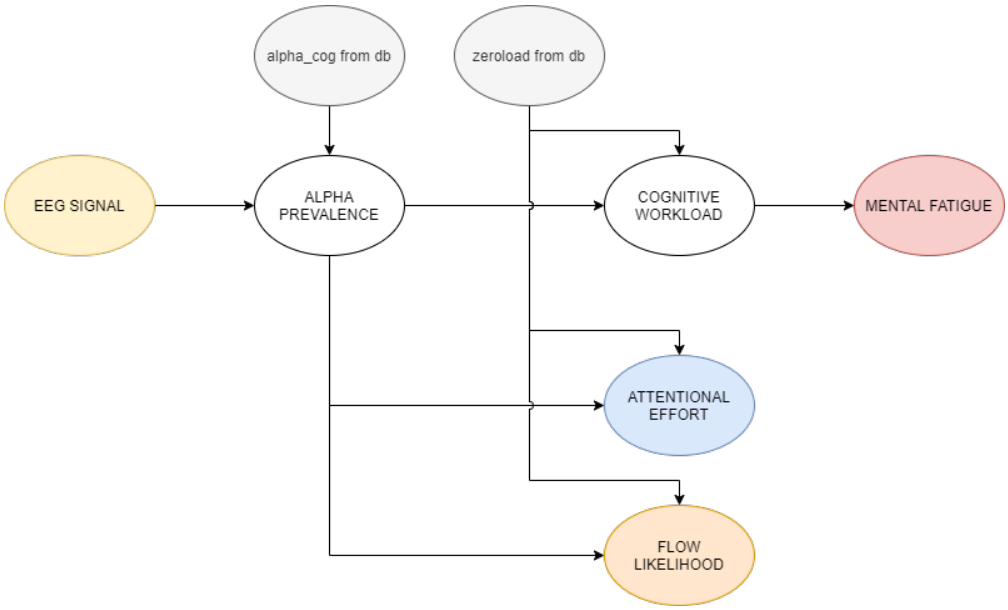

Figura 3.5: Flusso di monitoraggio

3.5 Architettura del sistema NeuroFrame

In fig. 3.6 sono mostrate tutte le entità distinte di *NeuroFrame*. In questa sezione spieghiamo l'architettura del *NeuroService* di *NeuroFrame* e come la *Source* e il *NeuroServer* interagiscono.

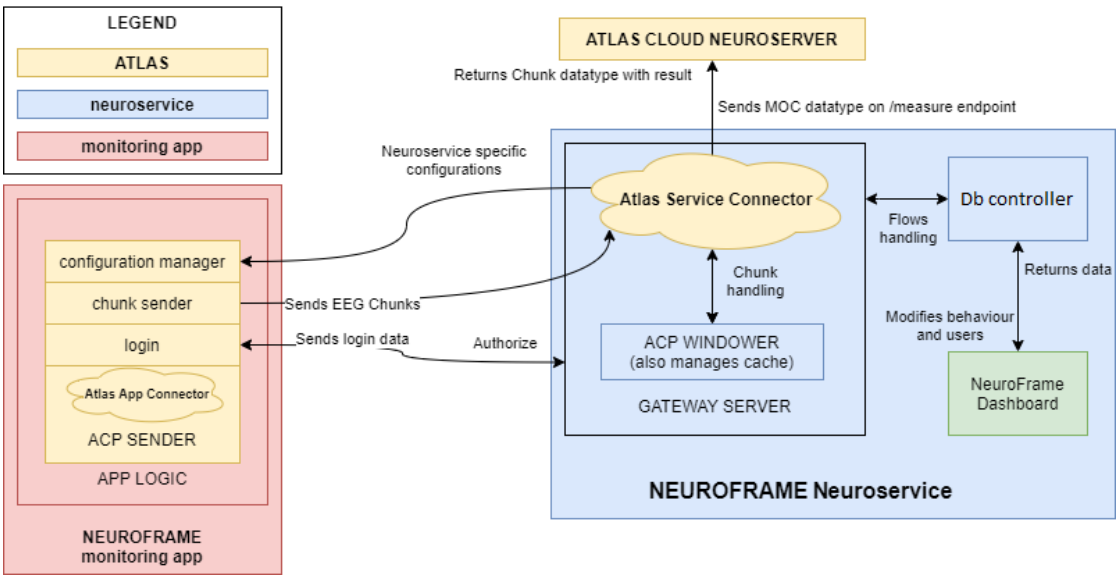

Figura 3.6: Architettura NeuroFrame

3.5.1 Db controller

Il *Db controller* è il server utilizzato per interagire con il database. Viene utilizzato sia dal *Gateway*, per il login e l'invio dei risultati, e sia dalla *Dashboard* per ottenere i valori mostrati che vengono ottenuti a polling[4]. Il *Db controller* ha lo scopo di fornire i dati per B2B[5] e gestisce gli utenti raggruppati in gruppi chiamati *Organization* con a capo un *Team Manager*. Il *Team Manager* è colui che è in grado di accedere ai risultati del soggetto, quindi ha un account speciale che gli concede questo permesso. Oltre alla gestione dei dati anagrafici dei clienti è utilizzato per salvare tutte le sessioni di registrazione, i dati di calibrazione e le metriche di *MindPulse* calcolate.

Nella corrente implementazione gestisce i dati con un database non relazionale, in particolare MongoDB. I database non relazionali sono particolarmente pratici nella programmazione agile perché permettono di cambiare il formato dei dati mantenendo la compatibilità con i formati precedenti. Un ulteriore vantaggio è il salvataggio dei dati in formato JSON, il che rende immediato salvare i messaggi dell'*ACP*.

In fig. 3.7 è schematizzata la struttura di *Db controller e dashboard*. Si nota che il *Gateway* interagisce con il *Db controller* in due modi:

[4]Verifiche cicliche sulla presenza di nuovi dati, di solito è considerato un metodo poco efficiente, perché si fanno richieste continue molte delle quali inutili.

[5]Business to Business. Si riferisce a prodotti pensati dalle aziende per le aziende.

Data elaborated: utilizzato per salvare i dati *MindPulse* elaborati in tempo reale, questa interfaccia richiede i dati salvati in blocchi.

ACP with User datatype: utilizzato per interagire con le strutture dati degli utenti.

L'interfaccia *Data elaborated* non utilizza *ACP* perché è un tipo di comunicazione interno al *NeuroService* e non è richiesto il mantenimento del protocollo in questo caso. L'interfaccia *ACP with User datatype* è visibile dall'esterno del *NeuroService*, quindi deve mantenere *ACP*.

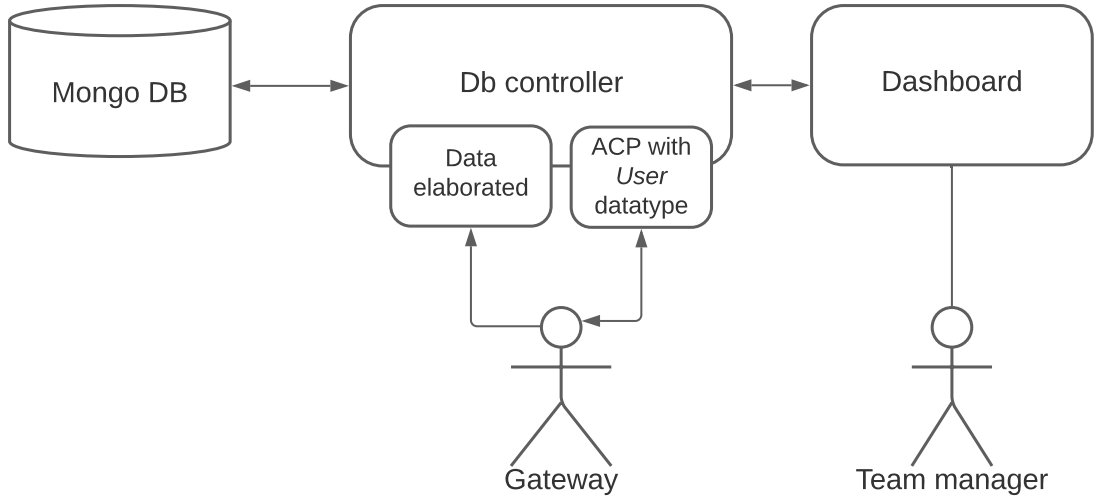

Figura 3.7: Struttura del Db controller e della dashboard

Dashboard

La *Dashboard* è il servizio che si occupa di fornire la visualizzazione dei dati in tempo reale al *Team Manager* del soggetto.

Il servizio rende disponibile una pagina web che utilizza la libreria React, che espone i risultati delle elaborazioni mostrando grafici aggiornati in tempo reale. I dati sono disponibili alla lettura solo al *team manager* del soggetto.

3.5.2 NeuroServer

Il *NeuroServer* è il componente che si occupa di effettuare le elaborazioni. La sua struttura logica è rappresentata in fig. 3.8. Le varie suite di algoritmi utilizzano un pacchetto chiamato *MindUtils* che contiene feature in comune. La struttura interna non verrà trattata in questo elaborato perché non è un argomento di tesi.

L'interfaccia verso l'esterno permette al *Gateway* il calcolo delle metriche richieste. L'interfaccia richiede in input un *MOC* e un *Chunk* con *signal_type* coerente alla metrica di *MindPulse* richiesta.

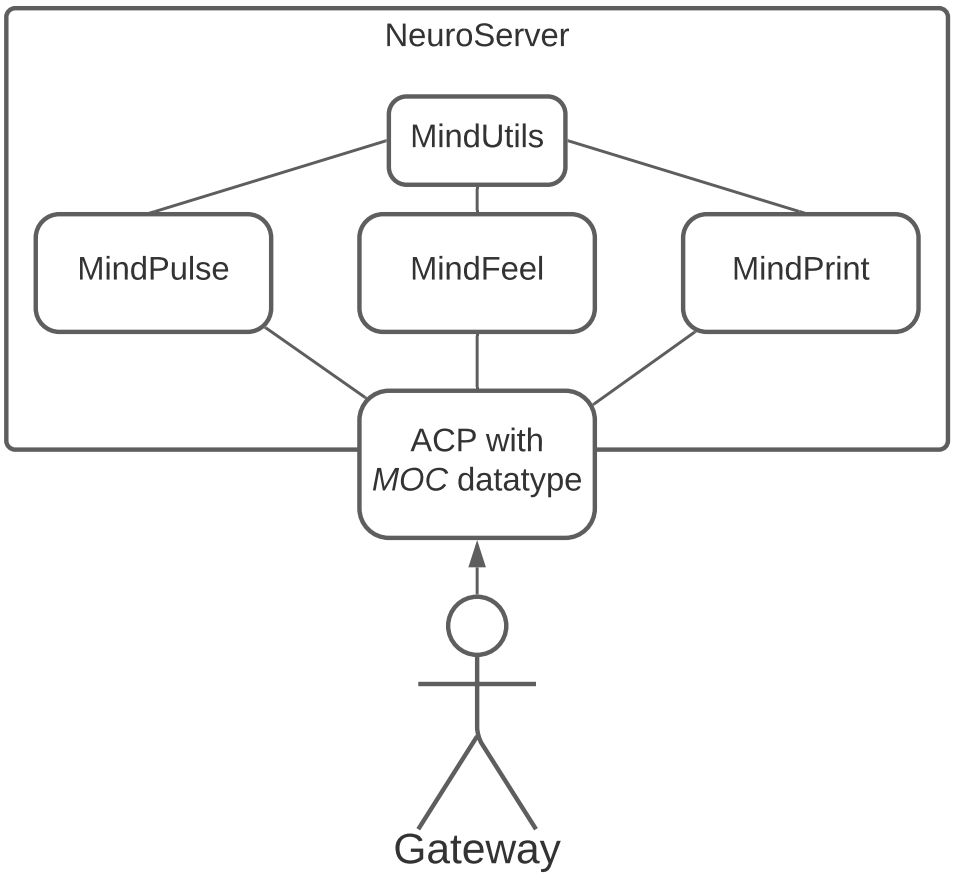

Figura 3.8: Pacchetti di algoritmi di NeuroServer

3.5.3 Atlas Service Connector e Gateway

Questo componente ha due denominazioni:

- *Connector*, utilizzato nella descrizione generale del framework Atlas.
- *Gateway*, utilizzato solo all'interno dell'implementazione *NeuroFrame* perché è concepito per essere come un gateway che collega la *Source* con il sistema distribuito.

In fig. 3.9 è schematizzata la struttura del *Gateway* mostrando anche gli attori che gli interagiscono. La struttura interna non prevede la cache, in quanto viene considerata un sistema indipendente e riutilizzabile anche da altri sistemi. Il modulo che interagisce con la *cache* è chiamato *cache_interface*, esso viene utilizzato dall'*ACP windower*, che viene a sua volta è utilizzato dal *Gateway*. La *cache* è un attore esterno utilizzato anche da altri sistemi, tuttavia viene riservato uno spazio di memoria per l'*ACP windower*, in maniera tale da impedire ad altre applicazioni di interagire con i dati salvati.

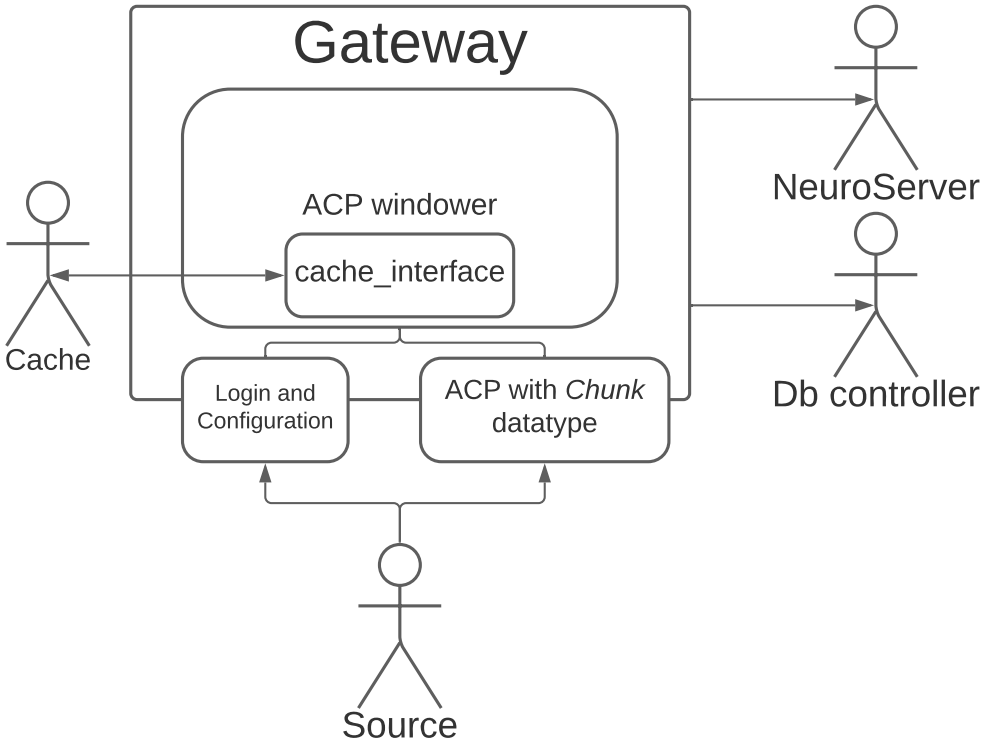

Figura 3.9: Struttura componenti gateway

L'interfaccia di comunicazione tra la *Source* e il *Gateway* utilizza il formato *Chunk* per l'invio delle registrazioni. Invece, per l'instaurazione della sessione e l'ottenimento dei dati di configurazione non sono stati definiti degli standard *ACP* da rispettare perché in quei messaggi non sono contenuti segnali neurali.

Il *Gateway*, l'*ACP windower* e la *cache_interface* sono approfonditi nel capitolo successivo dedicato al processo di sviluppo, mentre nel capitolo 5 si farà la validazione delle feature sviluppate e messe a confronto con i requisiti del progetto.

3.5.4 Endpoint Gateway

Il *Gateway* ha quattro endpoint: *configurations, login, endsession* e *chunk*

Configurations

Permette di ottenere i parametri di configurazione necessari alla *Source* per effettuare la registrazione.

- **Endpoint:** /configurations
- **Metodo HTTP:** GET
- **body:** ⟨none⟩
- **Valore esempio risposta affermativa:**

```
{
    calibration: {
        total_duration: 60,
        step_1: { duration: 30, type: "eyes closed" },
        step_2: { duration: 30, type: "eyes open" }
    },
    flag_list: {
        std: { type: "ranged" },
        marker: { type: "oneshot" }
    }
}
```

Listato 3.7: /configuration valore atteso

- **Valore esempio risposta negativa:** ⟨none⟩

Login

Permette di iniziare una sessione di registrazione.

- **Endpoint:** /login
- **Metodo HTTP:** POST
- **body:**

```
{
  vibre_id: string,
  password: string
}
```

Listato 3.8: /login body

- **Valore esempio risposta affermativa:** *User datatype* del protocollo *ACP*
- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 3.9: /login valore atteso risposta negativa

Endsession

Permette di chiudere una sessione. Chiudendo una sessione tutti i valori in cache vengono eliminati e di conseguenza la registrazione precedente non può essere ripresa. In ogni caso la sessione viene eliminata se l'utente non interagisce dopo un certo lasso di tempo.

- **Endpoint:** /endsession
- **Metodo HTTP:** POST
- **body:**

```
{
  session_id: string,
}
```

Listato 3.10: /endsession body

- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: "OK"
}
```

Listato 3.11: /endsession valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 3.12: /endsession valore atteso

Chunk

Permette di inviare i *Chunk* del protocollo *ACP* per poterli elaborare.

- **Endpoint:** /chunk
- **Metodo HTTP:** POST
- **body:** *Chunk* del protocollo *ACP*
- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: "OK"
}
```

Listato 3.13: /chunk valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 3.14: /chunk valore atteso risposta negativa

Capitolo 4

Sviluppo del progetto

In questo capitolo si mostrano le tecnologie utilizzate con i loro vantaggi e svantaggi. Successivamente verrà presentata la progettazione del *Gateway* e dell'*ACP windower*.

4.1 Tecnologie coinvolte

Il progetto è stato sviluppato interamente utilizzando Node.js come ambiente in cui eseguire codice in TypeScript. La scelta di utilizzare Node.js è stata fatta da Vibre perché permette di utilizzare un linguaggio full stack, cioè utilizzabile sia per il back-end sia per il front-end[1]. Il motivo principale per cui è stato scelto Node.js è la possibilità di riutilizzo dei pacchetti, cioè piccole unità software in grado di incapsulare funzioni anche molto complesse, anch'essi sia per il back-end che per il front-end.

4.1.1 Node.js

JavaScript è un linguaggio di programmazione inizialmente concepito per offrire un metodo per eseguire degli script sul browser, quindi solo durante la visione di una pagina web; tuttavia data la sua efficienza e il suo diffuso utilizzo è stato creato Node.js che permette di eseguire JavaScript al di fuori del browser. Grazie a Node.js JavaScript è diventato uno dei linguaggi full stack più utilizzati per le Single Page Application[2].

[1]Front-end e back-end sono i due lati del software che si occupano rispettivamente dell'interfaccia utente e delle funzioni che essa utilizza.

[2]Le single page application, o SPA, sono pagine web che interagiscono con l'utente riscrivendo la pagina stessa, cioè senza dover fare nuove richieste, rendendola simile ad un programma.[16]

Node.js utilizza l'engine V8, lo stesso che viene utilizzato dai Chromium browser[3]. V8 interpreta il codice JavaScript, quindi senza bisogno di compilazione, molto velocemente.

Le caratteristiche importanti di JavaScript, sono:

Single threaded: Il codice viene interpretato ed eseguito senza possibilità di multi thread. Sembra un malus, ma in realtà l'interpretazione resta comunque molto veloce e non bisogna gestire i problemi dovuti all'accesso concorrente alla memoria facilitando l'utilizzo di variabili condivise.

Linguaggio orientato agli oggetti: In JavaScript è possibile fare programmi utilizzando il paradigma ad oggetti. Il nome JavaScript deriva da Java ma l'unica cosa in comune è la programmazione ad oggetti che è uno standard de facto per programmi complessi.

Moduli: il supporto nativo all'import ed export di moduli rende il software molto versatile, potendo importare classi esterne al progetto stesso. Come modulo si intende tutto ciò che è contenuto in un singolo file come classi, interfacce, oggetti o metodi.

Promise: su JavaScript, e in particolar modo su Node.js, è importante l'uso di chiamate asincrone, cioè funzioni invocate senza attendere il risultato, ma invocate solo quando disponibile. Le Promise permettono di gestire i metodi asincroni facilmente.

NPM

NPM è l'acronimo di *Node packet manager*, è un gestore di pacchetti che permette di salvarli e utilizzali per il linguaggio JavaScript, ed è quello predefinito per Node.js. Consiste in un client, chiamato anch'esso NPM, e di un database online di pacchetti pubblici o privati, chiamato "NPM registry"[13]. L'azienda che possiede i diritti di NPM ha un suo registry, il quale nel 2020 ha superato i 1.3 milioni di pacchetti.

Un pacchetto è un insieme di uno o più moduli contenuti in una directory, la descrizione del pacchetto è contenuta nel file *package.json* che contiene anche le informazioni su come esportarlo. Un pacchetto può utilizzare anche altri pacchetti in cascata, essi possono quindi aumentare in maniera esponenziale e occupare molto spazio in memoria, ma di contro rendono il software molto versatile.

[3]Chromium è un browser open-source creato da Google. Dal suo codice sono stati realizzati altri browser, tra cui Google Chrome, Microsoft Edge e Opera. Chi lo utilizza viene chiamato Chromium browser, perché hanno molte caratteristiche e funzionalità in comune.

Verdaccio

È un NPM registry open source, può facilmente essere installata una sua istanza e gestita tramite docker. Permette di gestire i pacchetti NPM in modo privato e locale. I vantaggi nell'utilizzo di Verdaccio al posto del NPM registry di NPM sono diversi, tra cui il costo per la gestione dei pacchetti privati che nel caso di Verdaccio è gratis.

4.1.2 Express.js

Express.js è un framework per applicazioni web per Node.js. È un framework che fornisce strumenti minimali ma robusti per ricevere ed elaborare richieste HTTP. Il server per essere avviato necessita solamente di implementare gli endpoint richiesti e indicare la porta in cui rimanere in ascolto delle richieste in arrivo.

È utilizzato dal *Gateway* per ricevere e rispondere alle richieste provenienti dalla *Source*, utilizzando gli endpoint indicati nella sezione 3.5.4.

4.1.3 TypeScript

TypeScript è un linguaggio di programmazione sviluppato da Microsoft. Si tratta di un dialetto di JavaScript che lo estende con nuove funzioni, ma che rimangono totalmente compatibili. Un programma in TypeScript deve essere prima compilato in JavaScript per poi essere interpretato dall'engine. Quindi qualunque codice JavaScript è in grado di funzionare con TypeScript e viceversa previa compilazione.

TypeScript è stato creato per sopperire all'assenza di una tipizzazione in JavaScript, è molto utile per rendere il codice più leggibile e manutenibile. La tipizzazione aiuta a prevenire errori involontari nella fase di scrittura.

TypeScript deve essere compilato, cioè trasformato, in JavaScript. Questa operazione rende il codice poco leggibile e crea un file aggiuntivo per poterlo utilizzare come pacchetto, infatti NPM distribuisce solo JavaScript. Per rendere i pacchetti compatibili con la tipizzazione di TypeScript viene utilizzato un file aggiuntivo contenente le informazioni sulle interfacce dei moduli esposti.

Notazione Tipizzata

Qui viene posta particolare attenzione alla scrittura dei tipi perché è la caratteristica principale di TypeScript e perché è il formato in cui sono stati scritti tutti i listati di questo elaborato.

Il tipo di una variabile può essere un valore primitivo, un vettore, un alias, una interfaccia, una classe o un metodo. I tipi primitivi sono: number, bool, string, undefined e null. Undefined e null sono considerati diversi, in particolare undefined

è utilizzato quando il valore non è stato ancora assegnato, mentre null è utilizzato per rappresentare l'intenzione apposita a non avere un valore. L'interfaccia ha una struttura simile ad un oggetto JSON, tuttavia al posto del valore c'è un tipo. Il valore "*any*" sta ad indicare che non si vuole che il tipo venga gestito, quindi la variabile viene utilizzata senza nessuna verifica che le operazioni fatte siano compatibili.

```
type flags = Map<string, flag_range | flag_os | string>

interface flag_range {
    value: string | undefined,
    ts_range: [number, number]
}

interface flag_os {
    value: string | undefined,
    ts_list: number[]
}

export example_type(value: number, flag: flags){
    // code
}
```

Listato 4.1: esempio di TypeScript

4.1.4 Redis

Nel sezione 3.2.1 è stata introdotta la cache come dispositivo esterno all'*ACP windower* utilizzata per la memorizzazione delle finestre di segnale. È stato scelto di utilizzare Redis come cache.

Redis è un database open source che, a differenza di quelli convenzionali, salva i dati in memoria centrale (RAM). Redis fornisce strutture dati come stringhe, hash map, liste, set, set ordinati e altro. Sono strutture dati che sono disponibili anche su Node.js, tuttavia la scelta di utilizzo è giustificata dal mantenimento alto delle performance anche con numerosi valori salvati[6], e la possibilità di riprendere l'esecuzione di un'applicazione, a seguito di un crash imprevisto. senza perdita di dati.

Redis è utilizzabile tramite semplici comandi che interagiscono con il database. Nel listato 4.2 è mostrata una semplice interazione.

```
redis> GET "nonexisting"
(nil)
redis> SET "mykey" "Hello"
"OK"
redis> GET "mykey"
"Hello"
```

Listato 4.2: Esempio utilizzo comandi Redis

Nel caso di *ACP windower* i dati vengono salvati utilizzando una lista in cui ogni elemento contiene un sottoinsieme di lunghezza fissa di *sample*. L'approfondimento sull'algoritmo di finestratura verrà fatto in seguito. I comandi utilizzati dal *ACP windower* per l'algoritmo di finestratura sono:

RPUSH key element [element ...] inserisce, nella lista identificata da key, tutti gli elementi specificati in fondo alla lista.

LRANGE key start stop restituisce, nella lista identificata da key, tutti gli elementi, da sinistra a destra, contenuti nel range start$<-></$stop.

DEL key [key ...] rimuove tutti gli elementi specificati nelle key, siano essi delle liste o un qualunque altra struttura dati.

È possibile eseguire script in LUA[4] per interagire con le strutture dati. Gli script possono essere utilizzati per formattare i dati prima o dopo essere salvati. Nel progetto non sono stati utilizzati script in LUA perché si possono evitare gestendo le trasformazioni direttamente con Node.js.

ioredis

ioredis è un pacchetto NPM per interfacciarsi con un server Redis. Per connettersi al server è sufficiente indicare indirizzo ip, porta (di default è la 6379), password e indice del database.

Ogni comando viene subito inviato al server Redis, ciò può causare rallentamenti nel caso in cui si utilizzino molti comandi in successione, per questo ioredis implementa la pipeline che permette di inviare i comandi utilizzando una sola chiamata riducendo così il tempo di attesa totale.

4.2 Finestratura

La finestratura è l'algoritmo che deve dividere le finestre per renderle adatte al *NeuroServer*. La divisione viene fatta dividendo in celle di dimensione uguale

[4]LUA è un linguaggio di programmazione utilizzato per script ad uso generico

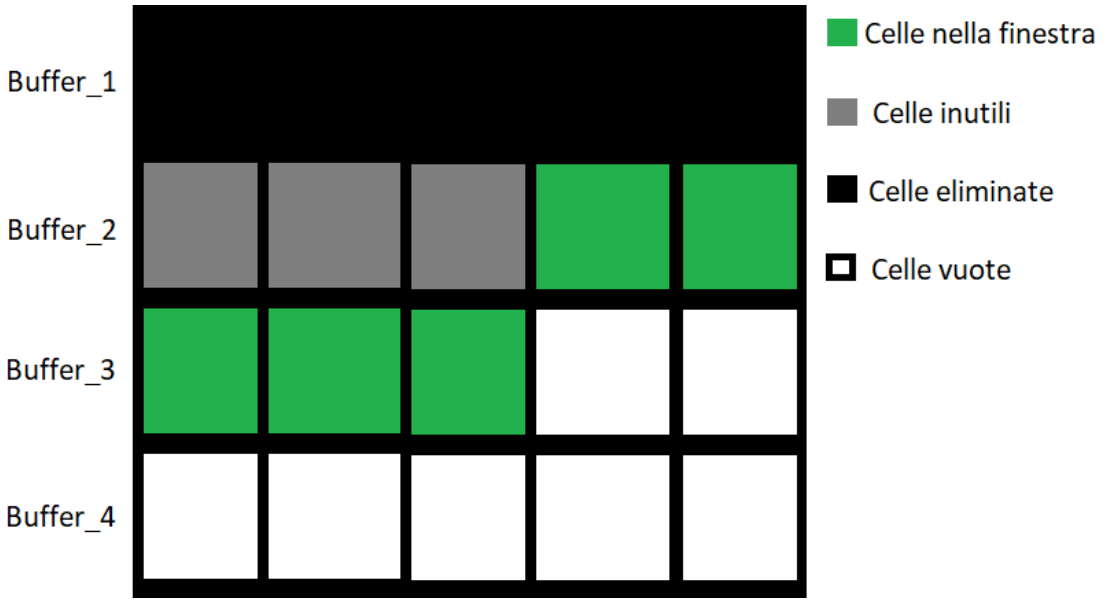

Figura 4.1: Esempio buffer per la finestratura

i *sample*. Le celle sono organizzate in una coda FIFO[5]. Una cella contiene un oggetto in formato JSON, nel caso di *signal* e *ts* si utilizza un vettore, altrimenti nel caso di *flags* si utilizza un oggetto generico. La dimensione della coda è pari al numero di celle necessarie a contenere interamente una finestra. La coda è salvata utilizzando dei buffer consecutivi, la lunghezza del buffer è tale da permettere di contenere per intero una finestra. In fig. 4.1 c'è un esempio più chiaro della lista. In nero sono indicate le liste dei buffer cancellabili perché già utilizzati e non più richiesti, in grigio sono indicate quelle non più utilizzati ma all'interno di un buffer ancora in uso, in verde quelle utilizzate nella finestra e in bianco quelle ancora non utilizzate.

Le dimensione delle celle di *sample* viene scelta come minimo comune multiplo tra lunghezza e step della finestra. La dimensione è scelta perché possano essere richieste e salvate solo celle intere.

[5]First in First Out, è un metodo per gestire le liste, in cui gli elementi da aggiungere vengono messi primi ad una coda e gli elementi da rimuovere vengo tolti in ordine di inserimento.

4.3 Architettura pacchetto ACP windower

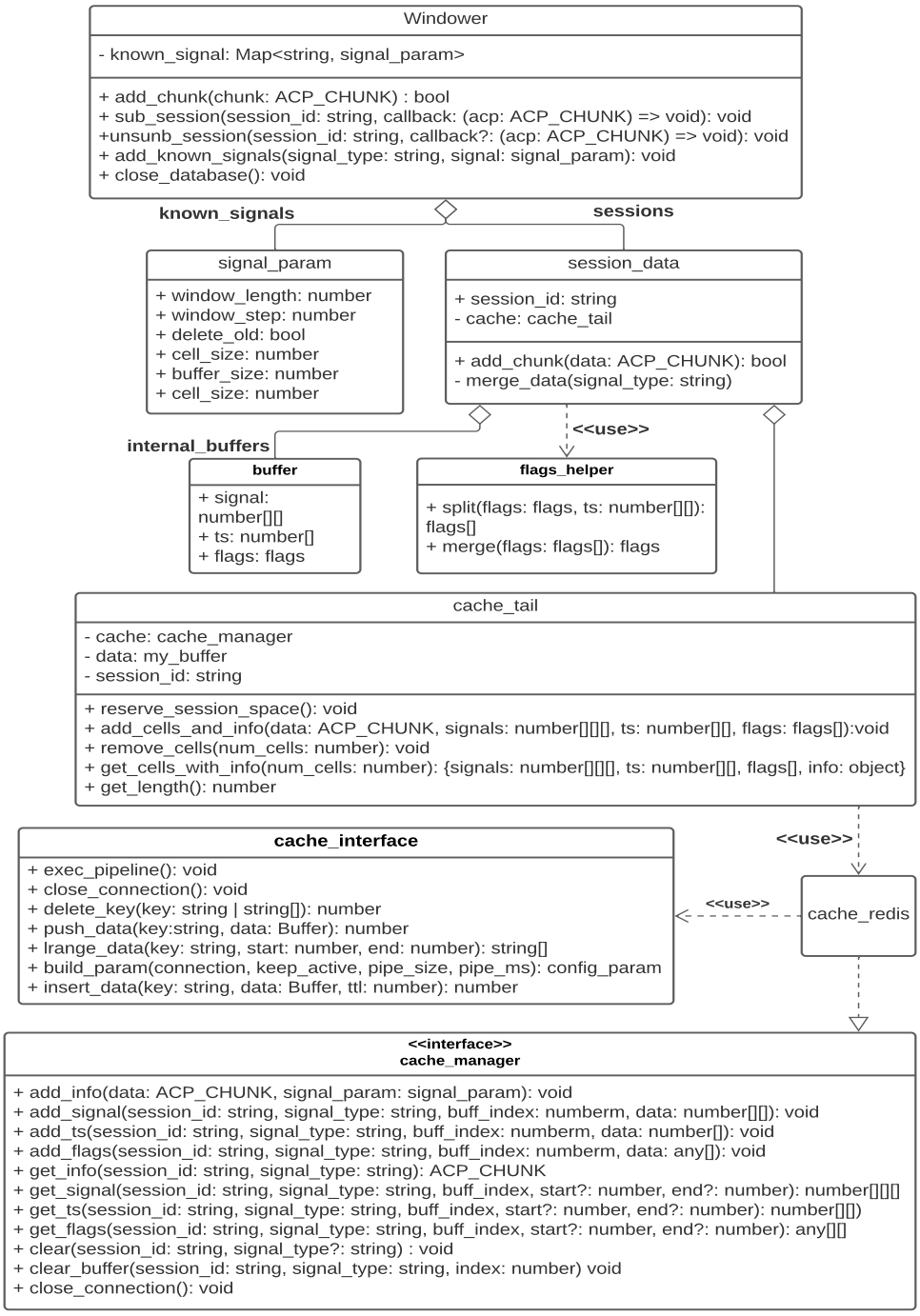

Figura 4.2: Diagramma classi completo per il Windower

Il pacchetto *ACP windower*, schematizzato in fig. 4.2, contiene le seguenti classi: *windower*, *signal_param*, *buffer*, *session_data*, *flags_helper*, *cache_tail*, *cache_manager* e *cache_redis*. Di seguito sono spiegate le caratteristiche delle varie classi.

4.3.1 windower

Questa classe è l'entrypoint del pacchetto e contiene, oltre alla classe stessa, anche le interfacce per la tipizzazione dei *Chunk* e *MOC* chiamati rispettivamente *ACP_CHUNK* e *ACP_MOC*.

Il suo utilizzo è basato sul pattern Observer:

sub_session: consente di aggiungere una funzione callback che viene chiamata quando una nuova finestra viene creata.

unsub_session: consente di rimuovere una funzione callback precedentemente sottoscritta. Se in una sessione non ci sono callback tutte le finestre vengono cancellate. Questa scelta è stata fatta per non mantenere informazioni inutili in memoria.

add_chunk: aggiunge un *Chunk* alla cache. Restituisce un valore booleano che indica se l'inserimento del *Chunk* causa la creazione di una nuova finestra.

4.3.2 signal_param

Interfaccia utilizzata per contenere le proprietà della finestra di un segnale.

window_lenght e window_step: sono rispettivamente la lunghezza e lo step della finestra richiesta in output. Lo step deve essere maggiore della dimensione del *Chunk* in input.

delete_old: parametro che indica che la finestra deve essere incrementale, cioè non si devono eliminare i vecchi dati ma solo aggiungere quelli nuovi. Viene utilizzata per la *mental fatigue*, che richiede una finestra incrementale di *cognitive workload*.

buffer_size e cell_size: sono utilizzati per l'algoritmo di finestratura, ed indicano rispettivamente il numero di celle in un buffer e il numero di *sample* in ogni cella.

4.3.3 session_data

Classe che incapsula il concetto di sessione e contiene le code utilizzate differenziandole in base al *signal_type*.

add_chunk: permette di aggiungere un *Chunk* alla sessione corrente. In questa classe vengono fatte le divisioni in celle per il *signal*, *ts* e *flags* del *Chunk* e si stabilisce se sia possibile creare una nuova finestra.

merge_data: metodo utilizzato in caso "add_chunk" renda possibile la creazione di una nuova finestra. Questo metodo viene quindi utilizzato per unire le celle salvate in un unico *Chunk*.

4.3.4 flags_helper

La divisione e l'unione dei flag viene gestita da una classe apposita. La divisione viene fatta utilizzando i timestamp di ogni cella. L'unione richiede solo i *flags* da unire.

4.3.5 buffer

Il *buffer* contiene i dati salvati localmente, sono il resto della divisione in celle del *Chunk* ricevuto. Le celle hanno una dimensione fissa e l'input potrebbe non essere interamente contenibile in una o più celle, per questo motivo devono essere salvate in locale. Questo rompe un po' il concetto di cache per evitare la perdita di dati, tuttavia quel requisito non è stato implementato perciò è stato fatto un workaround che sarà corretto in future release.

4.3.6 cache_tail

Questa classe incapsula il concetto di lista, come mostrato in fig. 4.1, contiene il concetto di buffer multipli e di lista come coda di celle. I *signal, ts* e *flags* vengono divisi su 3 liste distinte.

reserve_session_space: cancella le celle precedentemente salvate nella cache. Funzione utile per chiudere o aprire una sessione.

add_cells_and_info: metodo per salvare le celle. Viene utilizzato anche per le informazioni del *Chunk*, come *vibre_id*, *session_id* e *channels*.

get_cells_with_info: metodo per richiedere le celle precedentemente salvate.

get_length: ottiene il numero di celle salvate. Le celle salvate possono non essere sufficienti per comporre una finestra, questo metodo viene utilizzato per verificare la disponibilità nel creare la finestra.

remove_cells: rimuove un certo numero di celle dalla coda della lista.

4.3.7 cache_manager

Interfaccia per interagire con la base di dati, sia essa in locale o in remoto. L'interfaccia è stata creata per poter essere utilizzata come singleton per rendere possibile utilizzare implementazioni personalizzate che quindi non possono essere create insieme alla sessione.

add_info, add_signal, add_ts e add_flags: metodi per salvare una singola cella. Si indica l'indice del buffer, la cella viene aggiunta in coda alla lista.

get_info, get_signal, get_ts e get_flags: metodi per ottenere un range di celle di un buffer. Le celle vengono ottenute partendo dalla testa della lista.

clear e clear_buffer: cancella tutti i dati di un intero segnale nel caso di "clear", o un buffer specifico nel caso di "clear_buffer".

close_connection: chiude la connessione con il database. Questa operazione viene fatta per compiere una chiusura pulita del server senza lasciare aperte eventuali connessioni al database.

4.3.8 cache_redis

Implementazione di "cache_manager" con database Redis. I comandi vengono utilizzati con il pacchetto *cache_interface*.

4.3.9 Cache_interface

Pacchetto NPM sviluppato appositamente per questo progetto, tuttavia rimane disponibile per altri utilizzi. Il pacchetto utilizza ioredis per gestire la pipeline e i vari metodi, come indicato nella sezione 4.1.4.

La funzione di maggior rilievo supportata è la gestione automatica della pipeline, che permette di ottimizzare la gestione delle richieste. I comandi sono mantenuti in memoria fino a che non raggiungono una certa soglia o restano in coda per un certo lasso di tempo.

exec_pipeline: invia la pipeline al server Redis, tutti i comandi contenuti nella pipeline vengono inviati.

close_connection: chiude la connessione al server. La connessione viene gestita in due modi: il primo metodo prevede l'apertura e chiusura per ogni comando, facendo così la pipeline non viene utilizzata e ogni comando viene inviato singolarmente; il secondo metodo mantiene aperta la connessione, in questo

modo rimane aperta a tempo indeterminato. Questo comando viene applicato automaticamente nel primo caso e deve essere utilizzato manualmente nel secondo.

delete_key: cancella una o più chiavi nel database e restituisce il numero di valori eliminati.

push_data: aggiunge un elemento a destra della lista identificata dalla key. Se la lista non esiste viene creata.

lrange_data: restituisce gli elementi della lista identificata da key e contenuti nel range start$<->$end, limite chiuso.

build_param: metodo per strutturare più brevemente i parametri per il costruttore. Richiede i seguenti parametri:

connection: parametro definito da ioredis che contiene tutte le impostazioni per la connessione al server Redis.

keep_connection: flag che indica l'apertura e chiusura automatica della connessione al server Redis.

pipeline_size: capienza della pipeline. Quando raggiunge la capienza massima i comandi vengono inviati al server Redis.

pipeline_max_delay: imposta il numero di millisecondi in cui i comandi rimangono nella pipeline se non raggiunge la capienza massima.

altro: sono state implementate altre funzioni che aggiungono il concetto di pipeline automatica alle funzioni già presenti nel pacchetto NPM; in particolare, *insert_data*, *get_data* e *search_key* permettono di salvare, richiedere e ricercare un valore.

4.4 Architettura Server Gateway

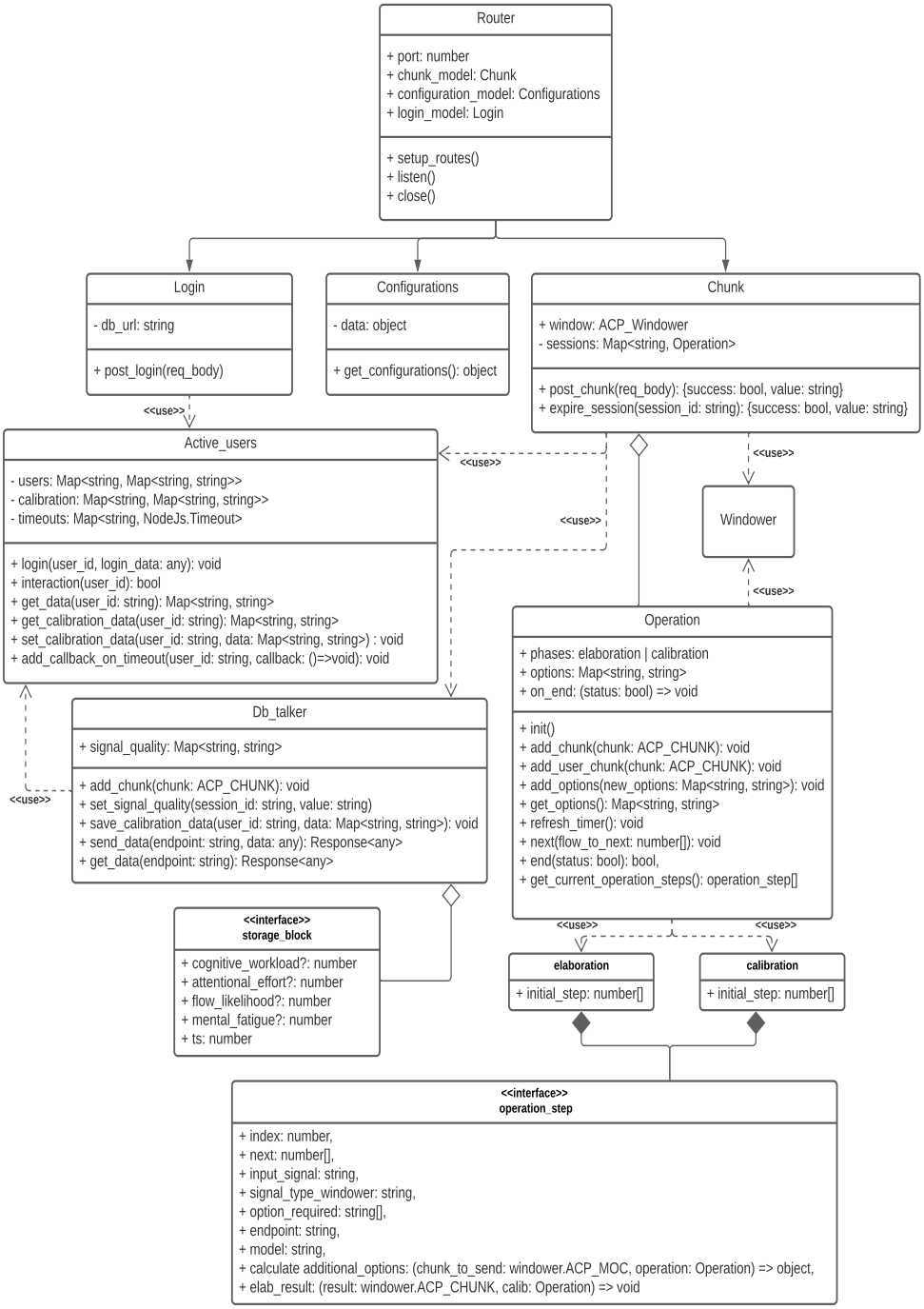

Figura 4.3: Diagramma classi completo per il Gateway

4.4.1 Router

Classe che gestisce gli endpoint del server, tramite essa è possibile accedere alle varie funzioni implementate, gli endpoint sono stati già spiegati nella sezione 3.5.4. Contiene tre metodi con le seguenti funzioni:

setup_router: configura i vari endpoint e indica le funzioni supportate da server.

listen: avvia l'esecuzione del server.

close: chiude le connessioni mantenute aperte e chiude il server. Viene utilizzato solo per fare la chiusura pulita durante i test altrimenti la connessione a Redis non verrebbe chiusa correttamente.

4.4.2 Login

Questa classe contiene solo un metodo che viene utilizzato per gestire la login. Per gestire l'accesso dell'utente le credenziali sono ritrasmesse al *Db controller* il quale, in caso siano corrette, restituisce i dati dell'utente comprendendo anche i valori di calibrazione. Quando l'utente viene verificato si può procedere alla creazione della sessione di login.

La sessione di login è diversa da quella di registrazione, infatti ha una durata maggiore e gestisce il *session_id* dei *Chunk*.

4.4.3 Configurations

Classe che espone i dati di configurazione. I dati di configurazione sono utilizzati per fornire alla *Source* i parametri per gestire il contesto in cui effettuare la registrazione.
I parametri contenuti sono:

calibration: oggetto che contiene i parametri per la calibrazione. Contiene i seguenti parametri:

- **total_duration:** durata totale della calibrazione espressa in secondi.
- **steps:** nome e durata in secondi di ogni flusso di calibrazione. Contiene due valori che indicano la durata della calibrazione a occhi aperti e ad occhi chiusi.

flag_list: lista di nome e tipo dei flag interpretati dal *Gateway* e che la *Source* può utilizzare.

chunk_length: lunghezza in secondi della durata dei *Chunk* che la *Source* deve inviare.

4.4.4 Chunk

Questa classe gestisce la sessione di registrazione e quindi i *Chunk* ricevuti dalle *Source*. Quando un *Chunk* viene ricevuto vengono fatti i seguenti passaggi:

1. verifica del *vibre_id* del *Chunk* che deve essere collegato ad una sessione di login precedentemente creata. Se è presente aggiorna il timer per non far scadere la sessione di login e di registrazione.

2. verifica se il *Chunk* ha il flag "ignore". Questo flag è utilizzato per mantenere le sessioni, di login e di registrazione, aperte. Utile alla *Source* per poter effettuare delle pause o per prepararsi all'inizio della calibrazione o del monitoraggio.

3. salvataggio del *Chunk* nel *Db controller*. Questa operazione viene fatta per avere sempre un backup dei dati per prevenirne la perdita accidentale.

4. gestione della sessione. Se la sessione non esiste e il *Chunk* contiene il flag "flag_calib" allora viene creata la sessione di calibrazione, altrimenti se non lo contiene viene creata la sessione di monitoraggio. Dopo aver creato la sessione, o in caso sia già presente, il *Chunk* viene delegato alla sessione, che viene incapsulata in un oggetto di tipo *Operation*.

La funzione "expire_session" viene utilizzata dall'endpoint *endsession*. Cancella i dati nel *ACP windower* e l'oggetto *Operation*.

4.4.5 Active_users

Classe singleton che gestisce la sessione di login.

login: funzione che crea la sessione di login. L'oggetto "login_data" contiene i dati dell'utente e i suoi valori di calibrazione che vengono memorizzati in locale. Quando viene creata viene anche avviato il timer per il logout automatico.

interaction: verifica che un utente abbia creato la sessione di login e azzera il timer in caso affermativo.

get_data: metodo per ottenere i dati dell'utente, come *organization_id* e *team_id*.

get_calibration_data e set_calibration_data: metodi per interagire con i dati di calibrazione di un soggetto. Queste operazioni non causano l'aggiornamento delle informazioni nel *Db controller* ma modificano solo i dati in locale.

add_timeout_callback: metodo per aggiungere un callback che viene invocato quando la sessione di login di un utente termina a seguito dello scadere del timer.

4.4.6 Windower

Questa classe è il pacchetto *ACP windower*.

4.4.7 Db_talker

Questa è la classe che interagisce con il *Db controller*. Viene utilizzata per aggiornare e richiedere i dati di calibrazione e salvare i risultati della sessione di registrazione.

add_chunk: metodo per compilare il blocco di dati contenente i dati finali delle elaborazioni di monitoraggio. Vedi sezione 3.4.3 per i dati contenuti nel blocco.

set_signal_quality: metodo per aggiungere l'informazione riguardante la qualità della misurazione. È una stringa che può assumere i valori "BAD", "GOOD", "AVERAGE".

save_calibration_data: metodo utilizzato per inviare i dati di calibrazione al *Db controller*. Quando invocato provoca l'aggiornamento anche dei dati salvati in *Active_users*.

send_data e get_data: metodi generici per interagire con il *Db controller*. Utilizzati soprattutto per intercettare i messaggi durante i test, vedi sezione 5.5.1.

4.4.8 storage_block

Struttura dati per contenere le informazioni da inviare al blocco dati. Non contiene i dati sulla qualità del segnale, perché essi vengono salvati a parte, in quanto sono ottenuti direttamente dalla *Source*. Vedi sezione 3.4.3 per la spiegazione del blocco dati.

4.4.9 Operation

Questa classe è utilizzata per incapsulare il concetto di sessione di registrazione. È strutturata per seguire un flusso, che può essere di calibrazione o monitoraggio. Una sessione di registrazione deve prevedere: salvare i *Chunk* nel *ACP windower*;

inviare le finestre pronte per essere elaborate al *NeuroServer*; gestire le finestre di risposta ricevute dal *NeuroServer*.

Questa classe gestisce sia il flusso di calibrazione, sia quello di monitoraggio ma essi hanno delle caratteristiche diverse: il flusso di calibrazione prevede due momenti distinti, ad occhi chiusi e ad occhi aperti; il flusso di monitoraggio deve invece salvare i risultati nel *Db_talker*. Per poter gestire ambe le funzionalità è necessario prevedere degli step intermedi che possono essere attivati o disattivati quando richiesto e delle funzioni che salvino il risultato o che lo reindirizzino all'*ACP windower*.

Con step si intende la logica per l'elaborazione di una metrica *MindPulse*, o di una qualunque altra suite, da inviare al *NeuroServer*. Lo step è salvato in un oggetto contenente:

- un tipo di finestra, identificata da lunghezza, step e *signal_type*.
- il *model* per indicare al *NeuroServer* quale algoritmo utilizzare.
- le opzioni richieste per elaborare l'algoritmo (vedi sezione 3.4.1).
- la funzione che elabora il risultato dello step.
- prossimi step da attivare nel caso in cui questo termini.

In una *Operation* ci possono essere numerosi step attivi contemporaneamente e ci può essere l'evenienza in cui essi richiedano di utilizzare in input lo stesso *signal_type*. Ciò porterebbe ad utilizzare la stesse caratteristiche della finestra. Questo perché l'*ACP windower* riconosce le dimensioni delle finestre in base al *signal_type*. Per ovviare a questo limite è stato reso possibile salvare i *Chunk* nel *ACP windower* con un alias, permettendo così di avere stesso *signal_type* ma con dimensioni delle finestre diverse.

Operation gestisce gli step utilizzando i seguenti metodi:

add_chunk: metodo per elaborare un *Chunk* ricevuto dal *NeuroServer*. Sono selezionati gli step attivi che richiedono l'utilizzo di finestre con *signal_type* del *Chunk*. Si assegnino gli alias degli step selezionati e si salvano i *Chunk* nel *ACP windower*.

add_user_chunk: metodo per elaborare i *Chunk* ricevuti dall'utente. La differenza con "add_chunk" è che in questo casi salva la qualità del segnale nel *Db_talker* e si resetta il timer della sessione di registrazione.

add_options e get_options: metodi per interagire con le *options* della sessione di registrazione. Le *options* possono essere i dati di calibrazione o altri parametri aggiuntivi da indicare nelle *options* del *MOC*.

refresh_timer: metodo per forzare il reset del timer della sessione di registrazione.

next: metodo per far progredire gli step. Come parametro si indicano gli step da disattivare. Quando uno step viene disattivato può indicare i successori, cioè quelli che diverranno attivi.

end: metodo per indicare la fine con successo della sessione. È utilizzata per la fine prevista della calibrazione. La sessione di monitoraggio termina solo quando la *Source* invia il comando di chiusura all'endpoint */endsession*.

get_current_operation_steps: ottiene le operazioni attualmente attive.

init: inizializza l'*ACP windower*. Effettua la "sub_session" per poter ricevere le finestre pronte per l'elaborazione. La funzione di callback invia la nuova finestra al *NeuroServer* e ne elabora la risposta.

4.4.10 operation_step

Interfaccia per la struttura degli step di un flusso di registrazione.

index: indice dello step. Deve essere univoco per il flusso utilizzato.

next: step da attivare quando questo viene portato a termine. Quando uno step viene portato a termine viene disattivato e non verranno più inviate le sue finestre al *NeuroServer*.

input_type: *signal_type* richiesto dallo step, cioè dal *NeuroServer* per poter eseguire l'elaborazione desiderata.

signal_type_windower: alias da utilizzare per poter memorizzare nell'*ACP windower* stessi segnali ma con finestre diverse.

option_required: vettore con i nomi delle opzioni richieste per l'elaborazione. Vedi sezione 3.4.1 per le opzioni delle metriche.

endpoint: endpoint da utilizzare per l'elaborazione. Nella corrente implementazione è sempre con il valore "/measure" (vedi sezione 7.1.2).

calculate_additional_options: funzione per elaborare *options* aggiuntive per la metrica richiesta.

elab_result: funzione per elaborare il segnale ricevuto dal *NeuroServer*. Solitamente viene utilizzata per aggiungere il *Chunk* alla *Operation* utilizzando il metodo *add_chunk*, oppure per aggiungere il risultato al blocco dati di *Db_talker*.

La struttura non è da creare per ogni sessione, ma è una costante. Le funzioni hanno un contesto che gli viene passato come parametro.

La loro struttura è stata scelta per poter facilmente aggiungere nuovi step; in seguito, si mostreranno le scelte fatte per poter estendere il *Gateway* ad altri scenari o a nuovi flussi nella sezione 4.6.

Esempio flusso di step Nel listato 4.3 è mostrata la struttura dati contenente gli *operation_step* per il flusso di calibrazione. È solo una porzione degli step utilizzati per la calibrazione, in particolare, è mostrato lo step per calcolare la *alpha_prevalence*. La struttura viene salvata in due oggetti, un vettore e una Map: la prima serve per indicare gli step attivi da subito, e la seconda è l'oggetto contenente i vari step.

```
let elaboration_phases: [number[],
    Map<number, Operation.operation_step>] =
[[1, 2, 3, 4, 5],
{
    1: {
        index: 1,
        next: [],
        input_signal: "eeg",
        signal_type_windower: "eeg",
        options_required: ["alpha_cog"],
        endpoint: "/measure",
        model: "alpha_prevalence",
        elab_result: async (result: Windower.ACP_CHUNK,
            calib: Operation.Operation) => {
            await calib.add_chunk(result)
        },
        calculate_additional_options: (chunk: Windower.ACP_MOC,
            calib: Operation.Operation) => {
            if (eeg_threshold) {
                return { threshold: eeg_threshold }
            }
            return {}
        }},
    ...
}]
```

Listato 4.3: Esempio di step per l'elaborazione

4.4.11 Elaboration e calibration

Questi sono i file che contengono la struttura dati che rappresentano rispettivamente il flusso di monitoraggio e di calibrazione.

4.4.12 Constant

Questo modulo è utilizzato per importare tutte le variabili d'ambiente utilizzate per configurare:

- i *signal_param* utilizzati per tutte le finestre
- gli indirizzi ip e porte per: *Db controller, NeuroServer e Redis*
- configurazione per Redis che comprende canale e password
- durata per sessione di login e di registrazione
- durata dei flussi di calibrazione, vengono utilizzati da *Configurations*
- lunghezza del *Chunk* che deve inviare la *Source*

4.5 Requisiti implementati

I requisiti più importanti sono stati tutti implementati con successo, la loro verifica è mostrata nel capitolo 5. I requisiti che non sono stati sviluppati sono quelli ritenuti opzionali o che aggiungono robustezza al software, qualità che non impedisce il funzionamento del software in condizioni controllate. Di seguito è mostrato il loro elenco con il motivo della non implementazione e una possibile via da percorrere per soddisare il requisito.

4.5.1 Gateway

Dovrebbe essere in grado mantenere i dati degli utenti anche a seguito di una chiusura inaspettata. Questo è un requisito *Should Have*, e per essere implementato deve essere possibile rilevare il crash e salvare dati degli utenti in una memoria esterna. Il rilevamento del crash può essere fatto utilizzando un try-catch che comprende tutta l'esecuzione del server, mentre i dati degli utenti devono essere mantenuti sempre aggiornati in una memoria esterna, a questo può essere utilizzato *cache_interface* per salvare e leggere i dati su server Redis.

Dovrebbe validare i dati in ingresso secondo il protocollo ACP Questo requisito non ha particolari complicazioni, non è stato implementato perché è una feature opzionale che non modifica il comportamento in un ambiente controllato, cioè in cui si assume la fedeltà dei client; inoltre, attivando questa validazione si aggiunge un passaggio intermedio nella elaborazione del segnale in arrivo che può causare rallentamenti. Un altro motivo per la non validazione è che il *vibre_id* viene comunque verificato quando si riceve un *Chunk* in ingresso, perciò una minima protezione dai client indesiderati è comunque presente.

4.5.2 ACP windower

I dati salvati devono essere temporizzati con un TTL Questo requisito chiede di cancellare i dati dopo il loro utilizzo perché non più utili. Può facilmente essere implementato utilizzando una funzionalità di Redis, che permette di cancellare automaticamente i valori dopo un certo lasso di tempo. È stato implementato ma non con un TTL, perché la cancellazione viene fatta in automatico sui buffer non più utilizzati. Affidare questa funzione a Redis potrebbe portare alla cancellazione involontaria dei dati nel caso in cui la *Source* sia in ritardo nella consegna dei *Chunk*.

Dovrebbe validare i dati in ingresso secondo il protocollo ACP Questo requisito non è stato implementato perché è già presente il requisito di validazione nel *Gateway*.

4.6 Meccanismi di estensibilità Gateway

La classe *Operation* è stata creata per rendere semplice l'estensione a nuovi step o flussi.

Per poter aggiungere una nuova metrica si devono eseguire i seguenti passaggi:

1. Creare il *signal_param* per indicare i parametri della finestra richiesta, è quindi necessario indicare lunghezza, step e *signal_type*. Questa operazione deve essere fatta nel modulo *Constant*, nel quale risiedono tutti gli altri *signal_param* per i flussi già esistenti.

2. Creare e aggiungere lo step nel flusso di registrazione da utilizzare. Questa operazione deve essere fatta in *elaboration* o *calibration*. La struttura dello step è stata già mostrata nel sezione 4.4.10.

3. Per poter salvare il nuovo valore deve essere aggiunto al blocco dati da inviare al *Db controller*. Per fare ciò bisogna aggiungere la nuova metrica allo

storage_block, e incrementare la variabile "number_of_values", appositamente creata per poter incrementare la dimensione del blocco da inviare.

Questi passaggi sono ritenuti sufficienti per soddisfare il requisito "*Dovrebbe essere semplice aggiornare i flussi per eseguire calibrazione e misurazione*".

Capitolo 5

Validazione del software creato

La validazione consiste nel prevedere il flusso di esecuzione che si deve verificare in determinate circostanze per poi confrontarlo con la reale esecuzione e se non corrispondono o ci sono errori imprevisti significa che il requisito non è rispettato.

La validazione può essere fatta sia sui piccoli componenti sia sul funzionamento generale. Verificare il funzionamento sui componenti piccoli aiuta a trovare e correggere errori su funzioni molto specifiche. La verifica del funzionamento generale aiuta a trovare errori createsi nella interazione tra i moduli, ciò aiuta a trovare errori non rilevabili altrimenti, ma li rende più difficili da risolvere perché il problema può provenire da molte fonti diverse. Per questo progetto è stata fatta una validazione a salire più generale, cioè partendo da *cache_interface* analizzando in dettaglio le sue funzioni, salendo al *ACP windower* per problemi di media entità fino ad arrivare al *Gateway* per il funzionamento generale.

5.1 Jest

Jest è un framework che permette di effettuare test in JavaScript. Data la sua grande popolarità hanno aggiunto il supporto a Node Js e TypeScript, rendendolo così il framework di test più utilizzato per questo linguaggio. Jest è facile, veloce e intuitivo ma al contempo offre diverse strategie per analizzare e talvolta modificare l'esecuzione del software.

I test sono divisi in *suite*, cioè gruppi di test con un obbiettivo in comune. Le *suite* vengono create con il comando "describe". All'interno di una "describe" sono contenuti i singoli test, che sono funzioni che confrontano il risultato atteso di un semplice script con quello effettivamente restituito. All'interno di una "describe" ci possono essere "beforeAll" e "afterAll", sono funzioni utilizzate per poter inizializzare e ripristinare correttamente le risorse utilizzate durante il test come, per esempio, database esterni o file modificati.

Un "test" è una funzione che esamina un singola interazione con l'oggetto in esame, ma talvolta se il caso lo richiede possono essere verficati anche più interazioni esaminando anche i risultati intermedi. Per verificare il risultato c'è la funzione "expect" che prende in input un valore e lo confronta con un altro. I metodi di confronto sono molteplici come, ad esempio, "toEqual" che verifica che due oggetti abbiano lo stesso valore o "toBeGreaterThan" che verifica che un numero sia maggiore di un altro. Nel listato 5.1 c'è un esempio di test suite.

```
describe("test suite for flags_helper", ()=>{
    beforeAll(async () => { ... })
    afterAll(async () => { ... })

    test("should split simple flags", async()=>{
        ...
        expect(result).toEqual(expected_result)
    })
    ...
})
```

Listato 5.1: Esempio Jest

Una funzione molto comoda è la funzione simulata, chiamata *mock function*, che permette di verificare il corretto utilizzo di moduli senza utilizzare l'effettiva implementazione. La *mock function* permette di osservare se i parametri di una funzione sono corretti e opzionalmente di sovrascrivere la funzione con un'altra finta. Il vantaggio è quello di poter verificare solo una entità alla volta senza avviare procedure collaterali. Viene utilizzato per verificare la corretta gestione dei dati senza salvarli o richiederli al database. Nel listato 5.2 è mostrato un esempio di utilizzo, in cui una funzione simulata viene utilizzata per non cercare in database remoto l'utente attivo permettendo di verificare solo la chiamata della funzione e non la funzione stessa.

```
let spy_function = jest.spyOn(object, "function")
    .mockImplementation(() => true)
...
test("should do a login", async() => {
    ...
    expect(spy_function).toHaveBeenCalledWith(parameter)
})
```

Listato 5.2: Esempio Jest

5.2 Validazione cache_interface

La feature più importante di *cache_interface* è la gestione automatica della pipeline ed è quella più critica, cioè quella che può causare un malfunzionamento generale.

5.2.1 Pipeline automatica

La pipeline automatica deve essere gestita in ogni comando supportato e deve attivarsi solo quando si supera una soglia di comandi contenuti o fino allo scadere di un timer. Non dovrebbe causare nessun mutamento nel funzionamento ma dovrebbe aumentare solo l'efficienza. Per poter verificare il funzionamento è sufficiente valutare la riduzione del tempo nell'esecuzione di più comandi in successione. Il test deve eseguire molti comandi perché utilizzare solo un comando alla volta ridurrebbe le performance, non riuscendo a riempire la pipeline e quindi dovendo attendere il tempo aggiuntivo del timer.

```
let redis_interface = new cache_interface(connection)
let start_1 = Date.now()
for (let i = 0; i < 10; i++) {
    await redis_interface.insert_data("test_pipeline",
    Buffer.from("mock_data"), 1000)
}
await redis_interface.close_connection()
let end_1 = Date.now()

redis_interface = new cache_interface(cache_interface
    .build_param(connection, true, 10, 1000))

let start_2 = Date.now()
for (let i = 0; i < 10; i++) {
    redis_interface.insert_data("test_pipeline",
    Buffer.from("mock_data"), 1000)
}
await redis_interface.close_connection()
let end_2 = Date.now()
expect(end_1 - start_1).toBeGreaterThan(end_2 - start_2)
```

Listato 5.3: Validazione della pipeline

NB. Questo test fornisce anche un valore indicativo sull'incremento delle performance. Svolgendo i test, con un RTT di 50 millisecondi, si ottiene un incremento

delle performance di 12 volte se si riempie il buffer, altrimenti se si attende l'intervento del timer, impostato a 100 millisecondi, si ottiene un incremento di 8.7 volte.

5.2.2 exec_pipeline

Questo comando è utilizzato per svuotare immediatamente la pipeline, quindi senza dover attenderne il riempimento o lo scadere del timer. Per verificarne il funzionamento è necessario confrontare il tempo trascorso dall'esecuzione del comando alla sua risposta con e senza "exec_pipeline". Utilizzando "exec_pipeline" il tempo dovrebbe ridursi. Il test è mostrato nel listato 5.4.

```
let redis_interface = new cache_interface(
    cache_interface.build_param(connection_options, true, 2, 1000))
let start = Date.now()
await redis_interface.insert_data("test_exec_pipeline",
    Buffer.from("mock_data"), 2000)
let half = Date.now()
redis_interface.insert_data("test_exec_pipeline",
    Buffer.from("mock_data"), 2000)
await redis_interface.exec_pipeline()
expect(half - start).toBeGreaterThan(Date.now() - half)
```

Listato 5.4: Validazione exec_pipeline

5.2.3 close_connection

Questo comando prevede, oltre alla chiusura della connessione, lo svuotamento della pipeline per non perdere i comandi in attesa di essere inviati. Il funzionamento non può essere valutato dall'esterno perché in caso la connessione sia stata chiusa e venga utilizzato un comando, essa viene riaperta in automatico; perciò l'unica possibilità è verificare che dopo aver svuotato la pipeline il client, cioè l'oggetto *ioredis* venga cancellato. Il test viene fatto come nel listato 5.5, e prevede l'inserimento di un dato, la verifica dell'apertura della connessione, la chiusura e la verifica della effettiva cancellazione. "redis_interface" è di tipo "any" perché altrimenti TypeScript impedirebbe di accedere alla proprietà "config" perché rivata.

```
let redis_interface: any = new cache_interface(
    cache_interface.build_param(connection_options, true))
expect(redis_interface.config.client).toBeFalsy()
await redis_interface.insert_data(key,
    Buffer.from("mock_data"), 2000)
expect(redis_interface.config.client).toBeTruthy()
await redis_interface.close_connection()
expect(redis_interface.config.client).toBeFalsy()
```

Listato 5.5: Validazione close_connection

5.2.4 insert_data e delete_key

Questo comando per essere verificato richiede l'utilizzo di un altro comando, perciò è possibile verificare il funzionamento di entrambi le funzioni. Semplicemente si salva una chiave, si cancella e se l'operazione è avvenuta con successo non dovrebbe essere possibile ottenere il valore precedentemente salvato. Il test è mostrato nel listato 5.6

```
let redis_interface = new cache_interface(connection_options)
let data = Buffer.from("mock_data")
await redis_interface.insert_data("test_delete_key", data)
await redis_interface.delete_key("test_delete_key")
expect(await redis_interface.get_data(
    "test_delete_key")).not.toEqual(data)
```

Listato 5.6: Validazione insert_data e delete_key

5.2.5 push_data e lrange_data

Questi comandi vengono verificati insieme: prima si esegue *push_data* con una coppia di valori, poi si verifica se la coppia sia salvata nel giusto ordine utilizzando *lrange_data*. Il test è mostrato nel listato 5.7

```
let redis_interface = new cache_interface(connection)
await redis_interface.push_data("test_list", Buffer.from("0"))
await redis_interface.push_data("test_list", Buffer.from("1"))

let result = await redis_interface.lrange_data("test_list", 0, 1)
expect(result).toEqual(["0", "1"])
```

Listato 5.7: Validazione push_data e lrange_data

5.3 Validazione ACP Windower

Per la validazione del *ACP windower* è stato scelto un compromesso tra test sulle piccole funzioni e sul funzionamento generale. La validazione è stata divisa in tre suite:

flags_helper Si verifica la corretta divisione e unione del campo flag del *Chunk*, i test sono fatti sulle singole funzioni.

cache_tail Si verifica la corretta gestione della coda su più buffer, implicitamente si verificano anche le funzioni di *cache_redis*.

windower Si verifica il funzionamento generale. Se i test delle altre suite falliscono anche questa dovrebbe fallire, quindi prima si devono correggere le altre prima di intervenire su questa.

Nei test di *cache_tail* e *windower* si utilizza una classe chiamata "test_util" che rende agile la creazione di *signal_param* e *Chunk*. Per verificare correttamente il *Chunk* i *sample* sono generati partendo da zero e incrementando il valore. Con *test_util* è possibile utilizzare dei parametri aggiuntivi come: *session_id*, *signal_type*, numero di canali, lunghezza del *sample* e valore iniziale dei *sample*. Questi parametri verranno utilizzati per poter fare i test senza doversi preoccupare di creare il *Chunk* ogni volta.

5.3.1 flags_helper

Questa suite contiene solo 2 test che verificano "split" e "merge". I test coprono diverse casistiche, soprattutto con i valori -1 che indicano che un estremo del range non finisce nella cella ma prosegue anche nelle precedenti o successive. Nel listato 5.8 è mostrata una parte del test per lo "split". I test vengono fatti creando dei flag che rappresentano i seguenti casi:

flag_os1: contiene la lista di timestamp che indicano gli istanti di tempo in cui il "value" fa riferimento. Il test verifica che in ogni cella ci sia un solo timestamp. Nella cella 0 deve esserci il valore 1.

flag_r: questo flag dovrebbe ripetersi in ogni cella.

flag_r1: questo flag dovrebbe ripetersi solo nella cella 0.

flag_m1m1: questo flag dovrebbe ripetersi in ogni cella.

flag_m18: anche questo flag dovrebbe ripetersi in tutte le celle.

flag_5m1: questo flag dovrebbe ripetersi solo nelle celle 1 e 2.

signal_quality: questo flag dovrebbe ripetersi in ogni cella indipendentemente dai timestamp delle celle.

Nel listato 5.9 è mostrata una parte del test per il "merge". La verifica della "merge" viene svolta creando dei flag per ogni cella che uniti ne formano uno, il test è l'operazione inversa dello "split".

```
let flag = {
    flag_os1: { ts_list: [1, 5, 7], value: "ts_list" },
    flag_r: { ts_range: [1, 9], value: "ts_range" },
    flag_r1: { ts_range: [2, 4], value: "ts_range1" },
    flag_m1m1: { ts_range: [-1, -1], value: "flag_m1m1" },
    flag_m18: { ts_range: [-1, 8], value: "flag_m18" },
    flag_5m1: { ts_range: [5, -1], value: "flag_5m1" },
    signal_quality: "GOOD"
}
let ts = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
let ret = flags_helper.split(flag, ts)
expect(ret[0].flag_os1.ts_list[0]).toBe(1)
expect(ret[0].flag_os1.value).toBe("ts_list")
expect(ret[0].flag_r.ts_range).toEqual([1, -1])
expect(ret[0].flag_r.value).toBe("ts_range")
expect(ret[0].flag_r1.ts_range).toEqual([2, -1])
expect(ret[0].flag_r1.value).toBe("ts_range1")
expect(ret[0].flag_m1m1.ts_range).toEqual([-1, -1])
expect(ret[0].flag_m1m1.value).toBe("flag_m1m1")
expect(ret[0].flag_m18.ts_range).toEqual([-1, -1])
expect(ret[0].flag_m18.value).toBe("flag_m18")
expect(ret[0].flag_5m1).toBe(undefined)
expect(ret[0].signal_quality).toBe("GOOD")
expect(ret[1].flag_os.ts_list.length).toBe(1)
...
```

Listato 5.8: Validazione flags_helper.split

```
let flags = [{
    flag_os1: { ts_list: [1], value: "ts_list" },
    flag_r: { ts_range: [1, -1], value: "ts_range" },
    flag_r1: { ts_range: [2, -1], value: "ts_range1" },
```

```
    flag_m1m1: { ts_range: [-1, -1], value: "flag_m1m1" },
    flag_m18: { ts_range: [-1, -1], value: "flag_m18" },
    signal_quality: "BAD"
}, {
    flag_os1: { ts_list: [5], value: "ts_list" },
    flag_r: { ts_range: [-1, -1], value: "ts_range" },
    flag_r1: { ts_range: [-1, 4], value: "ts_range1" },
    flag_m1m1: { ts_range: [-1, -1], value: "flag_m1m1" },
    flag_m18: { ts_range: [-1, -1], value: "flag_m18" },
    flag_5m1: { ts_range: [5, -1], value: "flag_5m1" },
    signal_quality: "AVERAGE"
}, {
    flag_os1: { ts_list: [7], value: "ts_list" },
    flag_r: { ts_range: [-1, 9], value: "ts_range" },
    flag_m1m1: { ts_range: [-1, -1], value: "flag_m1m1" },
    flag_m18: { ts_range: [-1, 8], value: "flag_m18" },
    flag_5m1: { ts_range: [-1, -1], value: "flag_5m1" },
    signal_quality: "GOOD"
}]
let flag = flags_helper.merge(flags)

expect(flag.flag_os1)
    .toEqual({ ts_list: [1, 5, 7], value: "ts_list" })
expect(flag.flag_r)
    .toEqual({ ts_range: [1, 9], value: "ts_range" })
expect(flag.flag_r1)
    .toEqual({ ts_range: [2, 4], value: "ts_range1" })
expect(flag.flag_m1m1)
    .toEqual({ ts_range: [-1, -1], value: "flag_m1m1" })
expect(flag.flag_m18)
    .toEqual({ ts_range: [-1, 8], value: "flag_m18" })
expect(flag.flag_5m1)
    .toEqual({ ts_range: [5, -1], value: "flag_5m1" })
expect(flag.signal_quality).toBe("GOOD")
```

Listato 5.9: Validazione flags_helper.merge

5.3.2 cache_tail

La validazione di *cache_tail* viene fatta inserendo, prelevando e cancellando le celle dalla lista. I test sono mostrati nel listato 5.10.

Il test prevede l'inserimento di due celle, la prima composta da un *sample* che contiene 1 e 2 e corrisponde al timestamp 1 e al flag "mock_1", la seconda composta da un *sample* che contiene 3 e 4 e corrisponde al timestamp 2 e al flag "mock_2". Se l'aggiunta è eseguita con successo allora vengono salvate 2 celle. Successivamente la prima viene prelevata e confrontata e in seguito si rimuovono entrambe le celle e non dovrebbero essercene altre salvate.

```
let session_id = "00"
let signal_type = "10-1"
let cache_test = new cache_interface( ... )
let cache = new cache_tail(session_id,  cache_test, signal_type,
    test_util.create_signal_map().get(signal_type)!)

// 2 cells: with 1 sample each. With 2 value for each sample
await cache.add_cells_and_info(test_util.
    create_random_ACP_CHUNK(session_id, signal_type, 2, 1),
    [[[1, 2]], [[3, 4]]], [[1], [2]], [
    [{ fl: { ts_list: [1, 2, 3], value: "mock_1" } }],
    [{ fl: { ts_list: [7, 8, 9], value: "mock_2" } }]])
// validazione inserimento
expect(cache.get_length()).toBe(2)

// validazione get
let res = await cache.get_cells_with_info(1)
expect(res.signals).toEqual([[[1, 2]]])
expect(res.ts).toEqual([[1, 2]])
expect(res.flags).toEqual([[
    { fl: { ts_list: [1, 2, 3], value: "mock_1" }}]])
expect(res!.info.session_id).toBe(session_id)
expect(res!.info.signal_type).toBe(signal_type)

// validazione cancellazione
cache.remove_cells(2)
expect(cache.get_length()).toBe(0)

```

Listato 5.10: Validazione cache_tail.add_cells_and_info

5.3.3 windower

Questa classe è verificata solamente per il comportamento generale. La scelta di fare solo i test generali è stata presa perché farli comprendenti di ogni possibile permutazione delle casistiche richiederebbe troppo tempo nella scrittura e nella esecuzione. In questo documento è mostrato il primo test che viene effettuato, quello avente finestra di lunghezza 1 e step 1. Il test è mostrato nel listato 5.11.

Il test prevede l'inserimento di un *Chunk* che causa il riempimento della finestra, che a sua volta chiama la callback utilizzato in "sub_session".

```
let w = new windower(test_util.create_signal_map(), cache)
let session_id: string = session_prefix + "1"

await w.sub_session(session_id, (acp: ACP_CHUNK) => {
    expect(acp).toBeTruthy()
    expect("chunk" in acp).toBe(false)
    if (!("chunk" in acp)) {
        expect(acp.channels.length).toBe(2)
        expect(acp.start).toBeTruthy()
        expect(acp.stop).toBeTruthy()
        expect(acp.session_id).toBe(session_id)
        expect(acp.signal_type).toBe("1-1")
        expect(acp.signal.length).toBe(1)
        expect(acp.ts.length).toBe(1)
    }
})
await w.add_chunk(test_util.
    create_random_ACP_CHUNK(session_id, "1-1", 2, 1))
await w.unsub_session(session_id)
await w.close_connection()
```

Listato 5.11: Validazione windower generale

5.4 Confronto requisiti ACP windower con validazione

Dopo aver mostrato i test effettuati nel *ACP windower*, si prosegue confrontando i traguardi raggiunti con quelli richiesti.

5.4.1 Must Have

Tutti i requisiti *Must Have* sono stati verificati con i test effettuati sulla classe *windower*, che verifica l'intero funzionamento del modulo. In particolare:

Memorizzazione dati Questo requisito è verificato con il test sul metodo "add_chunk" del *windower*, mostrato nel listato 5.11.

Lettura dati salvati Questo requisito è verificato con il test sul metodo "sub_session", questo metodo restituisce un *Chunk* quando una finestra è pronta. Il test è mostrato nel listato 5.11.

Formattazione ACP Questo requisito è verificato utilizzando le interfacce di TypeScript per il *Chunk*. Le interfacce vengono richieste per l'input e utilizzate nell'output. Questo requisito non necessita test perché è già verificato.

5.4.2 Should Have

Cancellazione dati vecchi Questo requisito è stato un po' modificato in fase di progettazione. La richiesta è che fosse temporizzato con un timer, ma si è scelto di implementarlo diversamente per evitare di aggiornare il timer nel caso in cui la *Source* ritardi nel consegnare un *Chunk*. Questo requisito è stato verificato manualmente andando a controllare la reale cancellazione dei dati sul server Redis.

Riusabilità Questo requisito è stato implementato correttamente rendendo possibile impostare la dimensione delle finestre tramite *signal_param*. La riusabilità è gestita da NPM che rende *ACP windower* esportabile come pacchetto e utilizzabile anche altrove.

Configurabilità Questo requisito è stato implementato offrendo la possibilità di fornire altre implementazioni di *cache_manager* e la possibilità di configurare le *signal_param* per ogni finestra che si vuole utilizzare.

Contemporaneità degli utenti Questo requisito è stato implementato gestendo i nomi delle chiavi nel database includendo come prefisso il *session_id* seguito dal *signal_type*. In questo modo ogni chiave diventa univoca e non si corre il rischio di unire i dati di contesti diversi.

5.4.3 Won't Have

Il requisito della validazione dei dati di input non è stato implementato.

5.5 Validazione Gateway

Il gateway è il componente che comprende il maggior numero di casi di utilizzo dell'intero progetto, perciò diventa arduo verificare che ogni singolo modulo sia stato implementato correttamente. La validazione viene effettuata nel dettaglio di *Db_talker* e in generale sugli endpoints *login, chunk, endsession* e *configurations*. La verifica del funzionamento è stata confermata utilizzando sessioni di registrazioni su utenti reali anche lunghe, non sono stati riscontrati problemi strutturali, ma sono stati messi in evidenza alcune criticità minori che verranno risolte in future release.

5.5.1 Db_talker

La validazione di questa classe è stata fatta in dettaglio solo nella creazione del blocco dati perché è possibile ottenere la non dipendenza da altri moduli utilizzando le funzioni simulate per l'invio dei dati al *Db controller* e la richiesta dei dati dell'utente al *Active_users*. Le altre funzioni non hanno dei test specifici perché sono verificate con altri test.

Per i test viene utilizzata una funzione chiamata "get_test_chunk", che crea un *Chunk* potendo impostare opzionalmente *signal_type* e un valore scalare contenuto nel *signal*.

Nel listato 5.12 è mostrato il test sulla corretta creazione del blocco dati da inviare al *Db controller*.

```
import db_talker from "models/db_talker"

beforeAll(async () => {
    spy_db = jest.spyOn(db_talker, "send_data")
        .mockImplementation(() => Promise.resolve(
        { data: "OK", config: {}, headers: {}, status: 200,
        statusText: "OK" }))
    spy_active_users = jest.spyOn(active_users, "get_data")
        .mockImplementation(() => {
        return { organization_id: "mock_1", user_id: "mock_2",
        team_id: "mock_3" }
    })
})
test("should compose a data block with a specific format", () =>{
    db_talker.set_signal_quality("session_id", "BAD")
    db_talker.add_chunk(get_test_chunk())
    db_talker.add_chunk(get_test_chunk("attentional_effort", 0.2))
```

```
    db_talker.add_chunk(get_test_chunk("flow_likelihood", 0.3))
    db_talker.add_chunk(get_test_chunk("mental_fatigue", 0.4))
    expect(spy_db).toHaveBeenCalledWith("/sendSignal", {
    keys: [
        { "name": "organization_id", "value": "mock_1" },
        { "name": "user_id", "value": "mock_2" },
        { "name": "team_id", "value": "mock_3" } ],
    values: [
        { "name": "cognitive_workload", "current": 0.1 },
        { "name": "attentional_effort", "current": 0.2 },
        { "name": "flow_likelihood", "current": 0.3 },
        { "name": "mental_fatigue", "current": 0.4 },
        { "name": "ts", "current": ts },
        { "name": "signal_quality", "current": "BAD" }]})
})
```

Listato 5.12: Validazione Db_talker generale

5.5.2 Endpoint: /login

I test sulla login sono effettuati utilizzando un utente nel database creato esclusivamente per i test. Se la login ha avuto successo l'utente viene salvato tra gli utenti attivi contenuti in *Active_users*. Il test è mostrato nel listato 5.13.

```
describe('When application request to login', () => {
    let login_data = { vibre_id: "###", password: "###" }
    let user_data = { organization_id: "###",
        vibre_id: "###", team_id: "###" }
    let spy_login: jest.SpyInstance
    let server: Server
    beforeAll(async () => {
        server = new Server(express(), 8000)
        let login = server.login_model
        spy_login = jest.spyOn(login, "post_login")
    })
    it("should save the user if the credential is correct",
        async () => {
        await request(server.app)
            .post("/login")
            .type('form')
            .set('Content-Type', 'application/json')
```

```
            .set('Accept', 'application/json')
            .send(login_data)
            .then(response => {
                expect(response.status).toEqual(200)
                expect(response.body).toEqual(
                { success: true, value: "vibre" })
                expect(spy_login).toHaveBeenCalledTimes(1)
                expect(spy_login).toHaveBeenCalledWith(login_data)
                expect(active_users.get_data(login_data.vibre_id))
                    .toEqual(user_data)
            })
    })
})
```

Listato 5.13: Validazione /login

5.5.3 Endpoint: /chunk

Il test viene effettuato analizzando solo la corretta inizializzazione e la composizione della prima finestra con la relativa elaborazione da parte del *NeuroServer*. Questo perché verificare per intero il corretto flusso richiederebbe numerosi test. La verifica della corretta esecuzione viene fatta manualmente confrontando i log con il flusso atteso in una sessione di registrazione reale. I test utilizzano una variabile "chunk_data_long" che contiene un *Chunk* EEG di 1 secondo. La finestra utilizzata per l'EEG è lunga 20 secondi con uno step di 10. I primi dati inviati al *NeuroServer* sono quelli per calcolare *alpha_cog* e *alpha_prevalence* (vedi fig. 3.4). Il primo test non dovrebbe inviare dati al *NeuroServer* e il secondo dovrebbe inviarne due.

```
import active_users from "models/active_users"
let server: Server
let spy_atlasServer: jest.SpyInstance
beforeAll(async () => {
    server = new Server(express(), 8000)
    spy_atlasServer = jest.spyOn(server_talker, "post")
        .mockImplementation(mock_server)
    jest.spyOn(active_users, "interaction")
        .mockImplementation(() => true)
})
it("should save the chunks inside cache",
    async () => {
```

```
    for (let i = 0; i < 19; i++) {
        await request(server.app)
            .post("/chunk")
            .type('form')
            .set('Content-Type', 'application/json')
            .set('Accept', 'application/json')
            .send(chunk_data_long)
            .expect(200, { success: true, value: "OK" })
    }
    expect(spy_atlasServer).toHaveBeenCalledTimes(0)
})
it("should read a given number of chunks from the cache
    and create a standard window", async () => {
    await request(server.app)
        .post("/chunk")
        .type('form')
        .set('Content-Type', 'application/json')
        .set('Accept', 'application/json')
        .send(chunk_data_long)
        .expect(200, { success: true, value: "OK" })
    expect(spy_atlasServer).toHaveBeenCalledTimes(2)
    // one for alpha_cog and one for alpha_prevalences
})
```

Listato 5.14: Validazione /chunk

5.5.4 Endpoint: /endsession

Per verificare che il test abbia avuto successo viene fatta una chiusura seguita da una ulteriore che dovrebbe dare responso negativo comunicando che la sessione non è stata trovata. Il test è mostrato nel listato 5.15.

```
describe("when ending the session", () => {
    it("should end the previous session", async () => {
        await request(server.app)
            .post("/endsession")
            .type('form')
            .set('Content-Type', 'application/json')
            .set('Accept', 'application/json')
            .send({ session_id: chunk_data_long.session_id })
            .expect(200, { success: true, value: "OK" })
```

```
    })
    it("should not found the session", async () => {
        await request(server.app)
            .post("/endsession")
            .type('form')
            .set('Content-Type', 'application/json')
            .set('Accept', 'application/json')
            .send({ session_id: chunk_data_long.session_id })
            .expect(200, { success: false,
                value: "Session not found" })
    })
})
```

Listato 5.15: Validazione /endsession

5.5.5 Endpoint: /configurations

Questo viene verificato semplicemente facendo la GET a */configurations* e confrontando il risultato con il valore atteso. Nel listato 5.16 è mostrato questo test.

```
describe('When application request the configurations', () => {
    let app_chunk_length = +process.env.APP_CHUNK_LENGTH
    let duration_1 = +process.env.DURATION_CALIB_STEP_1
    let duration_2 = +process.env.DURATION_CALIB_STEP_2
    let total_duration = duration_1 + duration_2
    it("should return the neuroservice configuratinons",
        async () => {
        await request(server.app)
            .get("/configurations")
            .type('form')
            .set('Content-Type', 'application/json')
            .set('Accept', 'application/json')
            .expect(200, {
                success: true, value: {
                    calibration: {
                        total_duration: total_duration,
                        steps: [
                            { duration: duration_1,
                                type: "eyes closed" },
                            { duration: duration_2,
```

```
                                type: "eyes open" }
                        ],
                    },
                    flag_list: [
                        { key: "std", type: "ranged" },
                        { key: "marker", type: "oneshot" }
                    ],
                    chunk_length: +app_chunk_length
                }})
    })
})
```

Listato 5.16: Validazione /configurations

5.5.6 Flusso atteso

In questa sezione è mostrato il flusso di interazioni necessarie per eseguire correttamente la sessione di calibrazione o il monitoraggio. I flussi vengono attivati dalla *Source* quando invia un *Chunk*, e sceglie se effettuare la calibrazione, in questo caso inizia il flusso di calibrazione, altrimenti si utilizza il flusso di monitoraggio. Questa sezione è utile per verificare i log in condizione di reale utilizzo. Per ogni fase ci dovrebbe essere un log che ne indica il passaggio, tuttavia verificare se il flusso di calibrazione o monitoraggio è corretto non è immediato perché ci sono molti messaggi in rapida successione, dovuti al riempimento delle finestre e la loro conseguente elaborazione del *NeuroServer*.

Fase 1 - Login

La prima fase comporta la login, viene svolta inviando *vibre_id* e *password* all'endpoint */login*. In questa fase, per verificare che i dati siano corretti, il *Gateway* deve inviare i dati dell'utente al *Db controller* (vedi sezione 7.1.1), il quale in caso siano corretti risponde con un *ACP User* (se l'utente non ha mai fatto una calibrazione vengono ricevuti dei valori di default).

Fase 2 - configurazione

In questa fase la *Source* deve ottenere i parametri per gestire il flusso. Questi dati contengono sia i valori per la calibrazione e sia per il monitoraggio, quindi con una sola richiesta la *Source* è in grado di avviare la sessione di calibrazione e di monitoraggio.

Fase 3 - Invio chunk dati

In questa fase la *Source* invia i dati al *Gateway*. Quando un *Chunk* viene ricevuto si verifica se il mittente ha precedentemente fatto la login. Successivamente il *Chunk* viene salvato nell'*ACP windower*. Questa fase deve essere ripetuta più e più volte per poter fornire i dati necessari per compiere le elaborazioni.

Fase 4 - Nuova finestra pronta

Quando è disponibile una qualunque nuova finestra la si ottiene dall'*ACP windower* e insieme ad eventuali *options*(vedi sezione 3.3.2) viene inviata al *NeuroServer*. Se il flusso richiede di elaborare la stessa finestra con algoritmi diversi vengono fatte più richieste al *NeuroServer*. Quando si riceve la risposta la nuova finestra viene inviata all'*ACP windower* oppure se è un valore finale viene salvato nel *Db_talker* per comporre il blocco di dati. Questa fase deve essere ripetuta ogni qualvolta che una nuova finestra viene creata dall'*ACP windower*.

Fase 5 - Nuovo blocco dati pronto

Quando vengono elaborati tutti i valori finali del flusso in corso, questi vengono inviati in un unico blocco al *Db controller* che li salva nel database. I dati di calibrazione vengono salvati direttamente nella tabella dell'utente, mentre per i dati di monitoraggio vengono salvati in una tabella apposita che verrà utilizzata dalla dashboard. I vecchi valori di calibrazione nel *Gateway* e nel *Db controller* vengono sovrascritti da quelli nuovi.

Fase 6 - Fine

I casi in cui la sessione può finire sono due: la calibrazione è finita oppure la *Source* ha fatto la chiusura manuale. In caso di fine calibrazione, che ha una durata fissa, si può continuare con una sessione di monitoraggio, in caso di fine monitoraggio si deve chiudere la sessione facendo una chiamata all'endpoint *endsession*.

Se la sessione viene terminata tutti i dati presenti nell'*ACP windower* vengono eliminati perché non più richiesti.

5.6 Confronto requisiti Gateway con validazione

Dopo aver mostrato i test effettuati nel *Gateway*, si prosegue confrontando i traguardi raggiunti con quelli richiesti.

5.6.1 Must Have

I requisiti sono verificati nei seguenti modi:

Flusso di monitoraggio Questo requisito viene verificato facendo i test manuali di utilizzo del *Gateway*. Si confrontano i log con il flusso atteso di registrazione.

Flusso di calibrazione Questo requisito è verificato in parte tramite i test, che ne verificano la corretta inizializzazione e la prima elaborazione, e in parte confrontando i log con il flusso atteso di registrazione.

Salvare i dati elaborati Questo requisito è stato verificato dal test su *Db_talker*. Se i dati gli vengono inviati esso provvederà a inviarli al *Cb controller* quando pronti.

Dati di configurazione Questo requisito è stato verificato dal test su */configurations* che verifica il valore restituito con quello atteso.

Gestione sessione utente Questo requisito è stato verificato in */login, /chunk* e */endsession*. Se la sessione è creata si ottengono i dati dell'utente e il *Chunk* viene accettato, se viene cancellata con successo non è possibile cancellarla nuovamente.

5.6.2 Should Have

I requisiti sono verificati nei seguenti modi:

Diverse sessioni utente contemporaneamente Questo requisito viene verificato con test manuali. In questo modo si possono ottenere valori indicativi sulla capacità di calcolo, memoria e tempo richiesti aumentando il numero di richieste.

Configurabile nei suoi parametri Questo requisito è verificato rendendo tutti i valori nel modulo *Constant* configurabili tramite variabili di ambiente.

Dovrebbe essere semplice aggiornare i flussi Questo requisito è verificato dalla struttura di *Operation* che permette di aggiornare i flussi (vedi sezione 4.6), e da *Constant* permette di aggiornare la dimensione delle finestre essendo parametri configurabili tramite variabili d'ambiente.

5.6.3 Won't Have

I requisiti non sono stati implementati.

Capitolo 6

Conclusioni

Il *Gateway* è un componente indispensabile per elaborare correttamente le metriche di *NeuroFrame*. La sua implementazione rende possibile effettuare sessioni di registrazione anche molto lunghe grazie alla gestione efficiente della cache, che permette di salvare i dati in tempo reale. Il *Gateway* deve interagire con altri sistemi: fare da tramite è un compito autorevole, perché si assume la responsabilità di gestire correttamente il flusso di operazioni per mettere in comunicazione i diversi sistemi rendendoli, dall'esterno, un'unica entità in grado di elaborare i segnali complessi.

Il progetto è stato implementato e viene utilizzato con successo e, data l'alta modificabilità dei flussi di elaborazioni, può essere utilizzato anche per *NeuroServices* diversi da *NeuroFrame*.

Questo componente è stato l'ultimo di *NeuroFrame* ad essere sviluppato, perciò ha reso possibile verificare l'intera piattaforma sottoponendola a clienti reali che ne hanno potuto apprezzare le caratteristiche.

Capitolo 7

Appendice

7.1 Endpoints

7.1.1 Endpoints Db controller

Il *Db controller* implementa numerosi endpoint per gestire le strutture dati. Gli endpoint di interesse al *Gateway* sono 4.

login_user

- **Endpoint:** /login_user
- **Metodo HTTP:** POST
- **body:**

```
{
    username: string,
    password: string
}
```

Listato 7.1: /login_user body

- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: "organization_id"
}
```

Listato 7.2: /login_user valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 7.3: /login_user valore atteso risposta negativa

sendSignal

- **Endpoint:** /sendSignal
- **Metodo HTTP:** POST
- **body:**

```
{
    "keys":[
      { "name":"organization_id", "value": string },
      { "name":"team_id", "value": string },
      { "name":"user_id", "value": string }
    ],
    "values":[
      { "name": string, "current": number }, // signal, value
      ...
    ]
}
```

Listato 7.4: /sendSignal body

- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: true
}
```

Listato 7.5: /sendSignal valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
```

```
    value: "description error"
}
```

Listato 7.6: /sendSignal valore atteso risposta negativa

getUser

- **Endpoint:** /getUser
- **Metodo HTTP:** POST
- **body:**

```
{
    user_id: string,
    organization_id: string
}
```

Listato 7.7: /getUser body

- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: {
        user_id: string,
        current_team: string,
        biometrics: Map<string, number>
    }
}
```

Listato 7.8: /getUser valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 7.9: /getUser valore atteso risposta negativa

updateUserBioSignal

- **Endpoint:** /updateUserBioSignal
- **Metodo HTTP:** POST
- **body:**

```
{
    user_id: string,
    current_team: string,
    biometrics: Map<string, number>
}
```

Listato 7.10: /updateUserBioSignal body

- **Valore esempio risposta affermativa:**

```
{
    success: true,
    value: number // n. of modified elements
}
```

Listato 7.11: /updateUserBioSignal valore atteso risposta positiva

- **Valore esempio risposta negativa:**

```
{
    success: false,
    value: "description error"
}
```

Listato 7.12: /updateUserBioSignal valore atteso risposta negativa

7.1.2 EndPoints NeuroServer

Il suo endpoint è solo uno e viene utilizzato per tutte le metriche implementate.

Measure

Elabora i dati ricevuti utilizzando il campo *model* che indica l'algoritmo da utilizzare.

- **Endpoint:** /measure

- **Metodo HTTP:** POST
- **body:** *MOC datatype* del protocollo *ACP*(sezione 3.3.2)
- **Valore esempio risposta affermativa:** *Chunk datatype* del protocollo *ACP*(sezione 3.3.2).
- **Valore esempio risposta negativa:** *Chunk datatype* del protocollo *ACP*(sezione 3.3.2) con valore *NaN* nei *signal*.

Bibliografia

[1] Wikipedia contributors. Distributed computing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Distributed_computing&oldid=1007699786`, 2021. Accessed: 2021-02-23.

[2] Wolfgang Emmerich. Distributed system principles. `/http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/ds98-99/dsee3.pdf`, 1997. Accessed: 2021-02-23.

[3] Juan Gonzalez-Calleros, Jan-Patrick Osterloh, Rene Feil, and Andreas Lüdtke. Automated ui evaluation based on a cognitive architecture and usixml. *Science of Computer Programming*, 86:43–57, 2014. Special issue on Software Support for User Interface Description Languages (UIDL 2011).

[4] Weidong Huang, Seok-Hee Hong, and Peter Eades. Predicting graph reading performance: A cognitive approach. In *Predicting graph reading performance: A cognitive approach*, volume 60, pages 207–216, 01 2006.

[5] InteraXon Inc. Muse™ - meditation made easy with the muse headband. `https://choosemuse.com/`. Accessed: 2021-03-03.

[6] Abdullah Talha Kabakus and Resul Kara. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences*, 29(4):520–525, 2017.

[7] William D.S. Killgore. Effects of sleep deprivation on cognition. In Gerard A. Kerkhof and Hans P.A. van Dongen, editors, *Effects of sleep deprivation on cognition*, volume 185 of *Progress in Brain Research*, pages 105–129. Elsevier, 2010.

[8] Andrew S. Tanenbaum Maarten van Steen. A brief introduction to distributed systems. `/https://www.cs.vu.nl/~ast/Publications/Papers/computing-2016.pdf`, 2016. Accessed: 2021-02-23.

[9] Ludmila Mládková, Jarmila Zouharová, and Jindřich Nový. Motivation and knowledge workers. *Procedia - Social and Behavioral Sciences*, 207:768–776, 2015. 11th International Strategic Management Conference.

[10] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162, 2009.

[11] John Sweller, Jeroen J. G. van Merrienboer, and Fred G. W. C. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10(3):251–296, Sep 1998.

[12] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[13] Wikipedia. Npm (software) — wikipedia, l'enciclopedia libera, 2020. [Online; in data 27-febbraio-2021].

[14] Wikipedia. Onde cerebrali — wikipedia, l'enciclopedia libera, 2020. [Online; in data 3-marzo-2021].

[15] Wikipedia. Psicologia positiva — wikipedia, l'enciclopedia libera, 2021. Accessed: 2021-02-23.

[16] Wikipedia contributors. Application software — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Application_software&oldid=1009245990`, 2021. [Online; accessed 27-February-2021].

[17] Wikipedia contributors. Spline interpolation — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Spline_interpolation&oldid=1002256014`, 2021. [Online; accessed 26-February-2021].

[18] F. Zeng, S. Rebscher, W. Harrison, X. Sun, and H. Feng. Cochlear implants: System design, integration, and evaluation. *IEEE Reviews in Biomedical Engineering*, 1:115–142, 2008.