



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

RT-Jam

Elaborato di Web and Real Time Communication Systems

Anno Accademico 2023/2024

Giuseppe Capasso
matr. M63001498

Indice

1	Architettura	3
1.1	Tecnologie utilizzate	3
1.1.1	Linguaggio: Rust	3
1.1.2	WebAssembly	11
1.1.3	Protocollo QUIC	13
1.1.4	Serializzazione binaria: protocol buffer	17
1.1.5	Broadcast: Nats	18
1.2	Servizio REST	20
2	Funzionamento dell'applicazione	21
2.1	Autenticazione e autorizzazione	21
2.1.1	Creazione account	21
2.1.2	Login	24
2.2	Piattaforma	26
3	Deploy	28
3.1	Bare metal	28
3.2	Docker	29
3.3	Test dell'applicazione	29
	References	32

Introduzione

Il progetto ha come obiettivo quello di esplorare un modo più moderno di approcciarsi al mondo dello sviluppo web. Oggigiorno le applicazioni devono essere altamente interattive e vanno oltre la semplice visualizzazione di immagini statiche e pagine HTML per cui era stato pensato il web inizialmente. Infatti, il web prende un'accezione sempre più larga in cui alla navigazione classica mediante un browser si aggiungono requisiti di sicurezza, performance e comunicazioni a bassa latenza.

Il progetto sfrutta la tecnologia WebAssembly e il protocollo QUIC per realizzare una soluzione di streaming portatile. Per la parte WebTransport, è stato riutilizzato il codice in (SecurityUnion, 2024) insieme alle definizioni ProtocolBuffer utilizzate anche nel server. In particolare, come spiegato in §1.1.3, è stato riutilizzato il codice della libreria chiamata *videocall-client* a cui sono apportate le seguenti modifiche:

- Eliminazione dell'implementazione mediante websocket;
- Eliminazione della parte di screen-sharing;
- Modifica della parte audio per inviare e ricevere audio stereo con codec opus;
- Aggiunta di una callback per recuperare l'id della sorgente audio selezionata, per implementare un oscilloscopio;

1 Architettura

Come mostrato in Figura 1, l'architettura del sistema è divisa:

- **frontend**: applicazione client utilizzata dall'utente per interagire con gli altri utenti;
- **backend**: applicazione server che si occupa di esporre un'interfaccia per la gestione di account e stanze attraverso un API JSON e di implementare il *broadcasting* dei contenuti multimediali generati dai client con il protocollo QUIC;

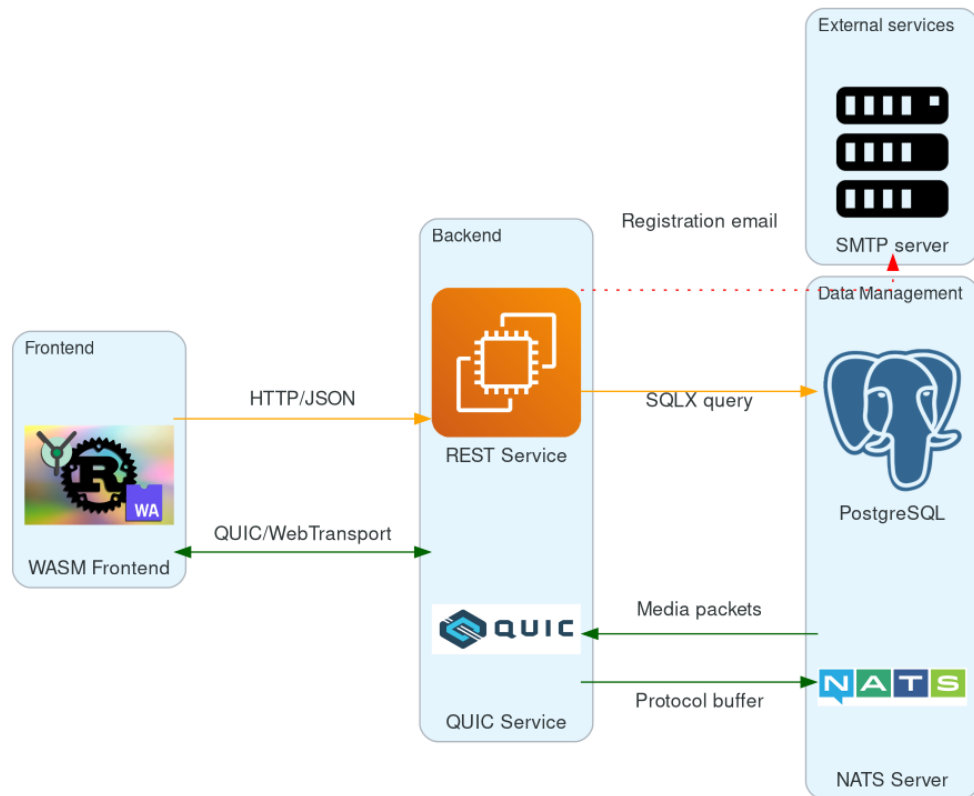


Figure 1: Architettura del sistema

Il sistema è stato modellato con il diagramma E/R in Figura 2, implementato nel database relazionale mostrato nell'architettura.

Il funzionamento generale dello scambio di informazioni multimediali è mostrato in Figura 3.

1.1 Tecnologie utilizzate

Di seguito è riportata una panoramica delle tecnologie impiegate.

1.1.1 Linguaggio: Rust

Rust è un linguaggio di quarta generazione focalizzato su *safety*, *performance* e *security*. Queste caratteristiche vengono implementate a tempo di compilazione attraverso il **borrow checker** che si occupa di garantire (attraverso degli opportuni controlli) che non si verifichino accessi in memoria illegali e, quindi, evita la presenza di vulnerabilità quali *double free use after free*. Questo approccio inoltre, rende non necessaria la presenza di un **garbage collector** e di un *runtime* rendendolo adatto sia per applicazioni embedded che per quelle che richiedono alte prestazioni. Questo aspetto viene implementato attraverso le *enum* che

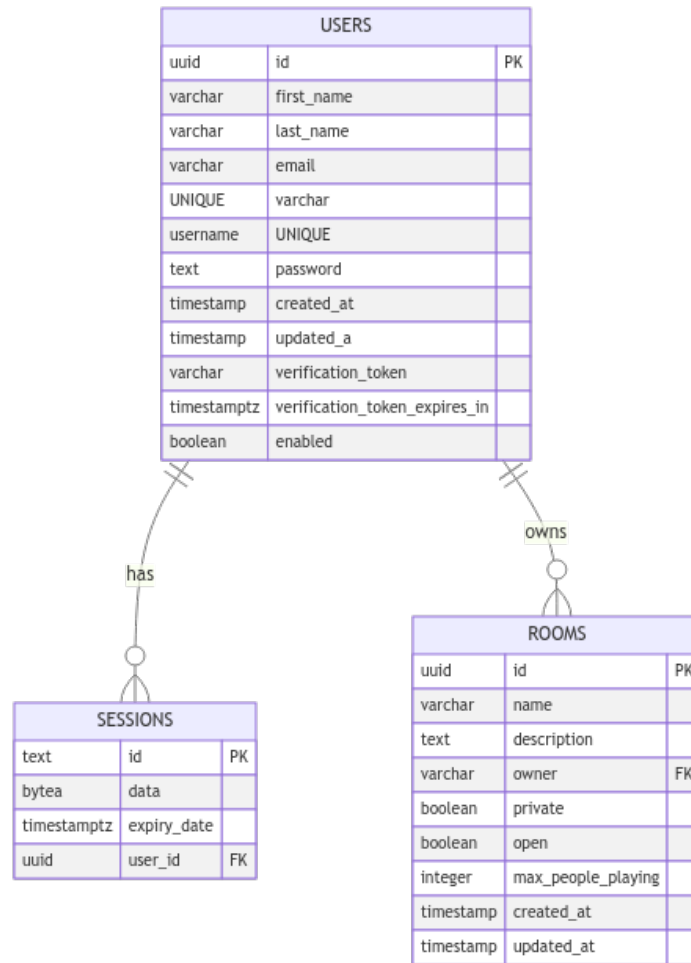


Figure 2: Diagramma E/R dell'applicazione

devono essere ispezionate con un pattern matching esaustivo. e sono utilizzate con un patter matching esaustivo. Un esempio lampante è l'enum ***Option*** $\langle T \rangle$ che può assumere due varianti: *None* o *Some* T . Un oggetto incluso in questo tipo di enum può essere nullo e l'accesso al suo valore forza il programmatore a estrarlo dall'enum Option.

```

let n: Option<i32> = Some(1); //diverso da i32 perchè potrebbe essere None
if let Some(n) = n {
    // si è sicuri che n è una variabile non nulla
}

```

Dualmente, l'enum *Result* $\langle T, Error \rangle$ indica un'operazione che può restituire un errore e implementa il concetto di ***error-as-value***. Questo costrutto forza il programmatore a gestire l'errore ed è possibile enumerare (essendo l'errore un tipo) i modi in cui un programma può fallire sapendo esattamente quali funzioni possono restituire errori. Al contrario, in un meccanismo basato su eccezioni non è possibile sapere quale funzione all'interno di un blocco *try-catch* abbia sollevato un'eccezione.

Inoltre, la presenza del borrow checker assicura che non siano effettuati accessi a puntatori invalidi riducendo il numero di crash. Infine, il linguaggio offre un meccanismo simile al poliformismo detto *Trait* che consente estendere comportamenti di tipi custom con tipi standard. Ad esempio, è possibile implementare

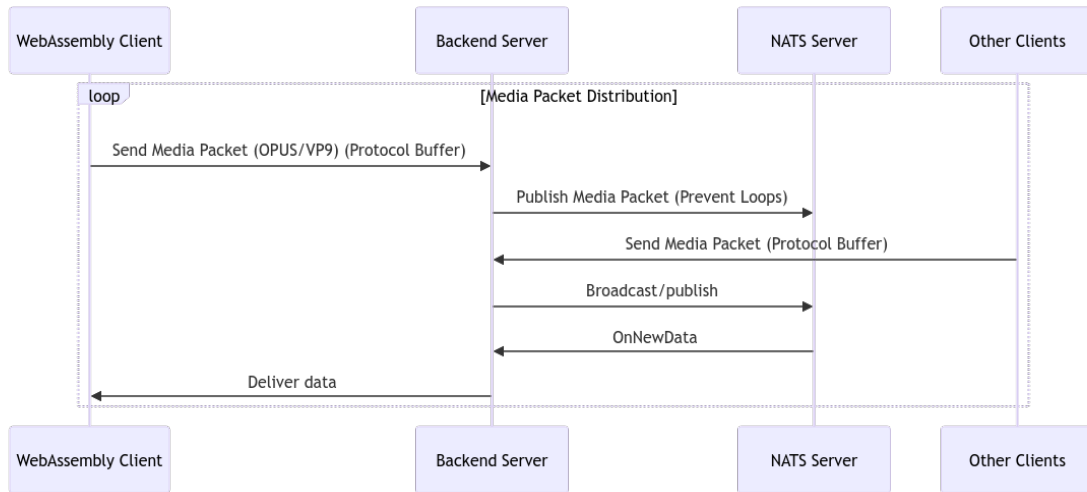


Figure 3: Descrizione dell'attività di broadcast

il modo in cui un oggetto viene stampato quando si è in fase di debug o implementare metodi di casting sicuro da un tipo ad un altro.

Web Framework: Axum Per realizzare le funzionalità Web, è stato adottato il framework Axum costruito sul *runtime* asincrono *Tokio* che consente di sviluppare servizi web con uno stile simile a *Springboot* utilizzando i pattern *IoC (Inversion of Control)* e *DI (Dependency Injection)*. Il framework introduce il concetto di stato dell'applicazione: un oggetto trasmesso agli handler che può essere estratto dal contesto dell'applicazione. Ad esempio, in listato 1, è possibile osservare un handler che effettua il login che prende in ingresso un oggetto di tipo *AppState* al cui interno sono presenti i servizi per gestire le sessioni (*session_service*) e autenticazione (*auth_service*).

```

1  async fn login(
2      State(AppState {
3          auth_service,
4          session_service,
5      }): State<AppState>,
6      cookies: Cookies<'_>,
7      Json(LoginRequest { username, password }): Json<LoginRequest>,
8  ) -> Result<impl IntoResponse> {
9      let user = auth_service.login(username, password).await?;
10     let expiration = {
11         let this = OffsetDateTime::now_utc();
12         let duration = Duration::days(7);
13         this.checked_add(duration)
14             .expect("resulting_value_is_out_of_range")
15     };
16     let token = session::Service::generate_token();
17     session_service
18         .create(
19         &token,
20         &session::SessionData::from(user.clone()),
21         expiration,
22     )
23     .await?;
  
```

```

24
25     let cookie = Cookie::build(Cookie::new(SESSION_COOKIE_NAME, token))
26         .http_only(true)
27         .secure(true)
28         .expires(expiration)
29         .path("/")
30         .same_site(SameSite::Strict)
31         .build();
32
33     cookies.add(cookie);
34
35     Ok(AJson(json!({
36         "result": { "success": true }
37     })))
38 }

```

Listing 1: Esempio di handler in Axum

Infine, il framework mette a disposizione un meccanismo di routing e middleware per consentire l'esecuzione di uno stack di funzioni prima di arrivare ad un handler (listato 2).

```

1 pub fn router(auth_service: auth::Service, session_service: session::Service) -> Router {
2     Router::new()
3         .route("/sign-in", post(login))
4         .route("/sign-up", post(register))
5         .route("/sign-out", post(logout))
6         .route("/me", get(me))
7         .route("/change-password", post(change_password))
8         .route("/start-reset", post(start_reset))
9         .with_state(AppState {
10             auth_service,
11             session_service,
12         })
13 }

```

Listing 2: Esempio di router in Axum

Interazione con database: SQLX Sqlx è una libreria che consente di interagire con molti database relazionali con la possibilità di effettuare query *type-safe* a tempo di compilazione. In particolare, necessita della variabile d'ambiente `DATABASE_URL` che punta al database in utilizzo e controlla, durante la fase di compilazione, che tutti i parametri in ingresso e in uscita dalla query siano coerenti con lo schema in uso. Inoltre, offre anche un *utility* da riga di comando per gestire migrazioni il cui risultato è presente all'interno della cartella `backend/migrations` del progetto. Ad esempio, in listato 3, è mostrata una query effettuata utilizzando la macro `sqlx::query_as!`

```

1 pub async fn login(&self, username: String, password: String) -> Result<User> {
2     let user = sqlx::query_as!(
3         User,
4         "SELECT_*_FROM_users_WHERE_username=_$_1_AND_enabled=_TRUE",
5         username
6     )
7     .fetch_optional(&self.db)
8     .await
9     .map_err(|e| Error::DatabaseError(e))?;
10
11     let user = match user {

```

```

12     Some(user) => user,
13     None => return Err(Error::InvalidCredentials),
14 };
15
16 let password_hash = user.clone().password.ok_or(Error::InvalidCredentials)?;
17
18 let password_hash = PasswordHash::new(&password_hash).map_err(|_e| Error::CryptoError)?;
19
20 Argon2::default()
21     .verify_password(password.as_bytes(), &password_hash)
22     .map_err(|_e| Error::InvalidCredentials)
23     .map(|_a| user)
24 }

```

Listing 3: Esempio di query in Rust con SQLX: il compilatore assicura che il tipo della variabile *username* sia compatibile con quello del campo della tabella

Protocollo SMTP: Lettre e Askama Lettre è una libreria che implementa il protocollo SMTP per l'invio di email. Infatti, l'applicazione invia una email:

- ogni volta che viene creato un account per verificare l'identità del nuovo utente e permettergli di impostare una password;
- per permettere all'utente di re-impostare la password in caso l'avesse dimenticata;

In listato 4, è mostrato come viene inviata un email nell'applicazione.

```

1 pub async fn send_verification_link(
2     &self,
3     token: &str,
4     user: &User,
5 ) -> Result<(), Box<dyn std::error::Error>> {
6     let link = format!("{}/change-password?token={}", self.app_url, token);
7     let subject = format!("Welcome, {}!", user.first_name);
8
9     let template = ChangePassword {
10         subject: subject.clone(),
11         user: user.clone(),
12         message:
13             String::from("Welcome onboard! Please, ..."),
14         link,
15     }.render()?;
16
17     let email = Message::builder()
18         .to(format!("{}", <{}>", user.first_name, user.email)
19             .parse()
20             .unwrap())
21         .reply_to(self.from.parse().unwrap())
22         .from(self.from.parse().unwrap())
23         .subject(subject)
24         .header(ContentType::TEXT_HTML)
25         .body(template)?;
26
27     self.transport.send(email).await?;
28
29     Ok(())

```



```
30 }
```

Listing 4: Esempio di invio email in Rust con Lettre

In particolare, l'email viene costruita in formato HTML con un *Askama*: una libreria per il templating dei file HTML che consente di effettuare il parsing di file di input e di dichiarare dei placeholder che possono essere sostituiti con dati veri a tempo di esecuzione. In, è mostrato il template usato nel progetto.

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7   <title>{{subject}}</title>
8
9   {% include "partials/email_style.html" %}
10 </head>
11
12 <body>
13   <table role="presentation" border="0" cellpadding="0" cellspacing="0" class="body">
14     <tr>
15       <td>&nbsp;</td>
16       <td class="container">
17         <div class="content">
18           <!-- START CENTERED WHITE CONTAINER -->
19           {% block content %} {% endblock %}
20           <!-- END CENTERED WHITE CONTAINER -->
21         </div>
22       </td>
23       <td>&nbsp;</td>
24     </tr>
25   </table>
26 </body>
27
28 </html>
```

Listing 5: Layout email utilizzato

Si noti come il contenuto principale dell'email sia stato indicato con il placeholder *block*. Questo file viene incluso dal file email che ne estende il comportamento dichiarando il contenuto del blocco content, mostrato in

```
1 {% extends "layouts/email.html" %}
2
3 {% block content %}
4   ...omesso per brevit
5   <p>Hi {{user.first_name}},</p>
6   <p>{{message}}</p>
7   <table role="presentation" border="0" cellpadding="0" cellspacing="0" ...>
8   <tbody>
9     <tr>
10       <td align="left">
11         <table role="presentation" border="0" cellpadding="0" cellspacing="0">
12           <tbody>
13             <tr>
14               <td>
```

```

15         <a href="{{link}}" target="_blank">Verify your account</a>
16     </td>
17 </tr>
18 </tbody>
19 </table>
20 </td>
21 </tr>
22 ...omesso per brevit
23 </tr>
24 </table>
25 {% endblock %}

```

Listing 6: Corpo dell'email utilizzato

Frontend framework: Yew Yew è un framework che permette di scrivere applicazioni web in Rust utilizzando WebAssembly. In particolare, Yew mette a disposizione la macro *html!* che permette di scrivere sintassi simile a *JSX* per creare pagine web type-safe a tempo di compilazione. Inoltre, la struttura delle applicazioni è molto simile a quella utilizzata in framework quali *React.js* ed è possibile scrivere componenti sia in forma funzionale che di classe. Infine, il framework mette a disposizione *hook* come *use_state*, *use_effect* che permettono di realizzare interfacce responsive. In listato 7, è mostrato un esempio di componente funzionale utilizzato nell'applicazione. Per la realizzazione dell'applicazione è stata utilizzata *Yewdux* (acceduta attraverso l'hook *use_store()* a riga 11): una libreria di *state management* simile a *Redux* basata su concetti di *getter* e *reducer*.

```

1  #[function_component(Session)]
2  pub fn session(Props { id }: &Props) -> Html {
3      let clipboard = use_clipboard();
4      let to_clipboard = {
5          let id = id.clone();
6          Callback::from(move |_|: MouseEvent| {
7              clipboard.write_text(id.to_string());
8          })
9      };
10     let navigator = use_navigator().unwrap();
11     let (store, dispatch) = use_store::<Store>();
12     use_effect(move || {
13         spawn_local(async move {
14             match Request::get("/api/auth/me").send().await {
15                 Ok(res) => {
16                     if res.ok() {
17                         let user = res.json::<UserResponse>().await.unwrap();
18                         dispatch.reduce_mut(move |s| s.auth_user = Some(user.into()));
19                     } else {
20                         navigator.replace(&Route::Login);
21                     }
22                 }
23                 // network error
24                 Err(err) => {
25                     log_1(&err.to_string().into());
26                 }
27             };
28         });
29     });
30
31     let user = store.auth_user.clone();

```

```

32     html! {
33         <>
34         if let Some(user) = user {
35             <Header />
36             <div class="flex flex-row px-4 py-2">
37                 <h1>{"Session_id:"}</h1>
38                 <div onclick={to_clipboard} class="block text-lg ... altre_classi">
39                     {id}
40                     <i class="ml-1 fa-clipboard fa-solid"></i>
41                 </div>
42                 <h1>{"_Share_it_with_your_friends!"}</h1>
43             </div>
44             <div class="flex justify-center">
45                 <Client username={user.username} id={id.clone().to_string()}>
46             </div>
47         }
48     </>
49 }
50 }

```

Listing 7: Esempio di componente funzionale

Yew offre un router per gestire le pagine dell'applicazione attraverso una *Enum* di Rust che grazie ad una funzione riportata in listato 9. Grazie, al pattern matching esaustivo è possibile creare dei check a tempo di compilazione per assicurarsi che l'applicazione mostra le pagine giuste con gli input corretti. In listato 8, è mostrato il router utilizzato nell'applicazione.

```

1  #[derive(Clone, Routable, PartialEq)]
2  pub enum Route {
3      #[at("/")]
4      Home,
5
6      #[at("/login")]
7      Login,
8
9      #[at("/register")]
10     Register,
11
12     #[at("/change-password")]
13     ChangePassword,
14
15     #[at("/start-reset")]
16     StartReset,
17
18     #[at("/session/:id")]
19     Session { id: String },
20
21     #[at("/create-room")]
22     CreateRoom,
23
24     #[not_found]
25     #[at("/not-found")]
26     NotFound,
27 }

```

Listing 8: Rotte utilizzate nel frontend

```

1 pub fn switch(routes: Route) -> Html {
2     match routes {
3         Route::Home => html! {
4             <SimpleLayout>
5                 <Home />
6             </SimpleLayout>
7         },
8         Route::Session { id } => html! {
9             <SimpleLayout>
10                <Session {id}/>
11            </SimpleLayout>
12        },
13        // altre rotte
14    }
15 }

```

Listing 9: Funzione switch utilizzata per il mapping di rotte con componenti sviluppati

Interazione con browser API: `web_sys` e `wasm_bindgen` Ogni applicazione che esegue in un browser deve utilizzare Javascript. Infatti, WebAssembly è una tecnologia che permette di scrivere applicazioni un linguaggio a scelta, ma ha bisogno di un *bundler* (spiegato in §1.1.2) che permette di effettuare il mapping tra le API del browser Javascript e quelle del codice in Rust. Ad esempio, per effettuare una richiesta HTTP si userà la corrispettiva funzione *fetch* nel browser (esempio a riga 14 di listato 7) o per stampare qualcosa nel browser si userà *console.log*. Le due librerie *web_sys* e *wasm_bindgen* offrono questi mapping, e, nella loro versione instabile offrono l'accesso a componenti quali **AudioEncoder** e **WebTransport** (implementazione del protocollo QUIC nei browser, spiegato in §1.1.3). Ad esempio, in listato 10, è mostrato come si configura l'AudioEncoder.

```

1 let mut audio_encoder_config = AudioEncoderConfig::new(AUDIO_CODEC);
2 audio_encoder_config.bitrate(AUDIO_BITRATE);
3 audio_encoder_config.sample_rate(AUDIO_SAMPLE_RATE);
4 audio_encoder_config.number_of_channels(AUDIO_CHANNELS);
5 let ok = AudioEncoder::is_config_supported(&audio_encoder_config);
6 log_1(&ok.to_string().js_typeof());
7 audio_encoder.configure(&audio_encoder_config);
8
9 let audio_processor = MediaStreamTrackProcessor::new(&MediaStreamTrackProcessorInit::new(
10     &audio_track.clone().unchecked_into::<MediaStreamTrack>(),
11 ))
12 .unwrap();

```

Listing 10: Accesso e configurazione di AudioEncoder

1.1.2 WebAssembly

WebAssembly (Rossberg, 2021) è un formato binario per una macchina a stack. I browser moderni inglobano al loro interno una macchina WebAssembly e, pertanto tutti i linguaggi, che producono un programma secondo la specifica di questa macchina possono eseguire nel browse. Questo consente di essere indipendente dal target e di avere un'esecuzione isolata dei processi (in maniera simile alla JVM) con il risultato di avere da un lato le performance del linguaggio scelto (ad esempio Rust, C++, C o Go) e dall'altro di avere un ambiente unificato di esecuzione (Figura 4). Oltre all'encoding in un formato binario molto più efficiente rispetto a quello classico testuale, WebAssembly specifica la struttura delle istruzioni, layout della memoria, formato dei dati e inoltre, specifica la struttura di un runtime asincrono per le applicazioni I/O intensive. Infine, con questa tecnologia è facile far eseguire le applicazioni in *sandbox* migliorando gli aspetti relativi alla sicurezza.

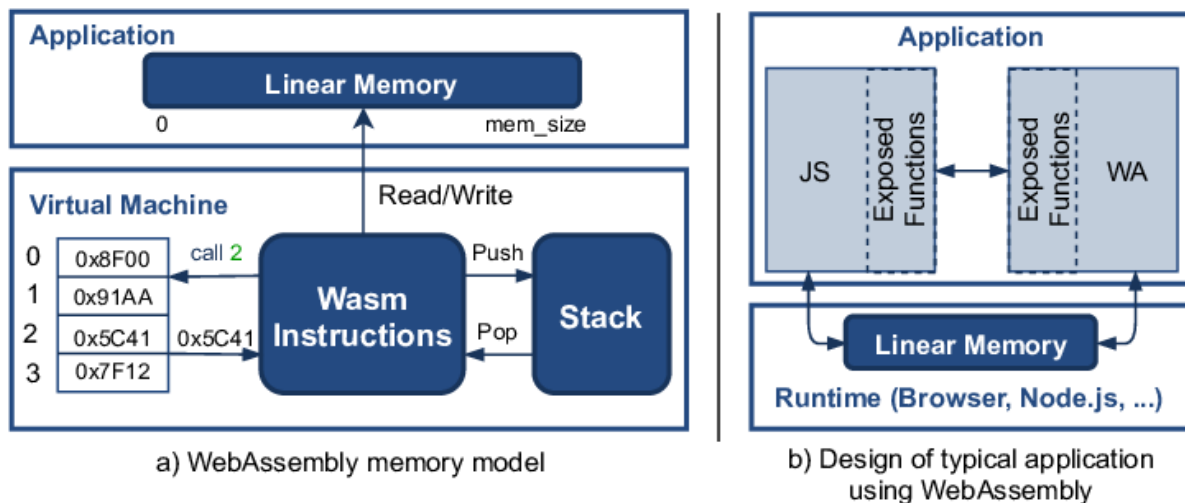


Figure 4: Architettura ad alto livello di una macchina WebAssembly

WASM bundler: Trunk Trunk è il bundler scelto per preparare l'applicazione per eseguire in una macchina WebAssembly. Innanzitutto, è stato aggiunto al compilatore Rust il target *wasm32-unknown-unknown* con il comando:

```
rustup target add wasm32-unknown-unknown
```

Il frontend può essere eseguito con il comando sottostante. Trunk include un proxy http che offre la possibilità di effettuare il routing delle richieste per uno specifico path ad una applicazione e si occupa di gestire i cookie interdominio e di propagare eventuali errori.

```
trunk serve --proxy-backend=http://localhost:3000/api/
```

Trunk può essere configurato attraverso il file *Trunk.toml* attraverso il quale è possibile indicare la cartella in cui si trova il codice sorgente, impostazioni di proxy e la cartella di uscita. In particolare, Trunk compila il codice sorgente dell'applicazione e crea una SPA (Single Page Application): infatti ha bisogno di un file *index.html* con il tag *body* vuoto che andrà a popolare con l'html generato dinamicamente (si tratta a tutti gli effetti di Client Side Rendering). Inoltre, vengono dichiarati alcuni tag speciali (*data-trunk*) per il serving statico dei file. In, è mostrato il file utilizzato da Trunk come entrypoint dell'applicazione; mentre in è mostrato il file di configurazione utilizzato.

```

1 <!DOCTYPE html>
2 <html class="dark" lang="en"
3   <head>
4     <title>RT-Jam</title>
5     <meta charset="UTF-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7     <link data-trunk rel="css" href="styles/output.css" />
8     <link data-trunk rel="copy-dir" href="static" />
9     <link rel="icon" href="static/favicon.ico" type="image/x-icon" />
10    ... altri link
11  </head>
12  <body class="text-white bg-gray-50 dark:bg-gray-900">
13
14  </body>
15 </html>

```

Listing 11: Entrypoint per applicazione WASM

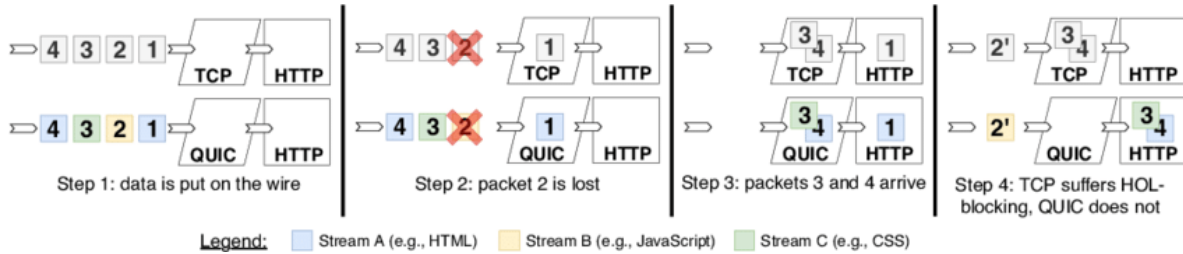


Figure 5: Head of Line blocking con 3 flussi di dati indipendenti (Marx et al., 2020)

1.1.3 Protocollo QUIC

QUIC (Quick UDP Internet Connections) §1.1.3 è un protocollo di livello trasporto basato su HTTP/3 pensato per risolvere il problema di *head-of-line blocking* che si incontrano in protocolli *stream oriented* quali TCP in cui un pacchetto appartenente a uno stream blocca tutti gli altri. QUIC risolve questo problema supportando nativamente il multiplexing dei flussi di dati; in questo modo i flussi più lenti o con un'alta *packet-loss* non influenzano gli altri.

Il problema dell'head-of-line blocking I protocolli basati su TCP come HTTP/1.1 e HTTP/2 risolvono in maniera diversa *HOL*. Infatti, per HTTP/1.1, i browser utilizzano fino a 6 connessioni concorrenti per ogni dominio in modo da minimizzare il tempo di download per le risorse; mentre HTTP/2 cerca di effettuare il multiplexing delle risorse su una singola connessione TCP dividendo i dati in **chunk** a cui viene associato un identificativo univoco e poi schedato sulla connessione. Il primo approccio introduce un problema di saturazione della banda (più richieste concorrenti utilizzano più risorse di rete per il server), mentre il secondo essendo basato su TCP soffre di HOL. In particolare, il problema riscontrato è detto **inter-stream head-of-line blocking**: stream diversi vengono interrotti dal flusso con alta *packet-loss* poiché TCP implementa una comunicazione con ritrasmissione. Con HTTP/3, questo problema viene risolto (Figura 5) con l'adozione di UDP, ma permane il problema dell'**intra-stream head-of-line blocking** (Marx, De Decker, Quax, & Lamotte, 2020).

QUIC server-side Per implementare il protocollo QUIC lato server sono state utilizzate le librerie *sec.http3* (responsabile della gestione di ALPN e della parte TLS) e *quinn* (responsabile dell'implementazione del protocollo QUIC). In listato 12, è mostrata la configurazione di un listener UDP su cui viene applicato un certificato self-signed e ALPN, dichiarati come costanti:

```
pub const WEB_TRANSPORT_ALPN: &[u8] = &[b"h3", b"h3-32", b"h3-31", b"h3-30", b"h3-29"];
pub const QUIC_ALPN: &[u8] = b"hq-29";

1 let (key, certs) = get_key_and_cert_chain(opt.certs)?;
2
3 let mut tls_config = rustls::ServerConfig::builder()
4   .with_safe_default_cipher_suites()
5   .with_safe_default_kx_groups()
6   .with_protocol_versions(&[rustls::version::TLS13])
7   .unwrap()
8   .with_no_client_auth()
9   .with_single_cert(certs, key)?;
10
11 tls_config.max_early_data_size = u32::MAX;
12 let mut alpn = vec![];
13 for proto in WEB_TRANSPORT_ALPN {
14   alpn.push(proto.to_vec());
```

```

15 }
16 alpn.push(QUIC_ALPN.to_vec());
17
18 tls_config.alpn_protocols = alpn;
19
20 // 1. create quinn server endpoint and bind UDP socket
21 let mut server_config = quinn::ServerConfig::with_crypto(Arc::new(tls_config));
22 let mut transport_config = quinn::TransportConfig::default();
23 transport_config.keep_alive_interval(Some(Duration::from_secs(2)));
24 transport_config.max_idle_timeout(Some(VarInt::from_u32(10_000).into()));
25 server_config.transport = Arc::new(transport_config);
26 let endpoint = quinn::Endpoint::server(server_config, opt.listen)?;

```

Listing 12: Configurazione del server QUIC

Mentre in listato 13, è mostrato il loop effettuato per servire le connessioni. In particolare, per ogni richiesta in arrivo, il server verifica che si tratta di un pacchetto QUIC. In caso positivo, procede a servirla trasformandola in una sessione QUIC.

```

1 // ...codice omissso
2 // 2. Accept new quic connections and spawn a new task to handle them
3 while let Some(new_conn) = endpoint.accept().await {
4     trace_span!("New connection being attempted");
5     let nc = nc.clone();
6
7     tokio::spawn(async move {
8         match new_conn.await {
9             Ok(conn) => {
10                 if is_http3(&conn) {
11                     info!("new http3 established");
12                     let h3_conn = sec_http3::server::builder()
13                         .enable_webtransport(true)
14                         .enable_connect(true)
15                         .enable_datagram(true)
16                         .max_webtransport_sessions(1)
17                         .send_grease(true)
18                         .build(h3_quinn::Connection::new(conn))
19                         .await
20                         .unwrap();
21                     let nc = nc.clone();
22                     if let Err(err) = handle_h3_connection(h3_conn, nc).await {
23                         error!("Failed to handle connection: {err:?}");
24                     }
25                 } else {
26                     info!("new quic established");
27                     let nc = nc.clone();
28                     if let Err(err) = handle_quic_connection(conn, nc).await {
29                         error!("Failed to handle connection: {err:?}");
30                     }
31                 }
32             }
33             Err(err) => {
34                 error!("accepting connection failed: {err:?}", err);
35             }
36         }
37     });

```

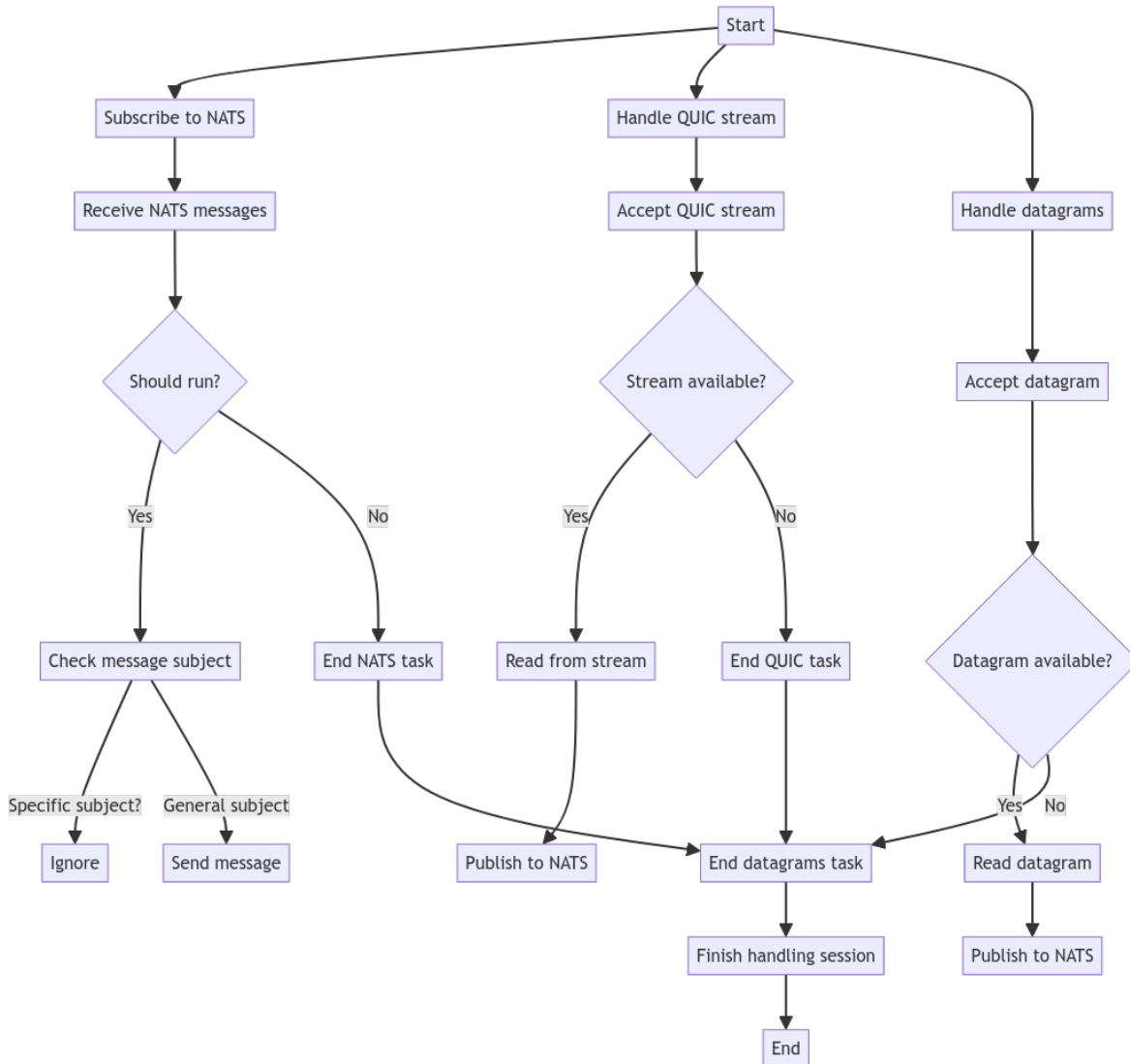


Figure 6: Diagramma di flusso della funzione `handle_session`

38 }

Listing 13: Loop del server UDP

Infine, una volta ottenuta una connessione QUIC, il server chiama la funzione `handle_session` che crea tre task concorrenti:

- ***nats_task***: si mette in ascolto per tutti i pacchetti in arrivo per il topic creato per la room in cui si è entrati e li manda sulla sessione corrente evitando di effettuare loop;
- ***quic_task***: per ogni flusso unidirezionale (altri utenti) che arriva dalla sessione crea un'ulteriore task da cui riceve pacchetti e li inoltra su Nats;
- ***datagram_task***: riceve i pacchetti dell'utente host della sessione e li inoltra su nats;

QUIC client-side I browser implementano il protocollo QUIC attraverso la WebTransport API. Come già spiegato in precedenza, per accedere alla classe WebTransport in un'applicazione WebAssembly è necessario effettuare il mapping delle funzioni nella macchina WASM. In Yew, la connessione è stata realizzata con la libreria *yew-webtransport*. In particolare, è stato riutilizzato il client *videocall-client* trovato in (SecurityUnion, 2024). Di seguito sono riportati i cambiamenti rispetto alla configurazione iniziale.

Essendo utilizzato in un contesto in cui la sorgente audio è la voce umana, il client di default invia audio mono con un bitrate di 8KHz utilizzando il codec OPUS. La prima modifica è stata quella di rendere l'audio stereo e scegliere un bitrate accettabile che può andare da 320kb/s a 450kb/s (basandosi su (Valin, Vos, & Terriberry, 2012)). La configurazione del codec è riportata in listato 14. Successivamente, per rendere effettive le modifiche dal punto di vista dei dispositivi si è intervenuto sia sull'encoder che sul decoder (situati rispettivamente nei file *microphone_encoder.rs* e *config.rs*), come mostrato listato 15 in e listato 16. Infine, per realizzare il componente AudioVisualizer il cui scopo è quello di fornire un feedback visivo a chi sta suonando è stata aggiunta una callback per catturare l'evento in cui l'utente cambia la sorgente del dispositivo (listato 17).

```
1 pub static AUDIO_CODEC: &str = "opus";
2
3 // ...altre costanti
4
5 pub const AUDIO_CHANNELS: u32 = 2u32;
6 pub const AUDIO_SAMPLE_RATE: u32 = 48000u32;
7 pub const AUDIO_BITRATE: f64 = 320000f64 //320kb/s;
8
9 // ...altre costanti
```

Listing 14: Configurazione utilizzata per trasmettere l'audio

```
1 let mut constraints = MediaStreamConstraints::new();
2 let mut media_info = web_sys::MediaTrackConstraints::new();
3 media_info.device_id(&device_id.into());
4 media_info.channel_count(&"2".into());
5 media_info.auto_gain_control(&"false".into());
6 media_info.echo_cancellation(&"false".into());
7 media_info.noise_suppression(&"false".into());
8 //...altro codice
9 let mut audio_encoder_config = AudioEncoderConfig::new(AUDIO_CODEC);
10 audio_encoder_config.bitrate(AUDIO_BITRATE);
11 audio_encoder_config.sample_rate(AUDIO_SAMPLE_RATE);
12 audio_encoder_config.number_of_channels(AUDIO_CHANNELS);
```

Listing 15: Configurazione audio stereo per l'encoder OPUS

```
1 pub fn configure_audio_context(
2     audio_stream_generator: &MediaStreamTrackGenerator,
3 ) -> anyhow::Result<AudioContext> {
4     let js_tracks = Array::new();
5     js_tracks.push(audio_stream_generator);
6     let media_stream = MediaStream::new_with_tracks(&js_tracks).unwrap();
7     let mut audio_context_options = AudioContextOptions::new();
8     audio_context_options.sample_rate(AUDIO_SAMPLE_RATE as f32);
9     let audio_context = AudioContext::new_with_context_options(&audio_context_options)
10         .unwrap();
11     let gain_node = audio_context.create_gain().unwrap();
12     gain_node.set_channel_count(AUDIO_CHANNELS);
13     let source = audio_context
14         .create_media_stream_source(&media_stream)
```

```

15     .unwrap();
16     source.set_channel_count(AUDIO_CHANNELS);
17     let _ = source.connect_with_audio_node(&gain_node).unwrap();
18     let _ = gain_node
19         .connect_with_audio_node(&audio_context.destination())
20         .unwrap();
21     Ok(audio_context)
22 }

```

Listing 16: Configurazione audio stereo per il decoder OPUS

```

1 pub enum Msg {
2     WsAction(WsAction),
3     MeetingAction(MeetingAction),
4     OnPeerAdded(String),
5     OnFirstFrame((String, MediaType)),
6     // aggiunta callback per ricavare l'id della sorgente audio
7     OnChangeMic(String),
8 }
9 // file host.rs
10 fn update(&mut self, ctx: &Context<Self>, msg: Self::Message) -> bool {
11     match msg {
12         // invocazione della callback dichiarata in precedenza
13         Msg::AudioDeviceChanged(audio) => {
14             if self.microphone.select(audio.clone()) {
15                 let link = ctx.link().clone();
16                 let timeout = Timeout::new(1000, move || {
17                     link.send_message(Msg::EnableMicrophone(true));
18                 });
19                 timeout.forget();
20             }
21             self.on_audio_src_changed.emit(audio);
22             false
23         }
24     }
25 }

```

Listing 17: Modifica del file host.rs per invocare la callback

1.1.4 Serializzazione binaria: protocol buffer

Protocol buffer (*Protobuf*, n.d.) è un formato di serializzazione binario sviluppato da Google. All'interno del progetto viene utilizzato per serializzare i pacchetti dei flussi multimediali che l'utente genera. La scelta di questo formato di serializzazione binaria deriva dalle necessità di effettuare trasmissione a bassa latenza. Infatti, come mostrato in (Popić, Pezer, Mrazovac, & Teslić, 2016), i messaggi serializzati in protocol buffer tendono a essere molto più piccoli rispetto a quelli JSON e quindi si ha una trasmissione più efficiente. Protocol buffer prevede la dichiarazione dei tipi che si andranno a serializzare e deserializzare in un linguaggio descrittivo in file *.proto*. In listato 18, è mostrato il file utilizzato per serializzare e deserializzare un pacchetto multimediale.

```

1 syntax = "proto3";
2
3 message MediaPacket {
4     enum MediaType {
5         VIDEO = 0;
6         AUDIO = 1;
7         HEARTBEAT = 3;
8     }

```

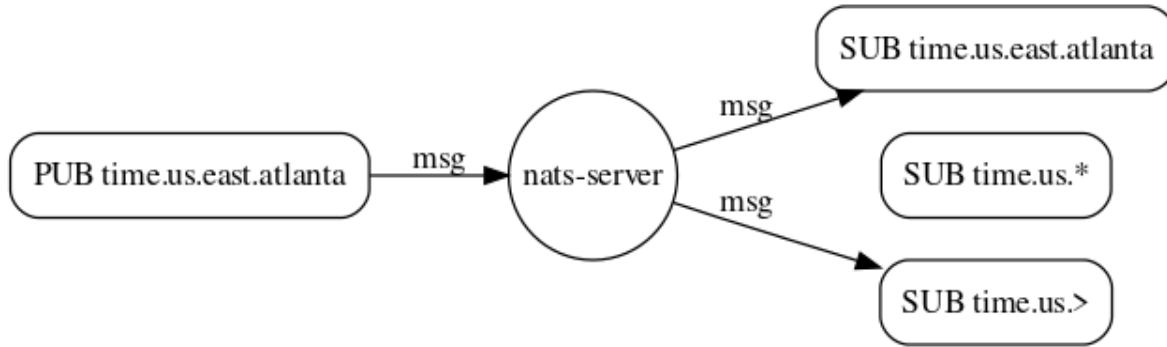


Figure 7: Funzionamento publish-subscriber nats

```

9  MediaType media_type = 1;
10  string email = 2;
11  bytes data = 3;
12  string frame_type = 4;
13  double timestamp = 5;
14  double duration = 6;
15  AudioMetadata audio_metadata = 7;
16  VideoMetadata video_metadata = 8;
17 }
18
19 message AudioMetadata {
20     string audio_format = 1;
21     uint32 audio_number_of_channels = 2;
22     uint32 audio_number_of_frames = 3;
23     float audio_sample_rate = 4;
24 }
25
26 message VideoMetadata {
27     uint64 sequence = 1;
28 }

```

Listing 18: Pacchetto utilizzato per trasmettere flussi multimediali

Protocol buffer genera, a valle del file `.proto`, i tipi corrispondenti in tutti i linguaggi di programmazione supportati mediante un compilatore. Ad esempio, nella cartella `protobuf`, viene utilizzato il seguente comando per compilare i file

```
protoc --rust_out=build/rust types/*.proto
```

1.1.5 Broadcast: Nats

Il broadcasting avviene attraverso l'utilizzo di Nats (*Nats*, n.d.): una piattaforma che implementa il pattern publish-subscribe (Figura 7). Nats consente di creare topic a cui sono associati gruppi di code e risulta particolarmente utile nel caso in cui si voglia mantenere la tracciabilità dei pacchetti trasmessi ed, allo stesso tempo evitare loop. Infatti, per ogni connessione ci si sottoscrive ad un topic del tipo `room.:room-id.*` con un gruppo `room.:room-id.:username`. In questo modo, usando il campo `subject` del *MediaPacket* mostrato in listato 18 si evitano i loop. La funzione `handle_session`, in listato 19, mostra la sottoscrizione mediante un client di Nats alla room.

```

1  async fn handle_session<C>(<
2      session: WebTransportSession<C, Bytes>,
3      username: &str,
4      lobby_id: &str,
5      nc: async_nats::client::Client,

```

```

6 ) -> anyhow::Result<()> where
7 // ... vincoli sul tipo generico C
8 {
9     let subject = format!("room.{}.*", lobby_id).replace(' ', "_");
10    let specific_subject = format!("room.{}.{}", lobby_id, username).replace(' ', "_");
11    let mut sub = match nc
12        .queue_subscribe(subject.clone(), specific_subject.clone())
13        .await
14    {
15        Ok(sub) => {
16            info!("Subscribed to subject {}", subject);
17            sub
18        }
19        Err(e) => {
20            let err = format!("error subscribing to subject {}: {}", subject, e);
21            error!("{}", err);
22            return Err(anyhow!(err));
23        }
24    };
25    // ... codice oresso
26    tokio::spawn(async move {
27        while let Some(msg) = sub.next().await {
28            if !should_run.load(Ordering::SeqCst) {
29                break;
30            }
31            if msg.subject == specific_subject_clone {
32                continue;
33            }
34            let session = session.read().await;
35            if msg.payload.len() > 400 {
36                let stream = session.open_uni(session_id).await;
37                tokio::spawn(async move {
38                    match stream {
39                        Ok(mut uni_stream) => {
40                            if let Err(e) = uni_stream.write_all(&msg.payload).await {
41                                error!("Error writing to unidirectional stream: {}", e);
42                            }
43                        }
44                        Err(e) => {
45                            error!("Error opening unidirectional stream: {}", e);
46                        }
47                    }
48                });
49            } else if let Err(e) = session.send_datagram(msg.payload) {
50                error!("Error sending datagram: {}", e);
51            }
52        }
53    })
54    // ... codice oresso
55 }

```

Listing 19: Broadcast di pacchetti multimediali in una sessione WebTransport

1.2 Servizio REST

Oltre alla parte QUIC, l'applicazione fornisce un servizio REST per interagire con le entità presentate nel diagramma E/R in Figura 2. Il testing di questo servizio è stato effettuato mediante la creazione di una collection Postman. In Figura 8 e Figura 9, sono mostrate le operazioni di creazione e ricerca di una stanza.

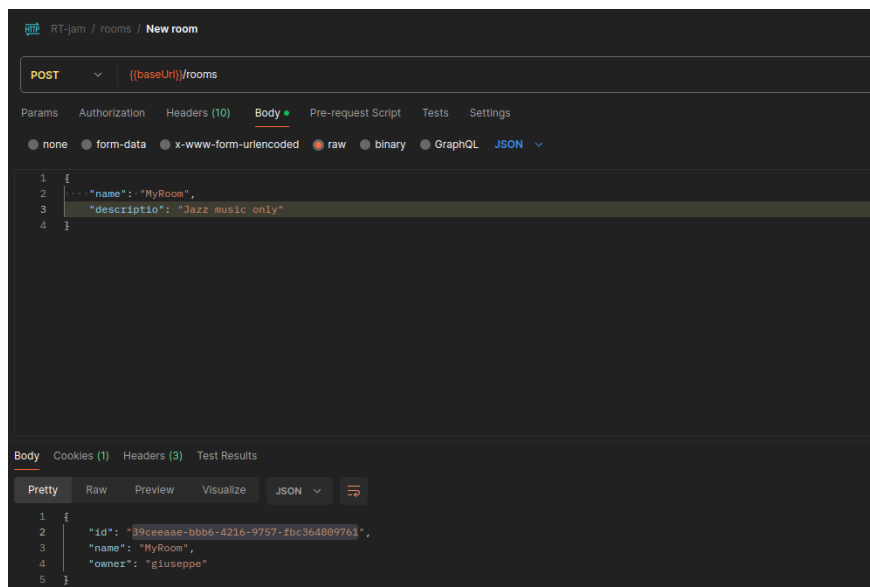


Figure 8: API per la creazione di una stanza

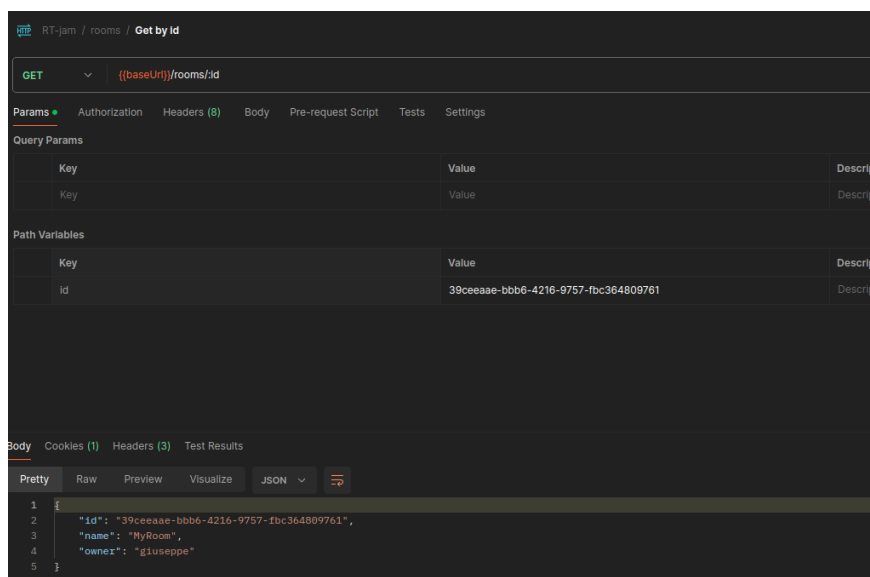


Figure 9: API per la ricerca di una stanza dato l'identificativo

2 Funzionamento dell'applicazione

2.1 Autenticazione e autorizzazione

2.1.1 Creazione account

Il flusso di registrazione account è illustrato in Figura 10.

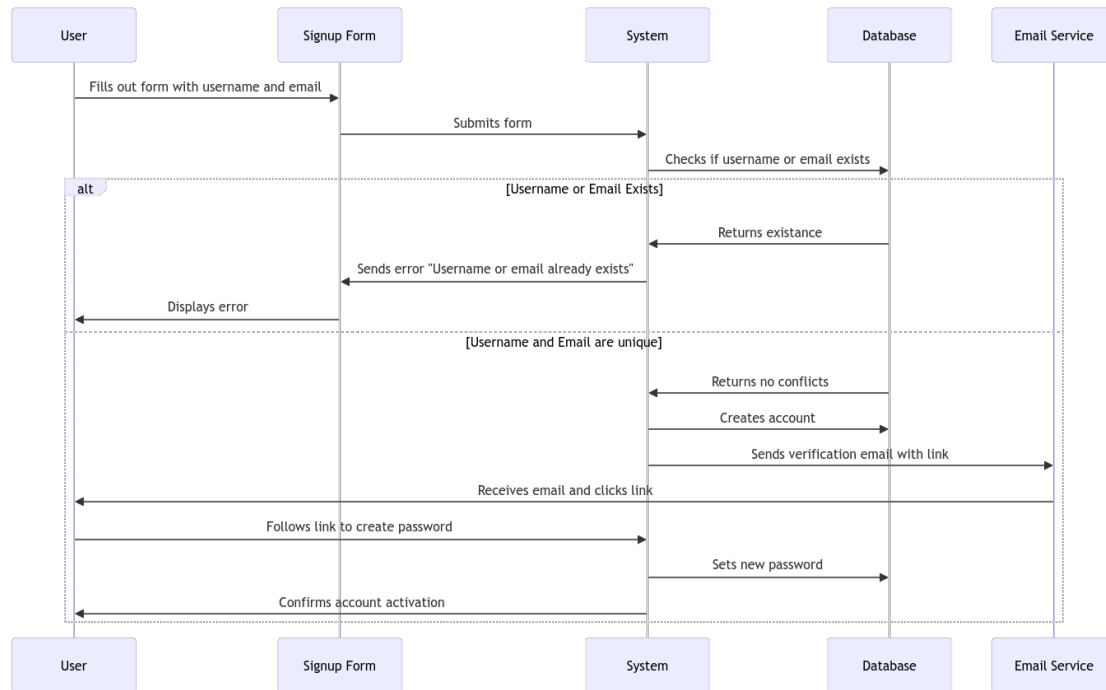
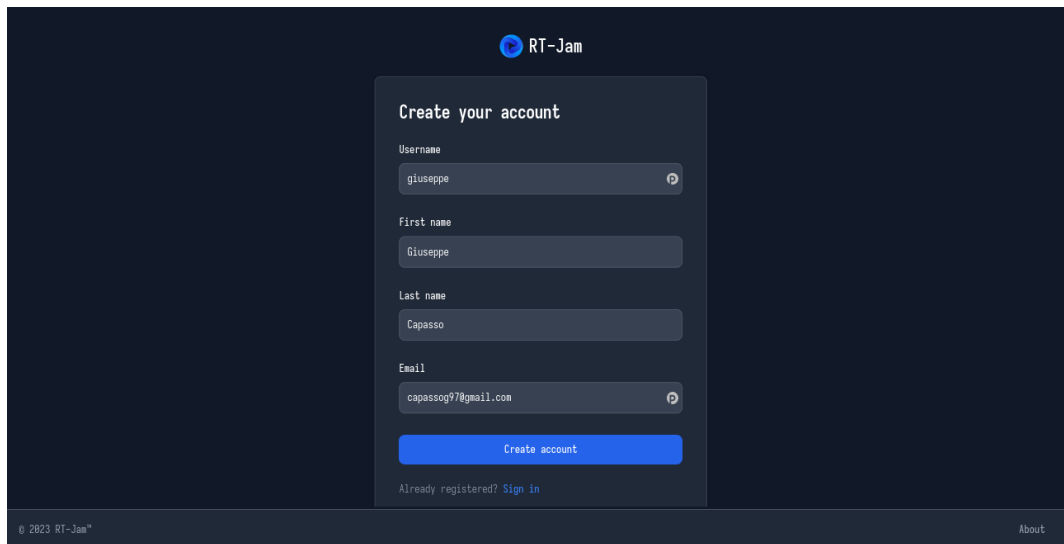


Figure 10: Sequence diagram processo di registrazione

L'applicazione prevede che un utente possa registrarsi previa compilazione di un form in cui indica la sua email (Figura 11). All'invio del form, viene inviata una mail (Figura 12) attraverso la quale l'utente potrà attivare il proprio account impostando una password (Figura 13).

Password hashing La password impostata dall'utente viene salvata sottoforma di hash. Per l'applicazione è stato scelto l'algoritmo *Argon2*. Per questo motivo, è stata implementata la funzionalità *password dimenticata* (Figura 14) che invia un email simile a quella in Figura 12 per reimpostare la password.



The screenshot shows a dark-themed web interface for 'RT-Jam'. At the top center is the 'RT-Jam' logo. Below it is a 'Create your account' form. The form contains four input fields: 'Username' with the value 'giuseppe', 'First name' with 'Giuseppe', 'Last name' with 'Capasso', and 'Email' with 'capassog77@gmail.com'. Each input field has a small circular icon to its right. Below the fields is a blue 'Create account' button. At the bottom of the form, there is a link: 'Already registered? Sign in'. The footer of the page shows '© 2023 RT-Jam*' on the left and 'About' on the right.

RT-Jam

Create your account

Username
giuseppe

First name
Giuseppe

Last name
Capasso

Email
capassog77@gmail.com

Create account

Already registered? [Sign in](#)

© 2023 RT-Jam* About

Figure 11: Pagina di creazione account

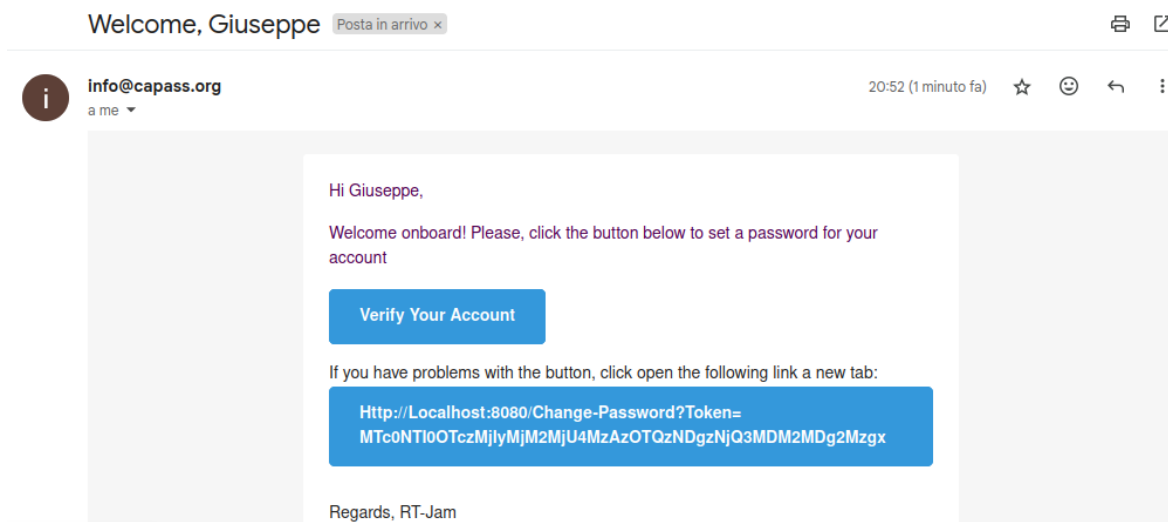


Figure 12: Email di conferma account

The screenshot shows a dark-themed web interface for RT-Jam. At the top center is the RT-Jam logo, which consists of a blue play button icon followed by the text "RT-Jam". Below the logo is a light gray rounded rectangle containing the title "Set up password". Under this title are two password input fields. The first field is labeled "Password" and the second is labeled "Confirm password". Both fields contain masked text (dots) and have icons for toggling password visibility (an eye) and a help icon (a question mark). Below the input fields is a blue button with the text "Change password". At the bottom of the page, there is a dark footer bar with the text "© 2023 RT-Jam" on the left and a link labeled "About" on the right.

Figure 13: Attivazione account attraverso l'impostazione della password

The screenshot shows a dark-themed web interface for RT-Jam. At the top center is the RT-Jam logo, which consists of a blue play button icon followed by the text "RT-Jam". Below the logo is a light gray rounded rectangle containing the title "Reset password form". Under this title is an email input field labeled "Email" with a placeholder text "Email" and a help icon (a question mark). Below the input field is a blue button with the text "Send reset link". At the bottom of the page, there is a dark footer bar with the text "© 2023 RT-Jam" on the left and a link labeled "About" on the right.

Figure 14: Pagina reset password

2.1.2 Login

Infine, l'utente potrà accedere alla piattaforma attraverso un form di login (Figura 15).

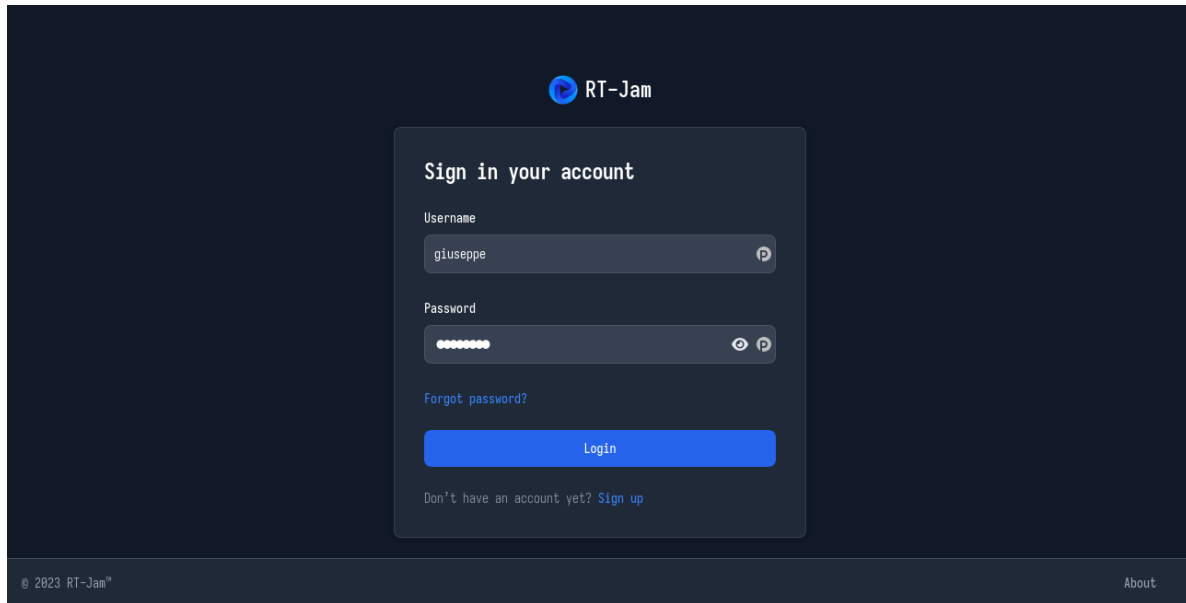


Figure 15: Pagina di login dell'applicazione

All'accesso viene creata una sessione della durata di 7 giorni con gli attributi *SameSite* e *HttpOnly*.

Cookies firmati La procedura di login produce un token che viene salvato come identificativo di sessione nell'apposita tabella (E/R). Il token viene prodotto con una funzione che implementa l'algoritmo *ChaCha* (un generatore di numeri casuali sicuro dal punto di vista crittografico) (*ChaCha implementation*, n.d.). Inoltre, i cookie vengono firmati con una chiave fornita come variabile d'ambiente. Per renderli disponibili a tutta l'applicazione è stato implementato un estrattore che prende i cookie dall'header della richiesta HTTP e lo decodifica (listato 21).

```
1 #[async_trait]
2 impl<S: Send + Sync> FromRequestParts<S> for Cookies<'_> {
3     type Rejection = Error;
4
5     async fn from_request_parts(parts: &mut Parts, state: &S) -> Result<Self, Error> {
6         debug!("{:12}_Signed_Cookies", "EXTRACTOR");
7         let key = SESSION_COOKIE_KEY
8             .get()
9             .expect("SESSION_KEY_IS_NOT_INITIALIZED");
10        let cookies = tower_cookies::Cookies::from_request_parts(parts, state)
11            .await
12            .unwrap()
13            .signed(key);
14        Ok(Cookies(cookies))
15    }
16 }
```

Listing 20: Extractor implementato per decodificare i cookie ad ogni richiesta

Implementazione rotte sicure Le pagine a cui bisogna accedere solo se si è effettuato l'accesso implementano un sistema di guardia in cui viene renderizzato il loro contenuto solo se la chiamata all'endpoint */api/auth/me* non restituisce errori.

```
1
2 let (store, dispatch) = use_store::<Store>();
3 use_effect(move || {
4   spawn_local(async move {
5     match Request::get("/api/auth/me").send().await {
6       Ok(res) => {
7         if res.ok() {
8           let user = res.json::<UserResponse>().await.unwrap();
9           dispatch.reduce_mut(move |s| s.auth_user = Some(user.into()));
10        } else {
11          navigator.replace(&Route::Login);
12        }
13      }
14      // network error
15      Err(err) => {
16        log_1(&err.to_string().into());
17      }
18    };
19  });
20 });
```

Listing 21: Extractor implementato per decodificare i cookie ad ogni richiesta

2.2 Piattaforma

Home page La pagina principale Figura 16 presenta un form con il quale si può creare una nuova stanza o entrare in una già esistente.

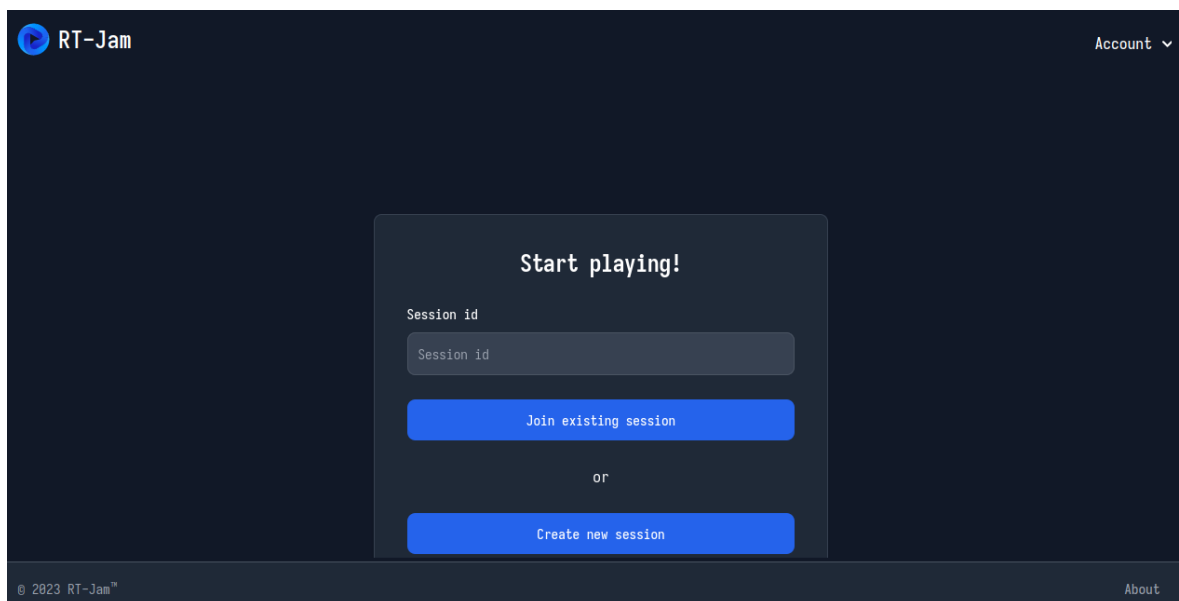


Figure 16: Home page dell'applicazione

Creata una sessione, si arriva alla pagina di sessione dove viene effettuata la connessione al server QUIC (Figura 17). La pagina supporta la trasmissione audio/video e consente agli utenti di scegliere le sorgenti multimediali.

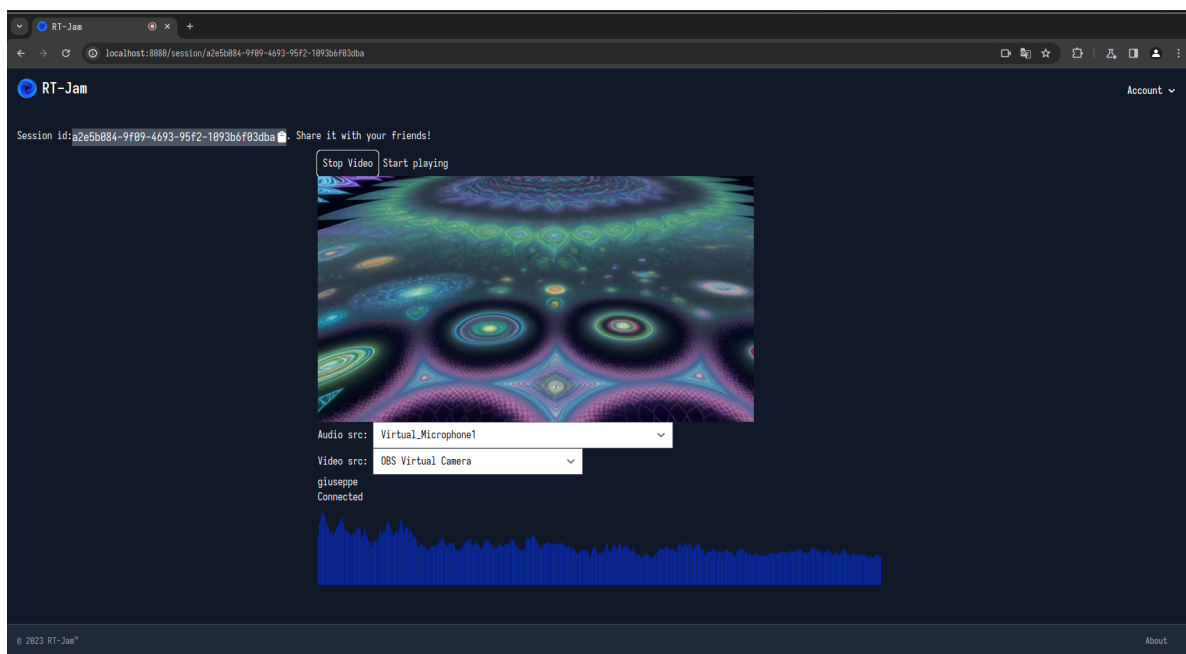


Figure 17: Sessione con trasmissione audio e video

3 Deploy

L'applicazione è gestita con un Makefile dotato di un help menu:

```
make help
Usage:
  up           executes the application using docker-compose
  down        deletes docker containers
  build-images build docker images
  build       statically build frontend and backend
  dev         creates nats and postgres container; executes backend and frontend locally
  help        prints this help message
```

3.1 Bare metal

L'installazione da sorgente richiede la toolchain rust che può essere ottenuta dal sito ufficiale. Una volta installato il compilatore rustc, bisogna installare il target per eseguire applicazioni in WebAssembly con il comando:

```
rustup target add wasm32-unknown-unknown
```

e successivamente si può installare il bundler Trunk con il comando:

```
cargo install --locked trunk
```

Se sono state effettuate modifiche alle dichiarazioni nella cartella *protocol buffer*, ricompilare i file con i comandi:

```
cd protobuf
make generate_rust
```

Infine, è possibile eseguire il comando

```
make build
```

per compilare staticamente *frontend* e *backend* in modalità *release*. Il processo di build produrrà due artefatti:

- Server: nella cartella *target/release* ci sarà un file chiamato ***backend***
- Frontend: nella cartella *frontend/dist* ci sarà il bundle dell'applicazione webassembly che potrà essere servita un qualsiasi webserver.

Terminato il processo di compilazione, bisogna eseguire almeno un'istanza di Nats e Postgres. Il modo più semplice di installare queste dipendenze è di farlo attraverso Docker:

```
docker run -d --network=host nats
docker run -d --network=host -e POSTGRES_PASSWORD=postgres postgres
```

In questo momento è necessario configurare l'ambiente con le variabili d'ambiente necessarie al funzionamento del backend. Un esempio di configurazione è la seguente:

```
export RTJAM_DATABASE_URL="postgres://postgres:postgres@localhost/postgres?sslmode=disable"
export RTJAM_LISTEN_ADDRESS="0.0.0.0:3000"
export RTJAM_SESSION_KEY="mN1GR7dsQ+Bj8NFIA+n/uvSbBcdyvHnVdFuJSJrQJ3g2/8gGYaATt3Wv7j3xKpD07652no/eddRdD"
export RTJAM_NATS_URL="localhost:4222"
export RTJAM_SMTP_HOST="mail.privateemail.com"
export RTJAM_SMTP_PORT="587"
export RTJAM_SMTP_USER="info@capass.org"
export RTJAM_SMTP_PASSWORD=""
export RTJAM_SMTP_FROM="info@capass.org"
```

```
export RTJAM_APP_URL="http://localhost:8080"
export RTJAM_WEBTRANSPORT_ADDRESS="0.0.0.0:4433"
export RTJAM_CERT_PATH="backend/certs/localhost.dev.pem"
export RTJAM_KEY_PATH="backend/certs/localhost.dev.key"
```

Configurato l'ambiente, è possibile eseguire il backend.

3.2 Docker

La modalità di installazione con Docker prevede l'esecuzione del comando:

```
make build-images up
```

Analogamente al caso *bare metal*, bisogna configurare l'ambiente con le variabili corrette nel file *docker-compose.json*.

3.3 Test dell'applicazione

Al momento dello sviluppo solo i browser chromium-based implementano l'API WebTransport: l'applicazione quindi **non** funziona su altri browser. Inoltre, il protocollo QUIC rende obbligatorio l'utilizzo del protocollo TLS. In questa repository, sono già forniti dei certificati generati con *openssl* per eseguire l'applicazione localmente.

Attraverso lo script *launch_chrome.sh* viene eseguita un'istanza di chrome con parametri disponibili solo da riga di comando che bypassano la verifica del certificato e forzano l'utilizzo di localhost per connettersi al protocollo QUIC.

```
google-chrome --origin-to-force-quic-on=127.0.0.1:4433 \
--ignore-certificate-errors-spki-list="$SPKI" \
--enable-logging --v=1
```

I certificati sono stati generati con i seguenti comandi:

```
openssl req -x509 -newkey rsa:2048 -keyout "backend/certs/localhost.dev.key" \
-out "backend/certs/localhost.dev.pem" -days 365 -nodes -subj "/CN=127.0.0.1"

openssl x509 -in "backend/certs/localhost.dev.pem" -outform der \
-out "backend/certs/localhost.dev.der"

openssl rsa -in "backend/certs/localhost.dev.key" -outform DER \
-out "backend/certs/localhost.dev.key.der"
```

Extra: creazione sorgenti audio virtuali (linux only) In Linux, è possibile effettuare il routing di dispositivi audio attraverso il modulo *null-sink* (l'audio che entra in questo nodo non viene riprodotto da nessun dispositivo). Per creare un modulo null-sink eseguire:

```
pactl load-module module-null-sink sink_name=virtmic1 \
sink_properties=device.description=Virtual_Microphone_Sink1
```

L'uscita di un modulo null-sink può essere catturata attraverso il modulo *remap* che effettua un reindirizzamento verso un ingresso a scelta

```
pactl load-module module-remap-source master=virtmic1.monitor source_name=virtmic1 \
source_properties=device.description=Virtual_Microphone1
```

Attraverso, un'interfaccia grafica come *pavucontrol* è possibile scegliere, per tutte le applicazioni, il sink (Figura 18).

Infine, basta scegliere la sorgente audio dall'applicazione, come mostrato in Figura 19.

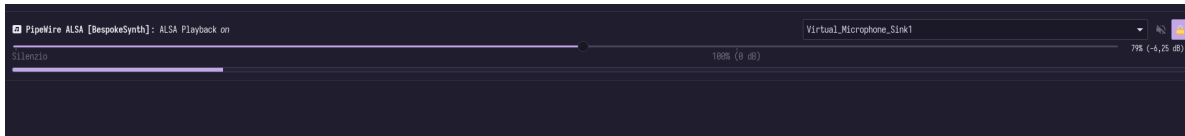


Figure 18: Routing dell'uscita di un'applicazione attraverso l'input creato

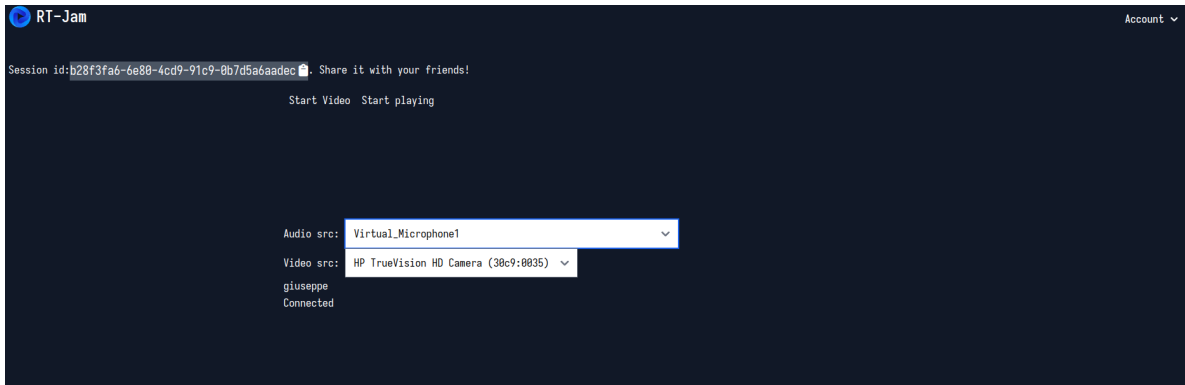


Figure 19: Selezione della sorgente audio nell'applicazione

Performance L'applicazione è intesa da utilizzare con dispositivi hardware (interfacce audio) con cui effettuare gran parte del workload con cui effettuare DSP. La demo con dispositivi virtuali serve a dimostrare il funzionamento, ma è possibile riscontrare bassa qualità dell'audio e fenomeni di glitch/distorsioni. Per rilassare lo stress sulla CPU è possibile aumentare la dimensione del buffer dell'encoder (aumentando una possibile latenza). Il buffer (`microphone_encoder.rs` della libreria `videocall-client` listato 22) può essere aumentato in accordo con le dimensioni dell'head messo a disposizione dalla macchina WebAssembly (circa 500.000 byte) oltre al quale si ottiene un errore di accesso in memoria.

```

1 pub fn start(&mut self) {
2     let device_id = if let Some(mic) = &self.state.selected {
3         mic.to_string()
4     } else {
5         return;
6     };
7     let client = self.client.clone();
8     let userid = client.userid().clone();
9     let aes = client.aes();
10    let audio_output_handler = {
11        let mut buffer: [u8; 500000] = [0; 500000];
12        let mut sequence = 0;
13        Box::new(move |chunk: JsValue| {
14            let chunk = web_sys::EncodedAudioChunk::from(chunk);
15            let packet: PacketWrapper =
16                transform_audio_chunk(&chunk, &mut buffer, &userid, sequence, aes.clone());
17            client.send_packet(packet);
18            sequence += 1;
19        })
20    };
21    let EncoderState {
22        destroy,

```

```
23     enabled,  
24     switching,  
25     ..  
26 } = self.state.clone();
```

Listing 22: Dimensione del buffer per l'encoder OPUS

References

- ChaCha implementation*. (n.d.). https://rust-random.github.io/rand/rand_chacha/struct.ChaCha8Rng.html. (Accessed: 2024-04-10)
- Marx, R., De Decker, T., Quax, P., & Lamotte, W. (2020). Resource multiplexing and prioritization in http/2 over tcp versus http/3 over quic. In *Web information systems and technologies: 15th international conference, webist 2019, vienna, austria, september 18–20, 2019, revised selected papers 15* (pp. 96–126).
- Nats*. (n.d.). <https://nats.io/>. (Accessed: 2024-04-10)
- Popić, S., Pezer, D., Mrazovac, B., & Teslić, N. (2016). Performance evaluation of using protocol buffers in the internet of things communication. In *2016 international conference on smart systems and technologies (sst)* (pp. 261–265).
- Protobuf*. (n.d.). <https://protobuf.dev/>. (Accessed: 2024-04-10)
- Rossberg, A. (2021). Webassembly specification. *WebAssembly Community Group*, 2.
- SecurityUnion. (2024). *Videocallrs*. <https://github.com/security-union/videocall-rs>. GitHub.
- Valin, J.-M., Vos, K., & Terriberry, T. B. (2012, September). *Definition of the Opus Audio Codec* (No. 6716). RFC 6716. RFC Editor. Retrieved from <https://www.rfc-editor.org/info/rfc6716> doi: 10.17487/RFC6716