POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Estensione delle funzionalità di streaming HLS per server Nginx e test di scalabilità

Relatore
prof. Antonio Servetti

Candidato
Davide Melchionda
matricola: 231782

Aprile 2018

Sommario

La tesi si concentra sull'implementazione di una soluzione efficiente e scalabile per distribuire contenuti multimediali tramite il protocollo di streaming HLS. In quest'ottica è stato sviluppato un modulo HLS per il web server Nginx . Il modulo in questione offre funzionalità di Network-DVR, consentendo a un client di effettuare richieste di playlist personalizzate, che godano cioè di specifiche caratteristiche a seconda della richiesta effettuata (es. richiesta di una playlist LIVE o EVENT). La scelta della tecnologia è legata all'ampia diffusione del web server Nginx e alla grande scalabilità ed efficienza nel servire contenuti che lo caratterizzano. Infine la tesi si concentra sulla valutazione delle performance della soluzione implementata, utilizzando tool di test lato server e lato client e confrontando i risultati ottenuti con una soluzione simile (ma proprietaria) implementata con tecnologie differenti.

Indice

Elenco delle tabelle

Elenco delle figure

Parte I

Introduzione allo streaming HTTP-based

Capitolo 1

Streaming HTTP-based

La distribuzione di contenuti multimediali sul Web sta acquisendo sempre maggiore rilevanza tanto per business di grandi e piccole dimensioni quanto per creatori di contenuti autonomi. In questo contesto, l'interesse verso le tecnologie di streaming va crescendo e con esso gli investimenti in ricerca e infrastrutture. La possibilità di riprodurre audio e video su dispositivi di natura molto diversa in termini di capacità computazionale e di banda di rete disponibile ha incentivato la diffusione di protocolli in grado di supportare tecniche di *Adaptive bitrate streaming*, strategie che consentono ai client di richiedere (in funzione delle proprie esigenze) versioni diverse dello stesso contenuto codificate in stream a bitrate differenti. Un'ulteriore spinta in questa direzione è stata data da HTML5, che consente di gestire nativamente audio e video nel browser tramite tag dedicati al multimedia. In HTML5 lo streaming passa su HTTP, sfruttando protocolli *HTTP-based*. L'utilizzo di questi protocolli ha profondamente modificato il modo di concepire lo streaming (storicamente fatto su UDP), proponendo soluzioni che sono in grado di lavorare in modo efficiente su TCP e che offrono, in cambio dei ritardi inevitabilmente introdotti dal trasporto affidabile, l'opportunità di sfruttare pienamente gli investimenti fatti per lavorare sul Web (es. CDN). Fare streaming su HTTP vuol dire poter utilizzare dei comuni web-sever per la distribuzione di flussi multimediali, una soluzione estremamente economica rispetto all'alternativa di acquistare media-server dedicati.

I protocolli di streaming HTTP-based prevedono di dividere il contenuto multimediale in frammenti di durata limitata, accessibili separatamente come comuni risorse web e dunque identificate da uno URL. La relazione (puramente logica) tra questi frammenti è realizzata esponendo un'ulteriore

risorsa, detta playlist o indice, che elenca gli URL dei frammenti appartenenti a uno specifico stream. La concatenazione dei frammenti nell'ordine corretto consente al player client di riprodurre lo stream originale. La catena di streaming sulla quale si basano questi protocolli prevede dunque tre entità: un *encoder* che codifichi lo stream in un formato opportuno, uno *stream segmenter*, che crei i frammenti segmentando il flusso originale e generi la playlist e un *distributor*, che è appunto un web-server, dal quale si può accedere alle risorse generate. Al momento non esiste un protocollo comunemente accettato in tutti gli ambienti e, come spesso accade al nascere di nuove tecnologie, diversi attori hanno proposto soluzioni differenti. Dei quattro protocolli più noti, quelli che sembrano riscuotere il maggior successo sono HTTP Live Streaming (HLS) di Apple e Dynamic Adaptive Streaming over HTTP (DASH). Sebbene quest'ultimo sia nato da uno sforzo di standardizzazione mirato a identificare una soluzione definitiva per lo streaming su HTTP, il fatto che HLS sia l'unico protocollo supportato nativamente dai dispositivi iOS ha fatto sì che esso sia di fatto quello più diffuso, così che risulta difficile immaginare una sua scomparsa dal mercato.

In questo scenario, le alternative per fare streaming si traducono nell'affidarsi a un media-server (sistemi pensati per la distribuzione di video-on-demand) o nell'utilizzare una soluzione basata su un comune web-server. Esistono molti media-server commerciali e open-source che supportano i più recenti protocolli HTTP-based e che aggiungono su di essi funzionalità aggiuntive utili in contesti applicativi come quelli dello streaming. Scopo di questo lavoro di tesi è stato quello di estendere una soluzione di streaming HLS basata sull'utilizzo del web-server Nginx introducendo funzionalità di Network DVR (Digital Video Recorder) e di confrontare le prestazioni del sistema risultante con quelle offerte da Wowza, uno dei media-server commerciali più diffusi nell'ambito dello streaming audio.

1.1 Evoluzione delle tecnologie e dei protocolli di streaming

1.1.1 Tecnologie di streaming

Uno *Streaming Media* è un contenuto multimediale che viene ricevuto in modo continuo da un client e presentato all'utente prima ancora che il trasferimento sia stato completato. Tipicamente tale client è un player in grado di riprodurre tale contenuto.

Il successo dello streaming multimediale è arrivato verso il finire degli anni '90 e nei primi 2000.A spiegare tale successo c'è principalmente la crescente diffusione dell'accesso a internet tramite connessioni a banda ultra-larga nell'ultimo miglio. A questo si è aggiunto il crescente utilizzo di formati e protocolli standard per la comunicazione su Internet, come HTML ed HTTP, che hanno reso Internet realmente accessibile.

Uno dei primi prodotti che hanno consentito lo streaming multimediale è stato ActiveMovie, sviluppato da Microsoft nel 1996, che offriva un runtime component integrato in Microsoft Internet Explorer per scaricare e riprodurre video e audio in molti dei formati disponibili all'epoca sul web. [13] Nello stesso periodo è nato anche RealPlayer, mentre qualche anno dopo, nel 1999, Apple ha integrato la possibilità di utilizzare formati e protocolli dedicati allo streaming nel player QuickTime 4. [14]

In questa fase la competizione tra le diverse tecnologie di streaming e la mancanza di standardizzazione obbligava gli utenti a installare un numero elevato di applicativi differenti per garantire compatibilità con i diversi ambienti. A partire dal 2000 si è sviluppato interesse verso queste tecnologie intese come canali di comunicazione da parte di diversi tipi di business verso i potenziali clienti. Il webcasting ha assunto un ruolo fondamentale nelle strategie comunicative delle aziende e questo ha spinto a cercare di individuare un formato di streaming comune e unificato. La grande diffusione di player Flash-based e lo sviluppo di un formato di streaming tramite Flash ha fatto sì che questa tecnologia fosse quella più utilizzata da attori di piccole e grandi dimensioni (come YouTube), divenendo uno standard de facto.

Nato per un contesto desktop, l'intera piattaforma Adobe Flash ha avuto difficoltà ad adeguarsi alla transizione verso il modello di fruizione dei contenuti incentrato sui device mobili. A causa di ciò, delle numerose vulnerabilità di sicurezza e del fatto che si tratti di una closed-platform, Flash è stato progressivamente abbandonato a favore di soluzioni più moderne. HTML5 ha contribuito fortemente a questa transizione introducendo il supporto nativo alla multimedialità nei browser, così che oggi le principali piattaforme di streaming sul web hanno potuto seguire questa tendenza abbracciando nuovi formati e protocolli. [15]

1.1.2 Protocolli di streaming

Falsh Player supporta due forme di comunicazione con il server per la fornitura di stream multimediali [16].

- Su *HTTP*, in una modalità di accesso ai contenuti detta *progressive download*. Si tratta di accedere a un contenuto web che è l'intero file multimediale e scaricarlo progressivamente, eventualmente iniziando la riproduzione prima che il download sia completato. Per lavorare in questa modalità è necessario che un web site consenta esplicitamente a Flash di contattarlo, possibilità che tipicamente non è abilitata al fine di evitare possibili attacchi che possono essere portati attraverso Flash Player.
- Su *RTMP*, o *Real Time Messaging Protocol*, un protocollo esplicitamente pensato per applicazioni client-server real-time e che è stato per molto tempo il protocollo più utilizzato per la distribuzione di contenuti in streaming. Nelle sue diverse varianti, RTMP offre numerose possibilità, tra cui lo streaming su connessioni sicure su SSL, la cifratura dello stream con RTMPE e l'incapsulamento di RTMP in pacchetti HTTP per superare i firewall. L'utilizzo di questo protocollo potrebbe infatti incontrare un ostacolo nelle regole d'accesso imposte dai firewall.

L'abbandono di Flash Player si è svolto in parallelo con la nascita di nuovi protocolli dedicati allo streaming che hanno cercato di superare i problemi dei due approcci sopra descritti. Questi protocolli sono quelli HTTP-based ai quali è dedicato il secondo capitolo di questo lavoro.

1.2 Modalità di streaming

Come detto, anche Flash Player consente una modalità di streaming basata su HTTP. Questa è tuttavia molto diversa dal tipo di streaming che viene realizzato tramite i protocolli HTTP-based. Per capire in cosa le due modalità differiscono è necessario introdurre una classificazione delle tecniche di streaming.

- *Progressive download*. È una modalità di streaming in cui il contenuto, interamente rappresentato da un file, viene scaricato progressivamente partendo dall'inizio del file. Il client bufferizza il contenuto man mano che il download prosegue, e può iniziane la riproduzione prima ancora che sia completato. Il progressive download streaming ha rappresentato un importante passo avanti rispetto alle precedenti tecniche che prevedevano la necessità di scaricare l'intero contenuto prima di poter iniziare la riproduzione. Tuttavia risulta troppo rigido per quei client che abbiano l'esigenza di fare il seek all'interno del flusso multimediale.

- *Chunk based delivery.* È una modalità di streaming che prevede che il flusso venga suddiviso in chunk che vengono distribuiti separatamente. Lo streaming su RTMP prevede l'utilizzo di questa strategia. I chunk sono scaricati separatamente dal client, così che non sia più necessario iniziare il download dall'inizio dello stream.

Storicamente il progressive download veniva fatto su HTTP, mentre il chunk based delivery su protocolli dedicati come RTMP. Lavorare su HTTP offre una serie di vantaggi legati all'ampio supporto di cui gode questo protocollo, ma il download progressivo risulta molto limitato per le esigenze degli applicativi moderni. Il successo dei protocolli HTTP-based deriva proprio dalla loro capacità di combinare i vantaggi di queste strategie, offrendo un modo per distribuire flussi multimediali in chunk indipendenti su HTTP.

1.3 Soluzioni di streaming

Grazie alla diffusione e alla semplicità di protocolli di streaming HTTP-based, chi desideri mettere in piedi un'architettura dedicata allo streaming ha oggi due possibilità.

- Affidarsi a *media-server* dedicati, pensati appositamente per la distribuzione di flussi audiovisivi. Esistono molte soluzioni di questo tipo, commerciali e non. Questi prodotti supportano tipicamente un'ampia varietà di protocolli e sono arricchiti di molte funzionalità (dedicate a diversi contesti) che si aggiungono alla semplice capacità di distribuire contenuti multimediali.

- Lavorare con soluzioni che incorporino funzionalità di media-server in comuni web-server. Un approccio di questo tipo è reso possibile dalla natura web-based dei protocolli di streaming HTTP-based. È una soluzione che ha enormi vantaggi in termini economici e che tipicamente è molto semplice da configurare. Tuttavia è evidente che i sistemi che si possono ottenere in questo modo sono molto più limitati rispetto ai classici media-server, in quanto mancano di molte funzionalità a supporto dello streaming.

1.3.1 Soluzioni proprietarie

Tra i media-server proprietari più utilizzati citiamo:

- *Wowza Streaing Engine*. Uno dei media server più utilizzati, sviluppato da Wowza Media System, offre supporto allo streaming audio e video. Gestisce i formati 3GP, QuickTime, FLV, MP4, MPEG-TS e ABR e lavora con tutti i più comuni protocolli di streaming multimediale HTTP-based e non.

- *Unified Streaing Platform*. Una recente soluzione di streaming cross-platform. Lavora di fianco a comuni web server come Apache, Nginx, IIS. Rinuncia al supporto di protocolli più datati come RTMP, concentrandosi su quelli HTTP-based come HLS e DASH.

- *SHOUTcast*. Un media-server dedicato ai contenuti audio, viene tipicamente utilizzato per pubblicare e distribuire podcast. Definisce un proprio protocollo di streaming su HTTP.

1.3.2 Soluzioni open-source

La quantità di media-server open-source disponibili è molto elevata, e il suo incremento è stato supportato anche dalla semplicità di lavorare con i protocolli HTTP-based. Uno dei più longevi è *Red 5*, in grado di distribuire contenuti audio e video su diversi protocolli e per il quale sono stati sviluppati numerosi plug-in di successo, come quello a supporto dello streaming HLS.

Capitolo 2

Architettura di streaming

2.1 Struttura di una catena di streaming

I protocolli di streaming HTTP-based consentono di inviare flussi audio o video da un comune web server utilizzando il protocollo HTTP. Per farlo prevedono che il flusso venga diviso in una serie di *frammenti* di durata limitata accessibili da un client come risorse HTTP qualsiasi. Ciascun frammento è indipendente dagli altri. Una *playlist* descrive il contenuto multimediale indicando quali sono e dove si trovano i frammenti che lo compongono, la durata di ciascuno di essi, in che sequenza debbano essere concatenati per ricostruire il flusso originale e fornendo altre informazioni specifiche del protocollo utilizzato. La playlist è anch'essa una comune risorsa web scaricabile da un web server.

L'architettura necessaria a mettere in piedi un sistema in grado di fornire un servizio di streaming tramite un protocollo HTTP-based prevede la presenza di tre componenti fondamentali.

- *Media encoder*. È l'elemento che si occupa di ricevere il flusso audio o video e di codificarlo in un formato opportuno per la distribuzione, così come previsto dal protocollo utilizzato.

  L'encoder potrebbe dover anche codificare il flusso a diversi bitrate per supportare l'*Adaptive Bitrate Streaming*.

- *Stream segmenter*. È il componente che effettua la suddivisione del flusso audio o video in frammenti. Lavora su uno stream già codificato dal precedente componente, dunque non fa altro che dividere tale stream nei segmenti che verranno richiesti separatamente dai client. Di fatto si tratta dell'elemento che crea i file da servire nelle risposte HTTP.

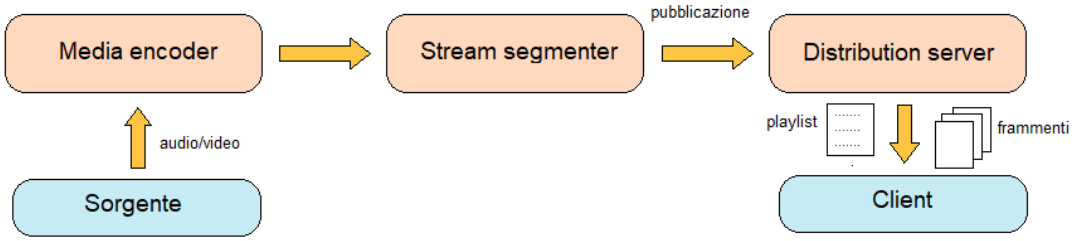

Figura 2.1. Architettura di streaming con protocolli HTTP-based

Si noti che tale componente fa da punto di contatto tra il flusso continuo e la logica di segmentazione del protocollo di streaming ed è dunque il responsabile della creazione e gestione della playlist che descrive il contenuto multimediale nel suo complesso.

Nel caso sia richiesto, il segmenter può avere il compito aggiuntivo di cifrare ciascun frammento creato e generare opportunamente una chiave[8].

- *Distribution server*. È il web server accessibile al client. Da esso un client può scaricare i frammenti del flusso audio o video separatamente, nonché le playlist.

L'architettura descritta è altamente modulare. Vale la pena di fare alcune considerazioni a tal proposito. Anzitutto si noti che la descrizione appena fornita non pone vincoli su come i diversi componenti debbano essere messi in relazione tra loro; la loro posizione, così come il modo in cui essi comunicano, possono variare in funzione delle esigenze implementative. È cioè possibile tanto posizionare tutti e tre i componenti su una stessa macchina fisica quanto dividerli geograficamente. Si noti, in particolare, che il distribution server può essere un web server usato per distribuire anche alti contenuti non legati allo streaming.

Il media encoder non sa nulla riguardo il protocollo di streaming utilizzato. Esso si limita a codificare un flusso continuo nel formato richiesto. Allo stesso modo, il distribution server è completamente ignaro del fatto che i frammenti che vengono pubblicati su di esso e che lui renderà disponibili siano parte di un contenuto multimediale. Di conseguenza questi componenti possono

essere implementati usando prodotti con i quali si ha già una certa confidenza e magari nati per rispondere ad esigenze totalmente diverse da quelle della fornitura di un servizio di streaming.

Ai tre componenti citati si aggiungono ovviamente la *sorgente* del flusso audio o video e il *client*, ovvero il consumatore finale dei frammenti e della playlist. Quando si parla di client in un'architettura di streaming HTTP si intende tipicamente un player dedicato alla riproduzione del contenuto multimediale. Qualsiasi sia lo specifico protocollo utilizzato, un player di questo tipo deve tipicamente lavorare eseguendo le seguenti operazioni.

1. Scaricare la playlist dal distribution server. Da essa è possibile ottenere informazioni sulla posizione dei frammenti che compongono il flusso.

2. Scaricare in sequenza i frammenti che compongono lo stream.

3. Una volta che dispone di un numero sufficiente di frammenti, iniziare la riproduzione del contenuto partendo dal primo frammento.

   A seconda del tipo di contenuto multimediale potrebbe essere necessario continuare a scaricare periodicamente dei frammenti. In caso di streaming live è richiesto che il client esegua il download della playlist aggiornata ogni volta che scade uno specifico intervallo di tempo, così da poter accedere a eventuali nuovi frammenti pubblicati.

2.2 Protocolli di streaming HTTP-based

I protocolli di streaming HTTP-based sfruttano l'infrastruttura web esistente per distribuire contenuti multimediali sul protocollo HTTP. I vantaggi di lavorare su HTTP sono sostanzialmente due:

1. La disponibilità di un'infrastruttura robusta e poco costosa per il delivering dei contenuti a livello globale, con possibilità di sfruttare elementi quali le CDN per ottimizzare la distribuzione delle risorse.

2. Il supporto diffuso di HTTP, che consente al protocollo di superare senza problemi firewall e proxy.

L'enorme successo di questo tipo di protocolli è senza dubbio legato a questi fattori. Inoltre esso offre un modo davvero elementare per effettuare *Adaptive Bitrate Streaming*, una tecnica di streaming che consente a un player di richiedere un stream a un bitrate adeguato alle proprie capacità in termini

di banda e di CPU. Con i protocolli HTTP-based il passaggio da uno stream a un bitrate a un altro si traduce semplicemente nel richiedere la playlist relativa al nuovo bitrate e iniziare il download dei frammenti in essa elencati. Le tecnologie di Adaptive Bitrate sono oggi quasi esclusivamente basate su protocolli HTTP-based.

I protocolli di streaming HTTP-based più rilevanti (che rappresentano di fatto le sole alternative disponibili) sono quattro.

- HTTP Live Streaming (*HLS*), sviluppando da Apple per i dispositivi iOS.
- Dynamic Adaptive Streaming over HTTP (*DASH*), un industrial standard sviluppato per offrire una soluzione aperta in alternativa alle dominanti soluzoni proprietarie.
- HTTP Dynamic Streaming (*HDS*), sviluppato da Adobe per applicazioni che utilizzano Flash Player.
- Microsoft Smooth Streaming (*MSS*), sviluppato da Microsoft a supporto di applicazioni che utilizzano Silverlight.

È chiaro che, come spesso succede quando una nuova tecnologia viene alla luce, ciascun produttore abbia cercato di imporre una propria soluzione a supporto dei propri prodotti. Pur se diversi in alcuni aspetti, tutti questi protocolli si somigliano molto negli elementi più importanti, che sono poi quelli che caratterizzano la famiglia dei protocolli di streaming HTTP-based. DASH nasce da un tentativo di fare ordine e individuare una tecnologia comune e definitiva, ma fin dalla sua nascita ha dovuto affrontare l'ostacolo imposto dalla possibilità delle aziende creatrici dei protocolli concorrenti di agire con forza sul mercato.

Le caratteristiche dei protocolli elencati sono messe a confronto in Tabella 2.1.

2.2.1 Playlist

La playlist è un file contenente l'elenco dei frammenti che compongono lo stream. La sintassi utilizzata varia a seconda del protocollo considerato: HLS adotta una propria sintassi, mentre HDS definisce un proprio XML Schema. In ogni caso, tramite la playlist è possibile accedere ai frammenti, per raggiungere i quali è fornito uno URL.

	DASH	**HLS**	**HDS**	**MSS**
Tipologia	Open	Proprietario	Proprietario	Proprietario
Codifiche video	MP4 fragments + MPEG-2TS	MPEG-2 TS	MP4 fragments	MP4fragments
Codifiche audio	AAC	AAC, MP3	AAC, WMA	AAC, MP3

Tabella 2.1. Caratteristiche dei protocolli di streaming HTTP-based [9]

Un client che ottenga la playlist relativa a uno stream può iniziare a scaricare i frammenti. Tipicamente esistono in questi protocolli due tipologie di playlist, a due diversi livelli.

1. Una playlist di primo livello, tipicamente detta *master playlist*, elenca non i frammenti, ma piuttosto tutte le possibili playlist di secondo livello tramite le quali si può ottenere l'elenco dei frammenti. Ciascuna delle playlist elencate, pur facendo riferimento allo stesso contenuto multimediale, consente l'accesso a una rappresentazione specifica di tale contenuto in uno stream dotato di caratteristiche specifiche. Il caso più classico è quello in cui la caratteristica che distingue i diversi stream sia il bitrate. Tuttavia è possibile pensare a casi diversi, come ad esempio quello in cui si vogliano differenziare degli stream video in funzione della lingua utilizzata per i sottotitoli. Non essendoci alcun legame, se non logico, tra i diversi stream, ancora una volta si ha ampia libertà di movimento nel definirne le caratteristiche peculiari.

2. Una playlist di secondo livello, detta spesso *media playlist*, che è quella che elenca i frammenti dello stream. Ciascun frammento è presentato al client con uno URL per recuperare la risorsa corrispondente e delle meta-informazioni che ne definiscono le caratteristiche. La più importante di queste informazioni è senza dubbio la durata del contenuto multimediale impacchettato in quel frammento.

La master playlist rappresenta il punto di accesso al contenuto multimediale, dunque il suo URL spesso è identificato con la posizione del contenuto sul web. Ottenuta la master playlist, il client può scegliere tra i diversi stream elencati quello più consono alle caratteristiche di rete e computazionali della macchina su cui esegue, nonché alle configurazioni impostate dall'utente. La

scelta non è definitiva, nel senso che in qualsiasi momento il client può scaricare nuovamente la master playlist (o accedere a quella che già possiede) per scegliere di accedere a uno stream diverso. Questa semplice operazione rappresenta tutto quanto è necessario in un protocollo HTTP-based per spostarsi da uno stream a un dato bitrate a un altro. Ovviamente, per implementare la tecnica dell'adaptive bitrate streaming, è previsto che i frammenti di ciascuno stream siano equivalenti in termini di durata, così che due frammenti con numero identificativo consecutivo siano concatenabili anche se appartenenti a stream diversi.

Ottenuta la media playlist il client può iniziare il download dei singoli frammenti e, quando ne ha a sufficienza, avviare la riproduzione.

2.2.2 Frammenti

I frammenti sono le unità di riproduzione elementari che compongono lo stream. Si tratta di file del tutto indipendenti tra loro e solo logicamente legati a uno stesso flusso multimediale. Dal punto di vista del web server essi non sono altro che risorse web statiche che vengono richieste tramite HTTP e il cui contenuto deve essere scritto nei body delle risposte da inviare ai client.

I frammenti sono file multimediali riproducibili autonomamente e codificate in uno dei formati ammessi dallo specifico protocollo. La durata del contenuto che rappresentano è un parametro fondamentale per la fruizione del contenuto in streaming. Essa definisce qual è l'unità temporale elementare cui si può accedere e che può essere dunque riprodotta. Il dimensionamento di questo parametro dovrebbe essere fatto con attenzione per rispondere a diverse esigenze proprie di contesti applicativi diversi: in alcuni casi potrebbe essere necessario offrire un livello di granularità più fine, in altri potrebbe non rappresentare una criticità.

Il dimensionamento della durata dei frammenti assume rilevanza anche nel dimensionamento del sistema dal punto di vista tecnico: frammenti di durata maggiore hanno dimensioni maggiori e richiederanno ai client un tempo maggiore per il download; d'altro canto un client ha anche più tempo per scaricare ciasun frammento prima di dover bloccare la riproduzione a causa della mancanza di nuove porzioni di contenuto.

2.2.3 Modalità: LIVE, VOD ed EVENT

I protocolli HTTP-based offrono tipicamente la possibilità di accedere a contenuti fruibili in modalità differenti. In particolare queste modalità di fruizione sono rese disponibili per rispondere a diversi casi d'uso propri dello striming su Internet:

1. La riproduzione di un *Video On Demand* (*VOD*), termine che identifica contenuti multimediali (in realtà tanto audio quanto video) che hanno un inizio e una fine e che un player può ottenere interamente (scaricandoli dalla rete) prima di iniziare la riproduzione.

   Se il player lo consente, l'utente può riprodurre qualsiasi porzione del contenuto multimediale, anche più volte.

2. La riproduzione di un evento *live*, dal vivo. È il tipico caso di trasmissione che l'utente può guardare o ascoltare solo restando allineato alla coda dello stream, ovvero in diretta.

   In questo caso non esiste una "fine" del contenuto, ma esso si protrae per un tempo indefinito. Allo stesso modo l'utente non può "tornare indietro" e muoversi liberamente all'interno del contenuto per riprodurne la porzione che desidera (o se può farlo è limitato a una ridotta porzione del contenuto stesso).

   Per la natura del caso d'uso, la fruizione di contenuti di tipo live pone sfide più complesse rispetto al caso dei VOD.

3. La riproduzione di un evento dal vivo che viene al contempo registrato sul server e per la cui riproduzione l'utente può decidere se restare allineato alla diretta o riprodurre porzioni precedenti. In questi casi si parla tipicamente di *event*, identificando con questo nome quella categoria di contenuti multimediali per i quali non si vincola l'utente a tenersi allineato alla coda dello stream.

   La distribuzione di contenuti multimediali in questa modalità è oggi sempre più comune e trova applicazione in contesti televisivi, sportivi e soprattutto su social network quali Facebook o YouTube, i cui creatori di contenuti desiderano rendere disponibile le proprie dirette a due tipologie di spettatori, quelli che le seguono in modo sincrono e quelli che le guarderanno o ascolteranno solo successivamente.

I protocolli di streaming HTTP-based garantiscono la possibilità di lavorare in ciascuna di queste modalità agendo esclusivamente sulle playlist.

Mantenendo inalterato il meccanismo fondamentale di richiesta dei frammenti, tali protocolli prevedono formati diversi per le playlist a seconda che si voglia dare accesso a un contenuto LIVE, VOD o EVENT. La descrizione di ciascuna di queste tipologie di contenuti resta un elenco di frammenti. La differenza fondamentale sta nel modo in cui questo elenco viene aggiornato (o non aggiornato).

Nel caso di contenuti VOD, caratterizzati dall'avere un inizio e una fine, la playlist non viene mai modificata. Si tratta infatti di stream immutabili, che sono memorizzati in qualche posizione sul web e sono accessibili "per intero". Non si ha contenuto da aggiungere, dunque la playlist corrisponde a un file statico su un Web Server.

Per i contenuti LIVE la playlist deve essere invece costantemente modificata. Man mano che l'evento che si vuole trasmettere in diretta si svolge, nuovi frammenti vanno ad aggiungersi alla playlist. Ma poiché ogni client deve essere forzato a restare allineato alla coda del contenuto, bisogna anche eliminare un numero corrispondente di frammenti dalla testa della playlist. Si definisce cioè un meccanismo a finestra mobile in cui la finestra è la dimensione (in numero di frammenti o in secondi) della porzione di contenuto multimediale che in un certo istante è visibile dal client. Chi scarica la playlist può accedere a tutti i frammenti in essa elencati, ma essi sono solo una piccola porzione dell'intero contenuto. Questo limita le possibilità del client di effettuare download di contenuti dal server e consente al server di cancellare i frammenti più datati (magari dopo un'opportuna finestra temporale di sicurezza che garantisca ai client più lenti di completare download pendenti di risorse) man mano che la diretta procede.

La modalità EVENT prevede di restituire al client delle playlist che elenchino tutti i frammenti generati fino al momento corrente. Registrare un evento corrisponde semplicemente a non cancellare i frammenti che vengono pubblicati sul server man mano che l'evento procede. Se si vuole che un client possa accedere in qualsiasi momento a questi frammenti, il meccanismo di aggiornamento della playlist deve corrispondere a una semplice concatenazione dei nuovi frammenti pubblicati sul server alla coda della playlist, senza cancellare quelli che si trovano in testa. Naturalmente questo comporta che maggiore è la durata dell'evento maggiore sarà la lunghezza della playlist. Ogni volta che scarica una playlist di questo tipo un client può scegliere se iniziare la riproduzione dall'inizio o dalla fine (in modalità live, di fatto). Soprattutto, se il player è sufficientemente intelligente può offrire all'utente la funzionalità di seek all'interno del contenuto.

2.2.4 Funzionalità di DVR

Un *Digital Video Recorder* (*DVR*) è un dispositivo che consente la registrazione di contenuti audiovisivi su un supporto di memoria non volatile e la loro fruizione in un momento successivo. Un *Network Digital Video Recorder* (*NDVR*) è un DVR network-based, ovvero tale che il supporto di memorizzazione in cui viene registrato il contenuto audiovisivo è posizionato nella rete di un provider. Non solo la registrazione, ma anche l'accesso al contenuto registrato avviene attraverso la rete.

Un client che desideri accedere a un contenuto registrato può tipicamente farlo partendo dall'inizio dello stesso o da un istante qualsiasi al suo interno. In questo senso, i protocolli di streaming HTTP-based supportano l'implementazione di meccanismi NDVR in modo molto elementare: da un lato, registrare equivale a garantire che i frammenti pubblicati sul web server non vengano cancellati e siano ancora accessibili al termine dell'evento dal quale vengono generati; dall'altro iniziare la riproduzione partendo da un istante qualsiasi all'interno del contenuto significa, data la playlist, calcolare qual è il frammento, tra quelli elencati, al cui interno è presente quel particolare istante e iniziare il download dello stream da esso.

Quando si parla di DVR, si prevede tipicamente che sia garantito all'utente di spostarsi all'interno del video o dell'audio che sta riproducendo in qualsiasi momento. Questa funzionalità di *seek* può essere implementata in modo molto semplice e con una logica abbastanza elementare da player che abbiano accesso a una playlist di tipo EVENT di un protocollo di streaming HTTP-based andando a scaricare solo i frammenti che vanno dall'istante richiesto dall'utente in poi. Ma su stream di durata molto elevata è anche possibile implementare questa logica sul backend, facendo sì che sia lo stesso web server che espone i contenuti multimediali ad esporre delle risorse che consentono, se accedute fornendo opportuni parametri, di ottenere delle playlist appositamente pensate per riprodurre il contenuto a partire da uno specifico istante ed eventualmente fino a uno un istante di terminazione.

2.2.5 HTTP Live Streaming (HLS)

HTTP Live Streaming è un dei più diffusi protocolli di streaming HTTP-based. Sviluppato da Apple per i dispositivi iOS, oggi è ampiamente supportato dalla maggior parte dei browser mobile [10] e sebbene molti browser desktop non lo supportino nativamente esistono numerose soluzioni per lavorare con questo protocollo anche in questi contesti. Le prime soluzioni di

questo tipo erano implementate in ActionScript e richiedevano il supporto di FlashPlayer per la riproduzione. Con l'avvento di HTML 5 e la diffusione delle Media Source Extension, la necessità di utilizzare Flash è stata superata e sono nate librerie JavaScript che consentono di riprodurre stream HLS tramite i tag media di HTML5 [11]. La grande diffusione di HLS deriva principalmente dal fatto che è l'unico protocollo di streaming supportato nativamente dai dispositivi iOS.

HLS definisce "il data format dei file e le azioni che devono essere intraprese dal server (sender) e dal client (receiver) dello stream"[1] pe riprodurre un contenuto multimediale. Le azioni compiute dalle due parti sono fondamentalmente quelle che accomunano tutti i protocolli HTTP-based, ovvero richiedere una master playlist e selezionare uno stream da riprodurre, richiedere la media playlist e iniziare a scaricare frammenti coerentemente con il tipo di contenuto (LIVE, VOD o EVENT) cui si sta accedendo. Il formato dei dati riguarda il formato dei frammenti, che prendono il nome di *media segment*. I formati supportati per i video sono MPEG-2 e Fragmented MPEG-4 (fMP4), mentre per l'audio sono accettate le codifiche Advanced Audio Encoding (AAC) con framing Audio Data Transport Stream (ADTS), MP3, AC-3 ed Enhanced AC-3 [1].

Caratteristiche del protocollo

HLS definisce due tipi di index file, o playlist. La *Media playlist* contiene un elenco di URL che puntano ai frammenti (detti *Media segment file*). Ciascuna Media playlist corrisponde a un flusso alternativo, detto *Variant*, dello stesso contenuto, che gode di specifiche caratteristiche che lo differenziano dagli altri Variant. La *Master playlist* contiene l'elenco delle Media playlist tra le quali il client può scegliere. Le playlist HLS hanno estensione `.m3u8`, mentre i Media Segment hanno estensione `.ts`. Ciascun Media segment deve contenere nel proprio nome un *Media sequence number*, ovvero un valore numerico che lo identifichi e che consenta l'ordinamento dei segmenti (segmenti successivi hanno Media sequence number consecutivi).

Le playlist contengono, oltre agli URL dei Media Segment o delle Media playlist, un insieme di tag. I tag servono a descrivere il contenuto della playlist o uno specifico segment. Ogni tag si trova su una riga dedicata, così come gli URL. Tutti i tag sono preceduti da un `#`, così come i commenti; in questo modo è possibile estendere il protocollo con nuovi tag avendo la garanzia che qualunque entità parli una versione non aggiornata del protocollo semplicemente li ignori. I tag principali sono[1]:

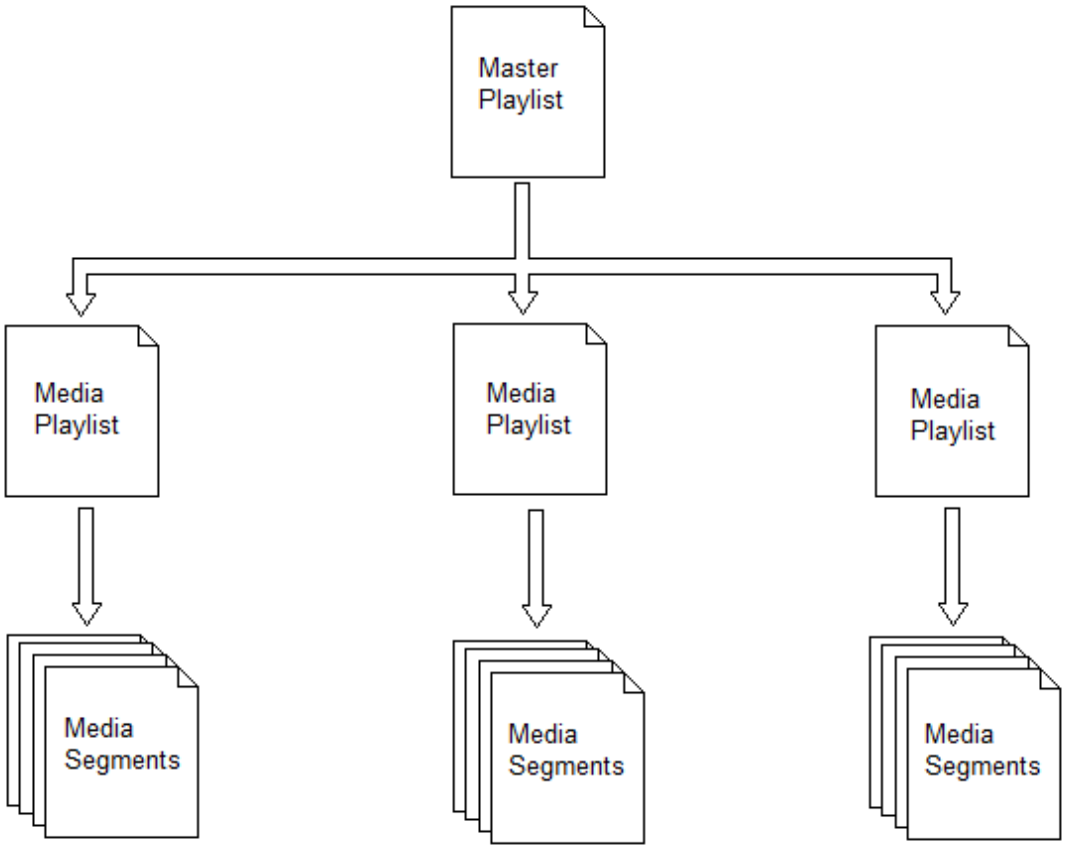

Figura 2.2. Architettura di streaming con protocolli HTTP-based

- `EXTM3U`. È la prima riga di ogni playlist e indica che il file è in formato *extended M3U*.
- `EXT-X-VERSION`. Indica la versione del protocollo.
- `EXTINF`. Nelle Media playlist, specifica la lunghezza di un Media Segment. Il valore consigliato è tra i 5 e i 10 secondi. La riga che segue un tag di questo tipo rappresenta lo URL di un Media segment.
- `EXT-X-TARGETDURATION`. Nelle Media playlist, indica la durata massima di ciascun Media sequence. Tipicamente è la durata che caratterizza ciascun segmento.
- `EXT-X-MEDIA-SEQUENCE`. Nelle Media playlist, indica il Media sequence number del primo segmento della lista.

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=2227464,RESOLUTION=960x540
v1/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=8178040,RESOLUTION=1920x1080
v2/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=6453202,RESOLUTION=1920x1080
v3/index.m3u8
```

Figura 2.3. Esempio di Master Playlist HLS

- `EXT-X-DISCONTINUITY`. Nelle Media playlist, indica un punto di discontinuità nello stream: il prossimo frammento elencato non segue in modo esatto, nel flusso originale, il precedente. L'utente può sperimentare della discontinuità nell'audio o nel video.
- `EXT-X-ENDLIST`. Nelle Media playlist, indica la fine di una del contenuto multimediale. È presente solo nelle playlist VOD. L'assenza di questo tag al termine della playlist indica a un client che dovrebbe chiedere nuovamente la playlist tra un certo intervallo di tempo, in quanto essa verrà probabilmente aggiornata.
- `EXT-X-PLAYLIST-TYPE`. Indica il tipo di Media playlist: LIVE, EVENT o VOD.
- `EXT-X-STREAM-INF`. Nelle Master playlist, fornisce informazioni riguardo una Media playlist, il cui URL si trova nella riga che segue questo tag. È seguito da una serie di attributi che descrivono il Variant, quali il bitrate e i codec usati.

Oltre questi tag fondamentali, HLS ne definisce molti altri di maggior dettaglio.

In Figura 2.3 si riporta un esempio di master playlist,
mentre in Figura 2.4 un esempio di Media Playlist.

Si noti che tutti gli URL sono considerati relativi al percorso della playlist corrente a meno di iniziare con il carattere `/`.

```
#EXTM3U
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://example.com/movie1/fileSequenceA.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceB.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceC.ts
#EXTINF:9.0,
http://example.com/movie1/fileSequenceD.ts
#EXT-X-ENDLIST
```

Figura 2.4. Esempio di Media Playlist HLS

Caratteristiche avanzate

Le caratteristiche descritte sono quelle necessarie a comprendere il funzionamento di base di HLS. Tuttavia il protocollo offre numerose funzionalità destinate a modalità di utilizzo più complesse di quelle descritte in questo capitolo. Gli attributi che possono essere allegati al tag `EXT-X-STREAM-INF` sono molti e consentono di descrivere nei dettagli i Variant. Ulteriori tag possono arricchire le playlist di informazioni utili in contesti applicativi più specifici.

Non essendo l'approfondimento delle funzionalità avanzate di HLS uno degli obiettivi di questo lavoro di tesi, si rimanda alla consultazione delle risorse elencate in bibliografia ([1], [7] e [12]) per ulteriori dettagli al riguardo.

Parte II

Streaming HTTP con Nginx

Capitolo 3

Soluzione di streaming con Nginx

3.1 Web server Nginx

Nginx è un server open-source ampiamente diffuso sul web, tra i più diffusi assieme ad Apache e a Microsoft Server. La scelta di utilizzarlo per implementare la soluzione oggetto di questa tesi è nata da una serie di considerazioni. Anzitutto la nota efficienza di questo web-server nel servire contenuti statici. Diversi test indipendenti riportano che Nginx riesce a servire circa il 40% di traffico in più rispetto ad Apache e che lo fa a una velocità 4.2 volte superiore [17]. Questo vuol dire che Nginx garantisce non solo grande velocità, ma anche scalabilità. Se i protocolli HTTP-based come HLS hanno ridotto le differenze tra il distribuire uno streaming media e il distribuire un contenuto statico da un web-server, tutto lascia supporre che Nginx possa mostrarsi altamente adatto allo scopo di fare streaming sul web.

Un'altra caratteristica che ha contribuito al successo di Nginx è l'estrema semplicità che contraddistingue il suo processo di configurazione: si tratta di agire su un file testuale diviso in moduli, in ciascuno dei quali possono essere specificate delle direttive nella forma `keyword valore` per abilitare o disabilitare determinati comportamenti. I moduli che compongono il file di configurazione sono ricorsivamente inclusi gli uni negli altri, e ciascuno di essi definisce un contesto. Le direttive presenti in un contesto si applicano ricorsivamente a tutti i contesti figli, in una gerarchia ad albero.

Il blocco più esterno definisce il *main context*, il contesto principale del server che contiene direttive di natura molto generica. Al suo interno possono

essere definiti diversi contesti, ma quelli che sono praticamente sempre usati sono [18]:

- *event*. Definisce il contesto in cui inserire le direttive relative al modulo `event` (si veda la Sezione 3.2), che si occupa di specificare in che modo Nginx debba gestire le connessioni. Viene inserito direttamente nel main context.

- *http*. Definisce in che modo Nginx dovrà gestire le richieste HTTP. Viene inserito direttamente nel main context. Altri contesti allo stesso livello di questo possono essere definiti per gestire richieste non HTTP.

- *server*. Viene inserito all'interno del contesto *http* (o in un qualsiasi contesto dello stesso livello). Il contesto *http* può contenere più blocchi *sever*, ciascuno corrispondente a un server virtuale che gestisce un certo sottoinsieme di richieste. Per differenziare le richieste e allocarle sui diversi server virtuali si possono usare le direttive `listen`, che consente di specificare una porta sulla quale si può contattare il server, e `server_name`, che viene utilizzata quando più server ascoltano sulla stessa porta andando a verificare se il suo valore corrisponde a quello dello header `Host` nella richiesta HTTP.

- *location*. È il contesto che rappresenta un endpoint su un server e che gestisce tutte le richieste che soddisfano certe caratteristiche (la cui query string soddisfi una certa regular expression). Possono essere innestate tanto nel blocco server quanto in altre location (in questo secondo caso vengono invocate seguendo una logica "general to specific").

In Figura 3.1 un esempio di file di configurazione Nginx.

La modularità non è esclusivamente una caratteristica che si ritrova in fase di configurazione: essa viene effettivamente mappata in una corrispondente modularità del server, che presenta moduli core che ne implementano le funzionalità fondamentali e moduli aggiuntivi che possono essere esclusi dalla compilazione o inclusi a seconda delle esigenze. Soprattutto è possibile implementare dei moduli specifici per rispondere alle necessità dello sviluppatore e compilarli con il codice sorgente al fine di ottenere un modulo a tutti gli effetti integrato nel server: le direttive che esso definisce sono utilizzabili esattamente come quelle dei moduli core all'interno del file di configurazione. In questo modo è possibile definire moduli *handler* (che rispondono cioè a richieste fatte su specifici URL), *filtri* e *load balancer* personalizzati.

```
worker_processes  auto;

events {
    ...
}

http {
    server {
        listen 80;

        ...
        location /hls {
            ...
        }
        location /dvr {
            ...
        }
}
```

Figura 3.1. Struttura di un file di configurazione Nginx

3.2 Moduli Nginx ed estendibilità

L'elevata modularità di Nginx è una delle ragioni del suo successo. I moduli di base sono quelli che costituiscono lo scheletro di Nginx. Si tratta del `core module`, dello `event module` e del `configuration module`.

Il *core module* è il componente che si occupa di gestire gli aspetti legati all'esecuzione di Nginx nel contesto del sistema su cui è installato. Offre infatti direttive per configurare il server nel modo più adeguato all'ambiente in cui verrà eseguito. Tra le principali direttive offerte dal modulo [2]:

- `worker_processes`. Consente di indicare il numero di *worker process* che utilizzerà Nginx. I worker process vengono avviati da un *master process*, e sono i processi che gestiscono le richieste. Si consiglia di impostare questa direttiva sul valore `auto`: il numero di worker process avviati sarà pari al numero di core disponibili.

- `error_log`. Abilita il logging a diversi livelli a seconda del valore indicato (`debug`, `info`, `warn`, `error`, ecc.). Consente anche di specificare il path del file di log.
- `thread_pool`. Definisce un *thread pool*, la sua dimensione e un nome tramite il quale identificarlo. In combinazione con la direttiva `aio` consente di servire file di grandi dimensioni in modo asincrono.
- `worker_rlimit_nofile`. Definisce il numero massimo di file che un worker process può usare contemporaneamente. È utile assegnare un valore molto alto a questa direttiva.

Lo *event module* consente di configurare le impostazioni relative alla rete. Tutte le direttive che definisce vanno inserite in un blocco `event` appositamente inserito nel main block esterno. Tra le direttive principali possiamo elencare `multi_accept`, che consente di abilitare la ricezione di più richieste in contemporanea da parte di ciascun worker process, e `worker_connections`, che indica il numero massimo di connessioni che un worker process può trattare simultaneamente. [2]

Il *configuration module* infine un modulo che abilita l'inclusione di file esterni nel file di configurazione.

I moduli *core*, *event* e *configuration* vengono sempre compilati con il server. Tuttavia esiste una serie di moduli complementari che possono essere inclusi nel processo di compilazione a discrezione dell'utente. Questi moduli forniscono una serie di servizi che estendono le funzionalità di Nginx.

Specificando le opzioni appropriate in fase di compilazione è possibile eseguire il build del sistema con l'integrazione di questi moduli. Esempi di opzioni di questo tipo sono `--with-thread`, che abilita l'utilizzo del thread pool, `--with-http_ssl_module`, che abilita l'utilizzo di SSL, `--with-http_stub_status_module`, per avere tenere traccia di metriche sullo stato del server e accedervi da un endpoint http.

Ciascuno di questi elementi può essere integrato solo se necessario.

3.2.1 Creazione di moduli per Nginx

Un modulo Nginx può appartenere a una tra tre categorie [3] [4]:

- *Handler*. Sono moduli dedicati a processare le richieste. Vengono registrati su una location (ogni location ha almeno uno handle; se in configurazione ne sono indicati di più ne sono di più uno solo di essi viene

registrato) e producono l'output per tutte le richieste destinate a quella location.

- *Filter*. Elaborano l'output prodotto dagli handler. Vengono invocati per processare la risposta destinata al client prima che essa lasci il server. Sono organizzati in una catena in cui l'output di un filtro diviene input del successivo. L'aspetto più interessante del loro funzionamento è che essi possono iniziare il processing prima ancora che il filtro che li precede abbia completato la generazione del proprio output. [3]

- *Load balancer*. Distribuiscono le richieste su eventuali backend server per i quali Nginx agisca da reverse proxy. Di default Nginx ha due moduli di load balancing, che eseguono gli algoritmi *round robin* e *IP hash*.

Durante tutto il proprio ciclo di attività il sever interagisce con i moduli che lo compongono in diversi momenti. Implementando opportune callback i moduli possono agganciarsi al server. Nginx invocherà queste callback nei momenti opportuni. Il modello di interazione è dunque di tipo asincrono.

Caratteristiche comuni ai diversi tipi di modulo

Un modulo Nginx è pensato per lavorare ad in un certo contesto. Ciascun modulo può definire un numero di `struct` di configurazione che è pari al numero di contesti innestati nei quali è inserito. Tipicamente è sufficiente implementare la struct relativa al contesto locale [4], ma in teoria si possono definire anche quelle riguardanti il contesto `server` e `main`. I suoi campi andranno riempiti con i valori passati alle direttive del modulo in fase di configurazione. Per convenzione il nome di tali strutture è nella forma `ngx_http_<module name>_(main|srv|loc)_conf_t`.

L'elenco delle direttive definite dal modulo è fornito in un vettore statico di `ngx_command_t`. Un `ngx_command_t` corrisponde a una direttiva. Ogni direttiva ha un nome che viene usato per invocarla nel file di configurazione. Il tipo è configurabile tramite opportuni flag e consente di definire caratteristiche quali lo scope della direttiva (`NGX_HTTP_MAIN_CONF`, `NGX_HTTP_LOC_CONF`, `NGX_HTTP_SRV_CONF`, `NGX_HTTP_UPS_CONF`), se essa riceve argomenti e in questo caso quanti (`NGX_CONF_NOARGS`, `NGX_CONF_TAKE1`, `NGX_CONF_TAKE2`, ecc.), se riceve valori di tipo `on` /`off` (`NGX_CONF_FLAG`), se c'è un numero minimo di parametri da fornirle (`NGX_CONF_1MORE`, ecc.) e altro ancora (si veda http://lxr.nginx.org/source/src/core/ngx_conf_file.h?v=nginx-1.12.0) [3].

```
struct ngx_command_t {
    ngx_str_t             name;
    ngx_uint_t            type;
    char                *(*set)(ngx_conf_t *cf,
                                ngx_command_t *cmd,
                                void *conf);
    ngx_uint_t            conf;
    ngx_uin\_t            offset;
    void                 *post;
};
```

Figura 3.2. Struttura `ngx_command_t` [3]

`set` è la funzione che verrà usata per tradurre il valore passato alla direttiva nel file di configurazione un data type adeguato e a memorizzare tale valore nella `struct` di configurazione. A tale scopo esiste una serie di funzioni messe a disposizione dall'ambiente Nginx per effettuare la traduzione nei tipi elementari gestiti dalla libreria `ngx\_core`: `ngx\_conf\_set\_flag\_slot`, `ngx\_conf\_set\_str\_slot`, `ngx\_conf\_set\_num\_slot`, `ngx\_conf\_set\_size\_slot` e altre disponibili in http://lxr.nginx.org/source/src/core/ngx_conf_file.h?v=nginx-1.12.0#0280.

Queste funzioni riescono a posizionare i dati automaticamente nel file di configurazione tramite i parametri `conf` e `offset`, che indicano rispettivamente se il parametro vada associato alle strutture di configurazione `NGX_HTTP_MAIN_CONF_OFFSET`, `NGX_HTTP_SRV_CONF_OFFSET` o `NGX_HTTP_LOC_CONF_OFFSET` e l'offset del campo dedicato in tale struttura.

Altro elemento di fondamentale importanza è la `struct ngx_http_module_t`, che consente di agganciare alcune callback definite dal modulo così che Nginx possa invocarle al momento opportuno.

Definire una struttura di questo tipo consente di specificare quali funzioni debbano compiere alcune azioni che ci si aspetta il modulo esegua in diversi momenti del ciclo di attività del server (fasi di pre-configurazione, post-configurazione, creazione del contesto main, sua inizializzazione, creazione del contesto server, merging del contesto server con quello main, creazione del contesto local e suo merging con quello server [3]).

Nginx offre delle funzioni che consentono i effettuare il merging e la creazione dei contesti in modo molto semplice. Per ulteriori dettagli si rimanda

```
typedef struct {
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);
    void       *(*create_main_conf)(ngx_conf_t *cf);
    char       *(*init_main_conf)(ngx_conf_t *cf,
                                  void *conf);
    void       *(*create_srv_conf)(ngx_conf_t *cf);
    char       *(*merge_srv_conf)(ngx_conf_t *cf,
                                  void *prev, void *conf);
    void       *(*create_loc_conf)(ngx_conf_t *cf);
    char       *(*merge_loc_conf)(ngx_conf_t *cf,
                                  void *prev, void *conf);
} ngx_http_module_t;
```

Figura 3.3. Struttura `ngx_http_module_t` [3]

alla guida di Evan Miller sulla creazione di moduli Nginx [3].

Fatta accezione per queste caratteristiche comuni, il resto dell'implementazione prosegue in modo diverso a seconda della tipologia di modulo che si vuole ottenere. In questa sede si parlerà esclusivamente de moduli *handler*, che sono i più diffusi e grazie ai quali è possibile rispondere alla maggior parte delle esigenze di chi desidera aggiungere funzionalità ad Nginx.

Struttura di un modulo Handler

Il cuore di un modulo handler è la funzione traite la quale gestisce le richieste a lui destinate. La signature di tale funzione è indicata di seguito.

```
static ngx_int_t
ngx_http_circle_gif_handler(ngx_http_request_t *r) {
        ...
}
```

Figura 3.4. Funzione di handling dell richieste in un modulo handler

Tutto il lavoro che svolge lo handler va eseguito in questa funzione. Per agganciare questa funzione alla gestione delle richieste dirette al modulo è necessario procedere in questo modo: tra le direttive dichiarate nel vettore di `ngx_command_t` bisogna dichiarare una direttiva dedicata all'abilitazione del modulo; a questa direttiva deve essere associata una callback che accedendo alla configurazione locale del modulo associ la funzione handler al campo `handler` di tale `struct ngx_conf_t`.

La funzione handler deve fare tre cose: generare una risposta, scrivere lo header e scrivere il body.Tutto questo partendo da una richiesta che è contenuta in una structure `ngx_http_request_t` che è nella forma seguente:

```
typedef struct {
    ...
    ngx_pool_t                      *pool;
    ngx_str_t                        uri;
    ngx_str_t                        args;
    ngx_http_headers_in_t            headers_in;
    ...
} ngx_http_request_t;
```

Figura 3.5. Struttura `ngx_http_response_t` [3]

Sono stati mostrati solo i campi principali di tale struttura.

La scrittura dello header prevede l'inizializzazione dei campi della struttura `ngx_http_header_out_t`.

I campi vengono tradotti nelle intestazioni della risposta HTTP. Per inviare lo header si può usare la funzione `ngx_http_send_header(ngx_http_header_out_t *h)`.

Infine il body della risposta va scritto in un buffer di tipo `ngx_buf_t` che va inserito in una catena di tipo `ngx_chain_t` che Nginx considera contenere tutti i buffer che concatenati costituiscono la risposta da inviare al client. Il buffer Nginx consente di specificare la posizione del buffer di memoria contenente la risposta e quando è stato assegnato alla catena si può invocare la funzione `ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *c)`, che invia il body di risposta.

```
typedef struct {
    ...
    ngx_uint_t                        status;
    size_t                            content_type_len;
    ngx_str_t                         content_type;
    ngx_table_elt_t                  *content_encoding;
    off_t                             content_length_n;
    time_t                            date_time;
    time_t                            last_modified_time;
    ..
} ngx_http_headers_out_t;
```

Figura 3.6. Struttura `ngx_http_header_out_t` [3]

3.2.2 File `config`

Perché possa essere utilizzato, qualsiasi modulo deve essere compilato con i sorgenti di Nginx. Per farlo bisogna configurare il server in modo tale che sia a conoscenza dell'esistenza di questo nuovo modulo.

Nginx offre un file di configurazione `configure` che deve essere eseguito prima di procedere al built tramite `make`. L'esecuzione di questo file è cruciale, in quanto definisce tutte le caratteristiche che verranno utilizzate da Nginx nel venire installato. Ad esempio si può definire se debba essere usato l'`ngx_http_ssl_module`, se si debba abilitare l'utilizzo del thread pool, ecc. Diversi attributi consentono di informare Nginx riguardo il modo in cui si desidera venga installato nel sistema.

Uno di questi attributi è `-add-module=/path/to/module`, che indica al server l'esistenza di un modulo custom che deve essere incluso nella compilazione. Il path passato a questo attributo deve essere quello della directory in cui si trova il modulo. In questa directory devono essere presenti due cose:

1. Il file `.c` che contiene il codice sorgente del modulo, ovvero tutte le strutture e le callback di cui si è parlato in questa Sezione.

2. Un file `config` che contenga l'elenco dei file che devono essere usati durante la compilazione perché il modulo possa funzionare. Un esempio di file di questo tipo è riportato di seguito.

```
ngx_addon_name=ngx_my_module
HTTP_MODULES="$HTTP_MODULES ngx_http_my_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS
                    $ngx_addon_dir/ngx_http_my_module.c"
```

Figura 3.7. Struttura `ngx_http_header_out_t` [3]

Configurato Nginx con questo parametro, si può eseguire il `make` per compilare.

3.3 Modulo RTMP per Nginx

La creazione di moduli di terze parti è ovviamente incentivata e ne esistono diversi che implementano funzionalità utili in vari contesti. Uno di questi è lo RTMP-module, che aggiunge a Nginx funzionalità di streaming proprie di un media-server per live streaming RTMP, HLS o DASH. Il modulo consente di configurare un server RTMP che espone un endpoint per accettare connessioni da un encored client. L'encoder può essere un qualsiasi applicativo in grado di generare uno stream in formato FLV su RTMP, come `ffmpeg`, e supporta H.624 e AAC. Lavorando come segmenter, divide il flusso in frammenti che salva in file in formato opportuno a seconda del protocollo utilizzato (`.ts` per HLS) e che sono a questo punto risorse sul web-server Nginx. Inoltre genera la playlist che descrive lo stream.

Nella catena di streaming propria dei protocolli HTTP-based (nella quale sono previsti un *Media encoder* per la codifica del flusso, uno *Stream segmenter* per la suddivisione in frammenti e la generazione delle playlist e un *Distribution server* per la pubblicazione di questi contenuti sul web) il modulo RTMP agisce da segmenter: definisce un endpoint RTMP su Nginx per ricevere connessioni da qualsiasi client (es. `ffmpeg`) in grado di generare un stream codificato H.624 o AAC su RTMP, frammenta questo flusso generando file che vengono pubblicati sul web-server e produce una playlist che descrive lo stream.

Oltre a consentire la pubblicazione di contenuti VOD (Video-On-Demand), per i quali tutti i frammenti dello stream sono disponibili sul server e la cui playlist non viene dunque modificata nel tempo, aderendo alle specifiche del

protocollo HLS il modulo supporta anche le modalità LIVE ed EVENT, entrambe dedicate alla pubblicazione di contenuti la cui registrazione avviene mentre i client riproducono lo stream e che richiedono dunque non solo che nuovi frammenti vengano pubblicati periodicamente sul server, ma anche che la playlist venga aggiornata di conseguenza. Secondo tali specifiche, una playlist LIVE contiene l'elenco di un numero limitato di frammenti, quelli che ricadono in una finestra temporale di dimensione limitata e allineata alla coda dello stream: aggiornare una playlist di questo tipo vuol dire aggiungere in coda il nuovo frammento pubblicato e rimuovere dalla testa quello meno recente. Una playlist EVENT contiene invece l'elenco di tutti i frammenti pubblicati dall'inizio dello stream fino all'istante corrente, così che l'aggiornamento della playlist si traduce in un'operazione di append in coda alla lista.

3.3.1 Configurazione del modulo RTMP per lo streming HLS

L'RTMP module è in realtà un complesso insieme di moduli che servono a diversi scopi. Esso interviene molto a fondo nella struttura modulare di Nginx, creando un nuovo tipo di contesto che può essere innestato all'interno del contesto *main* del file di configurazione. Si tratta del contesto *rtmp*, che definisce appunto un blocco in cui si possono creare dei server RTMP. Questi server comunicano utilizzando questo protocollo piuttosto che HTTP.

All'interno di un context *server* dichiarato nel contex *rtmp* si possono dichiarare delle *application*. Una *application* è un una `location` che implementa determinate funzionalità di streaming. L'application è un endpoint RTMP che può essere contattato da un media encoder. Il media encoder può utilizzare il base path associato all'`application` che vuole contattare e farlo seguire dal nome che desidera dare allo stream. L'application creerà una nuova playlist dedicata a quel particolare flusso in input, dando al manifest il nome fornito dal media encoder.

In una application è possibile abilitare l'utilizzo di un protocollo. Dato che tanto per DASH quanto per HLS sono definiti dei moduli dedicati, abilitare un protocollo di streaming piuttosto che un alto significa di fatto abilitare i corrispondenti moduli.

Per abilitare il modulo HLS si può utilizzare la direttiva `hls on`. All'interno del contesto della `application` si possono utilizzare tutte le direttive che il modulo offre e che riguardano HLS al fine di configurare il protocollo. Le principali direttive offerte sono analizzate di seguito.

- `hls_path`. Indica la directory in cui si troveranno la playlist e i fragment.
- `hls_fragment`. Consente di specificare la durata per i frammenti HLS.
- `hls_playlist_length`. Consente di specificare la lunghezza complessiva della playlist.
- `hls_type`. Indica di che tipo debba essere la playlist creata dal modulo, `LIVE` o `EVENT`.
- `hls_variant`. Consente di specificare un suffisso per i diversi variant stream che possono arrivare dal media encoder. Se l'`aplication` riceve diversi stream in input con lo stesso nome seguito da diversi suffissi, verifica se questi suffissi corrispondono a qualcuno di quelli dichiarati assieme a questa direttiva. In questo caso considera quegli stream come un variant di uno stesso stream avente come nome la porzione di nome comune a tutti i flussi.
- `hls_keys`, `hls_key_path`, `hls_key_url`, `hls_fragments_per_key`. Consentono di abilitare e gestire l'utilizzo di chiavi di cifratura per cifrare i frammenti.

Per completare la configurazione è necessario creare una `location` sul server HTTP che consenta l'accesso ai fragment. Per farlo è sufficiente creare una comune `location` e associare ad essa la gestione dei file nella directory in cui il modulo inserisce i frammenti e la playlist. Il classico meccanismo di handling di richieste HTTP per file statici messo in atto da Nginx farà il resto.

In figura 3.8 è mostrato un esempio di file di configurazione per abilitare il supporto ad HLS in Nginx.

3.3.2 Supporto ad altri protocolli

Il modulo consente di eseguire una configurazione analoga per abilitare il supporto a DASH. Data la natura estremamente simile dei due protocolli, le direttive disponibili nel modulo DASH sono sostanzialmente le stesse disponibili in quello HLS, con la variante del richiamo al nome del protocollo corretto.

```
worker_processes  auto;

http {
    server {
        listen 80;
        location /hls {
            types {
                application/vnd.apple.mpegurl m3u8;
            }
            root /tmp;
        }
    }
}

rtmp {
    server {
        listen 1935;
        application tv {
            live on;

            hls on;
            hls_path /tmp/hls;
            hls_fragment 10s;
        }
    }
}
```

Figura 3.8. File di configurazione di Nginx con supporto ad HLS tramite l'RTMP-module [5]

3.4 Funzionalità non offerte da Nginx rispetto a un media-server

Le funzionalità offerte dal modulo RTMP si limitano a quelle proprie di uno stream segmenter. Certamente il modulo RTMP abilita l'utilizzo di Nginx come media server e lo fa mantenendo la semplicità di configurazione che caratterizza il server in cui viene inserito. Tuttavia se si desidera ottenere

supporto a scenari applicativi più complessi è necessario aggiungere qualcosa a questo modulo.

Un media server dedicato consente ad esempio di accedere a una serie di servizi di NDVR che si manifestano, oltre che nella registrazione dello stream (cosa che anche l'RTMP module può fare, in qualche modo, semplicemente se configurato con opportune direttive per preservare i frammenti nelle posizioni in cui li ha creati), anche nell'offerta di API per l'accesso a playlist customizzate.

Con l'obiettivo di implementare proprio questo tipo di funzionalità si è scelto di sviluppare un modulo Nginx che possa essere integrato nel server HTTP per gestire alcune tipologie di richieste destinate allo streaming HLS in modo differenziato.

Capitolo 4

Funzionalità di streaming aggiuntive per Nginx

4.1 Modulo HLS-NDVR

L'HLS-NDVR module è un modulo handler per Nginx che implementa funzionalità di Network-DVR. Sebbene la configurazione di Nginx analizzata in questo lavoro di tesi lo veda installato sul server assieme all'RTMP module, il modulo è in grado di lavorare di fianco a qualsiasi componente (che si tratti di un'estensione integrata in Nginx o meno) che provveda a posizionare dei frammenti HLS e la relativa playlist nel file system del server.

L'HLS-NDVR module vuole consentire ai client che scelgano di lavorare con HLS di superare la staticità della playlist generata dallo RTMP module o da qualsiasi altro stream segmenter, il cui formato è tipicamente definito una tantum in fase di configurazione. Grazie allo HLS-NDVR module i client possono richiedere playlist diverse per uno stesso stream: playlist VOD che contengono solo una porzione di un contenuto registrato o ancora in fase di registrazione; playlist LIVE con dimensione della sliding window personalizzata; playlist EVENT che partano da un istante arbitrariamente scelto anziché dall'inizio dello stream.

Dal punto di vista logico il modulo non fa altro che indicare a un client che lo contatti quali frammenti debba recuperare per ottenere l'accesso allo streaming nella configurazione che preferisce. Di conseguenza non è necessario che esso intervenga in alcun modo sulla codifica dei fragment né sulla pubblicazione della playlist originale, ma è sufficiente che si limiti ad effettuare un'operazione di parsing su quest'ultima per filtrarne il contenuto e

renderlo visibile al client nella forma che desidera.

Il risultato è un componente la cu piena integrazione in Nginx dovrebbe garantire un alto livello di scalabilità e la cui elevata coesione interna dovrebbe mantenere aperta la strada all' introduzione di ulteriori funzionalità in futuro.

4.2 Design del modulo

La fase di design del modulo ha richiesto anzitutto la definizione delle API per accedere al servizio. Il modulo riceve richieste su URL che si presenteranno in uno specifico formato:

```
http://[hostname]/[module_location]/[stream_name]?\{attributes\}
```

In cui `module_location` è il base path della `location` gestita dal modulo HLS-NDVR, `stream_name` è il nome dello stream, corrispondente per convenzione al nome della playlist `.m3u8` che descrive il contenuto cui si fa riferimento, e `attributes` è l'elenco degli attributi che consentono di personalizzare la playlist che verrà ottenuta in risposta.

La struttura dello URL rispecchia una scelta implementativa, ovvero quella di realizzare un modulo handler per Nginx. Sebbene questa scelta non fosse obbligata la si è ritenuta la migliore tra le alternative possibili. Per i dettagli riguardo le ragioni di tale scelta si rimanda alla Sezione 4.3.

La porzione più rilevante di questo URL è rappresentata dagli attributi. Essi rappresentano il modo che ha il client di informare il modulo delle caratteristiche di cui deve godere la playlist. Si è quindi concentrata l'attenzione sulla definizione di un insieme di attributi che rispondesse alle esigenze dei casi d'uso del modulo.

4.2.1 Funzionalità richieste al modulo

Si possono riassumere le funzionalità che si vuole il modulo fornisca nelle seguenti:

- Generare una playlist VOD che comprenda la porzione di contenuto compresa tra i due istanti specificati dal client.
- Generare una playlist LIVE con una finestra di dimensione scelta dal client.

- Generare una playlist EVENT che parta dall'inizio dello stream o da un punto arbitrariamente scelto.

Per offrire queste tre funzionalità si è scelto di definire i seguenti attributi con i quali è ammissibile corredare la richiesta fatta al modulo.

- `start`. Il valore è un intero che rappresenta un numero di secondi dall'inizio dello stream registrato. Indica il secondo dal qual si vuole che parta la playlist.
- `duration`. Il valore è un intero che indica un numero di secondi. Specifica la durata desiderata per la playlist. Utilizzare questo attributo non equivale necessariamente a richiedere uno stream VOD: infatti la durata potrebbe essere maggiore della lunghezza dello stream; in questo caso il modulo arriva fino alla fine della playlist e non inserisce il tag di terminazione `#EXT-X-ENDLIST`.
- `window`. Il valore è un intero che indica in caso di playlist LIVE di quanti frammenti debba essere composta la sliding window.
- `live`. Non ha valore, e forza la playlist ad essere di tipologia LIVE.
- `event`. Non ha valore, e forza la playlist ad essere di tipologia EVENT.
- `vod`. Non ha valore, e forza la playlist ad essere di tipologia VOD.

Il modulo non definisce delle regole di priorità. I parametri agiscono indipendentemente. Una query che presenti sia il parametro `start` sia quello `duration` ma anche un parametro `window` comporta che il modulo cerchi di applicare una finestra di durata indicata allineandola alla coda della porzione di contenuto delimitata da `start` e `duration`.

I tre casi di utilizzo più tipici sono quelli che consentono di rispondere alle tre funzionalità elencate in precedenza. Un esempio di richiesta per ottenere un contenuto VOD è il seguente.

```
http://[hostname]/dvr/movie?start=300\&duration=40
```

Un esempio di playlist che potrebbe essere restituita dal modulo in risposta a questa query è presentato in Figura 4.1.

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:10.0
#EXT-X-MEDIA-SEQUENCE:30
#EXTINF:10.0,
movie30.ts
#EXTINF:10.0,
movie31.ts
#EXTINF:10.0,
movie32.ts
#EXTINF:10.0,
movie33.ts
#EXT-X-ENDLIST
```

Figura 4.1. Possibile risposta VOD a una query all'HLS-NDVR module

4.3 Scelte tecnologiche e implementazione

La scelta di implementare il servizi di NDVR su Nginx come modulo interno al server non è stata dettata da necessità. Altre soluzioni avrebbero potuto essere adottate per ottenere il medesimo risultato. Proprio perché indipendente dall'utilizzo dell'RTMP module o di qualsiasi altro tipo di stream segmenter, una tipologia diversa di soluzione, magari basata su un componente per il quale Nginx poteva agire da reverse proxy, sarebbe stata altrettanto valida.

I motivi per il quale si è optato per la creazione di un modulo Nginx piuttosto che per la realizzazione di un componente esterno sono due.

- Il desiderio di offrire una soluzione compatta e facilmente configurabile. Utilizzare una soluzione che combinasse Nginx e un'altra tecnologia avrebbe aumentato la complessità del sistema e la difficoltà di configurazione. Restare confinati all'ambiente Nginx ha d'altro canto consentito di esplorare interamente l'ambiente e le possibilità offerte da questo server.

- Il tentativo di sfruttare proprio le caratteristiche che contraddistinguono Nginx ai fini di ottimizzare il servizio di streaming. Combinare una

tecnologia differente con Nginx avrebbe potuto compromettere la capacità di questo web-server di tenere fede alle aspettative in termini di scalabilità e velocità.

Una volta scelto di implementare un modulo Nginx, la selezione della tipologia handler è venuta naturale noto il tipo di servizi da offrire.

Per l'implementazione del modulo si è fatto affidamento alla guida di Evan Miller alla scrittura di moduli Nginx [3], alla quale si è attinto anche per la trattazione teorica fatta nella Sezione 3.2.1. Si rimanda a tale Sezione per le caratteristiche generali del processo di scrittura di un modulo Nginx.

4.3.1 Caratteristiche della programmazione di moduli per Nginx

Per comprendere alcuni aspetti dello sviluppo dello HLS-NDVR module è necessario introdurre alcuni concetti propri della programmazione di moduli Nginx.

Tipi e funzioni wrapper in Nginx

Tipicamente un modulo Nginx lavora su tipi che sono definiti come wrapper per i tipi elementari del C. Le definizioni di questi tipi sono fatte all'interno del file `ngx_core`. La maggior parte delle funzioni messe a disposizione nell'ambiente Nginx lavora su parametri aventi uno di questi tipi.

I tipi principali definiti sono i seguenti.

- `ngx_uint_t`
- `ngx_int_t`
- `ngx_str_t`

Tramite il wrapping dei tipi Nginx offre un set di data type che modellano oggetti complessi come le stringhe o i file, nonché funzioni che sfruttano l'alto livello di astrazione derivante da questa scelta implementativa.

Si incentiva l'utilizzo delle funzioni wrapper offerte dall'ambiente. Queste offrono tre fondamentali vantaggi:

1. Lavorano direttamente sui tipi definiti da Nginx, ricevendoli come parametri e restituendoli.

2. Sono pensate per eseguire del logging, e spesso ricevono appunto un oggetto di tipo `ngx_log_t` come parametro.

3. Lavorano con delle macro che sono organizzate in un sistema di codici di errore e di successo proprio di Nginx, rendendo più esplicita la gestione di condizioni di errore rispetto al caso in cui essa venga fatta esclusivamente tramite interi.

Il tipo `ngx_str_t`

È particolarmente importante saper gestire il tipo `ngx_str_t` se bisogna lavorare con vettori di caratteri, cosa molto comune nell'implementare degli handler HTTP, che per definizione devono generare risposte per client sfruttando un protocollo testuale. Anche nel caso dello sviluppo dello HLS-NDVR module si è dovuto lavorare molto con le stringhe nella fase di parsing della playlist `.m3u8`.

La struttura `ngx_str_t` è definita come segue.

```
typedef struct {
        size_t len;
        u_char *data;
} ngx_str_t;
```

Figura 4.2. `struct ngx_str_t`

Contiene dunque un campo `u_char\*` che rappresenta il vettore di caratteri e una lunghezza espressa come `size_t`. Su questo tipo Nginx costruisce una serie di funzioni che in parte riproducono le funzionalità di quelle rese disponibili dallo header file `string.h` del C, in parte ne costruiscono di nuove. Alcune di queste sono utili all'inizializzazione di variabili di questo tipo [6]:

- `ngx_string(text)`. Crea una nuova variabile di tipo stringa inizializzando i campi coerentemente con il testo fornito come parametro.
- `ngx_null_string`. Crea una stringa vuota.
- `ngx_str_set(str,text)`. Da una stringa, la inizializza opportunamente con il testo passato come parametro.

- `ngx_str_null(str)`. Inizializza una stringa esistente a null.

Altre sempliicano il lovoro con i vettori di `unsignedchar` [6].

- `ngx_strcmp()`. Un wrapper per la `strcmp()`.
- `ngx_strncmp()`. Un wrapper per la `strncmp()`.
- `ngx_strstr()`. Un wrapper per la `strstr()`.
- `ngx_strlen()`. Un wrapper per la `strlen()`.
- `ngx_memcmp()`. Un wrapper per la `memcmp()`.
- `ngx_memcpy()`. Un wrapper per la `memcpy()`.
- `ngx_memmove()`. Un wrapper per la `memmove()`.
- `ngx_memzero()`. Imposta un blocco di memoria con tutti 0.
- `ngx_cpymem()`. Si comporta come la `ngx_memcpy()`, ma ritorna l'indirizzo di destinazione finale.
- `ngx_movemem()`. Si comporta come la `ngx_memmove()`, ma ritorna l'indirizzo di destinazione finale.
- `ngx_strlchr()`. Dati degli indici di delimitazione, erca un carattere in una stringa.
- `ngx_tolower()`. Trasforma l'intera stringa in lower-case.
- `ngx_toupper()`. Trasforma l'intera stringa in upper-case.
- `ngx_strcasecmp()`. Confronta le stringhe in modo analogo alla `strcasecompare` del C.
- `ngx_strncasecmp()`. Confronta le stringhe in modo analogo alla `strncasecompare` del C.

Gestione della memoria in Nginx

Nginx offre delle funzioni che wrappano l'allocazione e de-allocazione di memoria sullo heap di sistema. Queste funzioni sono le seguenti [6]:

- `ngx_alloc(size,log)`. Alloca memoria nello heap di sistema. Si tratta di un wrapper per l'allocazione di memoria con la `malloc()` del C che offre supporto al logging. Nel caso si verificasse qualche errore nella gestione
- `ngx_calloc(size,log)`. Wrappa la `calloc()` del C e aggiunge il logging in caso di errore.
- `ngx_memalign(alignment,size,log)`. Wrappa la funzione `posix_memalign()` su piattaforme che forniscano tale funzione.
- `ngx_free(ptr)`. Wrappa la `free(ptr)` del C, liberando la memoria puntata da `ptr`.

Tuttavia Nginx offre anche un altro sistema di allocazione, basato sui *pool*. Un pool è una zona di memoria in cui è possibile eseguire allocazioni, associato a un componente del ciclo di attività di Nginx e che ha una vita limitata a quella del componente cui è associato. Un pool è descritto dal tipo `ngx_pool_t`.

Diversi oggetti nel contesto Nginx sono dotati di un pool. Tipicamente tutti gli oggetti di configurazione ne hanno uno, così come l'oggetto richiesta di tipo `ngx_http_request_t`. Il grande vantaggio di allocare sui pool è che si tratta di memoria semi-gestita: di fatto non c'è bisogno di ricordarsi di liberare la memoria, in quanto l'intero pool verrà distrutto non appena l'oggetto cui appartiene terminerà il proprio ciclo di vita.

Un pool cerca di allocare fino a quando possibile blocchi di memoria consecutiva nella zona da lui gestita, andando sullo heap di sistema solo quando necessario. [6]

Si può lavorare con un `ngx_pool_t` tramite le seguenti funzioni [6]:

- `ngx_create_pool(size,log)` per creare un nuovo pool.
- `ngx_destroy_pool(pool)` per distruggere un pool.
- `ngx_palloc(pool,size)` per allocare sul pool un blocco di dimensione indicata. Un blocco allocato tramite questa funzione non avrà bisogno di essere liberato.

- `ngx_pcalloc(pool,size)` per allocare sul pool e riempire il blocco allocato di zero.
- `ngx_pfree(poolptr)` per liberare un blocco allocato sul pool.

È possibile eseguire molte operazioni complesse sui pool, come ad esempio registrare dei *cleanup handler* per personalizzare la liberazione della memoria gestita dal pool.

4.3.2 Implementazione

L'implementazione dell'HLS-NDVR module ha riguardato principalmente la scrittura della funzione handler che si occupa di gestire le richieste DVR. Come per ogni funzione di handling di una richiesta, i passaggi seguiti sono quattro:

1. Recuperare la local configuration.
2. Generare una risposta in funzione dello URL e dei parametri ricevuti.
3. Inviare lo header.
4. Inviare il body.

La funzione di handling è la `ngx_http_hls_ndvr_handler`. Al proprio interno essa invoca una funzione di generazione del manifest per elaborare la playlist, che ha la seguente signature:

```
ngx_int_t compute_playlist(ngx_pool_t *pool,
        u_char *outstr,
        u_char *stream,
        hls_ndvr_params_t *params,
        ngx_http_hls_ndvr_loc_conf_t *conf,
        u_char *paramsstr)
```

Figura 4.3. Funzione `compute_playlist` dell'HLS-NDVR module

I parametri ricevuti dalla funzione hanno i seguenti significati:

- `pool`. Un pool sul quale eseguire le allocazioni necessarie alla computazione della nuova playlist.

- `outstr`. Un vettore di `unsigned char` che verrà inizializzato con la playlist da inviare in risposta al client.
- `stream`. Una stringa che rappresenta il nome dello stream sul quale lavorare.
- `params`. Un'istanza di un tipo struttura dedicato a contenere i parametri che servono a computare la playlist.
- `conf`. La configurazione locale del modulo.
- `paramsstr`. La stringa che rappresenta l'insieme di parametri ricevuti con la richiesta.

Il tipo `hls_ndvr_params_t` è una `struct` che ha la seguente definizione:

```
typedef struct {
        playlist_type_t type;
        int start;
        int end;
        int window;

       int type_setted;
       int start_setted;
       int end_setted;
       int window_setted;
} hls_ndvr_params_t;
```

Figura 4.4. Struttura `hls_ndvr_params_t` dell'HLS-NDVR module

I primi quattro parametri corrispondono a quelli che possono essere passati nella query al modulo, mentre i successivi quattro servono a tener traccia di quali di questi parametri sono stati definiti dall'utente e quali assumono i valori di default (pensati per fornire, se l'utente non specifica preferenze, l'intera playlist). La struttura viene inizializzata tramite la funzione `init_request_params`, anch'essa definita nel modulo e che riceve la stringa dei parametri ottenuta dallo URL per eseguire la configurazione appropriata.

Gestione della playlist e sua scrittura nel buffer di risposta

La gestione della playlist viene effettuata dalla funzione `compute_playlist` tramite una opportuna libreria definita nei file `m3u8_playlist_util.h` ed `m3u8_playlist_util.c`. Questa playlist espone un tipo struttura `m3u8_playlist_t` che è dedicata a modellare una playlist HLS ed così definita.

```
typedef struct m3u8_playlist_s {
    m3u8_playlist_level_t   level;
    m3u8_playlist_header_t  *header;
    m3u8_playlist_body_t    *body;
    ngx_int_t               fragmentsnum;
    ngx_pool_t              *pool;
} m3u8_playlist_t;
```

Figura 4.5. Struttura `m3u8_playlist_t` dell'HLS-NDVR module

Il parametro level indica se quella contenuta è una master o una media playlist. La playlist agisce su un pool dedicato, mentre `fragmentsnum` indica il numero di frammenti in essa contenuto. La playlist contiene un puntatore allo header della playlist, definito in una struttura `m3u8_playlist_header_t`, e al body, che contiene l'elenco di frammenti ed è rappresentato da una struttura `m3u8_playlist_body_t`. Come si può vedere dalle seguenti Figure, la struttura dedicata allo header contiene campi relativi ai diversi possibili tag presenti nella prima parte di una qualsiasi playlist HLS, mentre la struttura per il body non è altro che un wrapper per una lista di stringhe.

```
typedef struct m3u8_playlist_header_s {
    short version;
    ngx_int_t media_sequence;
    float target_duration;
    playlist_type_t playlist_type;
} m3u8_playlist_header_t;
```

Figura 4.6. Struttura `m3u8_playlist_header_t` dell'HLS-NDVR module

```
typedef struct m3u8_playlist_body_s {
    unsigned char **lines;
    ngx_int_t length;
    ngx_int_t size;
} m3u8_playlist_body_t;
```

Figura 4.7. Struttura `m3u8_playlist_body_t` dell'HLS-NDVR module

Il body viene dunque visto come sequenza di righe in un file testuale (tali righe anche rappresentare quelle contenenti i tag `#EXTINF`).

La libreria offre la funzione `init_playlist` per inizializzare la struttura e un'altra serie di funzioni dedicate all'inserimento di elementi al suo interno.

- `add_extinf`. Aggiunge un frammento `.ts` e la relavita riga `#EXTINF` alla playlist.
- `slide_extinf`. Aggiunge un frammento `.ts` e la relativa riga `#EXTINF` alla playlist eliminando il più vecchio dei frammenti in essa presenti (utile per realizzar eil meccanismo di sliding window).
- `add_ext_x_stream`. Aggiunge un variant a una master playlist.
- `add_discontinuity`. Aggiunge una riga dedicata al tag `#EXT-X-DISCONTINUITY`.
- `add_extendlist`. Aggiunge il tag `#EXT-X-ENDLIST` di terminazione della lista.

Infine la funzione `m3u8_playlist` stampa in un buffer passato come parametro l'intera playlist in modo che sia conforme alle specifiche HLS. Invocando questa funzione la `compute_playlist` può ottenere la playlist che la funzione handler dovrà inserire nel body da inviare in risposta alla query del client.

Configurazione di Nginx per abilitare l'HLS-NDVR module

Le direttive offerte dal modulo sono tre: `hls-ndvr`, che va impostata a `on` per abilitare il modulo su una location; `fragments_location`, che riceve come parametro il path alla directory che contiene i frammenti e la playlist;

`ts_location`, che riceve come parametro il path relativo che deve precedere i frammenti `.ts` quando vengono elencati nella playlist.

Un esempio di configurazione è riportato di seguito.

```
...
http {
    server {
        listen 80;
        location /dvr {
            hls_ndvr on;
            fragments_location /path/to/frags
            ts_location /url/path/to/ts
        }
}
...
```

Figura 4.8. Struttura di un file di configurazione per abilitare l'uso dello HLS-NDVR module [19]

4.3.3 Introduzione del caching

Al momento il modulo elabora la playlist ogni volta che riceve una richiesta. Un comportamento di questo tipo risulta sicuramente adeguato al caso applicativo considerato, in cui un client richiede una playlist che però può subire in breve tempo variazioni e che quindi devono essere costantemente aggiornate.

È tuttavia importante tenere in considerazione due elementi che evidenziano quanto possa essere oneroso eseguire il calcolo della playlist ad ogni richiesta da parte dei client.

1. Un client richiede la playlist ogni N secondi, ma non è detto che essa abbia subito variazioni. Solo in un set limitato di circostanze la playlist è certamente stata modificata, e sono quelle in cui si lavora in modalità LIVE.

2. Quando si lavora in modalità LIVE o EVENT, è probabile che molti client effettuino in un certo momento richieste equivalenti che restituiscono la stessa playlist. La computazione della playlist in modo separato per ciascun client è inutile e onerosa.

Per rispondere alle esigenze che derivano in particolar modo dalla seconda considerazione si è pensato all'introduzione di un meccanismo di caching. Il caching sarebbe destinato a gestire le richieste per playlist che sono già state generate per qualche altra richiesta precedente.

Quando un client effettua una richiesta, il modulo dedicato al caching verifica anzitutto se sia possibile esaurirla con un contenuto cachato e solo in caso contrario scomoda lo handler per generare la playlist. In questo contesto è però necessario definire anche un meccanismo di disabilitazione selettiva dele entry nella cache: infatti all'aggiornarsi di una playlist è necessario che molte delle entry relative a quella playlist vengano eliminate dalla cache. "Molte", non "tutte", perché potrebbero esistere dei contenuti cachati che rappresentano playlist VOD che espongono solo una porzione del contenuto multimediale e la cui validità si preserva nel tempo.

Design del sistema di caching

Le possibilità per implementare un meccanismo di questo tipo sono diverse:

1. Implementare una cache interna al modulo e in-memory e far sì che la funzione handler verifichi la presenza di entry in questa cache prima di demandare la computazione alla funzione `compute_playlist`.

2. Implementare un modulo distinto che gestisca un sistema di caching dedicato alle playlist `.m3u8` e che comunichi in qualche modo con il modulo HLS-NDVR al fine di conoscere gli indici dei contenuti che tale modulo genera e gestirne la disabilitazione selettiva.

3. Sfruttare il sistema di caching di Nginx e definire la location contattabile dai client come una location che non esegue direttamente lo handler HLS-NDVR, ma piuttosto un sistema di caching. Tale location demanda in upstream la gestione della richiesta alla location su cui l'HLS-NDVR module è abilitato solo se il contenuto non è cachato.

Tra le tre opzioni si è optato per implementare l'ultima. La prima non convince per la necessità di scrivere tanta logica con il rischio di realizzare comunque un sistema poco efficiente, mentre la seconda sembra offrire pochi vantaggi (esclusivamente quello della personalizzazione del meccanismo di comunicazione tra i moduli) rispetto alla terza. La terza soluzione infatti ha l'enorme vantaggio di sfruttare una caratteristica che è tra le più apprezzate di Nginx, ovvero il sistema di caching altamente efficiente. La

qualità del caching fatto da Nginx è tale che spesso questo server viene usato esclusivamente come web-cache davanti a backend di vario tipo.

In un sistema strutturato come descritto, l'implementazione del caching si traduce in due operazioni:

- Definire una struttura del file di configurazione che permetta di sfruttare il caching Nginx.
- Implementare nel modulo HLS-NDVR un sistema che consenta di eliminare selettivamente gli elementi dalla cache quando le playlist vengono aggiornate.

Il file di configurazione dovrebbe assumere la seguente forma.

La direttiva `proxy_cache_path` consente di configurare tutti i parametri riguardanti il caching, quali il path della directory in cui verranno memorizzati i contenuti cachati, il nome della cache la durata massima di memorizzazione e altro. Per abilitare l'uso di questo sistema di caching si può usare la direttiva `proxy_cache` nella location che gestirà il caching.

Il file di configurazione così strutturato prevede che i client contattino il server sul path definito dal primo server, ovvero `/dvr`, e che qui venga implementato il classico meccanismo di caching di Nginx, basato sull'indicizzazione dei contenuti cachati tramite lo URL. Se il contenuto richiesto non è cachato la richiesta viene passata a un upstream che in realtà è un altro server implementato sullo stesso Nginx e che espone la location sulla quale è abilitato l'HLS-NDVR module. La direttiva `proxy_pass` server proprio a passare una richiesta a un upstram server.

Modifiche nel modulo a support del caching

Per supportare il meccanismo di caching descritto è necessario che qualcuno cancelli dalla cache i contenuti non più validi quando una playlist viene aggiornata. In teoria è necessario implementare un processo o thread che sia in ascolto in background sulle modifiche del file playlist e agisca in risposta a queste con la rimozione delle entry della cache interessate. Questa entità, che chiameremo *cache cleaner*, dovrà essere però informata dall'HLS-NDVR module riguardo gli elementi disponibili in cache: l'HLS-NDVR module conosce tali elementi perché lui stesso li ha generati in risposta a delle richieste dei client.

Una possibile modalità di funzionamento che però divide la singola entità in un'entità per ciascun file di configurazione è la seguente:

```
...
http {
    server {
        listen 80;
        proxy_cache_path /path/to/cache levels=1:2
                                keys_zone=my_cache:10m
                                max_size=10g
                                inactive=60m
                                use_temp_path=off;
        location /dvr {
            proxy_cache my_cache;
            proxy_pass 127.0.0.1:8090/uncached-dvr;
        }
    }
    server {
        listen 8090;
        location /uncached-dvr {
            hls_ndvr on;
            fragments_location /path/to/frags
            ts_location /url/path/to/ts
        }
}
...
```

Figura 4.9. Struttura di un file di configurazione per abilitare il caching sull'HLS-NDVR module [19]

1. L'HLS-NDVR module viene contattato per generare una risposta.
2. Dopo aver prodotto la risposta il modulo avvia un thread fornendogli passandogli due informazioni: lo URL della richiesta, che rappresenterà l'indice del contenuto cachato cui quel thread farà riferimento, e il path al file `.m3u8` le cui modifiche fanno scattare la cancellazione della entry in cache.
3. Il thread si mette in attesa che il file `.m3u8` venga modificato.
4. Il thread si sveglia e cancella la propria entry dalla cache.

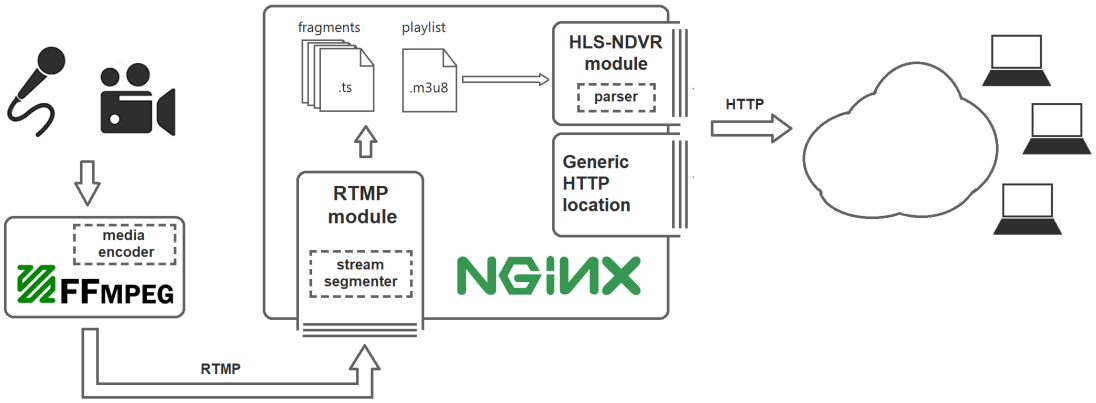

Figura 4.10. Struttura della soluzione di streaming basata su Nginx con RTMP-module e HLS-NDVR module

Un algoritmo di questo tipo può essere implementato utilizzando il supporto che Nginx offre all'utilizzo di *thread pool* creati in fase di configurazione, che consente di delegare task specifici ai thread del pool.

Riguardo la modalità di cancellazione, purtroppo Nginx non offre un metodo per effettuare cancellazione selettiva di entry in cache. Questa funzionalità è offerta da Nginx+ tramite il metodo PURGE di HTTP. Esistono moduli che aggiungono una funzionalità simile sulla versione free di Nginx, ma si tratta di repository datate e poco documentate. Si potrebbe dunque pensare, dato che Nginx implementa il caching tramite la memorizzazione di file su disco, di effettuare la cancellazione cancellando direttamente questi file dal fil system. Il path di questi file può essere facilmente calcolato facendo un hash MD5 dell'indice.

Il modulo attualmente non implementa un sistema di gestione della cache, ma futuri sviluppi possono prevedere una sua realizzazione seguendo lo schema descritto, che è stato progettato per essere il più effieciente possibile nel contesto di esecuzione di Nginx.

Parte III

Test di carico

Capitolo 5

Struttura dei test

5.1 Obiettivi del testing e funzionalità da testare

L'attività di testing è stata svolta con l'obiettivo di valutare la qualità della soluzione di streaming basata su Nginx con RTMP module e HLS-NDVR module, quantificando il numero di client che essa è in grado di servire in determinate condizioni. Oggetto del testing sono state le funzionalità LIVE e NDVR. I test della funzionalità LIVE sono stati effettuati su due tipologie di flusso, audio a 128 kbit/s e video a 1500 kbit/s. In entrambi i casi è stato simulato un certo numero di client connessi a uno streaming trasmesso in diretta segmentato in frammenti da 10 secondi e con playlist avente sliding window di 3 frammenti. La soluzione proposta consente di ottenere una playlist di questo tipo sia dallo RTMP module (agendo sul file di configurazione) sia tramite l'HLS-NDVR module (effettuando una richiesta con parametro `window=3`). I test sono stati eseguiti su entrambe queste configurazioni, così da valutare le prestazioni per i due diversi casi. I test sul NDVR hanno simulato dei client intenti a richiedere 5 minuti di streaming VOD su un flusso audio a 128kbit/s segmentato in frammenti da 2 secondi. La scelta di ridurre la dimensione dei frammenti è comune nei media server. In caso di utilizzo di funzionalità di DVR è infatti prevista che il client effettui delle operazioni di seek, il che vuol dire che desidera posizionarsi in un certo punto del flusso. Una granularità più fine nella segmentazione consente di diminuire il livello di approssimazione nel selezionare l'istante di inizio della riproduzione.

Tutti i test sono stati replicati nelle stesse condizioni sul media-server Wowza. In questo modo si è potuto fare un confronto tra la soluzione proposta

e quella offerta da questo media-server.

Bisogna definire cosa si possa intendere per *fallimento* di un test. Dato che si vuole simulare il comportamento di player multimediali su uno streaming in diretta, si può prendere come riferimento per definire il caso di fallimento il caso in cui il servizio che si vuole garantire non sia più fruibile agli utenti nel modo in cui si vorrebbe che lo fosse. In teoria vorremmo che tutti gli utenti fossero in grado di visualizzare il contenuto trasmesso senza alcuna interruzione e in modo fluido; talvolta potrebbe essere accettato, in condizioni di stress, che qualche utente sperimenti una breve interruzione dovuta alla necessità di scaricare un frammento. In questo caso si parla di bufferizzazione, ed è un segnale di cedimento del sistema. Dato che considerare un test fallito nel momento in cui un qualsiasi client bufferizzi potrebbe essere una metrica eccessivamente rigida ma al contempo considerare il test superato anche se molti client bufferizzano potrebbe significare dare eccessivo margine al Test Plan, si è scelto di definire una soglia, pari a un terzo del numero totale di client disponibili: se il numero di player che sta contemporaneamente bufferizzando supera tale soglia, il test viene considerato fallito.

5.2 Strumenti di testing

La simulazione dei client è stata effettuata utilizzando il software di testing Apache Jmeter. La scelta è stata guidata dalla grande diffusione di questo tool come strumento per effettuare test di carico. Essendo un software molto utilizzato è anche possibile reperire molto materiale riguardo il suo utilizzo. L'estesa documentazione ha senza dubbio semplificato la scrittura e poi il debugging dei test realizzati.

L'obiettivo del test è in entrambi i casi (quello LIVE e quello DVR), simulare il comportamento di client HLS che vogliano riprodurre uno stream. Per farlo questi client decono scaricare i frammenti `.ts`, e lo fanno in modo diversi a seconda del caso d'uso. Analizziamo dunque il comportamento di un player in modalità LIVE.

Un player client HLS richiede come prima cosa la master playlist relativa allo stream. Questa playlist contiene l'elenco dei variant che sono disponibili. Una volta ottenuta la playlist, il player può analizzare le caratteristiche di ciascun variant al fine di selezionare quello che risulta più adeguato alle sue esigenze e alle sue capacità computazionali e di rete. La scelta del variant può essere influenzata da diversi fatto, dunque, e tra questi c'è sicuramente un'eventuale configurazione riguardante questo aspetto eseguita dall'utente.

Selezionato il variant, il player può richiedere la relativa playlist al server. Costruisce lo url della media playlist di interesse ed effettua la richiesta. Il server restituisce la playlist, che rappresenta l'elenco dei frammenti che servono al client per iniziare la riproduzione del contenuto multimediale. Trattandosi di una playlist LIVE, il numero di frammenti contenuto nell'elenco sarà limitato dalla dimensione della sliding window. La dimensione della finestra si può esprimere anche in termini di secondi. Tipicamente una finestra di 30 secondi è adeguata tanto per mantenere la playlist di dimensioni ridotte tanto per offrire uno spazio di buffering adeguato al client.

Dall'elenco dei frammenti, il client può quindi determinare quali siano i frammenti che non possiede e scaricarli. All'inizio della riproduzione, naturalmente, nessuno dei frammenti elencati sarà in possesso del client, il quale dovrà effettuare più di un download. In una fase di regime, invece, mediamente il client deve scaricare un solo frammento alla volta. Una volta ottenuti abbastanza frammenti, il client può iniziare la riproduzione.

Si parla di fase di regime in quanto, dopo una prima fase necessaria a far partire la riproduzone, il player deve continuare a svolgere una serie di azioni previste dal protocollo. Ogni volta che trascorre un certo tempo (pari alla durata di un frammento) dall'ultimo download effettuato deve chiedere nuovamente la playlist: è infatti probabile che sia stato nel frattempo pubblicato un nuovo frammento e che dunque tale playlist risulti aggiornata. Ottenuta la playlist aggiornata si possono facilmente identificare in essa i frammenti che non sono stati già scaricati e rchiederli al server. A meno di ritardi nel download dei precedenti frammenti, il numero di frammenti richiesti dovrebbe essere uno.

Questo funzionamento consente di scaricare periodicamente le più recenti parti del flusso e riprodurle mantenendo un buffer di sicurezza che è pari alla durata complessiva dei frammenti dei quali si è già in possesso ma che non sono ancora stati riprodotti. Questo buffer è molto importante nella logica di funzionamento di un player di questo tipo. Infatti esso non dovrebbe mai essere vuoto. Minore è la quantità di contenuto memorizzata nel buffer minore è il margine di sicurezza che si ha a disposizione, minore è la capacità di continuare a riprodurre contenuto nel caso in cui il tempo richiesto per scaricare un frammento sia maggiore del previsto. Quando un frammento non arriva nel tempo richiesto, se il buffer è vuoto non si può fare altro che aspettare. La riproduzione si interrompe e l'utente sperimenta quella fastidiosa attesa che viene comunemente definita *buffering*.

Riprodurre questo tipo di funzionamento non è semplice. Per quanto implementare questo insieme di funzionalità possa essere per il singolo player

un'operazione davvero elementare, scrivere un test che simuli in modo davvero coerente il comportamento di migliaia di player di questo tipo può richiedere di affrontare delle sfide considerevoli. Il comportamento di questi client dipende infatti da moltissime variabili. Trattandosi di un'applicazione real-time, i tempi seguiti da ciascun client sono di cruciale importanza. Una delle problematiche principali da affrontare riguarda il modo di evitare sincronizzazioni dei client nel richiedere frammenti; la sincronizzazione potrebbe comportare carichi notevoli sul server in specifici momenti del test e periodi di quiete in cui le richieste da gestire sono poche o nulle. La natura asincrona del comportamento dei player non è facilmente replicabile. Infatti ciascun player dovrebbe richiedere la playlist a cadenza periodica ogni volta che scade un timeout di durata predefinita. Invece la richiesta dei frammenti può avvenire a intervalli diversi: la ricezione della playlist dovrebbe infatti avviare un meccanismo di selezione dei frammenti che non sono posseduti e di accodamento della loro richiesta in una FIFO dalla quale un esecutore delle richieste le invia al server. Un meccanismo di questo tipo non è di per sé difficile da implementare (basta avere un minimo di confidenza con la programmazione multithreading); farlo mantenendo un'elevata scalabilità sul client di testing (che deve simulare tutti questi agenti differenti) può essere difficile.

Questa difficoltà di implementazione può portare ad avere risultati inaccurati e che sovrastimano o sottostimano le capacità dei sistemi testati. Uno degli obiettivi della scrittura di test con Jmeter fatta in questo lavoro di tesi è proprio quello di individuare una metodologia di realizzazione per scrivere test che mettano alla prova in modo realistico le funzionalità live di un sistema di streaming per un generico protocollo HTTP-based (non solo per HLS).

5.2.1 Apache JMeter

Apache JMeter è un software open-source puramente scritto in Java pensato appositamente per realizzare test di carico per sistemi di vario tipo e misurare le performance di tali sistemi. Pensato inizialmente per il testing di applicazioni web, è stato in seguito espanso (grazie anche a una serie di plugin) per supportare altri tipi di target system [20].

Realizzare un test script in JMeter significa crere un *TestPlan*. Un TestPlan corrisponde a una serie di step che JMeter eseguirà quando il TestPlan verrà lanciato, e può essere implementato creando un albero di elementi,

ciascuno dei quali è responsabile di una forma di elaborazione dell'input ricevuto dagli elementi precedenti. L'esecuzione degli elementi dell'albero segue un criterio di tipo depth-first, ma l'ordine con cui vengono eseguiti ciascun nodo e i suoi figli dipende dal tipo di elemento che tali nodi rappresentano. Un pannello di configurazione del TestPlan consente di specificare variabili e costanti che potranno essere utilizzate nel test script e file .jar da usare per importare funzionalità necessarie all'esecuzione dei test (ad esempio per importare JDBC, così da realizzare test su database).

Il TestPlan presenta diversi componenti, ma ha sempre almeno un *ThreadGroup*. Guardando al TestPlan come una struttura ad albero che contiene un set di *Controller* e *Sampler*, i ThreadGroup sono gli unici nodi figli diretti del nodo radice (il TestPlan, appunto) e tutti i controller e sampler devono trovarsi nella gerarchia di un ThreadGroup. Ogni ThreadGroup eseguirà indipendentemente dagli altri e autonomamente otterrà delle risposte dal server applicativo. Dal pannello di configurazione di un ThreadGroup è possibile configurare il numero di thread, il tempo di rump-up e quante volte il test dovrà essere eseguito [22].

Il test lancerà dunque N thread (con N numero indicato nella configurazione del thread group) e userà il tempo di rump-up come tempo necessario per arrivare alla situazione di regime, in cui tutti i thread sono attivi. Con tempo di rump-up di T secondi e N thread da avviare, JMeter lancerà N/T thread ogni secondo. Il tempo di rump-up deve essere scelto in modo tale da essere sufficientemente lungo da evitare carichi eccessivi su JMeter all'avvio del test, ma anche sufficientemente corto da evitare che l'ultimo thread parta dopo la fine del primo [22].

JMeter offre due tipi di *Controller* [22]:

- *Samplers*. Inviano una richiesta al server e attendono una risposta. Sono processati nell'ordine in cui appaiono all'interno dell'albero.
- *Logic controllers*. Possono modificare la logica con la quale JMeter decide quando inviare le richieste.

Un test plan può essere composto da altre tipologie di elementi [22]:

- I *Listener* consentono di accedere alle informazioni che raccoglie JMeter durante l'esecuzione dei test. Diversi tipi di Listener forniscono rappresentazioni diverse (es. qualcuno riporta le risposte, altri i tempi di risposta, ecc.). In ogni caso i Listener salvano tutti gli stessi dati; semplicemente la visualizzazione che ne forniscono è diversa.

- I *Timer* consentono di introdurre un delay nel flusso di esecuzione del ThreadGroup. Le Assertion consentono di fare asserzioni riguardo le risposte ricevute dal server, così che si possa testare che il server stia effettivamente restituendo risposte in un formato che è quello atteso.
- I *ConfigurationElement* sono equivalenti ai Sampler, con la differenza che non inviano alcuna richiesta al server. Consentono tuttavia di aggiungere qualcosa alla richiesta o di modificarla. Possono essere specificati in un qualunque punto dell'albero e saranno accessibili da qualsiasi Sampler nel sotto-albero che parte dal nodo di cui sono figli.
- *PreProcessor* e*PostProcessor* eseguono delle operazioni immediatamente prima e immediatamente dopo un elemento Sampler.

Ulteriori dettagli riguardo il funzionamento di questi elementi possono essere recuperati dallo user manual di JMeter all'indirizzo `http://jmeter.apache.org/usermanual/index.html`.

Client HLS con Apache JMeter

Sono stati realizzati due tipi di test plan, uno per la funzionalità LIVE e uno per quella VOD. I Test Plan sono distinti per target, nel senso che quello di Wowza è diverso da quelli usati per testare Nginx con l'RTMP module ed Nginx con entrambi i moduli custom. In ogni caso la loro struttura è identica e verrà discussa una sola volta e in modo indipendente dallo specifico sistema sotto test.

Si ha un unico Thread Group che esegue tutto il lavoro. Le operazioni svolte sono le seguenti.

1. Esecuzione della query di richiesta della master playlist al server. Viene effettuata tramite un *HTTP Request Sampler*, pensato appunto per eseguire richieste HTTP.
2. All'interno dell'HTTP Request Sampler, estrazione dell'elenco di variant offerti per lo stream e il loro inserimento come stringhe in un vettore dedicato. L'estrazione viene fatta tramite un post-processor di tipo *Regular Expression Extractor*.
3. Selezione casuale del variant da utilizzare. La selezione casuale viene fatta tramite un *BeanShell Post-processor*. Si tratta di un particolare tipo di post processor che consente di implementare della logica applicativa in un linguaggio Java-based detto appunto BeanShell.

4. Richiesta della media playlist per il variant selezionato tramite un altro HTTP Request Sampler.

5. Estrazione della lista di segmenti con un Regular Expression Extractor.

6. Elaborazione della lista per valutare quali frammenti devono essere scaricati (quelli non posseduti). Questa operazione non è stata fatta con un BeanShell Post-processor, ma con un JSR223 Post-processor, che utilizza una sintassi che è proprio quella di Java. La scelta è stata obbligata dalla necessità di compiere operazioni non facilmente implementabili con BeanShell.

7. In un controller *ForEach*, richiesta tramite un HTTP Request Sampler di tutti i frammenti di interesse.

8. In un *While* controller compiere le seguenti operazioni:

   (a) Richiedere nuovamente la media playlist tramite un HTTP Request Sampler. Si noti però che tale richiesta viene posticipata da un Timer, in particolare un *JSR223 Timer*. Questo timer serve a posticipare la richiesta in modo da sincronizzarla orientativamente sui 10 secondi. Purtroppo non è possibile garantire che essa non avvenga in più di 10 secondi. Il motivo è che Jmeter non consente di effettuare richieste asincrone, così che bisogna aspettare il completamento delle operazioni di richiesta dei frammenti prima che la richiesta della playlist possa partire; se tali richieste richiedono più di 10 secondi la richiesta viene posticipata. Si noti tuttavia che questo comportamento non compromette eccessivamente la qualità dei test, in quanto la richiesta della playlist richiede tipicamente poco tempo rispetto a quella dei frammenti, così che il tempo aggiuntivo necessario a richiedere la playlist prima di eseguire la nuova richiesta per i frammenti è trascurabile. Anche questo HTTP Request Sampler contiene un Regular Expression Extractor e un BeanShell Post-processor per calcolare la lista di segmenti da scaricare.

   (b) Richiedere tutti i frammenti mancanti tramite un HTTP Request Sampler posizionato in un ForEach controller. In questo particolare HTTP Request Sampler va eseguito il calcolo del tempo che si dovrà trascorrere in attesa prima di eseguire la prossima richiesta; tale tempo viene calcolato come `[durata del frammento] - [tempo di risposta ultimo frammento]`.

Per ridurre al minimo l'aleatorietà dei test nell'eseguirli sui diversi sistemi si è scelto di configurare il server per offrire una sola tipologia di variant, così che la sua selezione casuale non rappresenti un elemento di disturbo per i risultati.

L'ultimo HTTP Request Sampler è importante per due ragioni: non solo calcola il tempo da attendere prima di eseguire la prossima richiesta partendo dal tempo di risposta per l'ultimo frammento, ma è anche il componente che si occupa di tenere traccia della dimensione del buffer. Il buffer è implementato come un intero che indica quanti secondi di ritardo ha acquisito il client simulato da questo thread durante la riproduzione. Ogni volta che un frammento viene scaricato in un tempo maggiore a quello atteso, la differenza tra tale tempo e quello atteso viene sommata all'intero che rappresenta il buffer. Quando tale intero raggiunge la soglia pari alla durata in secondi della playlist, il client viene considerato star bufferizzando.

In assenza di meccanismi di comunicazione più semplici, si è scelto di far sapere agli altri thread che qualcuno sta bufferizzando (questa informazione è necessaria per capire quando il test debba essere considerato fallito e dunque quando si possa terminarlo) tramite la scrittura su un file dedicato a segnalare questa condizione. Ogni thread che si ritrovi con un client in bufferizzazione scrive un carattere (un byte) in questo file.

Ovviamente un client più "rientrare" dal buffering. Rientrare dal buffering vuol dire scaricare un frammento in un tempo inferiore ai 10 secondi. In questo caso, anche se il client ha bufferizzato fino a questo momento si è rimesso in linea con la riproduzione LIVE e dunque l'utente sta nuovamente usufruendo del servizio nel modo corretto. Quando ciò avviene si scrive un carattere (un byte) in un file dedicato.

Il test viene terminato quando il numero di player che sta contemporaneamente bufferizzando raggiunge il 33% del numero di client simulati. Tale numero può essere in ogni momento calcolato come differenza tra la lunghezza in byte del file scritto in caso di bufferizzazione e quella del file scritto in caso di rientro dalla bufferizzazione. Questo meccanismo consente di tener conto della naturale propensione di un sistema di streaming a causare un minimo di buffering lato client in condizioni di stress. Non si considera dunque un problema il fatto che qualche client bufferizzi di tanto in tanto; il problema sorge esclusivamente quando il numero di player in buffering contemporaneamente supera una certa soglia.

5.3 Ambiente di testing

Come ambiente di testing è stato scelto il servizio Google Compute Engine di Google Cloud Platform. Sono stati utilizzati due tipi di macchine virtuali come server:

1. Macchina con 1 vCPU e 3.75 GB di RAM.
2. Macchina con 2 vCPU e 7.5 GB di RAM.

Tutte le macchine server sono di tipologia `n1`, che vuol dire che sono dotate di una virtual CPU implementata come singolo hardware con hyper-threading su un Intel Xeon E5 a 2.6 GHz, o un Itel Xeon E5 v2 a 2.5 GHz, o un Intel Xeon E5 v3 a 2.3 GHz, o un Intel Xeon E5 v4 a 2.2 GHz o un Intel Skylake a 2.0 GHz. I server sono stati istanziati tanto per Nginx quanto per Wowza partendo dall'immagine di *Debian GNU/Linux 9* offerta da Google Compute Engine. Sono state usate macchine distinte per Wowza e per Nginx (ma la stessa macchina con Nginx ha sostenuto i test per l'RTMP module e per l'HLS-NDVR module). È stata usata la versione `4.7.4` di *Wowza Streaming Engine* e la `1.12.2` (stable version al 27 Marzo 2018) per Nginx.

Per la simulazione dei client è stato utilizzato il software Apache Jmeter, eseguito anch'esso su Google Compute Engine in modalità distribuita. Lavorare in modalità distribuita con JMeter significa replicare l'esecuzione di un Test Plan su diverse macchine server, che chiameremo JMeter-server. I JMeter-server hanno il compito di eseguire il Test Plan, e dunque rappresentano gli agenti che inviano le richieste al server applicativo. Nessuno di essi è a conoscenza dell'esistenza degli altri, ma agiscono in modo completamente autonomo. Per gestire l'esecuzione del Test Plan sui JMeter-server è necessario definire un JMeter-client; questi è il nodo della rete che avvia il test. In realtà i JMeter-server non possiedono una copia del Test Plan, ma la ricevono proprio dal JMeter-client. Dunque i JMeter-server sono a conoscenza dell'esistenza di un JMeter-client, ricevono da esso il Test Plane e inviano ad esso i risultati della sua esecuzione [23]. Il JMeter-client può scegliere di aggregare i risultati in arrivo dai diversi server in modi diversi. Il metodo di aggregazione più comune è quello *statistico*, che prevede appunto di combinare tramite un'analisi statistica questi risultati. Le cinque macchine JMeter-server sono state configurate con 4 vCPU e 15 GB di RAM.

Riguardo la disposizione geografica di queste macchine vituali, si è scelto (per trovare un compromesso tra la necessità di mantenere il test realistico introducendo un elemento di distanza geografica e esigenze legate ai costi

di rete) di posizionare le macchine in aree geografiche diverse negli Stati Uniti. La zona in cui si trovano i server è la `us-east1-b`, mentre tutti i JMeter-server e il JMeter-client si trovano nella zona `us-central1-c`. È stato verificato che i tempi di risposta del server fossero per i client posizionati in questa zona quantomeno paragonabili a quelli ottenuti da un player in esecuzione su un browser di un PC a Torino.

Il dimensionamento di questi sistemi è dipeso principalmente da alcuni limiti imposti da Google Cloud Platform sul numero di vCPU attive e sul numero di IP address utilizzati contemporaneamente in una certa zona (un data center) e dall'impossibilità (dato l'hardware disponibile) di mantenere realistico il comportamento dei player client HLS al crescere del loro numero oltre una certa soglia. Se infatti il numero di thread da avviare su ciascun Test Plan (quindi su un JMeter-server) cresce, cresce anche l'instabilità del sistema nell'eseguire le richieste; volendo garantire la simulazione di una situazione quanto più possibile costante e dunque replicabile su tutti i sistemi allo stesso modo ci è scontrati con l'impossibilità di JMeter, in questa configurazione, di gestire più di 700 thread per macchina server; oltre questo numero le richieste non vengono più esaurite in modo costante nel tempo (come ci si aspetterebbe in condizione di regime), ma aumentano e diminuiscono causando momenti di picco che il server (tanto Nginx quanto Wowza) non può essere in grado di gestire. Dati questi limiti, i test effettuati sono stati in grado di simulare fino a 3500 client attivi simultaneamente. In alcuni casi (come quello dello streaming audio a 128 kbit/s) questo ha significato l'impossibilità di portare i server al punto rottura. In altri (nel caso dello streaming video a 1500 kbit/s) il numero di client si è rivelato adeguato a simulare un carico considerevole sui server, ma i limiti di banda hanno rappresentato un problema.

Capitolo 6

Risultati

L'analisi dei risultati si è concentrata principalmente su due elementi:

1. La capacità del server di continuare a servire i client anche in condizioni di carico molto elevato.

2. La velocità nel fornire ai client i frammenti che compongono lo stream.

I risultati ottenuti confermano che la scalabilità e la velocità di Nginx possono trovare concreta applicazione nel contesto dello streaming su HTTP. Nel servire uno streaming audio a 128 kbit/s in modalità LIVE Nginx risulta 1,5 volte più veloce di Wowza con 2500 client connessi contemporaneamente, addirittura 8 volte più veloce con 3500 client. Anche con un flusso più pesante come quello video a 1500 kbit/s Nginx serve i frammenti almeno 4 volte più rapidamente di Wowza, il quale non riesce a gestire più di 2500 client senza che questi sperimentino disagi legati alla bufferizzazione del video. In tutti questi casi il consumo di CPU è dalle 4 alle 6 volte inferiore rispetto alla controparte.

La funzionalità di DVR è stata testata sul solo flusso audio e ha evidenziato le differenze più rilevanti rispetto al media-server. Il tempo di risposta medio per servire i frammenti resta praticamente costante tra i 75 e gli 80 ms tanto con 2500 quanto con 3500 client, mentre per Wowza varia tra i 6 e gli 8 secondi. Nell'interpretare questi risultati si devono tenere in considerazione la natura del test effettuato (il download completo di un contenuto VOD) e il numero molto maggiore di funzionalità che Wowza aggiunge al proprio servizio DVR, tra le quali la capacità di lavorare con altri protocolli oltre ad HLS e un filtro per identificare le richieste degli utenti.

6.0.1 Terminologia e interpretazione dei risultati

La terminologia usata durante i test differisce leggermente rispetto a quella standard utilizzata da HLS. Si utilizzano infatti i seguenti termini:

- *playlist*, per indicare una *master playlist*.
- *chunk*, per indicare una *media playlist*.
- *stream*, per indicare un *fragment*.

Questa terminologia è quella utilizzata all'interno di JMeter nei Test Plan. Potrebbe quindi ricorrere in grafici e figure.

6.1 Test della funzionalità LIVE

I test effettuati servendo playlist LIVE hanno cercato di riprodurre quanto più possibile il comportamento di un comune player HLS. Il comportamento dei client è lo stesso sia che si stia lavorando con l'audio sia che si stia lavorando con il video. La differenza fondamentale nei due casi sta nella dimensione dei frammenti. Con il video a 1500 kbit/s i frammenti `.ts` sono di dimensioni molto maggiori rispetto al caso del test sull'audio a 128 kbit/s. Questo si traduce sia in un aumento del tempo necessario alla trasmissione del frammento, che comporta il fatto che le connessioni con i client restino aperte per molto tempo, sia in un aumento di banda occupata, che come si specifica nel seguito ha causato problemi nell'analizzare i risultati dei test.

Si specifica che durante l'esecuzione di questa tipologia di test è stato inviato al server il flusso audio o video tramite `ffmpeg` in esecuzione su un'ulteriore macchina esterna al sistema di testing descritto. L'elaborazione di questo flusso per farne segmentazione richiede ovviamente l'impiego di parte delle risorse computazionali del server, cosa che è stata tenuta in considerazione nell'analizzare i risultati in termini di consumo di CPU.

6.1.1 Test su audio a 128 kbit/s

Nel caso dei test audio, i frammenti di 10 secondi sono molto leggeri. Si tratta di file statici di piccole dimensioni che possono essere serviti tanto da Nginx quanto da Wowza in modo molto rapido. Essendo Nginx un webserver notoriamente molto veloce nel caso in cui si vogliano servire contenuti di questo tipo, ci si può aspettare che esso ottenga degli ottimi risultati in questi test.

A causa della dimensione ridotta dei frammenti, in nessuno dei casi in esame si è riuscito, con questo tipo di flusso, a causare la caduta dei server sotto test. Tuttavia con 3500 client Wowza inizia a sperimentare disagi sufficienti a portare, in alcuni momenti di maggiore stress, alla chiusura di alcune connessioni prima di fornire la risposta ai client. In tutte le altre circostanze, tanto Nginx quanto Wowza riescono sempre nel complesso a reggere il carico nelle condizioni prese in esame. A causa dei limiti dell'ambiente di testing dei quali si è parlato in Sezione 5.3, inoltre, è stato impossibile far crescere il numero di client al di sopra di una certa soglia. Per queste ragioni non si è potuto far crescere ulteriormente il numero di client per portare i server ancor di più in prossimità dei propri limiti fisici. Tuttavia si è potuto fare un confronto approfondito a parità di condizioni.

Da questi primi test è in generale risultato evidente l'uso molto più intenso della CPU fatto da Wowza. Anche l'utilizzo di memoria è estremamente elevato rispetto ad Nginx. Questo era prevedibile, soprattutto a causa del fatto che Wowza utilizza Java come main-language mentre Nginx è interamente scritto in C.

Inoltre Nginx risulta decisamente più veloce di Wowza in tutti i test eseguiti, sia che lavori solo tramte lo RTMP module sia che gestisca le richieste LIVE passando per lo HLS-NDVR module.

Le due tipologie di test effettuate sono le seguenti:

A. 500 thread su 5 JMeter-server e rampa di 10 thread al secondo. Ogni thread corrisponde a un client. Sono dunque 2500 client HLS contemporaneamente attivi e che cercano di riprodurre lo streaming LIVE.

B. 700 thread su 5 JMeter-server e rampa di 10 thread al secondo. Ogni thread corrisponde a un client. Sono 3500 client HLS contemporaneamente attivi e che cercano di riprodurre lo streaming LIVE.

Dai risultati prodotti si può riassumere quanto segue.

1. Wowza consuma molta CPU e molta memoria. Questo porta il server talvolta a rispondere più lentamente ai client. Nel test di tipo B (3500 client) molti client simulati dai thread sui JMeter-server hanno bufferizzato, salvo recuperare successivamente, mentre 27 volte la connessione è stata chiusa dal server senza fornire una risposta. Ci sono alcuni picchi in cui il download di un fragment ha richiesto 35 secondi. Se con Nginx non riusciamo a portare l'utilizzo della CPU da parte del server al 100%, con Wowza ci sempre, in tutti i test effettuati. Si può ipotizzare

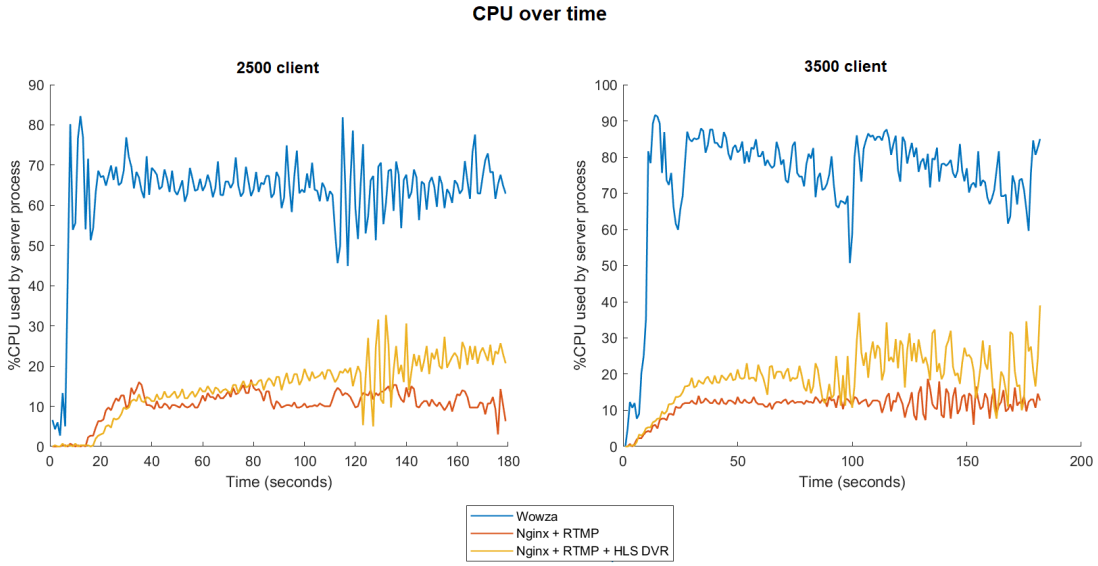

Figura 6.1. Confronto tra l'utilizzo di CPU nei test audio sui diversi server

che esista un legame tra le scarse prestazioni di Wowza in specifici momenti dell'esecuzione e la possibile esigenza d attivare periodicamente il garbage collector di Java. Questo spiegherebbe i picchi periodici che si presentano nei tempi di risposta. Un confronto tra i tempo di CPU richiesti è mostrato in Figura 6.1.

2. Nginx è più veloce a servire contenuti statici rispetto a Wowza, e su frammenti di dimensioni piccole funziona particolarmente bene. Nessuno dei test ha causato buffering da parte di alcuno dei client simulati. Questo lascia intuire che in queste condizioni il server è ancora lontano dai propri limiti fisici. In Figura 6.2 e in Figura 6.3 è mostrato un confronto tra le distribuzioni dei tempi di risposta di Nginx e di Wowza nei casi presi in esame.

3. Il modulo HLS-NDVR lavora in modo tale che man mano che aumenta il contenuto registrato sul server aumenta anche il tempo necessario per eseguire il calcolo della playlist personalizzata da restituire al client. L'uso di CPU è decisamente più altalenante rispetto al caso dell'utilizzo del solo RTMP module. Sicuramente viene usata più CPU, ma l'utilizzo sale anche gradualmente all'aumentare della lunghezza della playlist registrata. A 6 minuti dall'inizio dello streaming il server (non il processo Nginx) usa complessivamente il 25,6% della CPU, con una percentuale

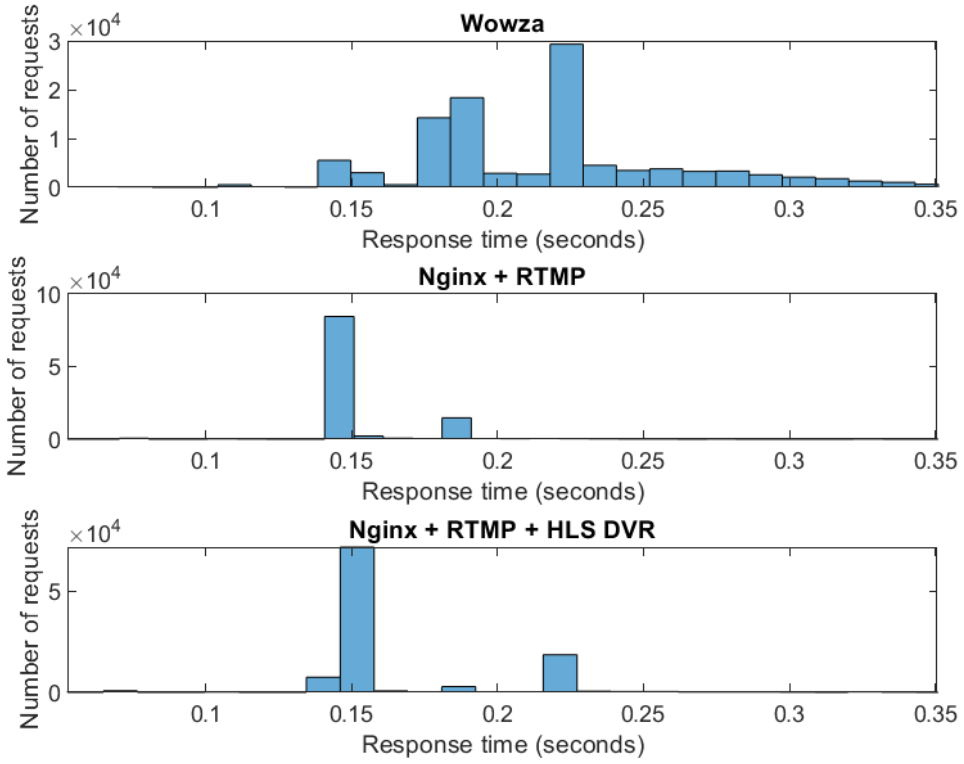

Figura 6.2. Confronto tra le distribuzioni dei tempi di risposta nei test audio con 2500 client

del 20% riportata dal tool `top` per il worker-process Nginx. L'aumento nell'utilizzo di CPU sperimentato da Nginx in queste condizioni è chiaramente visibile in Figura 6.1, ed è facilmente spiegabile se si analizza il modo in cui lo HLS-NDVR module crea le playlist: esso parte dall'inizio della playlist `.m3u8` da parsare e la scorre per mantenere esclusivamente quelle righe che devono far parte del contenuto da restituire al client. Maggiore è la lunghezza della playlist (dunque la quantità di contenuto registrato sul server), maggiore è il tempo che tale operazione richiede.

Questa limitazione non sembra inficiare particolarmente le prestazioni del sistema: i client registrano un tempo di download dei frammenti che è quasi equivalente al caso in cui la playlist venga richiesta senza l'intervento dello HLS-NDVR module, e un tempo di download delle playlist stessa tra i 130ms e i 190ms. Probabilmente il sistema è ancora poco stressato, il che significa che pur se il tempo di elaborazione delle

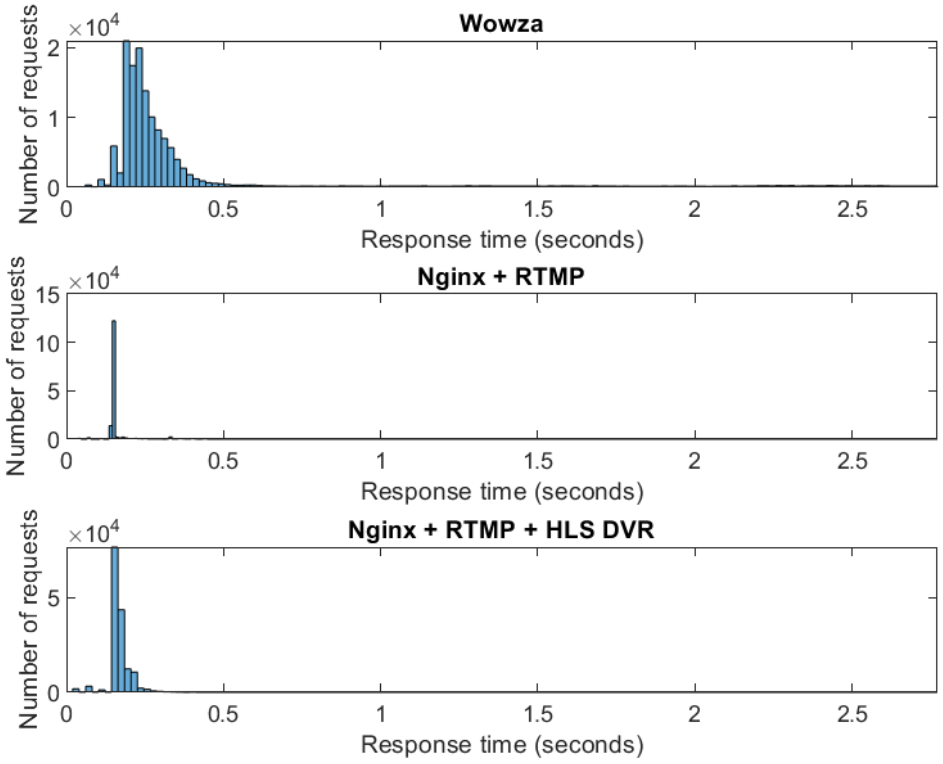

Figura 6.3. Confronto tra le distribuzioni dei tempi di risposta nei test audio con 3500 client

playlist aumenta questo non causano la compromissione della capacità del sistema di garantire il servizio ai client. Anche in questo caso, infatti, nessuno dei JMeter-server ha bufferizzato.

L'andamento crescente della percentuale di CPU complessivamente utilizzata lascia supporre che essa raggiunga il100% quando la quantità di stream registrato equivale a circa un'ora di contenuto. Si consiglia quindi di non superare questa soglia.

Il comportamento di Wowza nel tentare di servire i 3500 client e la chiusura delle connessioni che alcuni di essi sperimentano suggeriscono che questo possa essere considerato un limite superiore al numero di player che possono contemporaneamente essere serviti dal media-server in questa configurazione. D'altra parte, nelle stesse condizioni Nginx non sembra mostrare problemi anche con il più alto numero di client che si è riusciti a simulare e il basso

uso di CPU e memoria lascia supporre che tale numero possa ancora crescere senza compromettere la corretta fornitura dell servizio da parte del server.

6.1.2 Test su video a 1500 kbit/s

Per i test video, la dimensione dei frammenti è molto più elevata. Il loro download richiede dunque un tempo considerevole. Questo occupa CPU e banda e mantiene attive le connessioni per molto tempo.

I risultati dei test di questo tipo sono stati influenzati da un problema legato alla banda disponibile per i server sul cloud. Tale banda si limita a 2 Gbit/s per ciascuna vCPU disponibile sulla macchina. Di conseguenza il numero di richieste che possono essere servite contemporaneamente è limitato da questo vincolo.

Il tool `iperf` ha consentito di stimare la banda disponibile in uscita per i server da 1 vCPU e 3,75 GB di memoria, che è di 1,9 Gbit/s. Si può valutare che:

```
1.9 Gbit/s = 1992294 kbit/s =>
(1992294 kbit/s)/(1500kbit/s) = 1330 (circa)
```

1330 è il numero di client che il sistema dovrebbe essere in grado di servire contemporaneamente senza il verificarsi di disagi legati a interruzione per gli utenti. Naturalmente questa è una stima, ma ha trovato un riscontro nei test effettuati. Data l'impossibilità di superare questa soglia in termini di numero di client servibili, i test sulla macchina server con 1 vCPU e 3.75 GB di memoria hanno dato risultati difficilmente interpretabili e influenzati da questo limite di banda. Ci si è quindi spostati sulla macchina con 2 vCPU.

La macchina server con 2 vCPU e 7.5 GB di memoria dovrebbe garantire una banda più adeguata alle esigenze. Tale banda è stata stimata (sempre tramite il tool `iperf`) essere di 3.9 Gbit/s. Ripetendo l'analisi precedente, si stima che il server possa servire senza problemi un numero di client pari a:

```
3.9 Gbit/s = 4089446 kbit/s =>
(4089446 kbit/s)/(1500kbit/s) = 2726 (circa)
```

Test con 2500 client

Un primo test in questa configurazione ha caricato i server con 2500 client. In questo caso siamo al limite dell'utilizzo di banda disponibile. Nginx ha retto il carico, ma con qualche difficoltà. Mediamente impiega 10 secondi per servire i frammenti ai client, e infatti ci sono dei casi in cui qualche client

inizia a bufferizzare in attesa del frammento successivo. Tuttavia il test non fallisce, in quanto i client riescono a recuperare rientrando della bufferizzazione. Quindi il test viene superato, ma qualche utente può sperimentare disagi legati all'attesa del completamento del download di alcuni frammenti. L'uso della CPU da parte dei worker-process di Nginx (ci sono 2 worker-process, uno per ciascuna vCPU) è per la maggior parte del tempo tra il 54% e il 60%. La percentuale di CPU richiesta dai processi Nginx è dunque nel complesso di poco inferiore al 120%.

Wowza d'altro canto non supera il test, in quanto il tempo di risposta medio sale quasi subito intorno agli 11 secondi, così che molti client iniziano a bufferizzare. A causa dell'elevato numero di player in buffering, i JMeter-server si fermano quasi subito (non appena 1/3 dei client simulati nel loro Test Plan bufferizza). Il consumo di CPU da parte del processo di Wowza è molto, oltre il 180%.

Sia Wowza sia Nginx usano praticamente tutta la banda disponibile.

Il test è stato infine effettuato sulla configurazione che prevede di richiedere playlist LIVE al modulo HLS-NDVR configurato su Nginx. Quando si effettuano le richieste a quest modulo, Nginx si comporta esattamente come Wowza, ovvero termina a causa della bufferizzazione, ma richiede che passi un tempo maggiore prima che il test fallisca. Il tempo medio di risposta sale anche qui sopra i 10 secondi (10.57 secondi) dopo poco tempo e questo porta i client a bufferizzare. La CPU è utilizzata più del caso di Nginx nella versione con il solo modulo RTMP, ovvero sopra il 60% da entrambi i worker-process, per un totale del 120% richiesto dai processi Nginx.

L'impressione è che in questo test l'eccessiva prossimità al limite di banda possa compromettere l'accuratezza dei risultati. Sebbene lutilizzo di CPU da parte di Nginx sia molto inferiore a quello di Wowza, quest'ultimo non ha chiuso prematuramente alcuna connessione, continuando a servire i client (anche se impiegando molto tempo). Il test è terminato a causa della bufferizzazione da parte dei client, non perché il server abbia smesso di rispondere correttamente. Non possiamo dunque esser certi che la lentezza nel servire i frammenti ai client sia realmente dovuta all'incapacità del sistema di reggere il carico di elaborazione cui è stato sottoposto; potrebbe più semplicemente essere stata causata da limiti di banda di rete disponibile.

Il modo abbastanza particolare in cui variano i tempi di risposta per i test fatti su Nginx lavorando direttamente con lo RTMP module sembra suggerire la prossimità di una condizione particolare (si veda Figura 6.4).

Per avere risultati più affidabili, si è quindi effettuato un test con un numero minore di client al fine di confrontare i risultati in condizioni di

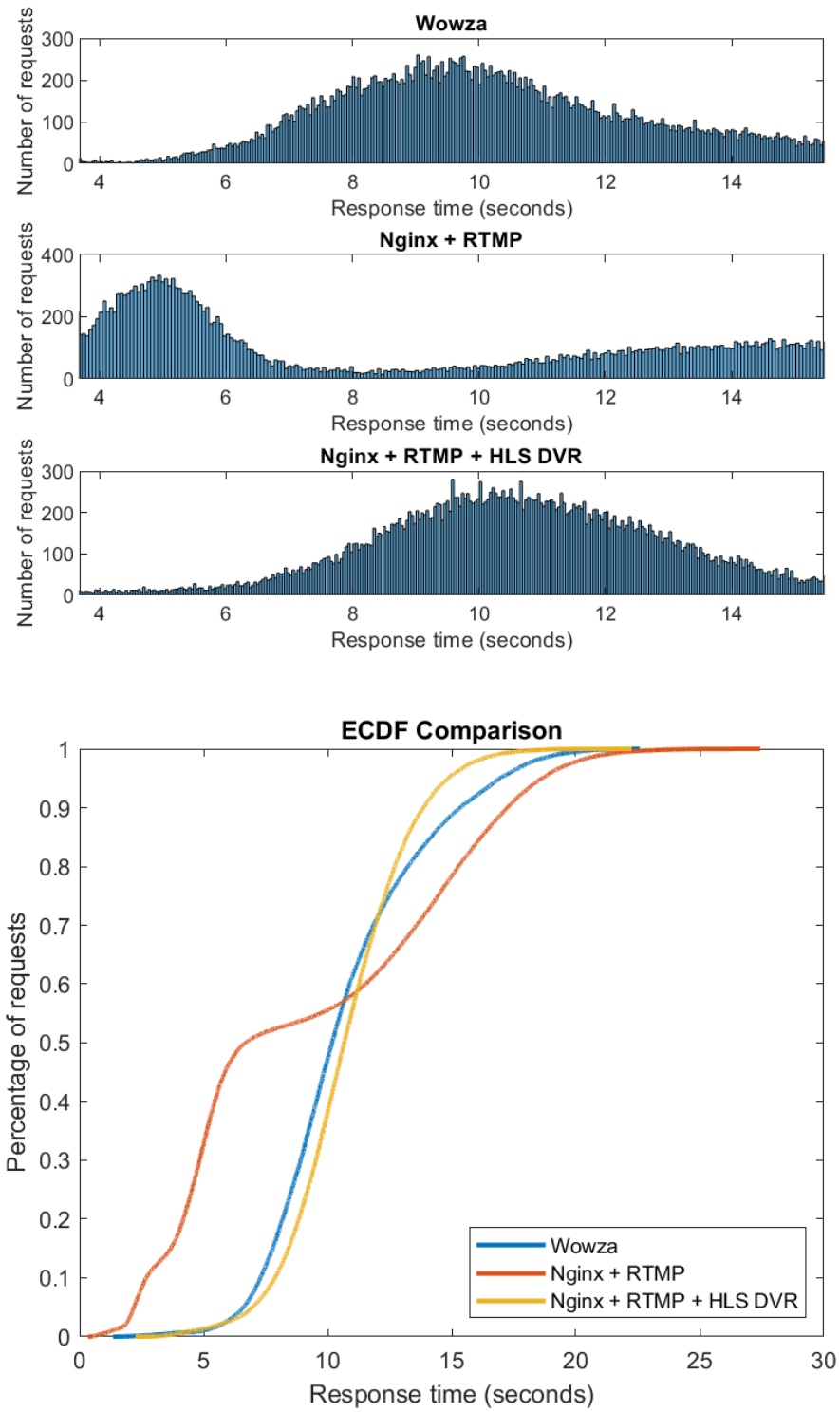

Figura 6.4. Confronto tra i tempi di risposta per i frammenti con streaming video a 1500 kbit/s

maggior stabilità.

Test con 2000 client

Il test viene superato sia da Wowza sia da Nginx in entrambe le configurazioni. I tempi medi per servire i frammenti sono di 5,2 secondi per Wowza, 1,5 per Nginx con il solo RTMP module e 1,8 per Nginx con la modalità LIVE dello HLS-NDVR module.

In Figura 6.5 è mostrata la tendenza di Nginx ad essere non solo più rapido nel servire i frammenti, ma anche più stabile nel farlo. Il tempo medio di download di questi oscilla in modo molto più significativo nel caso di Wowza, come mostrato in Figura 6.6.

Nginx sembra dunque essere in grado di sostenere meglio il carico non solo perché il tempo medio di risposta si mantiene più basso, ma anche perché esso subisce variazioni molto minori rispetto a quello di Wowza.

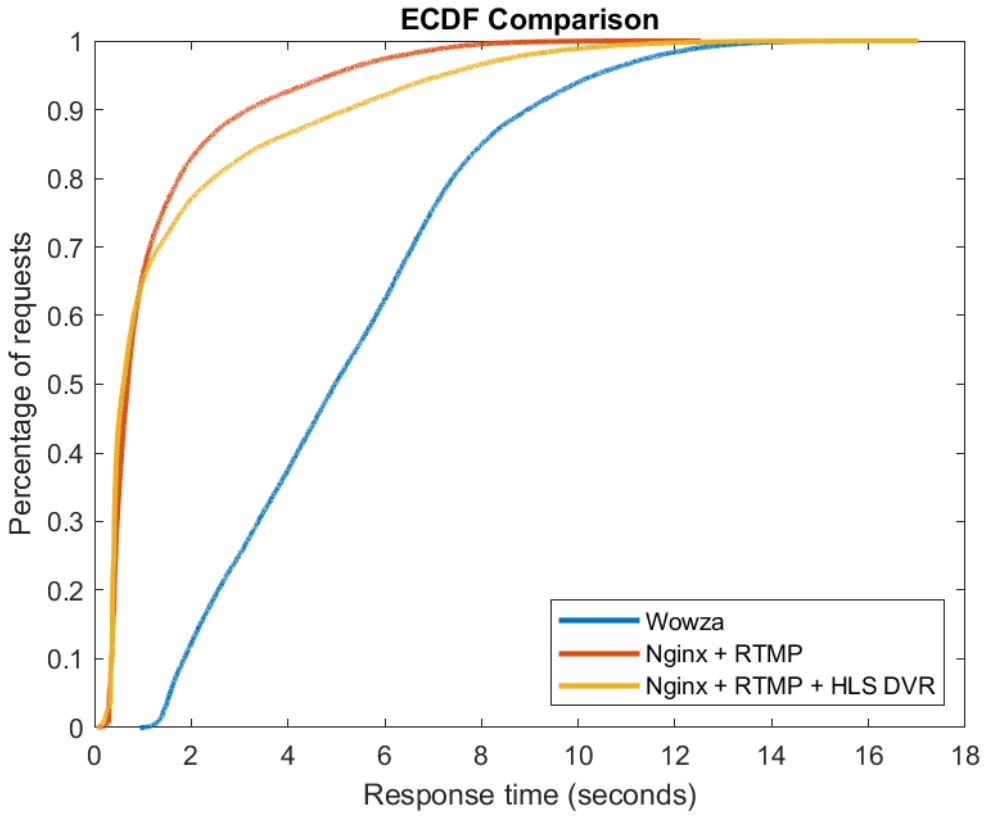

Figura 6.5. Confronto tra le ECDF dei tempi di risposta

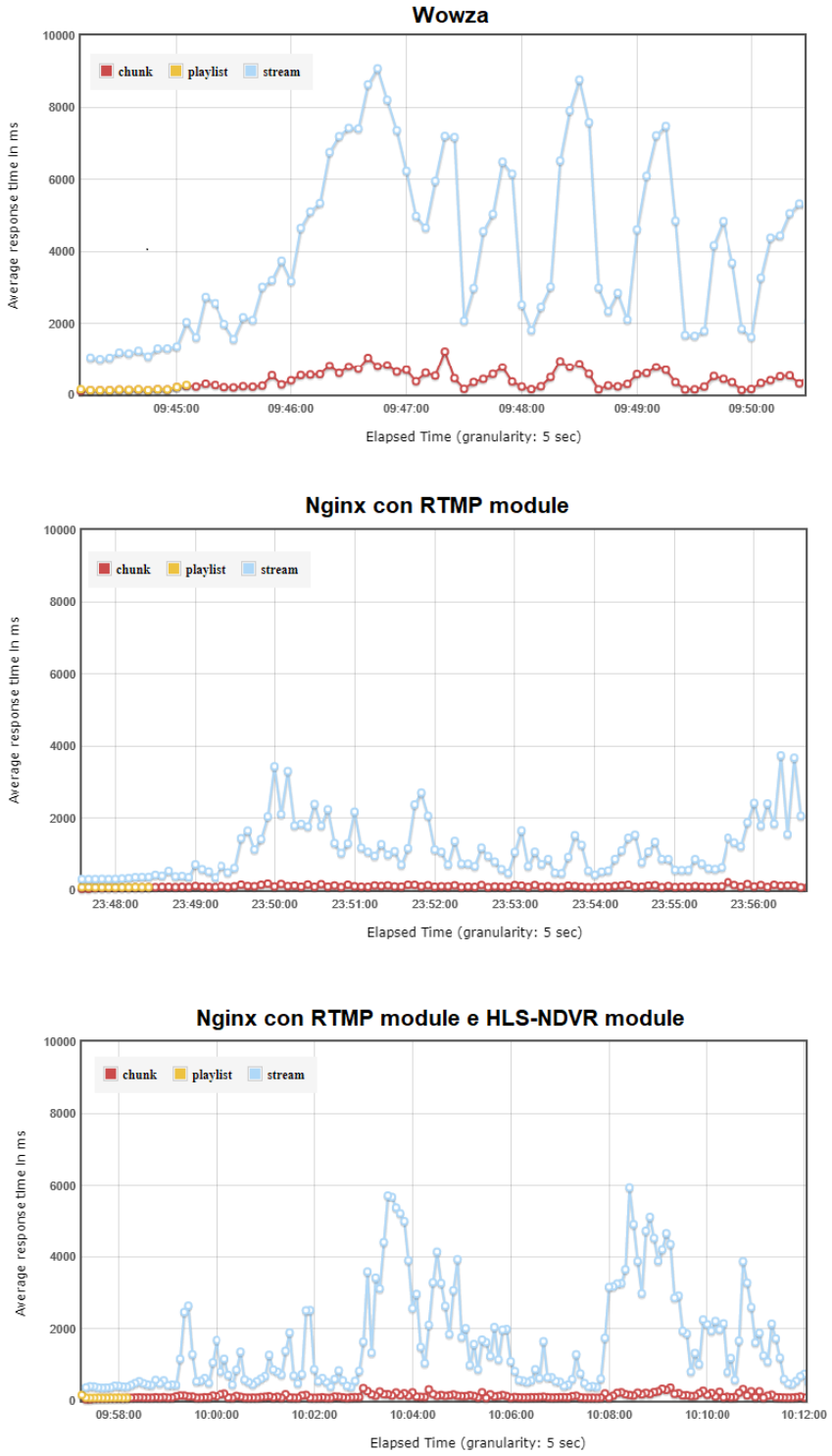

Figura 6.6. Andamento medio dei tempi di risposta per Wowza e Nginx senza e con l'uso dello HLS-NDVR module

Per Wowza, il tempo medio di download dei fragment oscilla vistosamente tra i 2 e i 9 secondi a regime; tuttavia nessuno dei client bufferizza. L'uso della CPU è intorno al 150%, quasi quattro volte quello di Nginx senza HLS-NDVR module e circa due volte Nginx con lo HLS-NDVR module.

Anche se riesce a reggere senza problemi il carico, lo HLS-NDVR module sperimenta anche in questo caso un vistoso aumento della percentuale di CPU utilizzata all'aumentare della lunghezza della playlist. Si noti tuttavia che la velocità con la quale questo incremento avviene è la stessa che è stata riscontrata nei test audio: essa non dipende in alcun modo dal tipo di stream. Il tempo di download dei fragment, che pure va aumentando, resta comunque circa quattro volte inferiore a quello di Wowza, e oscilla tra i 2 e gli 8 secondi.

Questo di 2000 client potrebbe essere considerato un limite superiore per Wowza. Per Nginx con il modulo HLS-NDVR questo limite potrebbe probabilmente essere superato. Tuttavia andrebbe considerato il limite legato alla durata della registrazione, perché maggiore è la lunghezza del contenuto registrato maggiore è il tempo per elaborare le playlist.

6.2 Test della funzionalità DVR

Nelt testare le funzionalità DVR si è scelto di lavorare con il solo flusso audio e con il server dotato di una sola vCPU e di 3.75 GB d memoria. La ragione è che la funzionalità DVR di Wowza è davvero poco performante e riesce a reggere un carico molto ridotto. Di conseguenza si è ritenuto poco utile effettuare il confronto su un carico come quello rappresentato dal video a 1500 kbit/s, molto pesante già in uno streaming LIVE.

Il tipo di test prevede (per semplicità) il download sequenziale dei frammenti che compongono una playlist. Non è un comportamento tipico dei player, ma ci si avvicina. In ogni caso simula il download di una risorsa dal server partendo da un contenuto multimediale registrato. In questo caso non si ha nessun media encoder `ffmpeg` che invii il proprio flusso RTMP al server, ma si suppone che l'intero stream sia già registrato.

I due tipi di test effettuati sono i seguenti:

A. 500 thread su 5 JMeter-server e rampa di 10 thread al secondo. Si hanno dunque 2500 client intenti ad effettuare il download sequenziale dei frammenti contenuti in un VOD la cui playlist viene generata dinamicamente.

B. 700 thread su 5 JMeter-server e rampa di 10 thread al secondo. Sono 3500 client intenti ad effettuare il download sequenziale dei frammenti contenuti in un VOD la cui playlist viene generata dinamicamente.

Sui server è stato registrato un audio di 20 minuti. Le query effettuate dai client sono costruite definendo un istante di partenza e una durata, con durata fissa a 300 secondi e istante di partenza scelto randomicamente tra 0 e 600 secondi. Si ha la garanzia, in questo modo, che i client selezionino porzioni abbastanza diverse tra loro ma che non superino mai i limiti dello stream nel richiedere i frammenti.

Dato che i frammenti hanno durata di 2 secondi, ci sono 150 frammenti elencati in ciascuna playlist richiesta da un client. Questo determinismo è necessario per evitare eccessiva aleatorietà tra i test eseguiti su Wowza e quelli eseguiti su Nginx.

Lo HLS-NDVR module riesce a reggere il carico. Esso spende una quantità di tempo abbastanza alta nell'elaborazione della master playlist; questo tempo è sempre maggiore di quello che spende nel generare le media playlist. Questo risultato non è del tutto inaspettato: si spiega facilmente con il fatto che l'accesso al file `.m3u8` che rappresenta la media playlist da parsare (quella che contiene l'elenco dei frammenti) viene fatto in modo diretto, mentre per ottenere la master playlist noto il nome dello stream è necessario scorrere le entry della directory contenente le playlist nel file system, operazione che fa sicuramente perdere un certo quantitativo di tempo. I tempi di risposta sono sia per il test A sia per il B estremamente bassi e, soprattutto, molto simili, e si attestano intorno a, circa 76 ms per gli stream e per le master playlist e circa 40 per le media playlist. Sono valori molto bassi, considerando il carico notevole che il server subisce in questo test (molte connessioni in contemporanea senza intervalli dovuti a temporizzazioni da parte dei client).

Cosa interessante, l'uso della CPU non supera mai il 20% in entrambi i test.

Wowza soffre molto sia nel test A sia nel test B. Arriva a richiedere mediamente 3,3 secondi per il download di ciascun fragment per il test A e oltre 6,3 secondi per il test B. La CPU utilizzata è sempre tra l'85% e il 90%. Come negli altri test, Wowza usa anche moltissima memoria.

Questi risultati evidenziano come in un applicazione come quella testata, che preveda un utilizzo del media server come sorgente di contenuti VOD generati dinamicamente partendo da un contenuto registrato, Nginx si confermi essere estremamente più scalabile di Wowza.

Le ragioni che motivano questa maggior capacità di Nginx di sostenere un carico di questo tipo possono essere molteplici. Un'ipotesi molto plausibile è che qui si senta la differenza tra Java e C in termini di prestazioni. C'è tuttavia anche da considerare quanto segue riguardo Wowza.

1. Effettua dell'elaborazione aggiuntiva filtrando le richieste per gestire dei tag che identifichino gli utenti e allegarli agli URL. Questo consente di tracciare i client, cosa interessante e utile dato che stiamo offrendo loro la possibilità di customizzare la playlist ricevuta. Un'operazione di questo tipo, per quanto possa essere realizzata in modo efficiente, sicuramente ha un impatto sulle prestazioni. Infatti gli stessi frammenti `.ts` che compongono lo stream vengono rinominati (solo virtualmente) concatenandovi l'identificativo del client.

2. È ipotizzabile che non memorizzi i fragment così come li serve ai client, ma in un formato che gli consenta di renderli disponibili in tutti i diversi protocolli supportati. Ne deriva che il media server deve pre-elaborare questi frammenti, prima di renderli disponibili alle richieste dei client.

Dalla Figura 6.7 risulta evidente quanto siano diverse in termini prestazionali la soluzione basata su Nginx e quella basata su Wowza.

6.3 Riepilogo dei risultati ottenuti

In Tabella 6.1 è mostrato un riepilogo dei risultati ottenuti. In Tabella, *Res.Time* indica il tempo di risposta per i frammenti, *Failures* il numero di fallimenti complessivi sul totale delle richieste. *A* e *B* indicano le tipologie di server utilizzato: A indica il server con 1 vCPU e 3.75 GB di memoria, mentre B indica il server con 2 vCPU e 7.5 GB di memoria.

6.4 Conclusioni riguardo il modulo HLS-NDVR

I test mostrano che l'implementazione del modulo risulta piuttosto efficiente rispetto a quella equivalente fatta da Wowza quando si tratta di delimitare la playlist basandosi su un istante iniziale e su uno finale. Tuttavia viene evidenziata una scarsa efficienza nella gestione della CPU quando si applica una sliding window di dimensione arbitraria.

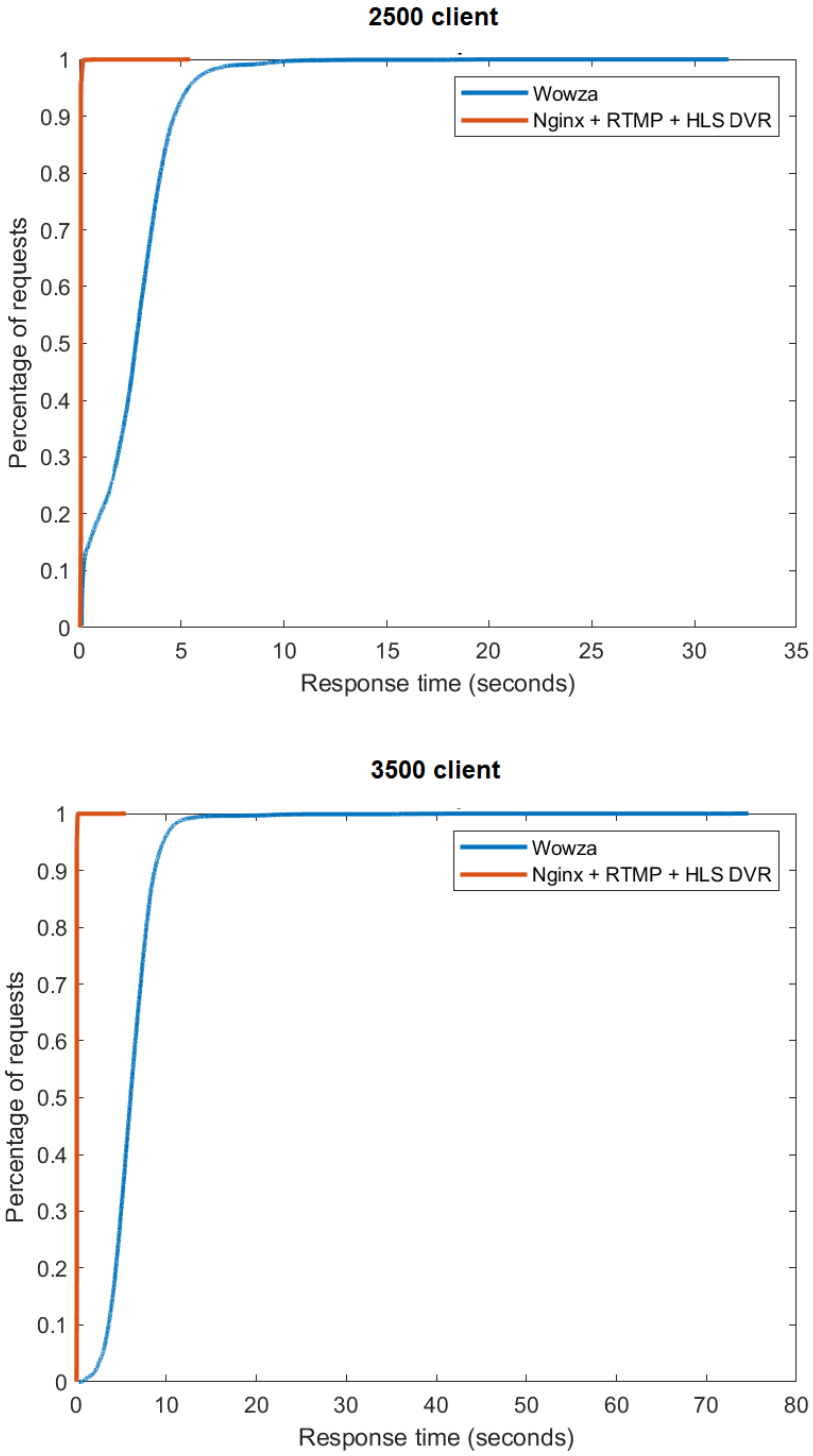

Figura 6.7. Andamento medio dei tempi di risposta per il modulo DVR di Wowza e per lo HLS-NDVR module di Nginx

Tabella 6.1. Riepilogo dei risultati dei test

		TEST					
		LIVE				DVR	
		Audio 128 kbps		Video 1500 kbps		Audio 128 kbps	
		2500	3500	2000	2500	2500	3500
		A	A	B	B	A	A
Wowza	Failures	0%	0.005%	0%	0%	0%	0%
	Res.Time	0.22 s	1.05 s	5.2 s	10.7 s	3.3 s	6.2 s
	CPU%	60 %	75.7 %	155 %	160 %	87 %	75 %
	Mem%	37 %	49.8 %	24.6 %	24.4 %	73 %	85%
Nginx RTMP module	Failures	0 %	0%	0%	0%	/	
	Res.Time	0.15 s	0.16 s	1.2 s	9.5 s	/	
	CPU%	10 %	11.2 %	40 %	85 %	/	
	Mem%	1 %	1 %	1.2 %	2.1 %	/	
Nginx HLS-NDVR module	Failures	0 %	0%	0%	0%	0%	0%
	Res.Time	0.15 s	0.17 %	1.8 s	10.6 s	0.08 s	0.09 s
	CPU%	10 %	22 %	80 %	92%	14 %	14 %
	Mem%	1 %	1 %	1.6 %	1.9 %	1 %	1.1 %

Come descritto in precedenza, questa carenza deriva essenzialmente dal modo in cui il modulo effettua il parsing della playlist `.m3u8` in questa circostanza, ovvero partendo dall'inizio della stessa e scorrendola interamente aggiornando man mano il contenuto della sliding window, che è la porzione di playlist che verrà restituita, con l'aggiunta di un nuovo frammento in coda e la rimozione di quello più vecchio dalla testa. La scarsa efficienza di questa modalità di elaborazione è evidente. Il suo impatto è stato sottovalutato in fase di progettazione e sviluppo.

Un possibile miglioramento nello sviluppo del modulo potrebbe essere legato alla correzione di questa anomalia. Per farlo si può agire sull'algoritmo di parsing per far sì che esso, nel caso in cui venga richiesta una playlist con sliding window, inizi l'elaborazione dalla coda della playlist sulla quale si basa. In alternativa l'algoritmo potrebbe effettuare una stima del numero di righe presenti nella playlist (la semplicità del protocollo HLS consente di farlo) per fare un seek nel file e spostarsi in corrispondenza di una riga sufficientemente prossima alla prima che dovrà rientrare nella sliding window.

Questo tipo di correzioni dovrebbe consentire di superare la limitazione ipotizzata riguardo la durata complessiva della registrazione sul server, arrivando a superare anche di molto la soglia di un'ora stimata.

Conclusioni

Il web-server open-source Nginx gode di una serie di caratteristiche che lo identificano come un ottimo punto di partenza per realizzare sistemi di streaming leggeri e scalabili, basate ovviamente su protocolli HTTP-based . Proprio per questa ragione, sono disponibili diverse soluzioni nate con l'obiettivo di sfruttare le caratteristiche che hanno fatto il successo di Nginx per realizzare un media server semplice da configurare, ad alte prestazioni e in grado di sostenere carichi molto elevati.

Questo lavoro di tesi ha cercato di affrontare e risolvere due lacune che caratterizzano questa tipologia di soluzioni.

1. L'assenza di tutta una serie di funzionalità a supporto di scenari applicativi più complessi della semplice riproduzione di un contenuto multimediale LIVE, EVENT o VOD.

2. La carenza di dati concreti che dimostrino la validità di queste soluzioni e che le mettano a confronto con sistemi commerciali noti e utilizzati nella pratica.

Il primo punto è stato esaurito con l'implementazione di un modulo Nginx con funzionalità limitate ma che sembra funzionare bene nel fare quello per cui è stato pensato. La struttura modulare e l'estendibilità di Nginx si sono rivelate essere un ulteriore punto di forza di questo web-server. Come si è dimostrato nei Capitoli 3 e 4, la semplicità con la quale è possibile realizzare nuovi moduli può rappresentare proprio il modo di colmare le lacune (intese in termini di funzionalità offerte) che rendono le soluzioni di streaming basate su Nginx ancora troppo rudimentali per essere utilizzare in contesti più articolati.

Il secondo punto è di particolare importanza se si vuole offrire un criterio in base al quale scegliere se valga la pena di affidarsi a questo tipo di soluzioni piuttosto che investire in media-server più complessi ed eventualmente a pagamento. È per questa ragione che la parte più rilevante dell'intero lavoro

presentato è molto probabilmente costituita dai test di carico (Capitoli 5 e 6).

I risultati ottenuti in questo lavoro di tesi confermano che una soluzione di streaming HLS basata su Nginx può offrire grandi vantaggi in termini di scalabilità e prestazioni. Se si è disposti a rinunciare ad alcune delle funzionalità messe a disposizione da media-server dedicati, la soluzione proposta può essere un ottimo modo per mettere in piedi un'architettura di streaming a basso costo e facilmente configurabile. L'HLS-NDVR module che è stato sviluppato aggiunge a Nginx alcune di queste funzionalità mancanti. Sebbene molte altre non siano ancora disponibili, i test effettuati hanno evidenziato che c'è ancora un ampio margine per estendere le caratteristiche di questo modulo o per aggiungerne altri complementari.

Bibliografia

[1] *HTTP Live Streaming* [RFC 8216]. (Agosto 2017). Da https://tools.ietf.org/html/rfc8216.

[2] C. Nedelcu *Nginx HTTP Sever*. Packt Publishing; 3 edizione. (18 Novembre 2015).

[3] E. Miller. *Emiller's Guide To Nginx Module Development*. (11 Agosto 2017). https://www.evanmiller.org/nginx-modules-guide.html#prerequisites.

[4] U. Dar.*Nginx Module Extension*. Packt Publishing. (26 Dicembre 2013)

[5] R. Arutyunyan. *NGINX-based Media Streaming Server*. (29 Nov 2017). https://github.com/arut/nginx-rtmp-module.

[6] *Nginx Development Guide* (Gennaio 2018). http://nginx.org/en/docs/dev/development_guide.html.

[7] *HTTP Live Streaming Overview*. (1 marzo 2016). Da https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html

[8] *HTTP Streaming Architecture*. (1 Marzo 2016). Da https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/HTTPStreamingArchitecture/HTTPStreamingArchitecture.html

[9] *Adaptive Bitrate Streaming over HTTP*. Da https://www.encoding.com/packaging/

[10] *HTTP Live Streaming (HLS)*. Da https://caniuse.com/#search=hls.

[11] E. Boyd, *Adaptive Streaming with HLS in HTML5*. (27 Aprile 2016). Da https://www.jwplayer.com/blog/hls-in-html5/.

[12] *Technical Note TN2288 - Example Playlist Files for use with HTTP Live Streaming*. (9 Maggio 2012) https://developer.apple.com/library/content/technotes/tn2288/_index.html.

[13] *Microsoft Announces ActiveMovie*. (5 Marzo 1996) https://news.microsoft.com/1996/03/05/microsoft-announces-activemovie/.

[14] J. Heid *Master Streaming with QuickTime 4*. (1 Ottobre1999) https://www.macworld.com/article/1015339/createmotionheid.html.

[15] *YouTube now defaults to HTML5 <video>*. (27 Gennaio 2015) https://youtube-eng.googleblog.com/2015/01/youtube-now-defaults-to-html5_27.html.

[16] *Video with Adobe Flash CS4 Professional: Delivery and Deployment Primer*. (9 Settembre 2009) http://www.adobepress.com/articles/article.asp?p=1381886&seqNum=2.

[17] *Apache vs Nginx* (3 Marzo 2017) https://www.upguard.com/articles/apache-vs-nginx

[18] *Understanding the Nginx Configuration File Structure and Configuration Contexts* (19 Novembre 2014) https://www.digitalocean.com/community/tutorials/understanding-the-nginx-configuration-file-structure-and-configuration-contexts

[19] *A Guide to Caching with NGINX and NGINX Plus* (23 Luglio 2015) https://www.nginx.com/blog/nginx-caching-guide/

[20] *Apache JMeter™* https://jmeter.apache.org/

[21] *Getting Started: Scripting with JMeter* (2 maggio 2017) https://www.blazemeter.com/blog/getting-started-scripting-jmeter

[22] *User's Manual* http://jmeter.apache.org/usermanual/index.html

[23] *Apache JMeter Distributed Testing Step-by-step*. https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html

[24] *Machine Types*. (2 Marzo 2018). https://cloud.google.com/compute/docs/machine-types