Una infrastruttura per la raccolta di dati di telemetria in ambienti distribuiti: OpenTelemetry

Michele Laddaga
michele.laddaga@studio.unibo.it

May 2022

Questo progetto ha lo scopo di analizzare e testare la struttura e le principali feature fornite dall'infrastruttura OpenTelemetry (https://opentelemetry.io). OpenTelemetry è uno standard aperto per i dati di telemetria, definisce inoltre la tecnologia per raccogliere ed esportare i dati dalle applicazioni **distribuite** per poterli monitorare e analizzare.

Indice

1 Obiettivo

Il progetto si focalizza sull'analisi delle caratteristiche dello standard OpenTelemetry (OTel) verificando inizialmente le soluzioni proposte atte alla risoluzione delle problematiche di gestione di dati di telemetria in ambienti distributi andandole in seguito a testare in un ambiente di prova .NET. Dopo una iniziale introduzione alla telemetria si andranno ad analizzare le caratteristiche e gli strumenti offerti da OpenTelemetry con una breve parentesi su alcuni sistemi back-end di visualizzazione dei dati raccolti. Si terminerà con un esempio pratico in .NET di raccolta di dati telemetrici attraverso OpenTelemetry.

2 Introduzione alla telemetria

La telemetria è un processo automatizzato di comunicazione attraverso il quale vengono effettuate misurazioni e vengono collezionati dati remoti [6].

In ambito software si tratta di recuperare dati relativi all'utilizzo del prodotto e alle performance di applicazioni e relativi componenti. Di seguito alcuni esempi:

- misurazioni dei tempi di avvio e di esecuzione degli applicativi
- frequenza e tempistiche di utilizzo di determinate feature
- tracciamento di anomalie/crash applicativi
- statistiche di utilizzo

Questa tipologia di raccolta dati può rivelarsi essenziale nello sviluppo software e nell'identificazione e risoluzione dei problemi permettendo di accedere a analizzare dati spesso distribuiti e provenienti da una grande varietà di endpoint. Si rivela poi strategica per identificare eventuali funzionalità apprezzate dal cliente su cui focalizzarsi o al contrario identificare quelle scarsamente utilizzate e quindi da rimuovere o porre in secondo piano.

3 Telemetria nei sistemi distribuiti

3.1 Cos'è un log?

Il termine viene utilizzato per indicare [1]:

- la registrazione sequenziale e cronologica delle operazioni effettuate da un utente, un amministratore o automatizzate, man mano che vengono eseguite dal sistema o applicazione;
- il file o insieme di file su cui tali registrazioni sono memorizzate ed eventualmente accedute in fase di analisi dei dati, detto anche registro eventi.

Il logging tradizionale generalmente mette a disposizione l'intero stack trace di esecuzione che può aiutare a risolvere eventuali errori applicativi.

Il problema di questi log è il grande numero di informazioni che si vengono a creare con una conseguente inefficienza di eventuali sistemi di ricerca di dati.

Archiviare e analizzare dati di log sono operazioni dispendiose quindi diventa essenziale gestire solamente ciò che può aiutare ad identificare le problemariche.

I **livelli di log** (Error, Warning, Info, Debug e Trace) permettono di categorizzare i messaggi aiutandone la comprensione e il monitoraggio. Potendo variare il livello di logging a runtime senza modificare il codice sorgente si possono ottenere dettagli diversi in base alle necessità del momento.

Un altro aspetto molto importante è la **standardizzazione del messaggio** per poterlo analizzare più semplicemente, Questo si ottiene memorizzando i log in un formato strutturato, ad esempio il JSON, che può essere analizzato (parserizzato) più semplicemente del semplice formato testuale.

3.2 Logging centralizzato e distribuito

In questo contesto, per logging centralizzato si intende l'*aggregazione* dei dati provenienti dai singoli applicativi *in una posizione centralizzata* per un accesso ed una analisi più semplice.

Nei **sistemi monolitici** le operazioni si strutturano all'interno della stessa macchina: in sistemi distribuiti si potrebbe utilizzare un simile approccio identificando ogni parte come un singolo monolite. Il problema di questo approccio è appunto che vengono raccolti solamente dati dei singoli applicativi permettendo quindi di risolvere solo problematiche del singolo processo.

Il **logging centralizzato** *colleziona ed aggrega i log* dai vari sistemi *in un solo punto centralizzato* dove vengono indicizzati in un database. In questo modo i dati di log possono essere ricercati, filtrati e raggruppati nel relativo software di gestione dei log per stato, host, livello, origine e timestamp ad esempio.

Questa gestione fornisce molti vantaggi permettendo di avere tutte le informazioni rilevanti in un solo punto e riducendo quindi le risorse e il tempo necessario alla loro analisi. Inoltre se un server o un container terminano in maniera imprevista anche i loro log non saranno più a disposizione: centralizzandoli questi restano su un repository prevenendone la perdita.

Poiché questi dati non vengono memorizzati come semplice testo ma attraverso una specifica formattazione, come accennato in precedenza, si ottiene la possibilità di effettuare ricerche più sofisticate e definite fornendo una prospettiva più chiara della situazione del sistema come unica entità.

Il **logging distribuito** al contrario lascia i file di log decentralizzati e questo può portare vari vantaggi in base alle dimensioni del sistema:

- spedire i log in un unico posto può consumare parecchia banda: in base alla rete, al numero e/o frequenza dei log generati si può creare una situazione problematica per il servizio stesso.

- alcuni sistemi di archiviazione dei log inoltre sono più affidabili quando sono più vicini al device che genera il log stesso.

- questa soluzione diventa preferibile per sistemi su larga scala rendendo la centralizzazione dei log più problematica e meno efficiente in termini di costi.

3.3 Cos'è un Trace?

Come è stato appena visto i log forniscono un contesto di esecuzione di una singola applicazione, di seguito si vedrà come il **trace** in un sistema distribuito invece sia uno strumento atto a tracciare le richieste mentre si spostano attraverso varie applicazioni interdipendenti.

Entrambi aiutano nelle operazioni di debugging e analisi dei problemi e permettono non solo di monitorare l'intero sistema in tempo reale ma anche di poter analizzare lo storico per identificare eventuali problemi e disservizi.

I **Log** catturano lo stato dell'applicazione e sono il più elementare strumento di monitoraggio.

Il **tracing** fornisce benefici quando si hanno richieste che attraversano più sistemi fornendo informazioni sui tempi delle richieste, entità coinvolte e latenza introdotta ad ogni interazione.

Ad ogni modo il solo tracing non può identificare la radice del problema alla base di un errore di servizio o una latenza, per quello è necessario analizzare i log. Il tracing aiuta a identificare dove l'anomalia si trova, il log fornisce dettagli aggiuntivi sull'eventuale problema.

Comparato al logging il trace aggiunge più complessità nello sviluppo e risulta anche più dispendioso in termini di risorse.

In base alla tipologia di sistema va valutato se e come abilitare il tracing: in una struttura a microservizi ad esempio diventa essenziale assieme all'utilizzo di log per il debug di errori e diagnostica di problematiche produttive.

Il Trace distribuito è un componente chiave di analisi in sistemi interconnessi e, come accennato in precedenza, si focalizza sul monitoraggio delle performance e sull'identificazione delle problematiche.

Tecnicamente un trace di un sistema distribuito è formato da un'insieme di **Span**: questi sono l'unita di trace più piccola. Possono essere una richiesta HTTP, una chiamata ad un database, l'esecuzione di un messaggio preso da una coda o altre operazioni di interconnessione.

Un Trace permette di vedere come una richiesta viene processata attraverso vari servizi che interagiscono fra loro.

Quando parte una request, ad esempio una HTTP request, viene creato ed assegnato un Trace ID univoco alla richiesta stessa. Man mano che la richiesta si muove nel sistema, ad ogni operazione effettuata su di essa, le viene assegnato un secondo identificativo, chiamato Span, il quale sarà composto dall'ID della request iniziale unito ad un suo ID univoco e ad un identificativo dell'operazione che ha generato la richiesta (chiamato "Parent Span"). Ogni Span è un singolo passo nel percorso che effettua la Request ed è codificato con importanti informazioni relative al processo del microservizio che sta eseguendo l'operazione, come ad esempio:

- il nome del servizio e l'indirizzo del processo che sta gestendo la richiesta;

- misurazioni, i Log e gli eventi che forniscono un contesto sulle attività del processo;
- Tag per cercare e filtrare le richieste ID di sessione, database host, metodo HTTP e altri identificativi;
- Stack trace dettagliati e messaggi di errori nel caso di eventi relativi a problematiche riscontrate.

Attraverso degli strumenti back-end di analisi di tracing distribuito, quali Zipkin o Jaeger ad esempio (quest'ultimo verrà approfondito nei prossimi capitoli), è possibile correlare poi i dati degli Span e formattarli in varie visualizzazioni disponibili attraverso interfacce web.

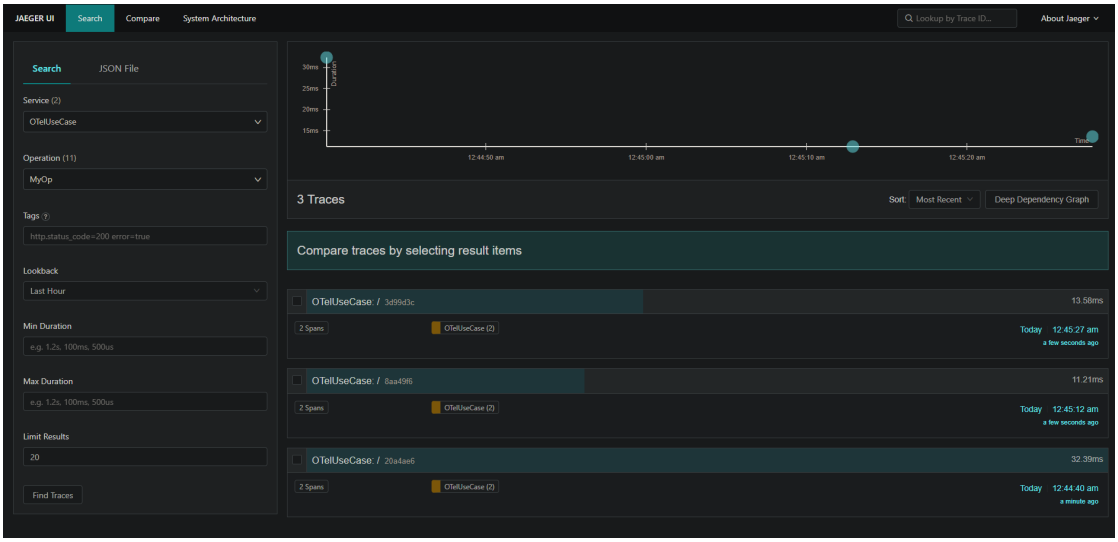

Figura 1: Esempio di query per il tracing con Jaeger

3.4 Best practice per il log/tracing di sistemi distribuiti

Come visto fin'ora il logging e il tracing di sistemi distribuiti sono guidati da una serie di **consuetudini** atte a gestire la natura modulare e potenzialmente eterogenea di questi sistemi. L'obbiettivo è quello di portare coerenza al sistema per una gestione più efficiente e accurata del debugging e più genericamente della risoluzione dei problemi.

Riassumendo quanto detto nelle sezioni precedenti queste consuetudini si traducono nei seguenti punti:

- **Correlare le richieste**: ogni servizio in un sistema distribuito interagisce con gli altri per gestire una richiesta. Marcando la richiesta iniziale con un ID univoco diventa possibile tracciarla facilmente attraverso tutto il sistema, identificando potenziali errori e rivelando dove questi nascano permettendo di comprendere se dipendono ad esempio dalla richiesta in analisi o dalla precedente/successiva. Lo

sviluppatore potrà cercare tale ID univoco nel software di ricerca dei log utilizzato per recuperare i log da tutti i vari servizi per l'analisi.

- **Informazioni da includere**: maggiore è il numero di log e maggiore sarà il contesto a disposizione per comprendere il problema. Il nome del servizio che ha generato il messaggio di log, l'ID di correlazione, l'indirizzo IP della parte server e quello della parte client che ha effettuato la richiesta, data e orario dell'invio del messaggio sono solo alcune delle informazioni da considerare di includere.

- **Come strutturare i dati di log**: uno dei vantaggi di poter operare in maniera distribuita è la possibilità di utilizzare diverse tecnologie per i vari servizi: tuttavia il risultante numero di *formati di log* spesso crea l'analisi di queste informazioni molto più complessa. Strutturare i dati in un *formato standard*, come può essere il JSON, e definirne una struttura ben precisa porterà a semplificarne l'analisi permettendone una ricerca semplificata.

- **Log centralizzati o distribuiti**: spesso è preferibile optare per un logging centralizzato per prevenire la perdita di dati in caso di problemi dei vari server e permettendone un'analisi più veloce. Tuttavia va valutata la situazione caso per caso come definito in precedenza in quanto tale opzione risulta scarsamente scalabile.

4 Perchè OpenTelemetry?

OpenTracing (progetto della Clound Native Computing Foundation - CNCF) e OpenCensus (Google Open Source) sono stati i due standard di riferimento nell'ambito di progetti open source per il tracing distribuito fino a maggio del 2019 quando si sono uniti in un progetto comune prendendo il nome di OpenTelemetry.

OpenTelemetry's Mission: to enable effective observability by making high-quality, portable telemetry ubiquitous. [5]

OpenTelemetry si pone come obiettivo di fornire agli sviluppatori uno strumento per la telemetria pronto all'uso e di alta qualità evitando di dover re-ingegnerizzare tutto internamente con evidenti costi in ambito di tempo e risorse.

L'idea è stata quella di fornire un set standardizzato e indipendente dai fornitori di software, di SDK, API e strumenti di recupero, trasformazione e invio di dati a software back-end di analisi (a loro volta open-source o proprietari).

A tale scopo sono stati fissati 5 punti chiave:

1. La telemetria dovrebbe essere **semplice**, soprattutto per l'utilizzatore finale. Questo significa che OpenTelemetry deve avere un time-to-value veloce, impostare dei default ragionevoli e allo stesso tempo permettere personalizzazioni, il tutto affiancato da un'ottima documentazione fornendo agli sviluppatori un'esperienza di livello superiore.

2. La telemetria dovrebbe essere **universale**, i protocolli e le convenzioni utilizzate dovrebbero essere unificate tra i vari linguaggi e tipi di segnali (tracing, logging,

metrica, ...). Questo significa che OpenTelemetry mira a fornire soluzioni tecniche che funzionano in maniera consistente sia localmente che globalmente.

3. La telemetria dovrebbe essere **indipendente dal produttore**, per decenni soluzioni di monitoraggio proprietarie sono state le scelte più optate per la telemetria. Sfortunatamente la mancanza di uno standard o di API comuni hanno portato gli sviluppatori a rimanere bloccati alle soluzioni proprietarie scelte inizialmente inibendo l'innovazione e blindando la raccolta di dati telemetrici con il relativo storage e l'analisi. OpenTelemetry ha quindi la missione di fornire un unico campo di lavoro per tutti i provider per l'osservabilità dei sistemi, evitare blocchi proprietari di qualsiasi fornitore e interoperare con qualsiasi progetto open source nell'ecosistema della telemetria.

4. La telemetria deve fornire strumenti con **basse inter-dipendenze**, l'utilizzatore finale deve poter scegliere lo strumento che vuole senza dover essere costretto ad aderire all'intero progetto. Per ottenere questo obiettivo l'*architettura software di OpenTelemetry è disaccoppiata dove possibile*. Come corollario questo significa anche che OpenTelemetry non vuole definire un "vincitore" quando si parla di determinati progetti o tecnologie: dove possibile preferisce fornire all'utente finale la *possibilità di scelta*.

5. La telemetria dovrebbe essere **built-in**, storicamente la telemetria è stata qualcosa che gli sviluppattori integravano manualmente o attraverso agenti in post compilazione. OpenTelemetry crede che una telemetria di alta qualità possa essere *integrata* all'interno dell'intero stack software, così come avviene per i commenti nel codice di programmazione ad esempio.

Mentre la struttura e i dettagli tecnici di OpenTelemetry potranno cambiare nel tempo questi 5 punti rimarranno invariati fino al raggiungimento della Mission guardando a loro per orientarsi nel futuro.

4.1 I principi ingegneristici alla base di OpenTelemetry

Alla base degli sviluppi di questo progetto si possono identificare quattro valori fondamentali dell'ingegneria di sistemi distribuiti:

1. **Compatibilità** : visto il numero di componenti in interesse e piattaforme supportate seguire specifiche e garantire interoperabilità diventa di vitale importanza per l'utilizzatore. OpenTelemetry si sforza di essere conforme agli standard, indipendente dai produttori (vendor-neutral), coerenti e costanti rispetto ai vari linguaggi e componenti.

2. **Stabilità** : poiché molte librerie hanno dipendenze sulle API di OpenTelemetry, la stabilità e retro-compatibilità di queste ultime diventa essenziale per gli utenti finali. Di conseguenza non vengono introdotti nuovi concept a meno di non avere la certezza che siano necessari ad una buona parte degli utilizzatori di OpenTelemetry.

3. **Resilienza al cambiamento** : OpenTelemetry è progettato per lavorare e continuare a raccogliere dati telemetrici anche quando l'applicazione non si comporta come previsto.

4. **Performace** : l'utente non deve scegliere tra telemetria di qualità e un'applicazione performante. Le alte prestazioni sono un requisito fondamentale e non è accettabile il presentarsi di interferenze inaspettate con l'applicazione host.

5 OpenTelemetry (OTel) API e SDK e Dati

Le specifiche OTel descrivono i requisiti e le aspettative cross-language per tutte le implementazioni attraverso API, SDK e Dati (formati e standard).

5.1 API

Definiscono i tipi di dato e le operazioni per la generazione e correlazioni dei dati di:

- Tracing: permettono di creare il Tracer (sorgente di Span) e generare **Span** i quali, come visto in precedenza, rappresentano una singola operazione all'interno di un trace. Tali **Span** possono essere innestati per generare un albero. Questi Span contengono proprietà poi utilizzate per la ricerca (Attributes). Per ogni Span vengono registrati eventi di log (Events). Ogni trace contiene uno Span radice, il quale tipicamente descrive l'intera operazione, e opzionalmente uno o più Span figli per le sue operazioni figlie. Questi Span sono poi correlati attraverso un contesto che viene propagato (SpanContext).

- Logging: come vedremo nell'esempio, spesso OpenTelemetry non introduce nuove API per il loggin ma va ad integrare le già presenti: ad esempio in ambiente Microsoft .Net si appoggia alle preesistenti Microsoft.Extensions.Logging API.

- Metrics: permettono all'utente di catturare misurazione sull'esecuzione di un programma a runtime. Queste API sono progettate per processare misure grezze, generalmente con l'intento di produrre delle sommarizzazioni continue di queste misurazioni.

- Baggage: queste API permettono di aggiungere contesto a metric, trace e log. Il baggage può essere propagato utilizzando Propagators.

5.2 SDK

Definiscono i requisiti per le implementazioni specifiche per i vari linguaggi delle API. Sono inoltre qui definiti i concetti di configurazione, data processing e esportazione: essi permettono quindi, attraverso specifiche librerie client per ogni linguaggio, di instrumentare manualmente un applicativo per generare metriche e tracing di telemetria. L'SDK permette di campionare, propagare il contesto ed effettuare altre varie procedure ed infine di esportare i dati, tipicamente ad un Collector OpenTelemetry (o altri collector

come quelli per Zipkin o per Jaeger tramite suite di SDK di esportazione specifiche). OTEL supporta inoltre l'instrumentazione automatica con integrazione a vari framework popolari e librerie così come agenti di auto instrumentazione: questo permette di ridurre il codice necessario per iniziare a collezionare dati telemetrici.

5.3 Dati

Definiscono il protocollo OpenTelemetry (OTPL) e le convenzioni semantiche indipendente dai fornitori di software e dai linguaggi per cui un back-end di telemetria può fornire supporto.

6 Raccolta dati, trasformazione ed esportazione

I dati campionati tramite SDK OpenTelemetry:

- possono essere inviati direttamente ai back-end supportati come Jaeger, Zipkin o Prometheus
- possono essere raccolti attraverso un Collector OTEL Agent che successivamente si occuperà di esportarli verso tali back-end o altri Collector di tipo Gateway.

Questi Collector quindi possono fungere sia da agenti locali posizionati sullo stesso host dell'applicazione da monitorare (più dispendioso ma permette di recuperare informazioni sulla macchina Host non recuperabili altrimenti) sia da servizio centrale di aggregazione per vari nodi applicativi avvalendosi di Collector Gateway, soluzione ottima in termini di scalabilità. Vedremmo successivamente un esempio pratico che replica anche questa seconda struttura.

Ogni Collector è composto da 3 parti principali:

1. **Ricevitori** per assimilare dati in ingresso in vari formati e protocolli come OTLP (OpenTelemetry Protocol), o altri formati come quello di Jaeger, Zipkin o altro (lista completa).
2. **Processatori** per effettuare aggregazioni, filtri e campionamenti.
3. **Esportatori** per trasmettere i dati telemetrici verso uno o più back-end (solitamente strumenti di analisi o aggregatori di più alto livello) in vari formati e protocolli come OTLP, Prometheus e Jaeger (lista completa).

7 Sistemi back-end di analisi dati OpenTelemetry

Come visto in precedenza OpenTelemetry si pone come obiettivi di standardizzare due cose:

- il modo di instrumentare il codice applicativo per generare dati telemetrici
- il formato dei dati telemetrici generati

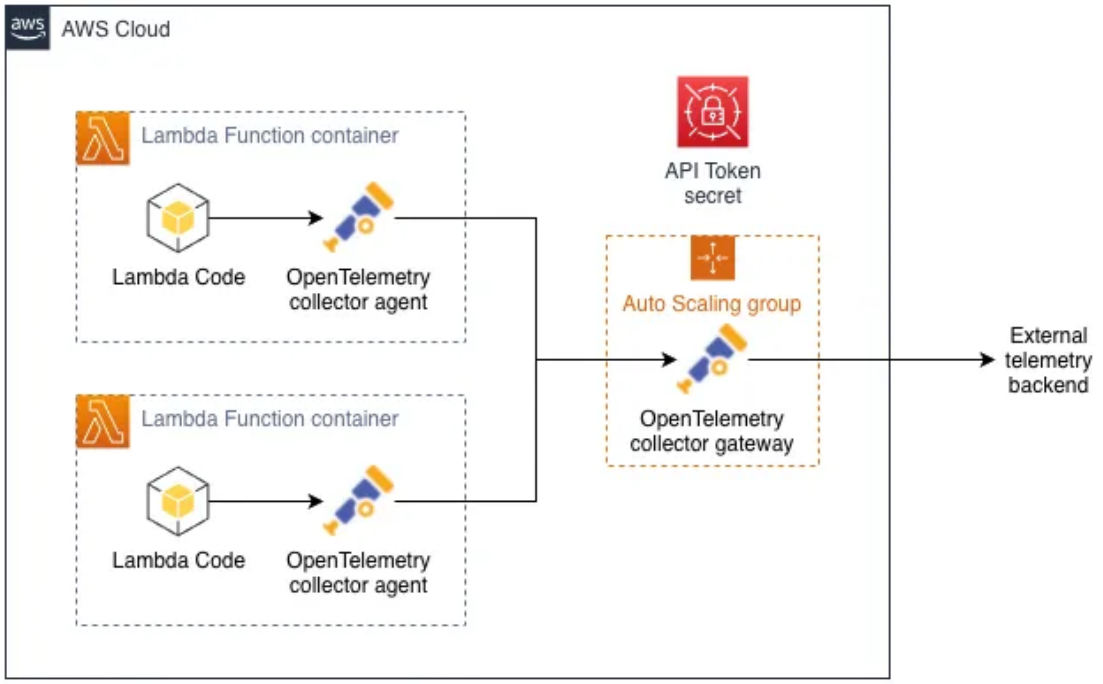

Figura 2: Esempio architettura con Collector Gateway [3]

Una volta creati i dati in formati standard essi possono essere inviati a qualsiasi back-end di analisi che supporti tali formati. Questo permette agli utilizzatori di poter scegliere il back-end di analisi più adatto alle proprie esigenze come ad esempio Zipkin o Jaeger. Sia Zipkin che Jaeger sono strumenti open-source per il tracing distribuito. Zipkin era originariamente inspirato a Dapper di Google ed è stato sviluppato da Twitter diventando in seguito un progetto open-source nel 2012. Jaeger invece è stato creato dai team di sviluppo in Uber diventando poi open-source nel 2015.

Questi strumenti hanno un'architettura simile composta dai seguenti componenti:

- Instrumentation Libraries: generazione dei dati di telemetria partendo dal codice applicativo.
- Collectors: ricevono i dati telemetrici creati tramite instrumentazione, li validano, li indicizzano, effettuano eventuali trasformazioni e infine li archiviano.
- Query Service e web UI: i dati vengono consultati tramite API e mostrati attraverso un'interfaccia web di consultazione.
- Database Storage: Cassandra ed Elasticsearch sono al momento i back-end di archiviazione/indicizzazione più utilizzati.

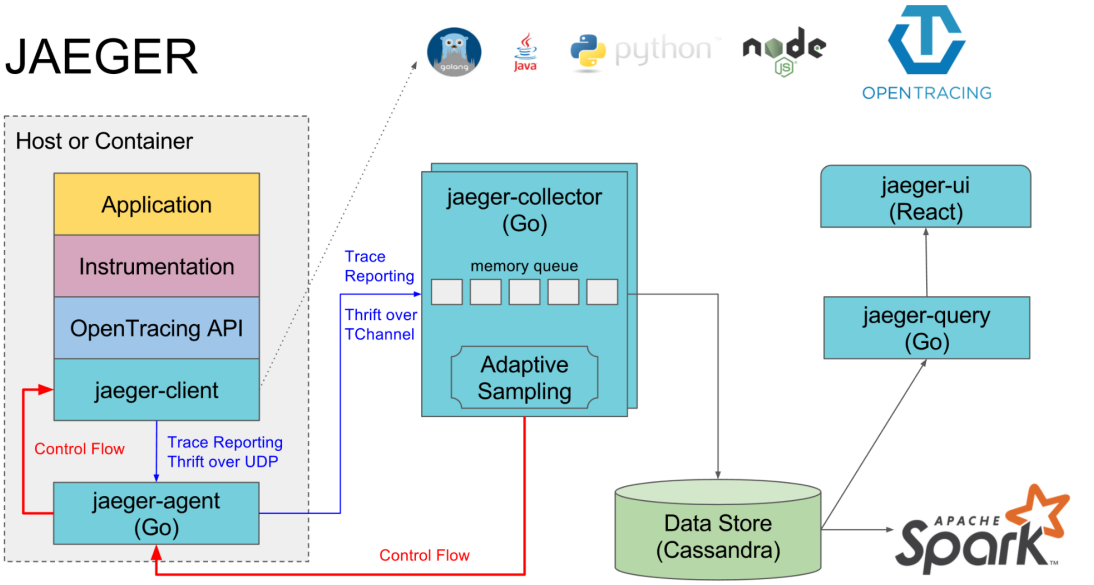

Figura 3: Architettura di Jaeger (source by:`https://www.jaegertracing.io`)

Quelli appena analizzati sono strumenti atti all'analisi di dati di trace, per quanto riguarda l'analisi dei dati metrici sono presenti altri sistemi (vedi Prometheus) che non analizzeremo in questa sede.

8 Un esempio pratico

Come esempio è stata creata una webApp ASP.NET Core che mostra come inizializzare OpenTelemetry per logging e tracing, analizzando anche i rispettivi exporter. I dati dai logging vengono esportati direttamente sulla console, consultare il relativo capitolo per l'utilizzo di altri tipi di exporter. I dati di tracing invece vengono esportati utilizzando lo Jaeger exporter per essere consultati attraverso Jager. Per avviare Jaeger e l'applicativo di test è stato creato un file di configurazione docker-compose.yml, è sufficiente posizionarsi nella radice della soluzione OTelUseCase ed eseguire la seguente riga di comando assicurandosi di avere Docker in esecuzione in modalità Linux:

```
docker-compose up -d
```

Jaeger sarà cosi consultabile in localhost alla porta 16686 , l'applicativo alla porta 5000 (entrambi i parametri sono configurabili nel file docker-compose.yml), infine partirà anche una coppia di Collector il cui utilizzo è spiegato nel'esempio successivo.

8.1 Partiamo con il Logging

Per gestire il logging in applicativi .NET Core in un progetto che si vuole tracciare va installato il pacchetto nuget Microsoft.Extensions.Logging. A questo punto è necessa-

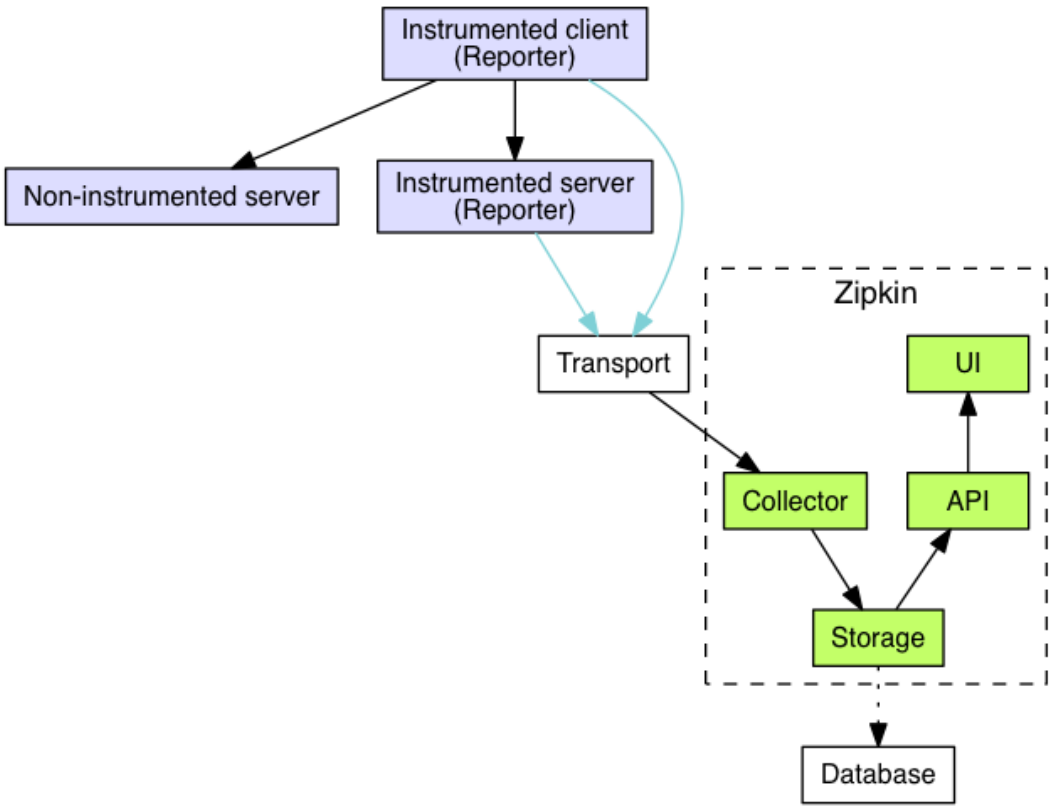

Figura 4: Architettura di Zipkin (source by:`https://zipkin.io`)

rio scegliere ed installare un exporter, nell'esempio proposto viene utilizzato l'exporter verso Console il quale è utilizzato per lo più a titolo esplorativo, sufficiente per comprendere come esportare tali dati: OpenTelemetry.Exporter.Console. L'inizializzazione del LoggerFactory può variare dal tipo di applicativo sviluppato ma avrà in ogni caso OpenTelemetry aggiunto come LoggerProvider [4] :

```
using var loggerFactory = LoggerFactory.Create(builder =>
{
    builder.AddOpenTelemetry(options =>
    {
        options.AddConsoleExporter();
    });
});
```

In applicativi .NET il LoggerFactory è lo strumento utilizzato per configurare un sistema di logging e per creare un istanza di Logger del LoggerProvider scelto (OpenTelemetry in questo caso). Il LoggerProvider è colui che custodisce i log, nel caso del ConsoleProvider questi verranno semplicemente direzionati verso lo standard output. A questo punto avremo il servizio di loggin registrato dal LoggerFactory e potremo utilizzarlo instanziando un logger direttamente o attraverso l'inversion of control come nei due esempi a seguire:

```
var logger = loggerFactory.CreateLogger<Program>();
logger.LogInformation($"Welcome to
   {Assembly.GetEntryAssembly().GetName().Name}.");
```

```
public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
        _logger.LogInformation("Log: IndexModel");
    }
```

A questo punto avremo in output dei trace simili al seguente:

```
LogRecord.Timestamp:                2022-11-27T22:27:43.8777094Z
LogRecord.CategoryName:             Program
LogRecord.LogLevel:                 Information
LogRecord.State (Key:Value):
  OriginalFormat (a.k.a Body): Welcome to OTelUseCase.
Resource associated with LogRecord:
service.name: unknown_service:iisexpress
```

Come visto nei capitoli precedenti tali log in un caso più complesso dovrebbero essere esportati e archiviati (in locale o in remoto come visto) per permetterne la consultazione.

8.2 Un esempio di Tracing

Nel runtime di .NET, prima ancora dello standard OpenTelemetry, era già presente il corrispettivo dello Span di OTel ovvero l'*Activity* il quale viene quindi riutilizzato da

OpenTelemetry .NET allo stesso scopo. L'utilizzo è molto semplice, facilitato come visto nei capitoli precedenti dagli strumenti di auto instrumentazione:

1. Installare i pacchetti Nuget di OpenTelemetry

```
<PackageReference Include="OpenTelemetry.Exporter.Jaeger"
   Version="1.4.0-beta.3" />
<PackageReference
   Include="OpenTelemetry.Exporter.Prometheus"
   Version="1.3.0-rc.2" />
<PackageReference
   Include="OpenTelemetry.Extensions.Hosting"
   Version="1.0.0-rc9.9" />
<PackageReference
   Include="OpenTelemetry.Instrumentation.AspNetCore"
   Version="1.0.0-rc9.9" />
<PackageReference
   Include="OpenTelemetry.Instrumentation.Http"
   Version="1.0.0-rc9.9" />
```

   I pacchetti OpenTelemetry.Instrumentation.* includono il middleware che creerà tutti i trace e le metriche per tutte le librerie Microsoft che vorremo utilizzare I pacchetti OpenTelemetry.Exporter.* contengono tutti gli strumenti per l'esportazione.

2. Procedere con la registrazione del servizio per il tracing, nell'esempio proposto utilizzeremo Jaeger come exporter e ci registreremo ad ogni trace prodotta con il nome definito dalla costante TelemetryConstants.MyAppTraceSource. Per adattare la soluzione all'utilizzo di docker-compose sono stati definiti host e porta del servizio di Jaeger attraverso le variabili di ambiente `Jaeger__Host` e `Jaeger__Port` usate quindi per definire l'azione di configurazione del metodo AddJaegerExporter nel nostro applicativo di esempio:

```
builder.Services.AddOpenTelemetryTracing(b =>
{
    // uses the default Jaeger settings
    //b.AddJaegerExporter();
    // get Jaeger configuration parameters from
       environment variables
    b.AddJaegerExporter(o =>
     {
         o.AgentHost =
            builder.Configuration.GetValue<string>("Jaeger:Host");
         o.AgentPort =
            builder.Configuration.GetValue<int>("Jaeger:Port");
     });

    // adding Otlp exporter
    var oTelExporter__Host =
       Environment.GetEnvironmentVariable("OTelExporter__Host");
    b.AddOtlpExporter(opt =>
    {
         opt.Protocol =
            OpenTelemetry.Exporter.OtlpExportProtocol.Grpc;
         if (!string.IsNullOrEmpty(oTelExporter__Host))
         {
             opt.Endpoint = new Uri(oTelExporter__Host);
         }
         System.Console.WriteLine($"OTLP Exporter is using
            {opt.Protocol} protocol and endpoint
            {opt.Endpoint}");
    });

    // receive traces from our own custom sources
    b.AddSource(TelemetryConstants.MyAppTraceSource);

    // decorate our service name so we can find it when we
       look inside Jaeger
    b.SetResourceBuilder(ResourceBuilder.CreateDefault()
         .AddService("OTelUseCase"));

    // receive traces from built-in sources
    b.AddHttpClientInstrumentation();
    b.AddAspNetCoreInstrumentation();
})
```

3. A questo punto, grazie ad AddOpenTelemetryTracing, nelle varie pagine della no-

stra webApp potremo instanziare, tramite inversion of control, un TracerProvider nel costruttore con il quale potremo creare un Tracer che scriverà su un canale il cui nome è quello su cui prima ci siamo messi in ascolto: TelemetryConstants.MyAppTraceSource.

```
public IndexModel(ILogger<IndexModel> logger,
   TracerProvider provider)
            {
                 _logger = logger;
                 _tracer =
                    provider.GetTracer(TelemetryConstants.MyAppTraceSour
                 _logger.LogInformation("Log:
                    IndexModel");
}
```

In questo modo ad ogni richiesta in Get della pagina, ad esempio, potremo creare uno Span da tracciare

```
public void OnGet()
{
  _logger.LogInformation("Log: Index OnGet");

  var tags = new TagList();
  tags.Add("user-agent", Request.Headers.UserAgent);
  TelemetryConstants.HitsCounter.Add(1, tags);
  using var mySpan =
     _tracer.StartActiveSpan("MyOp").SetAttribute("httpTracer",
     HttpContext.TraceIdentifier);
  mySpan.AddEvent($"Received HTTP request from
     {Request.Headers.UserAgent}");
}
```

Avviando la webApp e Jaeger sarà ora possibile tenere traccia e visualizzare i dati di tracing, ad esempio ricaricando la pagina principale

8.3 Un esempio di interoperabilità tra più collector

È possibile avviare l'esempio appena proposto analizzando anche l'esportazione del tracing utilizzando i Collector Gateway: infatti oltre ad essere impostata l'esportazione diretta verso Jaeger è attiva anche l'esporter Otlp configurato per inviare i dati dall'applicativo ad un Colletor Gateway il quale è stato configurato per inoltrare il tutto ad un secondo Collector e entrambi infine esportano verso il back-end. Come è possibile notare a fronte di una sola coppia di span ottenuti caricando la pagina iniziale del nostro applicativo abbiamo registrato 6 span totali (2 iniziali più due provenienti da ognuno dei collector che nell'esempio esportano anche sul back-end).

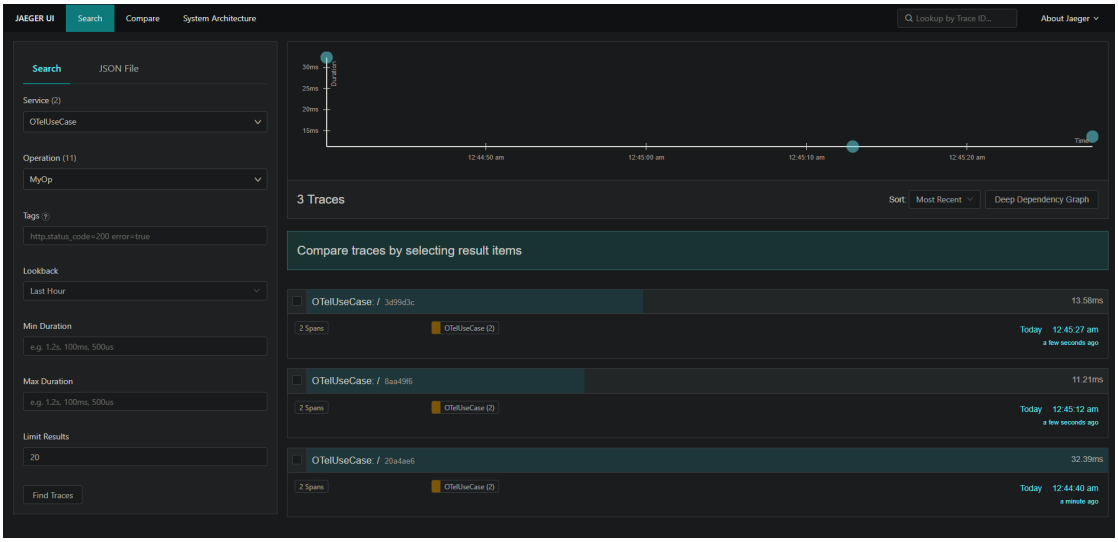

Figura 5: Esempio di query per il tracing con Jaeger

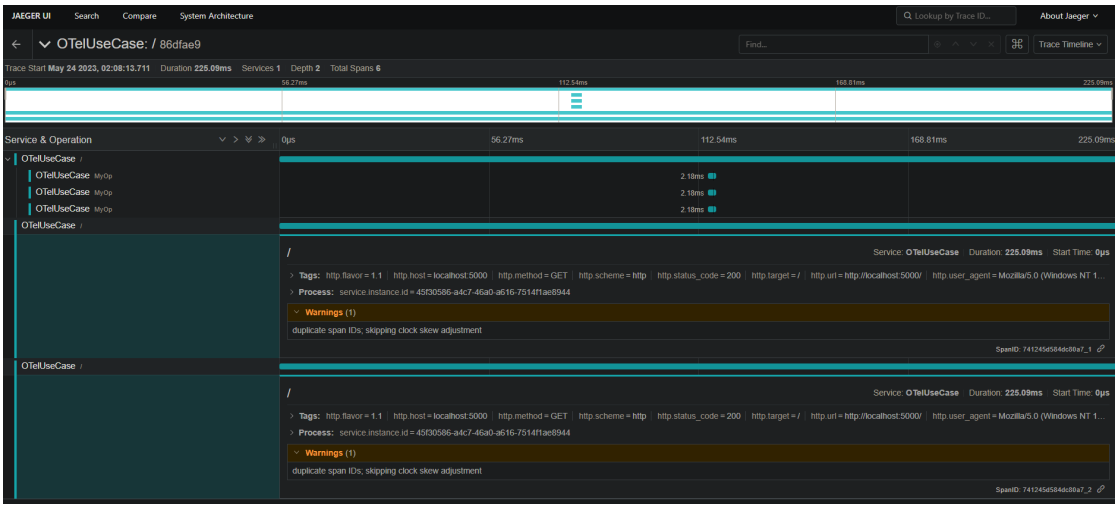

Figura 6: Esempio di span duplicato arrivato dall'applicativo di esempio e poi dal Collector

Come prima cosa è stato necessario aggiungere alla soluzione il pacchetto OpenTelemetry.Exporter.OpenTelemetryProtocol che al momento in cui scriviamo gestisce tracing e metriche (la gestione dei log è in sviluppo e verrà integrata a questo pacchetto una volta raggiunta una versione stabile). A questo punto è stato aggiunto l'exporter Otlp (già visibile dal codice mostrato in precedenza)

```
builder.Services.AddOpenTelemetryTracing(b =>
{
    ...

    // adding Otlp exporter
    var oTelExporter__Host =
       Environment.GetEnvironmentVariable("OTelExporter__Host");
    b.AddOtlpExporter(opt =>
    {
        opt.Protocol =
           OpenTelemetry.Exporter.OtlpExportProtocol.Grpc;
        if (!string.IsNullOrEmpty(oTelExporter__Host))
        {
            opt.Endpoint = new Uri(oTelExporter__Host);
        }
        System.Console.WriteLine($"OTLP Exporter is using
           {opt.Protocol} protocol and endpoint
           {opt.Endpoint}");
    });

   ....
})
```

Come accennato all'inizio di questo capitolo il file di configurazione docker-compose.yml è stato configurato per avviare anche dele istanze di un Collector OpenTelemetry che partirà con i settaggi di default (configurabili in ogni caso tramite lo stesso docker-compose.yml per le porte in ingresso e il relativo file di configurazione nella cartella `.\otelcollector\otelcol-config.yml` e `.\otelcollector\otelcol-config_middle.yml` per il resto delle impostazioni). Una volta avviato il tutto oltre a vedere i dati inoltrati sul back-end vengono messe a disposizione sulle porte 13133 e 13134 degli health check extension per monitorare lo stato dei Collector:

```
{
    "status":"Server available",
    "upSince":"2023-05-23T23:05:12.1377528Z",
    "uptime":"38.3599587s"
}
```

| Entrambi i collector sono configurati per ricevere dati otlp attraverso protocollo gRPC sulla porta standard 4317

```
receivers:
  otlp:
    protocols:
      grpc:
      http:
```

elaborarli, ed infine esportarli su Jaeger (sulla porta 14250) e sullo standard output degli stessi servizi. Il primo collector otelcol inoltre esporta i dati anche sul secondo collector otelcol-middle

```
exporters:
  otlp:
    endpoint: otelcol-middle:4317
    tls:
      insecure: true
  jaeger:
    endpoint: jaeger:14250
    tls:
      insecure: true
  logging:
    loglevel: debug
```

Come impostato nell'esempio è possibile concatenare più collector. In questo modo si potranno gestire, come anticipato nel relativo capitolo, le varie necessità di scalabilità intrinseche dei sistemi distribuiti.

Questo secondo esempio serve quindi a mostrare come sia possibile scalare molto semplicemente una struttura di tracing: invece di esportare i dati direttamente verso il back-end si procede impostando, come visto nel capitolo 6, degli Agent Collector negli host degli applicativi da monitorare (non obbligatorio ma consigliato per recuperare dati dell'host non recuperabili altrimenti) e gestendo quindi poi dei cluster di Collector Gateway che riceveranno tali dati, volendo anche con logiche di load ballancing, per scalare in maniera semplice la struttura in esame.

9 Conclusioni

La telemetria e in particolare il mondo di OpenTelemetry sono molto più vasti di quanto qui mostrato: oltre al supporto e alla presenza di SDK per molti linguaggi (Java, Go, Ruby, Javascript, Swift, Erlang, ecc) e alla presenza di vari strumenti per l'instrumentazione automatica [2] (la quale non richiede modifica del codice e permette, attraverso un agente che si collega all'applicativo in esecuzione, di estrarre i dati di trace), c'è un'intera comunity che sta lavorando per consolidare e ampliare questo standard portandolo ad essere a tutti gli effetti un progetto community-driven. Con l'incremento del numero di applicativi distribuiti che devono inter-comunicare risulta di primaria importanza tale standard per garantire uno strumento univoco per lo sviluppo, l'analisi a runtime, la manutenibilità e la gestione a 360 gradi dei sistemi distribuiti.

Riferimenti bibliografici

[1] *Definizione di log*. URL: https://it.wikipedia.org/wiki/Log.

[2] *Instrumentazione automatica*. URL: https://logz.io/learn/opentelemetry-guide/#auto.

[3] *LoggerCollectorArchitecture*. URL: https://betterprogramming.pub/using-aws-distro-for-opentelemetry-with-jaeger-acf4df3a8e37.

[4] *LoggerProvider*. URL: https://docs.microsoft.com/dotnet/core/extensions/logging-providers.

[5] *Open Telemetry GitHub*. URL: https://github.com/open-telemetry/community/blob/main/mission-vision-values.md#otel-mission-vision-and-values.

[6] *Telemetry communications*. URL: https://www.britannica.com/technology/telemetry.