Vert.x come MOM per Sistemi Distribuiti

Martina Magnani, Mattia Vandi
{martina.magnani8,mattia.vandi}@studio.unibo.it

C.d.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum — Università di Bologna, Cesena

Anno Accademico 2017/2018

Outline

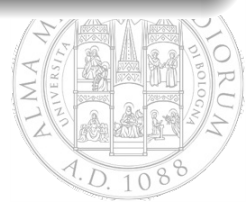
Che cos'è un Middleware? I

Dizionario - Google

"Insieme di software che fungono da intermediari fra strutture e programmi informatici, permettendo loro di comunicare a dispetto della diversità dei protocolli o dei sistemi operativi."

Umar, Object-Oriented Client/Server Internet environments

"Il Middleware è un insieme di servizi di supporto alla distribuzione indipendenti dalle applicazioni. In definitiva il middleware è il software che risiede al di sopra della rete ed al di sotto delle applicazioni software."



Che cos'è un Middleware? II

Una nostra definizione...

"Il Middleware è un programma software che sta nel mezzo tra il livello del Sistema Operativo e quello delle Applicazioni. Uno strato software in grado di fornire le astrazioni ed i servizi utili per lo sviluppo di applicazioni (anche eterogenee) distribuite."

Glue Technologies

"Le piattaforme middleware vengono anche definite "software di connettività tra applicazioni", o meglio "glue technologies" (tecnologie collanti), evidenziando la loro caratteristica di tecnologie di integrazione di applicazioni."

Perché utilizzare un Middleware?

Mascherare le eterogeneità

Lo strato middleware offre ai programmatori di applicazioni distribuite librerie, o middleware API (Application Programming Interface), in grado di mascherare i problemi dovuti all'eterogeneità dei sistemi su rete.

Offrire trasparenza

Una tecnologia è trasparente se nasconde la complessità sottostante. Obiettivo del middleware è dunque far "scomparire" la complessità introdotta dalla distribuzione.

Interoperabilità fra applicazioni

Obiettivo del middleware è garantire l'interoperabilità fra applicazioni che utilizzano lo stesso middleware e fra applicazioni che utilizzano middleware differenti (*Enterprise Application Integration*, **EAI**).

Middleware di integrazione I

Integrazione

- **Prima**: si ha un disastro totale (molti sistemi che utilizzano tecnologie diverse, interazioni ad-hoc, strutture irregolari e ogni elemento deve essere descritto nella propria cornice di riferimento).
- **Quindi**: si utilizza un middleware di integrazione (IM). I sistemi esistenti sono rimodellati utilizzando il nuovo modello supportato dall'IM.
- **Dopo**: il disastro iniziale è stato organizzato in oggetti coerenti CORBA o agenti JADE.

Middleware di integrazione II

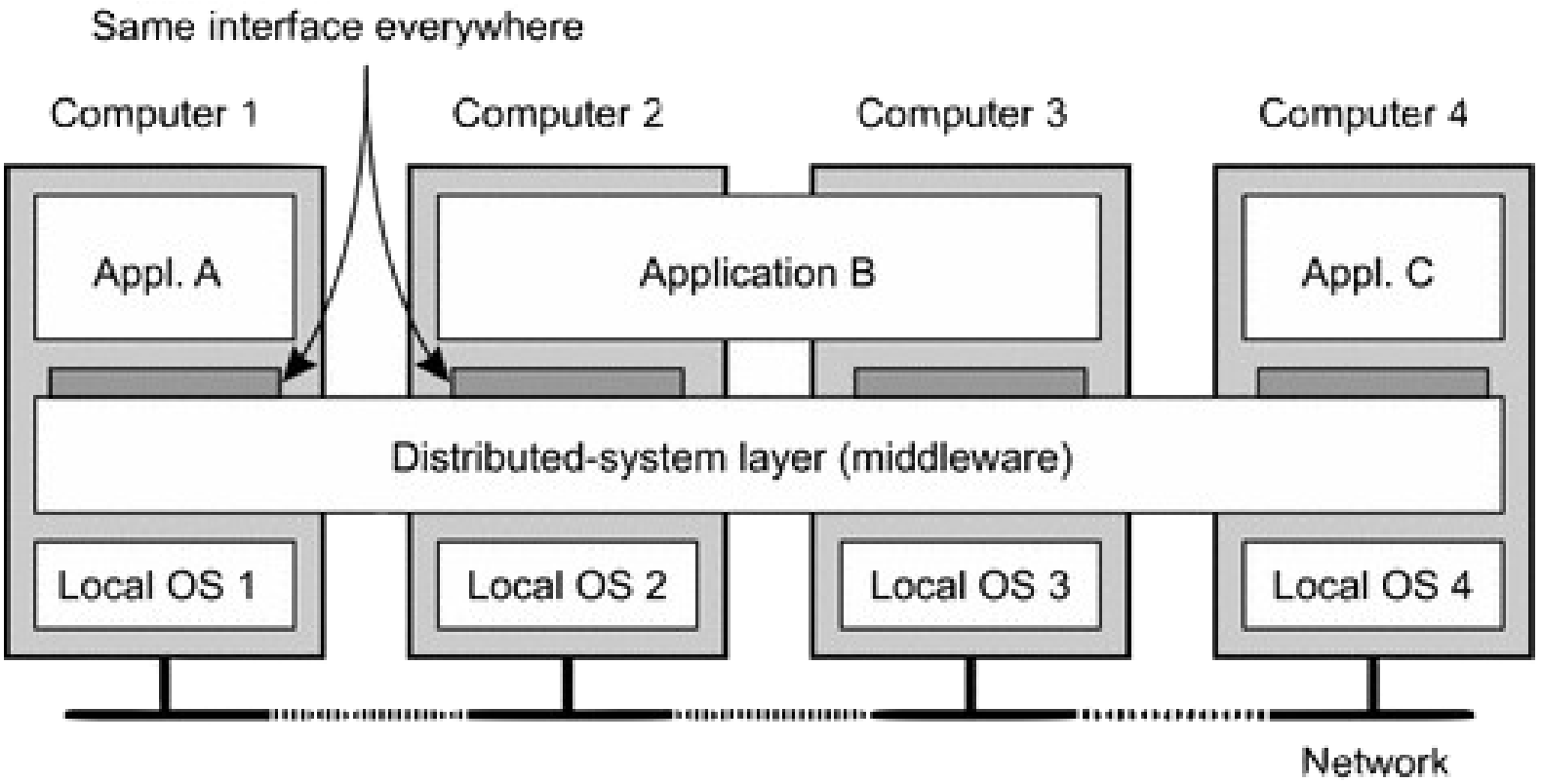
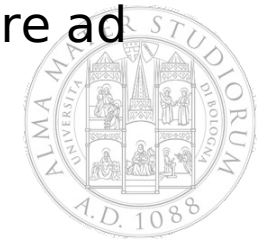


Figura: Il middleware è stato esteso su più elaboratori al fine di offrire ad ogni applicazione la stessa interfaccia di comunicazione.



Middleware Astratto vs. Concreto

Due livelli di Middleware

- **Middleware astratto**: definizione di uno standard (API, comunicazione, protocolli, ecc...).
- **Middleware concreto**: implementazione di un modello standard effettivamente utilizzabile.

Outline

Comunicazione

Definizione

“Descrive il processo e le modalità di trasmissione di un’informazione da un’entità a un’altra (o da un luogo a un altro), attraverso lo scambio di un messaggio elaborato secondo le regole di un determinato codice.”

Comunicazione nei Sistemi Distribuiti

La comunicazione nei sistemi distribuiti è un problema affrontato dai middleware e presenta sfide più difficili: l’inaffidabilità della comunicazione e la larga scala del sistema. L’obiettivo principale è garantire la trasparenza della distribuzione e della posizione.

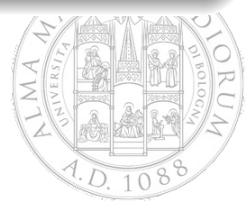
Comunicazione e Middleware

Problema

Il Middleware vive principalmente a livello di Applicazione ma i protocolli per i servizi middleware sono diversi dai protocolli di applicazione di alto livello. Come possiamo distinguere i diversi protocolli nello stesso livello?

Soluzione: Middleware Layer

Rimpiazzare il livello di sessione e il livello di presentazione con un livello middleware che include tutti i protocolli indipendenti dall'applicazione.



Tipi di comunicazione I

Persistente vs. Transitoria

- **Comunicazione persistente**: un messaggio inviato viene memorizzato dal middleware di comunicazione fino a quando non viene consegnato al destinatario. Non è necessario un accoppiamento temporale tra mittente e destinatario.
- **Comunicazione transitoria**: un messaggio inviato viene memorizzato dal middleware di comunicazione fintantoché sia il destinatario che il mittente sono in esecuzione. Deve esserci un accoppiamento temporale tra il mittente e il destinatario.

Tipi di comunicazione II

Asincrona vs. Sincrona

- **Comunicazione asincrona**: il mittente continua l'esecuzione dopo l'invio di un messaggio. Il messaggio deve essere memorizzato dal middleware.
- **Comunicazione sincrona**: dopo l'invio di un messaggio l'esecuzione del mittente si blocca e attende la risposta, fino a quando il middleware non conferma la trasmissione del messaggio, o finché il destinatario non conferma la ricezione o fino a quando il destinatario non ha completato l'elaborazione della richiesta.

Tipi di comunicazione III

Discreta vs. Continua

- **Comunicazione discreta**: il mittente invia un'unità completa di informazione (es: messaggio).
- **Comunicazione continua**: il mittente invia una sequenza di messaggi che rappresentano un flusso di informazione (es: stream audio, immagine).

Modelli di comunicazione

Principali modelli

- **Remote Procedure Call, RPC**: permette di invocare una procedura residente su di una macchina remota. Richiede la definizione di *Inteface Definition Language* (IDL).
- **Message-Oriented Middleware, MOM**: supporta lo scambio di messaggi tra applicazioni distribuite (IBM MQSeries, Sun ToolTalk).
- **Transaction Processing Middleware, TCM**: semplifica la progettazione di applicazioni che richiedono esecuzione di transazioni distribuite - TP monitors (IBM CICS, BEA TUXEDO, Transarc Encina).
- **Object-Oriented Middleware, OOM**: permette di implementare un'applicazione ad oggetti distribuiti e di riferirsi ad essi come se fossero oggetti locali.

Outline

Event-Based Architecture I

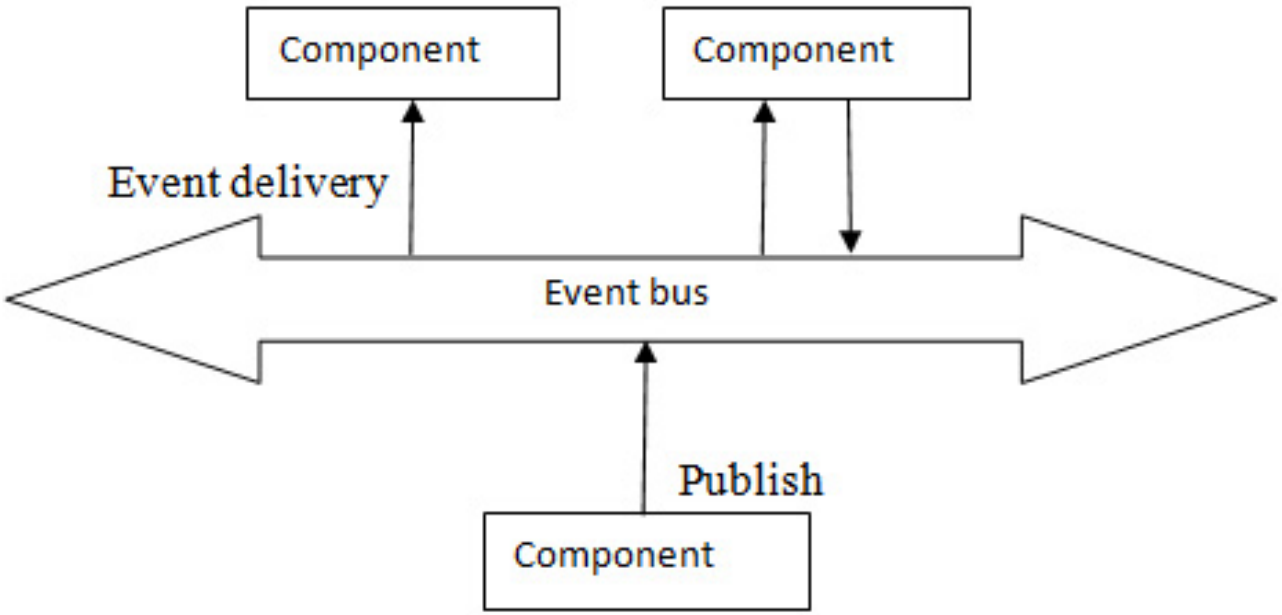
Idea

La comunicazione fra processi avviene attraverso lo scambio di messaggi che vengono propagati tramite un bus di eventi. I messaggi hanno la possibilità di trasportare dati.

Caratteristiche principali

- I processi possono comunicare senza bisogno di fare riferimento l'un l'altro, ovvero sono *referenzialmente disgiunti*.
- I processi possono comunicare senza la necessità di condividere lo stesso spazio, ovvero sono *spazialmente disgiunti*.

Event-Based Architecture II



Publish/subscribe

- Publishers: pubblicano eventi tramite il middleware.
- Subscribers: ricevono gli eventi a cui si sono sottoscritti.

Message-Oriented Middleware

Message-Oriented Middleware è stata una delle prime tecnologie che si è servita del concetto di un bus di comunicazione comune tra le applicazioni per l'interazione.

Funzionamento

Fornisce un servizio di memorizzazione di messaggi.

- Il messaggio spedito da un mittente viene consegnato ad una coda di destinazione.
- I destinatari recuperano i loro messaggi dalla coda di destinazione.
- Assenza di accopiamento temporale tra mittente e destinatario.

Servizi forniti

Un MOM fornisce

- Servizi per l'invio di messaggi alle code.
- Servizi per la ricezione dei messaggi presenti in una coda.
- Servizi per la creazione e la rimozione di code (permanenti o temporanee).
- Servizi aggiuntivi (autenticazione, crittografia, ecc.).

Architettura

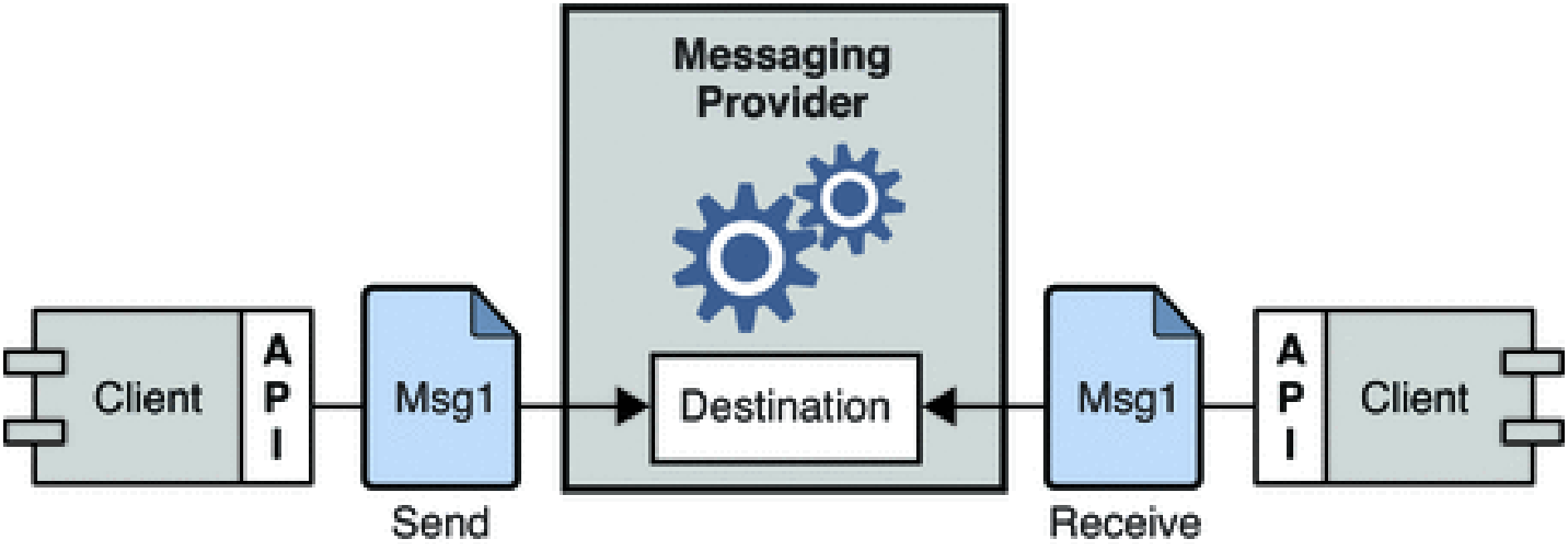


Figura: Architettura generale di un Message Oriented Middleware

Outline

Che cos'è Vert.x? I

Vert.x's Motto

*"Eclipse **Vert.x** is a tool-kit for building **reactive** applications on the **JVM**."*

Che cos'è Vert.x? II

Terminologia

- tool-kit: Vert.x fornisce semplicemente un insieme di strumenti e permette allo sviluppatore di costruire l'applicazione come preferisce.
- **reactive**: Vert.x è *guidato dagli eventi* e *non bloccante*, può gestire alti livelli di concorrenza utlizzando un numero limitato di thread di sistema.
- **JVM**: Vert.x è *poliglotta*, permette di scegliere il linguaggio da utilizzare in base al compito da svolgere e alle competenze del team di sviluppo.
  - Vert.x è utilizzabile in applicazioni multi-linguaggio, i linguaggi supportati sono: Java, JavaScript, Groovy, Ruby, Ceylon, Scala e Kotlin

“Simple but not simplistic”

Carattestiche

- Asincrono di natura
- Modulo core leggero (circa 650 kB)
- Modello di concorrenza molto semplice
- Basato sul pattern Multi-Reactor

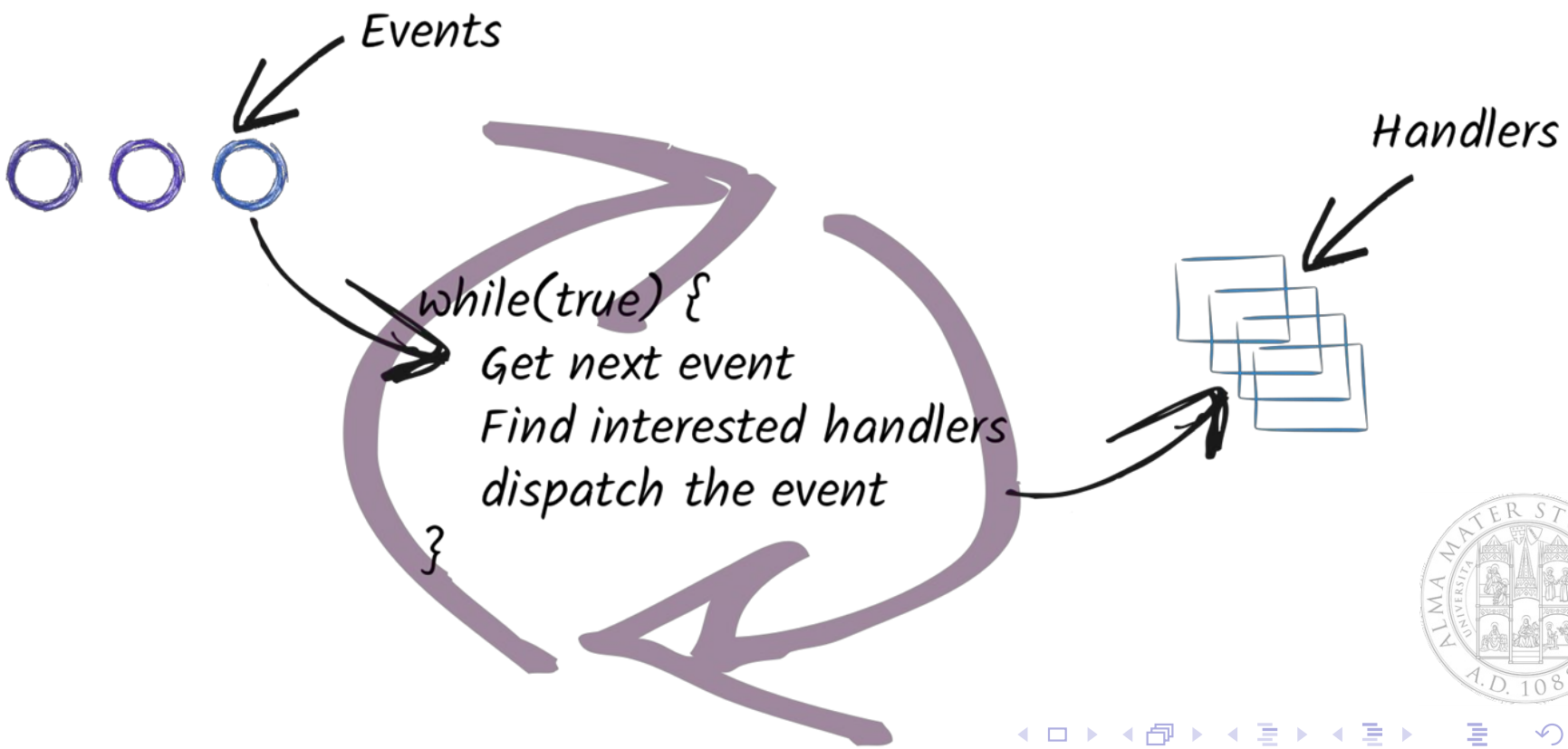
Terminologia

- **Istanza di Vert.x**: centro di controllo dell’ecosistema di Vert.x.
- **Verticle**: modello di concorrenza e di deployment opzionale simile al modello ad attori.
- **Event Bus distribuito**: permette a parti differenti di una stessa applicazione (ma anche di applicazioni diverse) di comunicare tra loro.
- **Handler**: procedura che riceve gli eventi quando disponibili.

Reactor Pattern I

Event loop

Un event loop viene assegnato ad un singolo thread di sistema, accoda gli eventi in una coda FIFO e li consegna agli handler interessati quando disponibili.



Reactor Pattern II

Vantaggi

- Utilizzo ottimale della CPU
- Può gestire più richieste a parità di hardware
- Può gestire un grande numero di client allo stesso tempo

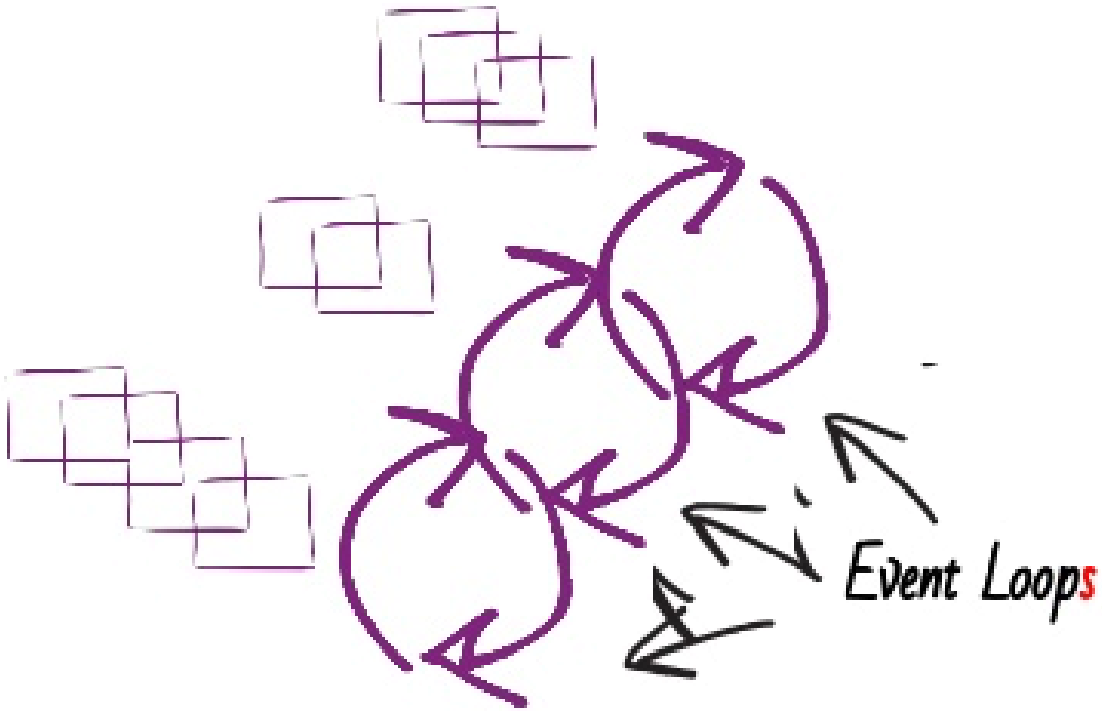
Svantaggi

- Struttura del codice difficile da comprendere
- Debug complesso

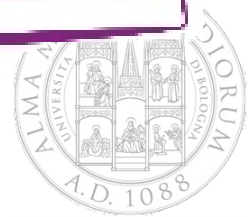
Multi-Reactor Pattern

Definizione

Mantiene diversi event loop permettendo ai singoli processi di scalare attraverso i CPU core della macchina sulla quale sta eseguendo.



Handlers are always executed in the same event loop.



“In the beginning there was Vert.x”

- Solitamente uno per istanza della JVM
- Punto d’ingresso per la API di Vert.x
- Contenitore per i `Verticle`

Creare un’istanza Vert.x

```
Vertx vertx = Vertx.vertx();
```

Creare un’istanza “raggruppata”

```
Vertx.clusteredVertx(new VertxOptions(), ar -> {
   if (ar.succeeded()) {
      Vertx vertx = ar.result();
      ...
   } else {
      System.err.println(ar.cause().getMessage());
   }
});
```

Verticle

- Pezzo di codice che viene deployato ed eseguito da Vert.x
- Può essere scritto in un qualsiasi linguaggio supportato
- Ogni `Verticle` può avere il proprio `ClassLoader`
- Ogni `Verticle` comunica con gli altri `Verticle` attravero l'`EventBus`

Scrivere un Verticle

```
class MyVerticle extends AbstractVerticle {
   public void start() {
   vertx.createHttpServer()
      .requestHandler(request -> request
          .response()
          .end("Hello from Vert.x"))
      .listen(8080);
   }
}
```

Event Bus

- **Sistema nervoso** di Vert.x
- Uno per ogni istanza di Vert.x
- Supporta vari pattern per il passaggio asincrono di messaggi
  - Publish/Subscribe
  - Point to Point
  - Request-Response
- Consegna best-effort
- Utilizza JSON come formato per lo scambio di dati
  - È possibile inviare messaggi arbitrari definendone una codifica personalizzata

Ottenere l'Event Bus

```
EventBus eventBus = vertx.eventBus();
```

“Don’t call us, we’ll call you”

In Vert.x gli eventi sono gestiti fornendo un handler alle API di Vert.x.
Alcuni esempi di eventi sono:

- Alcuni dati sono stati letti dal disco
- È stata sollevata un’eccezione
- Un server HTTP ha ricevuto una richiesta

Un generico event handler

```
@FunctionalInterface
public interface Handler<E> {
   /**
    * Something has happened, so handle it.
    *
    * @param event the event to handle
    */
   void handle(E event);
}
```

Ecosistema

Sistema di moduli

- Core
- Web (Web, Web Client, Web API Contract)
- Data Access (MongoDB client, JDBC client, etc.)
- Reactive (Vert.x Rx, Reactive streams, etc.)
- Microservices (Service Discovery, Circuit Breaker, Config)
- IoT (MQTT)
- Authentication and Authorisation (JWT auth, OAuth 2, etc.)
- Event Bus Bridge (TCP Eventbus Bridge, Camel Bridge)
- Devops (Docker, Stack Manager, etc.)
- Clustering (Hazelcast, Apache Zookeeper, etc.)
- E molti altri...

Outline

Architettura del sistema I

Il sistema *Event Tracker* permette il tracciamento nel tempo di eventi rilevati dai sensori presenti nell'ambiente e di gestire possibili situazioni di allarme.

Componenti

- Sottosistema S1 su Raspberry Pi, che funge da centralina
  - Led verde L1
  - Led verde L2
  - Led rosso L3
  - Pulsante tattile T1
- Sottosistema S2 su Rasperry Pi, con trasduttori
  - Sensore temperatura (analogico o digitale)
  - Rilevatore di movimento PIR
  - Dispositivo di attuazione a scelta A
- Sottosistema S3 su Android
- Sottosistema S4 su portatile

Architettura del sistema II

Funzionamento

- Sottosistema S1 - **Centalina**:
  - Far lampeggiare il led L2 ogni volta che avviene una comunicazione fra S1 e S2.
  - Accendere il led L3 quando viene segnalata una situazione di allarme.
  - Disattivare l'allarme alla pressione del pulsante tattile T1 per riportare il sistema al normale funzionamento.
  - Notificare S3 quando viene rilevata una presenza.
- Sottosistema S2 - **Sensori e attuatori**:
  - Notificare S1 quando viene rilevata una presenza tramite rilevatore di movimento.
  - Notificare S4 per il tracciamento della temperatura (considerando campionamenti con un certo periodo P) e gli eventi relativi alle presenze rilevate.
  - Attivare l'attuatore A in caso di allarme.

Architettura del sistema III

Funzionamento

- Sottosistema S3 - **Applicazione Android**:
  - Mostrare un alert quando viene rilevata una presenza per chiedere all'utente se notificare o meno S1 di una situazione di allarme.
  - Nel caso la risposta dell'utente non avvenga entro 10 secondi, S1 notificherà la situazione di allarme.
- Sottosistema S4 - **Web Server**:
  - Rendere accessibili/consultabili i dati rilevati via web.itemize
  - Salvataggio dei dati su piattaforma cloud DaaS (Database-as-a-Service).

Implementazione I

Il sistema è stato realizzato con Vert.x e si è scelta una decomposizione in `Verticle` delle funzionalità dei sottosistemi.

Verticle

- Sottosistema S1
  - `AlarmButtonManager`
  - `AlarmLightManager`
  - `AlarmManager`
  - `CommunicationBlinker`
- Sottosistema S2
  - `ActuatorManager`
  - `PresenceManager`
  - `TemperatureManager`
- Sottosistema S4
  - `EventTrackerServer`
  - `PersistenceManager`

Implementazione II

Il Sottosistema S3 è stato realizzato su piattaforma Android. La comunicazione degli altri sottosistemi con S3 è coordinata dal verticle `TcpCommunicationBridge`.

Problema

Android può mandare il messaggio `StartAlarm` ma deve anche stare in ascolto dello stesso. Questo comporta un autoinvio del messaggio.

Soluzione

Per evitare questa incoerenza è stato scelto di utilizzare `TcpCommunicationBridge` per differenziare i messaggi inviati da Android dai messaggi inviati dagli altri sottosistemi.

Implementazione III

Il sistema costruito è basato sullo scambio di messaggi tra i `Verticle` citati in precedenza.

Semantica dei messaggi

- `IgnoreAlarm`: messaggio inviato quando l'utente decide di non segnalare l'allarme.
- `StartAlarm`: messaggio inviato quando l'utente decide di segnalare l'allarme.
- `StopAlarm`: messaggio inviato quando viene premuto il pulsante tattile `T1`.
- `Temperature`: messaggio inviato periodicamente alla rilevazione della temperatura ambientale.
- `Presence`: messaggio inviato alla rilevazione di una presenza nell'ambiente.

Implementazione IV

Formato dei messaggi

- Per i messaggi è stato scelto di utilizzare il formato JSON.
- Tutti i messaggi contengono le chiavi “timestamp” e “ type”.
- Il messaggio `Temperature`, oltre alle chiavi sopracitate, contiene anche la chiave “value", rappresentante il valore della temperatura rilevata.

Esempio del messaggio `Temperature`

```
{
    "timestamp": "2018-04-02 15:55:56.378",
    "value": 18.996631622314453,
    "type": "temperature"
}
```

Implementazione V

È stato utilizzato l'Event Bus di Vert.x per lo scambio di messaggi tra i `Verticle`. Sono stati definiti i seguenti canali di comunicazione:

Canali di comunicazione

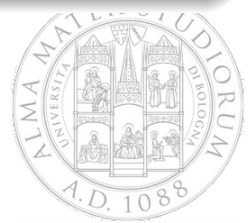
- `events`: canale utilizzato per il dialogo tra i sottosistemi S1, S2 e S4.
- `events.android.server`: canale utilizzato per mandare i messaggi `Presence` e `StartAlarm` ad S3.
- `events.android.client`: canale utilizzato per mandare i messaggi riguardanti le decisioni dell'utente (`StartAlarm` e `IgnoreAlarm`) da S3 a S1 e S4 .
- `events.web`: canale utilizzato per inviare i messaggi relativi agli eventi rilevati nell'ambiente ai client in ascolto.

Outline

Conclusioni

Abbiamo visto come...

- Vert.x può essere utilizzato come Message-Oriented Middleware in applicazioni distribuite.
- I `Verticle` possono essere utilizzati per decomporre l'applicazione in modoli disaccoppiati.
- L'`EventBus` può essere utilizzato come mezzo di comunicazione per lo scambio asincrono di messaggi tra `Verticle`.
- Vert.x, utilizzando il Multi-Reactor Pattern, permette al sistema di fornire tempi di risposta rapidi e consistenti.



Riferimenti I


Philip A Bernstein.
Middleware: a model for distributed system services.
Communications of the ACM, 39(2):86–98, 1996.


Toni A Bishop and Ramesh K Karne.
A survey of middleware.
In *Computers and Their Applications*, pages 254–258, 2003.


Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson.
The reactive manifesto.
`https://www.reactivemanifesto.org`, 2014.


Edward Curry.
Message-oriented middleware.
Middleware for communications, pages 1–28, 2004.

Riferimenti II

Tim Fox.

Eclipse vert.x.

https://vertx.io, 2011.

TechEmpower.

Web framework benchmarks.

https://www.techempower.com/benchmarks/#section=data-r8&hw=i7&test=plaintext, 2013.

Vert.x come MOM per Sistemi Distribuiti

Martina Magnani, Mattia Vandi
{martina.magnani8,mattia.vandi}@studio.unibo.it

C.d.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum — Università di Bologna, Cesena

Anno Accademico 2017/2018