actor-d: framework ad attori per D

Progetto di **Sistemi Distribuiti**

Alex Speranza

Introduzione a D

D è un linguaggio compilato nato come reingegnerizzazione di C++.

Pro:

- Esecuzione veloce (compilato in codice nativo)
- Multiparadigma (oggetti, funzionale, multithread, meta)
- Libreria standard molto vasta
- Compatibile con librerie e codice C/C++

Contro:

- Ecosistema poco sviluppato

Usato, tra le tante, da Facebook, Ebay e Netflix (fonte: https://dlang.org/orgs-using-d.html).

Esempio di Hello World:

```
import std.stdio;

void main()
{
    writeln("Hello, world!");
}
```

Il modello ad attori

È un modello per la computazione concorrente, parallela e/o distribuita.

L'unità fondamentale del sistema è l'**attore**, mentre l'unità di comunicazione è il **messaggio**.

Gli attori comunicano tra di loro inviando messaggi.

Un attore deve poter eseguire 3 operazioni fondamentali:
1) **send**: invio di un messaggio ad un altro attore
2) **become**: cambiare reazioni ai messaggi
3) **create**: creare nuovi attori

Gli attori sono entità reattive, in quanto si attivano solo in reazione ad un messaggio ricevuto.

Cosa esiste già

Search results for: *actor*

Package	Latest version	Date	Score	Description
eventcore	0.9.20	2021-Dec-17	4.7	Pro-actor based abstraction layer over operating system asynchronous I/O facilities.
dakka	0.0.4	2015-Jan-18	1.4	An actor framework, that supports remote method calls.

Found 2 packages.

Fonte (13/01/2022): https://code.dlang.org/search?q=actor

1) **eventcore:** event loop asincrono generico
2) **dakka**: framework ad attori, ultimo aggiornamento 2015, dipende da vibe-d

Obiettivi

Obiettivo principale:

- Implementazione modello ad attori

Obiettivi non funzionali:

- Astrazione della locazione degli attori
- Ottimizzazione delle risorse
- Disaccoppiamento di attori e thread

Proprietà:

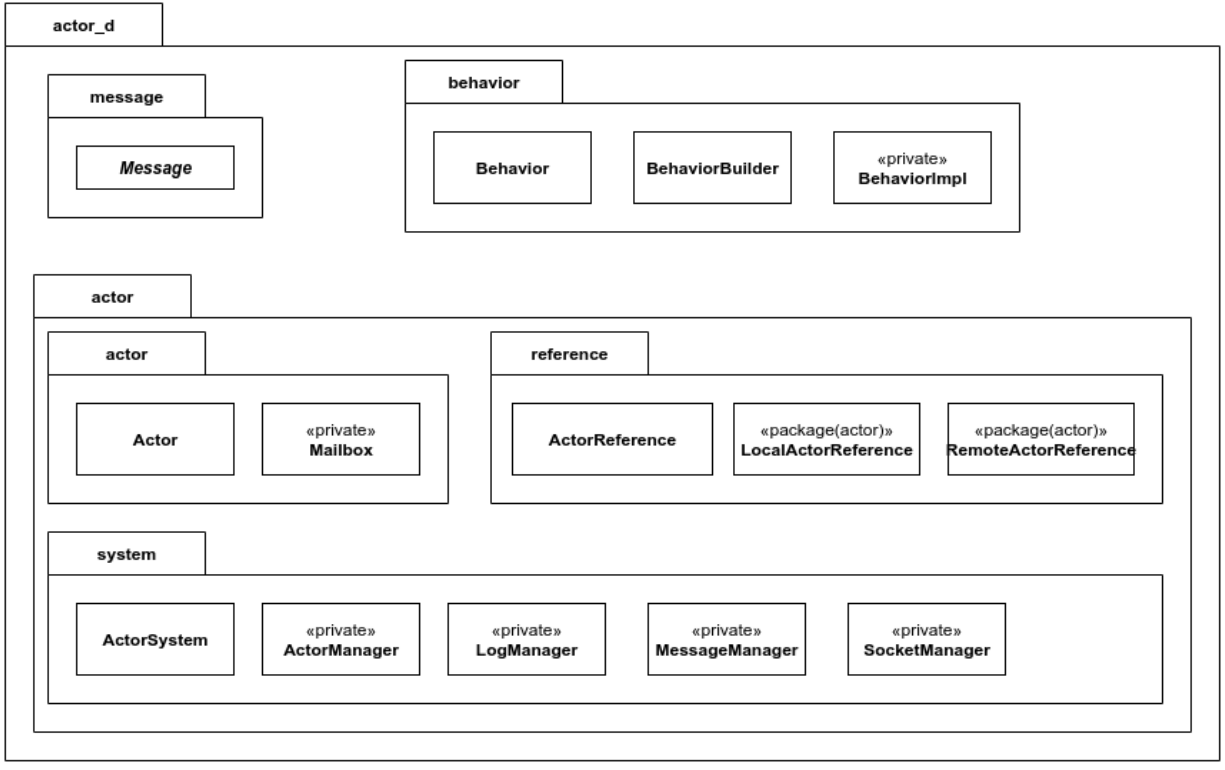
- Scalabilità
- Semplicità
- Astrazione

Struttura

La libreria è sviluppata seguendo il paradigma ad oggetti e la struttura e i componenti sono ispirati ad Akka.

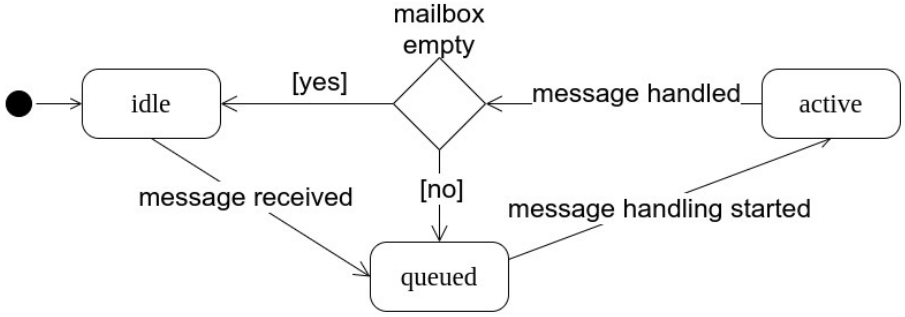
Classi principali:
- Actor
- Message
- ActorSystem
- ActorReference

Altre classi:
- Behavior
- Mailbox



Qualche dettaglio implementativo

L’astrazione scelta per il flusso di controllo degli attori è la **Task**.



Per far comunicare due attori remoti sono stati usati i **socket** e il formato **JSON** per la serializzazione.

```
{
  "messageType": <classinfo.name>,
  "message": <message.toJson>
}
```

Codice d’esempio

```
void main(string[] args)
{
    ushort portToListenTo = args[1].to!ushort;
    ushort portToSendTo = args[2].to!ushort;
    bool isStart = args[3].to!bool;

    ActorReference actor1 = ActorSystem.instance.create!PingActor(portToListenTo, portToSendTo);

    ActorSystem.instance.register!PingMessage(toDelegate(&jsonToPingMessage));

    if (isStart)
    {
        actor1.forward(new PingMessage(0));
    }
}
```

```
alex: .../actor-d/examples/remote-actors ( main ) [0]> ./remote-actors 10000 10001 fals
e
Actor: ping 1
Actor: ping 3
Actor: ping 5
Actor: ping 7
Actor: ping 9
Actor: ping 11
Actor: ping 13
Actor: ping 15
Actor: ping 17
Actor: ping 19
```

```
alex: .../actor-d/examples/remote-actors ( main ) [0]> ./remote-actors 10001 10000 true
Actor: ping 0
Actor: ping 2
Actor: ping 4
Actor: ping 6
Actor: ping 8
Actor: ping 10
Actor: ping 12
Actor: ping 14
Actor: ping 16
Actor: ping 18
Actor: ping 20
```

Conclusioni

- Modello ad attori implementato
- La località degli attori è astratta agli stessi
- Attori e thread disaccoppiati
- Risorse abbastanza ottimizzate

Grazie per l’ascolto!