QakActors25Demo

Progetto qakdemo24 : Esempi di uso di *QakActors25*

democodedqactor.qak	Sistema basato su istanze di un attore codificato in Kotlin, ciascuna delle quali emette un evento. Si evidenzia la opportunità di introdurre un linguaggio di modellazione.
demo0.qak	Interzioni con messaggi **dispatch, event**.
demoguards.qak	Uso di guardie sulle transizioni.
demorequest.qak	Interazione **request-reponse**.
demoasktocaller.qak	Un attore chiamato ad eseguire una richiesta, chiede dati ulteriori al chiamante.
demoresource.qak	Creazione incrementale di un sistema, a partire da un core stand-alone.
demoaddtocore.qak	Aggiunta al sistena precedente di un attore in un nuovo contesto
demoobservability.qak	Sistema con un actor che rende osservabile il suo stato ad altri attori, inclusi 'alieni'
demodelegate.qak	Sistema con un *server* che delega la gestione di alcuni tipi di messaggio (tra cui richieste) a un altro attore.
democreate.qak	Sistema in cui due produttori, reati dinamicamnete, inviano richieste a un consumatore che delega un tipo di richiesta a un altro attore, creato dinamicamente allo scopo.
demomqttexplicit.qak	Sistema con attori che fanno **publish/subscribe** sulla topic di un **mqtt broker** dichiarato a livello di attore, con connessione esplicita da parte dell'attore.
demomqttimplicit.qak	Sistema con attori che fanno **publish/subscribe** sulla topic di un **mqtt broker** dichiarato a livello di sistema, con connessione automatica alla creazione di ogni attore.
demostreams.qak	Sistema con attori che reagiscono a dati inviati attraverso **eventi-stream**.
demointerrupt.qak	Gestione di un evento con transizione a uno stato che opera come una interrupt-routine.

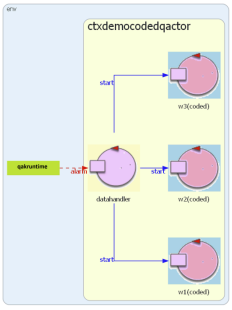
Esempio CodedQActor

democodedqactor.qak

`democodedqactor.qak code`_

Questo esempio introduce tre istanze di un attore *workactor* codificato in Kotlin, ciascuna delle quali emette un evento.

Modello del sistema.




```
System  qakdemo24
Dispatch start   : start( ARG )
Event alarm      : alarm( DATA )

Context ctxdemocodedqactor ip [host="localhost" port=8065]

CodedQActor w1 context ctxdemocodedqactor className "codedActor.workactor"
CodedQActor w2 context ctxdemocodedqactor className "codedActor.workactor"
CodedQActor w3 context ctxdemocodedqactor className "codedActor.workactor"

QActor datahandler context ctxdemocodedqactor{...}
```

---

datahandler

```
QActor datahandler context ctxdemocodedqactor{
[# var StartTime = 0L #]
  State s0 initial {
    printCurrentMessage
/*1*/[# val cpus =
      Runtime.getRuntime().availableProcessors() #]
    println("cpus= $cpus") color red
/*2*/forward w1 -m start : start(do)
/*3*/forward w2 -m start : start(do)
/*4*/forward w3 -m start : start(do)
  }
  Transition t0
    whenEvent alarm -> handleAlarm

  State handleAlarm {
    printCurrentMessage color blue
    [# var Elapsed = 0L #]
/*5*/setDuration Elapsed from StartTime
/*6*/memoCurrentTime StartTime
    println("$name alarm after $Elapsed") color cyan
  }
Transition t0
/*7*/whenTime 2500 -> end
/*8*/whenEvent alarm -> handleAlarm

  State end{
    [# var Elapsed = 0L #]
  /*9*/setDuration Elapsed from StartTime
    println("$name ENDS duration=$Elapsed")
  }
}
```

1. Acquisizione del numero di CPU disponibili
2. Invio di richiesta **start** a **w1**
3. Invio di richiesta **start** a **w2**
4. Invio di richiesta **start** a **w3**
5. Caloclo, usando *setDuration* del tempo trascorso prima della percezione dell'evento **alarm**
6. Memorizzazione del tempo corrente
7. Attesa di **2500** msec. Se non arriva l'evento **alarm** si va allo stato end e si termina
8. Attesa di un evento **alarm**

---

codedActor.workactor

Il codice di **workactor**, scritto in Koltin, è il seguente:

```
class workactor(name:String):
/*1*/      ActorBasic(name,
/*2*/                 confined=true){
var i = 0
  override suspend fun actorBody(msg:IApplMessage){
/*3*/if( msg.msgId() == "start"){
      workStep(   )
    }else
    CommUtils.outgreen("$name|received  $msg ")
  }

  suspend fun workStep(    ){
    i++
    val alarm=CommUtils.buildEvent(
                name,"alarm","alarm$name-$i")
  /*4*/emit( alarm )
    if( i == 3 ) terminate()
    else {
      delay( 2000L   )
      forward("start", "start(do)" , name )
    }
  }
}
```

1. La classe **workactor** estende *ActorBasic* e ridefinisce il metodo **actorBody**.
2. Il valore **confined=true** trasferito al costruttore implica che le istanza di questa classe sono attivate usando il dispatcher *newSingleThreadContext* -> che utilizza solo 1 Thread. (si veda *Coroutine context and dispatchers* ->).

   Ponendo :**confined=false** (valore di default), viene usato il dispatcher *newFixedThreadPoolContext* ->, che gestisce gli attori attivando tanti thread quante le CPU disponibili.
3. Attivazione del metodo **workStep** in risposta a un messaggio **start**.
4. Il metodo **workStep** emette un evento **alarm** e, dopo un certo numero di iterazioni, termina.

Codice vs. modelli

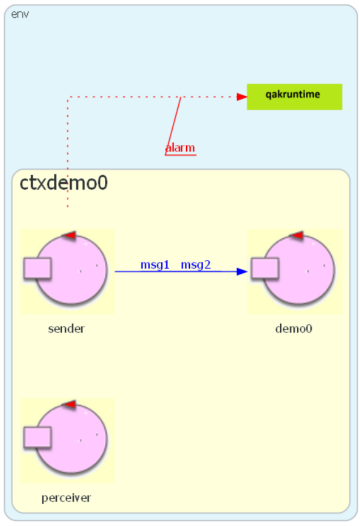
- notiamo che il **workactor** non esprime in modo evidente il tipo di informazione che emette come evento. Si tratta infatti di codice vero e proprio e *non di un modello* ;
- leggendo il codice, capiamo che l'evento emesso ha la forma **alarm:alarm(DATA)** e che corrisponde quindi a una dichiarazione Qak quale quella introdotta in *Modello del sistema*:

```
Event alarm : alarm( DATA )
```

Esempio demo0

demo0.qak

demo0.qak code descrive un sistema rappresentato nella figura che segue:



qakdemo24Arch

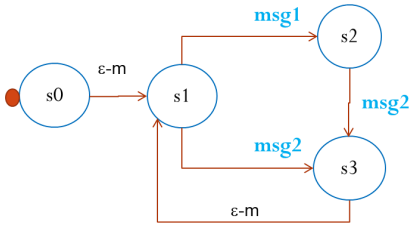
```
System qakdemo24

Dispatch msg1 : msg1(ARG)       "da sender a demo0"
Dispatch msg2 : msg2(ARG)            "da sender a demo0"
Event alarm   : alarm( KIND )

Context ctxdemo0 ip [host="localhost" port=8095]

QActor demo0 context ctxdemo0{ ... }
QActor perceiver context ctxdemo0{ ... }
QActor sender context ctxdemo0{ ... }
```

- *demo0* : definisce il modello eseguibile della figura a sinistra
- **sender**: attore che invia i messaggi gestiti da **demo0** e genera (opzionalmente) un evento
- **perceiver**: attore che gestisce gli eventi emessi da **sender**

QActor demo0

```
QActor demo0 context ctxdemo0{
  State s0 initial {
/*1*/ discardMsg Off  //discardMsg On
/*2*///[# sysUtil.logMsgs=true #]
    println("myself=${myself.name}")
    println("curState=${currentState.stateName}")
    println("currentMsg=${currentMsg}")
  }
/*3*/Goto s1

  State s1{
    printCurrentMessage
  }
  Transition t0 whenMsg msg1 -> s2
                whenMsg msg2 -> s3

  State s2{
    printCurrentMessage
/*4*/ onMsg( msg1:msg1(V)){
/*5*/ println("s2: msg1(${payloadArg(0)})")
      delay 1000
    }
  }
/*6*/Transition t0 whenMsg msg2 -> s3

  State s3{
    printCurrentMessage
/*7*/onMsg( msg2:msg2(1)){
      println("s3: msg2(${payloadArg(0)})")
    }
  }
  Goto s1
}
```

1. *discardMsg Off*: i messaggi che non sono di interesse nello stato vengono conservati, mentre con **discardMsg On**, essi vengono eliminati.
2. crea dei file di log dei messaggi ricevuti
3. *EmptyTransition*
4. *Accesso al contenuto dei messaggi*: esegue il body solo se il *messaggio corrente* ha identificatore **msg1** e il suo payload **msg1(ARG)** può essere **unificato** con il template della dichiarazione e con il template **msg1(V)**.
5. *payloadArg*: accesso al valore di **ARG**
6. *NonEmptyTransition*
7. considera solo messaggi con **msgId==msg2** e payload **msg2(1)**



La figura mostra *demo0* come *FSM*

---

demo0 sender

Timeout per transizioni

Il **sender**

```
QActor sender context ctxdemo0{
  [# var emitEvents = true
```

1. invia un primo dispatch **msg1** a *QActor demo0*
2. invia un secondo dispatch **msg1** a *QActor demo0*
3. dopo un tempo **DT** transita di stato
4. invia un dispatch **msg2** a *QActor demo0*
5. azione condizionale
6. emette l'evento **alarm**

invia alcuni messaggi e genera un evento se **emitEvents = true**. Si noti l'uso di *whenTimeVar*.

```
    var DT          = 1000L;
  #]
  State s0 initial {
    printCurrentMessage color black
    println( "sender sends ... ") color green
/*1*/forward demo0 -m msg1 : msg1(1)
    delay 300
/*2*/forward demo0 -m msg1 : msg1(2)
  }
  Transition t0
/*3*/ whenTimeVar  DT -> sendothermsgs

  State sendothermsgs{
    println( "sender sends again ... ") color green
/*4*/forward demo0 -m msg2 : msg2(1)
/*5*/if [# emitEvents #] {
/*6*&  emit alarm : alarm( fire )
    }
}
```

---

demo0 perceiver

Timeout per transizioni

Il **perceiver**

1. gestisce l'evento **alarm**
2. *whenTime* se scade un delay di **100** msec va a **s2**
3. se prima dei **100** msec è stato emesso l'evento **alarm** rimande in **s1**

```
QActor perceiver context ctxdemo0{
   State s0 initial {
     printCurrentMessage color black
   }
/*1*/ Transition t0 whenEvent alarm->s1

   State s1{
     printCurrentMessage color magenta
   }
   Transition t0
 /*2*/ whenTime 100    -> s2
 /*3*/ whenEvent alarm- > s1

   State s2{
     println("perceiver BYE")
   }
}
```

Analisi dei risultati

Output con discardMsg On

Caso sender: **emitEvents = false**

```
sender sends ...
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
demo0 in s2 | msg(msg1,dispatch,sender,demo0,msg1(1),10)
demo0 in s2 | msg1:msg1(1)
sender sends again ...
demo0 in s3 | msg(msg2,dispatch,sender,demo0,msg2(1),13)
demo0 in s3 | msg2:msg2(1)
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
```

Questo caso evidenzia anche che:

una *EmptyTransition* è realizzata con emissione di un evento **local_noMsg**; i messaggi in arrivo sono memorizzati in *msgQueueStore*

Caso sender: **emitEvents = true**

```
sender sends ...
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
demo0 in s2 | msg(msg1,dispatch,sender,demo0,msg1(1),10)
demo0 in s2 | msg1:msg1(1)
sender sends again ...
demo0 in s3 | msg(msg2,dispatch,sender,demo0,msg2(1),13)
demo0 in s3 | msg2:msg2(1)
perceiver in s1 | msg(alarm,event,sender,none,alarm(fire),14)
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
perceiver in s2 | msg(local_tout_perceiver_s1,
                event,timer,none,local_tout_perceiver_s1,15)
perceiver BYE
```

Questo caso evidenzia anche che:

lo scadere del *timeOut* provoca l'emissione di un evento con indentificatore **local_tout_perceiver_s1**.

Output con discardMsg Off

Caso sender: **emitEvents = false**

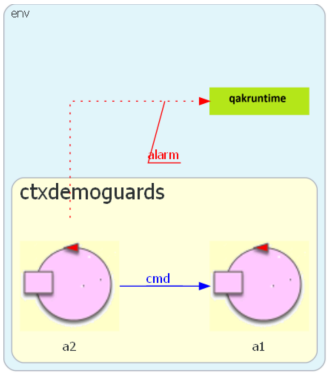
```
sender sends ...
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
demo0 in s2 | msg(msg1,dispatch,sender,demo0,msg1(1),10)
demo0 in s2 | s2:msg1:msg1(1)
  %%%  ActorBasicFsm demo0 | added
  msg(msg1,dispatch,sender,demo0,msg1(2),11) in msgQueueStore
demo0 in s3 | msg(msg2,dispatch,sender,demo0,msg2(1),12)
demo0 in s3 | s3:msg2:msg2(1)
demo0 in s1 | msg(local_noMsg,event,demo0,none,noMsg,4)
demo0 in s2 | msg(msg1,dispatch,sender,demo0,msg1(2),11)
demo0 in s2 | s2:msg1:msg1(2)
```

Si nota la memorizzazione in *msgQueueStore*

Esempio demoguards

demoguards.qak

demoguards.qak code è relativo all'uso delle guardie.




```
System qakdemo24
Dispatch cmd   : cmd( ARG )
Event alarm    : alarm( DATA )

Context ctxdemoguards ip [host="localhost" port=8065]

QActor a1 context ctxdemoguards{...}
QActor a2 context ctxdemoguards{...}
```

guards a1

```
QActor a1 context ctxdemoguards{
/*1*/[# var ready    = true
   var end       = false
   fun check( v : String ) : Boolean{
      return v =="cmd(continue)"
  }
#]
  State s0 initial {
    println("$name in s0 end=$end ready=$ready")
  }
  /*2*/Goto end if [# end #] else s1

  State end{
    println("$name BYE") color magenta
    [# System.exit(0) #]
  }

  State s1  {
      println("$name in s1 ready=$ready") color blue
  }
  Transition t0
  /*3*/whenEvent alarm and [# ready #]  -> handleAlarm
  /*4*/whenMsg cmd     and [# ready #]  -> handleCmd
  /*5*/whenMsg cmd     and [# ! ready #] -> checkCmd

  State handleAlarm  {
    println("$name in handleAlarm ready=$ready")
  /*6*/ [# ready = false #]
  }
  Goto s1

  State handleCmd  {
    printCurrentMessage
  /*7*/onMsg( cmd : cmd(end) ){
        [# end = true #]
    }
```

1. Variabili e funzioni booleane
2. *EmptyTransition* con guardia iniziale **false**
3. Transizione allo stato di gestione dell'evento, con guardia **ready==true**
4. Transizione allo stato di gestione del dispatch, con guardia **ready==true**
5. Transizione allo stato di gestione del dispatch, con guardia **ready==false**
6. La gestione dell'evento **alarm** imposta **ready=false**
7. La gestione del dispatch **cmd(end)** imposta **end=true**
8. *EmptyTransition* con guardia **end**
9. Gestione del dispatch **cmd(continue)** con guardia **ready** settata dalla funzione **check**

```
  }
  /*8*/Goto end if [# end #] else s1

  State checkCmd  {
    println("$name checks $currentMsg") color magenta
  /*9*/[# ready = check( currentMsg.msgContent() ) #]
  }
  Goto s1
}
```

---

guards a2

```
QActor a2 context ctxdemoguards{
  State s0 initial {
  /*1*/forward a1 -m cmd : cmd(10)
  /*2*/emit alarm:alarm(fire)
  /*3*/forward a1 -m cmd : cmd(20)
  /*4*/emit alarm:alarm(tsunami)
  /*5*/forward a1 -m cmd : cmd(continue)
  /*6*/forward a1 -m cmd : cmd(30)
  /*7*/forward a1 -m cmd : cmd(end)
  }
}
```

1. Invio di un dispatch **cmd(10)** a **a1**, che viene gestito
2. Emissione di un evento **alarm(fire)**, che viene gestito da **a1**
3. Invio di un dispatch **cmd(20)** a **a1**, che non viene gestito
4. Emissione di un evento **alarm(tsunami)**, che non viene gestito da **a1**
5. Invio di un dispatch **cmd(continue)** a **a1**, che viene gestito
6. Invio di un dispatch **cmd(30)** a **a1**, che viene gestito
7. Invio di un dispatch **cmd(end)** a **a1**, che viene gestito terminando il sistema

---

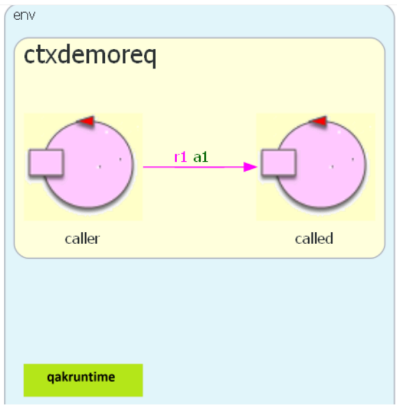
Esempio demorequest

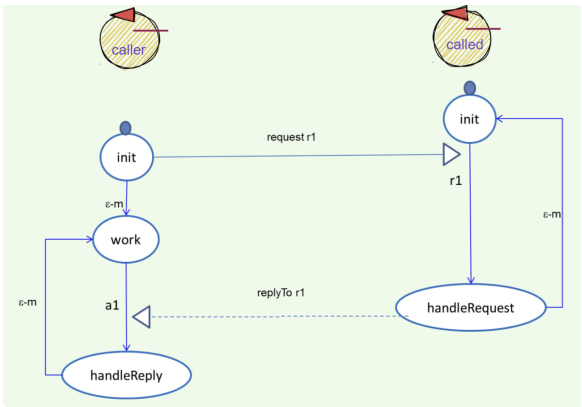
demorequest.qak

demorequest.qak code

Questo esempio è relativo alla interazione request-reponse.

Interazione *caller e called*





Messaggi e contesti:

```
System  -msglog qakdemo24
Request r1 : r1(X)
Reply   a1 : a1(X)

Context ctxdemoreq ip [host="localhost" port=8010]
```

Il flag *-msglog* popola, a run time, un insieme di log-files relativi ai messaggi ricevuti nella directory **logs**.

---

demorequest caller

```
QActor caller context ctxdemoreq {
   State init initial {
     println("caller starts") color blue
     delay 100 //give time to start called
 /*1*/request called -m r1 : r1(hello(world))
   }
   Goto work

   State work{
       //printCurrentMessage color blue
   }
 /*2*/Transition t0 whenReply a1 -> handleReply

   State handleReply{
     printCurrentMessage color blue
 /*3*/onMsg( a1:a1(ARG) ){
       println("answer=${payloadArg(0)}") color blue
       delay 1000
     }
   }
   Goto work
}
```

1. invia una request **r1** con payload **r1(hello(world))** all'attore locale **called**
2. attende il messaggio di reply **a1**.
3. Si ricordi lo schema di *Accesso al contenuto dei messaggi*.

L'output è:

```
answer=called_caller_hello(world)
```

---

demorequest called

```
QActor called context ctxdemoreq {
    State init initial {
      println("called waits ...") color green
    }
    Transition t0
      whenRequest r1 -> handleRequest

    State handleRequest{
      printCurrentMessage color green
      onMsg( r1 : r1(X) ){
        [# val Answer="${currentMsg.msgSender()}_${payloadArg(0)}" #]
        replyTo r1 with a1 : a1( $Answer )
      }
    }
    Goto init
}
```

called` attende una richiesta e quindi invia al chiamante la reply **a1** con payload

```
a1(called_caller_hello(world))
```

Si noti l'uso di **currentMsg** per definire il paylod **Answer** della reply.

Esempio demoasktocaller

demoasktocaller.qak

demoasktocaller.qak code

Un attore *caller* invia una richiesta **r1** a un attore *called* che, per rispondere, ha bisogno di altri dati. Per questa ragione, il *called* invia a sua volta una richiesta **r2** per i dati mancanti al *caller* e, solo dopo avere ricevuto i dati stessi come risposta a **r2**, elabora la richiesta **r1** mandando infine la risposta per **r1** al *caller*.

```
System /*-msglog  -trace */ qakdemo24

Request r1 : r1(X) "da caller a called"
Request r2 : r2(X) "da called a called"
Reply   a1 : a1(X) for r1
Reply   a2 : a2(X) for r2
```

Per realizzare questo tipo di interazione, linguaggio qak fornisce la primitiva *ask*

demoasktocaller caller

```
QActor caller context ctxask {
  State init initial {
    delay 500 //for called to be started
/*1*/request called -m r1 : r1(10)
    //delay 1000
/*2*/request called -m r1 : r1(20)
```

1. Invio prima richiesta **r1**
2. Invio seconda richiesta **r1**
3. Attesa risposta a **r1**
4. Attesa risposta a **r2**
5. Attesa richiesta dati

```
  }
  Goto work

  State work{
    //println("       caller work")
  }
  Transition t0
/*2*/    whenReply   a1 -> handleReply
/*3*/    whenReply   a2 -> handleReply
/*4*/    whenRequest r2 -> handleAskFromCalled

  State handleReply{
/*5*/ println("$name handleReply $currentMsg") color blue
  }
  Goto work

  State handleAskFromCalled{
/*6*/onMsg( r2 : r2(X) ) {
      println("       $name handles ask r2(${payloadArg(0)})")
      println("       $name replies with a2(90)")
/*7*/ replyTo r2 with a2 : a2(90)
    }
  }
  Goto work
}
```

6. Gestione delle reply dal *called*
7. Analisi della richiesat dati
8. Risposta alla rchiesta dati fatta dal *called*

---

demoasktocaller called

```
QActor called context ctxask {
[# var RequestArg = "0" #]
  State init initial {
  }
  Transition t0
/*1*/ whenRequest r1 -> handleRequest


  State handleRequest{
/*2*/onMsg( r1 : r1(10) ){
      [# RequestArg = payloadArg(0) #]
      println(" called receives the request r1($RequestArg)")
      println(" called askfor r2(theta)") color magenta
/*3*/ ask  r2 : r2(theta) forrequest r1
    }
/*4*/onMsg( r1 : r1(20) ){
      [# RequestArg = payloadArg(0) #]
      println(" called receives the request r1(${payloadArg(0)})")
      [# val R = "${RequestArg}_${payloadArg(0)}" #]
/*5*/ replyTo r1 with a1 : a1( $R   )
    }
  }
  Transition t0
/*6*/whenTime 1000 -> init
/*7*/whenReply a2  -> answerAfterAsk

  State answerAfterAsk{
/*8*/onMsg( a2 : a2(X) ){
      println(" called receives answer to ask a1(${payloadArg(0)})")
      [# val R = "${RequestArg}_${payloadArg(0)}" #]
      println(" called replies to orginal request with a1($R)")
/*9*/ replyTo r1 with a1 : a1( $R   )
    }
  }
  Goto init
}
```

1. Attesa della richiesta **r1**
2. Caso della richiesta **r1** con payload **r1(10)**
3. Invio di richiesta dati al *caller* per richiesta con payload **r1(10)**
4. Caso della richiesta **r1** con payload **r1(20)**
5. Invio di risposta immediata a **r1** con payload **r1(20)**
6. Attesa dei dati richiesti
7. Dopo **1 sec** si ritiene conclusa la interazione con il *caller* (caso **4**)
8. Gestione della reply del *caller*
9. Risposta alla prima richiesta **r1** del *caller*

Il punto **6-7** implica che il *called* decide di non gestire la richiesta **r1(20)** prima di ricevere la risposta alla sua *ask* per **r1(10)**. La richiesta **r1(20)** viene quindi accodata in *msgQueueStore*

Esempio ExternalQActor

demoresource.qak

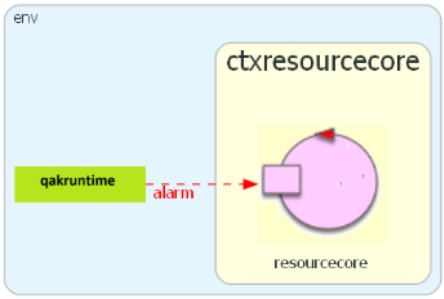
demoresource.qak code

Vi sono situazioni in cui un sistema si configura e si costruisce 'incrementalmente', partendo da un nucleo iniziale e poi aggiungendo altri componenti, che interagiscono con il nucleo e tra loro sempre mediante scambio di messaggi.

Supponiamo, come esempio, di introdurre come 'nucleo di base' un componente modellato come una singola risorsa/servizio.

Il sistema formato dalla sola *resourcecore* è descritto come segue:

```
System resourcecore

Request cmd        : cmd(X) //X=w|s|a|d|h
Reply   replytocmd : replytocmd(X)
Event    alarm     : alarm(V)

Context ctxresourcecore ip [host="localhost" port=8045]

QActor resourcecore context ctxresourcecore{...}
```



resourcecoreArch

---

resourcecore

```
QActor resourcecore context ctxresourcecore{
/*1*/[# var ready = true #]
  State s0 initial {
      println("$name waiting ... ") color blue
  }
  Transition t0
  /*2*/ whenRequest cmd and [# ready #]->handleRequestCmd
  /*3*/ whenEvent   alarm -> handleAlarm

  State handleAlarm{
    onMsg( alarm : alarm(on) ){
      println("$name alarm on") color red
  /*4*/[# ready = false #]
    }
    onMsg( alarm : alarm(off) ){
      println("$name alarm off") color red
  /*5*/[# ready = true #]
    }
  }
  Goto s0

  State handleRequestCmd{
    printCurrentMessage color blue
    onMsg( cmd : cmd(X) ){
      [# val ANSW = "answerFor_${payloadArg(0)}" #]
  /*6*/replyTo cmd with replytocmd : replytocmd($ANSW)
    }
  }
  Goto s0
}
```

1. **ready** variabile interna che controlla se il *resourcecore* è pronto a gestire una richiesta **cmd**.
2. Attesa di una richiesta **cmd** nel caso la *Guardia* **ready** sia vera
3. Attesa di un evento **alarm**
4. Nel caso di allarme **on**, **ready** diventa **false**
5. Nel caso di allarme **off**, **ready** diventa **true**
6. Risposta alla richiesta **cmd** con payload **replytocmd(ANSW)**

I messaggi di richiesta ricevuti durante gli allarmi sono memorizzati in *msgQueueStore* e gestiti ad allarme terminato.

---

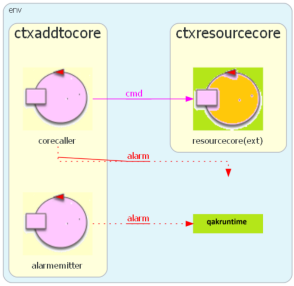
demoaddtocore.qak

demoaddtocore.qak code

Supponiamo ora che un ulteriore componente QakActor *corecaller* voglia 'fare sistema' con *resourcecore*, inviando una richiesta alla risorsa ed emettendo un evento **alarm(on)**.

Un altro componente *alarmoff*, emette, con un certo ritardo, l'evento **alarm(off)**.

```
System qakdemo24

Request cmd        : cmd(X) //X=w|s|a|d|h
Reply   replytocmd : replytocmd(X)
Event    alarm     : alarm(V)

Context ctxaddtocore     ip [host= "localhost" port=8038]
Context ctxresourcecore  ip [host= "127.0.0.1" port=8045]

ExternalQActor resourcecore context ctxresourcecore

QActor corecaller context ctxaddtocore{...}
QActor alarmoff context ctxaddtocore {...}
```



Si noti:

- la dichiarazione degli stessi tipi di messaggio di *resourcecore*.
- il componente *resourcecore* è dichiarato external: si veda *Attori external*.

I messaggi sono il 'collante' del sistema

---

corecaller

```
QActor corecaller context ctxaddtocore{
  [# var StartTime = 0L
    var Elapsed   = 0L
  #]
  State s0 initial {
    printCurrentMessage  color blue
/*1*/request resourcecore -m cmd : cmd(caller1)
/*2*/memoCurrentTime StartTime
  }
  Transition t0
     whenReply replytocmd -> handleReply

  State handleReply{
/*3*/setDuration Elapsed from StartTime
    println("$name handleReply Elapsed = $Elapsed ")
    println("$name handleReply: emit alarm(on)")
/*4*/emit alarm : alarm(on)
/*5*/request resourcecore -m cmd : cmd(2)
/*6*/request resourcecore -m cmd : cmd(3)
/*7*/memoCurrentTime StartTime
  }
  Transition t0
/*8*/ whenReply replytocmd -> handleReplyAgain

  State handleReplyAgain{
    printCurrentMessage color blue
/*9*/ setDuration Elapsed from StartTime
    println("$name handleReplyAgain Elapsed=$Elapsed")
  }
  Transition t0
      whenReply replytocmd -> handleReplyAgain
}
```

1. Invio di richiesta **cmd** a *resourcecore*
2. Memorizzazione del tempo corrente: si veda *memoCurrentTime*
3. Tempo trascorso alla ricezione della risposta a **cmd**
4. Emissione di evento **alarm(on)**
5. Invio di nuova richiesta **cmd** a *resourcecore*
6. Invio di ulteriore richiesta **cmd** a *resourcecore*
7. Attesa delle risposte
8. Tempo trascorso alla ricezione della risposta a **cmd**

---

alarmoff

```
QActor alarmoff context ctxcorecaller1 {
 State s0 initial {
    printCurrentMessage  color blue
/*1*/delay 2000
    println("$name alarmoff emit ")
/*2*/emit alarm : alarm(off)
 }
}
```

1. Attesa di **2 sec**
2. Emissione di evento **alarm(off)** che deattiva l'allarme e consetne a *resourcecore* di tornare **ready**

Esempio Osservabilità

demoobservability.qak

Un QActor ha la proprietà di essere **observable** da un altro actor o da un componente esterno al sistema (alieno).

In questa versione del sistema, l'attore **display** opera come **observer** dell'actor **worker** e visualizza sul Display embedded gli aggiornamenti che **worker** invia tramite un dispatch che l'*observer* dichiara di voler gestire col nome **info**.

osbserver

```
System qakdemo24

Dispatch info : info(SOURCE,TERM)
```

1. *observeResource worker msgid info* (si veda *Operazioni relative alla osservabilità*):

   l'attore *observer* opera come observer dell'actor *worker* che riceve aggiornamenti tramite il dispatch info:info(SOURCE,TERM)

2. *observer* può visualizzare un aggiormaneto usando la variabile ereditata

   currentMsg (si veda *Parti ereditate*)

   che referenzia il messaggio gestito nello stato corrente

3. *observer* può selezionare il contenuto del messaggio corrente usando la unificazione Prolog e il metodo ereditato *payloadArg(N)*

```
Context ctxobs ip [host="localhost" port=8004]

QActor observer context ctxobs {
  State s0 initial{
    delay 1000 //Give time for worker to start
/*1*/observeResource worker msgid info
  }
  Transition t0 whenMsg info -> handleinfo

  State handleinfo{
    printCurrentMessage
    println("$currentMsg") color blue
/*2*/[#
    CommUtils.outblue( currentMsg.toString() )
    CommUtils.outblue( "$currentMsg" )
    CommUtils.outblue( currentMsg.msgContent().toString() )
    CommUtils.outblue( "${currentMsg.msgContent()}" )
    #]
    onMsg( info:info(SOURCE,TERM)){
      [# val Source = payloadArg(0)
        val infoMsg = payloadArg(1)
        val M       = "$infoMsg from $Source"
      #]
      println("$M") color magenta
      //[# CommUtils.outmagenta( M ) #]
    }
  }
  Transition t0 whenMsg info -> handleinfo
}
```

---

worker come observable

1. emissione di informazione osservabile
2. emissione di informazione osservabile

```
QActor worker context ctxhello{
  State s0 initial{
    [# var n = 0 #]
/*1*/ updateResource [# "hello_${n++}" #]
    delay 2000
/*2*/updateResource [# "info(worker, hello_${n++})" #]
  }
}
```

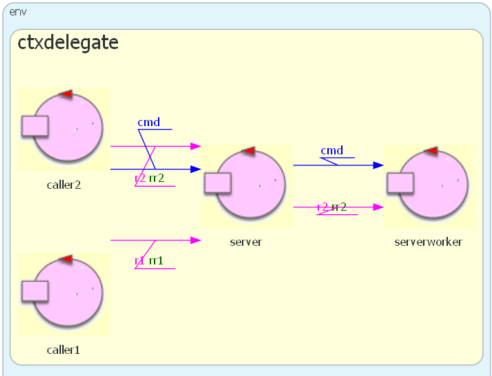
Esempio Delegazione

demodelegate.qak

demodelegate.qak code

- Il *server* gestisce il dispatch **cmd** e le richieste **r1** e **r2**
- Il server delega al *serverworker* la gestione del dispatch **cmd** e della richieste **r2**
- Il *caller1* invia la richiesta **r1** e gestisce la risposta inviata direttamente dal *server*
- Il *caller2* invia il dispatch **cmd** e la richiesta **r2** e gestisce la risposta inviata dal *serverworker*

```
System  qakdemo24

Request r1 : r1(X)
Reply rr1 : rr1(X) for r1

Request r2 : r2(X)
Reply rr2 : rr2(X) for r2

Dispatch cmd : cmd(X)

Context ctxdelegate ip [host="localhost" port=8045]

 QActor server context ctxdelegate{ ... }
 QActor serverworker context ctxdelegate{ ... }
 QActor caller1 context ctxdelegate{ ... }
 QActor caller2 context ctxdelegate{ ... }
```



---

server

server

```
QActor server context ctxdelegate{
  State s0 initial{
    delay 200
/*1*/ delegate r2 to serverworker
/*2*/ delegate cmd to serverworker
  }
  Goto work

  State work{
    println("$name working ...") color blue
  }
  Transition t0
/*3*/   whenRequest r1 -> handle_r1

  State handle_r1{
    printCurrentMessage color blue
/*4*/  replyTo r1 with rr1 : rr1(answer)
  }
  Goto work
}
```

1. Delega la gestione della request **r2** al *server-worker*
2. Delega la gestione del dispatch **cmd** al *server-worker*
3. Attesa della request **r1**
4. Risposta alla request **r1**

---

serverworker

serverworker

```
QActor serverworker context ctxdelegate{
  State s0 initial{
    //delay 1000
  }
  Goto work

  State work{

  }
  Transition t0
/*1*/  whenRequest r2 -> handle_r2
/*2*/  whenMsg cmd -> handlecmd


  State handle_r2{
    printCurrentMessage color blue
/*3*/  replyTo r2 with rr2 : rr2(answer)
  }
  Goto work


  State handlecmd{
/*4*/printCurrentMessage color blue
  }
  Goto work
}
```

1. Attesa della request delegata **r2**
2. Attesa del dispatch delegato **cmd**
3. Risposta alla request delegata **r2**
4. Gestion del dispatch delegato **cmd**

---

caller1

caller1

```
QActor caller1 context ctxdelegate{
  State s0 initial{
    delay 500
    println("$name request r1") color green
/*1*/ request server -m r1 : r1(10)
  }
  Transition t0
/*2*/ whenReply rr1 -> handleReply

  State handleReply{
/*3*/ printCurrentMessage color green
  }
}
```

1. Invia la request **r1** al *server*
2. Attesa della risposta
3. Gestione della risposta

---

caller2

caller2

1. Invia la request **r2** al *server*
2. Attesa della risposta

```
QActor caller2 context ctxdelegate{
  State s0 initial{
    println("$name request r2")    color magenta
/*1*/ request server -m r2 : r2(a)
  }
  Transition t0
/*2*/ whenReply rr2 -> handleReply

  State handleReply{
    printCurrentMessage color magenta
    delay 1000
    println("$name dispatch cmd")  color magenta
/*3*/ forward server -m cmd : cmd(stop)
    delay 1500
    [# System.exit(0) #]
  }
}
```

3. Invio del dispatch **cmd** al *server*
4. Termina il sistema (per evitare che rimanga attivo)

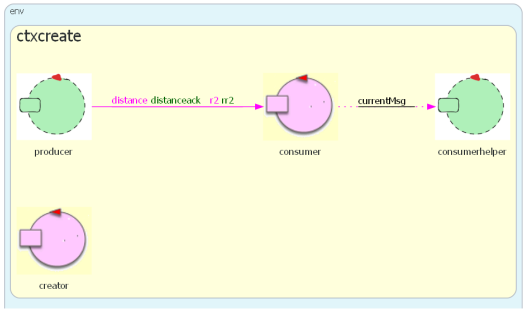
Esempio Creazione

democreate.qak

democreate.qak code

- Il *creator* crea due *producer*
- Ogni *producer* invia una request **distance** e una request **r2** al *consumer*
- Il *consumer* gestisce direttamente la request **distance**, ma delega la gestione della request **r2** al *consumerhelper* creato dinamicamente, usando la primitiva qak *delegateCurrentMsgTo*
- Il *consumerhelper* gestisce la request **r2** , invia la risposta al chiamante e termina

```
System  qakdemo24

Request  distance : distance(D)
Reply distanceack : ack(D) for distance

Request r2 : r2(X)
Reply rr2  : rr2(X) for r2

Context ctxcreate ip [host="localhost" port=8045]

QActor creator context ctxcreate{ ... }
QActor producer context ctxcreate
                          dynamicOnly{...}
QActor consumer context ctxcreate { ... }
QActor consumerhelper context ctxcreate
                          dynamicOnly{...}
```



---

creator

```
QActor creator context ctxcreate{
  State s0 initial{
    delay 500
/*1*/    create producer _"p1"
/*2*/    create producer _"p2"
    delay 1000
    [# System.exit(0) #]
  }
}
```

1. Crea un producer usando la primitiva *create*
2. Crea un altro producer usando la primitiva *create*

---

producer

producer

```
QActor producer context ctxcreate dynamicOnly {
  State s0 initial{
    println("$name STARTS") color blue
/*1*/request consumer -m distance : distance($MyName)
    delay 500
/*2*/request consumer -m r2:r2(10)
```

1. Invia la request **distance** al *consumer*
2. Invia la request **r2** al *consumer*
3. Gestisce le risposte

```
  }
  Transition t0
    whenReply distanceack -> handleAnswer
    whenReply rr2         -> handleAnswer

  State handleAnswer{
/*3*/ printCurrentMessage color blue
  }
  Transition t0
    whenReply distanceack -> handleAnswer
    whenReply rr2         -> handleAnswer
}
```

---

consumer

consumer

```
QActor consumer context ctxcreate {
  State s0 initial{
    println("$name STARTS") color green
  }
/*1*/Transition t0
      whenRequest distance -> handleRequest
      whenRequest r2       -> handleRequest

  State handleRequest{
    printCurrentMessage color green
    if [# currentMsg.msgId() == "r2" #]{
 /*2*/ delegateCurrentMsgTo consumerhelper
    }
    onMsg( distance : distance(D)){
      [# var D = payloadArg(0) #]
  /*3*/ replyTo distance with distanceack : ack( $payloadArg(0) )
    }
  }
  /*4*/Transition t0
       whenRequest distance -> handleRequest
       whenRequest r2       -> handleRequest
}
```

1. Attesa delle richieste **distance** e **r2**
2. Delega il messaggio corrente al *consumerhelper* se si tratta di una richiesta **r2**
3. Invia la risposta alla richiesta **distance**
4. Ccontinua ad attendere richieste

---

consumerhelper

consumerhelper

```
QActor consumerhelper context ctxcreate dynamicOnly {
  State s0 initial{
    println("$name STARTS") color green
  }
/*1*/Transition t0 whenRequest r2 -> handle_r2

  State handle_r2{
/*2*/ replyTo r2 with rr2: rr2(answer)
  }
  Transition t0 whenRequest r2 -> handle_r2

}
```

1. Attende la richiesta **r2** delegatagli dal *consumer*
2. Gestisce la richiesta e invia la risposta

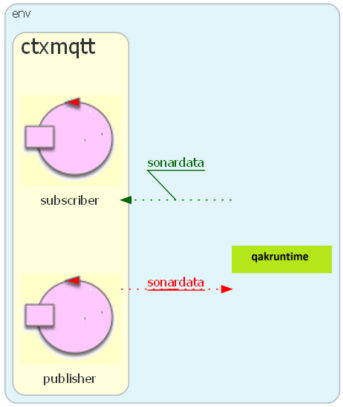
Esempio MQTT esplicito

demomqttexplicit.qak

demomqttexplicit.qak code

- L'actor *publisher(explicit)* si connette a un broker MQTT ed emette un evento sulla topic **sonardatatopic** usando *publish*
- L'actor *subscriber(explicit)* si connette allo stesso broker MQTT e fa una *subscribe* sulla topic **sonardatatopic** per ricevere l'evento

```
System qakdemo24
Event sonardata : sonardata(D)

Context ctxmqtt  ip [host="localhost" port=8920     ]

QActor publisher context ctxmqtt{ ... }
QActor subscriber context ctxmqtt{ ... }
```



---

publisher(explicit)

publisher

```
QActor publisher context ctxmqtt{

  State s0 initial{
/*1*/  connectToMqttBroker "tcp://broker.hivemq.com:1883"
/*2*/  publish "sonardatatopic" -m sonardata : sonardata(10)
  }
```

1. Si connette al broker
2. Pubblica un evento sulla topic **sonardatatopic**

---

subscriber(explicit)

subscriber

```
QActor subscriber context ctxmqtt{

  State s0 initial{
/*1*/  connectToMqttBroker "tcp://broker.hivemq.com:1883"
/*2*/  subscribe "sonardatatopic"
  }
      /*3*/Transition t0 whenEvent sonardata -> handlesonardata

  State handlesonardata{
    ...
  }
```

1. Si connette al broker
2. Si sottoscriver alla topic **sonardatatopic**
3. Attende l'evento

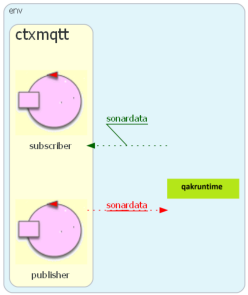
Esempio MQTT implicito

demomqttimplicit.qak

demomqttimplicit.qak code:

- La **dichierazione di usare un broker MQTT** avviene a livello di Sistema
- L'actor *publisher(implicit)* emette un evento usando *emit*
- L'actor *subscriber(implicit)* percepisce l'evento usando *whenEvent*

```
   System qakdemo24
/*1*/mqttBroker "broker.hivemq.com" : 1883 eventTopic "sonardatatopic"

   Event sonardata : sonardata(D)

   Context ctxmqtt  ip [host="localhost" port=8920     ]

   QActor publisher context ctxmqtt{ ... }
   QActor subscriber context ctxmqtt{ ... }
```



1. Dichierazione di usare un broker MQTT

---

publisher(implicit)

publisher

```
QActor publisher context ctxmqtt{

  State s0 initial{
/*1*/  emit sonardata : sonardata(10)
   }
```

1. Emette un evento

---

subscriber(implicit)

subscriber

```
QActor subscriber context ctxmqtt{

  State s0 initial{
    ...
  }
     /*1*/Transition t0 whenEvent sonardata -> handlesonardata

  State handlesonardata{
    ...
  }
```

1. Attende l'evento

Invio di dispatch via MQTT

Al momento, la capacità espressiva del linguaggio qak è limitata alla pubblicazione di eventi, ma è possibile inviare dispatch usando MQTT nella versione implicita, operarando a basso livello.

- La **dichierazione di usare un broker MQTT** avvienea livello di Sistema
- L'actor *demomqttsender.qak* crea un dispatch e lo pubblica sulla topic dichiarata a livello di sistema
- L'actor *demomqttreceiver.qak* riceve il dispatch

demomqttsender.qak

demomqttsender.qak code

```
    System qakdemo24
/*1*/mqttBroker "broker.hivemq.com" : 1883 eventTopic "sonardatatopic"

    Dispatch cmd : cmd(D)

    Context ctxmqttsender  ip [host="localhost" port=8920     ]

    QActor sender context ctxmqttsender{ ù
      delay 1000 //La connessione MQTT richiede tempo ...
      [#
/*2*/   val msg = MsgUtil.buildDispatch(name, "cmd", "cmd(30)", "receiver")
/*3*/   val topic = sysUtil.getMqttEventTopic()
/*4*/   mqtt.publish(topic, msg.toString())
      #]
    }
```

Operazioni di basso livello

1. Dichierazione di usare un broker MQTT
2. Creazione di un dispatch
3. Acquisizione della topic dichiarata a livello di sistema
4. Pubblicazione sulla topic dichiarata a livello di sistema

---

demomqttreceiver.qak

demomqttreceiver.qak code

```
    System qakdemo24
/*1*/mqttBroker "broker.hivemq.com" : 1883 eventTopic "sonardatatopic"
```

1. Dichierazione di usare un broker MQTT

```
    Dispatch cmd : cmd(D)

    Context ctxmqttreceiver  ip [host="localhost" port=8922  ]

    QActor receiver context ctxmqttreceiver{
      State s0 initial{
        ...
      }
/*2*/Transition t0 whenMsg cmd -> handlecmd

      State handlecmd{
/*3*/   onMsg( cmd : cmd(D)){
          println("$name| cmd ${payloadArg(0)}") color magenta
        }
      }
      Transition t0 whenMsg cmd -> handlecmd
    }
```

2. Attesa del dispatch
3. Gestione del dispatch

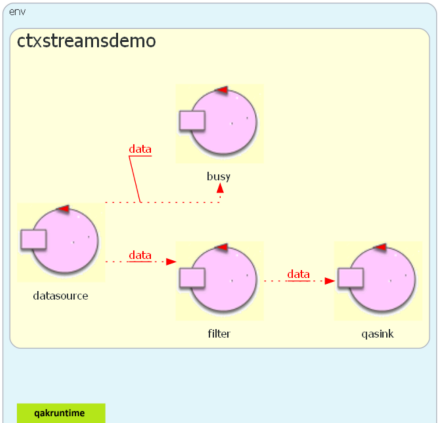
Esempio DataStreamer

demostreams.qak

demostreams.qak code

- Una sorgente *datasource* emette dati usando *emitlocalstream*
- Gli eventi sono trasmessi solo ai subscribers locali *busy* e *filter*
- L'actor *filter* (ri)emette solo i dati inferiori a un certo valore
- L'actor *qasink* è interessato agli eventi filtrati e non a quelli originali

```
 System qakdemo24
 Event  data : value(V)   "emesso da datasource "

 Context ctxmqtt ip [host="localhost" port=8045]

//Sorgente di eventi data
 QActor datasource context ctxmqtt{ ... }

//Propaga solo gli eventi data con valori inferiori a 40
 QActor filter context ctxmqtt{ ... }

//Interessato a data,  ma li considera solo dopo 3sec
 QActor busy context ctxmqtt{ ... }

//Interessato a eventi data opportunamente filtrati
 QActor qasink context ctxmqtt{ ... }
```



datasource

datasource

```
 QActor datasource context ctxmqtt{
  [# var N = 10 #]
  State s0 initial{
    delay 1000
  }
  Goto work

  State work{
    println("$name emitlocalstream $N") color green
  /*1*/ emitlocalstream  data : value($N)
    [# N = N + 10 #]
    delay 500
  }
  /*2*/ Goto work  if [# N < 80 #] else endOWork

  State endOWork{
    println("$name ENDS") color green
  /*3*/ [# System.exit(0) #]
  }

}
```

1. Usa la primsitiva *emitlocalstream* per emettere il dato corrente
2. Continua a emettere ogni **500msec** se il dato corrente non supera **80**
3. Termina il sistema

---

filter

filter

```
QActor filter context ctxmqtt{
[# var V = 0 #]

  State s0 initial{
/*1*/    subscribeTo datasource for data
  }
/*2*/Transition t0 whenEvent data -> handleData

  State handleData{
    //printCurrentMessage color magenta
    onMsg( data : value(V)){
      [#  V = payloadArg(0).toInt() #]
      println("$name handles $V") color magenta
      if [# V < 40  #] {
  /*3*/ emitlocalstream  data : value($V)
      }
    }
  }
  Transition t0 whenEvent data -> handleData
}
```

1. Si sottosrive a *datasource*
2. Attende un evento
3. Propaga l'evento se inferiore a **40**

---

qasink

qasink

```
QActor qasink context ctxmqtt{

  State s0 initial{
/*1*/  subscribeTo filter for data
  }
/*2*/ Transition t0 whenEvent data -> handleData

  State handleData{
/*3*/  println("$name handles ${currentMsg.msgContent()}")
  }
  Transition t0 whenEvent data -> handleData
}
```

1. Si sottosrive a *filter*
2. Attende un evento
3. Elabora l'evento

---

busy

busy

```
QActor busy context ctxmqtt{
  State s0 initial{
    println("$name STARTS") color cyan
/*1*/  subscribeTo datasource for data
  }
/*2*/Transition t0 whenTime 3000 -> work
```

1. Si sottosrive a *datasource*
2. Simula di essere occupato per **3sec**
3. Attende un evento generato da *data-source*
4. Elabora l'evento

```
  State work{
    println("$name PAY ATTENTION") color cyan
  }
/*3*/ Transition t0 whenEvent data -> handleData

  State handleData{
/*4*/  printCurrentMessage color cyan
  }
  Transition t0 whenEvent data -> handleData
}
```

Esempio Interruzione

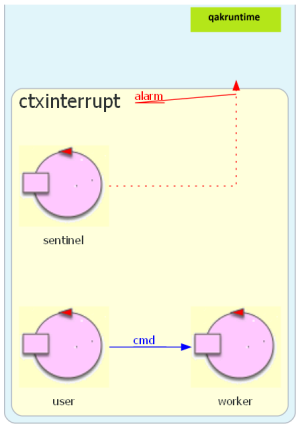
demointerrupt.qak

demointerrupt.qak code

whenInterrupt

- L'attore *worker* nello stato **work**, gestisce il dispatch **cmd** e l'evento **alarm**.
- L'evento **alarm** provoca il trasferimento allo stato **handlealarm** che termina le sue azioni con la primitiva *returnFromInterrupt* che restituisce il controllo allo stato **work** senza eseguirne le azioni, ma solo le transizioni. Nel caso di **alarm(tsunami)**. termina il sistema.
- L'attore *user* invia` dispatch **cmd** a *worker*.
- L'attore *sentinel* emette prina l'evento **alarm(fire)** e poi l'evento **alarm(tsunami)**.

```
System  qakdemo24

Dispatch cmd   : cmd(X)
Event    alarm : alarm(X)

Context ctxinterrupt ip [host="localhost" port=8045]

QActor worker context ctxinterrupt{ ... }
QActor user context ctxinterrupt{ ... }
QActor sentinel context ctxinterrupt{ ... }
```



worker

worker

```
QActor worker context ctxinterrupt{
  State s0 initial{
    println("$name STARTS") color blue
  }
  Goto work

  State work{
    println("$name WORKING ... ") color blue
    delay 1000
  }
  /*1*/Transition t0
    whenMsg cmd               -> handlecmd
    whenInterruptEvent  alarm -> handlealarm

  State handlecmd{
  /*2*/  printCurrentMessage color blue
  }
  Goto work

  State handlealarm{
    printCurrentMessage color black
  /*3*/  if[# currentMsg.msgContent() == "alarm(tsunami)" #]{
      println("$name WARNING tsunami ... ") color red
```

1. Attesa del dispatch **cmd** e dell'evento **alarm**.
2. Gestione di **cmd**.
3. Analisi del payload dell'evento **alarm**
4. Terminazione del sistema
5. Esecuzione di *returnFromInterrupt* che restituisce il controllo allo stato interrotto (**work**) senza eseguirne le azioni, ma solo le transizioni.

```
      delay 1000
  /*4*/ [# System.exit(0) #]
    }
    else{
  /*5*/ returnFromInterrupt
    }
  }
}
```

---

user

user

```
QActor user context ctxinterrupt{
  State s0 initial{
    delay 500
    println("$name forward  cmd(10) ") color cyan
/*1*/forward worker -m cmd : cmd(10)
    delay 500
    println("$name forward  cmd(20) ") color cyan
/*2*/forward worker -m cmd : cmd(20)
    println("$name ENDS") color cyan
  }
}
```

1. Invio del primo dispatch **cmd**
2. Invio del secondo dispatch **cmd**

---

sentinel

sentinel

```
QActor sentinel context ctxinterrupt{
  State s0 initial{
    delay 300
    println("$name emits alarm(fire)") color magenta
/*1*/ emit alarm : alarm(fire)
    delay 300
    println("$name emits alarm(tsunami)") color magenta
/*2*/ emit alarm : alarm(tsunami)
  }
}
```

1. Invio dell'evento *alarm* con payload **alarm(fire)**
2. Invio dell'evento *alarm* con payload **alarm(tsunami)**

Indice: *QakActors25Index*