```
-------------------------------------------------------------------------------
-- |                                                                         | --
-- | Progetto d'esame di "Linguaggi di Programmazione e Verifica del Software"   | --
-- | Sessione invernale a.a. 2022-2023                                       | --
-- | Studente: Petrelli Tommaso                                              | --
-- | Matricola: 305558                                                       | --
-- |                                                                         | --
-- | Modellare un'architettura Pub-Sub per l'interoperabilità tra le blockchain  | --
-- |                                                                         | --
-------------------------------------------------------------------------------



-- Il sistema che si vuole modellare si compone di un certo numero di blockchain.
-- I ruoli che un agente può coprire sono: publisher, subscriber e broker.
-- La topologia a cui si fa riferimento prevede:
--    - 1 publisher
--    - 1 broker
--    - 2 subscriber
-- Tutti e tre i ruoli che andremo a modellare rappresentano una rete blockchain.
-- I loro comportamenti verranno commentati nel dettaglio prima e durante la
-- definizione del modulo.

-- Nel modulo "main" andiamo a dichiarare tutti i moduli che abbiamo detto far
-- parte dell'architettura; quindi implementiamo la topologia descritta.

-- Per tutto il sistema si sceglie di trattare i topic ad un alto livello di astrazione.
-- Questi infatti verranno rappresentati semplicemente da un ID univoco per ogni topic.
-- Questa scelta viene fatta con l'idea di spostare il focus della modellazione sul
-- protocollo di comunicazione in sè e sulle sue caratteristiche. Come si vedrà,
-- infatti, anche le blockchain verranno trattate come delle macro-entità poiché
-- anche gli smart-contract sono stati implementati ad un alto livello di astrazione.

-- SCALABILITA' DELLA TOPOLOGIA.
-- Vediamo come risponde il sistema all'aumento del numero dei topic trattati.
--     - Immaginiamo che all'interno della topologia vengano trattati 2 topic.
--       Il numero degli stati raggiungibili è 2.66458e+006.
--     - Se aggiungiamo un topic il numero degli stati raggiungibili diventa 2.96658e+008.
--       Le dimensioni del modello diventano 100 volte maggiori.
-- Possiamo dire che le strutture dati utilizzate per la memorizzazione dei topic
-- non è molto efficiente da questo punto di vista. Tuttavia, per verificare le
-- proprietà del modello interessanti sono sufficienti 2 topic per cui è ragionevole
-- pensare che in questo caso la semplicità gestionale vince sull'efficienza.
--
-- Immaginiamo di aggiungere un subscriber. Con 2 topic e 3 subscriber il numero
-- di stati cresce a 1.10804e+008. Allora la scalabilità rispetto ai subscriber
-- (e analogamente rispetto ai publisher) è sostenuta un po' più facilmente se
-- confrontata con quella rispetto ai topic.

MODULE main

  VAR
    pub     :  publisher(broker, mutex.val);
    sub1    :  subscriber(1, broker, mutex.val);
```

```
sub2    :  subscriber(2, broker, mutex.val);
broker  :  broker(pub, sub1, sub2, mutex.val);
mutex   :  mutex(pub, sub1, sub2, broker);

-- sub3    :  subscriber(3, broker, mutex.val);
-- broker  :  broker(pub, sub1, sub2, sub3, mutex.val);
-- mutex   :  mutex(pub, sub1, sub2, sub3, broker);



--------------------------------------------------------------------------
-- |                  MODEL PROPERTY CHECKING                  |
--------------------------------------------------------------------------


-- PROPERTIES 1,2,3 ------------------------------------------------------
-- Nelle seguenti tre proprietà consecutive andremo a verificare se una volta
-- che una blockchain si è registrata al broker, questa non vi si disiscriva
-- in modo anomalo. In particolare:
--      1) verifico il comportamento del publisher
--      2) verifico il comportamento del subscriber 1
--      3) verifico il comportamento del subscriber 2
-- Mi aspetto che tutte le proprietà siano vere nel modello perché questo vuol
-- dire che una blockchain non si "sgancia" mai dal broker in modo involontario.
-- Invece di verificare una proprietà del tipo AFAG utilizziamo un quantificatore
-- esistenziale: EFAG. Questo perché una blockchain potrebbe non volersi mai
-- iscrivere alla rete. La scelta di registrarsi o meno alla rete è lasciata alle
-- blockchain, infatti, il mutex assegna il turno in modo non deterministico
-- e non ci sono regole di fairness. Allora il sistema modellato ammette futuri
-- in cui un publisher o un subscriber non voglia mai iscriversi alla rete.
CTLSPEC
EF(AG(pub.enrolled = TRUE))
-- RESULT: true

CTLSPEC
EF(AG(sub1.enrolled = TRUE))
-- RESULT: true

CTLSPEC
-- AF(AG(sub2.enrolled = TRUE))
EF(AG(sub2.enrolled = TRUE))
-- RESULT: true

-- Osserviamo però che avere un mutex non deterministico vuol dire che potrebbero
-- realizzarsi futuri in cui ad esempio il subscriber 1 prenda il turno solo una volta,
-- si registra, e poi rimane bloccato per tutto il resto del tempo non riuscendo
-- a fare richieste verso il broker. Questo lo possiamo verificare scrivendo
-- CTLSPEC
-- EF(AG(sub1.enrolled = TRUE -> !AF(sub1.state = subscribe)))
-- e il risultato sarebbe true. Allora, effettivamente esiste almeno un futuro
-- in cui il subscriber 1 è registrato ma non raggiunge mai lo stato di "subscribe".

-- Pensiamo però all'idea di come è stato progettato il mutex nel nostro caso.
-- Il turno viene ceduto da una blockchain quando si trova in uno stato di riposo;
-- non ci sono regole di scheduling. Per cui se andassimo ad implementare il sistema
```

-- che abbiamo modellato, il mutex non rappresenterà lo scheduler, bensì
-- la capacità del sistema a non entrare in una situazione di deadlock.

-- PROPERTY 4 ----------
-- Controllo che il publisher invii delle richieste di pubblicazione al broker
-- solo nei casi in cui sia registrato ad esso. Per come è stata scritta la
-- proprietà, mi aspetto che questa sia falsa. In tal caso, possiamo affermare
-- che un publisher raggiunga una configurazione che lo abilita ad avanzare
-- una richiesta verso il broker se e solo se questo risulta prima essere
-- registrato.
CTLSPEC
EF((pub.state = publish_new | pub.state = publish_update) & pub.enrolled = FALSE)
-- RESULT: false

-- PROPERTIES 5, 6 ----------
-- Controllo che i subscriber facciano delle richieste di sottoscrizione per
-- un topic verso il broker solo nei casi in cui siano registrati ad esso.
-- Per lo stesso motivo della PROPERTY 4, mi aspetto che PROPERTY 5 e
-- PROPERTY 6 siano false.
CTLSPEC
EF(sub1.state = subscribe & sub1.enrolled = FALSE)
-- RESULT: false

CTLSPEC
EF(sub2.state = subscribe & sub2.enrolled = FALSE)
-- RESULT: false

-- PROPERTIES 7,8 ----------
-- Il broker può inviare notifiche ai subscriber e qusti possono rispondere
-- accettando o meno la notifica. Per fare questo è essenziale che i subscriber
-- siano registrati al broker. Mi aspetto che le proprietà siano false, poiché
-- nel caso contrario vorrebbe il broker manda notifiche riguardanti i topic
-- anche a blockchain estranee al sistema e potenzialemente malevole.
CTLSPEC
EF((sub1.state = accept_update | sub1.state = refuse_update) & sub1.enrolled = FALSE)
-- RESULT: false

CTLSPEC
EF((sub2.state = accept_update | sub2.state = refuse_update) & sub2.enrolled = FALSE)
-- RESULT: false


```
-- PROPERTIES 9, 10, 11 ----------------------------------------------------
-- Si vogliono verificare alcune proprietà di fairness condizionata debole.
-- Nota che si parla di weak fairness perché le richieste di registrazione
-- vengono fatte con continuità, cioè la blockchain richiedente deve rimanere
-- in busy waiting.
-- In particolare, ci si vuole accertare che:
--     9) ogni qual volta un publisher fa una richiesta di enrolling
--        al broker, questa prima o poi verrà accolta da esso
-- 10/11) ogni qual volta un subscriber fa una richiesta di enrolling
--        al broker, questa prima o poi verrà accolta da esso
-- Mi aspetto che queste tre proprietà siano vere. In tal caso vorrebbe dire
-- che il broker è capace di gestire tutte le richieste provenienti, in questo
-- caso, dal publisher, dal subscriber 1 e dal subscriber 2.
LTLSPEC
F(G(pub.state = enroll)) -> G(F((broker.state = acknowledge | broker.state = refuse) & mutex.val = p))
-- RESULT: true

LTLSPEC
F(G(sub1.state = enroll)) -> G(F((broker.state = acknowledge | broker.state = refuse) & mutex.val = s1))
-- RESULT: true

LTLSPEC
F(G(sub2.state = enroll)) -> G(F((broker.state = acknowledge | broker.state = refuse) & mutex.val = s2))
-- RESULT: true


-- PROPERTIES 12, 13 ----------------------------------------------------------
-- Verifichiamo ancora proprietà di waek fairness. Infatti, anche in questo
-- caso è vero che un publisher fa le sue richieste con continuità finché
-- non viene soddisfatto.
-- Questa volta si vuole verificare che ogni qual volta un publisher
-- registrato fa:
--    12) richiesta di pubblicare un nuovo topic
--    13) richiesta di aggiornare un topic già esistente
-- il broker la gestisce accettandola o rifiutandola. Mi aspetto che queste
-- proprietà siano vere, e quindi che tutte le richieste vengano gestite.
LTLSPEC
F(G(pub.state = publish_new)) -> G(F((broker.state = accept_new | broker.state = refuse) & mutex.val = p))
-- RESULT: true

LTLSPEC
F(G(pub.state = publish_update)) -> G(F((broker.state = accept_update | broker.state = refuse) & mutex.val = p))
-- RESULT: true


-- PROPERTIES 14, 15 ----------------------------------------------------------
-- In modo completamente analogo alle PROPERTIES 12 e 13, si vuole verificare
-- se il broker soddisfi i subscriber ogni qual volta questi chiedono di
-- iscriversi ad un topic. Anche in questo caso mi aspetto che le
-- proprietà siano vere.
LTLSPEC
G(F(sub1.state = subscribe)) -> G(F((broker.state = accept_subscription_sub1 | broker.state = refuse) & mutex.val
= s1))
```

-- RESULT: true

LTLSPEC
F(G(sub2.state = subscribe)) -> G(F((broker.state = accept_subscription_sub2 | broker.state = refuse) & mutex.val = s2))
-- RESULT: true

-- PROPERTIES 16, 17 ----------
-- Infine, verifichiamo alcune proprietà di weak fairness che riguardano
-- il comportamento del broker. Quando il broker aggiunge un nuovo topic o
-- ne aggiorna uno già esistente, allora manda una notifica a tutti i
-- subscriber interessati al quel topic. Si vuole provare che tutte le sue
-- richieste di notifica vengono gestite dai subscriber e quindi mi aspetto
-- che entrambe le proprietà siano vere.
LTLSPEC
F(G(broker.state = notify_sub1)) -> G(F((sub1.state = accept_update | sub1.state = refuse_update) & mutex.val = b))
-- RESULT: true

LTLSPEC
F(G(broker.state = notify_sub2)) -> G(F((sub2.state = accept_update | sub2.state = refuse_update) & mutex.val = b))
-- RESULT: true

-- PROPERTY 18 ----------
-- Si vuole simulare una situazione che coinvolga tutti i componenti del sistema.
-- Lo scenario che andiamo a prendere come esempio è il seguente: un publisher
-- richiede di pubblicare un aggiornamento di un topic; il broker accetta
-- la richiesta e aggiorna il topic corretto (si assume che questo esista
-- all'interno del broker); il broker notifica ai subscriber che il topic
-- è stato aggiornato; i subscriber ricevono la notifica e la gestiscono.
-- L'obiettivo è quello di verificare se il broker sia sempre capace di rispondere
-- al publisher e di notificare i cambiamenti agli opportuni subscriber.
-- Per tale esempio, assumiamo che il topic da aggiornare abbiamo ID = 1 e
-- che solo il subscriber 1 vi sia iscritto.
-- Mi aspetto che la proprietà sia vera, ossia mi aspetto che un publisher
-- riesce sempre a pubblicare, che il broker notifica sempre i subscriber
-- giusti e che i subscriber ricevano solo le notifiche che gli interessano.

-- Considero solo i cammini in cui il subscriber 1 si iscrive al topic 1
-- JUSTICE (sub1.topics[1] = TRUE);
-- Dato che abbiamo detto che solo il subscriber 1 si iscrive al topic 1,
-- allora assumo che il subscriber 2 non è interessato a tale topic
-- JUSTICE (sub2.topics[1] = FALSE);

CTLSPEC
AG((pub.state = publish_update & pub.topic = 1) ->

```
  AF(
    AG( (broker.state = notify_sub1 & broker.topic_to_notify = 1) ->
      AF(
        sub1.state = accept_update
      )
      & !EF(broker.state = notify_sub2 & broker.topic_to_notify = 1)
    )
  )
)
-- RESULT: true (si deve usare JUSTICE)
-- La stessa proprietà è stata verificata anche per il topic 2 ed
-- è risultata essere sempre vera.
-- Possiamo dire che il sistema è affidabile.



------------------------------------------------------------------------
------------------------------------------------------------------------



-- PROPERTIES 19, 20 ---------------------------------------------------
-- Si vogliono verificare due proprietà di liveness del sistema. Le proprietà
-- che vengono descritte successivamente servono per provare che un broker
-- invia le notifiche solo ai subscriber iscritti al topic che ha subito
-- modifiche.
-- Immaginiamo di dover notificare il topic con ID = 2.
--    19) subscriber 1 non è iscritto al topic 2 e quindi il broker deve
--        saltare lo stato "notify_sub1" e raggiungere subito "notify2".
--    20) subscriber 2 non è iscritto al topic 2 e quindi il broker deve
--        saltare lo stato "notify_sub2" e raggiungere subito "sleep".
-- Mi aspetto che entrambe le proprietà siano vere, cioè mi aspetto sia vero
-- che ogni qual volta il broker manda delle notifiche si accorge sempre
-- di chi sono i subscriber giusti e quindi salta quelli non iscritti.
CTLSPEC
AG(broker.state = notify & broker.topic_to_notify = 2 & sub1.topics[2] = FALSE -> AF(broker.state = notify2))
-- RESULT: true

CTLSPEC
AG(broker.state = notify & broker.topic_to_notify = 2 & sub2.topics[2] = FALSE -> AF(broker.state = sleep))
-- RESULT: true



------------------------------------------------------------------------
------------------------------------------------------------------------



-- PROPERTY 21 ---------------------------------------------------------
-- Questa volta si vuole effettuare una verifica che testa la sicurezza del
-- del sistema. Si consideri uno scenario in cui un publisher malevolo vuole
-- inviare richieste di pubblicazione al broker anche se non è registrato.
-- Come risultato della proprietà mi aspetto vero, ossia mi aspetto che il
-- broker rifiuti tutte le richieste provenienti da entità esterne al sistema.
```

-- Per simulare un publisher malevolo si deve saltare la fase di enrolling,
-- passando direttamente a quella di request.
CTLSPEC
AG((pub.state = publish_new) -> AF(broker.state = refuse & mutex.val = p))
-- RESULT: true

-- La proprietà è stata testata considerando una richiesta di pubblicazione
-- di un nuovo topic. Non verifichiamo la proprietà considerando una richiesta
-- di aggiornamento di un topic poiché, per come è stato progettato il publisher,
-- prima di un aggiornamento devo aver creato il topic. Non essendo mai riuscito
-- a pubblicare un nuovo topic di conseguenza non riesco ad aggiornarlo.
-- Lo si può verificare con la seguente proprietà
-- CTLSPEC
-- !EF(pub.state = publish_update)
-- RESULT: true

-- PROPERTY 22 --
-- In modo analogo alla PROPERTY 21, si vuole verificare la stessa proprietà
-- di sicurezza dal punto di vista dei subscriber.
-- Per simulare un subscriber malevolo si deve saltare la fase di enrolling,
-- passando direttamente a quella di request.
-- Ci si aspetta che la proprietà sia vera. Quindi mi aspetto che, se un
-- subscriber non registrato al sistema faccia delle richieste di iscrizione
-- ad uno dei topic contenuti nel broker, queste verranno tutte rifiutate.
CTLSPEC
AG((sub1.state = subscribe) -> AF(broker.state = refuse & mutex.val = s1))
-- RESULT: true

-- In modo analogo si può provare a verificare la stessa proprietà prendendo
-- come riferimento il subscriber 2.
-- CTLSPEC
-- AG((sub2.state = subscribe) -> AF(broker.state = refuse & mutex.val = s2))
-- RESULT: true

-- PROPERTY 23 --
-- Si è già parlato di che cosa rappresenta in questo sistema il mutex.
-- Per verificare se sono presenti situazioni di deadlock si può andare a vedere
-- se esistono configurazioni in cui tutte le entità che partecipano alla topologia
-- si trovano in uno stato neutro. Se così fosse, vuol dire che tutti hanno ceduto
-- il loro turno e nessuno se lo è preso. Per come è stato ideato il mutex,
-- ci deve sempre essere qualcuno che subentra alla blockchain che ha appena
-- terminato il suo turno. Per come è formulata la specifica CTL, mi aspetto
-- che la proprietà sia vera.
CTLSPEC
!EF (EG (pub.state = write & sub1.state = read & sub2.state = read & broker.state = sleep));
-- RESULT: true

```
-- PROPERTY 24 ------------------------------------------------------------
-- Si vuole verificare un caso di possibile starvation. La proprietà vuole
-- testare se esiste una configurazione in tutto il modello in cui una blockchain
-- (diversa dal broker) rimane sempre in attesa di essere registrata.
-- Mi aspetto che questa proprietà sia vera, cioè che il broker sia sempre
-- reattivo rispetto alle richieste di registrazione provenienti dall'esterno.
CTLSPEC
!EF ((EG pub.state = enroll) | (EG sub1.state = enroll) | (EG sub2.state = enroll));
-- RESULT: true


-- PROPERTY 25 ------------------------------------------------------------
-- In modo complementare alla PROPERTY 24, si vuol verificare se casi di
-- starvation possono verificarsi in momenti in cui le blockchain sono già
-- registrate al broker. Quindi, vogliamo vedere se un publisher rimane
-- bloccato in attesa che il broker gestisca la sua richiesta di pubblicazione,
-- oppure se un subscriber rimane bloccato in attesa che il broker gestisca
-- la sua richiesta di iscrizione ad un topic. Si vuole anche verificare se
-- il broker rimane bloccato in attesa che un subscriber gestisca una sua
-- richiesta di notifica. Allora mi aspetto che la proprietà sia vera.
CTLSPEC
!EF ((EG pub.state = publish_new) | (EG pub.state = publish_new) |
    (EG sub1.state = subscribe)  | (EG sub2.state = subscribe)  |
    (EG broker.state = notify));
-- RESULT: true


-- PROPERTY 26 ------------------------------------------------------------
-- Per il non determinismo del mutex, è certamente possibile che una blockchain
-- rimanga da un certo punto in avanti costantemente in stato di ready ad esempio.
-- Questo può accadere e viene interpretato come se (in un sistema reale)
-- una blockchain non volesse più interagire con il broker.
-- Una situazione che possiamo considerare di starvation è quando ad esempio
-- un subscriber o un publisher invia una richiesta di pubblicazione o di
-- iscrizione e questa non viene mai soddisfatta (PROPERTY 25).
-- Un altra configurazione che vogliamo evitare e verrebbe interpretato come
-- un malfunzionamento del sistema è invece la seguente situazione di deadlock:
-- tutte le blockchain sono contemporaneamente in uno stato di "ready" e il broker
-- si trova in uno stato neutro (non sta gestendo richieste).
-- Avere tutte le blockchain nello stato di "ready" vuol dire che tutte hanno
-- lasciato il loro turno ma nessuna lo ha voluto prendere.
-- La situazione in cui tutte le blockchain non vogliano interagire con il broker
-- non è stata prevista nel modello per cui non la si può interpretare come
-- una situazione normale, ma come una situazione di deadlock. Se PROPERTY 25
-- risulta essere vera vuol dire che non c'è deadlock.
CTLSPEC
!EF (EG (pub.state = ready & sub1.state = ready & sub2.state = ready & broker.state = sleep));
-- RESULT: true


-- NOTA -------------------------------------------------------------------
-- Esiste la possibilità di considerare solo quei cammini in cui tutte
-- le blockchain partecipano, cioè tutte le blockchain prenderanno il proprio
```

```
   -- turno. Questo è possibile forzando il mutex ad essere equo.
   -- JUSTICE (mutex.val = p);
   -- JUSTICE (mutex.val = s1);
   -- JUSTICE (mutex.val = s2);
   -- JUSTICE (mutex.val = b);




--------------------------
-- | MODULO   *MUTEX*    |
--------------------------

-- Il modulo "mutex" definisce le regole con le quali si decide chi sarà la
-- prossima blockchain a prendere il turno in modo non deterministico.
-- Per questo progetto, la scelta non deterministica viene utilizzata per
-- rappresentare il fatto che una blockchain semplicemente cede il suo posto
-- senza preoccuparsi di chi sarà il suo successore o se effettivamente ce ne sarà uno.
-- In realtà, per fornire maggiore equità al sistema, si adottano due
-- meccanismi:
--    1. Quando una blockchain lascia il posto, questa non potrà riprenderselo subito.
--    2. Quando il broker prende il turno vuol dire che deve notificare un subscriber.
--       Questo vuol dire che ha appena gestito una richiesta da parte di un publisher,
--       e quindi un publisher ha appena avuto il suo turno. Per questa ragione,
--       quando il broker cede il suo turno, solo i subscriber potranno raccoglierlo
--       in quanto si presume che un publisher abbia avuto il suo poco prima.
-- Infine, notiamo che il broker è l'unica blockchain a non prendere il turno
-- non deterministicamente. Questo deriva da una scelta progettuale: le richieste
-- fatte dal broker verso l'esterno sono prioritarie, e quindi, non appena opportuno,
-- il broker otterrà il turno per notificare cambiamenti dei topic ai subscriber.

-- MODULE mutex(pub, sub1, sub2, sub3, broker)
MODULE mutex(pub, sub1, sub2, broker)

 VAR
   -- p: il turno è del              -- s1: il turno è del subscriber 1
   -- s2: il turno è del subscriber   -- b: il turno è del broker
   val: {p, s1, s2, b};
   -- val: {p, s1, s2, s3, b};

 ASSIGN
   -- Il publisher è il primo ad interagire con il broker. Può avere senso
   -- poiché è lui che crea i topic, ossia gli oggetti gestiti dal sistema.
   init(val) := p;

   next(val) :=
     case
       (val = p) & (pub.state = write) & (broker.state = sleep) : {s1, s2};
       (val = p) & (broker.state = notify) : b;
       (val = b) & (broker.state = sleep) : {s1, s2};
       (val = s1) & (sub1.state = read) & (broker.state = sleep) : {p, s2};
       (val = s2) & (sub2.state = read) & (broker.state = sleep) : {p, s1};

       -- (val = p) & (pub.state = write) & (broker.state = sleep) : {s1, s2, s3};
       -- (val = b) & (broker.state = sleep) : {s1, s2, s3};
```

```
      -- (val = s1) & (sub1.state = read) & (broker.state = sleep) : {p, s2, s3};
      -- (val = s2) & (sub2.state = read) & (broker.state = sleep) : {p, s1, s3};
      -- (val = s3) & (sub3.state = read) & (broker.state = sleep) : {p, s1, s2};

      TRUE : val;
    esac;



------------------------------
-- | MODULO   *PUBLISHER*     |
------------------------------

-- Questo modulo modella il comportamento di una blockchain che si vuole
-- registrare ad un broker come publisher. La prima cosa che un publisher vuole
-- fare è registrarsi al broker. A questo proposito, il publisher si accorge di
-- non essere registrato al broker guardando la variabile "enrolled". A questo
-- punto, inoltra una richiesta al broker e attende la risposta mettendosi in uno
-- stato di "enroll". Se la registrazione non va a buon fine, il publisher lascia
-- il suo turno, e riproverà a registrarsi alla prossima occasione.
-- Dopo che il publisher si è registrato, questo può inviare al broker due tipi
-- di richieste: pubblicare un nuovo topic ("publish_new") o pubblicare un
-- aggiornamento su un topic già esistente ("publish_update").
-- Nella restate parte dei momenti, il publisher si metterà a scrivere nuovi topic
-- ("write") lasciando libero il broker.

-- PARAMETRI: * broker : Blockchain del sistema che gestisce la interoperabilità tra subscriber e publisher.
--            * turn   : Indica di chi è il turno.

MODULE publisher(broker, turn)

  VAR
    -- start          : Stato di partenza.
    -- enroll         : Chidede di essere registrato nella rete del broker.
    -- ready          : Stato in cui il publisher si troverà nel momento in cui acquisirà nuovamente il turno.
    --                  Solo a partire da questo stato puà avanzare richieste di pubblicazione.
    -- publish_new    : Inoltra la richiesta di pubblicare un nuovo topic al broker.
    -- publish_update : Inoltra la richiesta di pubblicare un aggiornamento di un topic esistente al broker.
    -- write          : Scrive nuovi topic o aggiorna topic esistenti (stato neutro in cui non fa richieste e lascia il turno).
    state: {start, enroll, ready, publish_new, publish_update, write};

    -- Deve sempre essere possibile capire se il publisher è registrato (TRUE) o meno (FALSE) al broker.
    enrolled: boolean;

    -- Rappresenta il topic corrente su cui sta lavorando il publisher
    -- topic: {1, 2, 3, 4};
    -- topic: {1, 2, 3};
    topic: {1, 2};

    -- Una delle caratteristiche delle blockchain è che queste sono indipendenti
    -- e lavorano fortemente con i dati in loro possesso. Per mantenere questa
    -- qualità, il publisher andrà a gestire solo i suoi dati.
    -- A tal proposito si definisce una struttura dati dalla quale il publisher
    -- può capire se ha già pubblicato un certo topic oppure no.
```

```
  -- TRUE: il publisher ha già creato il topic i-esimo.
  -- FALSE: il topic i-esimo deve essere ancora creato.
  -- topics: array 1..4 of boolean;
  -- topics: array 1..3 of boolean;
  topics: array 1..2 of boolean;

ASSIGN
  init(state) := start;

  init(enrolled) := FALSE;

  -- init(topic) := {1, 2, 3, 4};
  -- init(topic) := {1, 2, 3};
  init(topic) := {1, 2};

  -- Inizialmente i topic saranno tutti da creare, allora si impostano tutte le
  -- celle della struttura "topics" a "FALSE".
  init(topics[1]) := FALSE;
  init(topics[2]) := FALSE;
  -- init(topics[3]) := FALSE;
  -- init(topics[4]) := FALSE;

  next(state) :=
    case
      -- Quando il publisher prende il suo turno può avanzare dallo stato di
      -- "start" allo stato di "enroll" invocando il broker. Il publisher
      -- attenderà nello stato di "enroll" finché il broker non avrà risposto.
      (turn = p) & (state = start) & (enrolled = FALSE) : enroll;            -- 'a'

      -- Il broker accetta il publisher e quindi quest'ultimo raggiunge uno stato
      -- dal quale potrà inoltrare richieste di pubblicazione al broker.
      (turn = p) & (state = enroll) & (broker.state = acknowledge) : ready;  -- 'b'

      -- Il publisher lascia il suo turno se il broker non lo accetta.
      (turn = p) & (state = enroll) & (broker.state = refuse) : write;       -- 'c'

      -- La regola seguente è essenziale affinché il publisher non monopolizzi
      -- il sistema. Immaginiamo una situazione in cui il publisher venga
      -- venga sempre rifiutato dal broker. Se tornasse subito nello stato di
      -- "enroll" si verrebbe a creare un loop "enroll-refuse" con il broker
      -- bloccando il sistema. In questo modo invece, il publisher cede il suo
      -- posto e riproverà a registrarsi non appena riuscirà a riacquisirlo.
      (turn = p) & (state = write) & (enrolled = FALSE) : start;             -- 'd'

      -- Questa regola va abilitata solo quando si vuole verificare la PROPERTY 21.
      -- Allo stesso tempo vanno disabilitate le regole 'a', 'b', 'c', 'd'.
      -- (turn = p) & (state = start) & (enrolled = FALSE) : write;

      -- Raggiungere lo stato di "ready" significa essere registrati al broker.
      -- Da qui il publisher può rimanere netruo, oppure può prendere il turno
      -- (se disponibile) per avanzare richieste al broker.
      (turn = p) & (state = write) : ready;

      -- Se già registrato, ogni volta che il publisher riprende il turno si
      -- trova nello stato di "ready". A questo punto è autorizzato a comunicare
```

```
        -- con il broker. Si fa notare come il turno viene mantenuto dal publisher
        -- per tutta la durata della comunicazione con il broker. L'idea è quella
        -- di instaurare una comunicazione diretta "pub-broker" chiusa, cioè
        -- il broker risulterà impegnato finché non avrà gestito la richiesta
        -- del publisher.
        (turn = p) & (state = ready) & (topics[topic] = FALSE): publish_new;
        (turn = p) & (state = ready) & (topics[topic] = TRUE): publish_update;

        -- Una volta che il publisher effettua la sua richiesta, attende la risposta
        -- del broker prima di tornare nello stato di "write" lasciando il turno
        -- ad un eventuale successore.
        -- Si fa notare che il cambiamento di stato del publisher non dipende dalla
        -- risposta del broker poiché in ogni caso andrebbe a finire in "write.
        -- Immaginiamo uno scenario in cui il broker rifiuta la pubblicazione di
        -- un publisher. L'idea dietro a progettuale è che il publisher si rimette
        -- a modificare il topic che aveva provato a ripubblicare e poi quando
        -- riotterrà il turno proverà a ripubblicare qualcos altro. Chiedere di
        -- pubblicare un topic finché il broker non accetta esporrebbe il sistema
        -- ad un rischio di starvation: il sistema entra in un loop "publish-refuse"
        -- e il turno non viene mai ceduto.
        (turn = p) & ((state = publish_new) | (state = publish_update)) & (broker.state = accept_new | broker.state = ac
cept_update) : write;
        (turn = p) & ((state = publish_new) | (state = publish_update)) & (broker.state = refuse) : write;

        TRUE : state;
      esac;

   next(enrolled) :=
    case
      (turn = p) & (state = enroll) & (broker.state = acknowledge) : TRUE;
      TRUE : enrolled;
    esac;


   -- Ogni volta che il publisher va nello stato di "write", in modo non deterministico
   -- "sceglie il nome" del nuovo topic che vuole pubblicare o aggiornare.
   next(topic) :=
    case
      -- (turn = p) & (state = write) : {1, 2, 3, 4};
      -- (turn = p) & (state = write) : {1, 2, 3};
      (turn = p) & (state = write) : {1, 2};
      TRUE : topic;
    esac;

   -- Si imposta la cella i-esima a "TRUE" quando il publisher pubblica un nuovo topic con ID = i.
   -- Prima di impostare a "TRUE" la cella, devo aspettare la risposta del topic.
   next(topics[1]) :=
    case
      (topic = 1) & (broker.state = accept_new) : TRUE;
      TRUE : topics[1];
    esac;
   next(topics[2]) :=
    case
      (topic = 2) & (broker.state = accept_new) : TRUE;
      TRUE : topics[2];
```

```
    esac;
  -- next(topics[3]) :=
  --   case
  --     (topic = 3) & (broker.state = accept_new) : TRUE;
  --     TRUE : topics[3];
  --   esac;
  -- next(topics[4]) :=
  --   case
  --     (topic = 4) & (broker.state = accept_new) : TRUE;
  --     TRUE : topics[4];
  --   esac;



------------------------------
-- | MODULO   *SUBSCRIBER*    |
------------------------------

-- Questo modulo modella il comportamento di una blockchain che si vuole
-- registrare ad un broker come subscriber. Come prima cosa un subscriber si deve
-- sempre registrare al broker. Un subscriber si accorge di essere registrato o
-- meno al broker guardando il valore della variabile "enrolled". Se la
-- registrazione non va a buon fine, il subscriber lascia il suo turno, e riproverà
-- a registrarsi alla prossima occasione. Solo dopo esserersi registrato può
-- interagire col sistema e quindi inoltrare richieste al broker. Un subscriber
-- è interessato a comunicare con il broker tutte quelle volte che gli interessa
-- un certo topic che ancora non ha. Un subscriber potrà inoltrare una richista
-- di iscrizione ad un certo topic "n" mettendosi nello stato "subscribe" e
-- impostando la variabile "topic_to_subscribe = n".
-- Un subscriber deve anche gestire le notifiche che gli arrivano da broker. Nel
-- momento in cui il broker manda la sua notifica al subscriber, allora quest
-- ultimo decide se accettare oppure no. Immaginiamo che il broker notifichi un
-- cambiamento riguardando un topic "k". Per scegliere, il subscriber che riceve
-- la notifica si basa su una struttura dati, "topics", da cui legge se è iscritto
-- o meno al topic "k". Il subscriber accetta solo se si accorge di essere iscritto.
-- Nella restate parte dei momenti, un subscriber si metterà a leggere i suoi topic
-- ("read") senza comunicare con altre blockchain.

-- PARAMETRI: * id     : Nome identificativo univoco di un subscriber.
--            * broker : Blockchain del sistema che gestisce la interoperabilità tra subscriber e publisher.
--            * turn   : Indica di chi è il turno.

MODULE subscriber(id, broker, turn)

  VAR
    -- start          : Stato di partenza.
    -- enroll         : Chidede di essere registrato nella rete del broker.
    -- ready          : Stato in cui il subscriber si troverà nel momento in cui acquisirà nuovamente il turno.
    --                  Solo a partire da questo stato puà avanzare richieste di iscrizione.
    -- subscribe      : Inoltra al broker una richiesta di iscrizione ad un topic.
    -- accept_update  : Accetta una notifica di aggiornamento proveniente dal broker.
    -- refuse_update  : Accetta una notifica di aggiornamento proveniente dal broker.
    -- read           : Legge i topic a cui è iscritto (stato neutro in cui non fa richieste e lascia il turno).
    state: {start, enroll, ready, subscribe, accept_update, refuse_update, read};
```

```
  -- Deve sempre essere possibile capire se il subscriber è registrato (TRUE) o meno (FALSE) al broker.
  enrolled: boolean;

  -- Rappresenta il topic corrente che il subscriber vorrebbe leggere. Se non
  -- vi è già, rappresenta anche il topic a cui il subscriber vorrebbe iscriversi.
  -- topic: {1, 2, 3, 4};
  -- topic: {1, 2, 3};
  topic: {1, 2};

  -- Rappresenta il topic al quale il subscriber vorrebbe iscriversi. Il broker
  -- si baserà su questa variabile per capire a che topic si vuole iscrivere
  -- il subscriber. Il valore "0" indica che il subscriber non vuole iscriversi
  -- a nessun topic per il momento.
  -- topic_to_subscribe: {0, 1, 2, 3, 4};
  -- topic_to_subscribe: {0, 1, 2, 3};
  topic_to_subscribe: {0, 1, 2};

  -- Con "topics" il subscriber tiene traccia dei topic a cui è iscritto.
  -- Analogamente al publisher, anche il subscriber vuole mantere una indipendenza
  -- per quanto riguarda la gestione delle informazioni che gli servono.
  -- TRUE: il subscriber è iscritto al topic i-esimo.
  -- FALSE: il subscriber è non iscritto al topic i-esimo.
  -- Da notare che l'array parte dall'indice "0". Questo viene fatto solo per
  -- gestire gli accessi con indici pari a "0" a tale struttura dati. Senza questo
  -- accorgimento non si riesce a compilare il file sorgente.
  --topics: array 0..4 of boolean;
  -- topics: array 0..3 of boolean;
  topics: array 0..2 of boolean;

  -- Il subscriber verifica se è il suo turno oppure no. Questo è necessario
  -- poiché sono stati previsti più subscriber nella topologia.
  DEFINE my_turn := ((id = 1) & (turn = s1) ? TRUE : ((id = 2) & (turn = s2) ? TRUE : FALSE));
  -- DEFINE my_turn := ((id = 1) & (turn = s1) ? TRUE : ((id = 2) & (turn = s2) ? TRUE : ((id = 3) & (turn = s3) ?
TRUE : FALSE)));


 ASSIGN
  init(state) := start;

  init(enrolled) := FALSE;

  -- init(topic) := {1, 2, 3, 4};
  -- init(topic) := {1, 2, 3};
  init(topic) := {1, 2};

  -- Inizialmente il subscriber non sa a che topic vuole iscriversi.
  init(topic_to_subscribe) := 0;

  -- Inizialmente un subscriber non sarà iscritto a nessun topic e quindi si
  -- impostano tutte le celle della struttura "topics" a "FALSE".
  init(topics[0]) := FALSE;
  init(topics[1]) := FALSE;
  init(topics[2]) := FALSE;
  -- init(topics[3]) := FALSE;
  -- init(topics[4]) := FALSE;
```

```
next(state) :=
  case
    --  Quando il subscriber prende il turno e si trova nello stato "start"
    -- vuol dire che non è ancora capace di fare richieste. Allora passa
    -- allo stato "enroll".
    (my_turn = TRUE) & (state = start) & (enrolled = FALSE) : enroll;            -- 'a'

    -- Se il broker accetta il subscriber allora può passare nello stato di
    -- "ready" dal quale può partire per inoltrare le sue richieste.
    (my_turn = TRUE) & (state = enroll) & (broker.state = acknowledge) : ready;  -- 'b'

    -- Se il broker non accetta la richiesta di registrazione del subscriber
    -- allora quest'ultimo lascerà il turno.
    (my_turn = TRUE) & (state = enroll) & (broker.state = refuse) : read;        -- 'c'

    -- In seguito ad un rifiuto della richiesta di registrazione, il subscriber
    -- passerà allo stato di "read". Dopodiché non essendo riuscito a registrasi
    -- torna allo stato iniziale.
    -- Anche in questo caso risulta ottimo passare per lo stato di "read" prima
    -- di ritentare la registrazione al broker. Infatti, in questo modo si evita
    -- di far cadere il sistema in un loop "enroll-refuse" tra il subscriber
    -- e il broker.
    (my_turn = TRUE) & (state = read) & (enrolled = FALSE) : start;              -- 'd'

    -- Questa regola va abilitata solo quando si vuole verificare la PROPERTY 22
    -- Allo stesso tempo vanno disabilitate le regole 'a', 'b', 'c', 'd'
    -- (my_turn = TRUE) & (state = start) & (enrolled = FALSE) : read;

    -- Se il subscriber raggiunge lo stato di "ready" significa che si è
    -- registrato al broker con successo. A partire da questo stato il
    -- subscriber può rimanere netruo, oppure può prendere il turno
    -- (se disponibile) per avanzare richieste al broker. Quando il subscriber
    -- riprenderà il suo turno si troverà nello stato di "ready".
    (my_turn = TRUE) & (state = read) : ready;

    -- Se il subscriber ha ottenuto il turno ed è registrato al broker, allora
    -- ha la possibilità di comunicare con il broker. Ci immaginiamo un
    -- subscriber che si trova in una fase in cui legge un certo topic. Nel
    -- momento in cui il subscriber si accorge che al topic che vorrebbe
    -- leggere non vi è registrato, manda una richista di iscrizione al broker.
    -- In particolare, per accorgersi se un subscriber è iscritto al topic
    -- "k" va a leggere la cella "topics[k]". Se la cella contiene un valore
    -- "FALSE" allora il subscriber manda la notifica.
    (my_turn = TRUE) & (state = ready) & (topics[topic] = FALSE) : subscribe;

    -- Se il valore nella cella è "TRUE" vuol dire che il subscriber è già
    -- iscritto al topic allora se lo legge e cede il turno ad un'altra
    -- eventuale blockchain.
    (my_turn = TRUE) & (state = ready) & (topics[topic] = TRUE) : read;

    -- Il subscriber attende la risposta del broker prima di cambiare stato.
    -- La scelta dello stato di arrivo del subscriber sarà indipendente dalla
    -- risposta del broker, infatti, il subscriber andrà comunque nello stato
    -- di "read" in modo tale da lascia il suo turno ad altri.
```

```
        (my_turn = TRUE) & (state = subscribe) & ((broker.state = accept_subscription_sub1) | (broker.state = accept_
subscription_sub2)) : read;
        -- (my_turn = TRUE) & (state = subscribe) & ((broker.state = accept_subscription_sub1) | (broker.state = acce
pt_subscription_sub2) | (broker.state = accept_subscription_sub3)) : read;

        -- Chiedere di essere iscritti ad un topic finché il broker non accetta
        -- esporrebbe il sistema ad un rischio di starvation: il sistema entra
        -- in un loop "subscribe-refuse" e il turno non viene mai ceduto. Per
        -- questo è importante che anche se il broker rifiuta la richiesta del
        -- subscriber, questo cede comunque il turno senza riprovare.
        (my_turn = TRUE) & (state = subscribe) & (broker.state = refuse) : read;

        -- Quando il broker inoltra una richiesta di notifica al subscriber,
        -- può far riferimento al subscriber 1 o al subscriber 2. Il subscriber
        -- controlla sempre se la notifica è rivolta a sé stesso e se così non
        -- fosse la rifiuta. Dopodiché il subscriber va a leggere nella struttura
        -- "topics" per vedere se è iscritto oppire no. Solo nel caso in cui il
        -- subscriber è iscritto al topic in questione la notifica viene accettata
        -- altrimenti si rifiuta.
        (state = ready) & (broker.state = notify_sub1) & (id = 1) & (topics[broker.topic_to_notify] = TRUE) : accept_u
pdate;
        (state = ready) & (broker.state = notify_sub2) & (id = 2) & (topics[broker.topic_to_notify] = TRUE) : accept_u
pdate;
        (state = ready) & (broker.state = notify_sub1) & ((id != 1) | (topics[broker.topic_to_notify] = FALSE)) : refuse
_update;
        (state = ready) & (broker.state = notify_sub2) & ((id != 2) | (topics[broker.topic_to_notify] = FALSE)) : refuse
_update;

        -- (state = ready) & (broker.state = notify_sub3) & (id = 3) & (topics[broker.topic_to_notify] = TRUE) : accept
_update;
        -- (state = ready) & (broker.state = notify_sub3) & ((id != 3) | (topics[broker.topic_to_notify] = FALSE)) : refu
se_update;

        -- Una volta gestita la notifica del broker, il subscriber torna nello
        -- stato di "ready" da cui potrebbe provare a fare richieste una volta
        -- ottenuto il turno.
        (state = accept_update) | (state = refuse_update) : ready;

        TRUE : state;
      esac;

    next(enrolled) :=
      case
       (my_turn = TRUE) & (state = enroll) & (broker.state = acknowledge) : TRUE;
       TRUE : enrolled;
      esac;

    -- Nel momento in cui il subscriber va nello stato di "read", in modo non deterministico
    -- viene selezionato il topic che vorrebbe leggere.
    next(topic) :=
      case
        -- (my_turn = TRUE) & (state = read) : {1, 2, 3, 4};
        -- (my_turn = TRUE) & (state = read) : {1, 2, 3};
        (my_turn = TRUE) & (state = read) : {1, 2};
        TRUE : topic;
```

```
    esac;

  -- Se il subscriber non è in possesso del "topic" che vorrebbe leggere allora
  -- raggiunge lo stato di "subscribe". La seguente variabile viene valorizzata
  -- proprio con il valore di "topic".
  next(topic_to_subscribe) :=
   case
    (my_turn = TRUE) & (state = subscribe) : topic;
    TRUE : topic_to_subscribe;
   esac;

  -- Quando il broker risponde positivamente alla richiste di iscrizione per
  -- il topic ID = i al subscriber si imposta la cella i-esima a "TRUE". In questo
  -- modo il subscriber tiene traccia dei topic in suo possesso.
  next(topics[1]) :=
    case
      (topic_to_subscribe = 1) & (id = 1) & (broker.state = accept_subscription_sub1) : TRUE;
      (topic_to_subscribe = 1) & (id = 2) & (broker.state = accept_subscription_sub2) : TRUE;
      -- (topic_to_subscribe = 1) & (id = 3) & (broker.state = accept_subscription_sub3) : TRUE;
      TRUE : topics[1];
    esac;
  next(topics[2]) :=
    case
      (topic_to_subscribe = 2) & (id = 1) & (broker.state = accept_subscription_sub1) : TRUE;
      (topic_to_subscribe = 2) & (id = 2) & (broker.state = accept_subscription_sub2) : TRUE;
      -- (topic_to_subscribe = 2) & (id = 3) & (broker.state = accept_subscription_sub3) : TRUE;
      TRUE : topics[2];
    esac;
  -- next(topics[3]) :=
  --   case
  --     (topic_to_subscribe = 3) & (id = 1) & (broker.state = accept_subscription_sub1) : TRUE;
  --     (topic_to_subscribe = 3) & (id = 2) & (broker.state = accept_subscription_sub2) : TRUE;
  --     TRUE : topics[3];
  --   esac;
  -- next(topics[4]) :=
  --   case
  --     (topic_to_subscribe = 4) & (id = 1) & (broker.state = accept_subscription_sub1) : TRUE;
  --     (topic_to_subscribe = 4) & (id = 2) & (broker.state = accept_subscription_sub2) : TRUE;
  --     TRUE : topics[4];
  --   esac;



-----------------------------
-- | MODULO   *BROKER*     |
-----------------------------

-- Questo modulo modella il comportamento di una blockchain che agisce all'interno
-- del sistema come un broker. Il broker si comporta come un gestore delle richieste
-- che arriveranno dai publisher e dai subscriber. Affinché un broker gestisca
-- le richieste provenienti da altre blockchain richiede che queste come prima cosa
-- si registrino ad esso. Solo quando il broker riconosce una blockchain passerà
-- poi a gestire una sua richiesta. Da questo punto di vista, diciamo che il
-- protocollo che si è modellato si basa fortemente sul concetto di autenticazione.
-- Questo meccanismo viene modellato grazie ad una variabile, "enrolled", che
```

```
-- dovrà appartenere sia al modulo che modella un publisher, sia al modulo che
-- che modella un subscriber. Si assume essere fondamentale il fatto che un broker
-- comunichi solo con blockchain registrate.
-- Essendo un gestore di richieste, normalmente il broker si troverà in uno stato
-- di riposo chiamato "sleep". Questo verrà svegliato nel momento in cui una blockchain
-- inoltra una richiesta. A questo punto è importante gestire la concorrenza delle
-- richieste. La scelta di progettazione adottata prevede che una blockchain potrà
-- prendere il turno solo se qualcuno lo ha già rilasciato e allo stesso tempo
-- il broker si trova proprio nello stato di "sleep", ossia non sta gestendo altre
-- richieste. Si immagini una situazione in cui un subscriber prenda il turno
-- ed è intenzionato a chiedere al broker qualcosa. Se il subscriber ha preso il
-- turno ci si aspetta che il broker fosse in "sleep". Il broker è libero di gestire
-- la richiesta del subscriber. Per tutto il tempo necessario a gestire la richiesta
-- del subscriber, il broker rimane bloccato in una comunicazione diretta e chiusa
-- "subscriber-broker". Durante questo periodo, tutte le altre blockchain estranee
-- a tale canale di comunicazione dovranno rimanere in attesa. Il broker torna
-- di nuovo disponibile quando il subscriber riceve la risposta che stava aspettando
-- dallo stesso.
-- Il secondo ruolo coperto dal broker è quello di notificare i cambiamenti dei
-- topic in suo possesso ai subscriber ad essi iscritti. Allora anche il broker
-- ha bisogno di acquisire un turno. Il sistema è stato modellato in modo tale per
-- cui le notifiche vengano inoltrate il prima possibile ai subscriber, infatti,
-- (si può vedere nel modulo "mutex") non appena un publisher avrà pubblicato
-- un aggiornamento di un topic, il turno verrà ceduto al broker che raggiungerà
-- uno stato di "notify" da cui potrà inoltrare le notifiche. Dopo aver completato
-- il giro dei subscriber da notificare, il broker cederà il suo turno. Per sapere
-- a chi mandare le notifiche, il broker sarà dotatao di una struttura dati che
-- per ogni topic mostra quali sono i subscriber che vi sono iscritti.

-- PARAMETRI: * pub       : Blockchain iscritta come publisher.
--            * sub1, sub2 : Blockchain iscritte come subscriber.
--            * turn       : Indica di chi è il turno.

-- MODULE broker(pub, sub1, sub2, sub3, turn)
MODULE broker(pub, sub1, sub2, turn)

  VAR
    -- sleep                : Il broker non sta gestendo alcuna richiesta.
    -- acknowledge             : Accetta una richiesta di registrazione alla rete.
    -- accept_new              : Accetta la richiesta da parte di un publisher di pubblicare un nuovo topic.
    -- accept_update           : Accetta la richiesta da parte di un publisher di aggiornare un topic.
    -- refuse                  : Rifiuta una generica richiesta che ha ricevuta.
    -- accept_subscription_sub1 : Accetta la richiesta da parte del subscriber 1 di iscrizione ad un topic.
    -- accept_subscription_sub2 : Accetta la richiesta da parte del subscriber 2 di iscrizione ad un topic.
    -- notify                  : Accetta un aggiornamento di un topic e vuole iniziare il processo di notifica.
    -- notify2                 : Inizia il processo di notifica dal subscriber 2 perché il primo non è iscritto al topic.
    -- notify_sub1             : Notifica il subscriber 1
    -- notify_sub1             : Notifica il subscriber 2
    state: {
      -- accept_subscription_sub3, notify_sub3, notify3,
      sleep, acknowledge, accept_new, accept_update, refuse,
      accept_subscription_sub1, accept_subscription_sub2,
      notify, notify2, notify_sub1, notify_sub2
    };
```

```
-- La variabile "topics" consiste in un array contenente tutti i topic che sono
-- stati pubblicati dai publisher. Questa struttura dati verrà interpellata
-- dal broker ogni qual volta deve decidere come gestire una richiesta di
-- pubblicazione proveniente da un publisher. La logica su cui il broker basa
-- il suo processo di decisione è la seguente. Immaginiamo che il publisher
-- voglia pubblicare il topic con ID = 1. Abbiamo due casi:
--    1) Il publisher chiede di pubblicare un nuovo topic. In questo caso, il
--       broker accetta la richiesta solo se nella cella corrispondente al
--       topic 1 trova il valore "non_esiste", altrimenti rifiuta.
--    2) Il publisher chiede di pubblicare un aggiornamento. Il broker accetta
--       la richiesta solo se dalla struttura dati legge che il topic esiste.
-- Si sceglie di utilizzare un array di enum piuttosto che di booleani per
-- aggiungere espressività al modello.
-- topics: array 0..4 of {non_esiste, prima_versione, ultima_versione};
-- topics: array 0..3 of {non_esiste, prima_versione, ultima_versione};
topics: array 0..2 of {non_esiste, prima_versione, ultima_versione};

-- Rappresenta il topic appena creato o aggiornato di cui il broker vuole
-- notificare i cambiamenti ai subscriber.
-- Il valore "0" indica che il broker non deve notificare alcuna modifica.
-- Da notare che l'array parte dall'indice "0".  Senza questo accorgimento
-- non si riesce a compilare il file sorgente.
-- topic_to_notify: {0, 1, 2, 3, 4};
-- topic_to_notify: {0, 1, 2, 3};
topic_to_notify: {0, 1, 2};

-- Il broker ha bisogno di sapere sempre quali sono i subscriber iscritti ad
-- ognuno dei topic pubblicati dai publisher. La struttura dati "sub_topics"
-- organizzata come una matrice contiene in ogni cella un valore booleano.
-- Si compone di tante colonne quanti sono i subscriber e tante righe quanti
-- sono i topic pubblicati. Ad esempio:
--
--      MATRICE 4x2
--
--                    |  sub1  | sub2
--                  ----------------------
--                  1  |  FALSE | FALSE
--                  2  |  TRUE  | TRUE
--                  3  |  FALSE | FALSE
--                  4  |  TRUE  | FALSE
--
-- Quindi, prendendo la matrice dell'esempio, si può dire che il subscriber 1
-- è iscritto ai topic con ID = 2 e ID = 4, mentre il subscriber 2 è iscritto
-- al topic con ID = 2.
-- Si notino le celle [0, sub1] e [0, sub2]. Queste due posizioni della matrice
-- vengono aggiunte per gestire quei casi in cui la variabile "topic_to_notify"
-- vale "0". Senza questo accorgimento il modello sarebbe soggetto ad errori
-- "out-of-bound" e non si riesce a compilare il file sorgente.
-- sub_topics: array 0..4 of array 1..2 of boolean;
-- sub_topics: array 0..3 of array 1..2 of boolean;
-- sub_topics: array 0..2 of array 1..3 of boolean;
sub_topics: array 0..2 of array 1..2 of boolean;
```

```
init(state) := sleep;

init(topic_to_notify) := 0;

init(topics[0]) := non_esiste;
init(topics[1]) := non_esiste;
init(topics[2]) := non_esiste;
-- init(topics[3]) := non_esiste;
-- init(topics[4]) := non_esiste;

init(sub_topics[0][1]) := FALSE;
init(sub_topics[1][1]) := FALSE;
init(sub_topics[2][1]) := FALSE;
-- init(sub_topics[3][1]) := FALSE;
-- init(sub_topics[4][1]) := FALSE;

init(sub_topics[0][2]) := FALSE;
init(sub_topics[1][2]) := FALSE;
init(sub_topics[2][2]) := FALSE;
-- init(sub_topics[3][2]) := FALSE;
-- init(sub_topics[4][2]) := FALSE;

-- init(sub_topics[0][3]) := FALSE;
-- init(sub_topics[1][3]) := FALSE;
-- init(sub_topics[2][3]) := FALSE;

next(state) :=
  case
    -- Gestione della richiesta di registrazione al broker da parte di un publisher.
    -- Il vincolo affinché un publsiher venga autentica è che questo non sia
    -- già registrato alla rete.
    (state = sleep) & (pub.state = enroll) & (pub.enrolled = FALSE) : acknowledge;
    (state = sleep) & (pub.state = enroll) & (pub.enrolled = TRUE) : refuse;

    -- Gestione della richiesta di registrazione al broker da parte di un subscriber.
    -- Anche in questo caso, l'unico vincolo è che un subscriber non sia già
    -- stato riconosicuto in precedenza dal broker.

    -- Gestione della richiesta da parte del subscriber 1.
    (state = sleep) & (sub1.state = enroll) & (sub1.enrolled = FALSE) : acknowledge;
    (state = sleep) & (sub1.state = enroll) & (sub1.enrolled = TRUE) : refuse;

    -- Gestione della richiesta da parte del subscriber 2.
    (state = sleep) & (sub2.state = enroll) & (sub2.enrolled = FALSE) : acknowledge;
    (state = sleep) & (sub2.state = enroll) & (sub2.enrolled = TRUE) : refuse;

    -- Gestione della richiesta da parte del subscriber 3.
    -- (state = sleep) & (sub3.state = enroll) & (sub3.enrolled = FALSE) : acknowledge;
    -- (state = sleep) & (sub3.state = enroll) & (sub3.enrolled = TRUE) : refuse;

    -- Gestione della richiesta di pubblicare un nuovo topic da parte di un publisher.
    -- Il broker verifica leggendo la struttura dati "topics" se il topic specificato
    -- dal publisher esista già. Solo nel caso in cui il topic non esiste il
    -- broker accetterà la richiesta. Ovviamente il publisher deve essere registrato.
    (state = sleep) & (pub.state = publish_new) & (pub.enrolled = TRUE) & (topics[pub.topic] = non_esiste) : acce
```

```
pt_new;
    (state = sleep) & (pub.state = publish_new) & ((pub.enrolled = FALSE) | (topics[pub.topic] != non_esiste)) : ref
use;

    -- Gestione della richiesta di pubblicare un aggiornamento di un topic già esistente da parte di un publisher.
    -- Viceversa, questa volta il broker deve accertarsi che il topic specificato dal publisher esista.
    -- Se il topic non esiste la richiesta viene rifiutata, altrimenti il broker provvederà
    -- ad aggiornare il topic ed in seguito a notificare i subscriber.
    (state = sleep) & (pub.state = publish_update) & (pub.enrolled = TRUE) & (topics[pub.topic] != non_esiste) : a
ccept_update;
    (state = sleep) & (pub.state = publish_update) & ((pub.enrolled = FALSE) | (topics[pub.topic] = non_esiste)) : r
efuse;

    -- Quando il broker riceve una richiesta di sottoscrizione ad un topic da parte di un subscriber,
    -- per prima cosa verifica che questo sia registato alla rete. Dopodiché, per confermare la
    -- richiesta del subscriber, il broker verifica di essere in possesso del topic specificato.

    -- Gestione della richiesta di sottoscrizione del subscriber 1
    (state = sleep) & (sub1.state = subscribe) & (sub1.enrolled = TRUE) & (topics[sub1.topic_to_subscribe] != non
_esiste) : accept_subscription_sub1;
    (state = sleep) & (sub1.state = subscribe) & ((sub1.enrolled = FALSE) | (topics[sub1.topic_to_subscribe] = non
_esiste)) : refuse;

    -- Gestione della richiesta di sottoscrizione del subscriber 2
    (state = sleep) & (sub2.state = subscribe) & (sub2.enrolled = TRUE) & (topics[sub2.topic_to_subscribe] != non
_esiste) : accept_subscription_sub2;
    (state = sleep) & (sub2.state = subscribe) & ((sub2.enrolled = FALSE) | (topics[sub2.topic_to_subscribe] = non
_esiste)) : refuse;

    -- Gestione della richiesta di sottoscrizione del subscriber 3
    -- (state = sleep) & (sub3.state = subscribe) & (sub3.enrolled = TRUE) & (topics[sub3.topic_to_subscribe] != n
on_esiste) : accept_subscription_sub3;
    -- (state = sleep) & (sub3.state = subscribe) & ((sub3.enrolled = FALSE) | (topics[sub3.topic_to_subscribe] = n
on_esiste)) : refuse;

    -- Dopo che il broker accetta di aggiornare un topic già esistente deve occuparsi della
    -- fase di notifica. Quando il broker raggiunge questo stato, ha bisogno del turno
    -- per poter procedere con l'effettivo invio delle notifiche.
    (state = accept_update) : notify;

    -- Si fa notare una scelta di progettazione. I subscriber verificano sempre
    -- la notifica che ricevono prima di accettarla e in caso di conferma aggiorneranno
    -- poi il topic (si veda il modulo "subscriber"). Esistono diverse strade
    -- percorribili per notificare gli aggiornamenti. Ad esempio, si è pensato ad
    -- una gestione della comunicazione secondo la metodologia broadcast; il vantaggio
    -- che avrebbe portato avrebbe riguardato l'efficienza, infatti, il broker non avrebbe
    -- dovuto fare altro che inoltrare a tutti i subscriber registrati una notifica.
    -- Dato che tra le qualità fondamentali di una blockchain vi è l'affidabilità, si
    -- è optato invece di comunicare un aggiornamento di un topic in modo sequenziale.
    -- In questo modo, il broker aggiunge uno strato di sicurezza controllando lui
    -- stesso se il subscriber che sta contattando è effettivamente interessato al topic
    -- oppure no. Grazie a questo controllo il broker deve riuscire ad inoltrare
    -- le notifiche solo ai subscriber interessati al topic in questione.

    -- Il broker verifica due cose prima di inoltrare la notifica al subscriber:
```

```
      --    1) il subscriber deve essere registrato alla rete
      --    2) il subscriber deve risultare iscritto al topic da notificare
      -- Se il subscriber rispetta queste due condizioni allora il broker inoltra
      -- la notifica, in questo caso, al subscriber 1. Se il broker decide di non
      -- inoltrare la notifica allora si mette nello stato "notify2" dal quale
      -- potrà verificare le stesse due condizioni per il subscriber 2.
      (turn = b) & (state = notify) & (sub1.enrolled = TRUE) & (sub_topics[topic_to_notify][1] = TRUE) : notify_sub1;
      (turn = b) & (state = notify) & ((sub1.enrolled = FALSE) | (sub_topics[topic_to_notify][1] = FALSE)) : notify2;

      -- Dopo aver gestito la notifica rivolta al subscriber 1, il broker passa a gestire quella
      -- per il subscriber 2 basandosi sulle stesse due regole di verifica. Dato che nella
      -- topologia sono previsti solamente due subscriber, dopo aver trattato la notifica per
      -- il subscriber 2 il broker rilascierà il suo turno.
      (turn = b) & ((state = notify2) | (state = notify_sub1)) & (sub2.enrolled = TRUE) & (sub_topics[topic_to_notify][2] = TRUE) : notify_sub2;
      (turn = b) & ((state = notify2) | (state = notify_sub1)) & ((sub2.enrolled = FALSE) | (sub_topics[topic_to_notify][2] = FALSE)) : sleep;

      -- (turn = b) & ((state = notify2) | (state = notify_sub1)) & ((sub2.enrolled = FALSE) | (sub_topics[topic_to_notify][2] = FALSE)) : notify3;
      -- (turn = b) & ((state = notify3) | (state = notify_sub2)) & (sub3.enrolled = TRUE) & (sub_topics[topic_to_notify][3] = TRUE) : notify_sub3;
      -- (turn = b) & ((state = notify3) | (state = notify_sub2)) & ((sub3.enrolled = FALSE) | (sub_topics[topic_to_notify][3] = FALSE)) : sleep;


      -- Dopo aver finito di gestire una qualunque richiesta proveniente da una blockchain,
      -- o dopo aver concluso il processo di notifiche verso i subscriber, il broker
      -- raggiunge uno stato neutro, "sleep". Se il broker ha raggiunto questo stato
      -- vuol dire che è in grado di iniziare a gestire una nuova richiesta.
      -- (state = notify_sub3) |
      (state = acknowledge) | (state = refuse) | (state = notify_sub2) | (state = accept_new)
      | (state = accept_subscription_sub1) | (state = accept_subscription_sub2) : sleep;

      TRUE : state;
    esac;

  -- Quando il broker accetta una richiesta di aggiornamento di un topic inoltrata
  -- da un publisher, deve preoccuparsi di mandare una notifica ai subscriber.
  -- La variabile "topic_to_notify" viene valorizzata con il topic che è stato
  -- specificato dal publisher. Leggendo questo valore, il broker e i subscriber
  -- sanno a che topic una certa notifica si riferisce.
  next(topic_to_notify) :=
    case
      (state = accept_update) : pub.topic;
      TRUE : topic_to_notify;
    esac;

  next(topics[1]) :=
    case
      (state = accept_new) & (pub.topic = 1) : prima_versione;
      (state = accept_update) & (pub.topic = 1) : ultima_versione;
      TRUE : topics[1];
```

```
  esac;
next(topics[2]) :=
  case
    (state = accept_new) & (pub.topic = 2) : prima_versione;
    (state = accept_update) & (pub.topic = 2) : ultima_versione;
    TRUE : topics[2];
  esac;
-- next(topics[3]) :=
--   case
--     (state = accept_new) & (pub.topic = 3) : prima_versione;
--     (state = accept_update) & (pub.topic = 3) : ultima_versione;
--     TRUE : topics[3];
--   esac;
-- next(topics[4]) :=
--   case
--     (state = accept_new) & (pub.topic = 4) : prima_versione;
--     (state = accept_update) & (pub.topic = 4) : ultima_versione;
--     TRUE : topics[4];
--   esac;

-- Nel momento in cui il broker conferma la richiesta di iscrizione ad un topic
-- da parte di un subscriber, imposta a "TRUE" la cella opportuna.
next(sub_topics[1][1]) :=
  case
    (state = accept_subscription_sub1) & (sub1.topic_to_subscribe = 1) : TRUE;
    TRUE : sub_topics[1][1];
  esac;
next(sub_topics[2][1]) :=
  case
    (state = accept_subscription_sub1) & (sub1.topic_to_subscribe = 2) : TRUE;
    TRUE : sub_topics[2][1];
  esac;
-- next(sub_topics[3][1]) :=
--   case
--     (state = accept_subscription_sub1) & (sub1.topic_to_subscribe = 3) : TRUE;
--     TRUE : sub_topics[3][1];
--   esac;
-- next(sub_topics[4][1]) :=
--   case
--     (state = accept_subscription_sub1) & (sub1.topic_to_subscribe = 4) : TRUE;
--     TRUE : sub_topics[4][1];
--   esac;


next(sub_topics[1][2]) :=
  case
    (state = accept_subscription_sub2) & (sub2.topic_to_subscribe = 1) : TRUE;
    TRUE : sub_topics[1][2];
  esac;
next(sub_topics[2][2]) :=
  case
    (state = accept_subscription_sub2) & (sub2.topic_to_subscribe = 2) : TRUE;
    TRUE : sub_topics[2][2];
  esac;
-- next(sub_topics[3][2]) :=
```

```
--   case
--     (state = accept_subscription_sub2) & (sub2.topic_to_subscribe = 3) : TRUE;
--     TRUE : sub_topics[3][2];
--   esac;
-- next(sub_topics[4][2]) :=
--   case
--     (state = accept_subscription_sub2) & (sub2.topic_to_subscribe = 4) : TRUE;
--     TRUE : sub_topics[4][2];
--   esac;

-- next(sub_topics[1][3]) :=
--   case
--     (state = accept_subscription_sub3) & (sub3.topic_to_subscribe = 1) : TRUE;
--     TRUE : sub_topics[1][3];
--   esac;
-- next(sub_topics[2][3]) :=
--   case
--     (state = accept_subscription_sub3) & (sub3.topic_to_subscribe = 2) : TRUE;
--     TRUE : sub_topics[2][3];
--   esac;
```