Corso di Laurea Magistrale in
Computer Engineering

Observe Extension of CoAP with QoS

Alessandro Sieni
Amedeo Pochiero
Roberto Magherini

Indice

Capitolo 1

Specifiche

1.1 Comunicazione

- La comunicazione tra i nodi sensori ed il proxy server avviene mediante protocollo CoAP con l'ausilio di un border router collegato al proxy server.
- Gli observer si collegano invece al proxy server utilizzando una rete ad-hoc (o preesistente) creata dallo stesso proxy server.

1.2 Ottenimento Lista Risorse da parte degli Observer

- All'avvio dell'osservatore verrà fatta una richiesta al proxy server per ottenere la lista delle risorse disponibili (nei nodi sensori) e il proxy stesso risponderà fornendo tutti i nodi sensori collegati ad esso e le relative risorse in grado di offrire.

1.3 Registrazione

- Il nodo sensore sta in attesa fino a quando non arriva la prima registrazione.
- Il processo di registrazione seguirà le specifiche indicate nel paper utilizzati i campi observe e token e il comando GET.
- Se il nodo sensore ha attiva una registrazione di livello i il proxy non invierà una richiesta di livello superiore in quanto ha già i dati necessari a servire il nuovo osservatore.

- La registrazione tra nodo sensore e proxy server avviene in modo leggermente diverso rispetto a quanto indicato nel paper, dato che rispettivamente i livello 1-2 e 3-4 servono solo a differenziare l'ordine con cui devono essere inviati i dati (solo critici o tutti), ma dato che nel nostro schema è presente solo un osservatore dal punto di vista del sensore questa differenziazione non è più necessaria, riducendo quindi i livelli di priorità a 2, il livello 1 solo per i valori critici ed il livello 2 per tutti i valori ottenuti dal sensore.Il livello 3 dello stesso campo sarà utilizzato dal proxy server per disiscriversi dal nodo sensore relativamente a quella risorsa.

- Per disiscriversi da un soggetto l'osservatore comunicherà al proxy tale richiesta secondo le specifiche indicate dal paper, e qualora nessun osservatore sarà più interessato ad essere notificato da tale soggetto, il proxy invierà al nodo sensore un messaggio GET indicando il livello 3 che viene usato per la disiscrizione.

1.4 Gestione dei Soggetti nel Proxy Server

- Al fine di garantire le migliori performance e la maggiore scalabilità possibile si intende realizzare un meccanismo di caching sul proxy server che sia in grado di memorizzare gli ultimi valori ottenuti da ciascun soggetto, in modo da poterli ottenere nel momento in cui devono essere inviati a tutti gli osservatori in ascolto.

- Per ottenere questo meccanismo sarà realizzata una struttura dati associata ad ogni nodo sensore, e ogni qual volta che un nuovo dato arriverà da un soggetto, la sua struttura dati sarà aggiornata e sarà notificata il modulo del proxy server che si occuperà di inviare questo nuovo valore a tutti gli osservatori registrati.

- Se un valore rimane nella struttura dati per un tempo maggiore rispetto al max-age indicato nel pacchetto associato, allora significa che il nodo sensore non è più in grado di produrre nuovi valori e quindi il proxy si considera disiscritto, rimuovendo anche la lista degli osservatori registrati a quella risorsa.

1.5 Gestione degli Osservatori nel Proxy Server

- Durante la fase iniziale, dopo aver inviato la lista delle risorse disponibili, il proxy server memorizza anche le informazioni dell'osservatore a cui ha inviato tale lista.

- Questo modulo del proxy server si occuperà di inviare i valori ottenuti dai soggetti qualora ne arrivi uno nuovo oppure appena dopo aver completato il protocollo di

registrazione con un osservatore, nel caso in cui sia disponibile il valore relativo a quella risorsa.

- La gestione della priorità dal lato proxy-osservatore reintrodurrà quattro livelli, seguendo le specifiche del paper, in modo da rendere il proxy trasparente dal punto di vista dell'osservatore. Lo stesso discorso si può applicare per il processo di disiscrizione che dal punto di vista dell'osservatore sarà identico a quello proposto nel paper.

- Qualora arrivasse una richiesta di disiscrizione da parte di un osservatore verso un soggetto, il proxy server non farà altro che rimuovere il soggetto in questione dalla lista di quelli iscritti a tale risorsa.

- Negoziazione della priorità di invio da parte del soggetto in base al livello di batteria presente nel nodo sensore a cui appartiene la risorsa interessata.

1.6 Ottenimento Risorse nel Nodo Sensore

- Un nodo sensore avvia un timer costante per effettuare il sensing qualora sia presente almeno un osservatore interessato ad almeno una risorsa.

- Allo scadere del timer si salverà il valore ottenuto confrontandolo con il valore precedentemente inviato per vedere se è rimasto all'interno di uno specifico range, oppure se completamente un altro valore.

- Il nodo sensore ferma il timer della risorsa nel momento in cui nessun osservatore è più interessato a qualche risorsa, risparmiando cosi energia.

1.7 Gestione Priorità dei Valori

- Il nodo sensore non avendo al suo interno una lista di osservatori, in quanto parlerà solo con il proxy, avrà per ciascuna risorsa un livello di priorità associato, che in questo caso sono due:

  - Critico : manda solo i valori considerati critici.
  - Non critico: manda tutti i valori ottenuti.

- Questo meccanismo permette quindi di ridurre allo stretto necessario il numero di invii, in quanto se ad una risorsa sono richiesti solo i valori critici, allora risulta inutile inviarli tutti.

- Per una corretta gestione di tutte le informazioni il soggetto assocerà alla risorsa il valore di priorità più basso tra quelli ottenuti : ovvero se al tempo 1 avrà associato solo valori critici ed al tempo 10 arriverà una registrazione per tutti i valori, da quel momento in poi il soggetto manderà tutti i valori, in quanto facendo l'opposto sarebbe certa la perdita di informazioni.

1.8 Gestione della Batteria

- Se il livello di energia è superiore al 30%, il nodo sensore invierà tutti i valori ottenuti, sia critici che non critici, mentre se il livello è sotto a tale soglia saranno inviati solo i valori critici, in modo da risparmiare energia.

- Per gli osservatori interessati a tutti i valori, avverrà un disiscrizione e qualora volessero riscriversi dovranno rieseguire il protocollo di registrazione.

- Qualora debba essere eseguita una registrazione, se siamo sopra a tale soglia tutte potranno essere accettate, altrimenti solo la registrazione che richiede i valori critici potrà andare a buon fine, mentre quella che richiede tutti i valori no.

1.9 Trasmissione dei Valori dal Nodo Sensore

- Se il valore appena ottenuto è considerato critico non saranno fatti calcoli o controlli su di esso, ma bensì sarà immediatamente inviato al proxy server, dato che in questa situazione la componente cruciale risulta essere il tempo.

- Se invece il valore ottenuto non è critico, allora sarà eseguito un confronto con il precedente per vedere se è necessario inviare il valore.

- Se siamo vicini alla scadenza del max-age allora obbligatoriamente l'ultimo valore registrato sarà inviato, in quanto altrimenti dal punto dell'osservatore risulterà che il nodo sensore l'abbia disiscritto dalla sua lista.

1.10 Gestione del max-age nel Nodo Sensore

- Al fine di ottimizzare il numero di trasmissione da parte del soggetto, è stato scelto di introdurre un max-age variabile nel tempo, in modo da diminuire, quanto possibile la trasmissione di valori simili o uguali tra loro.
- All'ottenimento di un nuovo valore verrà controllato che sia critico o meno:
  - Nel caso in cui sia critico il max-age sarà messo ad un preciso valore che deve essere basso, in quanto si presume che anche il prossimo sia critico
  - Nel caso in cui non sia critico verrà eseguito un confronto con il precedente valore:
    - Nel caso in cui il nuovo sia vicino al precedente di un certo intervallo:
      - Se caso in cui siamo prossimi alla scadenza del precedente max-age, allora il nuovo valore sarà inviato con un max-age superiore.
      - Altrimenti il valore misurato sarà scartato in quanto il confronto deve essere sempre fatto tra il valore misurato e l'ultimo inviato.
    - Altrimenti il valore sarà inviato ed il max age sarà impostato al minimo.
- I valori che potranno essere assunti dal max-age sono i seguenti:
  - Per un valore critico max-age = 2*sensingTime+soglia.
  - Per un valore non critico il max-age sarà all'interno della finestra [2*sensingTime+soglia; 5*sensingTime+soglia] ed incrementerà a step di sensingTime.
  - La soglia è stata introdotta per non far coincidere il max-age con il sensingTime, in quanto ciò potrebbe causare problemi di sovrapposizione.

1.11 Implementazione

L'architettura del nostro sistema si dividerà nei seguenti moduli:

- Il modulo osservatore che riceverà i dati.
- Il proxy che al suo al interno avrà due sottomoduli:
  - Il sottomodulo ProxyObserver responsabile della gestione e della comunicazione con gli osservatori, inoltrando al modulo ProxySubject le eventuali richieste di registrazione e di disiscrizione.

  - Il sottomodulo ProxySubject che invece si occupa di gestire e comunicare con i soggetti, fornendo al sottomodulo ProxyObserver le strutture dati necessari per poter trasmettere i valori agli osservatori.

- Il modulo che si occupa di gestire il soggetto, con le relative misure e trasmissioni.

1.11.1 Modulo Osservatore

Questo modulo risulterà essere un semplice client di prova senza interfaccia grafica realizzato in java che si occuperà solo di richiedere la lista delle risorse e di effettuare registrazione/disiscrizione, stampando a video i valori ricevuti dai soggetti.

1.11.2 Modulo Proxy

Il modulo che sarà eseguito sul proxy sarà sviluppato interamente in java, anch'esso senza interfaccia grafica, e si occuperà di far comunicare gli osservatori con i soggetti, cercando di massimizzare le performance ed il consumo di energia per i soggetti stessi.

Sottomodulo ProxyObserver

- Sarà un thread apposito per gestire le richieste di registrazione e disiscrizione, creando un sottothread per ciascuna di esse occupandosi anche di inoltrare (richiamando funzioni specifiche offerte dal sottomodulo ProxySubject) tali richieste al ProxySubject solo se la negoziazione ha avuto esito positivo.

- Sarà presente un altro thread per l'invio delle notifiche agli osservatori seguendo le priorità specificate durante il processo di negoziazione.

- Per quanto riguarda le strutture dati o classi saranno le seguenti:

  - Una classe che si occupa di gestire gli osservatori mantenendo le informazioni necessarie per la connessione.
  - Per ogni risorsa ci saranno quattro code in cui verrano inseriti le istanze degli osservatori registrati e inseriti nella relativa coda in base alla priorità negoziata.

Sottomodulo ProxySubject

- Sarà presente una lista di strutture dati che serviranno a virtualizzare i soggetti fornendo le seguenti informazioni:

  - L'indirizzo ip del nodo sensore.
  - Il livello di batteria del nodo sensore
  - Una lista delle risorse disponibili, virtualizzate tramite una struttura dati che mi fornisce i seguenti valori:
    * Il tipo della risorsa (temperatura, umidità,...).
    * L'ultimo valore ricevuto ed il max-age relativo per ciascuna risorsa ottenibile da tale nodo sensore.
    * Il livello dell'attuale registrazione con quel nodo sensore per ogni risorsa ottenibile da tale nodo sensore.

- Questo sottomodulo fornirà anche dei metodi per poter accedere ai valori della precedente struttura dati, in modo da poter completamente virtualizzate il nodo sensore dal punto di vista del ProxyObserver, come ad esempio:

  - Il metodo che restituisce il livello di batteria dello specifico nodo sensore.
  - Un metodo per ottenere l'ultimo valore registrato dal nodo sensore per una specifica risorsa.
  - Un metodo per potersi registrare allo specifico nodo sensore.
  - Un metodo per potersi disiscriversi dal nodo sensore.

1.11.3 Modulo Soggetto

- Implementato in C.
- Sarà presente una struttura dati con le informazioni necessarie per comunicare con il proxy
- Avrà delle soglie per determinare se un valore è critico o meno
- Avrà un livello di priorità per ogni risorsa alla quale è stata la registrazione.
- Avrà un timer, comune a tutte le risorse, per decidere quando effettuare il sensing del nuovo valore.
- Deciderà se effettuare il sensing di una risorsa se è presente una registrazione su quella risorsa.
- Avrà un timer per gestire la scadenza del max-age, come da algoritmo precedentemente riportato.

- Avrà un valore che virtualizzerà il livello di batteria, in particolare verrà aggiornato ad ogni sensing e ad ogni trasmissione di un valore costante, differente per i due casi.

1.12 Testing

Al termine dell'implementazione il sistema sarà testato con lo scopo di andare a valutare le performance del sistema, in particolare andremo ad analizzare:

- Delay: analisi del tempo necessario per la trasmissione del pacchetto dal soggetto all'osservatore differenziando il numero di osservatori collegati al proxy e le priorità da loro richieste.
- Energy Consumption: Analisi del consumo energetico sui nodi sensori distinguendo il caso in cui l'invio avviene sempre subito dopo il sensing oppure utilizzando l'algoritmo illustrato nelle specifiche.

Capitolo 2

Modulo ProxyObserver

2.1 Descrizione

Questo modulo permette la gestione della comunicazione tra il *Proxy* e l'*Observer* fungendo da CoapServer in ascolto sulla porta di default di CoAP 5683. La classe mantiene le informazioni relative allo stato degli osservatori e le risorse attualmente disponibili sui sensori visibili dagli osservatori. Quest'ultimi possono effettuare delle richieste di osservazione di una risorsa attualmente disponibile sul Proxy il quale gestisce la richiesta tramite il metodo *ObservableResource.handleGet(CoapExchange exchange)* che si occupa della negoziazione e degli invii delle notifiche.
Inoltre, fornisce i metodi per l'interazione con il modulo ProxySubject per l'aggiunta e la rimozione di risorse, notificare il cambio del valore della risorsa sul sensore ed eliminare le relazioni opportune dopo che lo stato di un sensore è cambiato.

2.2 Modifiche a Californium lato Server

Al fine di implementare le modifiche effettuate al protocollo CoAP, è stato necessario modificare anche la libreria Californium. In seguito verranno elencati i cambiamenti al codice della libreria, indicando il nome del file con un link alla repository con il codice originale e la motivazione di tale modifica. I file fanno riferimento alla package core di Californium:

2.2.1 coap.CoAP

Il livello di priorità viene indicato usando i primi 2 bit, rinominato campo *QoS* del campo Observe del pacchetto CoAP (3 bytes). Questa classe fornisce i valori che questo

campo può assumere

```
1  public class QoSLevel {
2
3    public static final int CRITICAL_HIGHEST_PRIORITY = 0xc00000; //
     12582912
4    public static final int CRITICAL_HIGH_PRIORITY = 0x800000; // 8388608
5    public static final int NON_CRITICAL_MEDIUM_PRIORITY = 0x400000; //
     4194304
6    public static final int NON_CRITICAL_LOW_PRIORITY = 0x000000; // 0
7
8    private QoSLevel() {
9      // prevent instantiation
10   }
11 }
```

Listing 2.1: coap.CoAP.QoSLevel, codice originale

2.2.2 coap.Request

Nel protocollo standard una Observe Request Registration può avere solo il valore 0 nel campo *Observe*. Nel caso considerato invece i valori possono essere 4, corrispondenti ai 4 livelli di priorità. Questa funzione deve ritorna *True* se contiene uno di questi valori.

```
1  public final boolean isObserve() {
2    // CHANGE_START
3    return isObserveOption(CoAP.QoSLevel.CRITICAL_HIGHEST_PRIORITY)
4        || isObserveOption(CoAP.QoSLevel.CRITICAL_HIGH_PRIORITY)
5        || isObserveOption(CoAP.QoSLevel.NON_CRITICAL_MEDIUM_PRIORITY)
6        || isObserveOption(CoAP.QoSLevel.NON_CRITICAL_LOW_PRIORITY);
7    // CHANGE_END
8  }
```

Listing 2.2: coap.Request, codice originale

2.2.3 observe.ObserveRelation

Una ObserveRelation mantiene le informazioni relative alla relazione tra un observer e la risorsa osservata. É stato aggiunto un campo alla classe per mantenere il livello di priorità di quella relazione, i suoi get e set e una funzione per comparare 2 relazioni in base a questo campo. Quest'ultima servirà nel meccanismo di notifica degli observer

basato su priorità.

```
1  // CHANGE_START
2  private int QoS;
3  // CHANGE_END
```

Listing 2.3: observe.ObserveRelation QoS, codice originale

```
1  // CHANGE_START
2
3  public void setQoS(int QoS) {
4    this.QoS = QoS;
5  }
6
7  public int getQoS() {
8    return QoS;
9  }
10
11
12 public int compareToDesc(ObserveRelation relation) {
13   if ( QoS == relation.getQoS() )
14     return 0;
15   if ( QoS > relation.getQoS() )
16     return -1;
17   return 1;
18 }
```

Listing 2.4: observe.ObserveRelation funzioni QoS, codice originale

2.2.4 server.ServerMessageDeliverer

Quando una *ObserveRelation* viene creata in seguito alla ricezione di una richiesta con l'opzione Observe, il campo *QoS* della relazione viene settato con il valore del campo Observe della richiesta.

```
1      if (request.isObserve()) {
2        // Requests wants to observe and resource allows it :-)
3        LOGGER.debug("initiating an observe relation between {} and
   resource {}", source, resource.getURI());
4        ObservingEndpoint remote = observeManager.findObservingEndpoint(
   source);
```

```
5           ObserveRelation relation = new ObserveRelation(remote, resource,
    exchange);
6           // CHANGE_START
7           relation.setQoS(request.getOptions().getObserve());
8           // CHANGE_END
9           remote.addObserveRelation(relation);
10          exchange.setRelation(relation);
11          // all that's left is to add the relation to the resource which
12          // the resource must do itself if the response is successful
13      }
```

Listing 2.5: server.ServerMessageDeliverer checkForObserveOption(...), codice originale

2.2.5 server.ServerState

Enumerato che definisce i 3 stati del nodo sensore:

1. **UNAVAILABLE**: il nodo sensore non è attivo, qualsiasi richiesta relativa a quel sensore è rigettata.
2. **ONLY_CRITICAL**: il nodo sensore ha un autonomia al di sotto di una certa soglia, quindi si accettano solo richieste da parte di observe richiedenti solo gli eventi critici della risorsa.
3. **AVAILABLE**: il nodo non ha problemi di autonomia quindi è possibile accettare qualsiasi tipo di richiesta.

```
1 public enum ServerState {
2   UNVAVAILABLE, ONLY_CRITICAL, AVAILABLE
3 }
```

Listing 2.6: ServerState

2.2.6 CoapResource

2.2.6.1 Aggiornamento relazioni dopo il cambio stato del sensore

Quando avviene un cambio di stato di un sensore, è necessario controllare che le relazioni già stabilite siano consistenti con il nuovo stato. Pertanto quando lo stato diventa **ONLY_CRITICAL**, le relazioni con livello *CoAP.QoSLevel.NON_CRITICAL_MEDIUM_PRIORITY* e *CoAP.QoSLevel.NON_CRITICAL_LOW_PRIORITY* vengono cancellate, mentre se si passa ad **UNAVAILABLE**, tutte le relazioni di quella risorsa vengono cancellate.

É stata quindi aggiunta una funzione per cancellare solo le relazioni con un livello non critico di priorità.

```
1  // CHANGE_START
2  public void clearAndNotifyNonCriticalObserveRelations(ResponseCode code)
     {
3    for (ObserveRelation relation : observeRelations) {
4      if (relation.getQoS() == CoAP.QoSLevel.NON_CRITICAL_MEDIUM_PRIORITY
5          || relation.getQoS() == CoAP.QoSLevel.NON_CRITICAL_LOW_PRIORITY)
     {
6        relation.cancel();
7        relation.getExchange().sendResponse(new Response(code));
8      }
9    }
10 }
11 // CHANGE_END
```

Listing 2.7: CoapResource clearAndNotifyNonCriticalObserveRelations, codice originale

2.2.6.2 Notifica basata su priorità

È stata aggiunta una lista *sortedObservers* di ObserveRelation ordinata in base al campo *QoS* di un ObserveRelation in modo decrescente. L'ordinamento è effettuato all'inserimento di una nuova relazione nel *ObserveRelationContainer observeRelations*. Questa nuova lista è impiegata per effettuare l'invio delle notifiche in ordine di priorità: l'invio delle notifiche è effettuato scansionando in modo sequenziale questa lista, partendo dal primo elemento (priorità maggiore), fino all'ultimo elemento (priorità minore). Per mantenere la lista consistente con l'ObserveRelationContainer, alla rimozione di un elemento da quest'ultimo, si rimuove lo stesso dalla sortedObservers.

```
1  // CHANGE_START
2  /* Sorted list of observers used to notifying following the correct order
     */
3  private ArrayList<ObserveRelation> sortedObservers;
4  // CHANGE_END
```

Listing 2.8: CoapResource sortedObservers, codice originale

```
1  @Override
2  public void addObserveRelation(ObserveRelation relation) {
3
```

```
4       if (observeRelations.add(relation)) {
5         LOGGER.info("replacing observe relation between {} and resource {} (
        new {}, size {})", relation.getKey(),
6             getURI(), relation.getExchange(), observeRelations.getSize());
7       } else {
8         LOGGER.info("successfully established observe relation between {} and
         resource {} ({}, size {})",
9             relation.getKey(), getURI(), relation.getExchange(),
        observeRelations.getSize());
10      }
11      for (ResourceObserver obs : observers) {
12        obs.addedObserveRelation(relation);
13      }
14
15      // CHANGE_START
16      sortedObservers = new ArrayList<ObserveRelation>(observeRelations.
        getRelations().values());
17      Collections.sort(sortedObservers, new Comparator<ObserveRelation>() {
18        public int compare(ObserveRelation o1, ObserveRelation o2) {
19          return (o1).compareToDesc(o2);
20        }
21      });
22      // CHANGE_END
23    }
```

Listing 2.9: CoapResource addObserveRelation(...), codice originale

```
1   @Override
2   public void removeObserveRelation(ObserveRelation relation) {
3     if (observeRelations.remove(relation)) {
4       LOGGER.info("remove observe relation between {} and resource {} ({},
      size {})", relation.getKey(), getURI(),
5           relation.getExchange(), observeRelations.getSize());
6       for (ResourceObserver obs : observers) {
7         obs.removedObserveRelation(relation);
8       }
9       // CHANGE_START
10      sortedObservers.remove(relation);
11      // CHANGE_END
12    }
13  }
```

Listing 2.10: CoapResource removeObserveRelation(...), codice originale

```
1  protected void notifyObserverRelations(final ObserveRelationFilter filter
     ) {
2    notificationOrderer.getNextObserveNumber();
3
4    // CHANGE_START
5    for (ObserveRelation relation : sortedObservers) {
6      if (null == filter || filter.accept(relation)) {
7        relation.notifyObservers();
8      }
9    }
10   // CHANGE_END
11 }
```

Listing 2.11: CoapResource notifyObserverRelations(...), codice originale

2.2.7 server.resources.CoapExchange

Il server può esprimere la sua volontà di avviare la registrazione rispondendo con un diverso valore del campo *QoS* della richiesta. Se invece accetta subito la relazione di osservazione allora risponde con lo stesso valore della richiesta. A tal fine, è stata ridefinita la funzione *CoapExchange.respond(Response response)* aggiungendo il parametro *observeNumber*. Quest'ultimo sovrascrive il valore del campo *Observe* che Califonium inserisce come numero per evitare riordinamento delle notifiche.

```
1  // CHANGE_START
2  public void respond(Response response, int observeNumber) {
3    if (response == null)
4      throw new NullPointerException();
5
6    // set the response options configured through the CoapExchange API
7    if (locationPath != null)
8      response.getOptions().setLocationPath(locationPath);
9    if (locationQuery != null)
10     response.getOptions().setLocationQuery(locationQuery);
11   if (maxAge != 60)
12     response.getOptions().setMaxAge(maxAge);
13   if (eTag != null) {
14     response.getOptions().clearETags();
15     response.getOptions().addETag(eTag);
16   }
17
18   resource.checkObserveRelation(exchange, response);
```

```
19      response.getOptions().setObserve(observeNumber);
20
21      exchange.sendResponse(response);
22    }
```

Listing 2.12: server.resources.CoapExchange, codice originale

2.3 Implementazione

2.3.1 class ProxyObserver

Classe che fornisce i metodi chiamati dal ProxySubject e la gestione delle risorse esposte agli osservatori.

2.3.1.1 Aggiunta delle Risorse

L'aggiunta delle risorse è effettuata dal modulo *ProxySubject* che effettua la discovery delle risorse e le aggiunge al modulo ProxyObserver tramite la funzione 2.13

```
1   public void addResource(SensorNode sensor, String resourceName, boolean
      first) {
```

Listing 2.13: ProxyObserver.addResource

2.3.1.2 Cambio del valore della risorsa

Una volta che il ProxyObserver crea delle relazioni con gli osservatori, deve notificare questi ultimi ogni volta che il valore della risorsa del sensore varia. Quando il ProxySubject riceve il nuovo valore, allora può notificare questo cambiamento al ProxyObserver tramite la funzione 2.14

```
1   public void resourceChanged(SensorNode sensor, String resourceName) {
2     SensorData data = requestValueCache(sensor, resourceName);
3     boolean isCritical = data.getCritic();
4     ObservableResource resource = getResource(sensor, resourceName);
5     Log.info("ProxyObserver",
6         "" + sensor.getUri() + "/" + resourceName + " changed, isCritical:
      " + data.getCritic() + ". Value: "
7             + data.getValue() + ". Current observers: " + resource.
      getObserverCount());
8
9     updateResource(sensor, resourceName, data);
```

```
10      if (!isCritical)
11        resource.changed(new CriticalRelationFilter());
12      else
13        resource.changed();
14    }
```

Listing 2.14: ProxyObserver.resourceChanged

2.3.1.3 Filtro Relazioni Critiche

Nel caso in cui la notifica abbia un valore non critico, questo deve essere inviato solo a quegli osservatori che hanno richiesto un livello di QoS relativo a tutti gli eventi, critici o non critici. Per effettuare questa distizione tra le varie relazioni, si usa l'*ObserveRelationFilter* implementato dalla classe 2.15

```
1  public class CriticalRelationFilter implements ObserveRelationFilter{
2
3    public boolean accept(ObserveRelation relation) {
4      return relation.getQoS() == CoAP.QoSLevel.NON_CRITICAL_LOW_PRIORITY ||
       relation.getQoS() == CoAP.QoSLevel.NON_CRITICAL_MEDIUM_PRIORITY;
5    }
6
7  }
```

Listing 2.15: CriticalRelationFilter

2.3.1.4 Cambio dello stato del sensore

Per avviare l'aggiornamento delle relazioni descritto in 2.2.6.1, il ProxySubject notifica il ProxyObserver di questo cambio tramite la funzione 2.16, in modo tale che possa eliminare le relazioni non più gestibili.

```
1    public void clearObservationAfterStateChanged(String sensorAddress,
       String resourceName, ServerState state) {
2      // Clear all the observaRelation of each resource of that sensor
3      String key = "/" + getUnbrachetAddress(sensorAddress) + "/" +
       resourceName;
4      ObservableResource o = resourceList.get(key);
5      Log.info("ProxyObserver", "Clear relations after sensor state changed:
       " + key + " | " + state);
6      if (state == ServerState.ONLY_CRITICAL) {
7        o.clearAndNotifyNonCriticalObserveRelations(CoAP.ResponseCode.
       FORBIDDEN);
```

```
8       } else if (state == ServerState.UNVAVAILABLE) {
9         o.clearAndNotifyObserveRelations(CoAP.ResponseCode.
        SERVICE_UNAVAILABLE);
10      }
11    }
```

Listing 2.16: clearObservationAfterStateChanged

2.3.1.5 Chiamate alle funzioni del ProxySubject

Il ProxyObserver può chiamare delle funzioni del ProxySubject per:

- Ottenere il nuovo valore della risorsa 2.17
- Richiedere che il ProxySubject effettui una richiesta di osservazione su un sensore se necessario 2.18
- Ottenere le informazioni relative ad un determinato sensore 2.19
- Richiedere la cancellazione di una relazione tra ProxySubject ed un sensore 2.20

```
1   public SensorData requestValueCache(SensorNode sensor, String
      resourceName) {
2     SensorData data = proxySubject.getValue(sensor.getUri(), resourceName);
3     return data;
4   }
```

Listing 2.17: requestValueCache

```
1   public boolean requestRegistration(SensorNode sensor, String resourceName
      , boolean critical) {
2     return proxySubject.newRegistration(sensor, resourceName, critical);
3   }
```

Listing 2.18: requestRegistration

```
1   public SensorNode requestSensorNode(String sensorAddress) {
2     return proxySubject.getSensorNode(sensorAddress);
3   }
```

Listing 2.19: requestSensorNode

```
1  public void requestObserveCancel(Registration registration) {
2    proxySubject.removeRegistration(registration);
3  }
```

Listing 2.20: requestObserveCancel

2.3.2 class ObservableResource extends CoapResource

Questa classe estende una CoapResource standard e ridefinisce i metodo necessari per la gestione delle richieste.

2.3.2.1 handleGET

Questa funzione viene chiamata in 2 casi distinti:

1. il CoapServer riceve una richiesta di tipo **GET** da parte di un osservatore per costruire una relazione di osservazione;
2. è stata chiamata la funzione *changed()* della risorsa e si avvia il meccanismo di notifica dell'osservatore;

Per questo motivo è necessario distinguere la gestione della registrazione dall'invio di una notifica nella stessa funzione.
Innanzitutto, la funzione controlla che non si tratti di una richiesta di cancellazione (campo *Observe*=1), caso in cui viene gestito tutto internamente da Califonium, quindi non bisogna fare nulla, oppure che il sensore non sia attualmente attivo.
Se è si tratta di una richiesta di registrazione, le informazioni relative a quella richiesta vengono salvate. Di queste il *MessageID* della richiesta viene usato per distinguere le notifiche dalla seconda parte della negoziazione. Infatti, nel caso in cui la funzione viene richiamata per effettuare un invio della notifica, il *CoapExchange* conterrà il MessageID della prima richiesta, mentre ciò non succede negli altri casi.

2.3.2.2 handleRegistration

Questa funzione è chiamata per gestire la richiesta di registrazione oppure per completare la negoziazione tra Proxy e osservatore. I due casi sono distinti grazie ad una variabile della classe *ObserveState* che viene settata solo se è stata avviata la negoziazione durante la registrazione. Analizzando ogni caso, si ha:

- Se si tratta della gestione della registrazione, bisogna verificare che il sensore sia capace di gestire una registrazione del genere:
  - Richiesti tutti i valori ma il sensore può mandare solo i valori critici: allora la negoziazione viene avviata rispondendo con **ResponseCode.NOT_ACCEPTABLE** e un valore di QoS gestibile dal sensore nel campo opportuno.
  - altrimenti accetta la registrazione, richiede al ProxySubject la registrazione con il sensore, se necessaria, tramite la funzione 2.18 e risponde con una notifica che contiene lo stesso valore di QoS della richiesta, indicando all'osservatore che la richiesta è stata accettata.
- Se invece si tratta della seconda parte della negoziazione, esegue le stesse operazioni del caso precedente per accettare la registrazione.

2.3.2.3 sendNotification

Questo metodo invia una notifica sfruttando il *CoapExchange* della richiesta. In particolare, prepara la risposta usando i valori ottenuti tramite la funzione 2.17, mentre il campo *Observe* è usato come numero di sequenza se si tratta di una notifica, oppure contiene lo stesso livello di *QoS* della richiesta se si sta accettando una registrazione senza avviare la negoziazione.

Capitolo 3

Modulo ProxySubject

3.1 Descrizione

Questo modulo permette la gestione della comunicazione tra il **Proxy** e i **Subjects** fornendo le informazioni necessarie al modulo **ProxyObserver**.
Le informazioni che dovranno essere mantenute da questo modulo sono tutte quelle necessarie ad evitare comunicazioni non necessarie tra gli osservatori ed i soggetti, in quanto dovrà essere proprio il modulo **ProxySubject** ad offire le informazioni richieste, qualora sia possibile.
In particolare il modulo si occuperà di gestire una struttura dati che dovrà rappresentare il soggeto dal punto di vista dell'osservatore, in particolare offrendo informazioni quali lo stato del soggetto (ovvero quali tipologie di richieste è in grado di accettare), le risorse offerte dal soggetto, in modo tale da non dover sprecare tempo ed energia per richiedere ogni volta se può o meno offrire una detemerminata risorsa su quali risorse il soggetto ha già ricevuto una registrazione, in quanto se un osservatore vuole effettuare una registrazione al proxy su una risorsa a cui un altro osservatore ha fatto la medisima richiesta, con le stesse caratteristiche in termini di priorità, allora quest'ultima non sarà inoltrata al soggetto in quanto il proxy è in grado di poterla gestire in completa autonomia.
In aggiunta il modulo **ProxySubject** offre anche un meccanismo di cache dei valori ricevuti, in modo tale da memorizzare l'ultimo valore ottenuto da un soggetto in una struttura dati, utile al modulo **ProxyObserver** qualore dovesse richiedere un valore ad un determinato soggetto, come ad esempio durante la fase di registrazione.

3.2 Implementazione

3.2.1 CacheTable

Questa classe si occupa di mantenere le informazioni ottenute dai soggetti e di renderle disponibili in qualunque momento.

3.2.1.1 Inserimento Valori

Quando viene ricevuto un nuovo valore dai soggetti è necessario inserire il valore nella cache, controllando se era già stato ricevuto un valore relativo a quella registrazione, in tal caso aggiornando il vecchio valore, oppure se inserire un nuovo valore, in quanto quello appena ricevuto risulta essere il primo valore ricevuto relativo alla registrazione associata ad esso.

```
28  synchronized public boolean insertData(SensorData data){
29    SensorData old = findSensorData(data.getRegistration());
30    // Checking if there is an old data with the same type and coming from
      the same sensor
31    if(old == null){
32      //In this case there isn't any value and so the new value is appended
33      cache.add(data);
34      notifyAll();
35      return true;
36    }
37    //Otherwise the old value is updated
38    old.updateValue(data.getValue(),data.getTime(),data.getObserve(),data.
      getCritic());
39    return false;
40  }
```

Listing 3.1: CacheTable.InsertData

3.2.1.2 Aggiornamento dei valori

Per mantenere consistenti i valori presenti in cache è opportuno che questi ultimi siano periodicamente aggiornati - abbiamo scelto una frequenza di aggiornamento di 1Hz - in quanto uno degli elementi su cui si basa l'intero protocollo è il valore MaxAge appartente ad ogni singolo valore, in quanto nel momento in cui tale valore raggiunge lo 0 e non si è ricevuto alcun valore successivo, allora si deve considerare terminato l'ascolto

verso il soggetto,e non aspettarsi più alcun valore.

```
14   synchronized public ArrayList<Registration> updateTime(int time){
15     ArrayList<Registration> toDelete = new ArrayList<Registration>();
16     for(Iterator<SensorData> i = cache.iterator();i.hasNext();){
17       SensorData d = i.next();
18 //       System.out.println(d.toString());
19       if(d.updateTime(time) == false){
20         i.remove();
21         if(countRegistration(d.getRegistration()) == 1){
22           toDelete.add(d.getRegistration());
23         }
24       }
25     }
26     return toDelete;
27   }
```

Listing 3.2: CacheTable.updateTime

3.2.2 Gestione delle notifiche

Per gestire le notifiche da parte dei soggetti abbiamo continuato ad apporgiarci a Californium, in particolare definendo una classe ResponseHandler come estenesione della CoAPHandler, in quanto erano necessarie alcune operazioni da effettuare subito dopo la ricezione del messaggio.
In particolare la prima operazione da eseguire è quella di discriminare se il messaggio appena ricevuto è critico o meno, risultato ottenuto analizzando il contenuto del messaggio, in quanto la politica di valutazione di un valore è implementata nei soggetti ed ignota al proxy. Questo significa che se al termine del valore è presente il caratte '!' allora significa che il valore ricevuto deve essere gestito come valore critico, altrimenti risulta essere un valore non critico.
Discriminato il valore dovrà essere creato un nuovo **SensorData**, che rappresenta in tutti gli aspetti il valore appena ricevuto, e che dovrà essere inserito in cache. Solo al termine di quest'ultima operazione dovrà esssre informato il modulo **ProxyObserver** in modo che notifichi il nuovo valore a tutti gli osservatori registrati per tale risorsa.
In seguito è riportato solo una parte del codice per la gestione del messaggio, ovvero solo quella ritenuta interessante.

```
41       String Message = response.getResponseText();
```

```
42        double Value;
43        long maxAge = response.getOptions().getMaxAge();
44        boolean critic;
45
46        if(Message.contains("!")) {
47          critic = true;
48          Value = Double.valueOf(Message.substring(0, Message.indexOf("!")));
49        }
50        else {
51          critic = false;
52          Value = Double.valueOf(Message);
53        }
54        Log.info("ResponseHandler", "Ricevuto nuovo valore: " + Value );
55        SensorData newData = new SensorData(this.registration,Value,maxAge,
      response.getOptions().getObserve(),critic);
56        //Log.info("Repsponse Handler", "Inserted data:" +newData.toString())
      ;
57        Log.LogOnFile("LogProxy.csv", newData.getExportLog());
58        cache.insertData(newData);
59        if(this.registration.isFirstValue() == true) {
60          //In this case the only thing to do is to set firstValue at false
61          this.registration.firstValueReceived();
62        }
63        else {
64          //Otherwise we must notify all the observers that a new value has
      arrived
65          SensorNode sensor = registration.getSensorNode();
66          String resourceName = registration.getType();
67          ObservableResource resource = proxyObserver.getResource(sensor,
      resourceName);
68          if (resource.getObserverCount() == 0) {
69            Log.info("ResponseHandler", "No Observe Relations on this
      resource");
70            proxyObserver.requestObserveCancel(registration);
71          }
72          proxyObserver.resourceChanged(sensor, resourceName);
```

Listing 3.3: Parte del metodo ResponseHandler.onLoad

3.2.3 Registrator

Questa classe serve a gestire il processo di registrare, in particolare ad analizzare se quest'ultima risulta essere necessaria o meno, oppure se la vecchia registrazione debba

essere aggiornata.

3.2.3.1 Inserimento di una nuova registrazione

Una volta ricevuta una richiesta di registrazione da parte di un osservatore è necessario verificare se tale registrazione è necessaria o meno, ed eventualmente portarla a termine mediante il metodo messo a disposizione dalla classe **Registration**.
Per controllare se a seguito della richiesta di registrazione ricevuta al proxy, è necessario inoltrarla al soggetto interessato o se invece le registrazioni già presenti riescono a coprire anche la nuova richiesta, evitando quindi spreco di tempo ed energia da parte del soggetto, è necessario analizzare il soggetto relativo a tale registrazione, la risorsa richiesta e il livello di criticità desiderato: se è già presente una registrazione avente lo stesso soggetto, associata alla stessa risorsa e con un livello di criticità minore rispetto a quello richiesto - in quanto se per esempio venisse fatta una richiesta solo per valori critici ma ce ne fosse una che li copre tutti, allora non sarebbe necessaria adempiere alla richiesta appena arrivata - allora il processo di registrazione verrbbe completato senza consultare il soggetto, in quanto per quest'ultimo non cambierebbe niente. In alternativa se è presente una registrazione con un livello maggiore rispetto a quello richiesto - si prenda ad esempio il caso opposto rispetto a quello illustrato prima - la vecchia registrazione non sarà sufficiente e quindi dovrà essere aggiornata in modo da coprire anche la richiesta appena arrivata.

```
13  synchronized public int newRegistration(Registration _r){
14    int registrationNeeded = this.RegistrationNeeded(_r);
15    if( registrationNeeded == 1){
16      Log.info("Registrator", "New registration needed");
17      boolean result = _r.register();
18      //boolean result = true;
19      if(result){
20        this.reg.add(_r);
21        return 1;
22      }
23      else
24        return -1;
25    }
26    else if (registrationNeeded == 2) {
27      Log.info("Registrator","Updating old registration");
28      //System.out.println("Registrator: Aggiornamento registrazione");
```

```
29        Registration r = findAssociate(_r);
30        this.removeRegistration(r);
31        boolean result = _r.register();
32        //boolean result = true;
33        if(result){
34          this.reg.add(_r);
35          return 2;
36        }
37        else
38          return -1;
39      }
40      else{
41        Log.info("Registrator", "New registration not needed");
42        return 0;
43      }
44    }
45    synchronized private int RegistrationNeeded(Registration newRegistration)
        {
46      for (Registration r: reg){
47        if(r.equals(newRegistration)) {
48          return 0;
49        } else if(r.getSensorNode().toString().equals(newRegistration.
      getSensorNode().toString()) && r.getType() == newRegistration.getType()
      ) {
50          if( newRegistration.isCritic() == false && r.isCritic() == true ) {
51            return 2;
52          }
53          else if (r.isCritic() == newRegistration.isCritic() || (
      newRegistration.isCritic() == true && r.isCritic() == false ) ) {
54            return 0;
55          }
56        }
57      }
58      return 1;
59    }
```

Listing 3.4: Registrator.newRegistration e Registrator.RegistrationNeeded

3.2.3.2 Rimozione di una registrazione

Qualora non si ricevessere più notifiche da parte del soggetto o se tutti gli osservatori - in quanto basta almeno un osservatore interessato per mantenere attiva la registrazione - non desiderassero ricevere più notifiche, risulta necessario rimuovere la registrazione e

notificare al soggeto che non siamo più registrati per tale risorsa.

```
69  synchronized public void removeRegistration(Registration _r){
70    if(reg.contains(_r)) {
71      reg.remove(_r);
72      _r.sendCancelation();
73      synchronized(_r) {
74        _r.notify();
75      }
76      Log.info("Registrator", "Registration requested removed");
77    }
78    else {
79      Log.info("Registrator", "Registration requested not found");
80    }
81  }
```

Listing 3.5: Registrator.removeRegistration

3.2.4 Virtualizzare soggetti e messaggi

Un aspetto molto importante del modulo **ProxySubject** è la capacità di virtualizzare sia i soggetti che i messaggi ricevuti da essi, in modo da rendere completamente trasparente il comportamento di nodi sensori per il modulo **ProxyObserver** e quindi di conseguenza per gli osservatori.
Tale scelta di sviluppo ha comportato la possibilità di sviluppare in modo indipendente i protocolli di comunicazione con gli osservatori da una parte e con i soggetti dall'altra, in quanto l'unica cosa a comune risultano essere proprio le strutture dati che *virtualizzano* i soggetti e i messaggi ricevuti da essi.

3.2.4.1 SensorNode

Questa classe è quella che si occupa di gestire un soggetto, mantenendo le informazioni necessarie, il suo URI, le risorse che mette a disposizione, il suo livello di batteria e lo stato in cui opera, che può essere:

- **UNAVAILABLE**: Il sensore non è raggiungibile in quanto le batterie sono scariche.
- **ONLYCRITICAL**: Il livello di batteria è troppo basso quindi il soggetto accetta solo registrazioni critiche.

- **AVAILABLE**: Il livello di batteria è sufficientemente alto per gestire qualsiasi tipo di registrazione

```
12  volatile String IPaddress;
13  volatile int Port;
14  volatile double battery;
15  volatile ServerState actualState;
16  ArrayList<String> resources;
```

Listing 3.6: Attributi offerti e gestiti dalla clase SensorNode

In particolare per la gestione della batteria è stato scelto di adottare un approccio che prevede l'invio di nuovi aggiornamenti da parte del soggetto solo qual'ora il livello della batteria sia inferiore alla soglia critica, in modo da ridurre il numero di messaggi inviati dal soggetto stesso, prolungando la durata della sua batteria. Ricevuto il valore poi sarà carico del proxy aggiornare lo stato del soggetto *virtuale* in modo che sia consistente con lo stato del relativo soggetto reale.

```
42  synchronized public ServerState updateBattery(double newBatteryValue,
      ProxyObserver po) {
43    battery = newBatteryValue;
44    if(battery <= 30)  {
45      if(battery <= 0) {
46        Log.info("SensorNode", "Sensor Dead");
47        actualState = ServerState.UNVAVAILABLE;
48      } else {
49        actualState = ServerState.ONLY_CRITICAL;
50        Log.info("SensorNode", "Battery Under Threshold");
51      }
52      for (String resource: resources) {
53        if ( !resource.equals("battery"))
54          po.clearObservationAfterStateChanged(getUri(), resource,
      actualState);
55      }
56    }
57    return actualState;
58
59  }
```

Listing 3.7: SensorNode.UpdateBattery

3.2.4.2 SensorData

Questa classe si occupa invece di gestire e rendere facilmente accessibili i messaggi ottenuti dai soggetti, con le relative informazioni di contorno, come il soggetto mittente, la registrazione a cui quel valore è associato, il maxage del messaggio e se è o meno un messaggio critico.

```
7   volatile double value;
8   volatile long maxAge;
9   volatile boolean critic;
10  volatile Registration registration;
11  volatile long observe;
```

Listing 3.8: Attributi gestiti e offerti dalla classe SensorData

Capitolo 4

Observer

4.1 Descrizione

Questo programma permette di usare un CoapClient e interagire con i sensori tramite il Proxy utilizzando una shell che dispone dei seguenti comandi:

1. Stampa il menu di aiuto
2. Richiedi una registrazione selezionando tra quelle disponibili, specificando la priorità e la volontà ad accettare una eventuale proposta del sensore durante la negoziazione
3. Richiedi la cancellazione di una relazione
4. Avvia la discovery delle risorse

4.2 Modifiche a Californium lato Client

Al fine di implementare le modifiche effettuate al protocollo CoAP, è stato necessario modificare anche la libreria Californium. In seguito verranno elencati i cambiamenti al codice della libreria, indicando il nome del file con un link alla repository con il codice originale e la motivazione di tale modifica. I file fanno riferimento alla package core di Californium:

4.2.1 CoapClient

Con l'introduzione della negoziazione del livello di priorità è necessario effettuare ulteriori controlli alla risposta ricevuta, riga 28 di 4.1. Nel caso in cui la negoziazione è

stata avviata, la risposta avrà come *ResponseCode* **NOT_ACCEPTABLE** e la CoapObserveRelation appena creata viene cancellata. Se la registrazione viene accettata senza negoziazione, allora bisogna controllare che il campo *Observe* della risposta coincida con quello della richiesta.

```
1  public CoapObserveRelation observeAndWaitNegotiation(Request request,
     CoapHandler handler) throws InterruptedException {
2
3    if (request.getOptions().hasObserve()) {
4      assignClientUriIfEmpty(request);
5      Endpoint outEndpoint = getEffectiveEndpoint(request);
6      CoapObserveRelation relation = new CoapObserveRelation(request,
     outEndpoint);
7      // add message observer to get the response.
8      ObserveMessageObserverImpl messageObserver = new
     ObserveMessageObserverImpl(handler, request.isMulticast(),
9          relation);
10     request.addMessageObserver(messageObserver);
11     // add notification listener to all notification
12     NotificationListener notificationListener = new Adapter(
     messageObserver, request);
13     outEndpoint.addNotificationListener(notificationListener);
14     // relation should remove this listener when the request is cancelled
15     relation.setNotificationListener(notificationListener);
16     CoapResponse response = synchronous(request, outEndpoint);
17     // CHANGE_START
18     synchronized(relation) {
19       if ( !relation.getResponseReceived() ) {
20         relation.setMainWaiting(true);
21         relation.wait();
22         relation.setMainWaiting(false);
23       }
24     }
25     if (!response.getCode().equals(CoAP.ResponseCode.NOT_ACCEPTABLE))
26       relation.resetOrder();
27
28     if (response == null || !response.advanced().getOptions().hasObserve
     ()
29         || response.getCode().equals(CoAP.ResponseCode.NOT_ACCEPTABLE) //
      Negotiation started
30         || (int) response.getOptions().getObserve() != (int) request.
     getOptions().getObserve() // Requested observe # doesn't match the
     response's one
```

```
31            ) {
32          // CHANGE_END
33          relation.setCanceled(true);
34        }
35        return relation;
36      } else {
37        throw new IllegalArgumentException("please make sure that the request
      has observe option set.");
38      }
39    }
```

Listing 4.1: CoapClient, codice originale

Dopo la prima notifica ricevuta, il contatore, usato per mantenere ordinate le notifiche, viene settato con il valore del campo *Observe* appena ricevuto e la prossima notifica deve avere un valore superiore a quest'ultimo per essere considerata fresca ed essere accettata. Se la registrazione è accettata subito, il contatore ottiene uno dei 4 valori del campo *QoSLevel* e difficilmente la prossima notifica ha un numero di superiore a questi. Pertanto il contatore deve essere resettato in modo tale che la prossima notifica non venga scartata. Quando si riceve la risposta possono attivarsi 2 thread:

- MainThread che continua l'esecuzione della observeAndWaitNegotiation subito dopo della *synchronous*, in cui viene effettuato il reset del contatore
- Coap.Endopoint-UDP che fornisce la risposta all'Handler della risposta stessa, il quale prima controlla che la notifica sia fresca e scrive nel contatore il valore attuale del campo *Observe*.

É necessario garantire che i 2 thread vengano eseguiti nel seguente ordine:

1. Coap.Endpoint-UDP scrive nel contatore il valore del campo *Observe*
2. MainThread resetta il contatore

Per garantire ciò, il MainThread attende tramite una *wait* (riga 21 di 4.1) sulla *CoapObserveRelation* che la risposta è stata gestita dal Coap.Endpoint che esegue una *notify* sullo stesso oggetto dopo che la freschezza della notifica è stata controlla nella funzione *deliver* 4.2. Inoltre, l'attesa è effettuata solo se la risposta non è stata ancora gestita, mentra la *notify* è effettuata solo se il MainThread è in attesa. Questi controlli sono effettuati tramite delle nuove variabili introdotti nella *CoapObserveRelation* indicate in 4.3

```
1   /**
2    * Checks if the specified response truly is a new notification and if, invokes
3    * the handler's method or drops the notification otherwise. Ordering and
4    * delivery must be done synchronized here to deal with race conditions in the
5    * stack.
6    */
7   @Override
8   protected void deliver(CoapResponse response) {
9     synchronized (relation) {
10      if (relation.onResponse(response)) {
11        // CHANGE_START
12        relation.setResponseReceived();
13        if ( relation.getMainWaiting() ) {
14          relation.notify();
15        }
16        // CHANGE_END
17        handler.onLoad(response);
18      } else {
19        LOGGER.debug("dropping old notification: {}", response.advanced());
20        return;
21      }
22    }
23  }
```

Listing 4.2: deliver, codice originale

4.2.2 CoapObserveRelation

Variabili usate nel meccanismo di *wait* e *notify* spiegato in 4.2.1.

```
1  // CHANGE_START
2  private AtomicBoolean responseReceived;
3  private AtomicBoolean isMainWaiting;
4  // CHANGE_END
```

Listing 4.3: CoapObserveRelation, codice originale

```
1  //CHANGE_START
```

```
2   protected CoapObserveRelation(Request request, Endpoint endpoint) {
3     this.request = request;
4     this.endpoint = endpoint;
5     this.orderer = new ObserveNotificationOrderer();
6     this.responseReceived = new AtomicBoolean(false);
7     this.isMainWaiting = new AtomicBoolean(false);
8   }
9
10  protected void setResponseReceived() {
11    this.responseReceived.set(true);
12  }
13
14  protected boolean getResponseReceived() {
15    return this.responseReceived.get();
16  }
17
18  protected void setMainWaiting( boolean state) {
19    this.isMainWaiting.set(state);
20  }
21
22  protected boolean getMainWaiting() {
23    return this.isMainWaiting.get();
24  }
25  // CHANGE_END
```

Listing 4.4: CoapObserveRelation, codice originale

Per resettare il gestore dell'ordine delle notifiche il contatore viene settato a 0.

```
1   // CHANGE_START
2   protected void resetOrder() {
3     this.orderer.resetNumber();
4   }
5   // CHANGE_END
```

Listing 4.5: CoapObserveRelation, codice originale

4.3 Implementazione

4.3.1 classe Observer

Questa classe utilizza un *CoapClient* per effettuare le richeste verso il Proxy. Inoltre, mantiene le informazioni relative alle risorse trovate durante la discovery e una lista delle

relazioni attualmente attive.

4.3.1.1 resourceDiscovery

Effettua la discovery delle risorse disponibili sul Proxy al quale viene inviata una richiesta di tipo **GET** sulla risorsa *<well-known>* che contiene la lista delle risorse presenti.

```
1  public void resourceDiscovery() {
2    Log.info("Observer", "Start Resource Discovery");
3    Set<WebLink> weblinks = observerCoap.discover();
4    resourceList.clear();
5    resourceList.addAll(weblinks);
6    Log.info("Observer", "Resources found: " + resourceList.toString());
7  }
```

Listing 4.6: ResourceDiscovery

4.3.1.2 resourceRegistration

Prepara una richiesta di tipo **GET** contenente nel campo *Observe* il valore specificato dall'utente e la invia utilizzando la funzione 4.1. Nel caso in cui la *CoapObserveRelation* venga costruita con successo, allora questa viene mantenuta in una lista in modo da poter essere usata in seguito per la cancellazione della relazione.

```
1  private void resourceRegistration(String resourceName, int priority,
     String path, boolean acceptProposal) {
2    Request observeRequest = new Request(Code.GET);
3    try {
4      // Set the priority level using the first 2 bits of the observe
     option value
5      observeRequest.setObserve();
6      observeRequest.setOptions(new OptionSet().addOption(new Option(
     OptionNumberRegistry.OBSERVE, priority)));
7    } catch (IllegalArgumentException ex) {
8      System.out.println("Invalid Priority Level");
9    }
10
11   String URI = "coap://" + this.ipv4Proxy + ":" + this.portProxy + path;
12   observeRequest.setURI(URI);
13   Log.info("Observer", "Request observation of " + path + " with priority
      " + getPriority(priority));
14   CoapObserveRelation relation = observerCoap.observeAndWaitNegotiation(
     observeRequest,
```

```
15          new ResponseHandler(this, priority, path, URI, acceptProposal,
        DEBUG));
16
17      if (relation.isCanceled()) {
18        Log.info("Observer", "Relation has been canceled or the negotiation
        started");
19      } else
20        relations.put(path, relation);
21    }
```

Listing 4.7: ResourceRegistration

4.3.1.3 resourceCancellation

Effettua la richiesta di cancellazione di una relazione inviando al Proxy una richiesta **GET** sulla risorsa di cui non si vuole più ricevere le notifiche, specificando nel campo *Observe* il valore 1.

```
1   private void resourceCancellation(String path) {
2
3     CoapObserveRelation relation = relations.get(path);
4     if (relation == null) {
5       Log.error("Observer", "Observe relation on " + path + " not found");
6       return;
7     }
8     Log.info("Observer", "Proactive cancel of " + path + " sent");
9     relation.proactiveCancel();
10  }
```

Listing 4.8: ResourceCancellation

4.3.2 classe ResponseHandler implements CoapHandler

Questa classe implementa l'interfaccia *CoapHandler* che gestisce le risposte ricevute in seguito all'invio di una richiesta. In particolare, ad ogni risposta ricevuta, la funzione *onLoad(CoapResponse response)* viene eseguita.

4.3.2.1 onLoad

Inizialmente la funziona effetta dei controlli sulla correttezza della risposta:

- response = *null*: la risposta è vuota e viene scartata

- **ResponseCode.FORBIDDEN**: la relazione è stata interrotta dal server in seguito ad un cambio di stato da **AVAILABLE** a **ONLY_CRITICAL**
- **ResponseCode.SERVICE_UNAVAILABLE**: la relazione è stata interrotta dal server in seguito ad un cambio di stato da **ONLY_CRITICAL** a **UNAVAILABLE**
- **ResponseCode.NOT_FOUND**: se viene richiesta una risorsa al Proxy, ma quest'ultimo non conclude con successo la registrazione con il sensore
- risposta senza opzione *Observe*: la risposta non è valida e viene scartata

In seguito controlla se la risposta contiene una notifica oppure sia l'inizio di una negoziazione:

- **ResponseCode.CONTENT**: la risposta contiene il nuovo valore della risorsa che viene stampa a video
- **ResponseCode.NOT_ACCEPTABLE**: la negoziazione è stata avviata, allora se l'accettazione delle proposte è abilitata, viene preparata un'altra richiesta **GET** contenente nel campo *Observe* il valore proposto dal Proxy, inviata tramite una semplice *observe(CoapRequest, CoapHandler)* a cui si passa la stessa istanza di questa classe come CoapHandler.

Capitolo 5

Subject

5.1 Descrizione

Questo è il firmware che verrà eseguito sui *nodi sensore*, i quali si comporteranno come dei CoapServer, avviando un server rest, in ascolto sulla porta di default di CoAP 5683, gestendo le richieste ricevute dal *Proxy*, che possono essere:

- Discovery delle risorse presenti sul nodo
- Registrazione ad una risorsa con uno specifico tipo di priorità
- Cancellazione dall'osservazione di una risorsa

Il nodo sensore, solo nel caso in cui è registrato almeno un *Observer* ad una risorsa, si occuperà di eseguire il sensing e l'invio di quest'ultima nel caso in cui il valore sia cambiato o il vecchio valore stia per scadere (questo secondo caso è gestito per evitare di perdere la registrazione dell'*Osservatore*).
Il nodo sensore si occupa anche di connettersi e di mantenere viva la connessione col *Border Router*, utilizzato dal *Proxy* per comunicare con i nodi sensore.

5.2 Modifiche al Border Router

Per poter compilare il codice del border router fornito con contikiOS per dispositivi di tipo SkyMote è stato necessario disabilitare il processo che esegue il web server in quanto il dispositivo non dispone di memoria sufficiente per poter utilizzare questa funzione; la quale veniva utilizzata semplicemente con lo scopo di far vedere all'utente le rotte presenti al momento e la lista dei vicini tramite interfaccia web, quindi rimuovendo questo

processo non vengono rimosse funzionalità necessarie ad un router.
Per rimuovere la funzionalità è bastato ridefinire WEBSERVER uguale a 0.

```
64 #define WEBSERVER 0
65 #if WEBSERVER==0
66 /* No webserver */
67 AUTOSTART_PROCESSES(&border_router_process);
```

Listing 5.1: BorderRouter, codice originale

5.3 Implementazione

5.3.1 Process Rest Server

Processo principale dei nodi sensore, si occupa di:

1. Ottenere un indirizzo IP connettendosi al border-router
2. Rendere disponibili le risorse presenti attivandole
3. Attivare i sensori relativi alle singole risorse

```
38 PROCESS_THREAD(rest_server, ev, data){
39   PROCESS_BEGIN();
40
41   /*
42    * Initializing IP address and connecting to the border router
43    */
44
45   PROCESS_PAUSE();
46
47   set_global_address();
48
49   /* new connection with remote host */
50   client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
51   if(client_conn == NULL) {
52     PRINTF("No UDP connection available, exiting the process!\n");
53     PROCESS_EXIT();
54   }
55   udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
56
57   UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));
58
```

```
59
60   /*
61    * Starting Erbium Server
62    */
63
64   PRINTF("Starting Erbium Server\n");
65
66   /* Initialize the REST engine. */
67   rest_init_engine();
68
69   /* Activate the application-specific resources. */
70   rest_activate_resource(&res_battery, "sensors/battery");
71   SENSORS_ACTIVATE(battery_sensor);
72
73   rest_activate_resource(&res_temperature, "sensors/temperature");
74   SENSORS_ACTIVATE(temperature_sensor);
75
76   //Resource used only for debug purposes
77   rest_activate_resource(&res_hello, "sensors/hello");
78
79   //Used only for Testing phase
80   printf("Time,IPAddress,Value,Type,Critic,Observe\n");
81
82
83
84   SENSORS_ACTIVATE(button_sensor);
85   while(1) {
86     PROCESS_WAIT_EVENT_UNTIL(ev==sensors_event && data==&button_sensor);
87     //Used to force the battery to go in the only critic connection
       accepted phase
88     critic_battery();
89   }
90
91
92
93   PROCESS_END();
94 }
```

Listing 5.2: Process RestServer

5.3.2 Risorse

Per avere una maggiore modularità è stato realizzato un file per ogni singola risorsa e risorse sono state definite come variabili di tipo **extern**

```
4 /*Defining the resources that are present in the sensor node*/
5 #include "dev/temperature-sensor.h"
6 extern resource_t res_temperature;
7
8 #include "dev/battery-sensor.h"
9 extern resource_t res_battery;
10
11 //For debug purposes
12 extern resource_t res_hello;
```

Listing 5.3: Risorse presenti sul nodo

5.3.2.1 Batteria

La batteria viene fornita come risorsa di tipo osservabile, l'unico osservatore che si iscrive per ricevere le notifiche relative a questa risorsa è il *Proxy*, in quanto se la batteria del sensore si trova sotto una specifica soglia vengono accettate soltanto richieste con un livello di priorità critico.
Il sensing del livello della batteria viene fatto periodicamente Le notifiche della batteria vengono inviate solo in due casi:

1. Valore della batteria è sotto la soglia critica
2. L'ultimo valore sta per scadere

5.3.2.2 Temperatura