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

2.2 Implementazione

2.2.1 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:

2.2.1.1 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

2.2.1.2 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

2.2.1.3 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

2.2.1.4 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

2.2.1.5 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

2.2.1.6 CoapResource

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

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