STUDIO DI INTEROPERABILITÀ TRA COAP E HTTP TRAMITE L'USO DI UN PROXY: IMPLEMENTAZIONE E SPERIMENTAZIONE

RELATORE: Prof. Michele Zorzi

CORRELATORE: Dott. Angelo Castellani

LAUREANDO: Simone Pinton

A.A. 2010-2011

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

STUDIO DI INTEROPERABILITÀ TRA COAP E HTTP TRAMITE L'USO DI UN PROXY: IMPLEMENTAZIONE E SPERIMENTAZIONE

RELATORE: Prof. Michele Zorzi

CORRELATORE: Dott. Angelo Castellani

CORRELATORE: Dott. Mattia Gheda

LAUREANDO: *Simone Pinton*

Padova, 24 ottobre 2011

Indice

Introduzione

Ci troviamo agli albori di una nuova era di Internet. Finora la rete è stata pensata, disegnata e creata per collegare tra loro computer: macchine create per il calcolo, con grandi capacità di elaborazione e di memoria. La rete stessa, per non essere il collo di bottiglia di queste macchine, è stata disegnata per grandi capacità di trasferimento e per la migliore affidabilità possibile nella trasmissione: agli svariati standard per sfruttare il tradizionale cavo di rame sono succedute le fibre ottiche, mentre si studiano standard sempre più performanti per le connessioni wireless.

Parallelamente allo sviluppo di Internet, stiamo assistendo all'evoluzione di una diversa tecnologia di comunicazione che, a differenza della precedente, è caratterizzata da risorse limitate sia in termini di velocità di trasmissione che di capacità di processamento. Tale tecnologia prende il nome di *rete constrained* con riferimento alla limitatezza di risorse. Generalmente le reti constrained sono reti wireless, con un elevata densità dei nodi nello spazio, e con nodi che non sempre sono fissi: si parla quindi del caso peggiore per quanto riguarda le trasmissioni wireless. In aggiunta, generalmente anche i nodi non hanno grandi capacità di calcolo o di memoria, o perché devono essere di dimensioni ridotte, o per ribassare il costo del singolo pezzo e compensare la spesa di disporre questi sensori nello spazio con elevata densità, oppure perché devono avere bassi consumi, in ogni caso auspicabili, ma ancor più quando si parla di apparecchi senza fili.

Il futuro invece guarda proprio alle reti constrained, immaginando un mondo in cui ogni apparecchiatura elettronica diventa potenzialmente accessibile da Internet. In particolare guarda alle reti di sensori: disporre dei sensori in gran numero all'interno dell'ambiente che si vuole monitorare aiuta sicuramente ad avere una visione più chiara della realtà. Mettendo in comunicazione questi sensori con altre macchine che agiscono sull'ambiente o comunque su altri nodi, significa automatizzare un enorme numero di operazioni, comportando una migliore qualità

della vita, nonché numerosi risparmi monetari dovuti alle automatizzazioni e alle ottimizzazioni che si possono fare. Il limite degli esempi che si possono fare è solo l'immaginazione.

Il problema quindi diventa inserire le reti constrained all'interno di Internet. Le tecnologie finora create, essendo state pensate per un altro tipo di rete con specifiche completamente diverse, non sono adatte ad una rete constrained, anzi alcune di esse renderebbero inagibile o impossibile l'uso della rete. La direzione che si sta attualmente prendendo è quindi sviluppare nuove tecnologie e protocolli specifici per le reti constrained.

Il lavoro descritto in questo elaborato si inquadra in questo contesto e costituisce uno degli elementi fondamentali per l'unione tra le due tipologie di reti. Anche il protocollo HTTP (HyperText Transfer Protocol), che è alla base dell'architettura client-server di Internet, non è ottimale per una rete constrained. All'interno dell'IETF (Internet Engineering Task Force) si sta lavorando per proporre un nuovo protocollo chiamato CoAP (Constrained Application Protocol) per portare l'architettura client-server nelle reti constrained.

Ma la creazione di nuovi protocolli unita al bisogno di connettere queste reti constrained ad Internet fa nascere la necessità di nodi intermedi alle due reti che traducano questi protocolli. In questo documento si descrive la creazione di un proxy che converte una richiesta HTTP proveniente da Internet in una richiesta CoAP diretta ad una rete di sensori, e ovviamente gestisce anche la risposta.

Questa creazione non parte dal nulla: in particolare lo stesso proxy non viene costruito ex novo ma a partire da Squid, il proxy open source più usato e rinomato. Sfruttando il fatto che Squid è un software libero e open source, non è stato necessario creare un nuovo proxy da zero, ma è bastato modificare alcune parti del codice sorgente in modo che Squid riconosca una richiesta diretta ad una rete che supporta CoAP anziché ad una normale rete Internet e la gestisca di conseguenza, convertendo i protocolli della rete Internet nei protocolli di una rete constrained, e facendo ovviamente l'operazione inversa alla ricezione della risposta. Il lavoro che deve fare il proxy non è però una semplice traduzione: deve anche gestire le diversità di regole di standard diversi, specie per quanto riguarda le perdite dei pacchetti e le ritrasmissioni. Sfruttando il nuovo modulo TPROXY del kernel Linux, il software è stato configurato per lavorare anche come transparent proxy: in questo modo il proxy intercetta i pacchetti diretti ad

una rete CoAP in modo trasparente rispetto al client. Data la dinamicità delle definizioni dei protocolli in Internet e la velocità di sviluppo di nuovo tecnologie, questo lavoro di tesi è stato focalizzato sullo sviluppo e sul test delle funzionalità principali, senza la pretesa di includere tutte le opzioni delle stesse.

Alla fine di questo lavoro il software viene testato: si testa sia il suo funzionamento base (cioè in condizioni ideali della rete), sia il suo funzionamento in caso di perdite dei pacchetti, o in caso di risposte deferred provenienti dalla rete constrained: una nuova modalità di comunicazione prevista da CoAP ma non da HTTP in cui un server può rimandare l'invio di una risposta perché non è pronta. Si mostrano infine i risultati di alcuni test prestazionali, fatti creando un testbed virtuale e simulando le limitazioni di una rete constrained: i test vogliono confrontare le prestazioni di CoAP e HTTP all'interno di una rete constrained.

Nel primo capitolo di questo elaborato si illustra più in dettaglio cos'è una rete constrained e perché in questi anni gli studi si stanno concentrando su queste reti. Si mostra poi quali sono i nuovi protocolli e qual è la nuova architettura proposta per le reti constrained. Si spiega infine perché c'è bisogno di un proxy che faccia da tramite tra la rete Internet ed una rete constrained.

Nel secondo capitolo si parla più in dettaglio dei punti di partenza su cui si è basato il lavoro di modifica del codice sorgente fatto su Squid. Si parla quindi del protocollo HTTP e del protocollo CoAP (per quest'ultimo si prende come riferimento la draft 4), del modulo del kernel Linux TPROXY che permette la configurazione di un transparent proxy, e si parla brevemente del funzionamento interno del proxy Squid.

Il terzo capitolo elenca e spiega le modifiche fatte a Squid in modo che possa gestire una richiesta proveniente da Internet e diretta ad una rete constrained che supporta CoAP. In particolare vengono elencate le modifiche fatte al parsing del file di configurazione, alla riscrittura degli URL delle richieste, all'interfacciamento con le funzioni che gestiscono i pacchetti CoAP, al riconoscimento di un pacchetto diretto ad una rete CoAP, e alla gestione vera e propria degli invii e delle ricezioni.

Il quarto capitolo documenta i test fatti sul prodotto finito. Si tratta di test che verificano che tutte le funzionalità introdotte lavorino correttamente, e poi di test che verificano le prestazioni del protocollo CoAP, confrontando il suo funzionamento con quello di HTTP all'interno della stessa rete constrained.

Capitolo 1

Le reti constrained

Per capire le motivazioni che hanno spinto a creare il nuovo protocollo CoAP, e in seguito l'utilità di un proxy tra HTTP e CoAP, bisogna prima conoscere i concetti di *Internet of Things* (IoT), di *reti constrained* e di *Wireless Sensor Network* (WSN), nonché i recenti sviluppi della ricerca riguardo a questi due argomenti. Il seguente capitolo darà appunto una visione generale riguardo a questi argomenti.

1.1 Internet of Things

L'*Internet of Things* (IoT) è sostanzialmente un nuovo modo di vedere la rete. É comparabile alla rivoluzione fatta negli anni '80, in cui i computer sono diventati da macchine che lavorano da sole a macchine che lavorano in connessione (più o meno stretta) con altre macchine. In questo caso la rivoluzione sta in quali macchine devono essere connesse ad Internet: se finora la rete è stata utilizzata prevalentemente dai computer veri e propri intesi come macchine progettate e adibite per il calcolo, la visione del futuro è quella di una Internet che connette ogni oggetto che possiede al suo interno un apparecchio elettronico embedded.

Le applicazioni immaginabili sono infinite. Anche senza cadere nel futuristico, si possono immaginare applicazioni nella domotica (controllare gli elettrodomestici anche se ci si trova lontani da casa), nel risparmio energetico (dei sensori ambientali che controllano l'impianto di riscaldamento/condizionamento di un ambiente), nella sanità (monitoraggio di pazienti a distanza), nei trasporti (automezzi che comunicano coi vicini automezzi), nella localizzazione (conoscere sem-

pre dove si trova una persona, oppure automatizzare una operazione all'ingresso o all'uscita di una certa persona in un ambiente).

L'Internet of Things punta quindi ad avere un accesso ad Internet in ogni apparecchio elettronico. La prima conseguenza è la possibilità di controllare da remoto lo stato di ognuno di questi oggetti, per la sola lettura o anche per la modifica. L'interazione macchina-uomo aumenterebbe di molto, e grazie a Internet sarebbe una interazione non solo diretta o locale, ma possibile da ogni parte del mondo. Se poi ad accedere allo stato stato dei dispositivi non è solo l'uomo ma sono altre macchine, anche l'interazione macchina-macchina diventa totale, e costruendo macchine che a seconda di alcune letture di altri dispositivi controllano e modificano lo stato di altri dispositivi, una enorme quantità di operazioni potrebbe venire automatizzata.

1.2 Constrained networks

Conseguenza naturale dell'Internet of Things è la nascita delle *Constrained Networks*. Esse sono reti in cui almeno alcuni dei nodi e dei collegamenti tra loro hanno delle limitazioni.

- La caratteristica dei nodi di essere dispositivi embedded li può limitare nella potenza di calcolo o nella capacità di memoria. I dispositivi embedded infatti devono essere nella maggior parte dei casi di dimensione ridotta, ma soprattutto devono avere bassi costi di produzione.

- Nella maggior parte dei casi si parla poi di reti wireless: questo porta a limitazioni nel throughput e nella lunghezza del pacchetto, e ad un alto grado di perdita dei pacchetti. Questi problemi si accentuano se si pensa che i nodi sono sparsi nello spazio con un'alta densità, aumentando così la possibilità di interferenze, e che i nodi possono essere mobili, creando difficoltà nel routing dei pacchetti nella rete.

- I nodi possono avere anche delle limitazioni in fatto di potenza disponibile, specie se si parla di nodi wireless che hanno quindi una alimentazione a batteria, che possibilmente non deve esaurirsi subito a causa della trasmittente. Ma anche in caso di alimentazione diretta, il fatto di avere un'alta densità di nodi che consumano continuamente molta corrente può diventare

oneroso dal punto di vista economico. Ecco perché la potenza di calcolo deve essere ridotta e le trasmissioni devono essere a bassa potenza. Un altro modo di risparmiare energia è quello di lasciare accesa la trasmittente solo per brevi periodi: la rete deve essere capace di inserire e disinserire nodi in ogni momento.

1.3 Wireless sensor networks

Le *Wireless Sensor Networks (WSN)*, italianizzate a *reti di sensori*, sono un sottotipo e un esempio di reti constrained. Si tratta di reti costituite da un grande numero di nodi (anche centinaia o migliaia) posizionati in maniera sparsa nello spazio, o anche mobili, interconnessi tra loro via radio. Ovviamente il gran numero comporta una maggior precisione nella conoscenza della realtà circostante, ma aumenta anche la difficoltà delle trasmissioni radio per via delle interferenze, oltre ovviamente ai costi hardware ed energetici.

I sensori possono essere di ogni tipo. A seconda delle componenti montate al loro interno, possono servire a misurare cose diverse (temperatura, umidità, distanza da altri sensori, e infinite altre cose), possono essere di diversa precisione, con diverse capacità di calcolo, quindi di diversi costi. Una caratteristica fondamentale che possiede una rete di sensori è quindi l'eterogeneità. Il caso più comune è però quello di sensori a bassa capacità di calcolo e di memoria.

Ma la risorsa di cui più scarseggiano i sensori è l'energia: di solito i sensori sono wireless, quindi alimentati a batteria. Oltre ai problemi di costo energetico già citati, per dei sensori bisogna aggiungere il fatto che solitamente essi sono dislocati in aree non facilmente accessibili: non è detto quindi che cambiare la batteria sia una operazione di routine. I sensori devono essere progettati per il minor consumo energetico possibile, e questo riguarda anche la progettazione del software e il sistema delle trasmissioni radio.

Di solito i nodi sono dislocati in modo da formare un grafo; non è detto che questo grafo sia fisso nel tempo: i nodi possono essere mobili (ad esempio nel caso della localizzazione) o possono entrare o uscire di continuo dalla rete (per il risparmio energetico, per la loro mobilità, ma anche a causa di un guasto); la topologia della rete è quindi dinamica. Poiché nel caso più comune i nodi non comunicano tutti fra loro (quindi il grafo della rete non è completo), per via delle

considerazioni energetiche appena fatte che limitano il range di ogni nodo, ma anche per via delle numerose interferenze che ne seguirebbero, se un nodo vuole comunicare con un altro nodo lontano deve appoggiarsi ad altri nodi che devono inoltrare il pacchetto. Alle reti di sensori serve quindi anche un algoritmo di routing che sia dinamico ma anche che non inondi la rete di trasmissioni.

Le reti di sensori all'interno dell'Internet of Things avrebbero infiniti campi di applicazione. Solo il fatto di avere un numero elevato di sensori dislocati nello spazio offre a un sistema di controllo una informazione migliore sull'ambiente che si sta monitorando. Inoltre l'eterogeneità dei sensori dà a un sistema di controllo un quadro totale dell'ambiente, permettendo il monitoraggio e quindi l'automazione di un maggior numero di attività.

1.4 Internet nelle reti constrained

Dal punto di vista tecnico il paradigma dell'Internet of Things fa nascere la necessità di connettere a Internet dei dispositivi constrained, cioè con possibilità di calcolo, di memoria, di energia, e di rate limitate. I sensori sono solo un esempio di questo tipo di dispositivi. Le caratteristiche peculiari di Internet, che quindi vanno trasferite nelle reti constrained, sono il protocollo IP e l'architettura client-server.

Il protocollo IP è ciò che tiene unito Internet, in particolare il metodo universale di indirizzamento di ogni dispositivo della rete: ogni interfaccia di rete che possiede un indirizzo IP ha la possibilità di connettersi ad Internet, ed ogni dispositivo della rete ha la possibilità di dialogare con quell'interfaccia per mezzo del suo indirizzo. L'Internet of Things prevede un'alta densità dei dispositivi che saranno connessi ad Internet nel futuro. Il protocollo IPv4 non ha uno spazio di indirizzamento tale da permettere la connessione a tutti questi dispositivi, anzi gli indirizzi si stanno già esaurendo al giorno d'oggi dove Internet è ancora confinato ai computer o al massimo ai dispositivi mobile. Il protocollo IPv6 invece, avendo uno spazio di indirizzamento di 666000 miliardi di miliardi di indirizzi per metro quadrato di superficie terrestre, ha indirizzi più che sufficienti per ogni dispositivo embedded ed è capace di fronteggiare qualsiasi densità di questi dispositivi. Per questo ogni tecnologia in via di studio sulle reti constrained utilizza già IPv6 al posto di IPv4.

Per quanto riguarda l'architettura client-server, nell'Internet tradizionale c'è uno scontro fra due possibilità: SOAP e REST. Data la maggiore leggerezza di REST, sia in termini di dimensione dei pacchetti, sia di elaborazione da parte dei nodi, questo paradigma viene nettamente preferito a SOAP. In REST ogni risorsa è identificata da un URL, mentre l'operazione da eseguire su di essa è già descritta dalla istruzione della richiesta HTTP (le fondamentali sono GET, POST, PUT e DELETE). Quindi l'operazione da eseguire e la risorsa destinataria sono già pienamente descritte nell'header HTTP senza bisogno di layer intermedi come in SOAP. Inoltre il payload delle risposte non ha bisogno di codifiche particolari, anche se in certi casi (come nel trasferimento di dati) viene comunque comodo usare XML.

1.5 Protocolli specifici per le reti constrained

Il lavoro di ricerca sulle reti constrained si è concentrato nell'individuazione di protocolli che potessero lavorare nonostante i limiti di queste reti. Fin da subito si è visto come i protocolli tradizionali di Internet non fossero adatti a queste reti, perché a loro volta progettati per soddisfare a requisiti completamente diversi. La lunghezza degli header della maggior parte dei protocolli di Internet non è accettabile per una rete contrained; alcuni protocolli inoltre utilizzano un numero troppo elevato di trasmissioni per il loro funzionamento (ad esempio il controllo di flusso di TCP o il neighbour discovery di IPv6). Si è ritenuto necessario quindi individuare o creare nuovi protocolli appositamente per le reti constrained.

Il protocollo di livello fisico e link layer adatto alle reti constrained è già stato individuato nello standard IEEE 802.15.4. Questo standard è stato disegnato appositamente per reti WPAN (Wireless Private Area Networks), per dispositivi a basso costo (quindi semplici da costruire a livello produttivo, e con poche esigenze in termini di calcolo e memoria), per bassi consumi energetici, e per trasmissioni a basso rate (le ultime versioni permettono trasmissioni al massimo fino a 250 kbit/s) con un raggio di copertura non molto elevato (dai 50 m negli interni ai 120 m negli esterni come valori massimi).

Già a partire dal livello rete si incontrano i primi problemi. Si è già spiegato nella sezione precedente l'importanza della presenza di IPv6; in realtà questo protocollo non può essere supportato dallo standard IEEE 802.15.4: la MTU

(Maximum Transmission Unit, cioè unità massima di trasmissione) di IPv6 è 1280 byte contro i 127 byte di IEEE 802.15.4, e ciò richiede il supporto alla frammentazione dei pacchetti a livello link layer, cosa non supportata da questo standard; inoltre i 40 byte dell'header di IPV6 riempirebbero già un terzo di un pacchetto.

Per far fronte a questi problemi il consorzio IETF (Internet Engineering Task Force) ha quindi creato tre gruppi di lavoro, tuttora esistenti. Il loro scopo è quello sviluppare soluzioni per reti constrained che siano compatibili con quelle esistenti per Internet per facilitarne l'integrazione.

- **IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) Working Group.** Lo scopo di questo working group è stato quello di permettere alle reti constrained che lavorano con lo standard di rete IEEE 802.15.4 di supportare delle trasmissioni IPv6 a livello rete. Il risultato è stato la creazione del protocollo 6LoWPAN, che non è altro che una versione compressa di IPv6. In particolare i principali problemi che il gruppo ha dovuto affrontare sono:

  - le differenze di MTU tra i due standard, di cui si è già accennato, problema risolto aggiungendo un layer fra i due che si occupa della frammentazione del pacchetto IP;
  - l'header IP troppo grande, problema risolto con delle regole di compressione dell'header e in particolare dell'indirizzo IP;
  - il neighbour discovery di IPv6 che usa il multicasting, non supportato da IEEE 802.15.4 che supporta solo il broadcasting (che va evitato), è stato risolto con un nuovo metodo di neighbour discovery.

- **Routing Over Low power and Lossy networks (ROLL) Working Group.** Lo scopo di questo working group era cercare l'algoritmo di routing più adatto ad una rete constrained, caratterizzata da un grafo costitutito da una quantità elevata di nodi e soprattutto dinamico, e con trasmissioni a basso rate e con alta probabilità di perdita del pacchetto. Una volta verificato che i numerosi protocolli di routing già esistenti non erano adatti a questo tipo di reti, è stato creato il protocollo RPL (Routing Protocol for Low power and lossy networks).

- **Constrained RESTful Environments (CoRE) Working Group.** Lo scopo di questo working group è di portare l'architettura REST nelle reti constrained, in modo da rendere veramente possibile le interazioni con i dispositivi connessi. I protocolli tradizionali di Internet vengono già utilizzati anche nelle reti wireless, quindi potrebbero funzionare anche nelle reti constrained, ma lavorerebbero sempre in condizioni critiche mettendo sempre in pericolo la stabilità della rete, a causa del basso rate, delle perdite dei pacchetti e della limitatezza dei nodi. Serve un protocollo molto più leggero ma che allo stesso tempo garantisca una certa affidabilità nelle trasmissioni. Il gruppo sta quindi lavorando nella creazione del protocollo CoAP (Constrained Application Protocol), che al momento è ancora in una versione draft.

In sostanza, con l'introduzione di questi nuovi protocolli viene proposta una nuova architettura di rete ottimizzata per reti constrained ma parallela a quella di Internet.

- Nei livelli fisico e link local si trova lo standard IEEE 802.15.4, pensato per reti wireless a basso rate e per dispositivi radio a basso costo.

- A livello rete si trova 6LoWPAN, versione compressa di IPv6 pensata per poter lavorare su reti basate sullo standard precedente.

- Il livello rete ha bisogno anche di un protocollo di routing: a questo proposito si usa RPL, creato appositamente per reti constrained.

- Nel livello trasporto si usa il tradizionale UDP (User Datagram Protocol), poiché questo è già un protocollo senza ritrasmissioni e col minimo header possibile. È ovvio che possiede già la piena compatibilità col protocollo IPv6 e col resto di Internet. Al contrario il protocollo TCP (Transmission Control Protocol) è da evitare, a causa dell'header più grande, ma soprattutto dell'uso di un numero molto elevato di trasmissioni.

- Nel livello applicazione si usa CoAP. Si può notare a questo punto che il protocollo non deve semplicemente portare l'architettura REST nelle reti constrained, ma ha anche il compito di riempire tutte le lacune di affidabilità e di sicurezza che UDP porta con sé.

- L'architettura REST permette un'alta flessibilità per quanto riguarda il formato del contenuto dei pacchetti. Comunque per dei messaggi macchina-macchina il formato da preferire sarebbe XML. Però si può subito notare da come è fatto il linguaggio che in esso c'è parecchia ridondanza, quindi pacchetti inutilmente lunghi: si usa quindi EXI (Efficient XML Interchange), creato dal W3C (World Wide Web Consortium) come versione compressa di XML.

Si nota immediatamente come questa nuova architettura crei il bisogno di uno (o più) nodi che facciano da gateway/proxy tra la rete Internet e la rete constrained, per tradurre IPv6 in 6LoWPAN, TCP in UDP, HTTP in CoAP, ed eventualmente XML in EXI.

1.6 I Proxy

I proxy sono programmi che lavorano ai livelli più alti dello stack protocollare. In Internet le comunicazioni normali avvengono tra un client e un server: il client si connette direttamente ad un server, e gli invia una richiesta diretta, ad essa segue una risposta altrettanto diretta del server. Un proxy è un intermediario in questa comunicazione: riceve una richiesta dal client, e decide se, dove e come deve inoltrarla ad un server. Nel caso servisse, si connette lui stesso al server, riceve la risposta, fa le eventuali elaborazioni necessarie, e poi risponde al client come se lui fosse il server.

I proxy possono avere vari scopi, e a seconda del loro scopo possono fare diverse operazioni intermedie nella comunicazione. Uno degli scopi più comuni è quello di superare certe limitazioni dovute all'indirizzo IP, come un server che risponde solo a certi utenti, o un client che può fare richieste solo a certi server: nel primo caso se il proxy ha i permessi per accedere al server, il client può connettersi al proxy invece che direttamente al server, e il proxy inoltrerà la richiesta; nel secondo caso il client si connette ad un proxy che non ha le sue stesse limitazioni, e che inoltra la richiesta per lui.

Altro scopo per cui un proxy è molto usato è il caching. Ci possono essere più macchine in una rete che richiedono molte volte la stessa risorsa. Un proxy messo tra questi client e il server inoltra la richiesta al server solo la prima volta, e nelle successive richieste risponde con la precedente risposta che nel frattempo

aveva salvato nella sua cache. Questo avviene fino a che il proxy ritiene che la risposta sia fresh, cioè sia la stessa che darebbe il server in questo momento; quando il proxy ritiene che la risposta sia diventata "vecchia", rifà la richiesta al server. Questa cosa riduce di molto le richieste della rete al server, aumentando le prestazioni complessive della rete.

Esistono tre tipi fondamentali di proxy.

- **Forward proxy.** Sono i normali proxy, che si comportano come sopra descritto. Ricevono una connessione diretta da parte di un client, si connettono al server per inoltrare la richiesta, ricevono la risposta e la rimandano al client.

- **Reverse proxy.** Sono proxy che al client appaiono come server. In realtà loro non hanno la risposta alla richiesta del client, ma conoscono l'indirizzo del server che ce l'ha, quindi lo contattano per ricevere la risposta e rimandarla al client.

- **Transparent proxy.** Sono proxy nascosti sia al client sia al server. Intercettano una comunicazione che in realtà il client aveva inviato direttamente al server, fanno le eventuali elaborazioni, e la inoltrano al server stando attenti ad usare come indirizzo sorgente quello del client. In pratica sono trasparenti alla comunicazione. Un vantaggio di questi proxy è che il client non ha bisogno di essere configurato, dato che in realtà è ignaro della sua presenza.

1.7 Obiettivi

L'obiettivo di questa tesi è quello di creare un proxy che metta in comunicazione la rete Internet (o una rete locale con la stessa architettura di Internet) e una rete di sensori. Il caso tipico che si vuole gestire è quello di una richiesta GET proveniente da Internet e diretta ad un nodo della rete di sensori. Il proxy dovrà ricevere la richiesta, riconoscere che è diretta ad una rete CoAP, e a questo punto aprire un socket UDP invece del solito socket TCP, e inviare un pacchetto CoAP invece del solito pacchetto HTTP. Dovrà inoltre gestire la risposta secondo le regole del protocollo CoAP, e riconvertirla in HTTP affinché possa essere inviata come risposta al client. Si vuole che il proxy possa lavorare sia in modalità normale

sia come transparent proxy: le connessioni dirette alla rete CoAP verranno così intercettate automaticamente senza bisogno di nessuna configurazione nel client.

L'obiettivo di questa tesi è quello di creare un proxy che riceva delle richieste HTTP provenienti da una rete locale o anche da Internet, rileva se queste sono delle richieste da inviare ad una rete CoAP, e in tal caso converta la richiesta da HTTP a CoAP, gestisca tutto il lato delle trasmissioni, e alla fine riconverta la risposta da CoAP ad HTTP in modo che possa essere compresa dal client.

Capitolo 2

Related work

Questo capitolo mira a descrivere più in dettaglio le varie tecnologie già esistenti che sono servite come base di partenza per lo sviluppo di un proxy HTTP-CoAP. In particolare si descrive il protocollo HTTP e il protocollo CoAP (seguendo come riferimento la draft 4). Si parla poi di Squid, il proxy usato come base di partenza per costruire il nostro, e del suo funzionamento interno. Si descrive infine il nuovo modulo del kernel Linux chiamato TPROXY, che permette la configurazione di un transparent proxy.

2.1 Il protocollo HTTP

Il protocollo HTTP (HyperText Transfer Protocol, cioè protocollo per il trasferimento di un ipertesto) è una delle colonne portanti del World Wide Web, assieme ad HTML (HyperText Markup Language) e agli URL (Uniform Resource Locator): vennero infatti sviluppati assieme alla fine degli anni '80. La prima versione definitiva, denominata HTTP/1.0, è descritta nella RFC 1945 del 1996.

In sostanza HTTP è un protocollo client-server: un client fa una richiesta HTTP ad un server, che solitamente è in ascolto sulla porta 80. Il server risponde con un altro pacchetto HTTP che al suo interno contiene la risorsa richiesta. Il protocollo HTTP si appoggia su TCP per le comunicazioni. Una delle caratteristiche fondamentali del protocollo è di essere completamente stateless: il server non tiene nessuna informazione di stato sulla comunicazione appena avvenuta.

Ben presto si sono notate delle limitazioni di questo protocollo.

- Non era possibile differenziare le richieste nel caso in cui più siti web fossero ospitati nello stesso server.
- La connessione TCP tra client e server veniva chiusa appena inviata la risposta. Ciò rallenta di molto le prestazioni della comunicazione e del server nel caso in cui ci siano più richieste consecutive dello stesso client (ad esempio quando appena scaricata una pagina web si devono scaricare anche le immagini al suo interno).
- Il protocollo aveva grandi lacune sull'aspetto della sicurezza.

Per risolvere queste mancanze, venne proposta la versione HTTP/1.1 contenuta nella RFC 2068 del 1997, poi sostituita dalla RFC 2616 del 1999.

In seguito verrà spiegato il funzionamento del protocollo HTTP. Non saranno spiegate tutte le funzioni, data la loro numerosità; ci si concentrerà sulle funzioni che risultano utili ai fini di questo elaborato.

2.1.1 Struttura del pacchetto

La struttura dei pacchetti di richiesta e di risposta è simile. Il pacchetto è formato da una prima riga, che contiene l'informazione principale della richiesta e della risposta. Seguono delle righe che contengono eventuali opzioni; il formato delle opzioni è `Nome-Opzione: valore` . Infine una riga vuota separa l'header dall'eventuale body. Lo standard dice che ogni riga deve essere separata dai caratteri `\r\n` .

Ciò che differenzia una richiesta da una risposta è la struttura (e di conseguenza il contenuto) della prima riga. Inoltre alcune opzioni hanno senso solo all'interno di una richiesta, viceversa altre hanno senso solo all'interno di una risposta.

2.1.2 Richieste

Il formato di una *request line*, cioè della prima riga di una richiesta, è il seguente:

```
METODO URI VERSIONE_DEL_PROTOCOLLO
```

- La versione del protocollo può essere la stringa `HTTP/1.0` oppure `HTTP/1.1` .

- L'URI (Uniform Resource Identifier) è la stringa che serve a identificare univocamente una risorsa all'interno di Internet. Di solito si tratta del percorso (assoluto o relativo) in cui si trova la risorsa.
- Il metodo è una parola che identifica l'operazione che si vuole svolgere. I metodi più comunemente usati sono:
  - **GET:** chiede di ottenere la risorsa indicata;
  - **POST:** come GET, ma nella richiesta si inviano dei dati che possono servire al server per elaborare una risposta;
  - **HEAD:** chiede di ricevere solamente l'header HTTP della risorsa indicata.

  A questi aggiungiamo altri due metodi che risultano fondamentali all'interno di una architettura REST.
  - **PUT:** carica nel server la risorsa indicata;
  - **DELETE:** chiede al server di eliminare la risorsa indicata.

Alla request line seguono le opzioni, di cui si parlerà nella sezione 2.1.4, una riga vuota, e l'eventuale body.

2.1.3 Risposte

Il formato di una *status line*, cioè della prima riga di una risposta, è il seguente:
`VERSIONE_DEL_PROTOCOLLO CODICE_DI_STATO DESCRIZIONE`

- La versione del protocollo può essere la stringa `HTTP/1.0` oppure `HTTP/1.1` allo stesso modo che nella richiesta. Ovviamente un server può rispondere con la stessa versione della richiesta, o con una minore, ma mai con una maggiore.
- La descrizione è una breve stringa associata al codice che serve semplicemente a rendere il codice comprensibile al linguaggio umano.
- Il codice di stato è la risposta del server. Ogni codice è formato da 3 cifre: la prima sta ad indicare la classe della risposta, le altre due aggiungono precisione alla risposta. Un server infatti può rispondere o con il codice

generico della classe, che è sempre `n00`, oppure con un codice più specifico, cioè `nxx`. Le classi sono:

- **1xx Informational:** messaggi informativi (solo per usi sperimentali)
- **2xx Success:** la richiesta è stata soddisfatta
- **3xx Redirection:** non c'è risposta immediata, ma la richiesta è sensata e viene detto come ottenere la risposta
- **4xx Client error:** la richiesta non può essere soddisfatta perché sbagliata
- **5xx Server error:** la richiesta non può essere soddisfatta per un problema interno del server

In tabella 2.1 vengono elencate le risposte più comuni. Altre risposte si possono trovare più avanti nella tabella 2.2, che mostra anche la mappatura tra i codici di stato HTTP e CoAP.

2.1.4 Alcune opzioni

Tra le innumerevoli opzioni esistenti, specie per la seconda versione del protocollo, vengono qui elencate solo le più importanti, perché spesso presenti nei pacchetti, o perché si dimostreranno utili successivamente.

- **Cache-Control.** Dentro l'opzione Cache-Control possono starci più variabili o flag, separati dal carattere ";", che intendono sovrascrivere i valori o le opzioni che usa il gateway attraverso il quale il pacchetto dovesse eventualmente passare per effettuare il caching. Verrà usata questa opzione per modificare il valore della `max-age` impostata in Squid: si tratta del tempo massimo nel quale il pacchetto può essere considerato fresh, cioè non c'è bisogno di richiederne una versione aggiornata. Lo standard CoAP impone per i propri pacchetti una `max-age` di 60 secondi, salvo che non sia specificato un diverso valore nel pacchetto stesso. L'opzione non ha senso nelle richieste, poiché sono le risposte ad essere poste in cache.
- **Connection.** Determina se la connessione TCP deve rimanere aperta o deve chiudersi dopo l'invio della risposta- Ha più senso quindi la sua presenza

Codice	**Descrizione**
200 OK.	Il server ha interpretato correttamente la richiesta e ha risposto correttamente.
301 Moved Permanently.	La risorsa che abbiamo richiesto è stata spostata in modo permanente.
302 Found.	La risorsa è raggiungibile con un altro URI indicato nel header con l'opzione Location.
400 Bad Request.	La risorsa richiesta non è comprensibile al server.
404 Not Found.	La risorsa richiesta non è stata trovata e non se ne conosce l'ubicazione. O l'URI fornito è incorretto, oppure la risorsa è stata eliminata.
500 Internal Server Error.	Il server non è in grado di rispondere alla richiesta per un suo problema interno.
502 Bad Gateway.	Il server in realtà è un gateway o un proxy, e ha ricevuto una risposta incorretta dal server che ha contattato a sua volta.
504 Gateway Timeout.	Il server in realtà è un gateway o un proxy, e in esso è scaduto il timeout per la ricezione della risposta da parte del server che ha contattato.

Tabella 2.1: Codici di stato HTTP più comuni.

in una richiesta piuttosto che in una risposta. Qui il comportamento è diverso a seconda della versione del protocollo. In HTTP/1.0 la connessione di default viene chiusa, e bisogna impostare l'opzione al valore keep-alive per mantenerla aperta. Al contrario in HTTP/1.1 di default la connessione rimane aperta, e bisogna impostare la variabile al valore close se si vuole chiuderla.

- **Content-Length.** Indica semplicemente la lunghezza del body del pacchetto in byte. È obbligatoria nel caso in cui il pacchetto abbia un body.
- **Content-Type.** Determina la tipologia del contenuto del pacchetto. La codifica è quella dei media type (una volta detti tipi MIME).
- **Date.** Indica la data in cui il messaggio è stato generato. La data deve essere scritta nel formato indicato dalla RFC 1123.

- **Host.** È la parte dell'URL che indica il nome dell'host (ed eventualmente la porta) in cui si trova il server. Ovviamente ha senso solo nelle richieste. In HTTP/1.1 è diventata una opzione obbligatoria.
- **Server.** È una stringa con cui il server che ha generato il pacchetto dà alcune informazioni di sé.
- **User-Agent.** È una stringa con cui il client che ha generato il pacchetto dà alcune informazioni di sé.
- **Via.** È una stringa con cui il proxy/gateway attraverso il quale il pacchetto è passato dà alcune informazioni di sé.

2.2 Il protocollo CoAP

Il Constrained RESTful Environments (CoRE) Working Group sta creando il protocollo CoAP (Constrained Application Protocol) con l'obiettivo di implementare l'architettura client-server di Internet anche in reti constrained che supportano a fatica un protocollo ridondante come HTTP. Al momento della scrittura il protocollo si trova ancora in una versione draft. In questo elaborato si farà sempre riferimento alla draft 4 del protocollo, poiché nell'implementazione si andrà ad usare una libreria basata su questa draft.

È chiaro che per la creazione di questo protocollo lo spunto principale è HTTP: il protocollo principe nell'Internet tradizionale per quanto riguarda le richieste client-server. Ma nella draft viene chiarito subito che questo protocollo non è una mera copia compressa di HTTP. Lo scopo è quello di creare un protocollo che implementi tutte le funzioni principali del pardigma REST (di cui si è già parlato nella sezione 1.4). In più il protocollo CoAP per essere veramente più leggero si appoggia ad UDP anziché a TCP: deve avere quindi un sublayer tra il livello trasporto e il livello applicazione che si occupa di garantire un minimo di affidabilità della trasmissione del messaggio in modo da colmare le lacune di UDP.

2.2.1 Il pacchetto CoAP

Viene mostrata in figura 2.1 la struttura di un pacchetto CoAP.

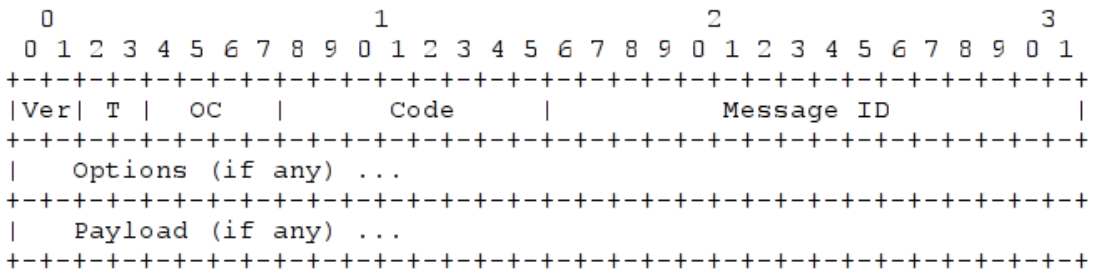

Figura 2.1: Struttura di un pacchetto CoAP.

Si noti che l'header del pacchetto è solamente 4 byte, al quale possono essere aggiunte delle opzioni.

- I primi due bit sono la versione del protocollo. Attualmente l'unico valore possibile è 01.

- Altri due bit servono ad indicare il tipo di pacchetto che si manda. Il pacchetto può essere di 4 tipi: confirmable, non confirmable, ack e reset. Il significato di questi tipi verrà spiegato in dettaglio nella sezione 2.2.2.

- I successivi 4 bit formano l'option count, cioè il numero di opzioni che il pacchetto possiede alla fine dell'header. Può essere anche 0.

- I successivi 8 bit formano il code. A seconda del suo valore può assumere diversi significati e funzioni. Quando il suo valore è 0, significa che il pacchetto è vuoto. I valori da 1 a 31 sono riservati per i tipi di richiesta, i valori da 64 a 191 per i codici di risposta. Entrambi verranno successivamente spiegati in dettaglio nella sezione 2.2.3.

- I rimanenti 16 bit formano il message ID. È un numero che serve ad associare un pacchetto al suo ack, e che serve a identificare eventuali pacchetti duplicati.

- Seguono gli spazi per le opzioni e per il payload, entrambi di dimensioni variabili.

2.2.2 Tipi di messaggio

Come già detto, CoAP implementa delle funzioni che garantiscono un minimo di affidabilità della trasmissione. Per questo motivo ogni pacchetto può essere di 4 categorie, la quale viene scritta nel campo type.

- **Confirmable (CON).** Un pacchetto confirmable richiede che il suo destinatario mandi un ack che testimoni l'avvenuta ricezione. In caso di perdita del pacchetto o dell'ack, il messaggio viene ritrasmesso, duplicando ogni volta il tempo tra una trasmissione e la seguente, e fino ad un massimo stabilito di ritrasmissioni. Come valori consigliati dallo standard, il tempo di attesa tra la prima trasmissione e la seguente è di 2 secondi, mentre il massimo numero di ritrasmissioni è 4 (nelle ritrasmissioni non si conta la prima trasmissione). Ovviamente in caso di perdita dell'ack, al destinatario arriveranno pacchetti duplicati: in questo caso il message ID fa da discriminante. L'ack deve avere lo stesso message ID del pacchetto che l'ha generato.

- **Non-confirmable (NON).** A differenza del precedente, un pacchetto non-confirmable non richiede l'invio di un ack che testimoni l'avvenuta ricezione. Viene usato quindi per trasmissioni che non richiedono nessuna affidabilità (ad esempio nell'invio continuato nel tempo delle rilevazioni di un sensore).

- **Acknowledge (ACK).** La funzione dell'ack è già stata spiegata parlando dei pacchetti confirmable. Non è stato detto che l'ack può essere pieno, quindi contenere la risposta all'interrogazione precedente (seguendo il meccanismo del piggy-backing), oppure vuoto, per implementare le trasmissioni deferred.

- **Reset (RST).** Viene inviato un messaggio di reset in risposta ad un pacchetto che per qualche motivo il server non è riuscito a processare. Un pacchetto di reset deve essere vuoto e come l'ack deve avere lo stesso message ID del pacchetto che lo ha generato.

Quando viene inviata una request, il server genera una response. Ma il protocollo deve tener conto che sta lavorando con dispositivi limitati, quindi potrebbe succedere che il server non abbia subito la risposta pronta. Quindi a differenza

dell'Internet tradizionale in cui si considera solamente il caso di risposte immediate, nel caso di dispositivi constrained si considera anche il caso di risposte di tipo deferred (differite). Nel caso in cui la request sia di tipo CON, è comunque necessario inviare un ack, altrimenti il client continua a ritrasmettere. È in questo caso che si usa un empty ack, che comunica quindi al client che il server ha ricevuto la richiesta, e che invierà al più presto la relativa risposta. Quando sarà pronta, essa sarà una nuova CON, che richiede quindi che il client invii un empty ack per l'avvenuta ricezione. Questa nuova comunicazione dovrà avere un message ID diverso.

A causa della possibilità di risposte deferred, può servire un metodo per associare una richiesta ad una risposta. Il message ID non lo può fare perché deve cambiare ad ogni singola trasmissione. Si usa quindi l'opzione Token. Ognuno dei pacchetti che appartengono allo stesso scambio di dati devono avere lo stesso token. Non è definita la sintassi del token, che può essere qualsiasi; anche la lunghezza può variare da 1 a 8 byte.

2.2.3 Richiesta e risposta

Ogni pacchetto CoAP può essere una richiesta o una risposta a seconda del valore scritto nel campo Code. Per ora CoAP supporta solo 4 tipi di richiesta, corrispondenti ai 4 tipi di richiesta basilari nel paradigma REST: GET (codice 1), PUT (codice 2), POST (codice 3), DELETE (codice 4). Le risposte definite sono molte di più, e sono raccolte nella tabella 2.2. Si può notare anche in questo caso una forte corrispondenza con HTTP, ma in questo caso i codici di risposta sono nettamente divisi in due parti: la classe che occupa 3 bit e il dettaglio che occupa 5 bit. Nel caso in cui un proxy debba convertire da CoAP ad HTTP un codice CoAP non previsto in HTTP, nel pacchetto HTTP andrà messo il codice 502 (Bad Gateway).

2.2.4 Opzioni

La tabella 2.3 mostra un elenco delle opzioni disponibili per i pacchetti CoAP.

Ogni opzione possiede un codice che la identifica, un formato, una lunghezza, e alcune un valore di default. Sono divise in due categorie: critical ed elective. Mentre quando un'opzione elective non viene riconosciuta deve essere semplicemente

Code	**Descrizione**	**Corrispondente HTTP**
64	2.00 OK	200 OK
65	2.01 Created	201 Created
66	2.02 Deleted	204 No Content
67	2.03 Valid	304 Not Modified
68	2.04 Changed	204 No Content
128	4.00 Bad Request	400 Bad Request
129	4.01 Unauthorized	400 Bad Request
130	4.02 Bad Option	400 Bad Request
131	4.03 Forbidden	403 Forbidden
132	4.04 Not Found	404 Not Found
133	4.05 Method Not Allowed	405 Method Not Allowed
141	4.13 Request Entity Too Large	413 Request Entity Too Large
143	4.15 Unsupported Media Type	415 Unsupported Media Type
160	5.00 Internal Server Error	500 Internal Server Error
161	5.01 Not Implemented	501 Not Implemented
162	5.02 Bad Gateway	502 Bad Gateway
163	5.03 Service Unavailable	503 Service Unavailable
164	5.04 Gateway Timeout	504 Gateway Timeout
165	5.05 Proxying Not Supported	502 Bad Gateway

Tabella 2.2: Mappatura tra codici di stato HTTP e CoAP

ignorata, quando un'opzione critical non viene riconosciuta si deve rispondere con un errore. Nel caso di una richiesta CON, la risposta deve essere "4.02 Bad Option"; nel caso di una risposta CON, si risponde con un RST, nel caso di un pacchetto NON, esso deve essere ignorato.

Viene brevemente descritta la funzione di alcune delle opzioni sopra elencate, quelle più utili e quelle che vengono effettivamente implementate dalla libreria che si andrà ad usare.

- **Content-Type.** È un codice che identifica il contenuto del pacchetto. Lo standard associa ad alcuni codici un sottoinsieme dei tipi MIME che si prevede verranno usato più spesso nelle comunicazioni. Il valore di default è 0, che corrisponde a text/plain; charset=utf-8.

Codice	CE	Nome	Formato	Lunghezza	Default
1	C	Content-Type	uint	1-2 B	0
2	E	Max-Age	uint	0-4 B	60
3	C	Proxy-Uri	string	1-270 B	(nessuno)
4	E	Etag	opaque	1-4 B	(nessuno)
5	C	Uri-Host	string	1-270 B	(vedi sotto)
6	E	Location-Path	string	1-270 B	(nessuno)
7	C	Uri-Port	uint	0-2 B	(vedi sotto)
9	C	Uri-Path	string	1-270 B	(nessuno)
11	C	Token	opaque	1-8 B	(vuoto)
15	C	Uri-Query	string	1-270 B	(nessuno)

Tabella 2.3: Opzioni di un pacchetto CoAP

- **Max-Age.** Esprime il massimo numero di secondi in cui la risposta deve essere considerata fresh, cioè in cui un proxy può usare la risposta che ha in cache anziché interrogare il server. Il valore di default è 60 secondi.

- **Token.** L'utilizzo di questa opzione è già stato spiegato precedentemente. Serve ad associare una richiesta ad una risposta, specie in caso di risposte deferred. Può essere di vari formati e di varie lunghezze.

- **Uri-Host, Uri-Port, Uri-Path, Uri-Query.** In un pacchetto CoAP l'URI della risorsa che si desidera viene inserito nel pacchetto già scomposto in host, porta, percorso e query. Inoltre ci possono essere più opzioni Uri-Path, poiché il percorso viene inserito nel pacchetto già scomposto nelle varie subdirectories. Se l'URI contiene caratteri in notazione percentuale, essi devono essere decodificati prima di essere inseriti nel pacchetto (tranne nella sezione query). Ovviamente queste opzioni hanno senso di esistere solamente all'interno di una request. In questo caso Uri-Host non deve mai essere vuoto, come minimo deve contenere l'indirizzo IP dell'host; Uri-Port se non specificato contiene la porta di default 5683, mentre le opzioni Uri-Path e Uri-Query possono non esserci.

Le opzioni all'interno del pacchetto sono situate alla fine dell'header. Devono essere poste in ordine di codice. In figura 2.2 viene visualizzato il formato di una opzione del pacchetto CoAP.

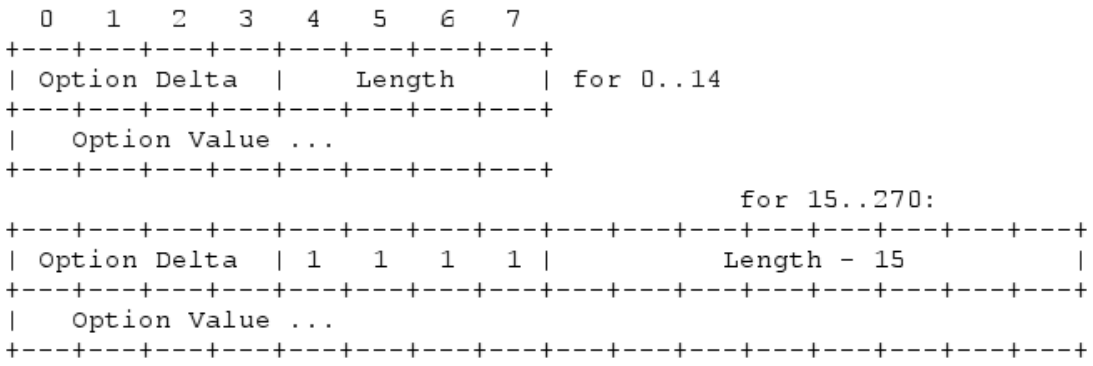

Figura 2.2: Struttura delle opzioni nel pacchetto CoAP.

- Il primo campo è l'Option Delta, largo 4 bit. Contiene non il codice dell'opzione, ma la differenza tra il codice dell'attuale opzione e il codice della precedente (o zero se è la prima).
- Il secondo campo contiene la lunghezza dell'opzione in byte. Essendo lungo 4 bit, sono permesse solo lunghezze tra 0 e 14. Mettendo come lunghezza 15, il campo viene esteso ad altri 8 bit, con cui si possono quindi rappresentare i numeri da 15 a 270.
- L'ultimo campo contiene il valore dell'opzione. Può quindi essere di lunghezza variabile.

2.3 La libreria CoAP

Anche per quanto riguarda CoAP il lavoro di creazione del proxy HTTP-CoAP non parte da zero. Parte da una libreria creata all'interno del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, che da anni lavora sulle reti di sensori all'interno del progetto europeo SENSEI e all'interno del progetto di ricerca WISE-WAI.

La libreria, scritta in codice C, definisce parecchie `enum` e `struct` che riguardano CoAP. Le `enum` più utili sono senza dubbio quelle che elencano il tipo del pacchetto, il tipo di richiesta, i codici di risposta, i tipi MIME. Le `struct` più utilizzate, in realtà definite con `typedef`, sono `coap_packet_t` e `coap_msg_t`: la prima non è altro che un buffer che contiene un pacchetto CoAP, la seconda è un elenco di variabili che contiene tutte le informazioni che si possono ricavare da un

pacchetto CoAP, o che servono a generare un pacchetto, cioè tipo di pacchetto, codice della richiesta/risposta, percorso della risorsa, e ovviamente payload.

La libreria contiene solo due funzioni che però fanno tutto ciò che serve. Entrambe ricevono come parametri dei puntatori alle due struct sopra citate. La funzione `coap_write_packet` a partire da un `coap_msg_t` già riempito con tutti i dati utili riempie `coap_packet_t` e quindi genera il pacchetto da inviare. La funzione `coap_parse_packet` fa l'esatto contrario, cioè a partire da un `coap_packet_t` già riempito riempie `coap_msg_t`.

Il difetto di questa libreria è che implementa poche opzioni del pacchetto CoAP. Per ora implementa solo Uri-Path, Token e Content-Type. A causa di ciò non si potrà rispettare a pieno lo standard che dice che l'opzione Uri-Host deve essere obbligatoria nelle richieste. D'altro canto per scopi sperimentali questa opzione non è necessaria.

2.4 TPROXY

Si è già descritta nella sezione 1.6 la differenza tra un proxy normale (un forward proxy) e un proxy trasparente (un transparent proxy). Per poter implementare un transparent proxy, è necessario però che il sistema operativo che si usa abbia alcune particolari caratteristiche:

1. deve essere in grado di redirigere pacchetti che non sarebbero destinati alla macchina verso un processo locale;

2. deve essere in grado di rimanere in ascolto di connessioni non destinate alla macchina;

3. deve essere in grado di aprire connessioni utilizzando un indirizzo sorgente che non appartiene alla macchina.

Esattamente con questi obiettivi è stato sviluppato il modulo TPROXY per il kernel Linux. La modifica più importante è l'aggiunta di una nuova opzione del socket: IP_TRANSPARENT.

1. La redirezione non può essere fatta usando una istruzione che riguarda il NAT, come ad esempio:

```
iptables  -t nat -A PREROUTING -j DNAT --to-dest IP_ADDRESS
          --to-port PORT
```

poiché verrebbero cambiati indirizzo di destinazione e porta del pacchetto, e il proxy non sarebbe più trasparente. È stato necessario creare una nuova istruzione:

```
iptables  -t mangle -A PREROUTING -p tcp --dport 80 -j TPROXY
          --on-port PROXY_PORT --tproxy-mark 0x1/0x1
```

dove l'indirizzo di default è localhost a meno che non venga sovreascritto con l'opzione `--on-ip`. In pratica vengono sia marcati sia mandati verso un nuovo target.

È necessario modificare anche la tabella di routing locale in modo che i pacchetti rediretti non vengano droppati.

```
ip rule add fwmark 1 lookup 100
```

```
ip route add local 0.0.0.0/0 dev lo table 100
```

2. Una volta che si ha a disposizione l'opzione del kernel IP_TRANSPARENT, l'ascolto diventa cosa facile. Basta aprire un socket con questa opzione, fare il bind usando un indirizzo esterno, e poi mettersi in ascolto. Bisogna però aggiungere delle nuove impostazioni nelle tabelle di routing:

   ```
   iptables -t mangle -N DIVERT
   ```

   ```
   iptables -t mangle -A PREROUTING -m socket -j DIVERT
   ```

   ```
   iptables -t mangle -A DIVERT -j MARK --set-xmark 0x1/0x1
   ```

   ```
   iptables -t mangle -A DIVERT -j ACCEPT
   ```

   Queste istruzioni creano una nuova catena DIVERT che marca i pacchetti allo stesso modo dell'istruzione del punto 1, e poi li accetta. Vengono inviati a questa catena i pacchetti che hanno un socket di destinazione corrispondente con un socket aperto nella macchina.

3. Per l'apertura di nuove connessioni valgono le stesse considerazioni fatte al punto 2. Basta aprire un kernel con l'opzione IP_TRANSPARENT, fare il bind usando un indirizzo esterno, e connettersi. Bisogna aggiungere alle tabelle di routing le stesse impostazioni del punto 2.

2.5 Squid

Squid è uno dei proxy più utilizzati al mondo, questo a causa della sua stabilità, velocità e alla sua completezza in fatto di funzionalità. È stato quindi scelto come base per l'implementazione di un proxy HTTP-CoAP. In particolare le caratteristiche che hanno fatto prediligere questo proxy piuttosto che altri, e piuttosto che una implementazione da zero di un proxy, sono:

- La stabilità e la velocità di Squid è già comprovata.
- Squid è il proxy più completo per quanto riguarda le sue funzionalità.
- Squid è il proxy più diffuso in questo ambiente.
- A partire dalla versione 3 Squid supporta nativamente il transparent proxying, in particolare supporta TPROXY v4.1, che è la versione implementata nel kernel Linux (le versioni precedenti esistevano solo come patch del kernel e non sono state mai inserite ufficialmente in esso).
- Squid supporta il caching, funzionalità molto utile per i nostri scopi in cui è necessario che nella rete di ssensori viaggino meno pacchetti possibile.
- Squid è open source e gratuito, distribuito sotto la licenza GNU GPL. Questo permette intanto di poter essere modificato per i propri scopi, e permette di avere un ottimo proxy senza costi.

2.5.1 Funzionamento di Squid

Per vari motivi risulta difficile capire a pieno il funzionamento di Squid. Il fatto che sia un progetto costruito su decenni di modifiche da parte di più persone, e la vastità stessa del progetto, rendono difficile comprendere a pieno la totalità del codice. Inoltre Squid in origine era scritto in C, poi si è cominciata la conversione a C++ che però non è mai finita: il codice viene quindi complicato da questa commistione di linguaggi. Comunque seguendo il percorso delle funzioni e soprattutto i messaggi di debug, si può arrivare a comprendere in linea di massima il suo funzionamento, e si può quindi concentrarsi nella comprensione delle aree da modificare.

In sostanza Squid è un sistema ad eventi. La funzione `main` non fa molto altro che parsare il file di configurazione, cambiare utente di lavoro, e avviare il loop

principale. Esso continua a controllare se all'interno degli engines registrati ci siano degli eventi da "consumare", e continua a rischedulare gli eventi a seconda del loro timeout o dell'avvenire di eventi esterni (come l'arrivo di un pacchetto).

La modularità di questo sistema, cioè la possibilità di gestire ogni sorta di evento in un solo loop, è garantita da un sistema di callback: ogni evento registrato contiene un puntatore alla funzione che deve essere chiamata nel caso in cui arrivi il momento di gestire questo evento.

Una funzione chiamata è ad esempio `clientReadRequest` (implemetata nel file `client_side.cc`), invocata quando arriva una richiesta HTTP. Da questa segue una cascata di chiamate ad altre funzioni. Viene eseguito il parsing della richiesta, poi viene controllato che questa sia una richiesta valida, poi si controlla l'ACL (Access Control List) che decide se si può continuare, infine si controlla che la risposta a questa richiesta non sia già in cache; in caso contrario viene chiamata la funzione connectStart, che apre un socket verso il server, poi altre funzioni che risolvono l'URL della richiesta e stabiliscono la connessione, infine viene creata una nuova richiesta HTTP e viene inviata con la funzione sendRequest. In sostanza anche Squid segue lo schema tradizionale dell'invio di un pacchetto tramite le funzioni `socket`, `connect` e `send`, solo che queste funzioni sono incapsulate in altre che gestiscono ogni caso e fanno ogni tipo di controllo di errore.

Altra funzione chiamata tramite callback è `readReply`. Essa fa il parsing della risposta, e poi richiama le funzioni che dovranno inserirla nella cache e inviarla al client.

Le tre funzioni sopra citate, cioè connectStart, sendRequest e readReply sono proprio quelle che andranno modificate. In connectStart bisogna intanto decidere se la richiesta è diretta ad una rete CoAP, poi aprire un socket UDP anziché TCP. In sendRequest si invia un pacchetto CoAP anziché HTTP, in readReply si riceve un pacchetto CoAP anziché HTTP, con tutta la gestione degli ack e dei timeout, pacchetto che si deve parsare e trasformare in HTTP in modo che Squid prosegua il suo cammino nel programma come se avesse ricevuto una risposta HTTP.

Capitolo 3

Sviluppo su Squid

Nel seguente capitolo sono descritte le modifiche fatte al codice sorgente di Squid. Lo scopo delle modifiche è quello di intercettare una richiesta proveniente dalla rete Internet, o da una rete locale con l'architettura di Internet, stabilire se questa richiesta è diretta alla rete CoAP, e gestire quindi inoltro e ricezione della risposta coi protocolli UDP/CoAP. La spiegazione è divisa per area del programma, a parte le funzioni `connectStart`, `sendRequest` e `readReply` che essendo le più importanti meritano una descrizione a parte.

Dapprima si modificano le funzioni che fanno il parsing del file di configurazione, in modo da aggiungere in Squid delle opzioni di configurazione strettamente inerenti a CoAP. La più importante di esse è l'opzione `coap_url`, che permette di inserire delle espressioni regolari per decidere in base all'URL se una richiesta è destinata o meno ad una rete CoAP. Vanno modificate anche le funzioni che fanno il parsing e la ricostruzione degli URL, per introdurre il protocollo CoAP. Sono state create inoltre delle funzioni che interfacciano Squid con la libreria CoAP di cui si parla nella sezione 2.3. A questo punto vanno modificate le funzioni `connectStart`, `sendRequest` e `readReply`. La prima è quella che decide se una richiesta è diretta o meno ad una rete CoAP in base all'URL, e poi apre un socket UDP anziché TCP. La seconda è quella che fa l'invio della richiesta, e che quindi deve generare il pacchetto CoAP. La terza riceve i pacchetti dal server, per cui deve gestire tutti i casi in base a ciò che è stato ricevuto. Un capitolo a parte è stato riservato a tutte le modifiche che riguardano i timeout e le ritrasmissioni, modifiche che sono abbastanza sparse all'interno del programma. Per finire sono annotate le modifiche da fare ai Makefile in modo che il programma compili.

3.1 I file di configurazione

La prima cosa che fa il programma una volta avviato è fare il parsing del file di configurazione. Il codice per fare questo si trova principalmente nel file `config.cci`, ma questo è un file autogenerato dal tool `cf_gen` ogni volta che si fa il make del programma. Il tool prende informazioni da molti altri file per generare questo codice.

Il primo file è `cf.data.pre`. Esso contiene tutte le opzioni che è possibile inserire nel file di configurazione, accompagnate dai loro relativi attributi. Tutto ciò è scritto in una sintassi particolare secondo lo schema CHIAVE: valore. In tabella 3.1 sono descritte le varie chiavi possibili e i relativi attributi.

Chiave	**Valore**
NAME	il nome dell'opzione
TYPE	il tipo di dato dell'opzione, tipo di dato che sarà inserito nel codice C++
LOC	il nome della variabile in cui verrà scritto il valore
DEFAULT	il valore di default che avrà la variabile
DOC_START	tag che indica l'inizio della descrizione dell'opzione; la descrizione va indentata e può essere su più righe
DOC_END	tag che indica la fine della descrizione dell'opzione

Tabella 3.1: Attributi del file cf.data.pre inerenti alle opzioni

Lo stesso file servirà per generare sia la documentazione HTML di Squid sia il file di configurazione vero e proprio. Per questo motivo, oltre agli attributi strettamente necessari alla programmazione, esistono degli attributi solamente stilistici. Essi vengono descritti in tabella 3.2.

Il tool a partire da questo file genera `cf.data`, che è molto simile al file precedente. A partire principalmente da queste informazioni, e dal file `cf.data.append` che contiene semplicemente una lista dei tipi di dati che verranno usati (con i nomi che possiedono all'interno del codice C), il tool può generare il file `config.cci`. Il file contiene la funzione che fa il parsing del file di configurazione: per ogni riga del file a seconda della prima parola della riga, che rappresenta il nome dell'opzione, decide qual è la funzione adatta a fare il parsing di quella riga e passa come

Chiave	**Valore**
COMMENT_START	indica l'inizio di una parte del file che serve solo da commento, che risulterà nella visualizzazione HTML
COMMENT_END	indica la fine della parte del file che serve solo da commento
NO_COMMENT_START	indica l'inizio della parte del file che andrà a comporre il file di configurazione
NO_COMMENT_END	indica la fine della parte del file che andrà a comporre il file di configurazione
EOF	indica la fine del file

Tabella 3.2: Attributi del file cf.data.pre inerenti allo stile

argomento il nome della variabile in cui verrà messo il risultato. Nello stesso file c'è anche la funzione che dealloca le variabili che contengono le opzioni, e quella che fa il dump delle opzioni.

È necessario che le variabili siano già state dichiarate all'interno del codice, ed è necessario che siano dichiarate in un file accessibile da tutto il resto del programma. La maggior parte delle variabili viene dichiarata nel file `structs.h`, alcune nel file `typedefs.h`. Entrambi i file sono richiamati da `squid.h`, che a sua volta è richiamato in tutti gli altri file del progetto Squid.

Inoltre c'è bisogno di un file di codice che contenga le implementazioni delle funzioni che fanno il parsing, o che fungono da distruttori. Tale file è `cache_cf.cc`. Perché la generazione automatica funzioni, c'è bisogno che all'inizio di questo file siano dichiarate le firme di tre funzioni, firme che devono avere una forma fissa:

```
parse_NOMETIPO(NOMETIPO *);

dump_NOMETIPO(StoreEntry *, const char *, const NOMETIPO);

free_NOMETIPO(NOMETIPO *);
```

Le implementazioni delle funzioni seguiranno in fondo al file.

3.2 Modifiche fatte ai file di configurazione

Nel file `cf.data.pre` sono state aggiunte le opzioni mostrate in dettaglio in tabella 3.3.

NAME	TYPE	DEFAULT	LOC
coap_url	coap_url_list	null	Config.Coap.coap_url
coap_server_port	ushort	61618	Config.Coap.server_port
coap_response_timeout	time_t	2 seconds	Config.CoapTimeout.response_timeout
coap_max_retransmit	int	4	Config.CoapTimeout.max_retransmit
coap_deferred_timeout	time_t	10 seconds	Config.CoapTimeout.deferred_timeout

Tabella 3.3: Opzioni aggiunte a Squid

Le opzioni vengono qui brevemente spiegate.

- **coap_url**

  In questa opzione va inserita una espressione regolare (regular expression, o abbreviato regex). Il programma farà il matching di ogni URL in arrivo, e se corrisponde il programma considererà la richiesta come diretta ad una rete CoAP. Inizierà quindi per il pacchetto la stessa procedura che sarebbe successa se fosse arrivato un URL che iniziava con "coap://".

  Con la stessa opzione si può fare il mapping di un URL in arrivo. Si può infatti far seguire alla espressione regolare un'altra stringa che funge da pattern. Le parti della stringa pattern composte da $num vengono sostituite con le parti tra parentesi trovate nella regex.

- **coap_server_port**

  Questa opzione definisce quale porta deve usare Squid come porta di default per le richieste CoAP.

- **coap_response_timeout**

  Lo standard dice che un client deve aspettare RESPONSE_TIMEOUT secondi prima di tentare una ritrasmissione del pacchetto. Ad ogni ritrasmissione poi questo tempo viene duplicato. Viene consigliato un tempo di 2 secondi. Questa opzione permette di modificare questo valore.

- **coap_max_retransmit**

  Lo standard dice che dopo la prima trasmissione, un client deve provare a ritrasmettere per MAX_RETRANSMIT volte, con valore consigliato 4. Questa opzione permette di modificare questo valore.

- **coap_deferred_timeout**

  Anche se lo standard non menziona questo timeout, è stata offerta la possibilità di impostare il tempo in cui il server aspetta una risposta deferred prima di chiudere la connessione per timeout scaduto.

Nel file `structs.h` sono state aggiunte le variabili indicate nella colonna LOC di tabella 3.3. Si è cercato di usare dei tipi di variabile già presenti nel file di configurazione originale di Squid, in modo che esistessero già le relative funzioni di parsing.

Ma per l'opzione `coap_url` si è dovuto però aggiungere un nuovo tipo di dato coap_url_list. Si è dovuto quindi aggiungere il nuovo tipo nel file `cf.data.append`, definire il nuovo tipo nel file `structs.h`, creare le tre funzioni relative a questo tipo nel file `cf_data.cc`, che quindi hanno firma:

```
parse_coap_url_list(coap_url_list **);
dump_coap_url_list(StoreEntry *, const char *, const coap_url_list *);
free_coap_url_list(coap_url_list **);
```

Il nuovo tipo non è altro che una struct che contiene la stringa che forma la regex, la stringa che forma il pattern, la regex già parsata, un flag che indica se c'è un pattern, un puntatore al prossimo elemento della lista. Per parsare la regex si è usata la funzione `regcomp` della libreria standard di C secondo lo standard POSIX.

L'unica delle tre funzioni per cui vale la pena spendere una parola è la prima, perché poco più complicata delle altre. Essa divide la riga in token. Verifica già da subito che il primo token sia una regex valida. Se lo è, crea un nuovo oggetto della lista, in cui mettere la regex sia in formato stringa (che servirà per la funzione dump o per motivi di debug) sia in formato regex già parsata. Poi se c'è un altro token, esso è il pattern, e viene messo come stringa nello stesso oggetto.

3.3 Gli URL

Una volta che Squid ha determinato che gli è arrivata una richiesta HTTP, deve fare il parsing dell'URL contenuto nella richiesta. Il parsing consiste nel dividere protocollo,

host, e path, determinare se l'host è numerico o se dovrà essere risolto, scrivere queste informazioni in una nuova istanza della classe HttpRequest. In successive parti del programma l'URL verrà ricostruito, ad esempio per il caching o al momento di eseguire la connessione al server.

Una prima modifica da fare al programma è aggiungere la stringa "coap" alla lista dei protocolli gestibili. Bisogna modificare all'interno del file `url.cc` la funzione `urlParseProtocol`. La funzione legge la parte iniziale dell'URL che riguarda il protocollo, e determina quale valore dell'enumerazione protocol_t associare, enumerazione che si trova nel file `enums.h`. Bisogna quindi modificare anch'essa, per aggiungere il valore PROTO_COAP.

Per lo stesso motivo di `urlParseProtocol`, bisogna modificare anche la funzione `urlDefaultPort`, che come dice il nome stesso determina qual è la porta di default del server nel caso non sia specificata nell'URL. A differenza degli altri protocolli in cui le porte sono valori fissi, nel nostro caso mettiamo come porta di default la variabile che contiene il valore proveniente dal file di configurazione (oppure il valore di default del file configurazione che è 61618).

A questo punto sembra tutto a posto, ma in questo modo Squid non funziona. Il programma infatti utilizza anche in seguito la variabile che contiene il codice del protocollo, e non sa come comportarsi qundo trova il valore PROTO_COAP. Il nostro obiettivo è che Squid prosegua nel suo corso come se fosse arrivato un pacchetto HTTP, e sviare dal programma principale solo al momento dell'invio e della ricezione dei pacchetti. Per questo nella funzione `urlParseFinish` è necessario ripristinare il valore PROTO_HTTP nella variabile protocol. Ma facendo semplicemente in questo modo, al momento della ricostruzione dell'URL verrebbe messo erroneamente "http" invece di "coap", mentre bisogna comunque che questa informazione venga salvata; ad esempio l'URL viene usato per determinare se una risorsa si trova già in cache, quindi se arrivassero due richieste `http://stringa` e `coap://stringa`, esse verrebbero considerate uguali. Si è preferito aggiungere alla classe HttpRequest un flag che segnala appunto che, nonostante il protocollo indicato è "http", in origine il protocollo arrivato era "coap". Le modifiche vanno fatte in `HttpRequest.h` dove viene dichiarata la classe, e in `HttpRequest.cc` nel costruttore della classe per inizializzare il flag a false. La funzione che si occupa di ricostruire l'URL si trova sempre in `url.cc` e si chiama `urlCanonical`: anche questa va modificata in modo che tenga conto del flag al momento della ricostruzione.

3.4 Funzioni dedicate a CoAP

Per la costruzione o il parsing dei pacchetti CoAP si è fatto uso di librerie già esistenti. Dato che queste erano state scritte per altri scopi, per altri programmi, e per un compilatore C e non C++, si è dovuto fare alcune modifiche minori, tipo spostare l'implementazione delle funzioni da un file .h a un file .cc o l'aggiunta di alcuni `include`. Ne risultano tre file: i file CoAP.h e CoAP-04.h (incluso nel primo) che contengono tutti i `typedef` e le `enum` che riguardano il protocollo CoAP, e le due funzioni `parse_packet` e `write_packet`, e il file `CoAP.cc` che contiene semplicemente le implementazioni delle due funzioni. L'uso di questa libreria è già stato spiegato nella sezione 2.3.

Queste strutture vanno però allocate prima di invocare le funzioni, pena degli errori di segmentazione. Per automatizzare questa cosa, nei file `coap_utils.h` e `coap_utils.cc` sono stati implementati dei costruttori e distruttori di queste strutture. Negli stessi file sono state implementate anche le funzioni `coap_create_packet` e `coap_parse_packet` che velocizzano ancora più le operazioni di riempimento delle strutture `coap_msg_t` e `coap_packet_t`.

Tutti i file appena descritti sono raccolti nella cartella CoAP. Assieme a questi ce ne sono altri che contengono funzioni che Squid richiama per gestire i pacchetti CoAP. Questi in particolare sono i file `coap_module.h` e `coap_module.cc`. Contengono parti di codice che si è potuto estrarre dal codice di Squid, e che ovviamente riguardano CoAP.

- `newMid` Genera un nuovo message ID casuale. È una semplice chiamata a `rand` che genera un numero intero a 16 bit.

- `buildCoapRequest` A partire da una istanza di HttpRequest che rappresenta la richiesta che il client ha inviato a Squid, costruisce il relativo pacchetto CoAP, e lo mette in un buffer passato come parametro, che poi Squid userà per inviare la richiesta. Per ora la funzione supporta solo il metodo GET. Dopo aver quindi determinato metodo e path della richiesta, la funzione richiama `coap_create_packet` che produce il pacchetto CoAP. Prima di uscire, il risultato viene copiato nel buffer passato come parametro. La funzione ritorna false se trova un pacchetto da convertire con caratteristiche che ancora non supporta.

- `buildCoapEmptyAck` Genera un empty ack CoAP, e lo mette in un `coap_packet_t` che funge da buffer. È una semplice chiamata a `coap_create_packet` con i parametri adatti a creare un empty ack.

- `translateCoapToHttp` A partire da un `coap_msg_t` che contiene una risposta CoAP genera il relativo pacchetto HTTP e lo mette in un buffer usato da Squid.

La prima riga del pacchetto HTTP deve contenere il codice della risposta, che viene determinato seguendo le associazioni di tabella 2.2. Poi vengono aggiunte alcune opzioni di HTTP: Content-Length è obbligatoria, Content-Type viene ricavata dal pacchetto CoAP; viene aggiunta l'opzione Cache-Control per sovrascrivere le impostazioni di Squid e mettere la max-age a 60 secondi come vuole lo standard CoAP. Per sicurezza, nonostante nella versione della risposta si scriva sempre HTTP/1.0, si aggiunge anche l'opzione `Connection: close` : in questo modo si è sicuri che Squid chiuda la connessione (in realtà non chiude la connessione ma chiude solo il socket UDP), quindi si è sicuri che ogni comunicazione usi socket diversi e non è necessario l'uso di token per distinguere diverse comunicazioni. Viene aggiunta anche la data del pacchetto: in realtà non è la data in cui il pacchetto è stato generato, come dice lo standard, ma la data in cui il pacchetto viene processato dal proxy. Si sopporta questa imprecisione poiché a Squid serve la data del pacchetto per poter gestire correttamente il caching. Il pacchetto che ne risulta viene infine scritto nel buffer passato come parametro, che è il buffer che in seguito Squid userà per il suo processing.

La cartella contiene infine altri due file, `coap_debug.h` e `coap_debug.cc`, che contengono solamente funzioni che hanno aiutato nel debugging del programma, mostrando l'ingresso o l'uscita dalle funzioni, o le variabili presenti in un dato punto del codice. A questo proposito bisogna spiegare un attimo come funzionano i messaggi di debug in Squid.

Quando si vuole emettere un messaggio di debug, si usa la macro `debugs(categoria, livello, messaggio)`. I messaggi di debug infatti sono divisi in categorie, in modo che si possa visualizzare solo una parte di essi nel file di log. Per quelli inerenti CoAP si è usata l'ultima categoria libera disponibile, cioè la 95. Poi i messaggi di debug hanno anche un livello che indica la loro importanza. L'ultimo parametro è il messaggio che si vuole visualizzare. Per lo sviluppo il programma è stato disseminato di questi messaggi. Il file `coap_debug.h` contiene funzioni che usano questa stessa macro e automatizzano messaggi di debug spesso usati.

3.5 La funzione connectStart

La funzione `connectStart` è un metodo della classe FwdState. Come dice il nome stesso è responsabile dell'inizio di una connessione al server nel caso ce ne sia bisogno (ad esempio nel caso in cui la risposta si trovi già nella cache la funzione non viene invocata). È in questo punto che bisogna capire se la trasmissione che andrà fatta al

server deve essere HTTP o CoAP: infatti nel secondo caso dobbiamo aprire un socket UDP anziché TCP. La funzione si occupa anche di impostare i timeout connect e i dati che servono a calcolare il timeout forward e le ritrasmissioni, e di decidere quale sarà l'indirizzo sorgente per la ritrasmissione, diverso a seconda che il proxy sia trasparente o meno. La funzione finisce chiamando la funzione `commConnectStart`. Il programma quindi prosegue risolvendo gli indirizzi non numerici e chiamando la funzione `connect`. Anche in caso di socket UDP la connessione verso il server funziona, anzi ha uno scopo di fare in modo che i pacchetti che provengono da socket diversi siano scartati.

Quindi subito prima dell'apertura del socket la funzone va modificata. Se l'URL inizia con `coap://` la richiesta viene gestita come una richiesta CoAP senza ulteriori controlli. Si imposta a true il flag `towards_coap_network`, che è stato aggiunto alla classe FwdState in modo da poter essere ripreso più tardi nella funzione e nel resto del programma.

Se invece l'URL inizia con `http://`, bisogna controllare le impostazioni ottenute dal file di configurazione. Bisogna controllare infatti se l'URL arrivato corrisponde ad una espressione regolare inserita nel file di configurazione con l'opzione `coap_url`; lo si fa con la funzione `regexec` delle librerie standard di C. Se si trova che l'URL corrisponde, anche stavolta si imposta il flag `towards_coap_network`. A questo punto si controlla se nella stessa opzione è stato inserito anche un pattern: bisogna quindi riscrivere l'URL, ma anche la variable host, e settare un host diverso anche nell'istanza della classe HttpRequest, dato che più avanti il programma userà proprio questa variabile per determinare la destinazione del pacchetto.

A questo punto si sa se la richiesta deve essere gestita come HTTP o CoAP, quindi con un semplice if si decide se aprire un socket UDP (usando una funzione di Squid che si trova nel file `comm.cc` e che incapsula la funzione `sendto`), o lasciare che il programma apra una connessione TCP come faceva di solito.

3.6 La funzione sendRequest

La funzione `sendRequest` è una funzione della classe HttpStateData, classe molto legata a FwdState dato che contiene una variabile che punta a quella classe. Si occupa di costruire la richiesta e di inviarla, dato un socket già aperto e connesso e conoscendo già la destinazione. Si occupa anche di impostare il timeout lifetime, cioè quanto tempo si può attendere una risposta.

A questo punto è semplice modificare il programma: abbiamo già uun flag che indica se il pacchetto debba essere CoAP o HTTP, quindi basta un if che distingua i

due casi. Se è HTTP il programma continua il suo percorso come al solito, chiamando la funzione `buildRequestPrefix` e inviando la richiesta con `comm_write_mbuf`. Se invece è CoAP, viene richiamata la funzione `buildCoapRequest` del modulo CoAP appositamente creata per generare la richiesta CoAP. Nel caso ci siano degli errori restituisce false e il programma chiude al connessione inviando al client un messaggio di errore. Se invece la conversione riesce, viene inviato il pacchetto UDP attraverso la funzione `comm_udp_sendto`.

3.7 La funzione readReply

La funzione `readReply` si trova anch'essa nella classe HttpStateData. Si occupa appunto di gestire la rcezione della risposta: principalmente gestisce ogni tipo di errore di ricezione possibile, e poi richiama le funzioni che fanno il parser dell'header e del body della risposta. Deve essere complicata di molto per gestire le risposte CoAP, a causa della possibilità di una risposta ritardata. Il solito flag dice se la risposta deve essere gestita come HTTP normale (quindi ignorare le modifiche fatte) o come CoAP. L'idea è infatti quella di prendere il buffer in cui è contenuta la risposta, e nel caso di una trasmissione CoAP leggerlo prima che arrivi alle funzioni di parsing. Principalmente possono accadere tre casi: una situazione di errore viene gestita inviando un messaggio di errore al client e chiudendo la connessione; un empty ack richiede che la funzione sia rischedulata e immediatamente chiusa; una risposta completa richiede che il buffer venga completamente riscritto convertendo la risposta CoAP in HTTP e che si lasci proseguire il programma per la sua solita strada, in modo che quando giunge alle funzioni di parsing trova un pacchetto HTTP e non si accorge dello scambio.

A questo punto bisogna introdurre una variabile che conta il numero di passaggi che sono stati fatti attraverso questa funzione. Le azioni da intraprendere al primo passaggio sono molto diverse da quelle del secondo, mentre il terzo non deve esistere quindi viene lanciato un errore. La variabile è stata messa nella classe FwdState assieme al flag che determina se la trasmissione è CoAP.

Nel primo passaggio, dato che Squid è stato programmato per inviare sempre richieste di tipo CON, si possono ricevere solo messaggi di tipo ACK, empty ACK e RST con lo stesso message ID della richiesta. Altri arrivi conducono a emettere un errore.

- Nel caso in cui ricevo un ACK, la trasmissione con il server è conclusa. Uso la funzione `translateCoapToHttp` per fare il parsing del pacchetto ricevuto, e per convertirlo in un pacchetto HTTP che andrà scritto nel buffer della risposta.

- Nel caso in cui ricevo un RST, la trasmissione con il server anche in questo caso è conclusa, ma stavolta il proxy manda il client una risposta con l'errore 502 Bad Gateway.

- Nel caso in cui ricevo un empty ACK, non si può proseguire con la funzione, perché la trasmissione col server non è stata completata, anzi non si ha ancora una risposta da inviare. Quindi bisogna svuotare il buffer in modo che possa ricevere un'altro pacchetto, schedulare una seconda esecuzione della funzione `readReply` chiamando la funzione `maybeReadVirginBody`, e interrompere la funzione.

Quindi in un caso potrebbe esserci un secondo passaggio nella funzione `readReply`. In questo caso si può ricevere solo una CON con un nuovo message ID, altri casi portano all'emissione di un errore. Nel caso di una CON, è stata ricevuta una risposta, che quindi può essere tradotta in HTTP usando sempre la funzione `translateCoapToHttp`. Ma la comunicazione col server non è finita, perché devo mandare un empty ack al server. Si usa quindi la funzione `buildCoapEmptyAck` per generare il pacchetto, che viene inviato come pacchetto UDP sempre con la funzione `comm_udp_sendto`. A questo punto sorge un problema: se questo ack viene perso, il server continuerà a ritrasmettere la risposta, ma il proxy non la riceverà mai e non manderà nessun ack, perché Squid ha già chiuso il socket e ha già inviato la risposta al client. Si poteva rischedulare la funzione `readReply` in modo che attendesse eventuali ritrasmissioni, ma in questo caso si sarebbe ritardata la trasmissione della risposta al client. Per ovvi motivi di velocità si preferiscono un po' di ritrasmissioni nella rete CoAP piuttosto che degli ingiustificati ritardi nella risposta.

In questa funzione bisogna porre particolare attenzione alle variabili che tengono conto della dimensione del pacchetto. Esse infatti sono più di una, e vanno sincronizzate. Il problema più grosso è la classe MemBuf, cioè il buffer di Squid, che non è certo conforme ai principi dell'incapsulamento dell'informazione dato che si può aggiungere o togliere dati al buffer senza cambiare il campo size, o viceversa cambiare il campo size senza modificare il buffer. La non correttezza di queste variabili genera problemi a monte: la connessione non viene chiusa perché si aspetta altri byte rispetto a quelli arrivati, e il caching non viene effettuato.

3.8 Timeout e ritrasmissioni

Le regole per i timeout e le ritrasmissioni sono diverse tra HTTP e CoAP. Bisogna quindi, usando sempre come discriminante il flag `towards_coap_network`, cambiare le

funzioni e gli indici che regolano i timeout e le ritrasmissioni in caso si abbia una trasmissione CoAP, seguendo le regole dettate dallo standard.

La prima funzione da modificare è `checkRetry` della classe FwdState che si trova nel file `forward.cc`. Essa viene chiamata al momento di controllare se si deve fare una ritrasmissione. È una serie di if che restituiscono false se non si devono fare ritrasmissioni. Bisogna aggiungere un bivio al percorso della funzione, mettendo un if che a seconda del valore del flag `towards_coap_network` usa dei parametri diversi negli if della funzione. Il flag a questo punto del programma è sicuramente già stato settato, dato che per chiamare la funzione che decide le ritrasmissioni, bisogna almeno aver fatto una trasmissione.

Vanno cambiati gli if che controllano le variabili `n_tries` e `origin_tries` della classe FwdState, che conta il numero di trasmissioni fatte; vanno confrontati con il valore `max_retransmit` delle opzioni di configurazione (tenendo conto che quest'ultimo conta le ritrasmissioni, mentre le variabili di Squid includono anche la prima trasmissione). Poi va cambiato l'if che controlla se è stato superato il forward timeout, cioè il timeout che controlla per quanto tempo Squid deve tentare l'inoltro di una richiesta. Si è scelto di impostare comunque un forward timeout, ma che sia poco più grande del tempo massimo stimato per una trasmissione CoAP, ritrasmissioni e risposte deferred comprese.

Si definiscono quindi t_r il response timeout, r il numero massimo di ritrasmissioni, t_f il forward timeout, t_d il deferred timeout. Nel caso peggiore una trasmissione dura

$$
\begin{aligned}
& t_r + 2\dot{t}_r + 4\dot{t}_r + ... + 2^r \dot{t}_r + t_d = \\
& = \left(2^0 + 2^1 + 2^2 + ... + 2^r\right) \dot{t}_r + t_d = \\
& = \frac{1 - 2^{r+1}}{1 - 2} \dot{t}_r + t_d = \\
& = 2^{r+1} - 1 + t_d \leq t_f + t_d \, .
\end{aligned}
$$

Quindi si può porre $f_t = 2^{r+1} + t_d \leq t_f + t_d$. In questo modo il forward timeout è abbastanza largo da poter permettere una corretta trasmissione.

Agli if modificati serve aggiungere un nuovo controllo solo per CoAP. Nel caso arrivi come risposta un empty ack, significa che si deve aspettare una risposta deferred. Anche in questo caso si devono annullare le ritrasmissioni. Va modificata la funzione `readReply` in `http.cc` nel punto in cui si gestisce la ricezione degli empty ack. Prima di schedulare una prosssima esecuzione della funzione stessa, si deve modificare la durata del timeout, impostato secondo le regole del response timeout, e metterlo al valore del deferred timeout impostato nel file di configurazione. A questo punto si imposta un

nuovo flag creato appositamente per dire che si sta aspettando una risposta deferred, posizionato nella classe FwdState come gli altri flag, e che verrà appunto usato dalla funzione `checkRetry`.

Il timeout viene settato per la prima volta all'inizio della funzione `sendRequest` sempre nel file `http.cc`, sostituendo il timeout lifetime. Quindi ogni volta che viene inviato un pacchetto viene settato anche il suo relativo timeout. Viene usata la funzione `commSetTimeout`, che richiede il file descriptor del socket su cui si sta lavorando, il valore del tempo di timeout, e un puntatore alla funzione di callback che deve essere chiamata allo scadere del timeout. Il valore temporale viene calcolato come descritto nello standard: la prima volta vale come settato in `response_timeout` nella configurazione di Squid, e ad ogni ritrasmissione questo valore viene raddoppiato. Per sapere a quale ritrasmissione ci si sta riferendo, basterebbe leggere il valore del campo `n_tries` della classe FwdState, ma questo è un campo privato. Per ovviare a questo dettaglio, si definisce un nuovo campo pubblico `coap_n_tries`, che va aggiornato ogni volta che si aggiorna `n_tries`, cioè all'interno della funzione `connectStart` nel file `forward.cc`.

Riassumendo, ad ogni ritrasmissione viene chiamata la funzione `sendRequest`, che fissa ogni volta un timeout sempre maggiore. Il valore di questo timeout viene però modificato in `readReply` nel caso arrivi un empty ack, quindi nel caso in cui si stia attendendo una risposta deferred. Il pacchetto viene ritrasmesso fintantoché la funzione `checkRetry` restituisce true. Essa restituisce false quando viene superato il numero massimo di ritrasmissioni consentito, oppure se si sta aspettando una risposta deferred. È stato praticamente evitato lo scadere del forward timeout.

3.9 I makefile

I makefile di Squid sono generati automaticamente dai tools `autoconf` e `automake`. Il file `bootstrap.sh` richiama questi tools nel giusto ordine. Poi per la compilazione bastano i soliti comandi `configure`, `make`, `make install` (nel nostro caso configure va richiamato con l'opzione `--enable-linux-netfilter`). I tools automatici hanno bisogno anch'essi di alcune configurazioni, che si trovano nel file `configure.in` nella cartella principale del progetto, e nei file `Makefile.am` che si trovano in ogni cartella del progetto.

I file sorgente di Squid si trovano nella subdirectory `src` del progetto. Al suo interno è stata aggiunta una nuova subdirectory `coap` che contiene i file sorgente creati appositamente per CoAP. Per fare in modo che vengano compilati anche i nuovi file aggiunti, bisogna modifcare i seguenti file di configurazione.

- Nella cartella `coap` da noi creata bisogna aggiungere un file `Makefile.am`, che contiene alcune opzioni ma soprattutto la lista di tutti i file da compilare.
- Bisogna modificare il file `Makefile.am` della cartella `src`, inserendo `coap` nella lista delle subdirectories e `libcoap.la` nella lista delle librerie utilizzabili (il file viene generato automaticamente nella carella `coap`).
- Nel file `configure.in` bisogna aggiungere il file `src/coap/Makefile` nella lista dei Makefile da generare.

Capitolo 4

Test

Il capitolo descrive i test fatti sul software finito. Sono divisi in test di funzionamento, in cui si testa se le caratteristiche implementate effettivamente funzionano, e in test di prestazioni, che vogliono confrontare le prestazioni che si ottengono caricando la rete e il proxy con richieste dirette allo stesso server, ma che in un caso vengono convertite in CoAP, nell'altro rimangono in HTTP. Per i test non si è usata una vera rete di sensori, ma delle macchine virtuali che simulano la rete Internet e la rete di sensori con il proxy in mezzo alle due reti; nel capitolo viene brevemente descritto come sono state configurate queste macchine virtuali, concentrandosi in particolare sulla configurazione del proxy.

4.1 Il Testbed

Per i test si sono usate delle macchine virtuali create con VirtualBox. Nelle macchine sono state installate varie distribuzioni di Linux a seconda delle necessità; si sono comunque usate distribuzioni Debian-based, come Debian stesso o (K)Ubuntu. Grazie a VirtualBox si possono creare in modo molto flessibile delle reti virtuali che collegano queste macchine: si possono creare reti isolate, reti connesse alla macchina host, o usare l'opzione NAT per connettere le macchine a Internet (può essere utile per l'installazione di pacchetti). Per i test si consiglia ovviamente di impostare manualmente gli indirizzi IP (sia IPv4 sia IPv6) modificando il file `/etc/network/interfaces`. Vanno anche settate correttamente le tabelle di routing, sia che si lavori su macchine virtuali sia su reti normali: sia i client sia i server devono essere configurati in modo che i pacchetti vengano instradati attraverso la macchina che contiene il proxy. Ad esempio, dato che una rete constrained di solito è chiusa, si può impostare il proxy come default gateway per ogni nodo. Questo è ancor più vero se si lavora in transparent mode: la

rete lato client deve conoscere la rete lato server, e viceversa, ma comunque tutte le comunicazioni devono passare per il proxy. Non bisogna dimenticare di abilitare la macchina proxy al forwarding.

Le configurazioni possibili per i client e i server sono varie a seconda della topologia della rete che si vuole creare o dell'hardware che si ha a disposizione, ma si limitano alle configurazioni di rete, cioè indirizzi IP e tabelle di routing. Per la macchina proxy invece servono configurazioni particolari. Se si vuole abilitare il transparent proxying bisogna configurare correttamente le tabelle di routing. Poi bisogna installare e configurare Squid. Queste procedure vengono spiegate nelle successive sezioni.

4.1.1 Configurazione di TPROXY

La configurazione di Squid come transparent proxy segue principalmente la guida fornita nel sito ufficile di Squid. La prima cosa da controllare è che il kernel supporti TPROXY. Questo succede a partire dalla versione 2.6.28 per IPv4 e dalla versione 2.6.37 per IPv6. Inoltre il kernel deve essere stato compilato con le opzioni `NF_CONNTRACK`, `NETFILTER_TPROXY`, `NETFILTER_XT_MATCH_SOCKET`, `NETFILTER_XT_TARGET_TPROXY` attive, in caso contrario va ricompilato. Questi requisiti sono rispettati in pieno a partire dalla versione 11.04 di Ubuntu.

A questo punto è necessario configurare le tabelle di routing utilizzando `iptables` (un tool di Netfilter). Il problema è che `iptables` non supporta la configurazione di TPROXY per IPv6 prima della versione 1.4.11, che non è installata di default nemmeno nella sopra citata Ubuntu 11.04. Bisogna quindi scaricare i sorgenti dell'ultima versione dal sito di Netfilter e compilarli.

Ora si possono configurare le tabelle di routing. Uno script come il seguente fatto partire all'avvio della macchina esegue tutte le operazioni necessarie.

```
#!/bin/bash

# Per IPv4

sudo ip rule add fwmark 1 lookup 100
sudo ip route add local 0.0.0.0/0 dev lo table 100

sudo iptables -t mangle -N DIVERT
sudo iptables -t mangle -A DIVERT -j MARK --set-mark 1
sudo iptables -t mangle -A DIVERT -j ACCEPT
```

```
sudo iptables -t mangle -A PREROUTING -m socket -j DIVERT
sudo iptables -t mangle -A PREROUTING -p tcp --dport 80 -j TPROXY \
              --tproxy-mark 1 --on-port 3129

# Per IPv6

sudo ip -6 rule add fwmark 1 lookup 100
sudo ip -6 route add local ::/0 dev lo table 100

sudo ip6tables -t mangle -N DIVERT
sudo ip6tables -t mangle -A DIVERT -j MARK --set-mark 1
sudo ip6tables -t mangle -A DIVERT -j ACCEPT

sudo ip6tables -t mangle -A PREROUTING -m socket -j DIVERT
sudo ip6tables -t mangle -A PREROUTING -p tcp --dport 80 -j TPROXY \
               --tproxy-mark 1 --on-port 3129
```

Lo script segue le indicazioni date dalla guida e le cose dette su TPROXY nella sezione 2.4: in pratica le connessioni dirette alla porta 80 (di qualunque destinazione) vengono intercettate e mandate alla porta 3129 su cui Squid è in ascolto. Rispetto alla guida le istruzioni sono però ripetute due volte per IPv4 e IPv6, e dalle istruzioni che fanno il match del socket viene tolto il parametro -p tcp: infatti noi stiamo lavorando sia con pacchetti TCP sia UDP, quindi la regola con il parametro non catturerebbe i pacchetti in arrivo dalla rete CoAP.

4.1.2 Installazione di Squid

Ovviamente non si usa lo Squid fornito con la distribuzione di Linux usata, ma quello modificato e ricompilato. Squid supporta TPROXY solo a partire dalla versione 3.1, e IPv6 solo dalla versione 3; la versione usata in questo elaborato è la 3.1.9. Dato che la compilazione di Squid si basa sui tools `autoconf` e `automake` che generano i `Makefile`, bisogna (installare questi tools e) eseguirli prima della compilazione. Lo script `bootstrap.sh` contenuto nella cartella di Squid automatizza tutto ciò. Poi l'installazione procede come una normale installazione di Linux: `./configure` , `make` , `make install` , tranne per il fatto che per la configurazione serve l'opzione `--enable-linux-netfilter` che appunto abilità l'uso di TPROXY. Squid viene installato nella directory `/usr/local/squid`.

4.1.3 Configurazione di Squid

Prima di essere usato, c'è bisogno di cambiare alcune opzioni nel file di configurazione di Squid. Quella principale è settare la porta usata per il transparent proxying. Sotto la riga `http_port 3128` che indica la porta usata dal proxy per il suo normale funzionamento, bisogna aggiungere la riga `http_port 3129 tproxy` che indica la porta che il proxy deve usare per connessioni trasparenti.

Altra cosa importante da cambiare è l'utente che Squid usa per lavorare. Squid deve essere avviato da root (altrimenti non può usare TPROXY). Una delle prime cose che fa è aprire un sottoprocesso che lavora a nome di un altro utente. Si consiglia quindi di creare un nuovo utente e un nuovo gruppo ad-hoc, e aggiungere nel file di configurazione le righe `cache_effective_user UTENTE` e `cache_effective_group GRUPPO`. Perché il tutto funzioni, la cartella in cui Squid è installato deve appartenere all'utente e al gruppo appena creati.

Ultima cosa che è necessario configurare è l'ACL (Access Control List). Il modo più semplice di farlo è sostituire la riga `http_access deny all` con la riga `http_access allow all`; in questo modo è come se fosse disabilitata. Se si vogliono configurazioni più fini, bisogna mettere prima della regola precedente altre regole `http_access allow NOMERETE`, quando prima ancora si era dato un nome (qualsiasi) ad una rete di ingresso o di uscita, attraverso l'istruzione `acl NOMERETE src NETWORKADDRESS` se si tratta di una rete in ingresso al proxy, o `acl NOMERETE dst NETWORKADDRESS` se si tratta di una rete in uscita dal proxy.

Le opzioni aggiunte con le modifiche per CoAP non necessitano di essere configurate, dato che tutte hanno dei valori di default. Se necessario possono essere aggiunte delle istruzioni `coap_url`. Ad esempio se si vuole che tutte le richieste HTTP siano convertite a CoAP basta aggiungere la riga `coap_url http://*` .

Altra opzione che potrebbe risultare utile è quella che decide quali messaggi di debug visualizzare. Si aggiunge la riga `debug_options 95,9` per visualizzare tutti i messaggi di debug relativi a CoAP. Si aggiunge la riga `debug_options ALL,9` per visualizzare tutti i messaggi di debug.

Infine c'è il caching. La configurazione di default in Squid imposta il tempo minimo in cui una risposta deve essere considerata valida a 0 minuti. Per attivare la cache basta aumentare questo valore. Purtroppo la cache non si può configurare in secondi, ma la massima granularità è quella dei minuti. Se ci fosse il bisogno di disabilitare del tutto la cache, si aggiunge l'opzione `cache deny all`.

4.1.4 Client e server ad hoc

Per fare i test sono stati creati dei client e dei server ad-hoc. Il client non ha nulla di speciale per quanto riguarda le operazioni che fa e l'implementazione, in quanto deve fare richieste HTTP (quindi TCP) o ad un proxy o direttamente al server. Per i test possono quindi essere usati vari client o tools già esistenti, o addirittura un browser, se si configura il server a dare una risposta da esso leggibile, e se si fanno richieste con URL che inizia per `http://`. È stato creato un client che semplicemente velocizza l'operazione di creazione e invio di una richiesta, magari usando sempre gli stessi parametri di indirizzi e porte, e che visualizza il pacchetto HTTP arrivato come risposta.

Più complicato è stato creare un client per i test di velocità, che mandi n richieste al secondo. Il problema si può risolvere in diversi modi, ma spesso ci si scontra con i limiti software delle macchine reali o virtuali che si usano. L'approccio preferito è stato quello dei thread: vengono aperti più thread ognuno dei quali è un client. Il problema è che raggiungere il limite dei thread apribili è facile se si lavora su una macchina virtuale, bisogna per forza usare una macchina reale come client. Il secondo limite che si raggiunge è quello dei socket aperti, sia nella macchina client che nel proxy.

Si è dovuto per forza invece costruire un server ad-hoc, in quanto riceve e invia pacchetti CoAP, e secondo le regole del protocollo CoAP. Per funzionare il server creato utilizza la stessa libreria CoAP usata per modificare Squid di cui si parla in sezione 2.3. Il server sta sempre in ascolto su una porta; quando riceve un pacchetto legge il path: se è "/" o "/hello" risponde subito con codice `2.00 OK`, se è "/def" risponde con un empty ack, con altri percorsi risponde subito con codice `4.04 Not Found`. Per gestire le risposte deferred si usano i thread: dopo aver mandato l'empty ack, si apre un thread, che dopo un tempo di attesa casuale manda la risposta; il thread continua a ritrasmettere finché non raggiunge MAX_RETRANSMIT, e poi si chiude, oppure può venire chiuso dal programma principale quando questo riceve un ack con lo stesso mid della risposta deferred. È un server sicuramente lontano dalla perfezione e con poche funzionalità, ma che fa il minimo indispensabile per i test.

Nel caso dei test di velocità, un server così costruito potrebbe inficiare sul calcolo dei ritardi. Un modo brutale di risolvere il problema è aprire più server, su porte diverse o magari su più macchine virtuali adibite a server, che simulano la presenza di più sensori.

4.2 Test di funzionamento

4.2.1 Richiesta normale

Il primo test che si può fare è inviare una GET al proxy che chiede una risorsa CoAP. Si può fare benissimo anche con il programma `telnet`. Per connettersi basta dare il comando:

`telnet INDIRIZZO_PROXY PORTA_PROXY`

Il programma attende che venga immessa la richiesta HTTP. Basta scrivere:

`GET coap://INDIRIZZO_SERVER/ HTTP/1.0`

e andare a capo due volte (ciò indica che l'header del pacchetto è finito).

Una possibile risposta corretta è:

```
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 9
Cache-Control: max-age=60
X-Cache: MISS from proxy
Via: 1.0 proxy (Squid/3.1.9)
Connection: close

It works!
```

Da questo esempio base si possono cambiare varie cose, sia nella richiesta sia nelle impostazioni del proxy. Si può provare a mettere indirizzi entrambi IPv4 o IPv6, o provare le combinazioni miste. Si può provare a mettere un path diverso, nel nostro caso l'unico implementato è "/hello". Si possono provare le risposte deferred mettendo come path "/def"; la risposta arriva ma si nota che non è immediata come nei casi precedenti.

Usando un packet sniffer tipo tcpdump o Wireshark si può osservare cosa accade nella rete. Nel caso di risposte immediate si vedono transitare solo due pacchetti UDP: la richiesta e la risposta. Nel caso di risposte deferred si vedono 4 pacchetti: la richiesta, il suo ack vuoto, la risposta deferred, il suo ack che dal proxy va al client.

4.2.2 Caching

Abilitando il caching, si nota che questo succede solo per la prima richiesta inviata. Se si inviano successivamente richieste uguali, nessun pacchetto transita nella rete lato server, anzi dai file di log si nota che il programma non passa per le funzioni di invio

e ricezione che sono state modificate. La risposta contiene l'opzione `X-Cache: HIT from proxy`.

4.2.3 Transparent proxying

Finora le prove descritte usano il proxy in modalità forward. Per usarlo in modalità transparent bisogna connettersi direttamente al server, col comando:
`telnet INDIRIZZO_SERVER 80`
A questo punto il proxy ha intercettato la connessione e l'ha inviata a Squid nella porta 3129. Squid sta di nuovo aspettando che venga inserita una richiesta HTTP. Come prima basta scrivere:
`GET coap://INDIRIZZO_SERVER/ HTTP/1.0`
e si ottiene di nuovo la risposta vista in precedenza. Anche qui si possono fare le stesso prove di prima. Bisogna però tener conto di una differenza col caso precedente: se si vuole connettersi ad un server IPv6 bisogna possedere un indirizzo IPv6. Nel caso precedente si poteva connettersi al proxy con indirizzo IPv4, e poi nella GET richiedere una risorsa IPv6.

4.2.4 URL matching e mapping

Altro test che si può fare è quello del funzionamento dell'opzione `coap_url`. Finora le GET dovevano avere un URL che iniziava con `coap://` per essere convertite in un pacchetto CoAP, altrimenti l'inoltro era una normale richiesta HTTP. Una prova semplice è inserire l'opzione `coap_url http://*` : ora anche tutte le richieste con URL che inizia con `http://` vengono convertite in CoAP. Da qui in poi si possono fare vari test, ad esempio si può fare in modo che vengano convertiti solo i pacchetti HTTP diretti alla rete constrained con l'opzione `coap_url http://2001:123:456:789::*` (ovviamente con indirizzo variabile).

4.2.5 Perdite di pacchetti

Per testare cosa succede in caso di perdite dei pacchetti, c'è bisogno di un server che riceve le richieste ma che non risponde, o risponde solo ad alcune (dipende da cosa si vuol testare). Mandare pacchetti con il server spento genera messaggi ICMP di errore, che Squid raccoglie e usa per terminare la connessione con un messaggio di errore verso il client.

A questo punto si può fare un test omettendo tutte le risposte: con un packet sniffer si vedono arrivare tutte le richieste sempre più distanziate nel tempo, fino allo scadere

dell'ultimo timeout e del numero massimo di ritrasmissioni. Si possono omettere solo le prime risposte: si vede che alla prima risposta che arriva, Squid inoltra la risposta al client e chiude normalmente la trasmissione. Si può ritardare od omettere l'invio della risposta deferred: il timeout impostato nel file di configurazione scade e non ci sono altre ritrasmissioni.

4.3 Test sulle prestazioni

Vengono presentati alcuni test fatti sulle prestazioni del programma finito. Essi vogliono verificare intanto che le modifiche non abbiano intaccato il funzionamento di Squid, e poi a confrontare le prestazioni di una rete constrained inondata di pacchetti CoAP con la stessa rete inondata di pacchetti HTTP.

Sono serviti degli accorgimenti per superare alcuni limiti software. Intanto per scaricare il lavoro sul server creato, sono stati creati 4 server virtuali, e su ciascuno sono stati aperti 2500 server CoAP su porte diverse: un numero sicuramente sovrabbondante che vuole impedire connessioni simultanee sullo stesso server. Come server HTTP è stato usato il classico Apache, aperto in tutti e 4 i server virtuali ma su una porta sola: Apache non ha di certo problemi nel gestire la concorrenza delle richieste. Invece per poter fare misure accurate il client deve essere un programma unico che gira su una macchina unica. Il limite sul numero dei thread apribili da un processo impedisce ogni test usando una macchina virtuale come client: bisogna usare la macchina host e connetterla al proxy; non è una operazione difficile in VirtualBox.

I test richiedono che la rete tra il proxy e i server sia una rete constrained. Si possono aggiungere dei limiti alla rete con il programma di Linux `tc`: esso raccoglie in sé vari moduli che riescono ad alterare la coda di invio dei pacchetti di una interfaccia di rete: si possono così imitare delle costrizioni di una rete. Per limitare il rate si usa il comando:

```
tc qdisc add dev eth0  root handle 1:0 tbf rate 100kbit buffer 1600
                       limit 3000
```

che va applicato sia nel proxy sia nei server. Il programma `tc` costruisce un albero in cui si possono assegnare varie regole ai vari rami, e in cui si possono indirizzare diverse classi di traffico a seguire le regole di diversi rami. In questo momento è stata popolata la radice dell'albero. Per aggiungere un ramo alla radice che introduca sia un ritardo sia delle perdite casuali nella rete si usa il comando:

```
tc qdisc add dev eth0 parent 1:0 handle 10:0 netem delay 50ms loss 10%
```

Nel nostro caso introdurremo le perdite in entrambi i sensi della comunicazione, mentre il ritardo verrà introdotto solo nel lato server per simulare un ritardo di servizio.

4.3.1 Variazione della banda disponibile

Il grafico 4.1 è stato ottenuto generando un numero fisso di 200 richieste al secondo per 5 secondi, ma variando la banda della rete proxy-server ad ogni prova. La cosa è stata ripetuta prima con richieste `coap://` e poi con richieste `http://`, e le due curve sono state sovrapposte. In ascissa quindi si trova la banda, con valori da 10 a 250 kbit/s che sono valori tipici di una rete constrained. In ordinata si riportano il numero di risposte corrette ricevute interrompendo la ricezione appena trascorsi i 5 secondi. Il grafico 4.2 riguarda lo stesso esperimento, solo che mostra in ordinata il ritardo di servizio medio in secondi.

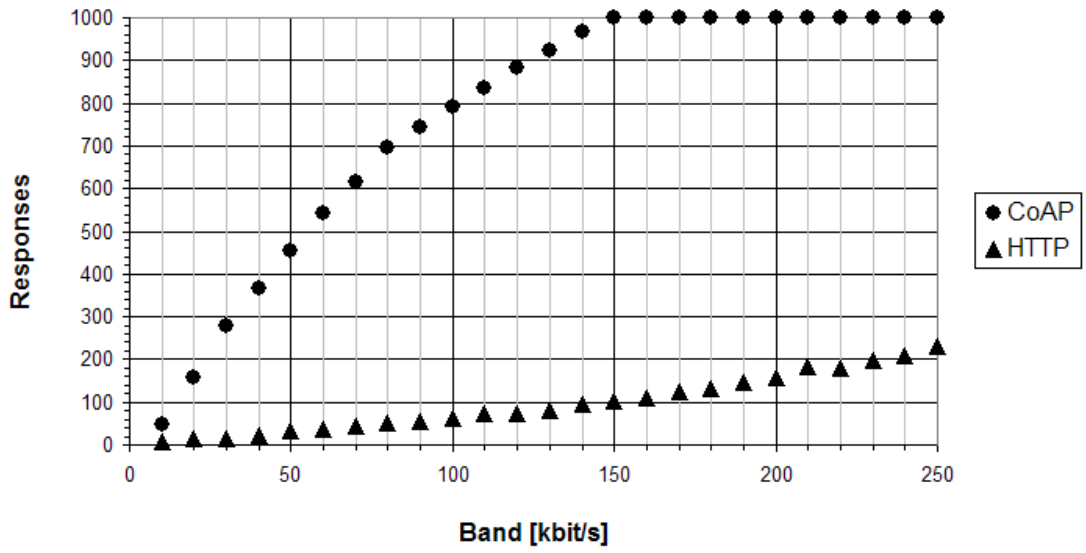

Figura 4.1: Risposte corrette ricevute in 5 secondi al variare della banda della rete constrained, generando un numero fisso di 200 richieste al secondo.

Si nota la netta superiorità di CoAP rispetto ad HTTP sia nel numero di pacchetti gestiti nello stesso tempo e nelle stesse condizioni di banda, sia nel ritardo di servizio. In particolare CoAP raggiunge velocemente il valore di 1000 risposte ricevute, che corrisponde al numero di risposte inviate, mentre HTTP arriva a gestirne circa 200 coi valori più elevati di banda. Per quanto riguarda il ritardo di servizio, esso descresce all'aumentare della banda in entrambi i casi, ma il delay di CoAP è sempre inferiore a quello di HTTP. Questo divario si può spiegare pensando alle dimensioni e al numero di pacchetti da spedire: mentre CoAP ha solo i due pacchetti di richiesta e risposta, HTTP deve stabilire una connessione, inviare e ricevere e mandare un ack (l'ack della richiesta infatti viene incluso nella risposta), e poi chiudere la connessione. In realtà il delay osservato per HTTP misura solo il round trip time (RTT) della richiesta:

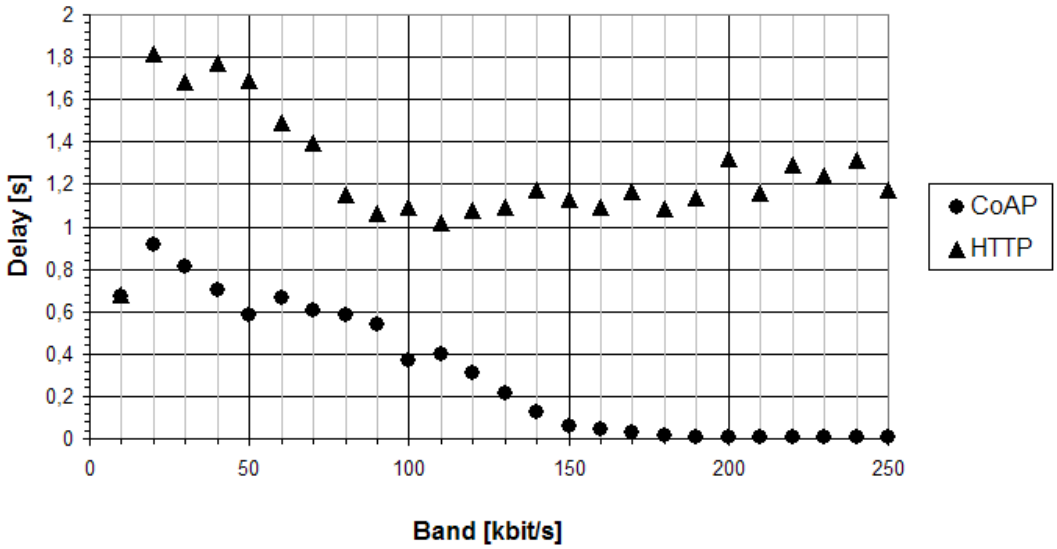

Figura 4.2: Ritardo di servizio delle risposte corrette ricevute in 5 secondi al variare della banda della rete constrained, generando un numero fisso di 200 richieste al secondo.

dovendo tenere conto dell'apertura e della chiusura della connessione, esso sarebbe ancora maggiore.

Il grafico 4.3 è stato generato allo stesso modo del grafico 4.1, ma aggiungendo nella rete proxy-server una perdita di pacchetti del 10%, e un ritardo di servizio del server di 50 ms, emulando così una rete constrained più realistica della precedente, che si avvicinava più alla situazione ideale. Il grafico 4.4 riguarda lo stesso esperimento, e mostra il ritardo di servizio medio.

I risultati mostrano le stesse cose dei due grafici precedenti: anche in caso di perdite dei pacchetti CoAP è nettamente superiore di HTTP, nonostante il grafico evidenzi che non tutte le richieste hanno ottenuto una risposta veloce (cioè all'interno dei 5 secondi), quindi qualche ritrasmissione ha portato la risposta ad arrivare in ritardo. I ritardi di servizio della risposta vengono peggiorati dal ritardo di servizio del server aggiunto, come è giusto che sia, ma gli andamenti sono più o meno gli stessi, a parte dove la banda è troppo piccola in cui il calcolo del ritardo medio conta solo i pochi pacchetti arrivati subito.

Confrontando questi ultimi due grafici con i precedenti si nota però che paradossalmente l'introduzione di un ritardo di servizio nel server aiuta le trasmissioni HTTP. Probabilmente questo ritardo decongestiona il proxy che riesce a gestire meglio l'arrivo delle risposte e la chiusura delle connessioni.

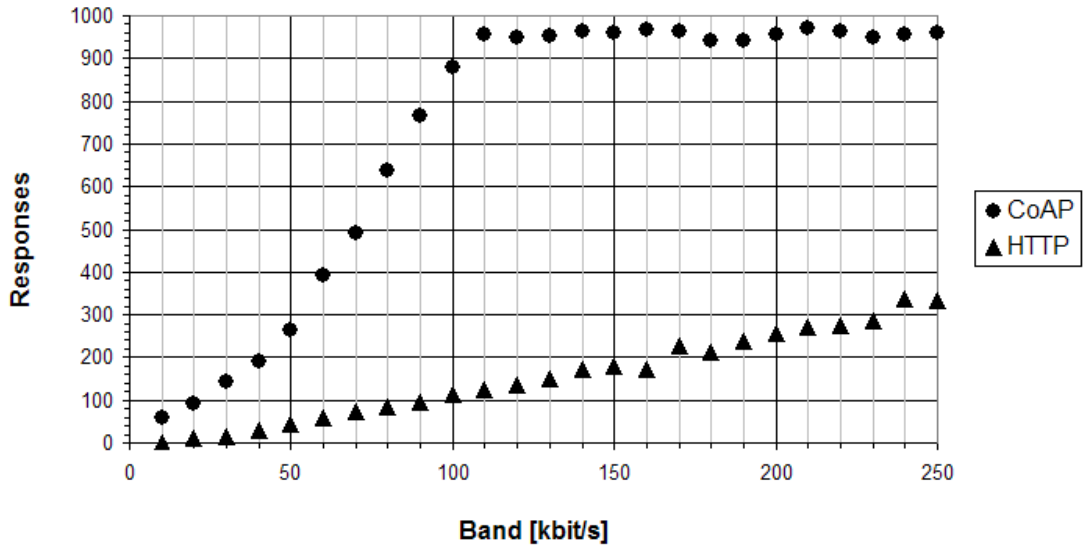

Figura 4.3: Risposte corrette ricevute in 5 secondi al variare della banda della rete constrained, generando un numero fisso di 200 richieste al secondo, e aggiungendo un ritardo di servizio di 50 ms e un 10% di perdita dei pacchetti.

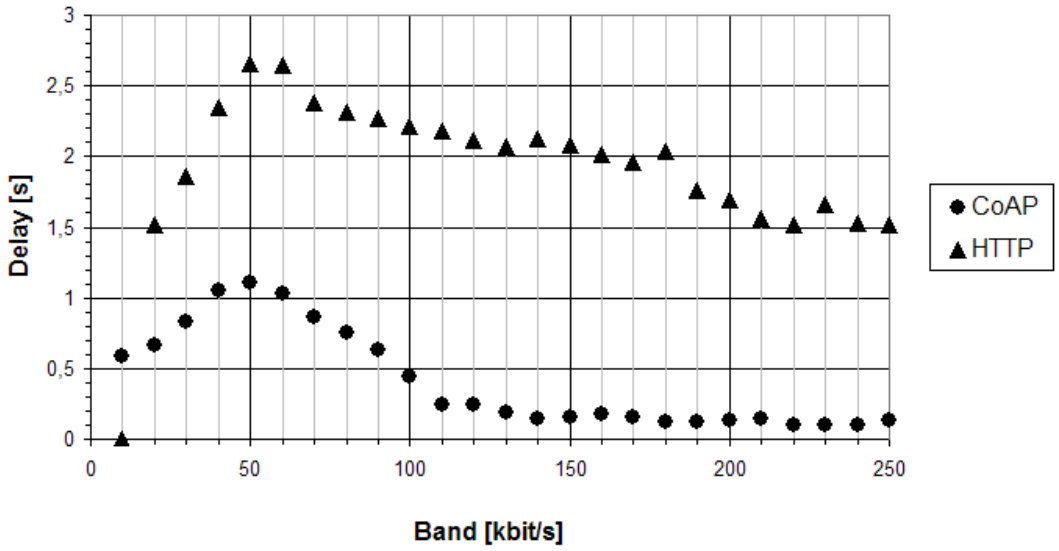

Figura 4.4: Ritardo di servizio delle risposte corrette ricevute in 5 secondi al variare della banda della rete constrained, generando un numero fisso di 200 richieste al secondo, e aggiungendo un ritardo di servizio di 50 ms e un 10% di perdita dei pacchetti.

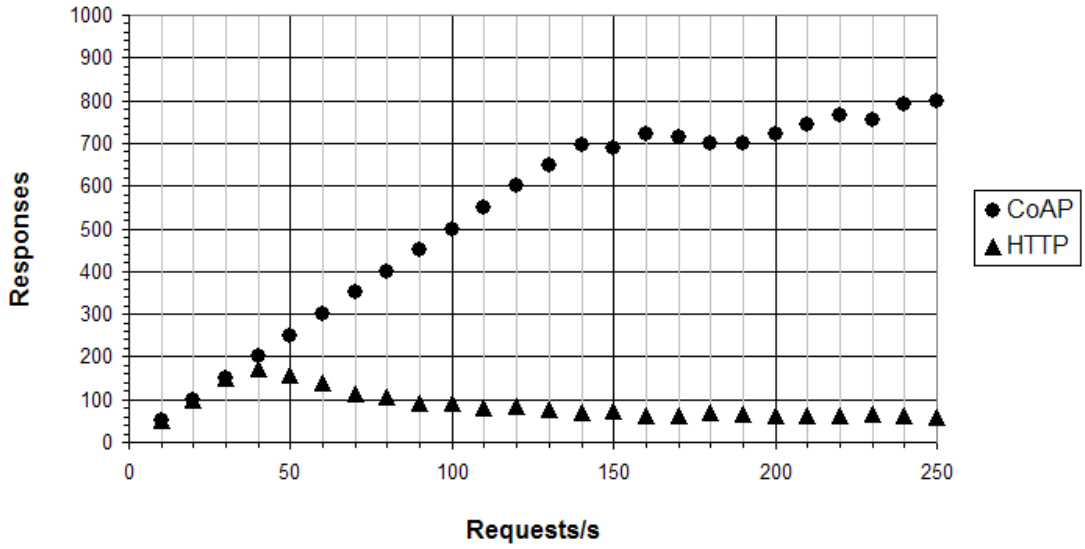

Figura 4.5: Risposte corrette ricevute in 5 secondi al variare del numero di richieste al secondo effettuate, tenendo fissa la banda della rete constrained a 100 kbit/s.

4.3.2 Variazione del numero di richieste al secondo

Per ottenere il grafico 4.5 si è tenuta fissa la banda della rete proxy-server a 100 kbit/s, e si è variato il numero di richieste al secondo fatte, sempre all'interno di un intervallo di 5 secondi. Come nel caso precedente è stata fatta la stessa prova prima con richieste CoAP e poi con HTTP. In ascissa quindi troviamo il numero di richieste al secondo inviate, in ordinata il numero di risposte corrette ricevute interrompendo il conteggio a 5 secondi. Il grafico 4.6 riguarda lo stesso esperimento, ma riporta in ordinata il ritardo di servizio medio.

Si può notare la netta differenza tra CoAP e HTTP: il secondo comincia ad andare in crisi attorno alle 40 richieste al secondo, con un forte aumento del delay e addirittura un continuo calo di risposte servite all'aumentare del numero di richieste; CoAP invece deve arrivare fino alle 140 richieste al secondo prima di mostrare dei problemi di non servizio e di aumento del ritardo, problemi che però sono meno marcati di quelli di HTTP dato che il numero di risposte non va a decrescere ma continua ad aumentare anche se di poco.

Imitando gli esperimenti della sezione precedente, il grafico 4.7 è stato ottenuto con le stesse modalità del grafico 4.5, ma con l'aggiunta di una perdita di pacchetti del 10% nella rete proxy-server e di un ritardo di servizio di 50 ms applicato al server per simulare una condizione più vicina alla realtà. Il grafico 4.8 riguarda lo stesso esperimento ma mostra in ordinata il ritardo medio di servizio.

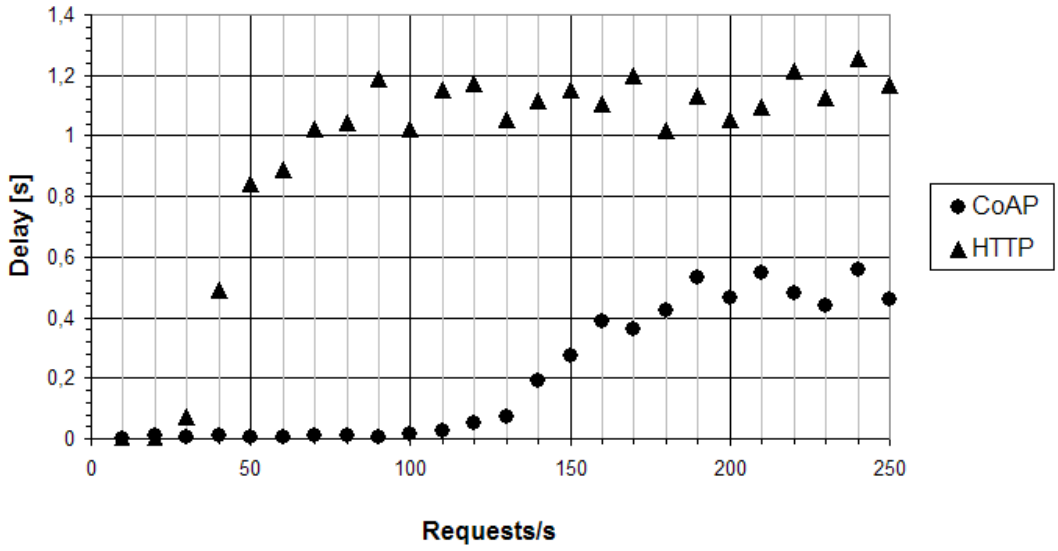

Figura 4.6: Ritardo di servizio delle risposte corrette ricevute in 5 secondi al variare del numero di richieste al secondo effettuate, tenendo fissa la banda della rete constrained a 100 kbit/s.

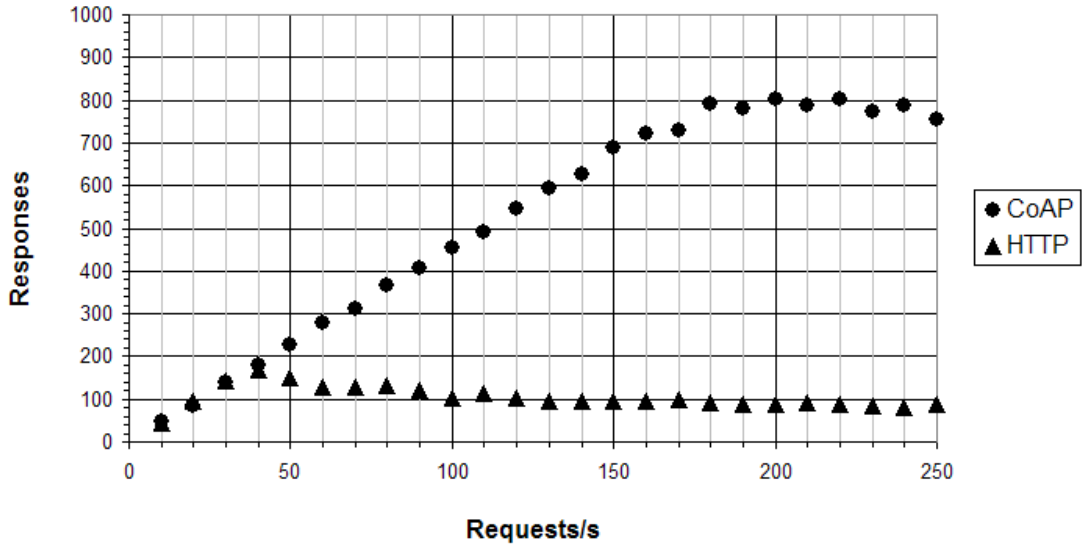

Figura 4.7: Risposte corrette ricevute in 5 secondi al variare del numero di richieste al secondo effettuate, tenendo fissa la banda della rete constrained a 100 kbit/s, e aggiungendo un ritardo di servizio di 50 ms e un 10% di perdita dei pacchetti.

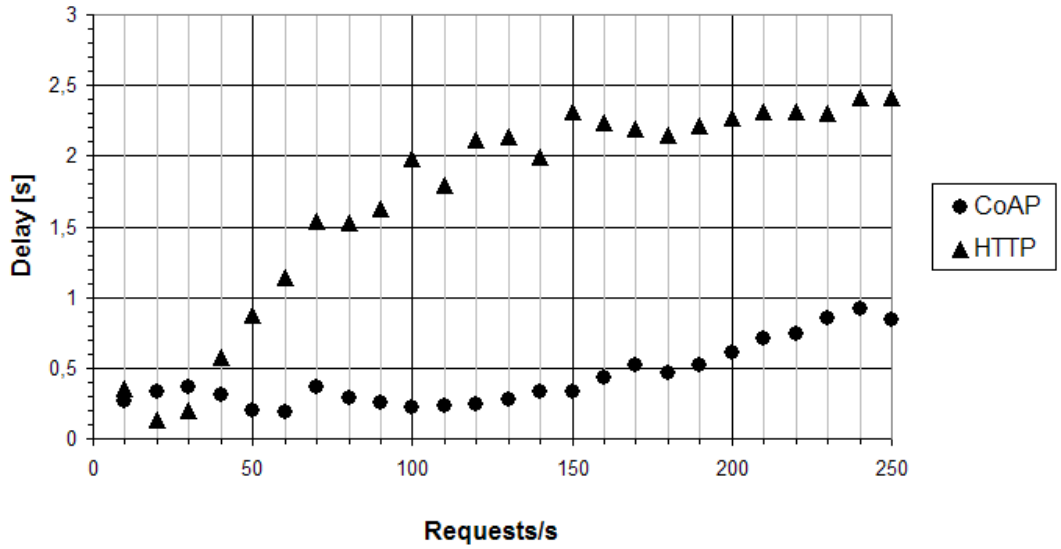

Figura 4.8: Ritardo di servizio delle risposte corrette ricevute in 5 secondi al variare del numero di richieste al secondo effettuate, tenendo fissa la banda della rete constrained a 100 kbit/s, e aggiungendo un ritardo di servizio di 50 ms e un 10% di perdita dei pacchetti.

Non si notano grandi differenze con i due grafici precedenti. Anche in questo caso si nota però che paradossalmente il ritardo di servizio aiuta il proxy nella gestione delle risposte, probabilmente rendendole meno concorrenti e più diluite nel tempo. Si nota però come HTTP nel caso di un numero basso di richieste si avvicini di molto alle prestazioni di CoAP, ma poi raggiunge ben presto il limite di banda: probabilmente i sistemi di ritrasmissione di HTTP riescono a gestire con più efficienza le perdite dei pacchetti, ma questo a scapito di un grande overhead che una rete constrained non può permettersi.

Conclusione

Il risultato del lavoro descritto in questo elaborato è un proxy che, in aggiunta a tutte le caratteristiche intrinseche di Squid come velocità, affidabilità e gestione di ogni tipo di imprevisto, include anche un modulo interno che gestisce le richieste dirette ad una rete constrained che supporta il protocollo CoAP. In questo modo Squid può effettivamente essere usato come proxy tra la rete Internet (o una rete locale basata sullo stack protocollare TCP/IP) e una rete constrained basata su CoAP. Il proxy è stato modificato in modo che i parametri del protocollo CoAP siano completamente configurabili dal gestore della rete; in particolare si riesce ad impostare facilmente quali richieste il cui URL non inizia esplicitamente con il prefisso `coap://` devono essere considerate comunque dirette ad una rete CoAP. Si è mostrato inoltre come configurare Squid in modo che esso possa lavorare anche in modalità trasparente, con l'ausilio del modulo del kernel TPROXY. Il proxy sa gestire bene le differenze di regole tra i protocolli HTTP e CoAP, in particolare sa gestire le risposte deferred, introdotte dallo standard CoAP per gestire il caso in cui un dispositivo a causa delle sue capacità limitate non possa fornire subito la risposta ad una richiesta.

Si è mostrato infine come Squid sappia gestire non solo il caso ideale, ma anche casi con banda limitata e perdite di pacchetti che costringono l'uso di ritrasmissioni, cioè i casi più comuni per quanto riguarda le reti constrained. I test effettuati dimostrano che Squid sa gestire richieste in arrivo ad alta frequenza: ciò mostra innanzitutto che le modifiche fatte al codice sorgente non inficiano sulle prestazioni di Squid. I test inoltre mettono in evidenza la netta superiorità del protocollo CoAP rispetto ad HTTP quando si lavora su reti constrained: l'overhead introdotto da CoAP in una rete è nettamente inferiore di quello di HTTP, e questo permette che un numero di richieste molto più elevato vengano portate a termine con successo.

Grazie a questo lavoro di tesi, ogni risorsa all'interno di una rete constrained è veramente divenuta accessibile a tutta la rete Internet. Si è compiuto così un passo avanti verso l'Internet of Things e il futuro delle telecomunicazioni.

Bibliografia

[1] Angelo P. Castellani, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, Michele Zorzi, *Architecture and protocols for the Internet of Things: A Case Study*. [Online]. Available: www.webofthings.com/wot/2010/pdfs/144.pdf

[2] Dr.P.C.Jain, Arti Noor, Vinod Kumar Sharma, *Internet of Things - An Introduction*. [Online]. Available: www.cdacnoida.in/ascnt2011/Ubiquitous Computing/Paper/1. IOT.pdf

[3] Carsten Bormann, JP Vasseur, Zach Shelby, *The Internet of Things*. IETF Journal. [Online]. Available: www.isoc.org/wp/ietfjournal/?p=2066

[4] Z. Shelby, K. Hartke, C. Bormann, B. Frank, *Constrained Application Protocol (CoAP)*, draft-ietf-core-coap-04, January 2011. [Online]. Available: tools.ietf.org/id/draft-ietf-core-coap-04.txt

[5] A. Castellani, S. Loreto, A. Rahman, T. Fossati, E. Dijk, *Best practices for HTTP-CoAP mapping implementation*, draft-castellani-core-http-mapping-01, July 2011. [Online]. Available: tools.ietf.org/id/draft-castellani-core-http-mapping-01.txt

[6] T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945, May 1996. [Online]. Available: tools.ietf.org/rfc/rfc1945.txt

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616, June 1999. [Online]. Available: tools.ietf.org/rfc/rfc2616.txt

[8] Balabit IT Security, *TPROXY*, [Online]. Available: www.balabit.com/support/community/poducts/tproxy

[9] Laszlo Attilla Toth, Krisztian Kovacs, Amos Jeffries, *TPROXY version 4.1 + Support*. [Online]. Available: wiki.squid-cache.org/Features/Tproxy4

Ringraziamenti

Per la stesura di questo elaborato si ringraziano il prof. Michele Zorzi, i dott. Angelo Paolo Castellani, Mattia Gheda e Nicola Bui, e tutte le persone del SIGNET Group del Dipartimento di Ingegneria dell'Informazione dell'Università degli Studi di Padova per l'aiuto fornito, per la pazienza e per la disponibilità.