UNIVERSITÀ
DEGLI STUDI
DI MILANO

Ohm-o-matic

Un sistema distribuito per la gestione intelligente dell'energia elettrica prodotta da un complesso di case.

Maione Michele – 931468 – michele.maione@studenti.unimi.it

Indice

1 – Descrizione del progetto

Lo scopo del progetto è quello di realizzare un sistema per la gestione intelligente dell'energia elettrica prodotta da un complesso di case. Il consumo elettrico di ogni casa viene misurato costantemente da uno smart meter che monitora l'utilizzo degli elettrodomestici. L'architettura del sistema da sviluppare è mostrata in Figure 1: Architettura del sistema.

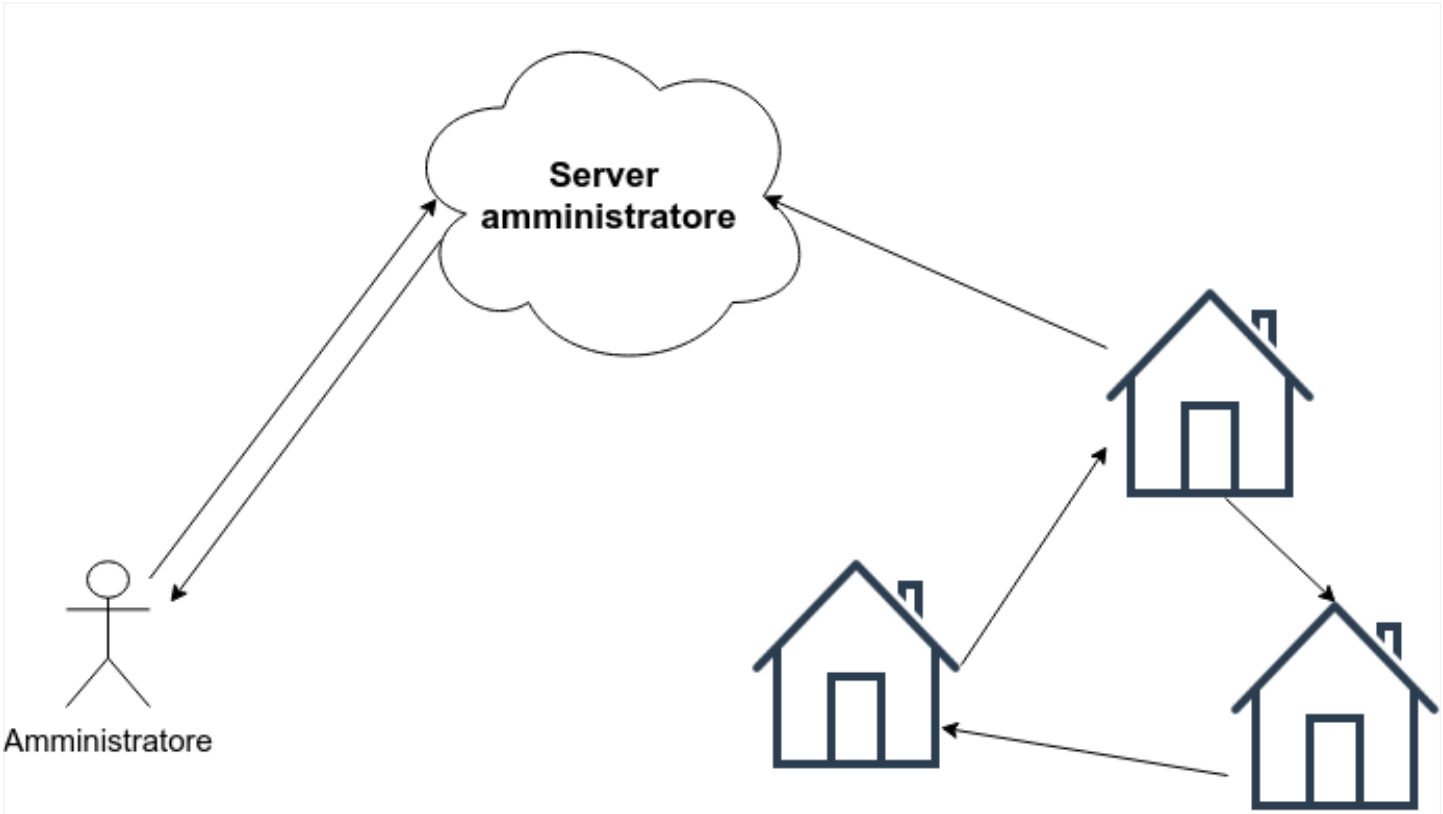

Figure 1: Architettura del sistema

Il complesso di case costituisce una rete peer-to-peer dinamica volta alla gestione decentralizzata del consumo elettrico. All'interno di questa rete, ogni Casa deve informare il vicinato del suo consumo elettrico in tempo reale, in modo tale che ogni casa possa calcolare il consumo energetico complessivo. Inoltre, quando una casa necessita di consumare una quantità di energia superiore alla norma, deve prima chiedere il consenso alle altre case in modo tale da non superare la quota complessiva prevista per il complesso. Le informazioni relative al consumo di corrente condominiale vengono periodicamente fornite dalle case ad un server remoto, chiamato Server amministratore, tramite il quale gli amministratori possono consultare i consumi energetici per calcolare le spese. Il Server amministratore permette anche alle case di aggiungersi o rimuoversi alla rete peer-to-peer del condominio in modo dinamico.

2 – Implementazione

2.1 – Comunicazione

Per tutti i tipi di comunicazione, sia nella rete P2P sia con il server REST è stato usato Google gRPC 1.21 + Google Protocol Buffers 3.

2.2 – Server REST

Il server REST è stato sviluppato utilizzando Oracle Grizzly 2.29 + Oracle Jersey 2.29 + Google Protocol Buffers 3.

```java
@POST
@Path("/iscriviCasa")
@Consumes("application/x-protobuf")
@Produces("application/x-protobuf")
public listaCase iscriviCasa(casa par)
{
    synchronized (elencoCase)
    {
        elencoCase.add(par);

        synchronized (notifiche)
        {
            notifiche.add(new Pair<>(
                new Date(),
                "Casa " + par.getIdentificatore() + " si è aggiunta!")
            );
        }

        return buildListaCase();
    }
}
```

Figure 2: Metodo REST per l'iscrizione di una casa utilizzando Protocol Buffers, con notifica all'amministratore.

2.3 – Architettura P2P

È stato implementato "Chord" (Stoica, Morris, Karger, Kaashoek & Balakrishnan; MIT 2001), un protocollo e algoritmo per DHT. Chord è completamente decentralizzato e simmetrico. Ha la caratteristica di essere particolarmente efficiente all'inserimento e rimozione di nodi al sistema, e ha una complessità temporale logaritmica rispetto al numero di nodi. L'algoritmo di ricerca di Chord è provatamente robusto in presenza di nodi guasti o riconnessioni.

2.3.1 – Funzionamento

Chord si basa su un overlay ad anello con $N = 2^M$ GUID. Il nodo responsabile di una determinata chiave K è quello con il primo identificativo W tale che W succede K in senso orario.

Ogni nodo X in Chord mantiene informazioni su gli M successori più il predecessore:

- predecessore (per robustezza usando il metodo stabilize)
- fingers: un insieme di M nodi distanziati esponenzialmente dal nodo X, vale a dire l'insieme dei nodi che si trovano a distanza 2i da X, con $0 \leq i \leq M - 1$ (per lookup e stabilizzazione)

La correttezza di Chord si basa sul presupposto che ogni peer conosca esattamente i successori e i finger. Tale requisito può essere compromesso dalla dinamicità dei peer. Per mantenere la correttezza dei dati di routing sono eseguiti periodicamente appositi algoritmi di stabilizzazione.

2.3.2 – Analisi spazio temporale

Ogni nodo conserva informazioni su (M + 1) nodi, $O(\log N)$.
L'operazione di lookup richiede M messaggi, $O(\log N)$.
In caso di connessione alla rete o abbandono della rete il numero di messaggi è $\Omega(\log 2N)$.

2.3.3 – Hasing

L'algoritmo Chord necessità di un GUID sia per i nodi sia per le chiavi nella DHT, gli autori proponevano di utilizzare SHA-1 (160 bit), ma è stato scelto di usare SHA3-512 (512 bit), perché palesemente più resistente alle collisioni.

2.3.4 – Comunicazione Chord

Ogni nodo è rappresentato dalla classe **NodeLink**, che viene poi wrappata nella classe Protocol Buffers **casa** (sarebbe stato più corretto usare lo stesso nome).

```
public NodeLink(final String identificatore, final String IP, final int port)
{
    this.IP = IP;
    this.port = port;
    this.identificatore = identificatore;
    this.ID = GB.SHA(indirizzo());
}
```

Figure 3: Costruttore di NodeLink

```
message casa {
    bytes ID = 1;

    string identificatore = 2;

    string IP = 3;
    int32 port = 4;
}
```

Figure 4: Equivalente di NodeLink in Protocol Buffers

```
public static BigInteger SHA(final String s)
{
    if (_shaStrings.containsKey(s))
        return _shaStrings.get(s);

    final var b = new BigInteger(duSHA3_512.digest(s));
    _shaStrings.put(s, b);

    return b;
}
```

Figure 5: Hashing a 512 bit

2.3.5 – DHT

Il fulcro di tutta l'architettura P2P è la DHT. In pratica viene utilizzata in questo modo:

Chiave	Valore
Lista peer	[peer1; peer2; …; peerN]
peer1	3
peer2	2
peerJ	x
peer1_1	[17/07/2019 23:12:01; 0,45 kW · h]
peer1_2	[17/07/2019 23:12:03; 0,73 kW · h]
peer1_3	[17/07/2019 23:12:05; 0,64 kW · h]
peer2_1	[17/07/2019 23:12:02; 1,11 kW · h]
peer2_2	[17/07/2019 23:12:04; 0,97 kW · h]
peerJ_a	[17/07/2019 23:12:05; 0,32 kW · h]
peerJ_b	[17/07/2019 23:12:07; 0,47 kW · h]
…	…
peerJ_x	[17/07/2019 23:12:44; 0,22 kW · h]

2.3.5.1 – Lista peer

Per avere la lista dei peer viene cercata la chiave "Lista peer".

2.3.5.2 – Generazione nuova media

In pratica quando viene generata una nuova media, un peer (in modo sincrono e atomico) incrementa e ottiene il valore associato al suo id, ad esempio peer1 otterrebbe 4 (3+1). Quindi inserisce con la chiave ID_N la media ottenuta con la data, ad esempio peer1_4 → [17/07/2019 23:12:08; 0,34 kW · h].

```java
// T(N): O(log N)
public BigInteger incBigInteger(final BigInteger key)
{
    while (true)
    {
        Serializable r = _functionDHT(RichiestaDHT.incBigInteger, key, null);

        if (r instanceof BigInteger)
            return (BigInteger) r;
        else
        {
            System.out.println(key + " inc non trovata");
            GB.sleep(_sleepTime);
        }
    }
}
```

Figure 6: Chord, funzione di incremento contatore

```java
Serializable incBigInteger(BigInteger key)
{
    synchronized (_data)
    {
        var val = (BigInteger) _data.getOrDefault(key, BigInteger.ZERO);
        val = val.add(BigInteger.ONE);

        _data.put(key, val);

        return val;
    }
}
```

Figure 7: DHT: funzione di incremento contatore

2.3.5.3 – Calcolo media condominiale

Il calcolo della media condominiale viene fatto ogni 2 secondi, ed inviato al server REST.

In modo sincrono, viene presa la lista dei peer (inclusi i morti) per ogni peer viene presa la chiave del suo ID, così da ottenere il numero di medie generate (K), poi da 1 a K, viene cercata la chiave ID_K, e si ottiene il valore e viene aggiunto ad una lista. Infine si cicla la lista e per date simili (scarto di 1 secondo) vengono sommati i valori (quindi se tre case hanno prodotto valori alle 23:12:02 vengono sommati).

```java
public synchronized void calcolaStatistiche()
{
    for (final var peer : chord.getPeerList())
    {
        final var lastNumero = ultimoAggiornamentoGrafico.getOrDefault(peer, BigInteger.ZERO);
        final var curNumero = chord.getOrDefault(peer.ID, BigInteger.ZERO);
        ultimoAggiornamentoGrafico.put(peer, curNumero);

        synchronized (daInviareCondominio)
        {
            for (var statisticaAltroPeer : chord.getIncrementals(peer.ID, lastNumero, curNumero))
            {
                final var p = Pair.<Double, Date>fromSerializable(statisticaAltroPeer);
                final var attuale = condominioGrafico.getOrDefault(p.getValue(), 0d) + p.getKey();

                condominioGrafico.put(p.getValue(), attuale);
                daInviareCondominio.add(new Pair<>(p.getValue(), attuale));

                if (peer.ID.equals(chord.getID()))
                {
                    final var attuale_m = mioGrafico.getOrDefault(p.getValue(), 0d) + p.getKey();
                    mioGrafico.put(p.getValue(), attuale_m);
                }
            }
        }
    }
}
```

Figure 8: Calcolo statistiche condominiali e generazione grafico

```
public void invioStatisticheCondominiali()
{
    mutexInvioStatisticheCondominio.invokeMutualExclusion(() ->
    {
        GB.waitfor(() ->
        {
            synchronized (daInviareCondominio)
            {
                _gRPCtoRESTserver.aggiungiStatisticaGlobale(daInviareCondominio);
                daInviareCondominio.clear();
            }

            synchronized (sharedVars)
            {
                return !invioStatisticheCondominioInEsecuzione;
            }
        }, 2000);
    });
}
```

Figure 9: Mutex e invio statistiche condominiali al server REST ogni 2 secondi

2.4 – Mutua esclusione

È stato implementato l'algoritmo “An Optimal Algorithm for Mutual Exclusion in Computer Networks” (Ricart, Agrawala; University of Maryland 1981), una versione estesa e ottimizzata dell'algoritmo di Lamport. Esso viene usato sia per il boost, sia per l’invio delle statistiche condominiali al server REST.

2.4.1 – Sincronizzazione

La sincronizzazione implementata è esattamente quella mostrata nel paper.

```
P (Shared_vars)
  Comment Choose a sequence number;
  Requesting_Critical_Section := TRUE;
  Our_Sequence_Number := Highest_Sequence_Number + 1;
V (Shared_vars);
```

Figure 10: Sincronizzazione secondo Ricart & Agrawala

```
synchronized (shared_vars)
{
    // Choose a sequence number
    requesting_critical_section = true;
    our_sequence_number = highest_sequence_number + 1;
    outstanding_reply_count = peerList.length - numberOfResources;
}
```

Figure 11: Implementazione in Java

2.4.2 – Aggiunte

È stato aggiunto il parametro stringa *identificativoRisorsa* che viene utilizzato per utilizzare l’algoritmo sia per chi deve usare il boost sia per chi deve inviare le statistiche condominiali al server REST.

```
public MutualExclusion(String identificativoRisorsa, int numberOfResources, Chord chord)
{
    this.me = chord.getNodeLink();
    this.chord = chord;
    this.numberOfResources = numberOfResources;
    this.identificativoRisorsa = identificativoRisorsa;
}
```

Figure 12: Costruttore classe con identificativo risorsa

```
public void reply(String _identificativoRisorsa)
{
    synchronized (shared_vars)
    {
        if (identificativoRisorsa.equals(_identificativoRisorsa))
            outstanding_reply_count--;
    }
}
```

Figure 13: metodo reply con identificativo risorsa

2.5 – Visualizzazione dati

Per la visualizzazione dei dati, oltre che da terminali ci sono dei grafici a linee implementati con la libreria per chart Knowm Xchart 2.5.4.

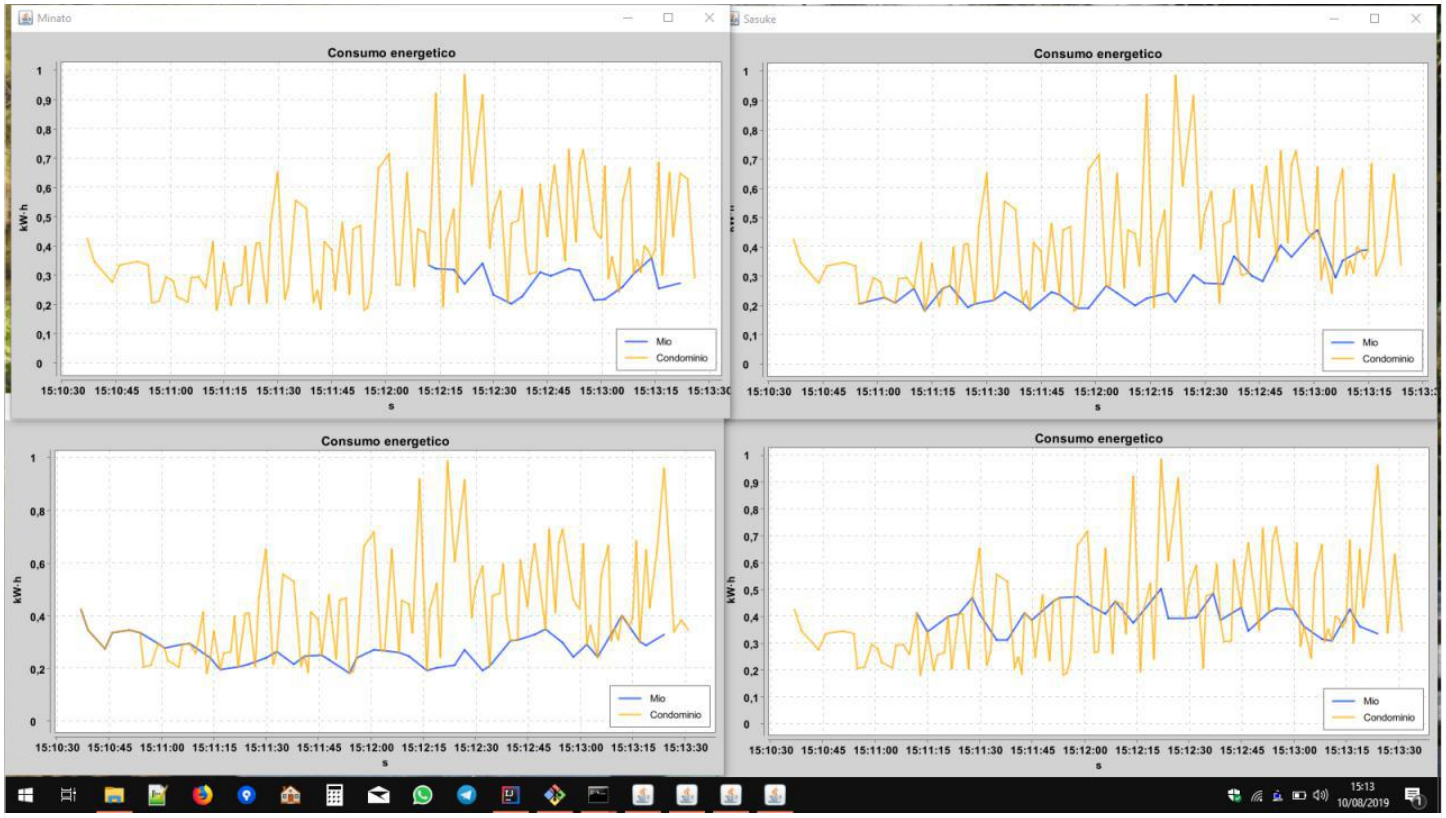

Figure 14: Grafici a linee di 4 case

2.6 – Notifiche push

Attualmente esistono vari tipi di strategie per implementare le notifiche push: Webpush, HTTP server push, Pushlet, Long polling, Flash XML Sockets relays, Reliable Group Data Delivery, Push notification.
È stato scelto di implementarle utilizzando la tecnica di Long polling.
Il client fa una richiesta al server, e il server ha due possibilità:

1. Ci sono notifiche che il client non ha riceveuto, e allora risponde.
2. Non ci sono notifiche e allora non fa nulla lasciando il canale di comunicazione aperto, fintantoché non ci saranno notifiche disponibili, a quel punto si comporterà come 1.

Il client nel caso 1 riceve subito i dati, e rifà immediatamente la stessa richiesta al server, finendo nel caso 2, in cui rimarrà il canale di comunicazione aperto attendendo una risposta.

```
private void start()
{
    GB.loopUntil(() ->
    {
        _gRPCtoRESTserver.getNotifiche();

        synchronized (sharedVars)
        {
            return !inEsecuzione;
        }
    });
}
```

Figure 15: Procedura di Long polling lato client

```
@GET
@Path("/getNotifiche")
@Produces("application/x-protobuf")
public PushNotification.notificaRes getNotifiche()
{
    var R = PushNotification.notificaRes.newBuilder()
            .setStandardRes(buildStandardRes());

    GB.waitfor(() ->
    {
        synchronized (notifiche)
        {
            if (notifiche.size() > 0)
            {
                for (var n : notifiche)
                    R.addNotifiche(PushNotification.notifica.newBuilder()
                            .setData(n.getKey().getTime())
                            .setMsg(n.getValue())
                    );

                notifiche.clear();
                return true;
            }

            return false;
        }
    }, 500);

    return R.build();
}
```

Figure 16: Metodo REST per ottenere le notifiche utilizzando Protocol Buffers e la tecnica del Long polling