Università di Pisa – CdL in Ingegegneria Informatica

Progetto di Reti Informatiche

Documentazione di Pietro Tempesti – mat. 564673

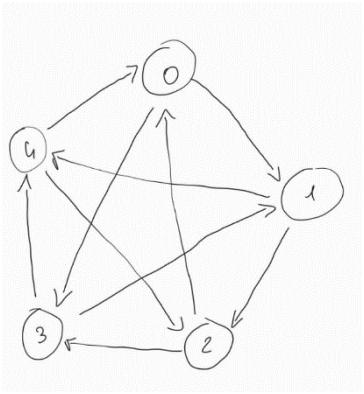
1. Distribuzione dei neighbor

Il Discovery Server, ad ogni connessione e disconnessione dei peer, aggiorna una lista ordinata per numero crescente di porta di strutture dati `peer`:

```
struct peer{
    struct sockaddr_in addr;   //indirizzo del peer
    struct peer* shortcut;     //Puntatore al peer "shortcut"
    struct peer* next;         //puntatore all'elemento successivo in lista
    uint16_t pos;              //posizione del peer della lista
    int dirty;                 //Tiene traccia dell'aggiornamento della lista
};
```

I neighbors saranno quindi il peer *next,* ovvero il peer successivo in lista, e il peer *shortcut,* calcolato come il peer di posizione (*pos + tot_peers/2) % tot_peers*; assumendo che i peer siano disposti circolarmente, ogni peer avrà come neighbors il peer immediatamente successivo e il peer diametralmente opposto.

Con questo approccio in fase di flooding dovremo fare un massimo di tot_peers/2 hops, anche se in un ambiente reale la comunicazione col peer *shortcut* prevederà una latenza maggiore rispetto a quella con il peer *next.*



Si è scelto di aggiornare sempre i neighbors ad ogni modifica della rete: questo permette di avere sempre aggiornato lo stato della rete ed un migliore bilanciamento delle operazioni di flooding al costo di maggiori comunicazioni con il DS, che aggiornerà più volte i neighbors.

2. Comunicazioni col DS

Le comunicazioni tra il DS ed i peer si svolgono sempre mediante protocollo UDP

Boot di un peer e aggiornamento dei neighbors

- Il peer invia un messaggio `PEER` (il proprio numero di porta) al DS;
- Il DS aggiorna la lista dei peer e i neighbors assegnati ad ogni peer; dopodiché invia ad ogni peer i cui neighbors sono cambiati (dirty == 1) un messaggio `IP:PORT IP:PORT` con IP e porta dei due neighbors assegnati (nel caso in cui non ci siano neighbors da assegnare si invia la coppia di valori 0:0).
- I peer (tra cui il peer che sta eseguendo il boot) ricevono la comunicazione dei nuovi neighbors, se li salvano e rispondono con lo stesso messaggio in echo

Stop di un peer

- Il peer invia il messaggio STOP [T] [N] al suo peer next (mediante TCP) per inoltrargli i dati raccolti durante la giornata (invia solo al peer next per evitare duplicazione dei dati) T, N numero di tamponi/nuovi casi raccolti e non salvati nel register.
- Il peer invia il messaggio STOP al DS, che lo rimuove dalla lista dei peer, risponde con un echo al peer in fase di stop e aggiorna i neighbor dei peer secondo l'algoritmo sopracitato.

Comando ENDD: chiusura dei register della giornata

- Alle ore 18 il DS invierà a tutti i peer connessi un messaggio ENDD, che indica ai peer di salvare nel register le entries della giornata.

- I peer ricevono l'istruzione, aggiornano il register ed inviano un echo come acknowledgement

Comando QUIT: spengimento del DS

- Quando viene eseguito il comando ESC il DS invia ad ogni peer il messaggio QUIT: questi salvano le proprie entries nel register, rispondono con un echo e si spengono.
- Quando tutti i peer hanno risposto, il DS si spenge.

3. Comunicazione tra i peer e gestione dei valori aggregati

I valori aggregati sono salvati come una lista di `struct aggr`:

```
struct aggr{
    struct tm d1, d2; //date di inizio e fine periodo
    char type; //tipo di valori (T o N)
    char aggr_type; //Tipo di valore aggregato (T o V)
    char* value; //Stringa di valori (totali giornalieri o variazioni)
    struct aggr* next; //puntatore alla struttura successiva in lista
};
```

Si utilizza una stringa di valori (formattati [v1] [v2] …) al fine di potersi ricalcolare anche sottoinsiemi di valori (es. posso dedurre i valori del periodo 11:04:2021-12:04:2021 da quelli del periodo 10:04:2021-13:04:2021).

Le entries usate dai peer e successivamente salvate nel register si basano sulla `struct entry`:

```
struct entry{
    struct tm data;
    char type;
    int value;
};
```

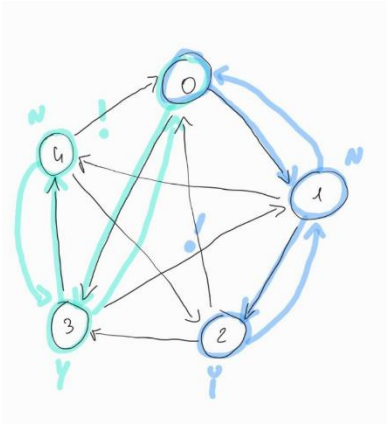
Tutte le comunicazioni tra i peer si basano sul protocollo TCP e sono precedute dalla comunicazione della dimensione del messaggio.

Messaggio REQUEST_DATA

`REQD [peer] [aggr_T] [T] [data1-data2]`: specifica che dati aggregati sto cercando, viene inviato da un peer ad entrambi i suoi neighbors

Messaggio REPLY_DATA

`REPD [Y/N] [v1 v2 …]`: Risposta che specifica se il neighbor ha (Y) o meno (N) i valori aggregati richiesti; se li ha li specifica in coda al messaggio.



Messaggio FLOOD_FOR_ENTRIES

`FFEN [peer] [T] [data1-data2], [peer1]:[Y/N] [peer2]:[Y/N]…` Il peer richiedente lo invia ai propri neighbors: loro controllano se hanno l'informazione e la propagano al proprio next, a meno che il flooding non si sia concluso (tra i miei neighbor ho il peer richiedente); quando il flooding si è concluso il messaggio viene rispedito indietro fino al peer richiedente che invierà le REQUEST_ENTRIES

Messaggio REQUEST_ENTRIES

REQE [peer] [T] [data1-data2] Il peer richiedente invia questo messaggio a tutti i peer che hanno risposto positivamente durante il flooding. Il peer che riceve questo messaggio risponde con un messaggio [v1 v2 …] con tutte le entries relative ai periodi (se non ha valori per un giorno inserisce uno 0). Il messaggio di risposta avrà sempre un numero di valori pari ai giorni del periodo.

Altre note relative al meccanismo di aggregazione

Poiché i valori saranno frammentati tra i vari peer (ogni giorno il valore totale di tamponi e nuovi casi è quello derivante dalla somma dei valori presenti su tutti i peer connessi nel momento del messaggio di ENDD o allo spegnimento del DS), dovrò sempre eseguire il flooding se il valore aggregato non è stato già calcolato o non è deducibile da altri valori calcolati dal peer richiedente e dai suoi neighbors, aumentando il traffico di dati, ma al contempo garantendo sempre la coerenza di questi (quando eseguo il flooding e la richiesta di entries il valore aggregato calcolato sarà basato su tutte le entries al momento disponibili).