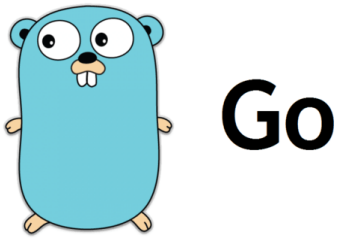
Documentazione Websocket Server

Marioemanuele Ghianni, DISIT Lab, University of Florence 08/08/2018

Introduzione

La documentazione esposta nelle successive righe del seguente documento è relativa al Websocket server sviluppato per la gestione delle connessioni tramite NodeRed, l'api snap4city e per la gestione delle relative dashboard. Il codice del server è stato scritto in linguaggio Go partendo da un'implementazione base precedente in php e possiede un sistema di log per ogni errore riscontrabile in modo da facilitare il debug. È stato scelto Go come linguaggio perchè permette un'ottima efficienza, rapidità di esecuzione e concorrenza, poco consumo di memoria e un fattore di scalabilità maggiore rispetto all'implementazione php oltre ad essere un linguaggio facilmente assimilabile e manutenibile. Il server supporta inoltre Redis come Pub/Sub broker per una futura implementazione a cluster in prospettiva di scalabilità. Per maggiori dettagli, visitare: `https://hashrocket.com/blog/posts/websocket-shootout`, `https://hackernoon.com/scaling-websockets-9a31497af051` .



Librerie esterne

-**`https://github.com/gorilla/websocket`** -implementazione Websocket.
-**`https://github.com/go-sql-driver/mysql`** -driver per mySQL.
-**`https://github.com/gomodule/redigo`** - go client per Redis.
-**`https://github.com/go-ini/ini`** -parsing file ini.
-**`https://github.com/coreos/go-oidc`** -abilita il supporto all'openID connect.
-**`https://github.com/golang/oauth2`** -implementazione per la specifica OAuth 2.0.
-**`https://github.com/satori/go.uuid`** -per la generazione degli id da assegnare alle connessioni.

File: user.go

Semplice file che contiene la struct **WebsocketUser** a cui sono associate delle stringhe relative all'id, al nome della metrica e al tipo di user; contiene inoltre un widgetUniqueName di tipo interfaccia che verrà assegnato in seguito al messaggio ricevuto. Alla struct è infine associato un socket relativo alla connessione websocket e un canale send su cui saranno fatti transitare i messaggi di risposta.

File: connection.go

Il file connection contiene varie struct e la funzione per la connessione openID e il salvataggio su ownership.

WebsocketServer: struct contenente i parametri di inizializzazione del server e una lista dei clientWidgets; verrà usata nel *buildAndInit()* del file *server.go*.
Message : semplice struct necessaria per il marshal e l'unmarshal dei messaggi nel caso "AddMetricData".
callBody: struct che rappresenta il corpo della chiamata all'api ownership.
ClientManager: struct per la gestione dei widgets con 3 canali; uno per l'iscrizione, uno per la disiscrizione e l'ultimo è un canale per il broadcast.

openIDConnect(): funzione che istanzia un callBody tramite i parametri passati, prende un context di default e provvede a creare un nuovo provider. Successivamente, viene configurato lo oauth2 con un clientID, un clientSecret e un EndPoint e eseguita una Post request sull' url dell'api contenente i dati del nuovo inserimento.

File: server.go

Il file server è di importanza centrale in quanto contiene tutte le funzioni, la logica e l'inizializzazione del websocket server.

Variabili globali e costanti:

- *var mu* mutex per l'accesso alla lista dei clientWidgets.
- *var manager* istanzia un ClientManager (definito in connection.go).
- *var db* riferimento a sql.DB che verrà inizializ. in InitDB().
- *var ws* è il risultato della funzione buildAndInit().
- *var dashboard* tiene in memoria il nome del database da sostituire nelle varie query.
- *var addr* stringa di indirizzo usata dal ListenAndServe().
- *writeWait, pongWait, pingPeriod* sono costanti per i tempi di messaggistica e di ping/pong.

buildAndInit(): funzione per il parsing dei file ini; crea e ritorna un WebsocketUser inizializzando i parametri. Verrà infine assegnato alla variabile ws.
start(): goroutine lanciata dal main() che si occupa del sub/unsub; gestisce il ClientManager e la chiusura delle connessioni.
closed(): funzione di chiusura connessione; viene chiamata dalla routine start() nel momento in cui avviene un unsub. Provvede a fare l'unset dei parametri ; nel caso in cui si chiuda la connessione di un clientWidget che è rimasto solo in ascolto per i dati di una determinata metrica, si esegue l'unsubscribe del listener sul channel della suddetta metrica. Per eseguire l'unset si cerca fra gli user relativi alla metrica e quando troviamo l'user corrispondente lo mettiamo in coda alla lista, lo settiamo a nil e infine accorciamo la lista.
processingMsg() e processingMsg2(): funzioni per la decodifica dei messaggi json; la seconda salva semplicemente la decodifica su una map[string]interface mentre la prima rende anche un parsing su un particolare parametro. Sono usate entrambe all'interno del file dbProcessing.go nella funzione dbCommunication.
reader(): goroutine lanciata dall' handleConnections() attraverso un riferimento a WebsocketUser che si occupa di settare le deadline per la lettura, l'handler per il pong e, attraverso un for loop, leggere i vari messaggi in ingresso sulla connessione per poi lanciare la funzione dbCommunication() del file dbProcessing.go passando come parametro il messaggio letto e il riferimento al relativo WebsocketUser.
writer(): goroutine lanciata dall'handleConnections() attraverso un riferimento a WebsocketUser in concomitanza con quella di read; istanzia un ticker passando il pingPeriod per i ping msg e, attraverso un for loop, riceve e manda i messaggi da mandare dal canale send del relativo WebsocketUser mentre si comporta allo stesso modo per i ping msg da mandare quando segnalato dal ticker.
handleConnections(): handler per le connessioni; ogni volta che si stabilisce una nuova connessione, fà l'upgrade al protocollo websocket, istanzia un WebsocketUser con un id univoco generato dalla libreria, un socket che contiene il riferimento alla connessione e un canale send, viene registrato il nuovo client sul manager e infine, lancia su di esso una routine di write e una di read.
NOTA: In sostanza, per ogni connessione concorrente vengono generate due goroutines concorrenti (poco consumo di memoria -> le routines sono una specie di threads molto leggeri, velocità di esecuzione -> concorrenza e uso multi-core)

InitDB(): apre la connessione al database sulla variabile db che verrà poi usata da tutte le funzioni del file dbProcessing.go. La libreria sql/database assicura la safety dell'uso concorrente della suddetta connessione da parte di svariate goroutines.
startRedis(): funzione che viene lanciata tramite goroutine dal main(); fa partire il listener sul canale Pub/Sub di Redis impostando le funzioni onStart() e onMessage() e passa "newData" come channel per la registrazione.
listenPubSubChannels():funzione chiave per l'ascolto su Redis; setta i messaggi di ping, l'iscrizione ai channels e lancia una goroutine per ricevere le notifiche dal server Redis.
publish(): funzione chiamata nel caso "AddMetricData"; apre una connessione col server Redis e pubblica i nuovi dati passati su un canale corrispondente al metricName.

main(): la funzione main inizialmente setta il log in modo da mostrare riga, data e ora nelle print, setta l'address per il server Redis e il context e lancia una routine di start sull'istanza di manager. Successivamente, lancia un'altra routine di start per l'ascolto su Redis e setta l'handler per "/server", fà un log del servizio e lancia il ListenAndServe() con parametro un riferimento all' addr e nil come handler(usa quello di default).

File: dbProcessing.go

Contiene tutte le funzioni di interazione col database e la logica sui messaggi in entrata al server.
Si occupa dell' elaborazione del messaggio e della composizione del nuovo messaggio di ritorno che sarà poi gestito dal writer.

dbCommunication(): prende in ingresso il messaggio dal reader e un riferimento al WebsocketUser corrispondente. Inizialmente istanzia la risposta e fà il processing del messaggio attraverso la funzione processingMsg() poi vi è uno switch in base al msgType. Al termine viene mandata la risposta delle operazioni all'user corrispondente.

-*AddEditMetric*: viene inizialmente fatta una query di delete su Dashboard.NodeRedMetrics per cancellare la metrica precedente (se presente) e successivamente viene inserita la nuova metrica tramite una query insert poi, in base al tipo di metrica, tramite switch viene assegnato un valore al dataField che verrà usato poi per fare una insert su Dashboard.Data dei nuovi dati (se non già presenti). Succesivamente, si controlla se esiste già una dashboard con titolo uguale, altrimenti, sene crea una inserendo in Dashboard.Configdashboard, si salva la corrispondenza tra utente e dashboard nelle API tramite la funzione openIDConnect() e si aggiunge il widget tramite addWidget().

-*AddMetricData*: crea inizialmente un messaggio tramite la struct Message di cui verrà fatto il marshal e spedito tramite il manager ai vari user connessi. Con l'implementazione tramite Redis, viene chiamata la funzione publish() per pubblicare il messaggio. Succes-

sivamente, viene fatto uno switch in base al msg "metricType" dove si assegnano valori diversi alla variabile val per poi chiamare la funzione caseQuery(), adibita all'inserimento in Dashboard.Data, passandola come parametro.

-*ClientWidgetRegistration*: attraverso un publish() sul canale "newData" viene fatto il Subscribe del listener sul canale corrispondente al metricName; poi,con accesso gestito dal mutex, ottiene la lista dei clientsWidgets dal wsServer e cerca se la metrica personale su cui insiste il widget è già presente in memoria; in caso affermativo, inserisce in coda altrimenti crea una cosa e inserisce il widget.

-*DelMetric*: vengono contati quanti widgets insistono sulla metrica in esame; se sono più di uno, viene rimosso il widget istanziato in precedenza con una delete lasciando gli altri e la metrica personale, se ne è presente solamente uno, parte una transazione dove attraverso le delete si rimuovono widget, metrica e dati.

addWidget(): inizialmente ricava l'id della dashboard da Dashboard.Configwidgetdashboard poi controlla se è presente una dashboard precedente al relativo nodeId, in caso negativo procede all'inserimento con la funzione insertW(), alla creazione del nome del widget e a ritornare nome e responso, altrimenti, se la dashboard è presente, si controlla che non sia cambiata; in caso positivo si procede a cancellare il widget dalla dashboard vecchia tramite una delete e a fare l'insert tramite la funzione insertW() per poi creare il nome e ritornare come nell'altro caso.

caseQuery(): semplice query chiamate nel caso "AddMetricData"che inserisce in Dashboard.Data in base ai parametri passati.

findKey(): funzione che ricerca, nella mappa dei WebsocketUser, una particolare chiave passata nei parametri e ritorna il booleano corrispettivo in base al successo della ricerca.

insertW(): Inizialmente esegue una query per prelevare i parametri di default da un left join tra Dashboard.IconsMap e Dashboard.Widgets poi, viene fatto il processingMsg2() del defaultParametersMainWidget. Si procede verificando se il widget selezionato è di tipo "Mono" e di tipo singolo poi viene selezionato l' AUTO_INCREMENT per il calcolo del nextId, e il max(n_row + size_rows) per il calcolo del first free row. Successivamente, si costruisce il nome del widget combinando l'id della dashboard, il tipo, il nextId e eseguendo dei replace. Infine, viene fatto l'insert su Dashboard.Config_widget_dashboard e ritornato il successo assieme all' id della dashboard.

In prospettiva futura..

Il server implementato, grazie a go e alle goroutines, dovrebbe permettere un quantitativo molto alto di connessioni simultanee mantenendo una certa soglia di efficienza; tuttavia, nel codice è stato implementato il supporto a Redis in prospettiva futura per un impiego del server in cluster in modo che i vari cluster possano avere un Pub/Sub broker comune per comunicare. Per quanto riguarda la parte relativa al load balancing, HAProxy sembra attualmente la soluzione open source migliore; con un buon settaggio si può anche superare ampiamente i limiti imposti dalle porte. (`https://medium.com/@e.n.a.lartey/load-balancing-http-traffic-using-haproxy-d49494ccb9e4`) , (`https://www.linangran.com/?p=547`)

Fonti e links utili

- `https://godoc.org/`
- `https://redis.io/`
- `https://www.golang-book.com/books/intro/10`
- `https://blog.golang.org/concurrency-is-not-parallelism`
-`https://scotch.io/bar-talk/build-a-realtime-chat-server-with-go-and-websockets`
- `https://godoc.org/github.com/gomodule/redigo/redis`