Relazione progetto finale SDCC

Giuseppe Marseglia
matricola n. 0350066
Università di Roma Tor Vergata

***Abstract*—Questo documento ha lo scopo di illustrare il design, i dettagli implementativi e i risultati ottenuti di un sistema per la trasparenza alla replicazione di un servizio. In particolare il meccanismo di comunicazione è basato su chiamate a procedura remota, RPC, ed il linguaggio utilizzato è Go.**

I. Introduzione

La specifica assegnata era quella di sviluppare un progetto che attuasse la trasparenza alla replicazione e che in particolare ponesse attenzione sui seguenti punti:

- Implementare diversi servizi RPC, tra cui uno computazionalmente pesante.
- Gestire la concorrenza.
- Gestire il service discovery.
- Gestire il crash dei server.
- Progettare e valute un meccanismo di bilanciamento basato sullo stato.
- Progettare e valutare un meccanismo di riduzione della latenza di coda.
- Testare la soluzione.
- Fare il deploy attraverso Docker Compose e istanze EC2.

II. Struttura generale

A. Componenti

Il sistema è composto da tre tipi di nodi:

1) Uno o più **client**: sono i nodi che iniziano le richieste.
2) Uno o più **worker**: sono i nodi che computano le risposte alle richieste.
3) Un singolo **server**: è il nodo che si occupa del service discovery e che orchestra la comunicazione.

B. Flussi di comunicazione

Nel sistema sono presenti due principali flussi di comunicazione, basati sull'invocazione di chiamate a procedura remota, in particolare tramite **gRPC**.

Il primo, quello che realizza la trasparenza alla replicazione:

1) I client contattano il server, inviando le loro richieste.
2) Il server contatta i worker, inoltrando, dopo aver manipolato se necessario, le richieste dei client.
3) Il server raccoglie e incorpora le risposte dei worker, e invia la risposta finale al client, che è totalmente allo scuro di chi, quali e quanti nodi abbiano evaso la richiesta.

Il secondo flusso è quello che permette al sistema di autoconfigurarsi:

- I worker, all'avvio, periodicamente e all'arresto inviano un messaggio di ping al server, includendo informazioni relative al proprio stato.

C'è quindi una comunicazione bilaterale tra server e worker, che permette lo scambio di informazioni sullo stato e l'invocazione e la cancellazione delle richieste.

III. Scelte progettuali

A. Servizi implementati

I servizi implementati si basano sulle funzionalità di una basilare rete convoluzionale, liberamente ispirata alle operazioni richieste da **LeNet**.

In particolare è stata implementata la logica di:

- Applicazione di uno o più **kernel** ad una matrice, entrambi composti da `float32`, in particolare con padding dimensionato in modo che l'output abbia le stesse dimensioni dell'input e stride fissato ad `(1,1)`.
- Applicazione della funzione **sigmoid** ad ogni elemento di una matrice.
- Applicazione di un layer di **average pooling** di dimensione configurabile e stride fissato ad `(1,1)`.

I servizi sono raccolti all'interno di una singola invocazione RPC altamente configurabile, che permette la concatenazione di queste applicazioni per ottenere il risultato di un layer convoluzionale.

Una particolarità che è stata poi sfruttata nel progetto è che una richiesta per un servizio del genere **può essere scomposta in sotto richieste** e successivamente riassemblata. Questo è possibile perché la gran parte del carico computazionale è dovuta al fatto che alla matrice vengono applicati i kernel in maniera indipendente: se una richiesta chiede di applicare 6 kernel diversi, questa può essere suddivisa in 2 richieste che chiedono di applicare 3 kernel ciascuna.

B. Concorrenza

I serventi di gRPC sono nativamente concorrenti e l'utilizzo estensivo di goroutines e meccanismi di sincronizzazione permette:

- ai client di attendere risposte per più richieste.
- ai worker di computare le risposte per più richieste.
- al server di mantenere la corrispondenza tra client e worker per più richieste contemporaneamente.

Come verrà meglio specificato in seguito, le richieste possono essere a loro volta suddivise in sotto richieste tramite il meccanismo di bilanciamento. Questo aumenta la complessità ma anche il grado di concorrenza e parallelismo delle richieste.

C. Service discovery

Nello scenario ideale l'unica informazione che client e worker devono avere a disposizione è l'indirizzo IP del server, unico punto di accentramento.

I worker quando vengono attivati si notificano come attivi al server, indicando l'indirizzo e la porta su cui il server dovrà invocare le chiamate RPC. A seconda del contesto dove viene fatto il deployment, potrebbe essere necessario esplicitare al worker quale indirizzo debba notificare al server per essere contattato. Ad esempio, su una istanza EC2 deve essere fornito esplicitamente al worker l'indirizzo IP pubblico dell'istanza.

Le porte dei servizi possono essere configurate all'avvio, tramite flag e/o variabili d'ambiente. Lo stesso meccanismo permette di indicare tutte le configurazioni del sistema.

D. Gestione dei crash

I crash dei client non comportano nessuna conseguenza al sistema.

Il server mantiene una lista dei worker che si sono notificati come attivi, mantenendo l'indirizzo per la comunicazione e informazioni di stato. Quando viene rilevato un crash di un worker, allora questo viene rimosso dalla lista dei worker attivi e le sue informazioni rilasciate. I crash dei worker possono essere individuati in 2 modi:

1) Se il server rileva che durante l'invocazione a procedura remota un worker risulta esplicitamente come non raggiungibile.
2) Se il server rileva che il tempo passato dall'ultimo ping è maggiore di un intervallo fissato.

La motivazione dell'utilizzo di entrambi i metodi per la rilevazione del crash è che in questo modo il server è in grado di rilevarli sia in presenza di un elevato traffico di richieste sia in sua assenza.

E. Bilanciamento basato sullo stato

Naturalmente il server vorrebbe inviare la richiesta al worker meno carico.

Per modellare questo comportamento il server mantiene un costo stimato per ogni worker, il cui valore viene calcolato in base allo stato di questo.

Nell'attuale implementazione i fattori che influenzano il costo sono:

- Il numero di invocazioni RPC attive parallelamente sul worker.
- La percentuale di uso della CPU.

In particolare l'utilizzo della CPU è una metrica che in alcuni ambienti di deployment non è disponibile, come le istanze EC2. Il server si adatta automaticamente a questa possibilità ed è in grado di operare anche in ambienti in cui alcuni worker hanno a disposizione la metrica dell'utilizzo della CPU e altri non la hanno.

Le informazioni sopra citate riguardanti lo stato vengono comunicate al server dal worker come parte della richiesta di ping periodico.

Per evitare comportamenti in cui il worker considerato meno carico venga sovraccaricato di richieste se queste arrivano in burst, viene applicato un meccanismo di incremento artificiale della stima del costo del worker quando questo viene selezionato. Maggiori dettagli nella sezione di scelte implementative.

F. Riduzione della latenza di coda

La soluzione proposta per realizzare un meccanismo di riduzione della latenza di coda era quella di inviare le richieste a 2 worker contemporaneamente e:

- accettare la risposta che arriva per prima.
- annullare la computazione rimanente per la risposta che deve ancora arrivare.

Partendo da questa idea e sfruttando il fatto che la richiesta del servizio implementato può essere **scomposta in sotto richieste** è stato progettato il sistema di riduzione della latenza di coda riportato in fig-1.

Il sistema di riduzione della latenza di coda è configurabile attraverso due parametri (x, y)

1) $1 \leq x$: il numero di worker che riceveranno parte della richiesta in parallelo.
2) $1 \leq y \leq x$: il grado di replicazione della richiesta originale.

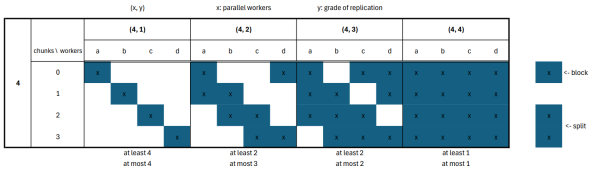

Fig. 1. Esempio di 4 possibili divisioni

Il meccanismo funziona nella seguente maniera:

1) La richiesta, in particolare la lista dei kernel da applicare, viene divisa in x blocchi che formano una partizione della lista iniziale.
2) Vengono selezionati x worker che evaderanno le sotto richieste.
3) Ad ogni worker vengono assegnati y blocchi, che vengono indicati come split. In particolare i blocchi in uno split vengono assegnati in maniera contigua per semplicità di implementazione.
4) Quando un worker termina, consegna al server lo split che ha completato.
5) Appena il server ha un sottoinsieme di split completati tali che i blocchi permettano di ottenere la lista completa dei risultati, questa viene assemblata e consegnata al client, evadendo così la richiesta originale.
6) La computazione di eventuali split ancora da completare viene annullata. Questo step è importante per non sprecare capacità di computazione.

Facendo riferimento alla figura possiamo affermare che:

- Se viene applicata la divisione $(4, 1)$ è necessario che tutti i worker completino per ottenere la lista completa. Questa soluzione è quella che massimizza la parallelizzazione della richiesta, ma è quella che minimizza la riduzione della latenza di coda.

- Se viene applicata la divisione $(4, 4)$ è necessario che un solo worker completi per ottenere la lista completa. Questa soluzione è quella che massimizza la riduzione della latenza di coda, ma è quella che minimizza l'efficacia della parallelizzazione.
- Le divisioni intermedie sono compromessi tra i due estremi, permettendo una riduzione della latenza di coda, in quanto il numero di worker che devono completare è sicuramente minore di x, ammettendo anche un certo grado di parallelismo.

Variando (x, y) si possono ottenere anche la soluzione proposta e l'assenza di un meccanismo di riduzione della latenza di coda:

- $(1, 1)$ è la soluzione in cui la richiesta viene inoltrata totalmente ad un solo worker.
- $(2, 2)$ è la soluzione proposta.

IV. Dettagli implementativi

A. Servizi implementati

La richiesta di applicazione del layer convoluzionale permette di specificare:

- Grandezza e valori della matrice di input (chiamata `target` nel codice).
- Numero, grandezza e valori delle matrici di kernel.
- Grandezza del kernel di average pool.
- Utilizzo o meno della funzione di attivazione sigmoid.
- Utilizzo o meno di valori causali, ottimi per fare testing di prestazioni, senza dover impostare manualmente le matrici.
- Utilizzo o meno di valori impostati manualmente.

Il servizio, oltre che a implementare una funzione relativamente utile, permette anche di testare il sistema sotto diversi carichi:

- Si può testare l'applicazione sotto un elevato carico di rete: questo se la grandezza dell'input viene impostata alta, ad esempio 1000, viene disattivato l'average pool, ad esempio indicandone come grandezza 1, e viene indicato un alto numero di kernel, la cui grandezza non influenza quella del risultato. gRPC pone un limite alla grandezza massima dei messaggi che può essere facilmente e affidabilmente raggiunto.
- Si può testare l'applicazione sotto un elevato carico computazionale: questo se la grandezza dell'input viene tenuta nella media, ad esempio 500, viene applicato un layer di average pool di grandezza uguale a quella dell'input, e viene specificato un numero molto alto di kernel, ad esempio 1800. In alcuni ambienti di deployment, come Docker Compose, si può facilmente saturare la memoria a disposizione.

B. Concorrenza e gestione dei crash

Poiché il meccanismo di riduzione della latenza introduce una elevata parallelizzazione della richiesta è necessario un meccanismo abbastanza complesso di gestione di goroutine concorrenti nel server.

Nel codice è possibile trovare un uso estensivo di `sync.RWMutex` e `sync.WaitGroup`, oltre che di canali per la sincronizzazione ed il passaggio dei dati tra goroutine.

In particolare, la gestione dell'evento di crash di worker assegnato ad uno split è di difficile gestione. La goroutine che si occupa della richiesta di quello split deve essere momentaneamente messa in pausa, mentre un'altra goroutine si occupa della sostituzione del worker. I dettagli sulla scelta dei worker vengono specificati in una sezione successiva.

Se la richiesta di uno split ritorna un errore fatale, l'intera richiesta viene annullata e viene restituito l'errore.

Appena gli split completati permettono di costruire la lista intera dei risultati, allora questa viene consegnata al client e le richieste ancora aperte vengono indicate ai rispettivi worker per essere annullate.

C. Bilanciamento e scelta dei worker

La primitiva che il server usa per selezionare i worker a cui inviare le richieste degli split richiede due parametri:

1) Il numero di worker da ottenere.
2) Una facoltativa lista di worker da evitare.

Supponendo che il server abbia necessità di assegnare split a x worker possono avvenire i seguenti casi:

- Nel caso migliore il server ha a disposizione un numero di worker che non sono nella lista di worker da evitare maggiore di x. In questo caso ordina per costo e seleziona gli x worker con costo minore.
- Il server potrebbe avere a disposizione un numero di worker che non sono nella lista di worker da evitare minore di x, indicato con y. In questo caso il server assegna i primi y worker con quelli che a disposizione e poi cerca di assegnare i restanti $x - y$ worker tra gli y che a disposizione. Avverrà quindi che qualche worker riceverà più split.
- Il server ha a disposizione solo worker che sono indicati nella lista di worker da evitare. In questo caso, ignora la lista, e assegna i worker.
- Il server non ha a disposizione worker. In questo caso si mette in attesa finché un nuovo worker non si segnala come attiva.

Il costo di un worker verrebbe normalmente aggiornato solo quando questo invia un segnale di ping al server, ma questo potrebbe comportare situazioni anomale di sovraccarico. Per sopperire a questo problema, quando un worker viene scelto per l'esecuzione di una richiesta di uno split, il suo costo viene incremento di un fattore 1.5. Sperimentalmente questo valore si è dimostrato adatto a garantire un buon bilanciamento delle richieste.

D. Deployment

Nell'attuale stato del progetto, il deployment di questo può essere fatto:

- Direttamente sulla macchina, tramite l'esecuzione in Go.
- In un container Docker, che può essere lanciato:
  - Manualmente.

– Tramite Docker Compose.

Questo può essere fatto anche su una istanza EC2 di AWS, e le varie soluzioni sono in grado di comunicare tra di loro correttamente.

In particolari i test sono stati eseguiti con il seguente setup:

- 1 server e 4 worker in container lanciati su istanze EC2.
- 1 slient lanciato direttamente su WSL.

V. Risultati

A. Bilanciamento del carico

Il bilanciamento del carico tra i worker è stato raggiunto e verificato sperimentalmente.

In 10 run su macchina nativa, senza container, con un client che invia 16 richieste, con 4 worker e setup $(2, 2)$, quindi un totale di 32 richieste, sono state rilevate le seguenti ripartizioni di richieste:

- $(7, 8, 8, 9)$ per 8 volte.
- $(8, 8, 8, 8)$ per 2 volte.

B. Riduzione della latenza di coda

Gli esperimenti sono stati condotti con il seguente setup:

- Server in container su EC2.
- 4 worker in container su EC2.
- 1 client su WSL che lancia per 2 volte 5 richieste da 1000 kernel ciascuna, con target di grandezza 100×100 e average pool size di 4.
- I tempi sono presi direttamente dal server per diminuire l'impatto della rete.

Le misure rilevate dei tempi di completamento delle richieste complete sono quelli riportarti in fig-2.

(1,1)		(2,2)		(4,1)		(4,2)		(4,3)	
1679	152803	**2442**	284942	**899**	1429	**2201**	12122	**3496**	52441
1828	291492	**2081**	29860	**782**	6273	**2251**	25632	**3296**	841
1795	256948	**2412**	253814	**883**	475	**2179**	7762	**3449**	33124
1829	292573	**1877**	973	**1029**	28157	**2214**	15154	**3267**	0
1017	73495	**1738**	28968	**872**	117	**2184**	8668	**3371**	10816
942	119785	**1856**	2725	**1053**	36787	**2187**	9235	**3226**	1681
990	88864	**1553**	126167	**848**	174	**2110**	365	**3359**	8464
912	141451	**1643**	70331	**1088**	51438	**2253**	26276	**3210**	3249
946	117032	**1723**	34299	**261**	360240	**1326**	585072	**3263**	16
943	119094	**1757**	22861	**897**	1282	**2004**	7552	**2733**	285156
AVG	VAR	AVG	VAR	AVG	VAR	AVG	VAR	AVG	VAR
1288	165354	1908	85494	861	48637	2091	69784	3267	39579

Fig. 2. In grassetto le misure in ms

Le misure sembrano mostrare che maggiore è il grado di parallelizzazione e migliori sono le performance in termini di tempo medio di completamento, come ci si aspettava.

Allo stesso tempo, all'aumentare del grado di replicazione la varianza dei risultati sembra diminuire, evidenziando quindi un certo potere nel ridurre la latenza di coda.

Il processo di evasione della richiesta sembra abbastanza stabile nei tempi e questo potrebbe spiegare perché l'effetto della parallelizzazione sia così marcato.

VI. Discussione

Le problematiche maggiori di questa soluzione sono:

- L'elevata complessità del codice, dovuta alla necessità di gestire e sincronizzare più goroutine concorrenti per ogni richiesta, la cui condizione di completamento non è banale.
- Il SPOF nel server. In parte è inevitabile in quanto svolge la funzione di service registry. Si potrebbe progettare di dividerlo in più componenti, ma questo aumenterebbe ancora di più la complessità del codice, già elemento problematico.
- La grande mole di dati in movimento. Si può arrivare facilmente a parlare di MB di dati per richiesta, e questi creano problemi sia per quanto la limitazione della grandezza dei messaggi gRPC sia per quanto riguarda il consumo di memoria.
- La necessità dei worker di conoscere il proprio indirizzo IP in alcuni situazioni di deployment.

VII. Conclusioni

Complessivamente credo il progetto sia stato un ottimo modo per confrontarmi direttamente con le possibilità offerte e i limiti imposti dalle tecnologie usate, quasi tutte per la prima volta:

- Go: un linguaggio veramente adattato alla concorrenza e all'efficienza, credo molto adatto per un compito del genere.
- gRPC: una tecnologia intuitiva e dal rapido utilizzo, oltre che affidabile e perfetta per il suo scopo.
- Docker: una tecnologia ormai fondamentale, che se usata a dovere permette una portabilità di codice veramente stupefacente.
- EC2 di AWS: interfaccia guidata ottima per un primo approccio e credo perfetta per casi di studio come questo.