Documentazione DALIIRC

Vittorio D'Antico

August 2024

1 Introduzione

Questa documentazione è atta a essere un documento funzionale e tecnico per il progetto DALIIRC.
DALIIRC è un'implementazione basilare del protocollo IRC (Internet Relay Chat) utilizzando il framework per sistemi MAS DALI.
In seguito mostreremo i vari requisiti funzionali e non funzionali, gli artefatti generati durante la fase di engineering del progetto per poi passare al prototipo che mostra l'implementazione del tutto come artefatto finale.

2 Requisiti

2.1 Requisiti funzionali

I requisiti funzionali sono stati stilati tenendo a mente le basi del funzionamento del protocollo IRC.

- Creazione client e server IRC
- Connessione e comunicazione tramite server
- Connessione a peer tramite server
- Comunicazione P2P
- Supporto bot proattivi

2.2 Requisiti non funzionali

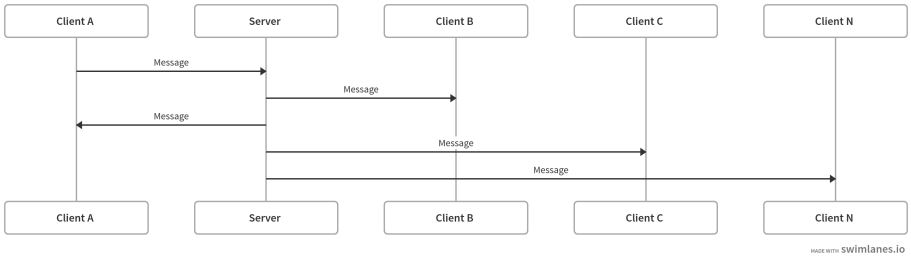
I requisiti non funzionali sono i seguenti

- Creazione dinamica agenti
- Web UI
- Comunicazione tramite prolog

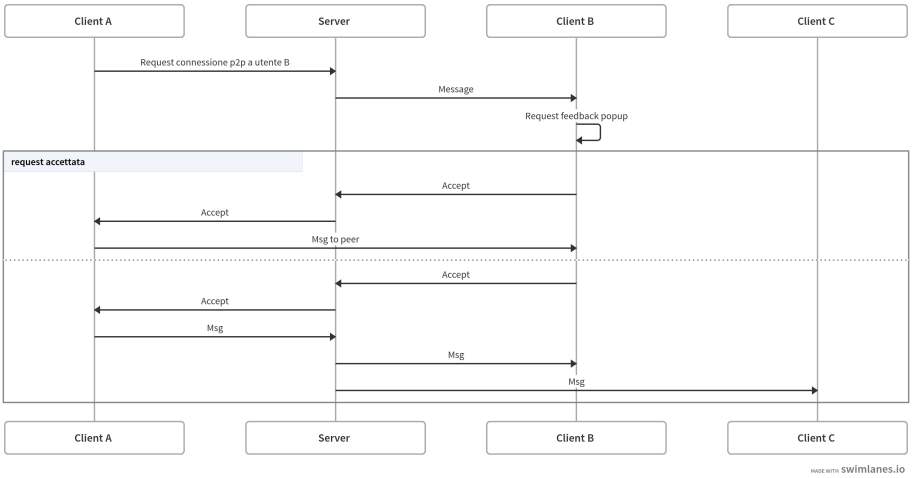
Il primo requisito serve per simulare un ambiente dinamico dove client e server vengono creati in maniera, appunto, dinamica, il secondo permette di avere un'interfaccia fruibile da diversi dispositivi mentre, il terzo, nonostante possa sembrare ovvio ad alcuni, serve per mantenere intatta la filosofia di DALI. La Web UI sarà quindi necessaria soltanto per generare chiamate verso gli agenti DALI che poi comunicheranno tra di loro in sottofondo e risponderanno alle proprie interfacce web che hanno lanciato le chiamate.

3 Engineering

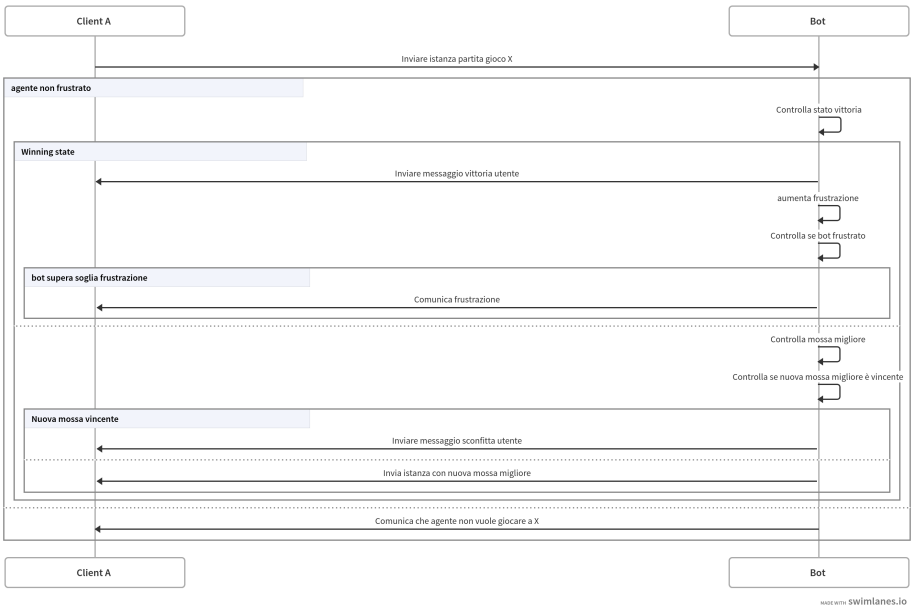
Mostreremo in questa sezione una serie di artefatti di engineering atti a mostrare come verranno realizzati i requisiti funzionali
Mostreremo alcuni diagrammi swimlane atti a mostrare alcune delle procedure più complesse. Nella prima immagine mostreremo uno swimlane diagram per la gestione di messaggi su server.



Il client contatta il server su cui è connesso con il proprio messaggio, il server farà poi un multicast verso tutti i client connessi sul server incluso il client che ha mandato il messaggio.
I client poi, a ricezione del messaggio dal server, visualizzeranno il messaggio sul proprio client.
Nel prossimo diagramma mostreremo l'inizializzazione di una connessione P2P tra due client connessi sullo stesso server.



Il client richiederà al server di inviare una richiesta di comunicazione P2P al client che vuole contattare, il server invia al client tale messaggio.
Il client target della comunicazione decide, tramite un sistema che gli permette di dare un feedback, di accettare o declinare la richiesta.
Nel caso di accettazione viene stabilito un canale di comunicazione tra i due client altrimenti la comunicazione rimane come quella mostrata nel primo diagramma swimlane.
Infine, vediamo la comunicazione e le interazioni con un bot proattivo



Assumendo che il bot sia già stato contattato dal client tramite una comunicazione P2P il client invia istanze di gioco al bot, il bot, in base a partite

precedenti, controlla il suo livello di frustrazione per il gioco in questione.
Se il bot non è frustrato si procede con il gioco altrimenti il bot suggerisce un'alternativa.
Se il livello di frustrazione di un agente raggiunge il limite dopo una sconfitta l'agente comunicherà in maniera proattiva la volontà di cambiare gioco.

4 Artefatto tecnico

Qui mostreremo l'implementazione tecnica. Questa sottosezione è divisa in quattro parti

- Premesse
- Adattamento di DALI al progetto
- Comunicazione tra prolog e browser
- Prototipo finale

4.1 Premesse

Questo documento tecnico terrà conto del funzionamento del prototipo in locale. Non ci dovrebbe essere perdita di generalità distribuendo il sistema.

4.2 Adattamento di DALI al progetto

L'interprete è stato adattato per poter essere containerizzato in un docker container per rendere l'accesso al progetto più facile.
Il progetto utilizza la versione 4.9.0 di Sicstus Prolog. Il progetto viene lanciato tramite uno script bash che scarica su una base image di Ubuntu minimal il tarball di Sicstus Prolog 4.9.0 e effettua il setup dell'intero ambiente il quale consiste in un'istanza di un tmux (terminal multiplexer) server.
Il tmux server hosterà le componenti di base, che verranno descritte tra poco, necessarie per il funzionamento dell'interprete e gli agenti che verranno creati dinamicamente il quale meccanismo descriveremo in seguito. Dividiamo quindi questa parte ulteriormente in

- Docker container
- Architettura top level
- Creazione dinamica degli agenti

4.2.1 Docker container

Prima di inizializzare il docker container è necessario inserire la propria licensa Sicstus Prolog 4.9.0 nel file installcachetoinject.cache per poter effettuare in automatico l'installazione. Nel docker container vengono installati, assieme a pacchetti comuni per ubuntu, installa Sicstus Prolog 4.9.0 per poi copiare il resto del progetto all'interno della cartella home della base image per poi far partire lo script windowlayout.sh.
Tale script è composto da tre payload

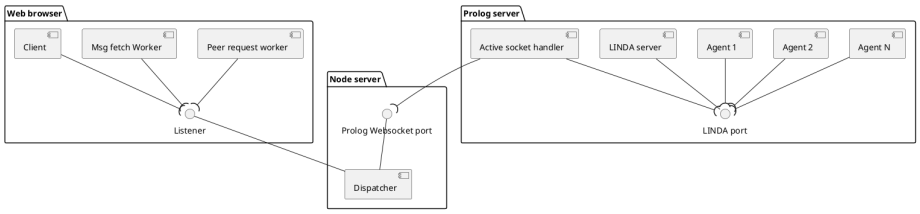
- buildup: inizializza gli agenti necessari per il funzionamento del prototipo.
- serverlaunch: fa partire il server linda.
- tmuxlaunch: lancia gli agenti necessari per il funzionamento del sistema e inizializza il server tmux.

Il container, inoltre espone le seguenti porte:

- 8080: per esporre il servizio web
- 9090/9091: per contattare/ricevere dal server nodejs utilizzato per il dispatching dei messaggi verso il server prolog.

4.2.2 Architettura top level

Qui mostriamo un diagramma architetturale del prototipo



Abbiamo a disposizione tre macroblocchi:

- Web browser
- Node server
- Prolog server

Il primo contiene le componenti del browser.
Il browser, oltre al contenimento delle componenti UI con cui un client interagisce, contiene anche due worker i quali contattano periodicamente il dispatcher per ricevere messaggi e richieste P2P; questo meccanismo è necessario per far funzionare il sistema sotto scheda di rete loopback permettendo una comunicazione su macchina locale.
Tutti i componenti usano l'interfaccia dell' listener erogata dal dispatcher del node server.

Tale dispatcher gestisce e converte i messaggi per renderli fruibili allo script prolog che si occupa del dispatching interno dei messaggi.
Nel prolog server vivono componenti statiche (sempre in running) come l'active socket handler, che si occupa di comunicare e di fare da dispatcher tra il server linda e il server node, e il LINDA server al cuore di DALI.
Gli agenti sono componenti create in un secondo momento dall'active socket handler in base a richieste browser. Vedremo questa creazione in seguito

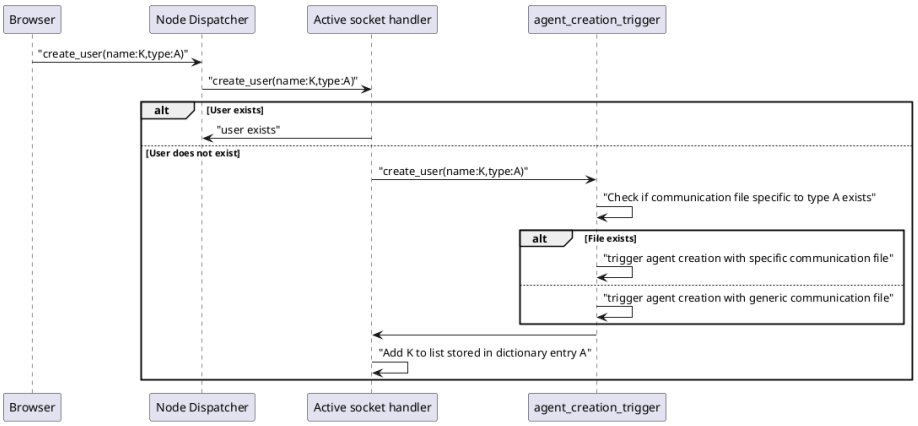
4.2.3 Creazione dinamica degli agenti

Anzitutto inquadriamo la composizione dei tipi e il come sono strutturati. L'interprete DALI è stato modificato per permettere l'aggiunta di queste due cose:

- Inizializzazione dinamica autonoma di un agente
- Communication stubs per tipi

Ipotizziamo la creazione dell'agente K di tipo A.
In DALI classico, ovvero DALI dove gli agenti sono inizializzati in base a ciò che viene scritto pre runtime, K viene inizializzato con ciò che viene scritto nel tipo A e con quello che viene scritto nel file di comunicazione senza alcun trigger che da uno start all'agente.
Questo approccio per noi non conviene in quanto preferiamo un agente autonomo, dinamico e con i propri protocolli di comunicazione. Nel DALI portato nel progetto ogni tipo presenta un punto di inizializzazione ,indicato con il nome del tipo, che permette un'inizializzazione/setup dell'agente appena un agente parte permettendo, ad esempio, di inizializzare dynamics o di inviare messaggi ad altri agenti rendendo l'agente autonomo e dinamico. Per separare i meccanismi di comunicazione, invece, dichiariamo dei communication stubs. I communication stubs sono file di comunicazione separati dal file di comunicazione principale di DALI i quali sono utilizzati per comporre un file di comunicazione che, combinato con il file di comunicazione originale, genera un file di comunicazione utilizzabile dal tipo di agente a cui lo stub è indirizzato.

Prendendo di nuovo l'esempio dell'agente A di tipo K; il nostro agente avrà nel suo tipo una sezione dedicata all'auto inizializzazione e uno stub chiamato communication_K_stub.com che conterrà i vari filtri tell/told e vari altri cambiamenti al protocollo DALI originale.
Mostreremo un sequence diagram che mostra la creazione di un agente.



4.3 Comunicazione tra prolog e browser

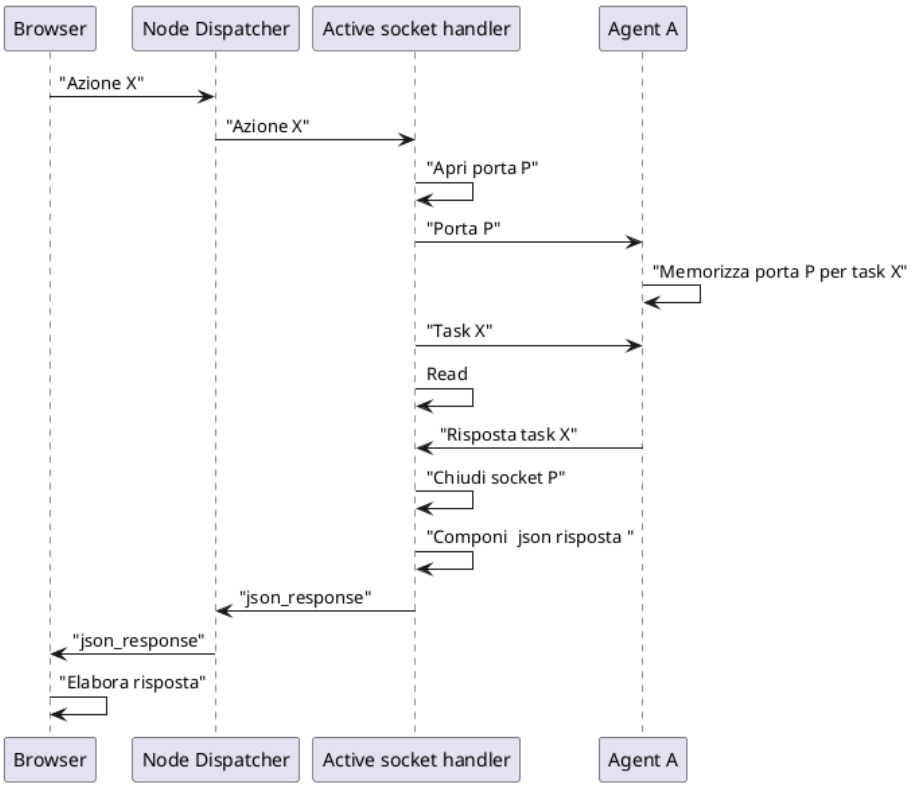
A ogni azione che può essere fatta tramite browser corrisponde uno scambio di messaggi json tra browser e server prolog. I messaggi sono composti da due parti

- action
- item

Dove l'action indica l'azione da effettuare e l'item è il payload del messaggio. Quasi tutti i messaggi sono composti in questo modo.

4.3.1 Comunicazione

Sia X un evento generato dal browser, tale evento è inviato al dispatcher di nodejs il cui compito è quello li di inviare tale chiamata verso l'active_socket_handler. L'active_socket_handler controlla il contenuto del parametro action e, in base ad esso, effettua una determinata azione. L'azione X può implicare il ritorno di dati verso il browser; nel caso ciò sia necessario viene aperta una socket per ricevere messaggi da un agente o da un altro script: ipotizziamo che la porta P viene aperta per la ricezione del messaggio: all'apertura di P si invia il numero di porta verso l'agente che deve inviare dati tramite un messaggio linda.
L'active socket handler attende i dati su quella socket mentre l'agente memorizza il numero di porta, effettua le proprie azioni e invia sulla porta P i dati necessari.
L'active socket handler formatta i dati inserendoli in un json post lettura, chiude la socket P e invia i dati alla socket verso la porta del server node aperta all'inizio della comunicazione verso il nodeserver. Il nodeserver poi si occupa di inviare il messaggio verso il browser che ha effettuato la richiesta il quale elabora i dati che riceve concludendo il ciclo.
Mostreremo qui un sequence diagram atto a mostrare questa procedura.



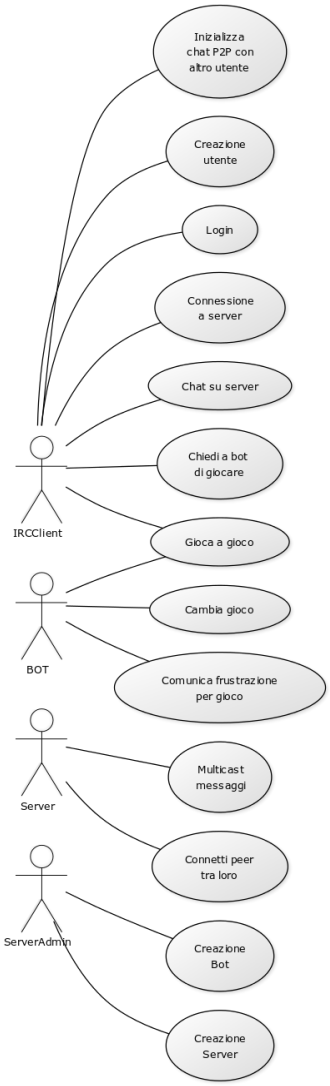
La procedura di creazione agente, nel prototipo, utilizza questa procedura di comunicazione (ovviamente senza il feedback di ritorno)

4.4 Prototipo finale

Il prototipo finale, oltre ad avere i meccanismi precedentemente illustrati, presenta tre tipi di agente:

- User
- Server
- Bot

Il primo agente simula un utente IRC normale, il secondo un server IRC mentre il terzo un Bot proattivo.
Le azioni più importanti effettuate dagli agenti sono le seguenti:



Il bot è proattivo ed è in grado di comunicare i livelli di frustrazione per i singoli giochi implementati (nel caso del prototipo nim e tris) e frustrazione totale nel caso dovesse perdere troppe volte ad entrambe i giochi. Al ricevimento di un'istanza di gioco viene fatta partire una procedura di ricerca best first per prendere la risposta migliore da restituire al browser. Dopo una determinata soglia di sconfitte per il bot ad un singolo gioco viene comunicata la volontà di non giocare e di cambiare gioco. Se il bot perde più volte anche all'altro gioco allora viene comunicata la volontà di smettere di giocare interamente. Se il giocatore prova a tentare di giocare a un determinato gioco dove la soglia di

frustrazione è alta allora il bot risponderà dicendo di essere frustrato.
La parte proattiva tiene traccia del sistema di frustrazione e comunica, tramite eventi interni, se il bot è frustrato del giocare ad un determinato gioco.

5 Conclusioni

Abbiamo mostrato in questo prototipo un'integrazione di un sistema MAS collegando un'istanza dell'interprete DALI ad una web application. L'applicativo permette un'interazione dinamica con il sistema DALI sottostante abilitando il sistema a creare agenti dinamici, autonomi, autoinizializzanti e con capacità comunicative separate per tipo. È stata migliorata l'interfaccia dell'interprete incapsulandolo all'interno di un terminal multiplexer il tutto incapsulato in un container docker di utilizzo triviale

6 Suggerimenti per il futuro

Questo lavoro potrebbe essere utile per migliorare il funzionamento di DALI: si è parlato del desiderio di un cambio di message broker durante uno dei seminari su DALI durante l'anno accademico. L'approccio di active socket handler potrebbe essere primitivo per questo scopo ma potrebbe aprire una strada verso lo sviluppo di un brokerage di messaggi diverso da LINDA. L'autore di questo documento ha pensato ad alcune soluzioni durante lo sviluppo di questo progetto come l'implementazione di un sistema publish-subscribe (per evitare la meccanicità di un semplice listener come quello illustrato nel prototipo).