# Architettura sistema

Il sistema si compone di due programmi eseguibili:

* **Client**, appena avviato effettua una connessione al server tramite TCP e si mette in attesa di istruzioni impartite da linea di comando. Digitando help è possibile accedere alla lista di tutti i comandi disponibili, ma gli unici che per primi saranno accettati dal server sono login o register. Quest’ultimo effettua una registrazione al servizio WORTH tramite RMI e in caso di successo manda automaticamente anche il comando di login al server tramite TCP. Nella procedura di login il client si registra anche alle notifiche di tipo RMI callback, che gli permetteranno di ricevere oggetti di tipo Notification. L’oggetto di tipo Notification contiene una lista degli utenti registrati a WORTH, ciascuno associato al suo stato attuale (online, offline), ed una lista dei progetti di cui l’utente fa parte, ciascuno associato all’indirizzo IP multicast della relativa chat. All’arrivo dell’oggetto il client:
  + Aggiorna la sua lista locale di coppie <username, stato>
  + Crea una nuova chat per ogni nuovo progetto a cui è stato aggiunto. Se questa è la prima notifica che riceve creerà le chat di tutti i progetti di cui è membro.
  + Elimina le chat dei progetti che sono stati eliminati
* **Server**, quando viene avviato ripristina i dati relativi ad utenti e progetti dalla cartella storage. Dopodiché gestisce le chiamate RMI tramite metodi syncronyzed e le richieste TCP tramite il multiplexing dei canali mediante NIO. Quando un client effettua il login con successo, la key del suo canale, presente nel selettore, viene salvata nell’oggetto di tipo User corrispondente all’utente per il quale ha richiesto il login. Key e User rimangono associati fino a che non viene effettuato un logout. Allo stesso modo alla richiesta di registrazione alla callback RMI, l’interfaccia remota del client viene salvata nell’oggetto User corrispondente, per poi essere cancellata al momento dell’annullamento della registrazione. Le callback RMI vengono effettuate in seguito alla creazione/cancellazione di un progetto, aggiunta di un nuovo membro ad un progetto, login di un utente.

# Threads

La concorrenza viene gestita nei seguenti casi:

* Il client ha un thread per ogni chat che rimane sempre in ascolto di nuovi pacchetti UDP sul gruppo mulitcast. Quando un messaggio arriva, lo inserisce in una struttura dati MessageQueue. MessageQueue funge da coda che viene riempita e svuotata in maniera sincrona, in modo così da non creare inconsistenze
* L’oggetto di tipo Users mantiene al suo interno la lista degli utenti registrati, che deve venir aggiornata all’invocazione di register da RMI. Potenzialmente si potrebbe creare un conflitto con le richieste TCP che hanno bisogno di accedere alla lista degli utenti. Per questo motivo i metodi della classe Users che accedono alla lista degli utenti, quali register, getByUsername e notifyAll, sono syncronyzed.

# Classi

Le principali classi sono:

* **StorageManager**, si occupa di ripristinare i dati e di aggiornare i file json nella cartella storage
* **Projects**, contiene la lista degli oggetti Project e fornisce metodi per creare/cancellare progetti, ottenere un progetto dal nome, ritornare tutti i progetti di cui fa parte un certo user
* **Project**, rappresenta un progetto e contiene una lista di oggetti Card per ogni possibile stato delle card. Fornisce metodi per spostare una card da una lista ad un’altra, ottenere info su una card, aggiungere una nuova card, aggiungere un mebro, ottenere l’indirizzo mutlicast della chat di progetto (generato casualmente alla creazione dell’oggetto)
* **Users**, contiene la lista degli oggetti User e fornisce metodi per login, logout, registrazione di un nuovo utente, mandare notifiche ed ottenere la lista degli utenti. Contiene anche una lista di coppie <key, username> che rappresenta l’associazione tra gli utenti connessi ed il loro canale di connessione
* **User**, rappresenta un utente e contiene username, password, stato (online/offline) ed interfaccia remota del client, che è diversa da null solamente se lo stato è online e viene usata dal metodo notify() per mandare una notifica tramite RMI callback
* **Notification**, viene usata per le RMI callback e contiene una lista di coppie <utente, stato> ed una di coppie <nomeprogetto, ip\_chat>
* **Response**, viene usata in risposta alle richiese TCP. Cotiene un boolean che rappresenta l’esito dell’operazione richiesta, un messaggio ed una lista opzionale di stringhe aggiuntive
* **ChatThread**, è utlizzata dal client per creare thread che rimangono in ascolto di messaggi sulla chat
* **ServerMain**, è la classe server, che esegue le richieste dei client utilizzando i metodi forniti da Projects, Users e persiste il suo stato grazie a StorageManager
* **ClientMain**, classe client che legge i comandi e li inoltra al server. Mantiene in locale una lista degli utenti, del loro stato ed una lista delle chat di progetto

# Utilizzo