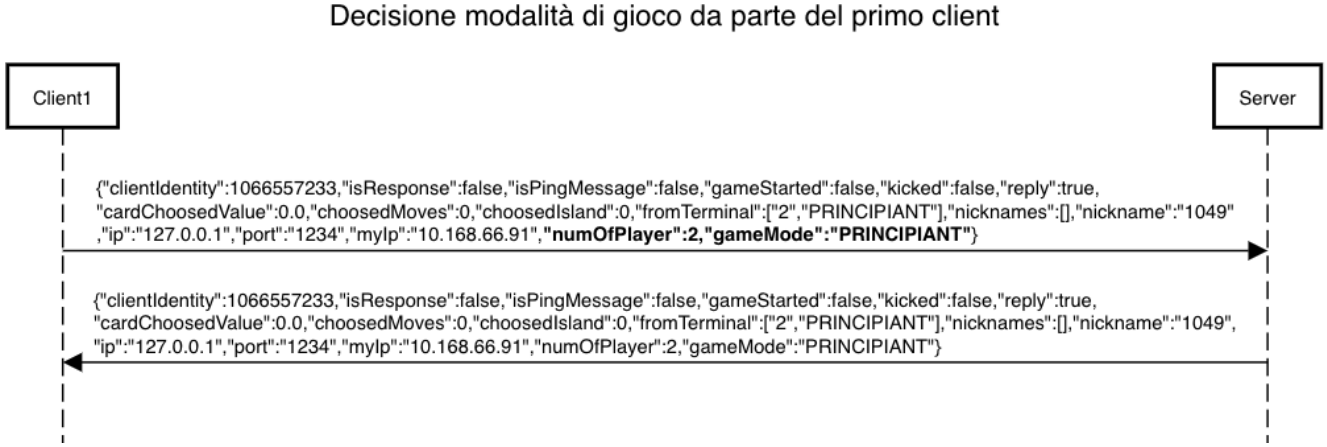
Dettagli del protocollo di rete

In questa trattazione ignoreremo i dettagli di basso livello delle socket TCP ed illustreremo il significato dei messaggi scambiati in formato json tra clients e server.

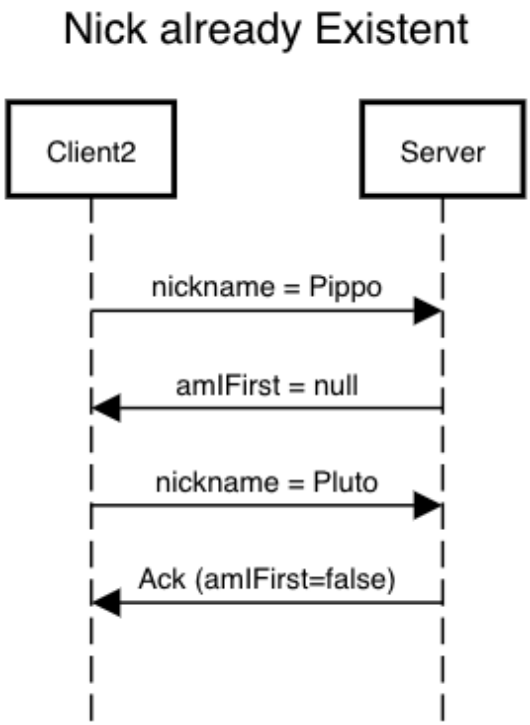
Siccome ***l'indirizzo ip*** non è sufficiente per determinare l'identità del client (il gioco deve poter essere testato sullo stesso pc), e né tantomeno il ***nickname***, in quanto all'inizio supponiamo che l'utente possa inviare per sbaglio un nickname già esistente nel server, è stata presa la decisione di inviare un numero scelto randomicamente come ***clientIdentity***. Il significato di questo campo in fase di handshake non è altro che il campo mittente.
Appena il server vedrà il primo client, risponderà con un messaggio con lo stesso campo *clientIdentity*, ma questa volta ci sarà anche un campo ***amIfirst=true*** ad indicare che il client è proprio il primo ad essersi collegato.



Successivamente il primo client invierà in chiaro il numero di giocatori e le modalità di gioco, e come ack il server reinvierà lo stesso messaggio.
A questo punto il server è in attesa dei rimanenti giocatori.

In maniera analoga avverrà l'handshake con i client successivi, la differenza sta nel campo **amIfirst=false** inviato dal server, a segnalare che il client non dovrà scegliere modalità di gioco e numero di giocatori.

Ma cosa accade in caso di inserimento di un nickname già posseduto da un altro giocatore? Ricordiamo che il server deve garantire l'unicità dello username, vediamo come è stato implementato.



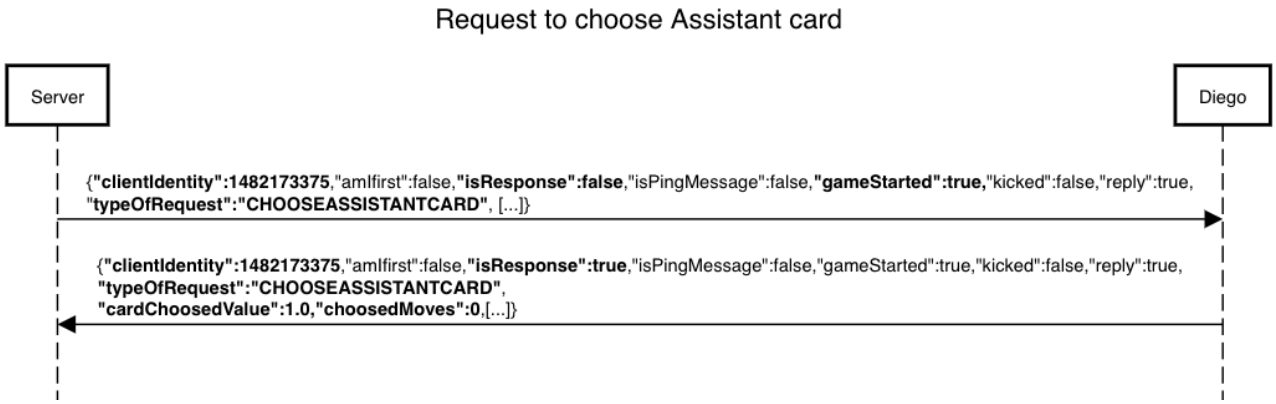
In caso di nickname già esistente il server semplicemente invia l'ack senza il campo amIfirst, in questo modo *amIfirst=null,* ed il client provvederà ad inviare un nuovo nickname.
Contestualmente, se questa volta il nickname è diverso da quelli già memorizzati nel server, il server risponderà con l'ack contenente *amIfirst=false*

Una volta che tutti i client sono collegati, verrà alzato un flag booleano *gameStarted=true*.
A questo punto l'interazione *broadcast* è la seguente:

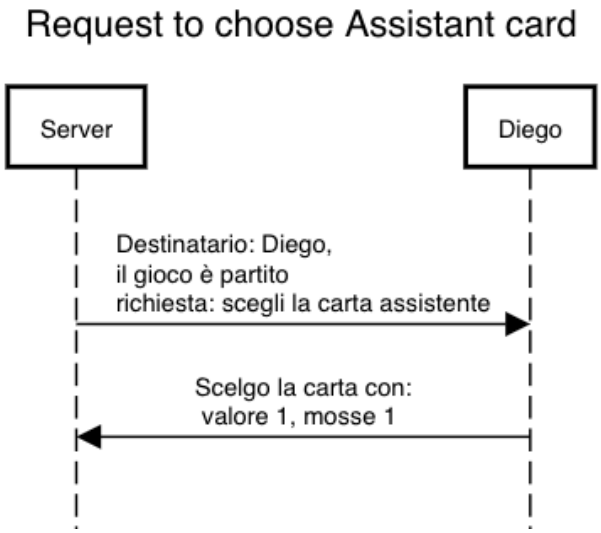
- Il server invia ad un client specifico (individuato dal campo *clientIdentity*) un comando di interazione con la command line
- Ciascun client legge questo messaggio e confronta il campo *clientIdentity* con il proprio *clientIdentity* locale. Se tale confronto risulta positivo, ovvero:
    - *my clientIdentity == received clientIdentity*

  Allora il client interpreterà tale messaggio come una **requestToMe**, ovvero richiesta del server a cui il client dovrà rispondere con una **response**.
- In caso contrario, ovvero *my clientIdentity != received clientIdentity*, il client interpreterà tale messaggio come una ***requestToOther***, e quindi non dovrà inviare alcuna response.

Vediamo nel dettaglio un'interazione:



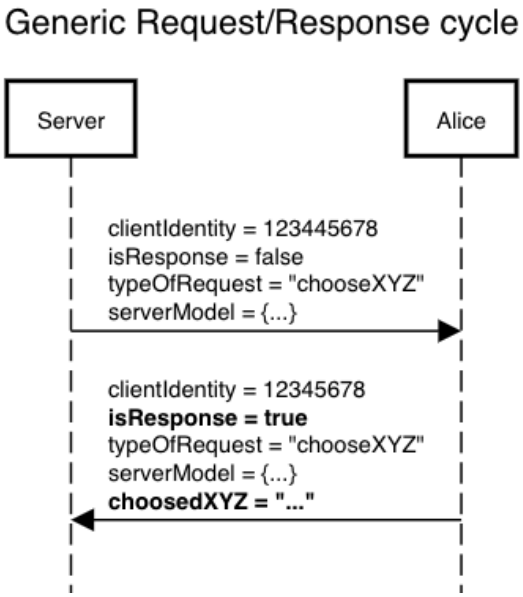
Traducendo in linguaggio *human readable*, il contenuto dei due messaggi si potrebbe tradurre in questo modo:



Nota: nel messaggio inviato dal server è presente anche tutto il model, in tal modo è possibile ricevere direttamente aggiornamenti della view contestualmente alla ricezione di richieste/risposte (risparmiando un ulteriore messaggio).

Le possibili interazioni server/client

Un modello di interazione generico una volta che il gioco è partito.
Il vantaggio della nostra implementazione consiste nel fatto che, una volta raggiunto il numero di giocatori adeguato ed incominciato il gioco vero e proprio, tutti i client si trovano in uno stato di *attesa comandi dal server*, in tal modo il server potrà cambiare solo alcuni campi del payload del messaggio, e non tutto il messaggio.



Il server modifica nel payload del pacchetto json il campo **typeOfRequest** inserendo il comando richiesto (azione da far effettuare al client), provvede poi a porre **isResponse = false**, in modo tale da segnalare che si tratta di una richiesta (un comando) e non di una risposta.

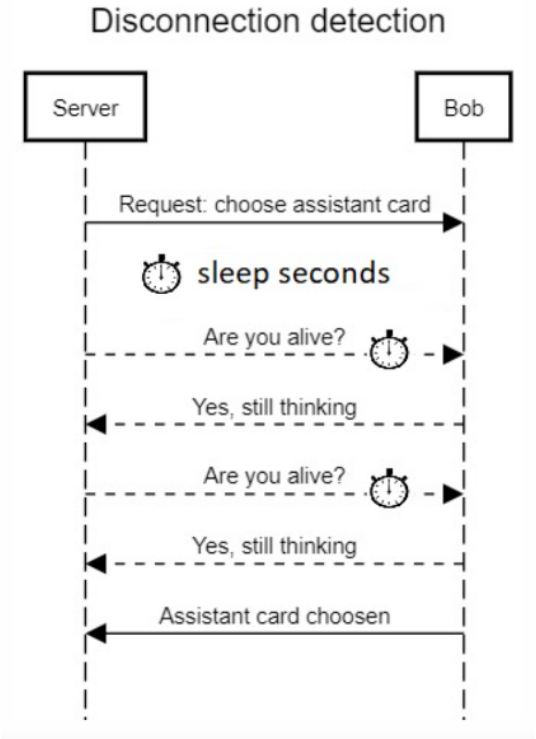
Il client provvederà a porre **isResponse = true** e compilare i campi del payload giusti, in base al tipo di richiesta.

Definiamo ora tutte le possibili tipologie di richieste che il server può inoltrare al client, e come ci si aspetta che il client risponda:

Tipo di comando (typeOfRequest)	**Risposta aspettata (campi aggiuntivi)**
TRYTORECONNECT	nessuno
DISCONNECTION	nessuno
CHOOSEASSISTANTCARD	cardChoosedValue
CHOOSEWHERETOMOVESTUDENTS	choosedColor
CHOOSEWHERETOMOVEMOTHER	choosedMoves
ISLAND	choosedIsland
CHOOSECLOUDS	cloudChoosed
GAMEEND	nessuno

Per i dettagli sul tipo di oggetto dei singoli parametri presenti nella seconda colonna, si faccia riferimento alla classe it.polimi.ingsw.client.model.ClientModel, che di fatto è l'oggetto java che viene serializzato ed inviato tramite socket.

Rilevamento delle disconnessioni, protocollo di ping (sfida e risposta)



L'idea per rilevare le disconnessioni è la seguente, se la risposta ad una request dal server tarda ad arrivare, probabilmente il client ha perso la connessione.

Per verificare ciò occorre che il software a lato client risponda in automatico a dei messaggi di ping, per escludere la possibilità che l'utente sia invece effettivamente pensando alla risposta che deve dare.

La figura a sinistra mostra quello che è il protocollo ad alto livello, dove le frecce tratteggiate sono i messaggi di ping rispettivamente del server e del client in RequestToMe.

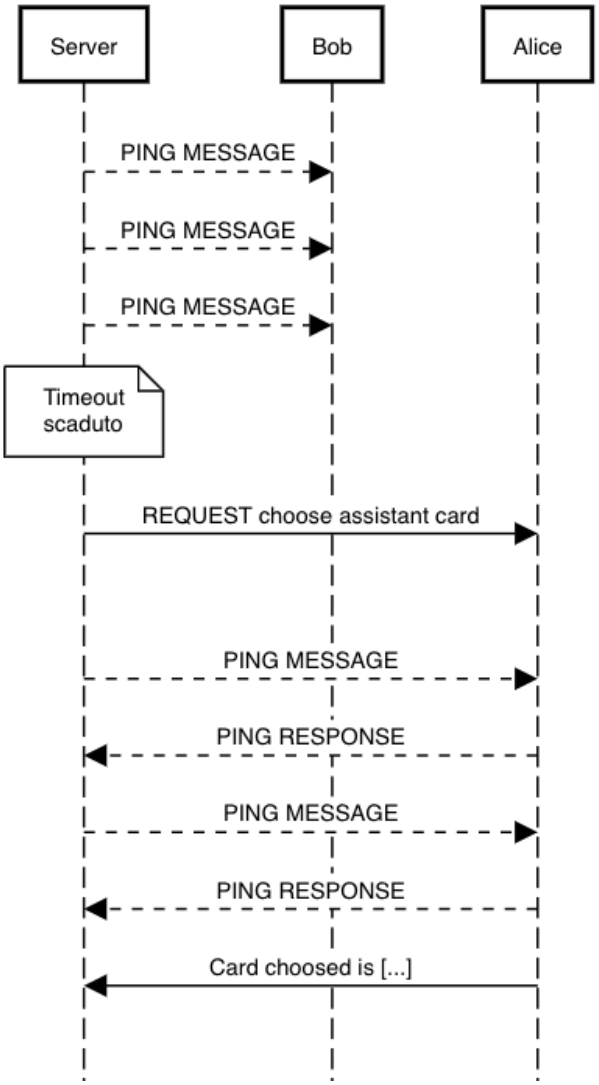
La finestra temporale entro cui considerare un client disconnesso è personalizzabile dal programmatore. Una finestra temporale di 10 secondi ci è sembrata ottimale.

Un esempio concreto:

Fino a quando l'utente non avrà inviato la risposta al server, il server continuerà a mandare messaggi di ping ed il client a rispondere con lo stesso messaggio ripetuto.

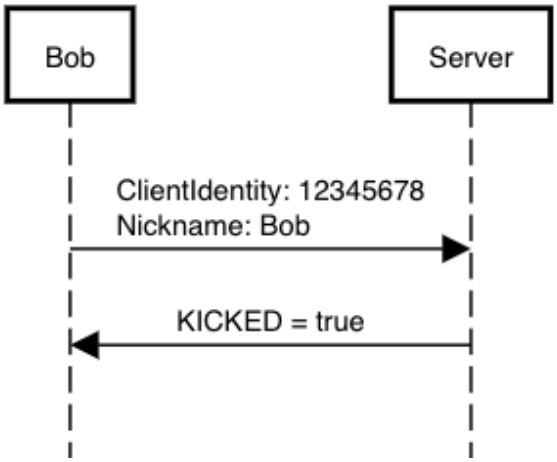
Salto di turno (per funzionalità avanzata di resilienza alle disconnessioni)

Fare riferimento ai threads di ping contenuti in it.polimi.server.controller.pingThread per vedere come è stato implementato il meccanismo qui spiegato.

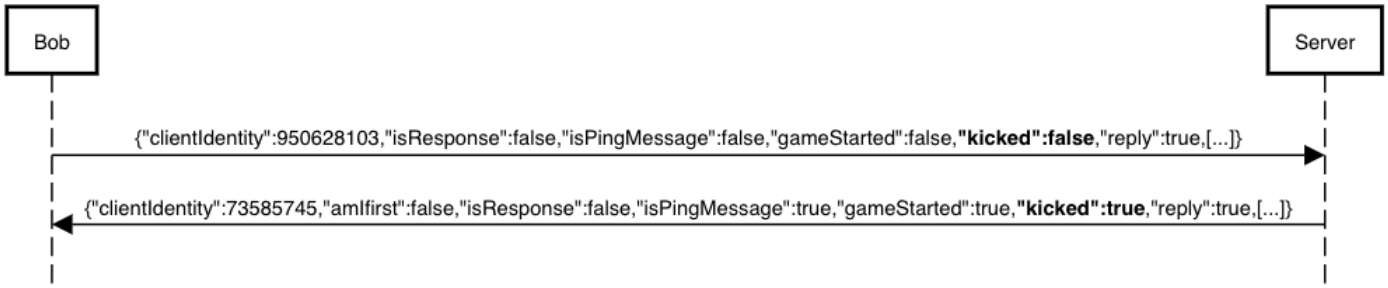


Il salto di turno viene implementato come da immagine, ovvero se un client non risponde ai messaggi di ping, dopo un certo intervallo di tempo si passa a mandare la request ad un altro giocatore (eccezion fatta nel caso in cui rimanga un solo giocatore, in tal caso non avrebbe senso mandare la partita avanti, e viene quindi offerto un intervallo temporale di 40 secondi nel quale la partita viene *congelata* e, qualora nell'arco di questi 40 secondi nessun giocatore si riconnetta, il server notifica l'unico giocatore rimanente e la partita termina.

Cacciare un client in eccesso (Kick)

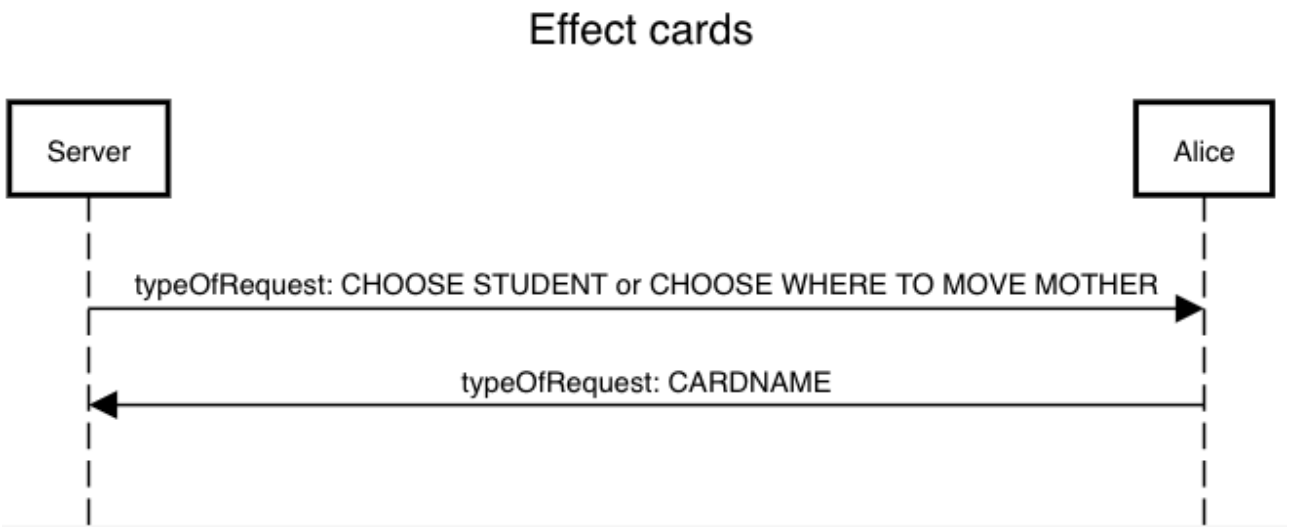


Nel caso in cui la partita arrivi a capienza completa, è necessario un meccanismo per poter notificare l'eventuale client in eccesso e cacciarlo.
Ciò viene attuato con un flag booleano "kicked", ad indicare al client che non è stato possibile aggiungerlo alla partita esistente.
Un esempio concreto:



Carte effetto

Le carte effetto possono essere attivate da parte del client solo in due momenti: o durante la scelta del posizionamento dello studente o durante la scelta di dove muovere madre natura.
Grossomodo lo schema di interazione client server è il seguente:



Uno switch case lato server provvederà a riconoscere il nome della carta e gli eventuali parametri passati dal client per la carta specifica.
Di seguito si riporta uno screenshot del codice di handling delle carte effetto a lato server (Motherphase, da riga265):

```
switch (type) {
    case "MUSHROOMHUNTER":
        ((MushroomHunter) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getChosenColor(), model.getTable());
        break;
    case "THIEF":
        ((Thief) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, model.getPlayers(), currentPlayerData.getChosenColor(), model.getTable());
        break;
    case "CENTAUR":
        ((Centaur) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, model.getTable());
        break;
    case "FARMER":
        ((Farmer) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, model.getTable(), model.getPlayers());
        break;
    case "KNIGHT":
        ((Knight) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, model.getTable());
        break;
    case "MINSTRELL":
        ((Minstrel) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getColors2(), currentPlayerData.getColors1(), model.getTable(), model.getPlayers());
        break;
    case "JESTER":
        ((Jester) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getColors2(), currentPlayerData.getColors1());
        break;
    case "POSTMAN":
        ((Postman) model.getTable().getCharacters().get(j)).useEffect(currentPlayer);
        break;
    case "PRINCESS":
        ((Princess) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getChosenColor(), model.getTable(), model.getPlayers());
        break;
    case "GRANNY":
        ((Granny) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getChosenIsland(), model.getTable());
        break;
    case "MONK":
        ((Monk) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getChosenColor(), currentPlayerData.getChosenIsland(), model.getTable());
        break;
    case "HERALD":
        boolean check = ((Herald) model.getTable().getCharacters().get(j)).useEffect(currentPlayer, currentPlayerData.getChosenIsland(), model);
        if (check) {
            gameEnd().fireStateEvent();
            return super.entryAction(cause);
        }
```