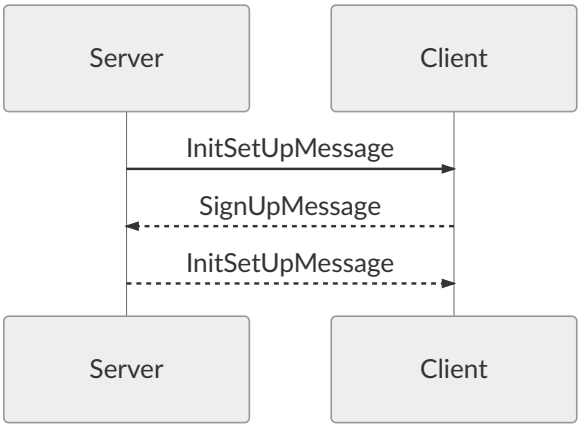
Communication protocol

We decided to use Java's native serialization to send the messages, as we know the clients will be running the JVM as well. Instead of using a single message with many properties, we rather created many types of event messages having a minimal set of properties, depending on the information possibly updated by the event.
There are six classes of event messages: **SetUpMessage, SignUpMessage, ConnectionMessage, ControllerResponse, VievEvent, ModelEvent.**
These classes have many subtypes which have properties that are specific to each one of them. To avoid a long series of `instanceof` queries, we use a Visitor pattern, implemented passing a visitor object to the `accept()` method of each event.
A detailed explanation of the event classes follows.

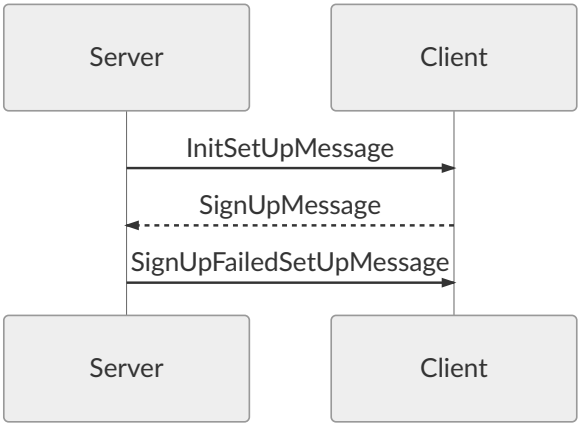
SetUpMessage and SignUpMessage

The client will attempt to open a socket to the server address on startup. If the connection is successful, the server will send a *InitSetUpMessage* with a parameter which tells the user whether he has to insert the number of players to play the game (2 or 3).

Successful signup



Failed signup



Classes

InitSetUpMessage

- SignUpParameter *START_GAME*, *NICKNAME*, *CORRECT_SIGNUP_WAIT*, *CORRECT_SIGNUP_LAST*
- Player

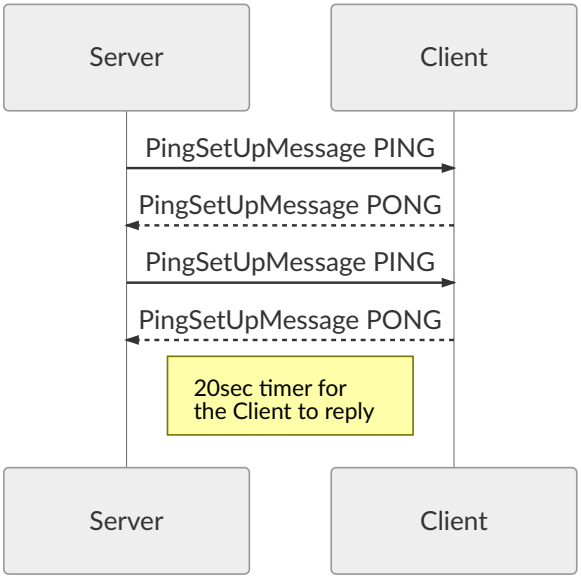
SignUpMessage

- String *Nickname*
- Integer *numPlayers*

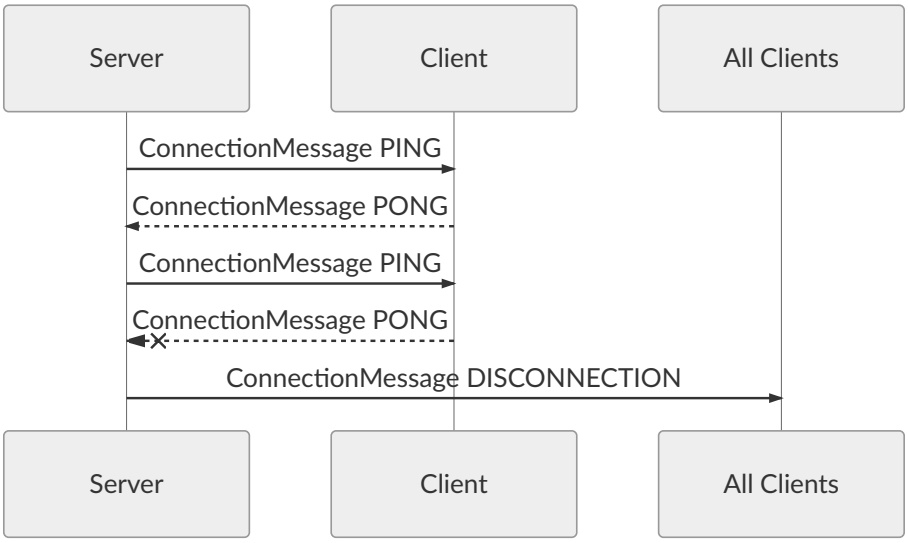
SignUpFailedSetUpMessage

- SetUpType *TOO_MANY_PLAYERS*, *SIGNUP*
- Reason *INVALID_NICKNAME*, *INVALID_NUMPLAYERS*

Ping message



If PONG is not received in 20s, the Server will declare the connection down and request disconnection to all players



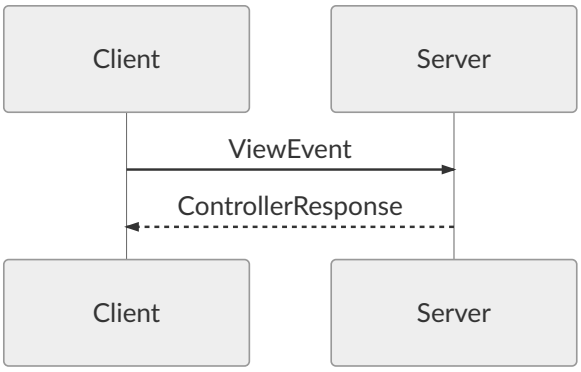
Classes

ConnectionMessage

- Type *PING*, *PONG*, *DISCONNECTION*

ViewEvent and ControllerResponse

Every significative input in the client generates a *ViewEvent*. The server immediately replies with a *ControllerResponse*.



If the *ViewEvent* couldn't successfully produce a change in the Model state, the *ControllerResponse* will have a dynamic type indicating the real reason of the failure, between:

Classes

FailedOperationControllerResponse

- WorkerViewEvent
- Operation *MOVE*, *BUILD*, *PLACE*
- Reason *NOT_ALLOWED*, *BLOCKED_BY_OPPONENT*, *NOT_FEASIBLE*, *NOT_CURRENT_WORKER*, *DESTINATION_NOT_EMPTY*

FailedUndoControllerResponse

- UndoViewEvent
- Reason *NOT_AVAILABLE*, *TIMER_EXPIRED*

IllegalCardNameControllerResponse

- CardViewEvent
- List<String> ExpectedCardNames

IllegalCardNamesListControllerResponse

- ChallengerCardViewEvent

IllegalFirstPlayerControllerResponse

- FirstPlayerViewEvent
- Reason *ALREADY_SET*, *NOT_EXISTENT*

IllegalTurnPhaseControllerResponse

- ViewEvent
- TurnPhase *requiredPhase*

NotCurrentPlayerControllerResponse

- ViewEvent

RequiredOperationControllerResponse

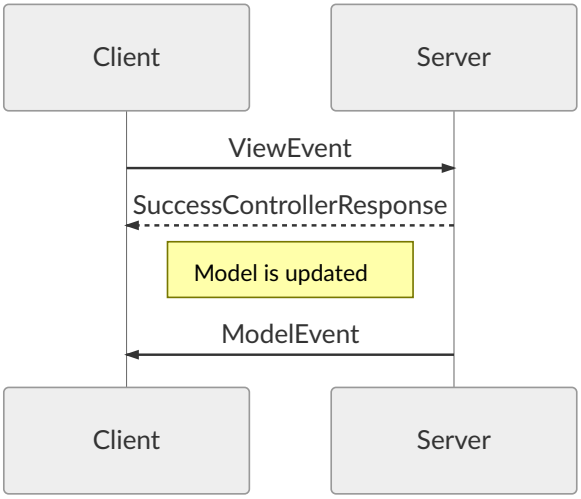
- ViewEvent
- Operation *MOVE*, *BUILD*, *PLACE*

In case of a *InfoViewEvent* the Controller will generate an object containing all the information about the moves available to the player and send it via a *TurnInfoControllerResponse*.

TurnInfoControllerResponse

- List<Position> feasibleMoves
- List<Position> feasibleBuilds
- boolean isRequiredToMove
- boolean isRequiredToBuild
- boolean isAllowedToMove
- boolean isAllowedToBuild
- boolean isUndoAvailable

If the *ViewEvent* could successfully trigger a change in the Model state, the *ControllerResponse* will be a *SuccessControllerResponse* and will be followed by a *ModelEvent*

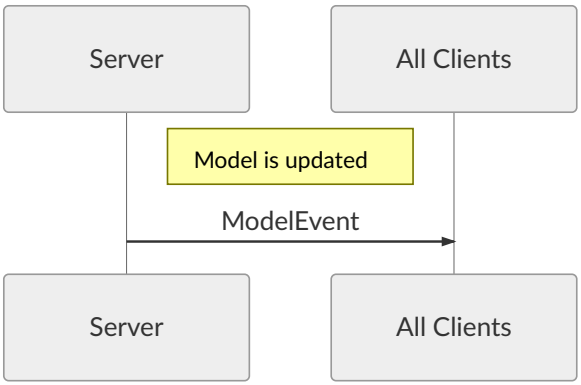


SuccessControllerResponse

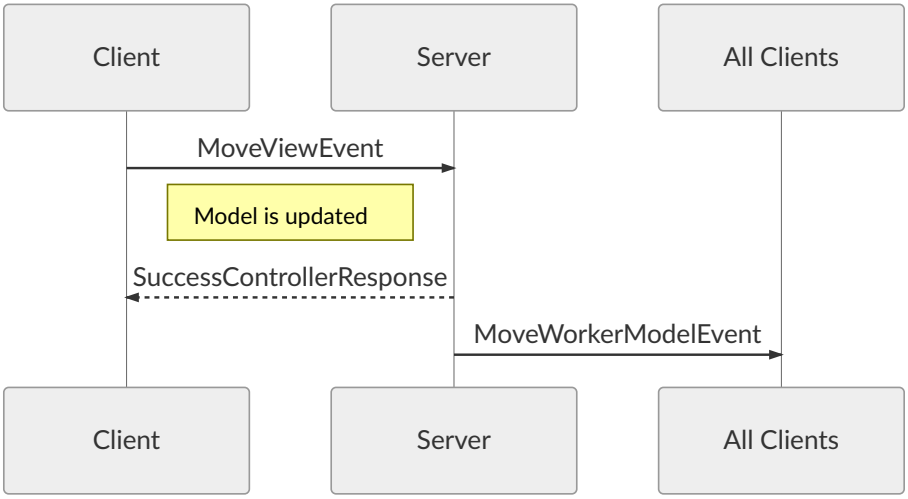
- ViewEvent

ModelEvent

Every change in the model emits a ModelEvent, which is sent to each one of the clients. The clients will then show the received updates to the user.



Example below: MoveWorkerModelEvent



Classes

BuildWorkerModelEvent

- Player
- Position *startPosition*
- Position *destinationPosition*
- boolean *isDome*

MoveWorkerModelEvent

- Player
- Position *startPosition*
- Position *destinationPosition*
- Position *pushPosition*

PlaceWorkerModelEvent

- Player
- Position placePosition

ChosenCardsModelEvent

- Player
- List<String> *ChosenCards*

FullInfoModelEvent

- InfoType *UNDO*, *INIT_GAME*, *PERSISTENCY*
- TurnPhase
- Board *board*
- List<Player> *players*
- Player *currentPlayer*

NewTurnModelEvent

- TurnPhase
- List<Player> *players*

PlayerDefeatModelEvent

- Player
- boolean *isUndoAvailable*

SetCardModelEvent

- Player
- String *cardName*

WinModelEvent

- Player

Mattia Bianchi
Alessandro Duico
Francesco Dolci