COMP361D1

M3: Requirements Specifications Models

Colt Express

Hector Teyssier (260840287)
Milo Sobral (260771581)
Mathieu Dufour (260870527)
Christina Pilip (260887943)
Lukas Liu (260839912)
Rose Liu (260843650)

Table Of Contents:

1. Architectural Decision

For our implementation, we have decided to go with a **Client-Server architecture**. This means that we will build two executables. The Client executable will be a Unity Project written in C#. The Server executable will also be written in C#. Each player will be running the client executable on their local machine and will be connected over the network to the Server executable which will reside on an Azure Virtual Machine in the cloud.

The Server executable is responsible for all of the game state for each of the players. It will always store the result of every action taken by players and will then notify the players of the information about the actions that are specifically relevant to them. The Client executable will store the minimum state that is necessary to display the information that needs to be displayed on the GUI. The Client and the Server will be communicating over the network using messages that notify which action is taken by a player. The client and the server will then stay synchronized by applying the same functions to their respective stored game state.

2. Requirements Specification Models

2.1. Structural Requirement

Environment Model (Server)

Please see the attached .pdf file `environmentModelM3Server`, which contains the diagram of the environment model for the Server - the various messages are provided in the following tables for additional reference.

Menu/Lobby/Start

Input Messages	**Output Messages**
From Client: - request (re: request_name) <<enumeration>> Request_name - finalStatus - pause - pauseOver - backup - checkMessage(m: message) - initializeNewGame() - playerLeavingServer(idg: Game ID) - changeSettings(s: Settings, v: Value) - cancelModificationRequest() - canPlayerLeaveGame() - nextPlayerAck() - nextTurnAck() - nextRoundAck()	**To Client:** - sendPlayersStatus() - messageBroadcasted() - messagesNotBroadcasted() - requestReceived() - gamePaused() - gameResumed() - gameCantBePaused - gameBackupReady() - gameBackupNotPossible() - settingsChangedAck() - modificationCanceledAck() - playerCanLeaveGameAck() - playerCanNotLeaveGameAck() - leaveGameServerAck() - initializeNewGameAck() - nextPlayer(p: Player) - nextTurn(t: TurnType) - nextRound(Integer: currentRound)
From Lobby: - gameInfo() - peopleInTheGame() - findBackupLocationAck()	**To Lobby:** - playerHasReconnected(idp: Player ID) - sendCurrentSessionID() - playerLeavingLobby(idp: Player ID) - gameInfoAck() - findBackupLocation()

Phase 1

Input Messages	Output Messages
From Client: - positionCharacterRequest(r: ActionRequest) - chooseCardRequest(r: ActionRequest) - normalWhiskeyRequest(r: ActionRequest) - oldWhiskeyRequest(r: ActionRequest) - ActionCardRequest(r: ActionRequest)	**To Client:** - characterCabin(c: Position) -stateChange(p: Phase1Action) - positionCharacterInvalid(s: String) - normalWhiskeyUsed(m:String) - normalWhiskeyInvalid(s: String) - playerChoosedCard(m: String) - oldWhiskeyUsed(m: String) - oldWhiskeyInvalid(s: String) - ActionCardDistributed(m:String)

Phase 2

Input Messages	Output Messages
From Client: - validateChosenMove(p: Position) - validateChosenRideMove(p: Position) - validateChosenRobberyLoot(l: Loot) - validateChosenBanditFire(b: Bandit) - validateChosenCharacterPunch(pc: PunchableCharacter) - validateChosenMarshalMove(p: Position) - validateChosenHostage(h: Hostage)	**To Client:** - promptMoveOptions(ps: Set{Position}) - promptRideOptions(ps: Set{Position}) - promptRobberyOptions(l: Set{Loot}) - promptFireOptions(bs: Set{Bandit}) - promptPunchOptions(pcs: Set{PunchableCharacter}) - promptMarshalMoveOptions(ps: Set{Position}) - promptHostageOptions(hs: Set{Hostage}) - refreshGameState(pa: Phase2Action)

Environment Model (Client)

Please see the attached .pdf file `environmentModelM3Client`, which contains the diagram of the environment model for the Client - the various messages are provided in the following tables for additional reference.

Menu/Lobby/Start

Input Messages	**Output Messages**
From GUI: - readRule() - getCharacterInfo() - goBack() - signIn(idp: Player ID, psw: Password) - changeSettings() - modifySettings(s: Settings; v: Value) - cancelModification() - sendChat (m: message) - chooseGame() - chosenGameAck(idg: Game ID) - createGame() - newLobbyInfoAck(n: name, mx: Maximum Player, mn: Minimum Player) - startGame() - loadGame() - chosenSavedGameAck(idg: Game ID) - openMenu() - closeMenu() - callAPause() - pauseIsOver() - saveGame() - specifyBackupLocation() - quitGame() - leavingGameAck() - notLeavingGameAck - closeProgram() - closeApplicationAck() - notCloseApplicationAck() - backToLobbyAck()	**To GUI:** - displayRules() - displayCharacterInfo() - goToPreviousWindow() - signInSucceeded() - incorrectCredentials() - displaySettings() - displayChat(m: message) - displayGameToChoose() - needMorePlayers() - chooseName() - chooseMinPlayer() - chooseMaxPlayer() - startColtExpress() - displaySavedGames() - displayMenu() - gamePaused() - pauseIsOver() - gameSuccessfullySaved() - leavingGameConfirmation() - closeApplicationConfirmation() - networkConnectionFailure() - displayResult() - goBackToLobby() - displayNextPlayer() - displayNextTurn() - displayNextRound()
From Server: - sendPlayersStatus()	**To Server:** - request (re: request_name)

- messageBroadcasted() - messagesNotBroadcasted() - requestReceived() - gamePaused() - gameResumed() - gameCantBePaused - gameBackupReady() - gameBackupNotPossible() - settingsChangedAck() - modificationCanceledAck() - playerCanLeaveGameAck() - playerCanNotLeaveGameAck() - leaveGameServerAck() - initializedNewGameAck() - nextPlayer(p: Player) - nextTurn(t: TurnType) - nextRound(Integer: currentRound) - endOfGame(gr: Game Result)	<<enumeration>> Request_name - finalStatus - pause - pauseOver - backup - checkMessage(m: message) - playerLeavingServer(idg: Game ID) - changeSettings(s: Settings, v: Value) - cancelModificationRequest() - canPlayerLeaveGame() - initializeNewGame() - nextPlayerAck() - nextTurnAck() - nextRoundAck()
From Lobby: - chosenGameAvailable() - signInSuccessful() - incorrectCredentials() - signInFailed() - gameEnterredAck() - notEnoughPlayersAck() - enoughPlayersAck() - leaveGameLobbyAck() - playersConnectedAck() - playersNotAllConnectedAck() - newLobbyCreatedAck()	**To Lobby:** - isGameAvailable() - signInRequest (idp: Player ID, psw: Password) - enterGame(idg: Game ID) - isNumberOfPlayerEnough() - playerLeavingLobby(idp: Player ID) - arePlayerConnected(idp: Player ID) - createNewLobby(n: name, mx: Maximum Player, mn: Minimum Player)

Phase 1

Input Messages	**Output Messages**
From GUI: - startColtExpress(g:Game) - positionCharacter(c:Position) - doAction(a:Phase1Action)	**To GUI:** - requestAction() - randomAction() - positionYourCharacter()

- chooseCard(c:Player) - useNormalWhiskey(w:Player) - useOldWhiskey(w:Player) - pickThreeActionCard(p:Player) - Time-triggered: timer()	- positionCharacterFail(s:String) - currentRoundCard(s:Game) - currentTurn(t:Game) - distributeCard(c:Player) - currentPosition(c:Position) - moveCard(c:Player) - returnThreeCard(c: Player) - chooseCardFail(s:String) - normalWhiskeyFail(s:String) - oldWhiskeyFail(s:String) - pickThreeCardFail(s:String)
From Server: - characterCabin(c: Position) -stateChange(p:Phase1Action) - positionCharacterInvalid(s: String) - normalWhiskeyUsed(m:String) - normalWhiskeyInvalid(s: String) - playerChoosedCard(m: String) - oldWhiskeyUsed(m: String) - oldWhiskeyInvalid(s: String) - ActionCardDistributed(m:String)	**To Server:** - positionCharacterRequest(r:ActionRequest) - chooseCardRequest(r:ActionRequest) - normalWhiskeyRequest(r:Actionequest) - oldWhiskeyRequest(r:ActionRequest) - ActionCardRequest(r:ActionRequest)

Phase 2

Input Messages	**Output Messages**
From GUI: - chosenMove(p: Position) - chosenRideMove(p: Position) - chosenRobberyLoot(l: Loot) - chosenBanditFire(b: Bandit) - chosenBanditPunch(pc: PunchableCharacter) - chosenMarshalMove(p: Position) - chosenHostage(h: Hostage) - randomHostage(h: Hostage)	**To GUI:** - displayMoveOptions(ps: Set{Position}) - displayRideOptions(ps: Set{Position}) - displayRobberyOptions(l: Set{Loot}) - displayFireOptions(bs: Set{Bandit}) - displayPunchOptions(pcs: Set{PunchableCharacter}) - displayMarshalMoveOptions(ps: Set{Position}) - displayHostageOptions(hs: Set{Hostage}) - refreshGameState(pa: Phase2Action)
From Server: - promptMoveOptions(ps: Set{Position})	**To Server:** - validateChosenMove(p: Position)

- promptRideOptions(ps: Set{Position}) - promptRobberyOptions(l: Set{Loot}) - promptFireOptions(bs: Set{Bandit}) - promptPunchOptions(pcs: Set{PunchableCharacter}) - promptMarshalMoveOptions(ps: Set{Position}) - promptHostageOptions(hs: Set{Hostage}) - refreshGameState(pa: Phase2Action)	- validateChosenRideMove(p: Position) - validateChosenRobberyLoot(l: Loot) - validateChosenBanditFire(b: Bandit) - validateChosenCharacterPunch(pc: PunchableCharacter) - validateChosenMarshalMove(p: Position) - validateChosenHostage(h: Hostage)

Concept Model (Server)

Please see the attached .zip file `conceptModelM3`, which contains the TouchCORE concept model for the Server. A .png file, `conceptModelM3Server`, of the model is also provided.

Concept Model (Client)

Please see the attached .zip file `conceptModelM3`, which contains the TouchCORE concept model for the Client. A .png file, `conceptModelM3Client`, of the model is also provided.

2.2. Behavioural Requirement

Operation Model (Server)

Menu/Lobby/Start

Operation: Server::request(request_name: String)
Scope: Client, Server
Messages: Server::{requestReceived}, Server::{sendPlayerStatus}, Server::{gamePaused}, Server::{gameCantBePaused}, Server::{gameResumed}, Server::{gameBackupReady}, Server::{gameBackupNotPossible}
Post: The Server is asked to do a special request, 4 requests possible:
- for the finalStatus request, it gathers the final Status of the game and sends to Client requestReceived() and sendPlayersStatus() messages.
- for the pause request, the game state is paused, no one can complete their action before the pause is over, Server sends to Client requestReceived() message and either gamePaused() message or gameCantBePaused() message.
- for the pauseOver request, the game state is resumed, Server sends to Client requestReceived() and gameResumed() messages.
- for the backup request, the server gathers metadata of the current game, packed it and outputs it to the client that initiated the request. Server sends to Client requestReceived() message and either gameBackupReady() or gameBackupNotPossible() messages.

Operation: Server::checkMessage(message: String)
Scope: Server, Client
Messages: Server::{messageBroadcasted}, Server::{messageNotBroadcasted}
Post: The Server receives the message that wants to be sent by a client in the chat. It checks that it does not have illegal characters. If the message is valid, he will broadcast the message to Client and send messageBroadcasted() message. If it is invalid, Server will send messagesNotBroadcasted() message.

Operation: Server::playerLeaving(ID: Integer)
Scope: Server, Client, Lobby
Messages: Server::{playerLeavingLobby}, Server::{leaveGameAck}
Post: The Server is informed that the client is leaving the game. It sends to lobby playerLeavingLobby(ID: Int) message and sends to client a leaveGameAck() message

Operation: Server::initializeNewGameAck()
Scope: Server, Client, Game
Messages: Client::{startColtExpress}
New: newGame: Game;
Post: The Server acknowledges a request to start a new game.

- The gameboard is initialized with the corresponding number of train cars to players and the locomotive.
- Five Rounds and their corresponding turn types and events are randomly picked.
- Loot and whiskeys are distributed in the train cars.
- The Marshal is placed; the Stagecoach is placed with the Shotgun and his Strongbox.
- The set of Hostages is initialized.
- The first player is picked.
- Initial loot is distributed to the players and player decks are created.

The game state is then relayed to the GUI of every Client in the game using *startColtExpress*.

Operation: Server:: nextPlayer(p: Player)
Scope: Server, Client, Game, Player
Messages: Client::{nextPlayerAck}
Post: The Server informs all Clients who the next/current player is by sending a nextPlayer message. Each Client responds that it has received the current player.

Operation: Server:: nextTurn(t: TurnType)
Scope: Server, Client, Game, TurnType
Messages: Client::{nextTurnAck}
Post: The Server informs all Clients that it is the next turn, and what the next turn type is, by sending a nextTurn message. Each Client responds that it has received the next turn.

Operation: Server:: nextRound(nextRound: Integer)
Scope: Server, Client, Game, Round
Messages: Client::{nextRoundAck; endOfGame(gr: GameResult)}
Post: The Server informs all Clients that it is the next round by sending a nextRound message. Each Client responds that it has received the current round. If we are at the end of round 5, the Server informs all Clients that it is the end of the game by sending an endOfGame message to all of them.

Other messages do not trigger anything They are either acknowledge-messages or methods whose roles have already been mentioned.

Phase 1

Operation:Server::normalWhiskeyRequest(r:ActionRequest)
Scope:Player,ActionRequest,Phase1Action
Messages: client::{normalWhiskeyUsed(s:String); normalWhiskeyInvalid(s:String); stateChange(p:Phase1Action)}
Post: Client sends the request to Server. After Server has verified the normal Whiskey action, it changes the state of the player, executes the action, and sends the successful action message and updates the state to all clients. Otherwise, if the request does not pass the verifier, it is not executed because the current turn is not a normal turn, and a fail message is sent.

Operation:Server::positionCharacterRequest(r:ActionRequest)
Scope:Player,ActionRequest,Phase1Action
Messages: client::{**characterCabin(c:position); positionCharacterInvalid(s:string); stateChange(p:Phase1Action)**}
Post: At the beginning of the game client sends the position character request to Server. After the Server verifies it, it changes the state of the player, executes the action and sends the current cabin where the current Player is back to every client. Else Server sends a fail message if Player chose to place the character on the locomotive.

Operation:Server::oldWhiskeyRequest(r:ActionRequest)
Scope:Player,ActionRequest,Phase1Action
Messages: client::{oldWhiskeyUsed(s:String); oldWhiskeyInvalid(s:string); **stateChange(p:Phase1Action)**}
Post: Client sends the request to Server. After Server verifies the old Whiskey action, it changes the state of the player, executes the action, and sends the successful action message and updated game state to all clients. Otherwise, if the request is not validated by Server, it does not execute because the current turn is not a normal turn, and a fail message is sent.

Operation:Server::chooseCardRequest(r:ActionRequest)
Scope:Player,ActionRequest,Phase1Action
Messages: Client::{playerChoosedCard(s:string); **stateChange(p: Phase1Action)**}
Post:client sends the request to Server. After the Server verifies the chosen card action, it changes the state of the player, executes the action, and sends the successful action message and the updated player state back to all clients.

Operation:Server::ActionCardRequest(r:ActionRequest)
Scope:Player,ActionRequest,Phase1Action
Messages: client::{ActionCardDistributed(s:String); **stateChange(p: Phase1Action)**}
Post: Client sends the request to Server. After Server verifies the pick 3 card action, it changes the state of the player, executes the action, and sends the successful action message and the updated player state back to all clients.

Phase 2

Operation: Server::validateChosenMove(p: Position)
Scope: Game, Player, MoveAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};
Post: The behaviour of *validateChosenMove* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player's chosen Position is valid; checking if the current Player picked an illegal Position somehow provided by the GUI) and execute it. The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenRideMove(p: Position)
Scope: Game, Player, RideAction (Phase2Action, ActionRequest), Position
Messages: GUI::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};
Post: The behaviour of *validateChosenRideMove* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked an illegal Position somehow provided by the GUI; checking Horse Positions, etc.) and execute it. The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenRobberyLoot(l: Loot)
Scope: Game, Player, RobAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};
Post: The behaviour of *validateChosenRobberyLoot* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked an illegal Loot somehow provided by the GUI) and execute it. The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenBanditFire(b: Bandit)
Scope: Game, Player, ShootAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};
Post: The behaviour of *validateChosenBanditFire* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked another Bandit out of range that was somehow provided by the GUI) and execute it (e.g. giving the targeted Bandit's Player a Bullet Card). The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenCharacterPunch(pc: PunchableCharacter)
Scope: Game, Player, Shotgun, PunchableCharacter, PunchAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};;
Post: The behaviour of *validateChosenCharacterPunch* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked a Character out of range that was somehow provided by the GUI; checking if the Loot dropped was correctly removed from the targeted Character, etc.) and execute it (e.g. changing the targeted Character's Position). The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenMarshalMove(p: Position)
Scope: Game, Player, Marshal, MoveMarshalAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};

Post: The behaviour of *validateChosenMarshalMove* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked an illegal Position somehow provided by the GUI) and execute it (e.g. giving a Bullet Card to every Bandit's Player in the Marshal's new Position and changing their Positions). The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server::validateChosenHostage(h: Hostage)
Scope: Game, Player, TakeHostageAction (Phase2Action, ActionRequest), Position
Messages: Client::{refreshGameState(pa: Phase2Action); invalidRequest_e(m: String)};
Post: The behaviour of *validateChosenHostage* is to verify that the Phase2Action (subtype of ActionRequest) coming from the Client is valid (e.g. checking if the current Player picked an illegal Hostage somehow provided by the GUI) and execute it (e.g. removing the Hostage from the remaining available Hostages). The effect of this operation is to output the result of the action on the Game's state to every Player's client. Otherwise, an invalid request error message is sent to the Client that sent this request.

Operation: Server:: refreshGameState(pa: Phase2Action)
Scope: Game, Player, (all objects affected by the Phase2Action)
New: gameboard: GameBoard;
Messages: Client::{refreshGameState(pa: Phase2Action)}
Post: The behaviour of *refreshGameState* is to have the Server refresh the game state for every Client with the validated ActionRequest; the relevant parts of the game state that have been updated on the Server are packaged and sent out to each Client in the current Game. Each Client's GUI reflects the resulting changes.

Operation Model (Client)

Menu/Lobby/Start

Operation: Client::readRule()
Scope: Client, GUI
Messages: Client::{displayRule}
Post: The player requested to read the rules of the game. displayRule() message is sent to GUI (the GUI moves to the Rule Window).

Operation: Client::getCharacterInfo()
Scope: Client, GUI
Messages: Client::{displayCharacterInfo}
Post: The player requested to read the Character information. displayCharacterInfo() message is sent to GUI (GUI moves to the Characters window)

Operation: Client::goBack()
Scope: Client, GUI
Messages: Client::{goToPreviousWindow}
Post: The player requested to go back to the previous window. goToPreviousWindow() message is sent to GUI (GUI goes back to the previous window)

Operation: Client::signIn(ID: String, Password: Integer)
Scope: Client, GUI, Lobby, Server
Messages: Client::{signInRequest}, Client::{signInSuccessful}, Client::{incorrectCredentials}, Client::{signInFailed}
Post: The player wants to sign-in to the Lobby Service with his ID and Password. Client sends signInRequest(ID: String, Password Integer) to Lobby service. Lobby Service responds with signInSuccessful() / incorrectCredentials() / signInFailed() to Client. Client displays to GUI signInSucceded() or incorrectCredentials()

Operation: Client::changeSettings()
Scope: Client, GUI
Messages: Client::{displaySettings}
Post: The player wishes to change some settings. He will specify which setting to modify after. Client sends displaySettings() to GUI, i.e. GUI goes to the Settings Window.

Operation: Client::modifySettings(s: settings; value: Float)
Scope: Client, Server
Messages: Client::{changeSettings}, Server::{settingChangedAck}
Post: The player specifies which particular setting he wants to change, specifying a particular value for this setting. Client sends changeSettings(s: settings, v: value) message to Server. Server acknowledges with message settingChangedAck().

Operation: Client::cancelModification()
Scope: Client, Server
Messages: Client::{cancelModificationRequest}, Server::{modificationCanceledAck}
Post: The player wishes to cancel the modifications that have been previously done. The settings would be reset to normal settings. Client sends cancelModificationRequest() message to Server. Server acknowledges with modificationCanceledAck() message.

Operation: Client::sendChat(message: String)
Scope: Client, GUI, Server
Messages: Client::{checkMessage}, Server::{messageBroadcasted}, Server::{messageNotBroadcasted}, Client::{displayMessage}
Post: Client receives the chat that wants to be sent by the player. It sends the message to Server that will check that it does not have illegal characters. If the message is valid, the Server will broadcast the message. Client sends checkMessage(message: String) message to Server. Server responds either with messageBroadcasted() if the message was valid or with messagesNotBroadcasted() if the message was not valid. Client sends displayChat(message: String) message to GUI.

Operation: Client::createGame()
Scope: Client, GUI, Lobby, Server
Messages: Client::{chooseName; chooseMinPlayer; chooseMaxPlayer}, GUI::{newLobbyInfoAck}, Client::{createNewLobby}, Lobby::{newLobbyCreatedAck}
Post: The player creates a game that will appear in the Lobby list of all active games on the server. Client sends chooseName(), chooseMinPlayer() and chooseMaxPlayer() messages to GUI. GUI responds with newLobbyInfoAck(n: name, mx: Maximum Player, mn: Minimum Player) message. Client sends createNewLobby(n: name, mx: Maximum Player, mn: Minimum Player) message to Lobby. Lobby creates a new Game and sends newLobbyCreatedAck() message to Client.

Operation: Client::chooseGame()
Scope: Client, GUI, Lobby Service
Messages: Client::{displayGameToChoose}, GUI::{chosenGameAck}, Client{enterGame}, Lobby::{gameEnterredAck}
Post: The player chooses a game in the list of currently opened games. Client sends displayGameToChoose() message to GUI, GUI responds with chosenGameAck(ID) message. Client sends enterGame(ID: Int) message to Lobby. Lobby puts the player in the game and sends gameEnterredAck() message to Client.

Operation: Client::loadGame()
Scope: Client, GUI, Lobby
Messages: Client{displaySavedGames}, GUI::{chosenSavedGameAck}, Client::{arePlayerConnected}, Lobby::{playersConnectedAck; playersNotAllConnectedAck},

Post: The player chooses to load a game that has previously been saved on its computer. Client shows a list of saved games to GUI, displaySavedGames() message is sent to GUI. Player chooses a game from the list, so GUI sends chosenSavedGameAck(ID: Int) message to Server. Client now asks Lobby to see if the players from this particular save are connected, arePlayerConnected(ID: Int) message sent to Lobby. If the players are all connected, Lobby sends playersConnectedAck() message to Client. If that is not the case, Lobby sends playersNotAllConnectedAck() message to Client and everyone is sent back to the Game Lobby.

Operation: Client::initializeNewGame()
Scope: Server, Client
Messages: Server::{initialiseNewGameAck}
Post: The player wants to start a new game with the players in the Lobby. Client requests that the Server initialize a new game. The Server responds with initialiseNewGameAck().

Operation: Client::startGame()
Scope: Client, GUI, Lobby Service, Server
Messages: Client::{isNumberOfPlayerEnough}, Lobby::{EnoughPlayersAck, notEnoughPlayersAck}, Client::{needMorePlayers}, Lobby::{peopleInTheGame}, Server::{initialiseNewGameAck}
Post: The player wants to start the game. Client checks with the Lobby if there are enough players in the game: isNumberOfPlayerEnough() sent to Lobby. Lobby either acknowledges with EnoughPlayersAck() message or doesn't with notEnoughPlayersAck() message. If there are not enough players, then Client sends needMorePlayers() message to the GUI. Else, Lobby sends the peopleInTheGame() message to the Server.
Then the player/Client is asked to either load a saved game via loadGame() or start a new game via initializeNewGame(). The Server will create the game.

Operation: Client::openMenu()
Scope: Client, GUI
Messages: Client::{displayMenu}
Post: The player wants to open the menu. Game execution is not paused. The client can now call a pause, save the game, read the rules, quit the game or send a chat. Client sends displayMenu() message to GUI, and GUI displays the menu objects.

Operation: Client::closeMenu()
Scope: Client, GUI
Messages: Client::{closeMenu}
Post: The player closes the menu. Client sends closeMenu() message to GUI and GUI hides the menu objects.

Operation: Client::callAPause()
Scope: Client, GUI, Server
Messages: Client::{request(pause)}, Server::{gamePaused},

Post: During the game, in the menu, the client can call a pause. This will pause the execution of the game. Client sends request(pause) message to Server. Server stops execution of the game and sends gamePaused() message to all clients enrolled in the game.

Operation: Client::pauseIsOver()
Scope: Client, GUI, Server
Messages: Client::{request(pauseOver)}, Server::{gameResumed}
Post: When the client who initiated the pause wants to resume the game. Client sends request(pauseOver) message to Server. Server resumes the execution and sends gameResumed() message to all clients

Operation: Client::saveGame()
Scope: Client, GUI, Server, LobbyService
Messages: Client::{request(backup)}, Client::{findBackupLocation}, Lobby::{findBackupLocationAck}, Server::{gameSevedAck}
Post: The player wishes to save the current state of the game. GUI asks for a new save of this particular game. The Client asks the Server for a game backup with the request(backup) message. Then, the Client asks the Lobby to find a backup location by sending a findBackupLocation() message. Lobby acknowledges with a findBackupLocationAck() message. Server saves games status at specified location. If the operation is a success, the Server sends a gameSevedAck() message to Client.

Operation: Client::quitGame()
Scope: Client, Server, GUI
Messages: Client::{canPlayerLeaveGame}, Server::{playerCanLeaveGameAck; playerCanNotLeaveGameAck}, Client::{leavingGameConfirmation}, GUI::{leavingGameAck; notLeavingGameAck}, Client::{playerLeavingServer}, Server::{leaveGameAck}, Client::{playerLeavingLobby}, Lobby::{leaveGameLobbyAck}
Post: The player wishes to quit the game. First, the Client asks if the player can quit the game by sending a canPlayerLeaveGame() message to Server. Server acknowledges or not and sends back either playerCanLeaveGameAck() message or playerCanNotLeaveGameAck() message to Client.
Client asks the GUI for a confirmation by sending a leavingGameConfirmation() message. GUI responds with either leavingGameAck() or with notLeavingGameAck(). If the player still wishes to leave, Client notifies Server with playerLeavingServer(ID: Integer) message. Server updates the list of currently enrolled players and acknowledges with a leaveGameAck() message. Similarly, Client notifies lobby and sends playerLeavingLobby(ID: Integer) message to Lobby. Lobby also updates the list of currently enrolled players and responds with a leaveGameLobbyAck() message.

Operation: Client::closeProgram()
Scope: Client, GUI
Messages: Client::{closeApplicationConfirmation}, GUI::{closeApplicationAck}

Post: The player wishes to close the application. The Client sends a closeApplicationConfirmation() message to GUI. If the player confirms that he wants to close the application, GUI sends a closeApplicationAck() message to Client. Client then closes the application. If the player changes his mind, GUI sends a notCloseApplicationAck() message to Client.

Operation: Client::endOfGame(gr: GameResult)
Scope: Client, Server, GUI
Messages: Client::{displayResult}, Client::{goBackToLobby}, GUI::{backToLobbyAck; quitGame}
Post: Client is notified by the Server that the game is over (5 rounds have passed). Client sends displayResult() message to GUI. Clients ask the player if he wants to go back to the menu. Sends goBackToLobby() message to GUI. GUI responds either with backToLobbyAck() or quitGame().

Other messages do not trigger anything. They are either acknowledge-messages or methods whose role has already been mentioned.

Operation: Client:: nextPlayerAck()
Scope: Client, GUI, Game, Player
Messages: GUI::{displayNextPlayer}
Post: The Client acknowledges to the Server that it has received the next/current Player, and updates the GUI accordingly to reflect the current Player with *displayNextPlayer*.

Operation: Client:: nextTurnAck()
Scope: Client, GUI, Game, TurnType
Messages: GUI::{displayNextTurn}
Post: The Client acknowledges to the Server that it has received the next/current type of Turn, and updates the GUI accordingly to reflect the Turn with *displayNextTurn*.

Operation: Client:: nextRoundAck()
Scope: Client, GUI, Game, Round
Messages: GUI::{displayNextRound}
Post: The Client acknowledges to the Server that it has received the next/current Round, and updates the GUI accordingly to reflect the current Round with *displayNextRound*.

Phase 1

Operation: Client::startColtExpress(g:Game)
Scope: player,Game
Messages: GUI::{currentRoundCard(s:Game);currentTurn(t:Game)}
Post:The System with the current round and corresponding turns, distributes 6 action cards to all players, and selects a new first/current player.

Operation: Client::positionCharacter(c:Position)
Scope: player,Position
Messages:
GUI::{positionYourCharacter();currentPosition(c:Position);positionCharacterFail(s:String);}
Post: In the beginning of the game, the System asks the player to choose a cabin to start with, the player can then choose the cabin. If the player choose the locomotive it will display the fail message to player and if there is network error it will display the error message.

Operation: Client::doAction(c:Phase1Action)
Scope: player,Phase1Action,Game
Messages: GUI::{requestAction();}
Post: System requests the action from the Player and player input the action he or she wants to play. If the player does not choose an action in time the system will choose a random action for the player.

Operation: Client::timer()
Scope: Player,Phase1Action,Game
Messages: GUI::{randomAction()}
Post: If the player does not choose an action in time, the system will choose a random available action for the player. Applies to Phase 1 and Phase 2.

Operation: Client::chooseCard(c:Player)
Scope: Player,Phase1Action,Game
Messages: GUI::{moveCard(c:Player);chooseCardFail(s:String)}
Post: Player does the choose card action and the system will output the chosen card to the stack and remove one card from the current cards that the player owns. Output chooseCardFail if player have connection issue.

Operation: Client::useNormalWhiskey(w:Player)
Scope: Player,Phase1Action,Game
Messages:
GUI::{returnThreeCard(c:Player);moveCard(c:Player);normalWhiskeyFail(s:String);chooseCardFail(s:String)}
Post: Player does the use normal whiskey action and the system will add and display three new action cards to player then player could play one card or the action fail if player use normal whiskey in a not standard turn and output normalWhiskeyFail.Output normalWhiskeyFail or chooseCardFail if player have connection issue.

Operation: Client::useOldWhiskey(w:Player)
Scope: Player,Phase1Action,Game
Messages: GUI::{moveCard(c:Player);oldWhiskeyFail(s:String)}
Post: Player do the use old whiskey action and the player can do the choose card action twice .System will remove two cards from the player and add it to the played card set or the action fail

because player use old whiskey in the not standard turn.Output oldWhiskeyFail if player have connection issue.

Operation: Client::pickThreeActionCard(p:Player)
Scope: Player,Phase1Action,Game
Messages: GUI::{returnThreeCard(c: Player);pickThreeCardFail(s:String)}
Post: Player do the pick three card action and the system will output and add the three card for this player.Output pickThreeCardFail if player have connection issue.

Phase 2

Operation: Client::promptMoveOptions(ps: Set{Position})
Scope: Game, Player, MoveAction (Phase2Action, ActionRequest), Position
New: actionPrompt: MoveAction
Messages: GUI::{displayMoveOptions(ps: Set{Position})};
Post: The behaviour of *promptMoveOptions* is to have the Server request that the Client display the possible movement options on the GUI, since the current Action Card in the current Player's sequence of played cards is a Move Action Card or a Floor Change Action Card. The request is relayed to the GUI by sending it an output message with the set of possible Positions to display.

Operation: Client::promptRideOptions(ps: Set{Position}))
Scope: Game, Player, RideAction (Phase2Action, ActionRequest), Position
New: actionPrompt: RideAction
Messages: GUI::{displayRideOptions(ps: Set{Position})};
Post: The behaviour of *promptRideOptions* is to have the Server request that the Client display the possible rideable horses on the GUI, since the current Action Card in the current Player's sequence of played cards is a Ride Action Card. The request is relayed to the GUI by sending it an output message with the set of possible Positions to display.

Operation: Client::promptRobberyOptions(l: Set{Loot})
Scope: Game, Player, RobAction (Phase2Action, ActionRequest), Position
New: actionPrompt: RobAction
Messages: GUI::{displayRobberyOptions(l: Set{Loot})};
Post: The behaviour of *promptRobberyOptions* is to have the Server request that the Client display the possible loot targets on the GUI, since the current Action Card in the current Player's sequence of played cards is a Rob Action Card. The request is relayed to the GUI by sending it an output message with the set of possible loot to display depending on where the Player's Bandit is.

Operation: Client::promptFireOptions(bs: Set{Bandit})
Scope: Game, Player, ShootAction (Phase2Action, ActionRequest), Position
New: actionPrompt: ShootAction
Messages: GUI::{displayFireOptions(bs: Set{Bandit})};

Post: The behaviour of *promptFireOptions* is to have the Server request that the Client display the possible Bandits in range that can be fired at on the GUI, since the current Action Card in the current Player's sequence of played cards is a Shoot Action Card. The request is relayed to the GUI by sending it an output message with the set of possible Bandits to fire at.

Operation: Client::promptPunchOptions(pcs: Set{PunchableCharacter})
Scope: Game, Player, Shotgun, PunchableCharacter, PunchAction (Phase2Action, ActionRequest), Position
New: actionPrompt: PunchAction
Messages: GUI::{displayPunchOptions(pcs: Set{PunchableCharacter})};
Post: The behaviour of *promptPunchOptions* is to have the Server request that the Client display the possible Punchable Characters in range that can be fired at on the GUI, since the current Action Card in the current Player's sequence of played cards is a Punch Action Card. Note that this may include the Shotgun character on the Stagecoach, and not just Bandits The request is relayed to the GUI by sending it an output message with the set of possible Punchable Characters to punch.

Operation: Client::promptMarshalMoveOptions(ps: Set{Position})
Scope: Game, Player, Marshal, MoveMarshalAction (Phase2Action, ActionRequest), Position
New: actionPrompt: MoveMarshalAction
Messages: GUI::{displayMarshalMoveOptions(ps: Set{Position})};
Post: The behaviour of *promptMarshalMoveOptions* is to have the Server request that the Client display the possible movement options for the Marshal on the GUI, since the current Action Card in the current Player's sequence of played cards is a Marshal Action Card. The request is relayed to the GUI by sending it an output message with the set of possible Positions to display.

Operation: Client::promptHostageOptions(hs: Set{Hostage})
Scope: Game, Player, TakeHostageAction (Phase2Action, ActionRequest), Set{Hostage}
New: actionPrompt: TakeHostageAction
Messages: GUI::{displayHostageOptions(hs: Set{Hostage})};
Post: The behaviour of *promptHostageOptions* is to have the Server request that the Client display the possible Hostages on the GUI, since the current Player's Bandit's resulting Position directly after a Move or Ride Action Card was the inside of the Stagecoach. The request is relayed to the GUI by sending it an output message with the set of possible Hostages.

Operation: Client::chosenMove(p: Position)
Scope: Game, Player, MoveAction (Phase2Action, ActionRequest), Position
Messages: Server::{validateChosenMove(p: Position)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenMove* has the Client relay the Position after the result of the Move/Floor Change Actions from the GUI to the Server by sending it an output message with the new Position.

Operation: Client::chosenRideMove(p: Position)
Scope: Game, Player, RideAction (Phase2Action, ActionRequest), Position
Messages: Server::{validateChosenRideMove(p: Position)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenRideMove* has the Client relay the Position after the result of the Ride Action from the GUI to the Server by sending it an output message with the new Position.

Operation: Client::chosenRobberyLoot(l: Loot)
Scope: Game, Player, RobAction (Phase2Action, ActionRequest), Loot
Messages: Server::{validateChosenRobberyLoot(l: Loot)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenRobberyLoot* has the Client relay the chosen Loot after the result of the Rob Action from the GUI to the Server by sending it an output message with the chosen Loot.

Operation: Client::chosenBanditFire(b: Bandit)
Scope: Game, Player, ShootAction (Phase2Action, ActionRequest), Bandit
Messages: Server::{validateChosenBanditFire(b: Bandit)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenBanditFire* has the Client relay the chosen Bandit to fire at after the result of the Shoot Action from the GUI to the Server by sending it an output message with the chosen Bandit.

Operation: Client::chosenCharacterPunch(pc: PunchableCharacter, l: Loot, p: Position)
Scope: Game, Player, Shotgun, PunchableCharacter, PunchAction (Phase2Action, ActionRequest), Loot, Position
Messages: Server::{validateChosenCharacterPunch(pc: PunchableCharacter, p: Position)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenCharacterPunch* has the Client relay the chosen Punchable Character to punch, the Loot the Player has selected them to drop, and the new Position they are left in after the result of the Punch Action from the GUI to the Server by sending it an output message with the chosen Punchable Character, Loot, and Position.

Operation: Client::chosenMarshalMove(p: Position)
Scope: Game, Player, Marshal, MoveMarshalAction (Phase2Action, ActionRequest), Position
Messages: Server::{validateChosenMarshalMove(p: Position)}
Pre: GUI sent a response within the time limit.
Post: The effect of *chosenMarshalMove* has the Client relay the chosen Position for the Marshal after the result of the Marshal Action from the GUI to the Server by sending it an output message with the new Position of the Marshal.

Operation: Client::chosenHostage(h: Hostage)
Scope: Game, Player, TakeHostageAction (Phase2Action, ActionRequest), Hostage
Messages: Server::{validateChosenHostage(h: Hostage)}

Pre: Occurs if *validateChosenMove* or *validateChosenRideMove* results in the Player's Bandit's Position being the inside of the Stagecoach and the Player does not have a Hostage yet. GUI must have sent a response within the time limit.
Post: The effect of *chosenHostage* has the Client relay the chosen Hostage from the GUI to the Server by sending it an output message with the chosen Hostage.

Operation: Client::randomHostage(h: Hostage)
Scope: Game, Player, TakeHostageAction (Phase2Action, ActionRequest), Hostage
Messages: Server::{validateChosenHostage(h: Hostage)}
Pre: Occurs if *validateChosenMove* or *validateChosenRideMove* results in the Player's Bandit's Position being the inside of the Stagecoach and the Player does not have a Hostage yet. GUI did not send a response within the time limit.
Post: The effect of *chosenHostage* has the Client relay a randomly selected Hostage from the GUI to the Server by sending it an output message with the Hostage.

Protocol Model (Server)

Please see the attached .zip file `protocolModelM3Server`, which contains the jUCMNav protocol models for the Server. A .pdf file, `protocolModelM3Server`, of the models is also provided.

Protocol Model (Client)

Please see the attached .zip file `protocolModelM3Client`, which contains the jUCMNav protocol models for the Client. A .pdf file, `protocolModelM3Client`, of the models is also provided.