Introduction and Goals

Develop a Distributed Tron Game where people can play the together while being on different machines.

The Game is a clone of the "Light Cycles Mode" in the original Tron Video Game from 1982.

Requirements Overview

Players of this game can create a room (host) or join one (guest). After joining a room the game will start. Each player controls the game with a keyboard to move his tron.

A Tron is a representation of the player and will continuously move forward. It can be moved left and right but not backward. When the Tron moves it leaves a trail behind it. The objective is to force the opponent tron into walls or trails, while simultaneously avoiding them.

If the player restarted or existed the game. The opponent wins and vise versa.

derived use-cases

ID	Use-Case	Description
UC01	create game	as a host I want to create a room to host a game
UC02	join game	as a guest I want to see available rooms to join one and play against the host
UC03	start game	as a host I want the game to start when to guest enters
UC04	play game	as a player I want to see real time updates to play the game
UC05	keyboard controls	as a player I want to control the game with a keyboard
UC06	restart game	as a player I want to restart a game by creating or joining one
UC07	exit game	• as a player I want to exit the game anytime i want • as a player i want to be notified when my opponent exited the game

Quality Goals

ID	Goal	Description
QG01	Server stability	server should not be blocked by the players
QG02	Fairness	response time should not favour one player over the other

Stakeholders

Role	Expectations

Role	Expectations
Customer	• game demo with n instances and complete documentation • fixed method for project management (proof) • fixed method for documentation (important: systematic and faithful to the method) • Protocol definition with error semantics • clear representation of the structure in at least 2 hierarchy levels: component diagram, class diagram, deployment diagram • clear representation of the behavior through sequence diagram, activity diagram, state diagram • problem-solving strategies must be derived from reference literature or accepted third-party literature • code must match the documentation and documentation must match the code • Implementation in an object-oriented language • RPC interface - Own RPC implementation no framework • musst use Dependency-inversion-principle • The use of frameworks must be approved by the customer • a maximum of 2 players per game • each player can start a new game or enter a playroom
Developer	• Understanding distributed systems in a practical way • getting PVL

Architecture Constraints

Technical Constraints

ID	Constraint	Description
TC01	Programing language	Implementation in an object-oriented language
TC02	Communication	RPC interface - Own RPC implementation no framework
TC03	Implementation	musst use Dependency-inversion-principle
TC04	Frameworks	The use of frameworks must be approved by the customer
TC05	2 players	a maximum of 2 players per game
TC06	Multiple games	N Games should be supported by the system
TC07	Multiplayer	each player can start a new game or enter a playroom
TC08	User interface	Each player gets his owen GUI
TC09	Motion	• player moves in a straight line automatically • player can manipulate direction, but can't go back
TC10	Keyboard control	player can use the keyboard to control the game
TC11	Resources usage	musst not use more than 80% of the machine resources

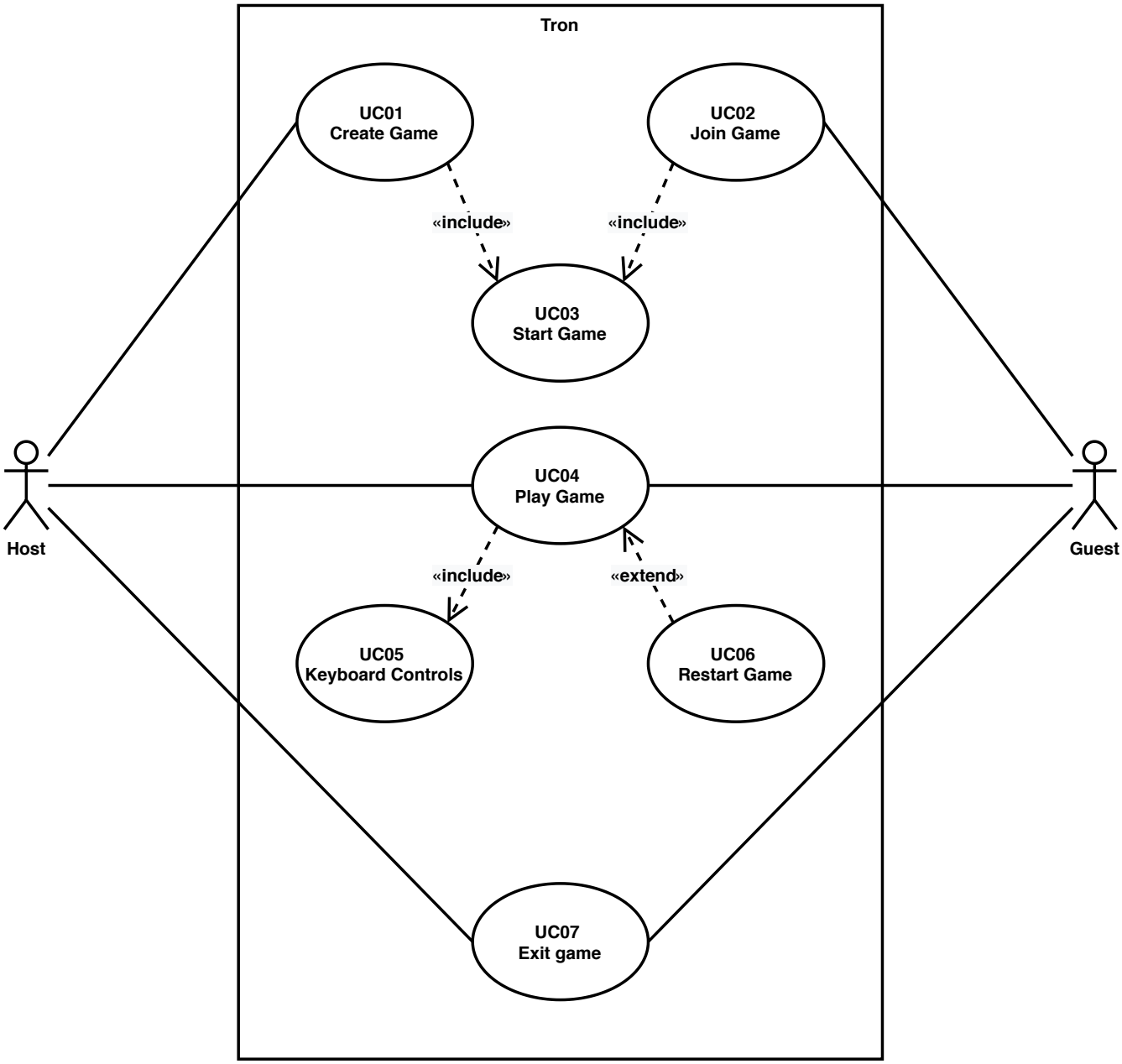
Organisational constraints

ID	Constraint	Description

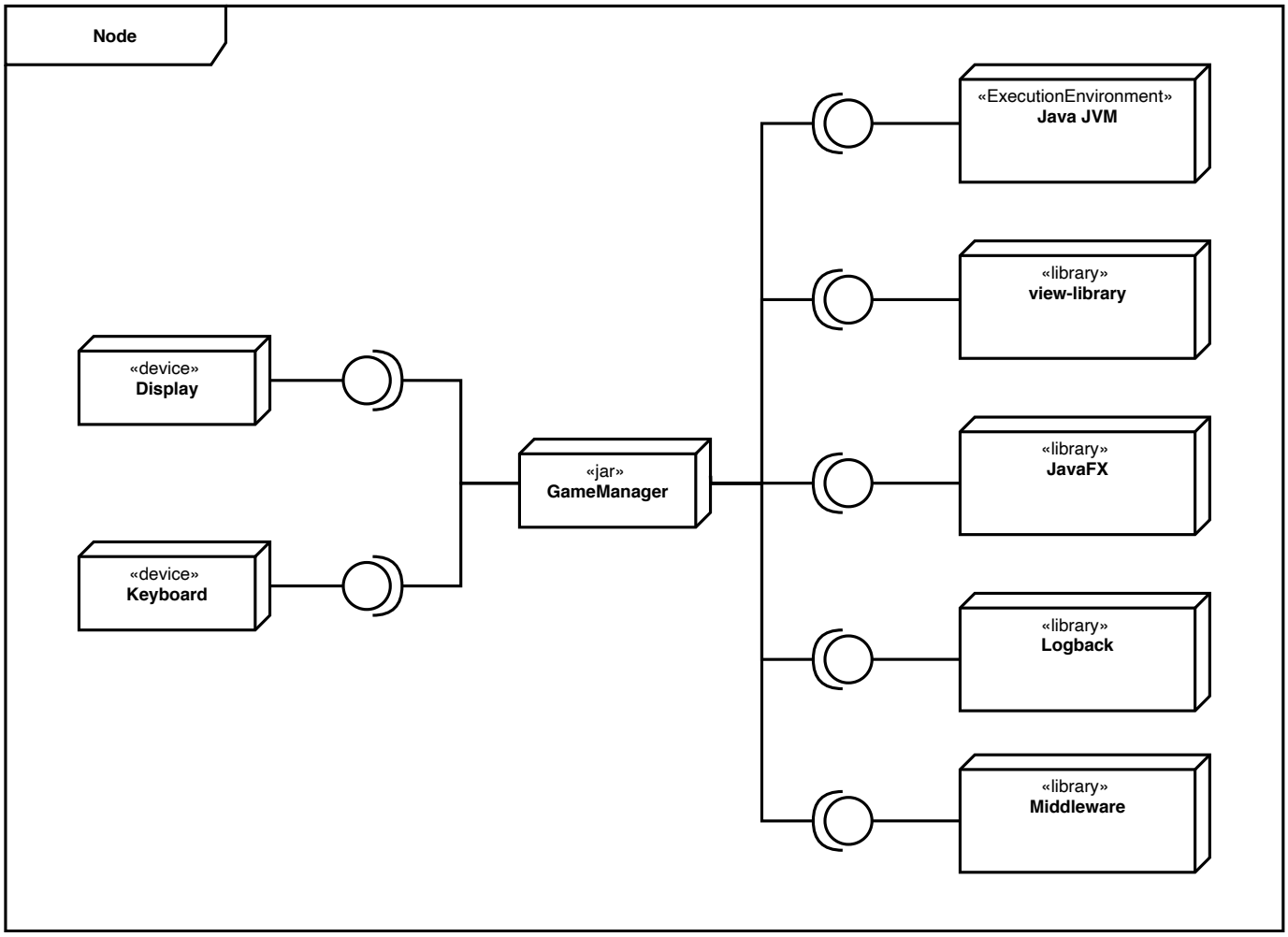
ID	Constraint	Description
OC01	Documentation	clear representation of the structure in at least 2 hierarchy levels: component diagram, class diagram, deployment diagram (ARC42)
OC02	Project Management	fixed method for project management (proof)
OC03	Problem Solving	problem-solving strategies must be derived from reference literature or accepted third-party literature
OC04	Deadline	project musst be delivered by 27.01.2022 23:59 UTC

System Scope and Context

Business Context



Technical Context



Solution Strategy

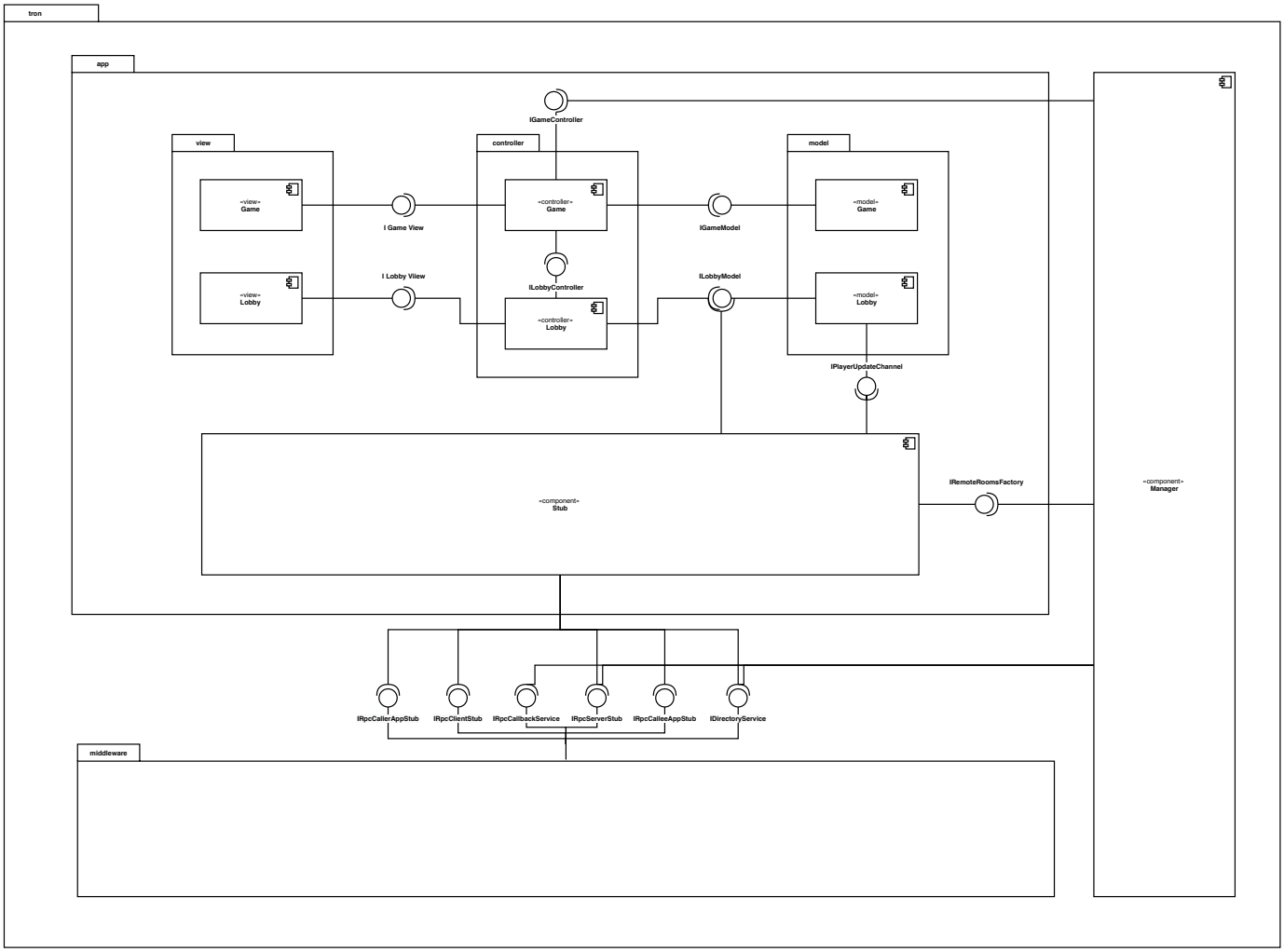
Actor	Function	UCID	Semantics	Precondition	Postcondition
Game Controller	void showStartMenu(String message)	UC01	forward the call to the view component with the needed handlers to show the start menu	user clicked new game button in the game manager	the view component has the needed handlers to generate the needed UI components and the player can change his name choose to create or join a game
Game Controller	• void createGame() • void joinGame()	• UC01 • UC02	setup the game environment with the appropriate handlers to start the game	player clicked create/join game	player has to wait for someone to join or choose a room to join

Actor	Function	UCID	Semantics	Precondition	Postcondition
Lobby Controller	void createRoom(IPlayerUpdateChannel updateChannel)	UC01	create a room from the player update channel that was provided by the game controller	player clicked create game	a room was created with the corresponding update channel that can be exchanged with the joiner (guest). The room is also registered by the lobby model and listed for others to join
Game Controller	void startGame(String opponentName)	UC03	removes the room, resets the view and start the game updater	the guest double clicked a room to join	game updater is listening for updates and players can play the game
Game Controller	void updateGame()	UC04	updates the game state based on player input while insuring fairness against the opponent and insuring both have the same opportunity to react to updates	game is running in an infinite loop	player inputs are reflected in the game
Game View	void showGame(IGame game)	UC04	present the current state of the game	game is running in an infinite loop	player inputs are presented in the UI
Game Model	void steer(Direction newDirection)	UC05	checks if player bike direction input is allowed based on current direction and steer it	player pressed one of the defined keyboard keys to move his bike	movement may be applied or ignored
Game Controller	void endGame(String message)	UC06	ends the current game and calls void showStartMenu(String message)	player won/lost the current game	player can start a new game or join one

Actor	Function	UCID	Semantics	Precondition	Postcondition
Game Controller	boolean fairPlayInsured()	UC07	on each tick check if the received update version match with the current player update and if not or no update was received call void endGame(String message) to end the game	the game loop is runniing	the player won the game and can start a new game or join one

Building Block View

Overall System White Box



Tron

Component	Description
Manager	the starting point of the system. Handles configuring and starting games

App

X referees to the component name in the model/view/controller packages (for example X Model correspond to Game Model)

Component	Description
X Modle	The central component of the MVC pattern. directly manages the data, logic and rules of the application.
X View	represent the information received from the model through the controller
X Controller	Accepts input and converts it to commands for the model or view
Stub	the fusing layer between the middleware and the application

Model Black Box

Interface	Description
IXModel	handles the data and state including the logic

View Black Box

Interface	Description
IXView	handles the representation and generate the needed UI components

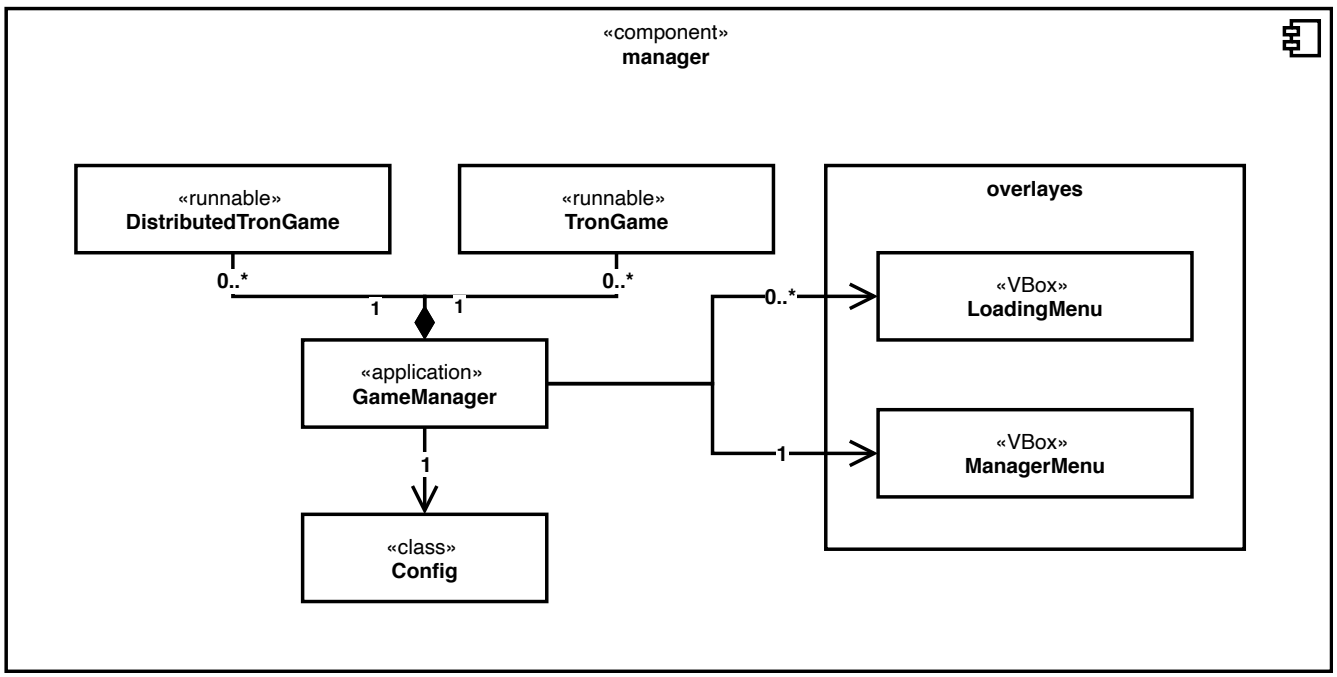
Controller Black Box

Interface	Description
IXController	enables the interconnection between the view and model so it acts as an intermediary.

Stub Black Box

Interface	Description
IRemoteRoomsFactory	maps the local rooms to remote ones and vise verse

Manager



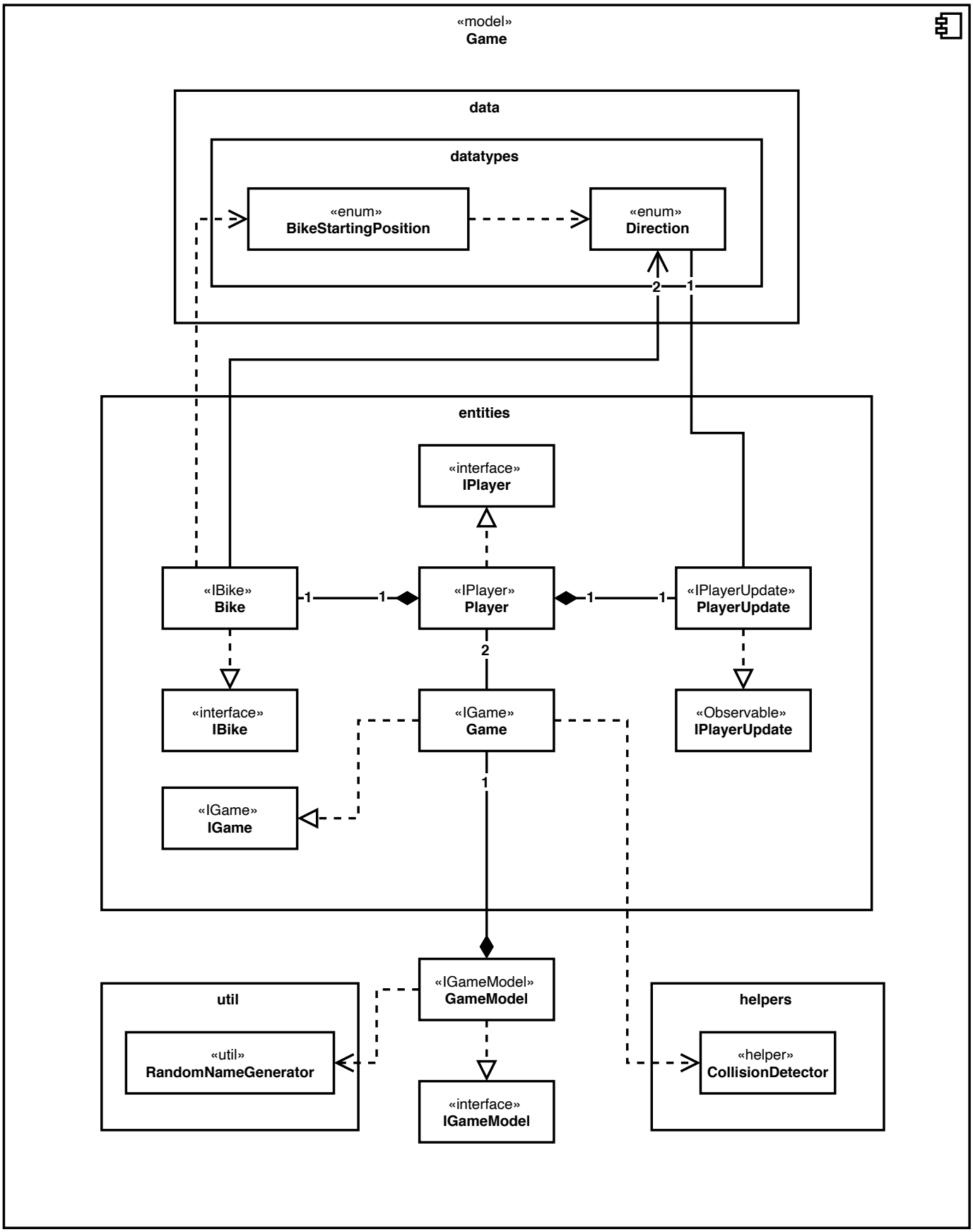
Middleware

see Middleware

Level 2

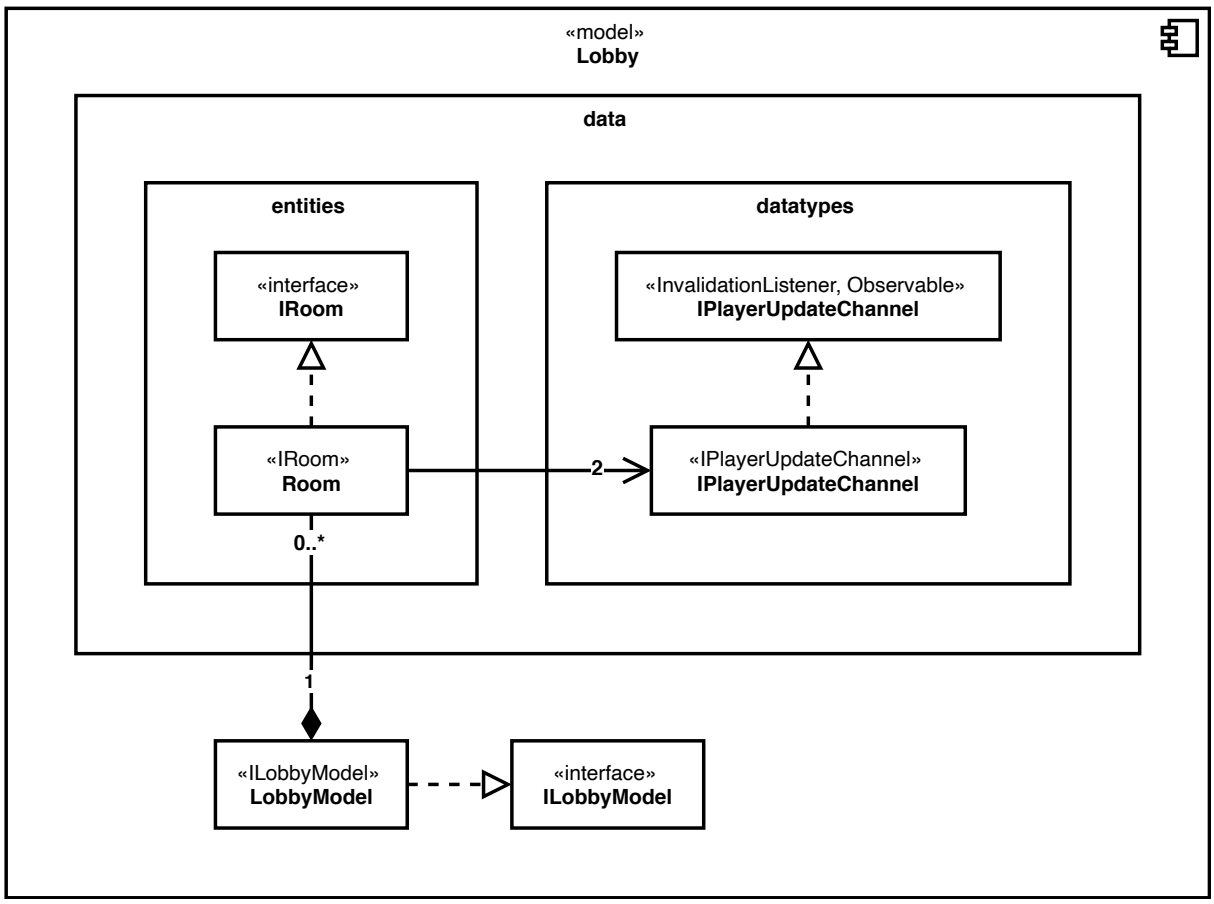
Model White Box

Game



Interface	Description
IBike	The Bike the Player is riding on, responsible for the Trail and Movement
IPlayer	The representation of a Player in our Game, receives and creates PlayerUpdates
IPlayerUpdate	An Update to an existing Player, could be Alive State or Movement
IGame	The representation of the Game Board which contains the Players and outcome of a match
IGameModel	The abstraction of the Game Board, creates and Consumes Player and Game State Updates

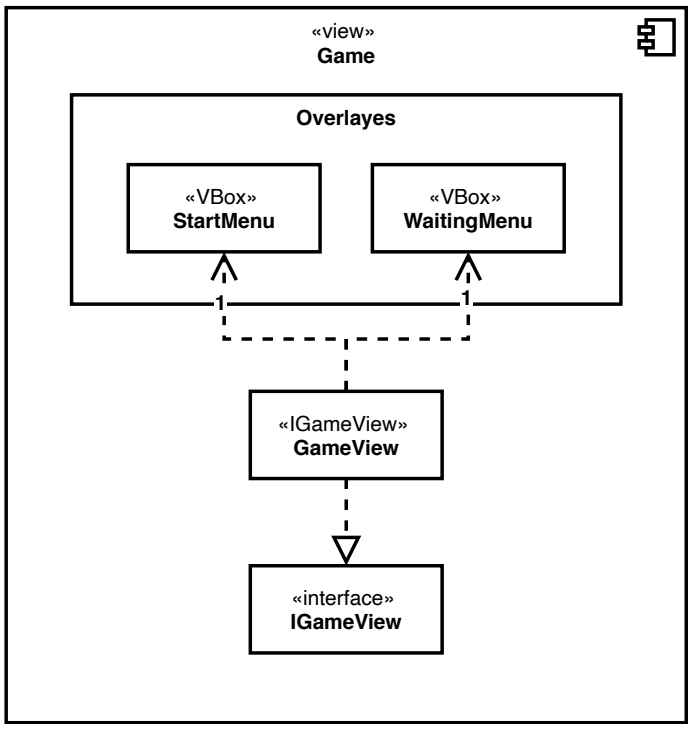
Lobby



Interface	Description
IRoom	The representation of a Room the Player can join to start a Game
ILobbyModel	The representation of a Lobby, containing multiple Rooms
IPlayerUpdateChannel	A channel for the two Player Updates in a Room

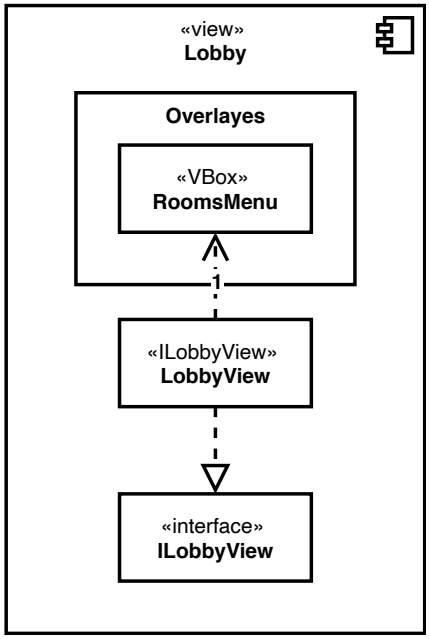
View

Game



Interface	Description
IGameView	The View of a Game in Progress, holds Information about the Menus a Player can show

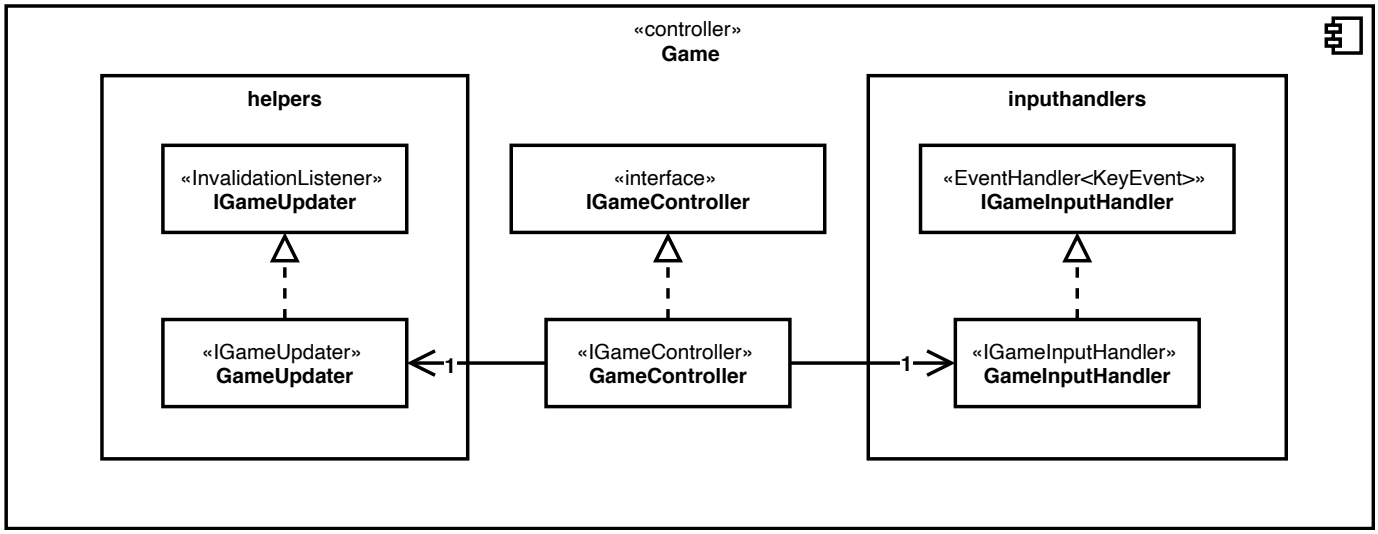
Lobby



Interface	Description
ILobbyView	The View of a Lobby, holds Information about the Menu a Player can show

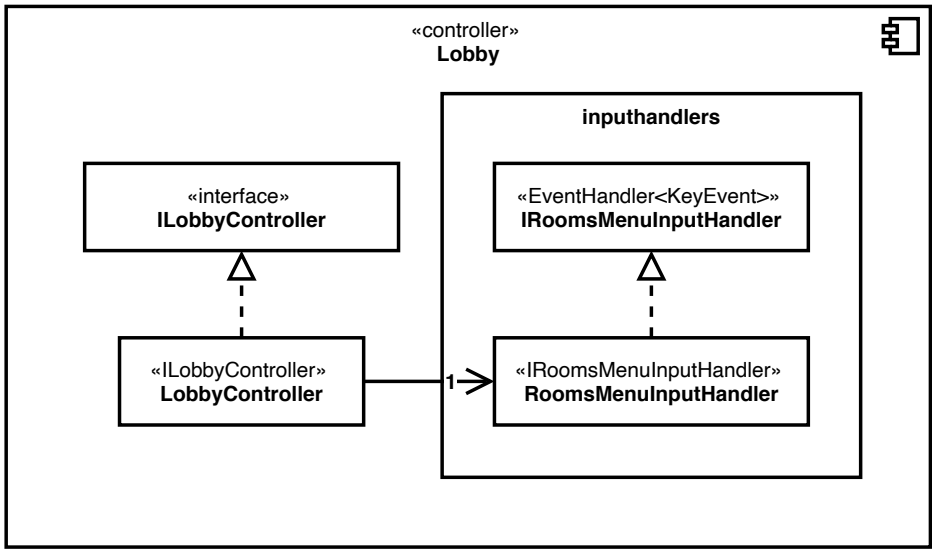
Controller

Game



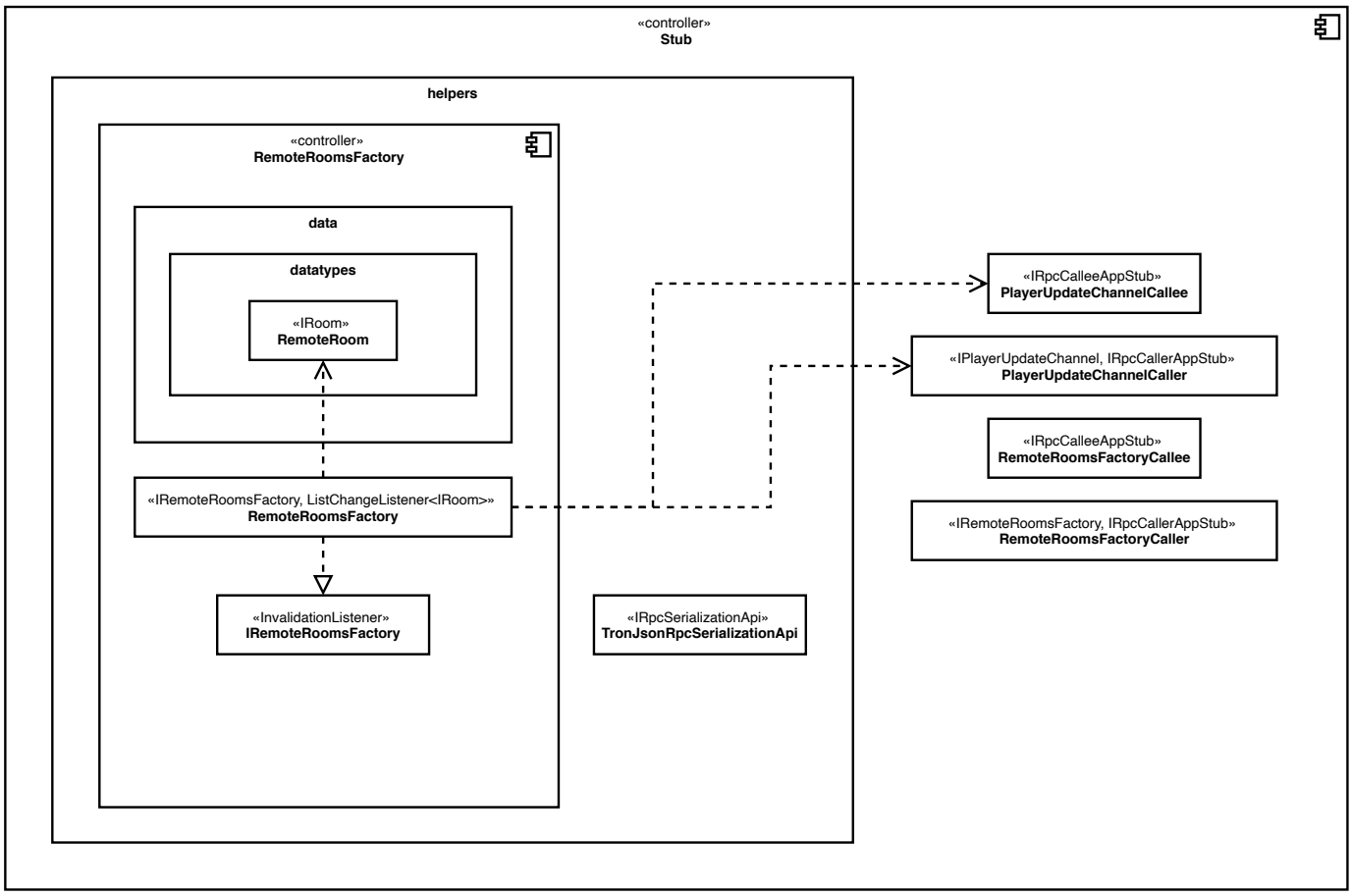
Interface	Description
IGameUpdater	Handles the Game Updates from the Game Model
IGameController	Holds all relevant Information of a Game, the Ability to create, join or quit a Game, connects all other Game Components
IGameInputHandler	Reacts to KeyEvents and handles those changes for the Controller

Lobby



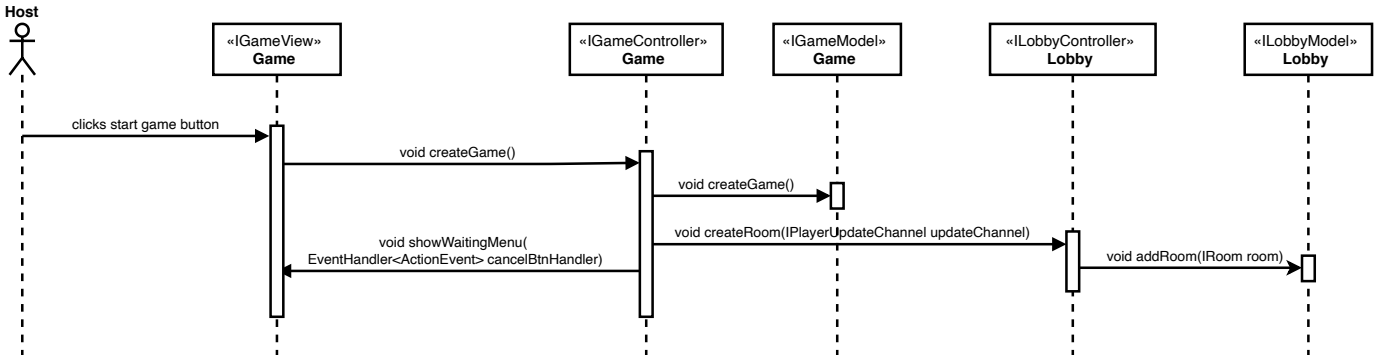
Interface	Description
ILobbyController	Holds all relevant Information of a Lobby, the Ability to create or join Rooms
IRoomsMenuInputHandler	Handles the KeyEvents in a Room such as the Menu

Stub White Box

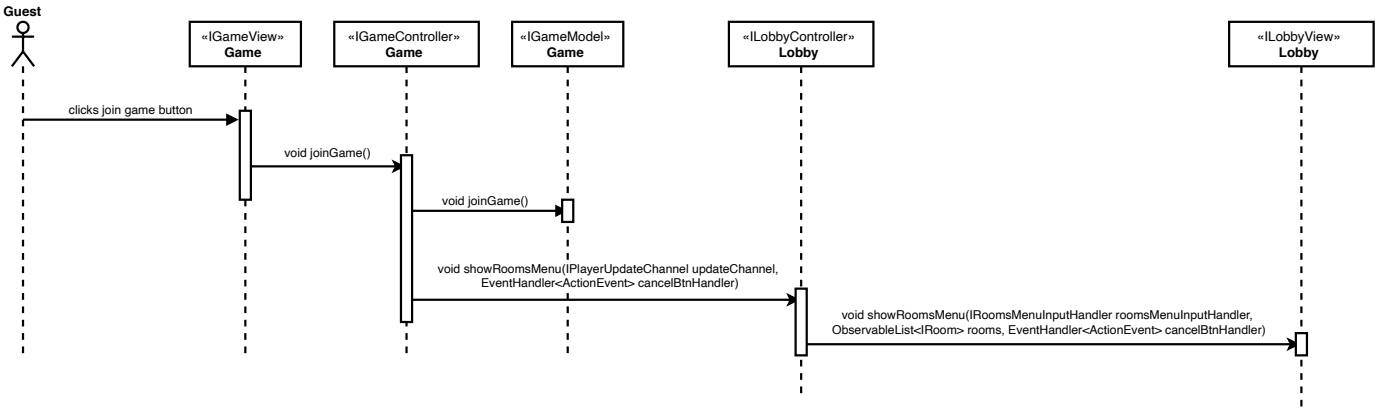


Runtime View

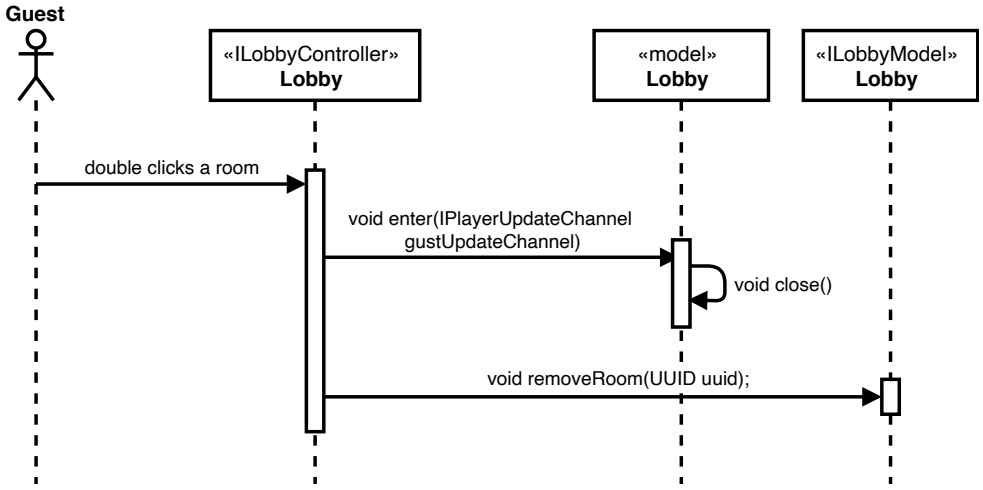
UC01: Create Game



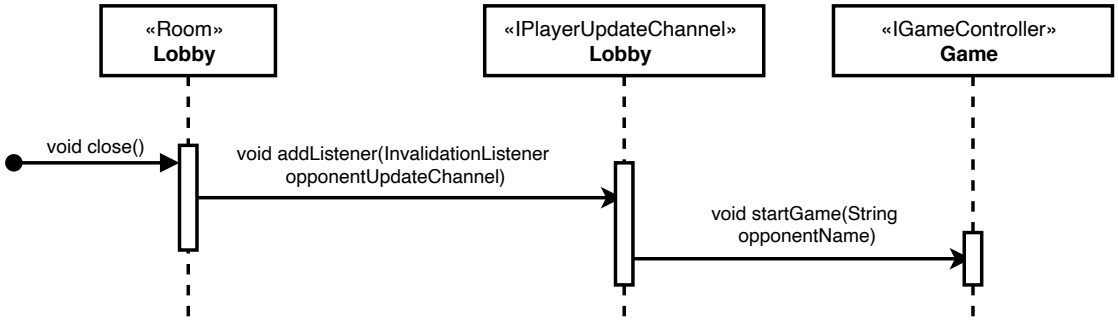
UC02: Join Game



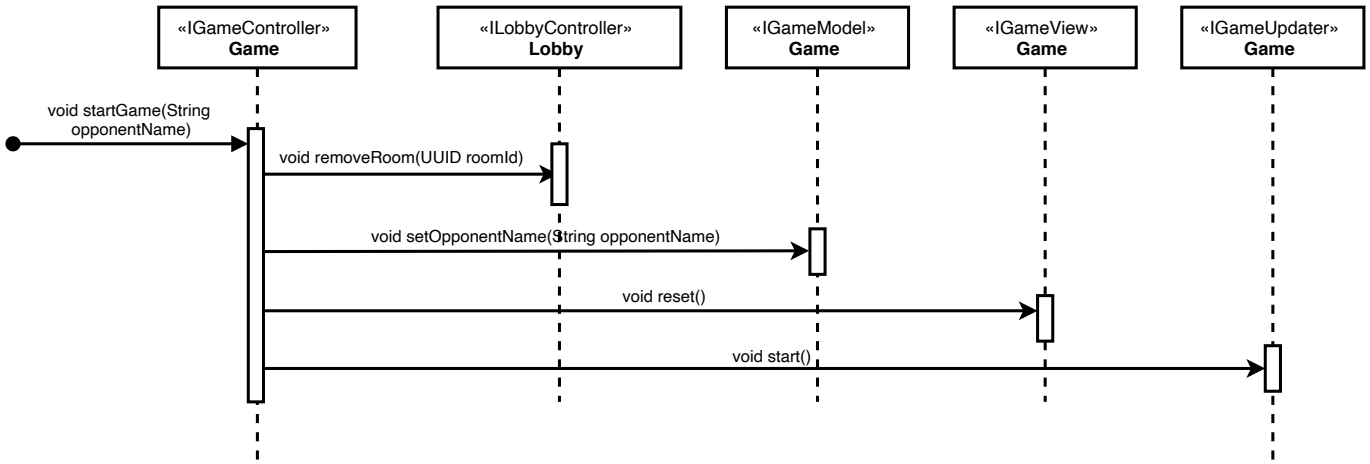
Join Room



Room is Full

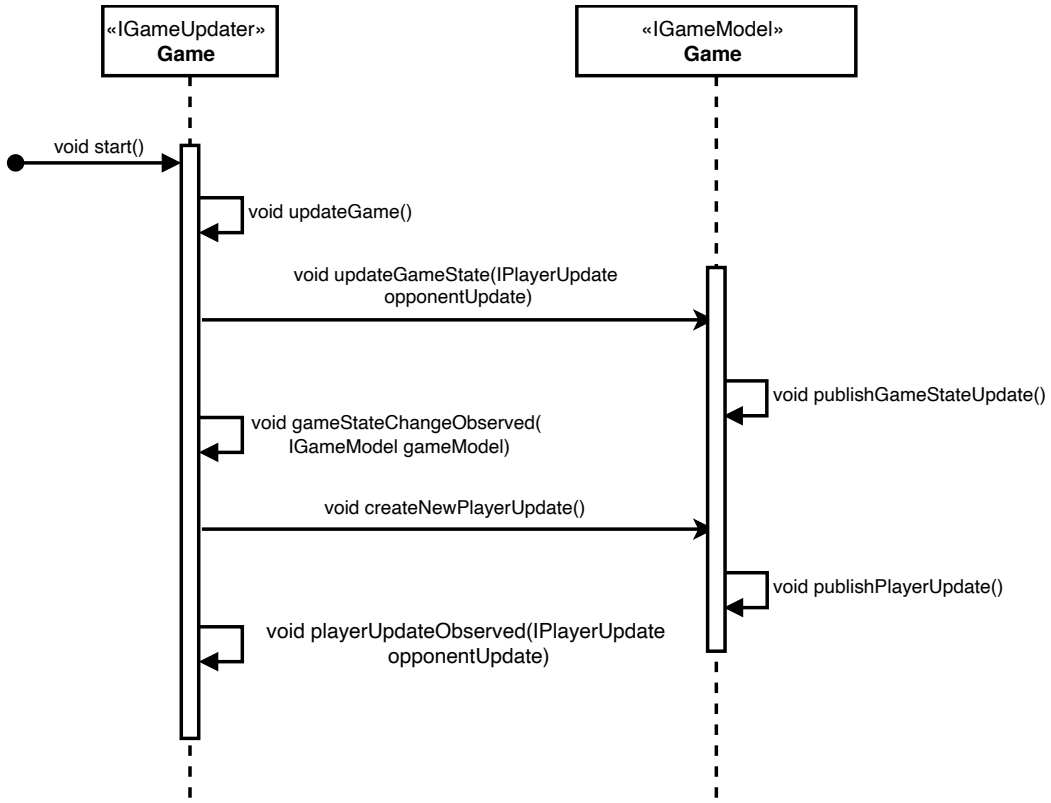


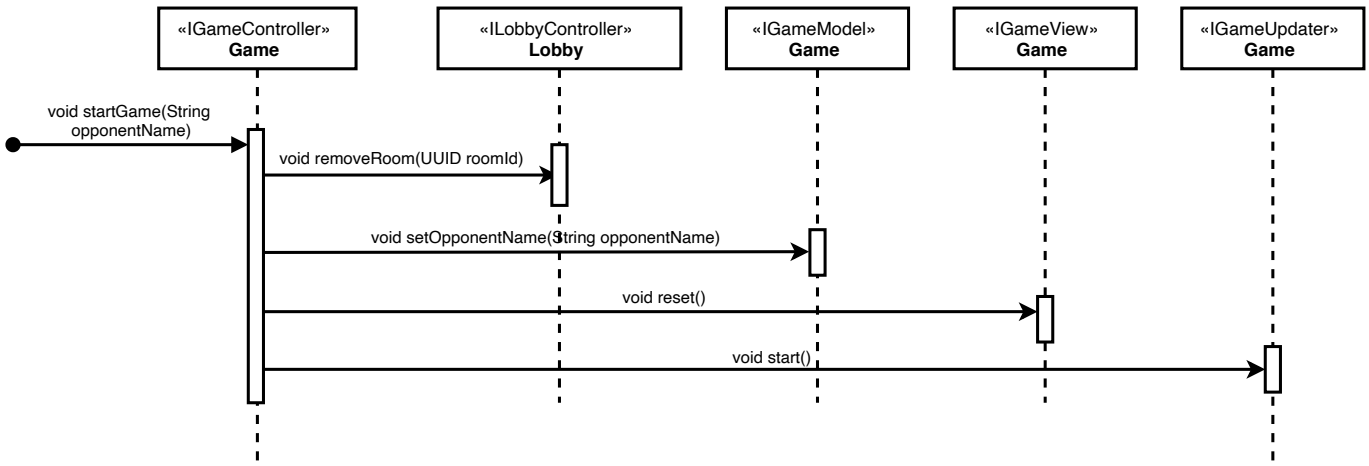
UC03: Start Game



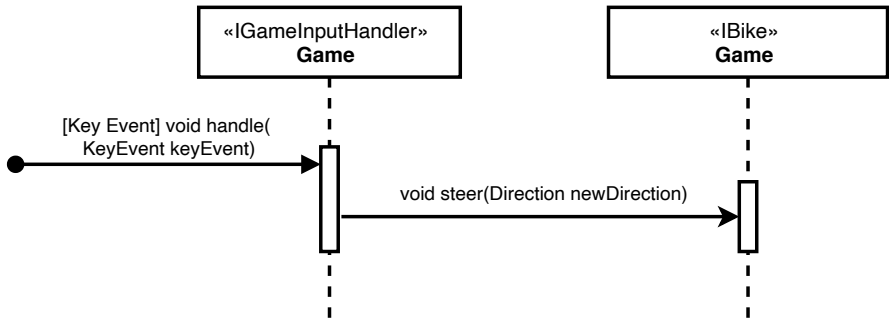
UC04: Play Game

Player Update Observed

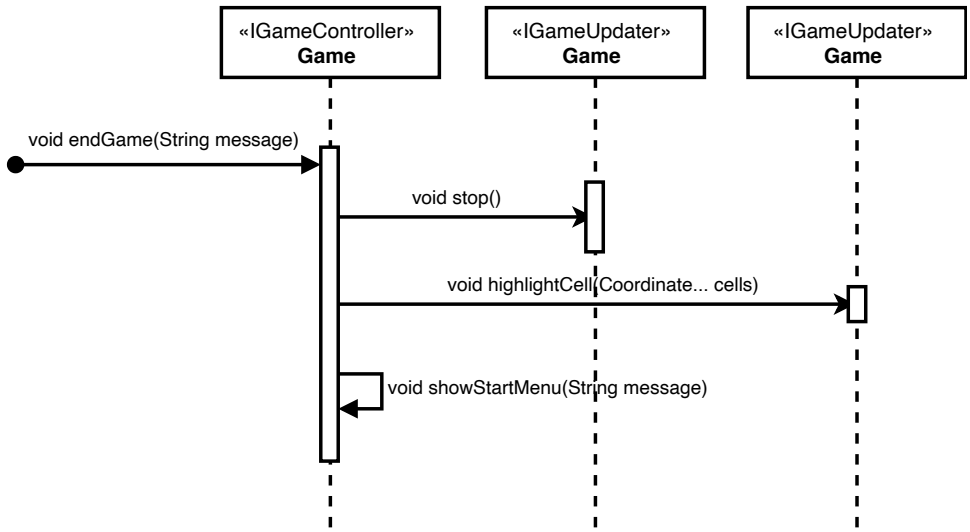




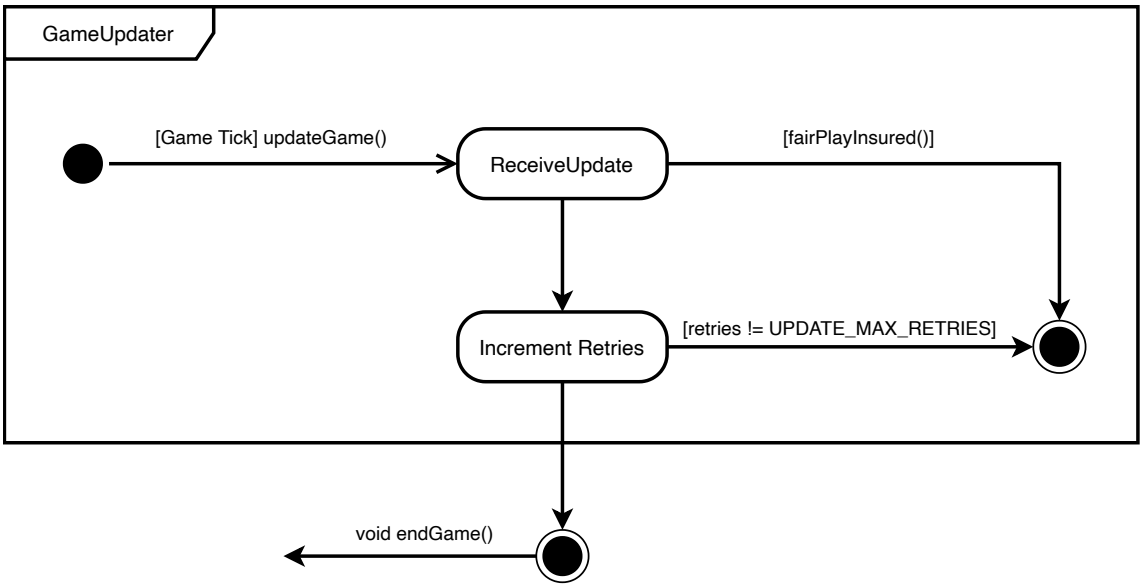
UC05: keyboard controls



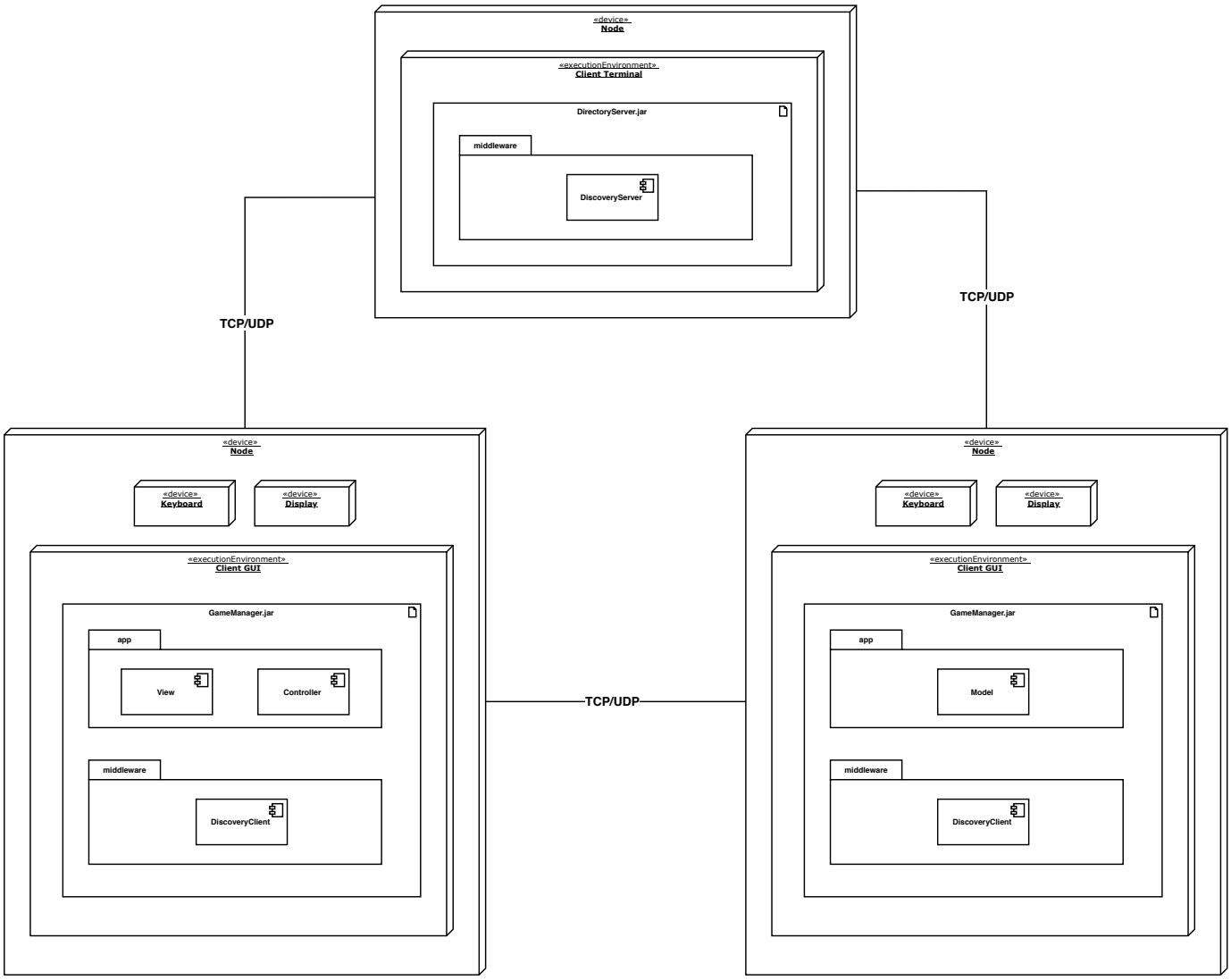
UC06: Restart Game



UC07: Exit Game



Deployment View



Cross-cutting Concepts

Technical Decisions

TD01: limiting the number of Threads

TC11 requires the system to not exceed 80% of resource usage. This is easily achievable by parametrizing the JVM.

However and to ensure fairness between the different instances of the game java ExecutorService is used throughout the system. this insures a limiting factor on the number of threads created.

the number of threads per instance can be callculated as follows:

GameUpdater -> number of available processors
RpcServerStub -> number of available processors + 1 for the server thread

in the case of 8 processors each instance has a limit of 17 controlled threads

TD02: allowing for a small update cache

the GameUpdater observes multiple invalidated states coming from different threads this means, while highly unlikely but in case of, for example, two blocked threads that delivers the current and next PlayerUpdate simultaneously ConcurrentHashMap is used to store the received updates and make them available for the next game updating iteration.

OUV: the version of the observed PlayerUpdate with the same version as the currently processed PlayerUpdate

the cache can never grow outside two elements as per design current update version + 1 is only created when an update with the same version is observed (OUV) and therefore the maximum case of updates observed occurs when the game updater observes a OUV + 1 while still awaiting the OUV update

TD03: game state lock

in case updating the state took longer than the tick waiting period, the next update call is blocked

TD04: ensure fair play

to ensure fair play as described in QG02 it is ensured that both players are observing the same state (matching PlayerUpdate versions).

It is however possible that the opponent disconnects by for example closing the game therefore a "Fairness limit" is set by waiting UPDATE_MAX_RETRIES / FRAMES_PER_SECOND seconds before endinig the game in favour of the player

TD05: UI lock

the UILock is needed to

1. ensure that the player had the chance to observe the new state and act accordingly (even if it's not really possible but at least he can estimate)
2. ensure that the game state doesn't change while the GUI is still updating

TD06: direction locking

this allows the player to spam the input keys without really effecting the end result.

This also helps the player to make a "last minute decision" when for example pressing the wrong key (weather it's possible or not depends on multiple factories one of them is how fast the player fingers are 😃)

TD07: game instance coupled to player id

a game and all it's component are coupled (when needed) to one player per instance and therefore one random id is generated on the start of the instance (new GameModel -> new Player -> random player id)

TD08: triggering start game event

implementing the state pattern for a Room to for example kick off the start game event onFull() was considered and founded to be overkill as the a game can only handle two players TC05 and the Room is carbeged after the UpdateChannels were exchanged (by closing the room). this exchange will trigger the game to start.

other consideration were:

1. passing the game starter handler to the room itself
    - The GameController has no reference of the room and therefore can't pass the correct handler reference to start the game
2. triggering a start game event when an update is received
    - the requires the game updater to start when it's not really necessary and will result in a conditional check on each update that only needed for the first one.

TD09: config

it is recognized that this is not the best way/practise to do it but in this case it should be more than enough.

TD10: singleton game model

in the standalone version the model is shared as a the source of truth for rooms, while in the distributed one the directory server is the source of truth

Design Decisions

Design Decisions are referenced from the referenced literature

DD01 local game model

> reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. Page 21

All computations are done locally on the node, and the end-result is insured based on the PlayerUpdate and it's version, which will always lead to same result unless it was tampered with but security, however, is not a requirement.

DD02 remote rooms factory

> An important goal of distributed systems is to separate applications from underlying platforms by providing a middleware layer. Page 55

to keep the game as a black box and never make it depends on the the underlying platform the RemoteRoomsFactory was implemented. this way The game can be played locally or in a distributed fashion.

DD03 peer to peer

> In horizontal distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. Page 81

having a local game model (DD01) make a "game server" obsolete, which renders it as a bottleneck that simply transfer game updates between nods.

For the shared data (rooms information) a directory server is to be spawn up anyway. Therefore, to avoid the single point of failure the game is implemented in a peer to peer fashion.

Introduction and Goals

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. The Middleware is the same to a distributed system as what an operating system is to a computer.

The important goals that should be met to make building a distributed system worth the effort are:

A distributed system:

- should make resources easily accessible.
- should hide the fact that resources are distributed across a network.
- should be open
- should be scalable.

Requirements Overview

The middleware assist the development of distributed applications and it is considered as a manager of resources offering its ap- plications to efficiently share and deploy those resources across a network.

Next to resource management, it offers services that can also be found in most operating systems. The main difference, is that middleware services are offered in a networked environment.

ID	Use-Case	Description
UC01	offer services	as an application I want to specify and offer my services
UC02	lookup services	as an application I want to lookup and use the offered services
UC03	invoke functions	as an application i want to invoke a function that is implemented and executed on a remote computer as if it was locally available.

Quality Goals

ID	Goal	Description
QG01	Access	Hide differences in data representation and how an object is accessed
QG02	Location	Hide where an object is located
QG03	Relocation	Hide that an object may be moved to another location while in use
QG04	Migration	Hide that an object may move to another location
QG05	Replication	Hide that an object is replicated
QG06	Concurrency	Hide that an object may be shared by several independent users
QG07	Failure	Hide the failure and recovery of an object

Stakeholders

Role	Expectations

Role	Expectations
Customer	• fixed method for project management (proof) • fixed method for documentation (important: systematic and faithful to the method) • Protocol definition with error semantics • clear representation of the structure in at least 2 hierarchy levels: component diagram, class diagram, deployment diagram • clear representation of the behavior through sequence diagram, activity diagram, state diagram • problem-solving strategies must be derived from reference literature or accepted third-party literature • code must match the documentation and documentation must match the code • Implementation in an object-oriented language • musst use Dependency-inversion-principle • The use of frameworks must be approved by the customer

Application	• should be easy to integrate
	• should have comprehensive Dokumentation to work with
	• Adhear to the design principles and avoid common pitfails like:
	  ◦ The network is reliable
	  ◦ The network is secure
	  ◦ The network is homogeneous
	  ◦ The topology does not change
	  ◦ Latency is zero
	  ◦ Bandwidth is infinite
	  ◦ Transport cost is zero
	  ◦ There is one administrator

Developer	• should be maintainable • should be expandable

Architecture Constraints

Technical Constraints

ID	Constraint	Description
TC01	Programing language	Implementation in an object-oriented language
TC02	Implementation	• musst use Dependency-inversion-principle • should be easy to integrate • should be maintainable • should be expandable
TC04	Frameworks	The use of frameworks must be approved by the customer

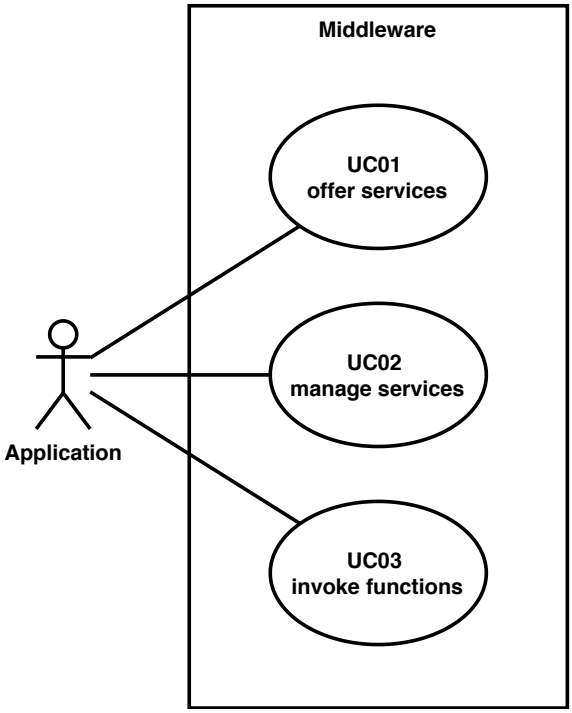
Organisational constraints

ID	Constraint	Description

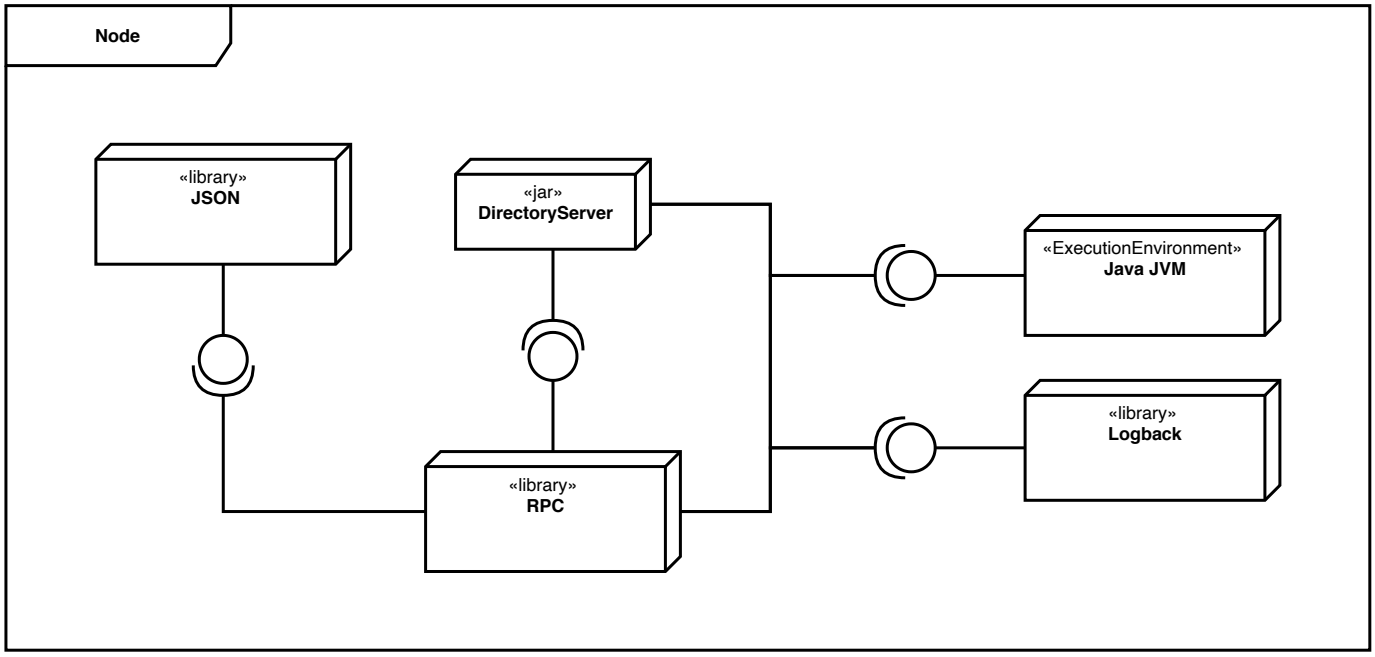
ID	Constraint	Description
OC01	Documentation	clear representation of the structure in at least 2 hierarchy levels: component diagram, class diagram, deployment diagram (ARC42)
OC02	Project Management	fixed method for project management (proof)
OC03	Problem Solving	problem-solving strategies must be derived from reference literature or accepted third-party literature
OC04	Deadline	project musst be delivered by 27.01.2022 23:59 UTC

System Scope and Context

Business Context



Technical Context



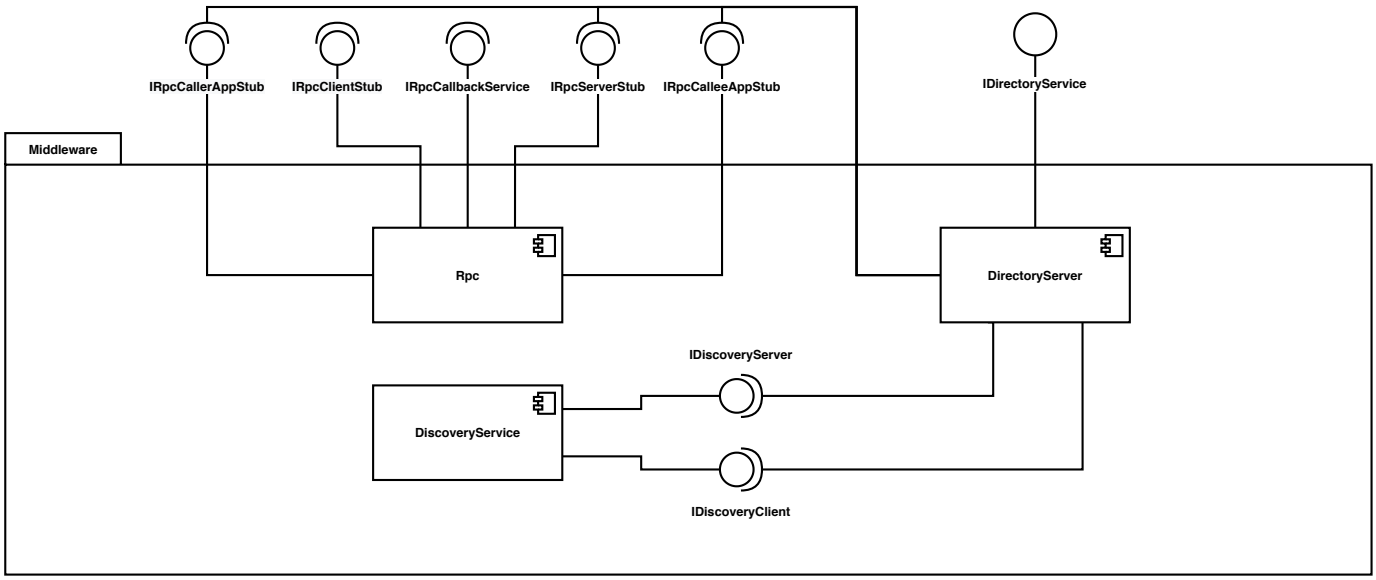
Solution Strategy

Actor	Function	UCID	Semantics	Precondition	Postcondition
ServerStub	void start()	UC01	starts the rpc server which setup a udp and tcp server on the same port and start listening for income sockets		rpc receiver is up and listening on a random port
ServerStub	void receive()	UC01	accept incoming sockets and forward the data to the rpc unmarshaller	rpc receiver was started	the server thread is blocked until socket connections are made
ServerStub	void register(IRpcCalleeAppStub rpcCalleeAppStub)	UC01	register a services to be called if a rpc call was made to the same service id	rpc server stub was initialised	the service is registered and can be called
Discover	static InetSocketAddress discover()	UC01	start listening on the specified discovery address and port		the directory server address was discovered and returned
DirectoryServer	void register(IDirectoryEntry directoryEntry)	UC01	register a service to announce to listeners	the directory server is up and running on a node	the service is registered and will be announced to current listeners

Actor	Function	UCID	Semantics	Precondition	Postcondition
DirectoryServer	void addListenerTo(UUID serviceId, InvalidationListener listener)	UC02	offers a push approach to lookup all services under the same service id		all new services with the same service id will be announced to service listeners
ClientStub	void invoke(UUID serviceId, IRpcCallbackHandler rpcCallbackHandler, boolean isBestEffort, Method method, Object... args)	UC03	forwards the invoked function with application stub preferences to the rpc marshaller	application called a function on a remote service	rpc marshaller received the function call and started the marshalling process
Callback	void register(UUID requestId, IRpcCallbackHandler rpcCallbackHandler)	UC03	registers a callback handler that will be invoked when the result of the request/invocation was received	rpc callback handler was provided which implies that the application is awaiting a callback	the handler is registered and can be used by a callback
Callback	void retrn(UUID requestId, Object result)	UC03	sets the invocation result of the rpc call for the callback handler	a handler exists under the same request id otherwise the result is dropped	the callback handler is invoked with the received result

Building Block View

Overall System White Box



Component	Description

Component	Description
RPC	allow programs to call procedures located on other machines
DirectoryServer	holds and shares all Network relevant information about other participants of the distributed System
DiscoveryService	handles announcing and discovering messages on the local-network

RPC Black Box

Interface	Description
IRpcCallerAppStub	to be implemented when RPC calls are to be made on a service
IRpcClientStub	dose the RPC call on behalf of the application stub
IRpcCallbackHandler	handles the callback result
IRpcServerStub	handles RPC calls and forward them to the application stub of the service to be called
IRpcCalleeAppStub	to be implemented when RPC's are offered from a service

Directory Server Black Box

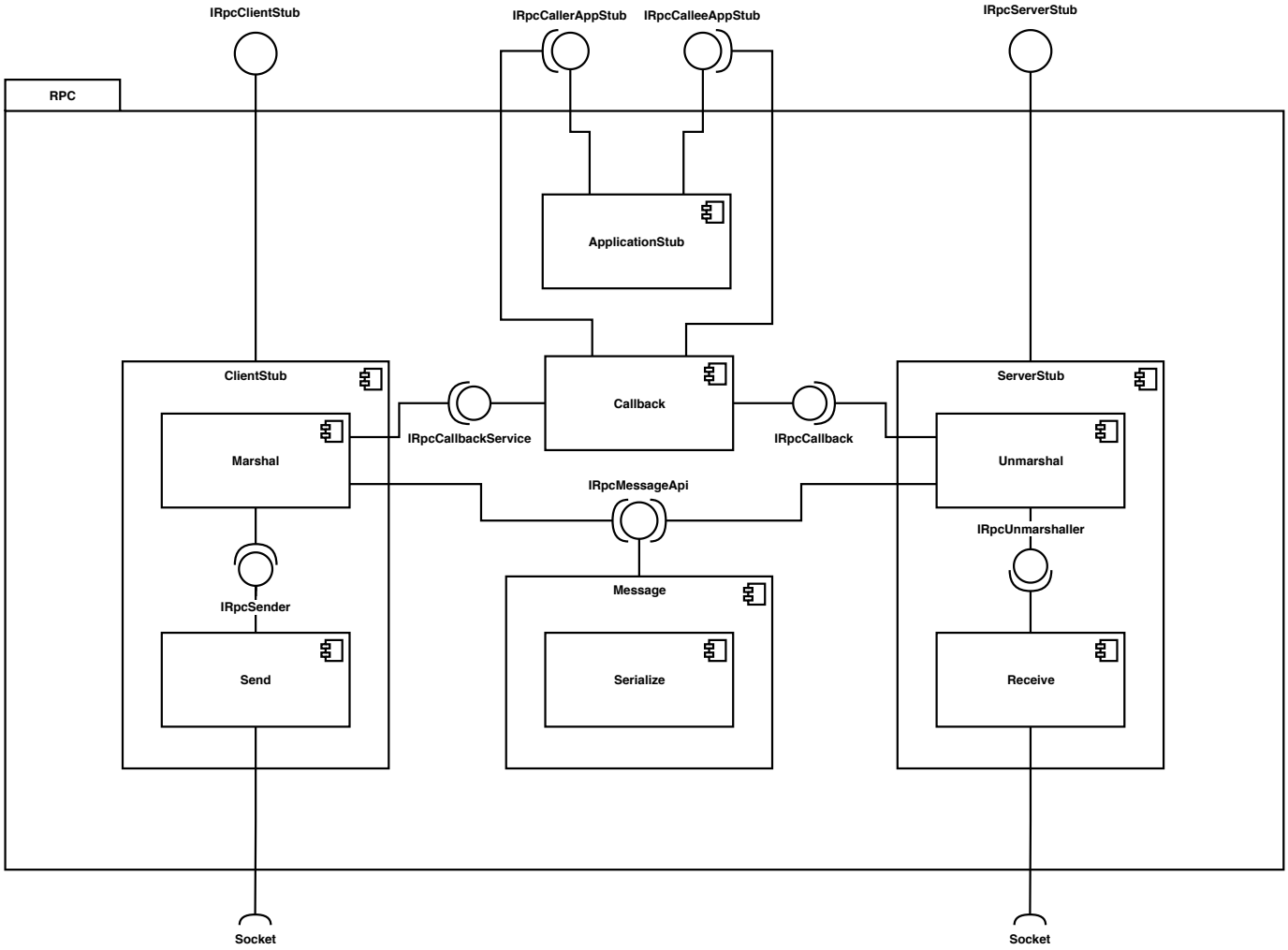
Interface	Description
IDirectoryService	keep track of all offered services and announce them to interested listeners

Discovery Server Black Box

Interface	Description
IDiscoveryServer	broadcast Announcements over the Network
IDiscoveryClient	listening for the discovery Server Announcements and handle them

Level 2

RPC White Box



Component	Description
ApplicationStub	handles the Interfacing with the Middleware
ClientStub	dose RPC calls based on the application stub demand
Marshal	marshals the function to be invoked and it's parameter to be sent over the network
Send	sends the marshaled function over the network
Message	handles meta-information to pack functions and its parameters into RPC Requests
Receive	receives RPC calls and forward them to the unmarshaller
Unmarshal	parses RPC calls and forward them to the server stub to be called
ServerStub	calls the application stub of the received function to be called and do callback if needed
Callback	Handles the Callback of RPC Call invocation result

Client Stub Black Box

Interface	Description
IRpcSender	sends the parsed rpc request over the Network

Message Black Box

Interface	Description

Interface	Description
IRpcMessageApi	handles meta-information to pack functions and its parameters into RPC Requests

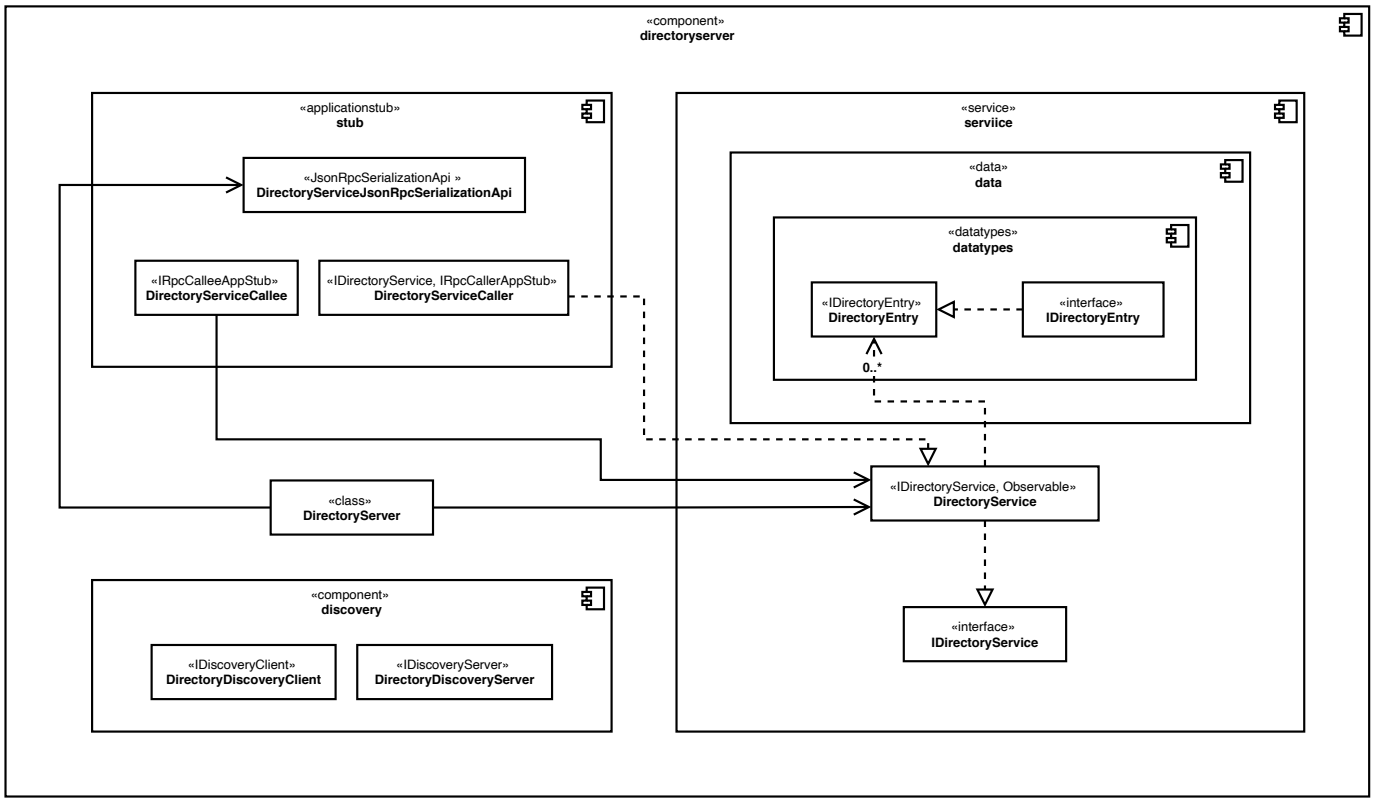
Callback Black Box

Interface	Description
IRpcCallbackService	handles the registration of callback handlers and invoke them when receiving the result
IRpcCallback	returns the result of a RPC Call

Server Stub Black Box

Interface	Description
IRpcUnmarshaller	unmarshals the received RPC request into a RPC Call
IRpcServerStub	Interface for a Server Stub which handles a list of all Methods it can call

Directory Server White Box

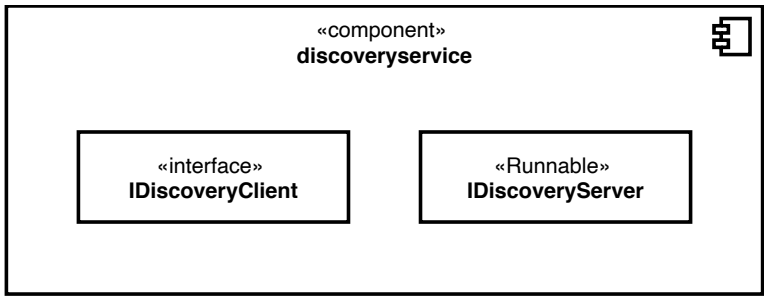


Component	Description
Discovery	handles announcing and discovering the directory server address

Service Black Box

Interface	Description
IDirectoryEntry	a service entry that can be registered and announced

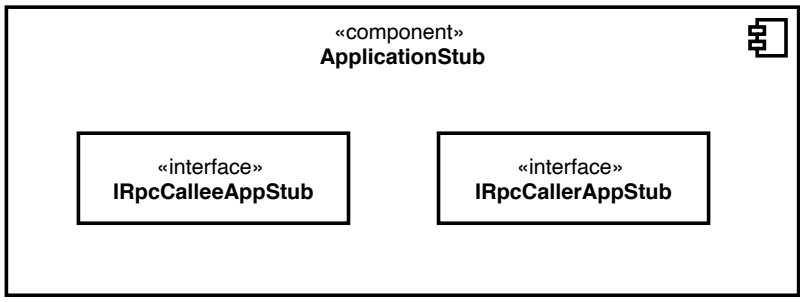
Discovery Service White Box



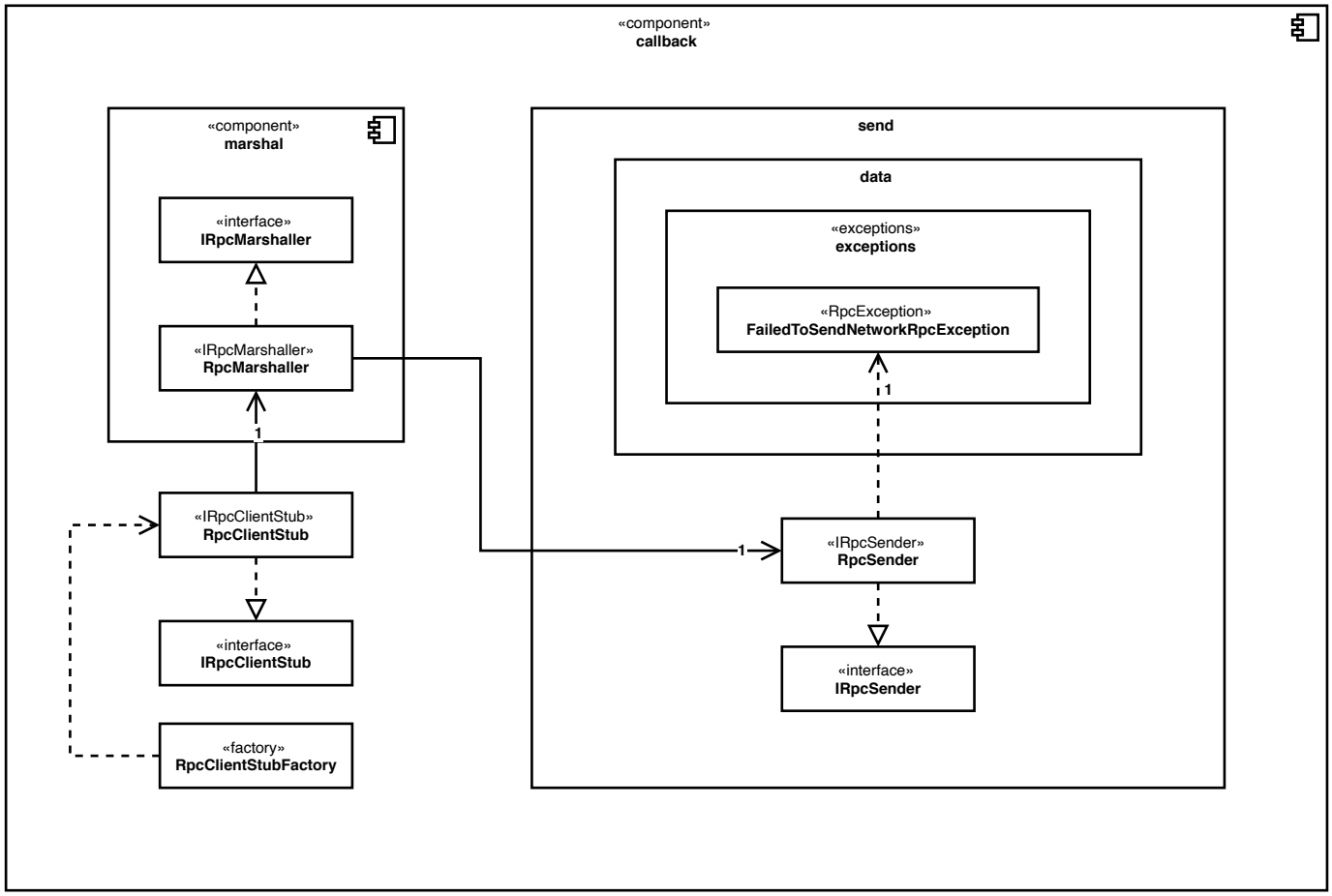
Level 3

RPC White Box

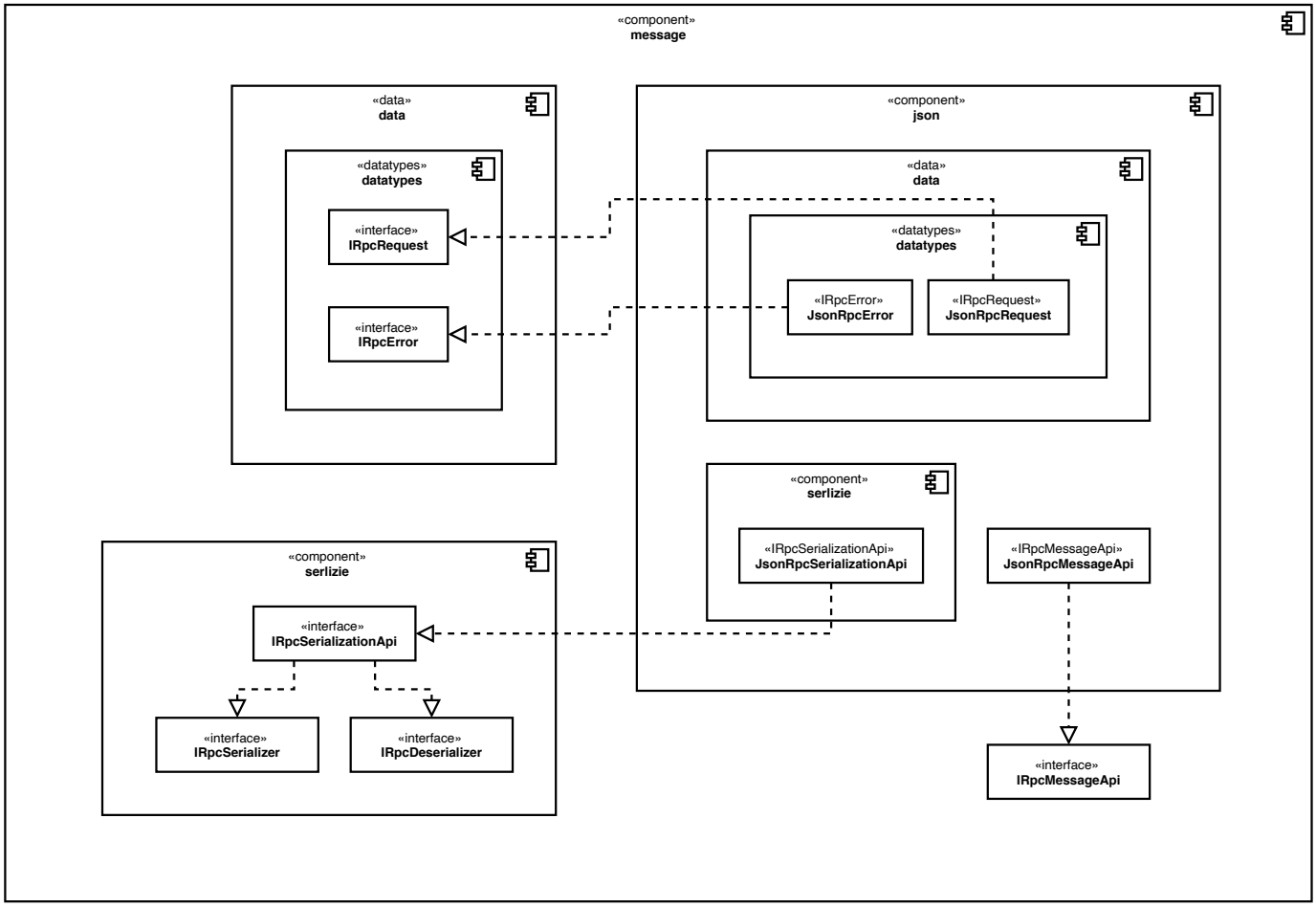
Application Stub White Box



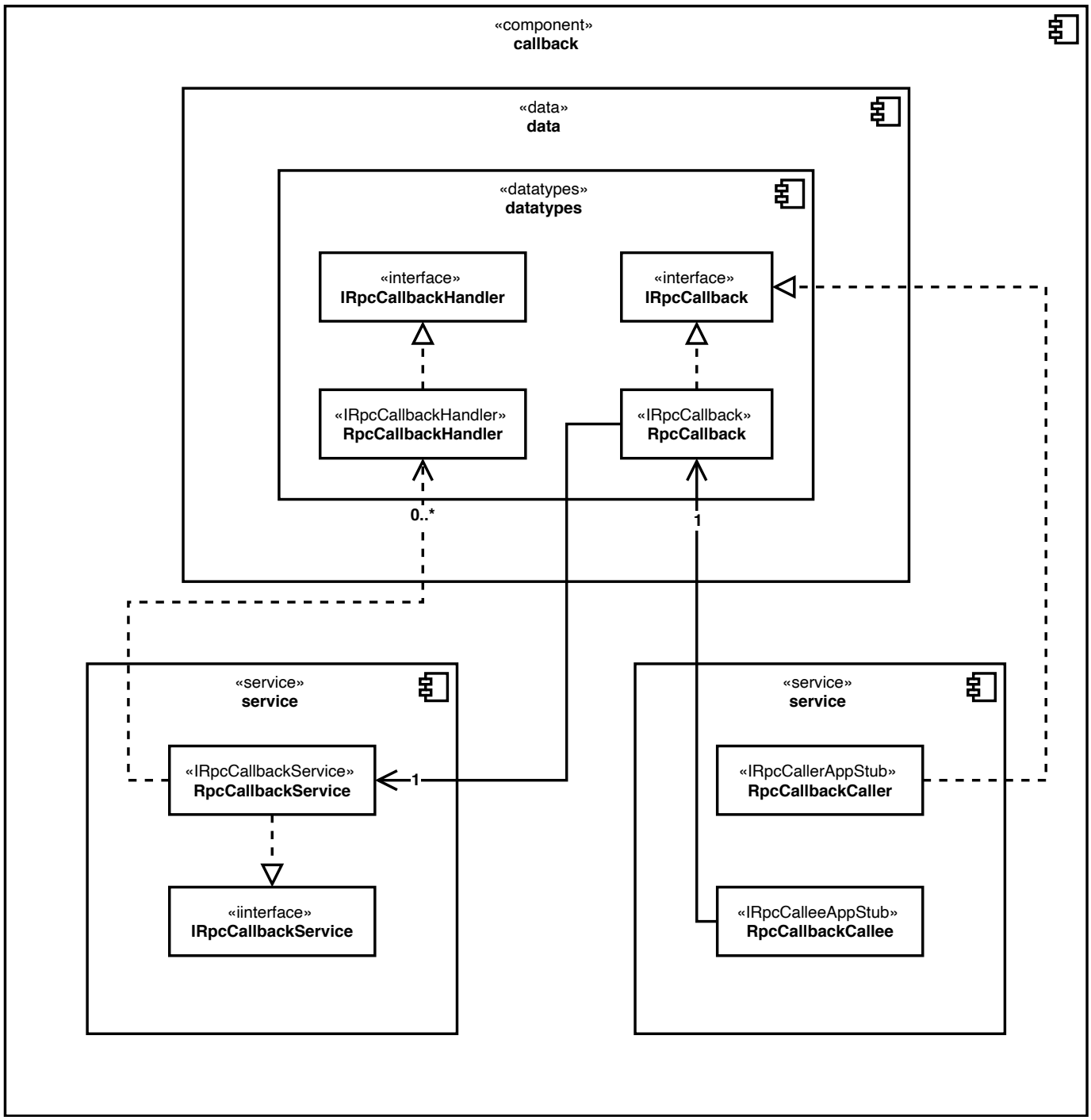
Client Stub White Box



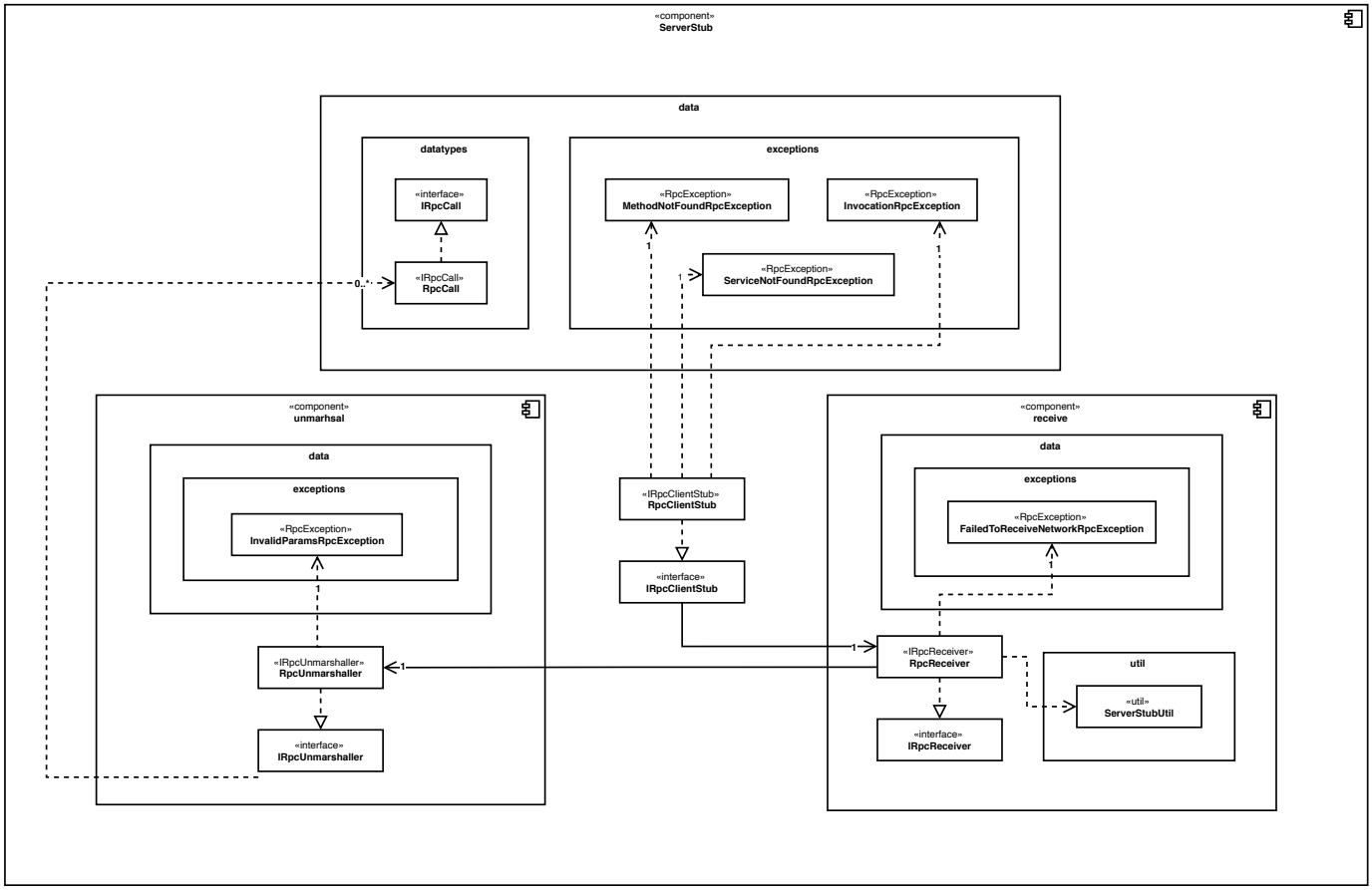
Message White Box



Callback White Box



Server Stub White Box



Runtime View

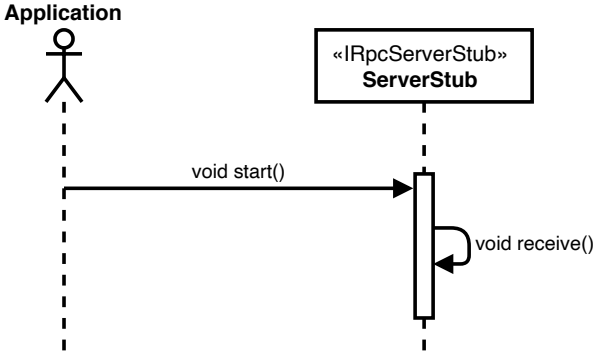
fault tolerance is not a requirement and therefore all sequences describe the best case scenario

Level 1

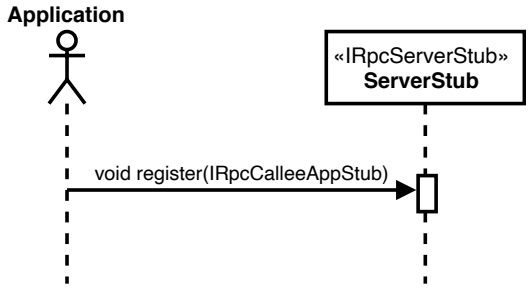
UC01: Offer Services

Starting the rpc server

Application

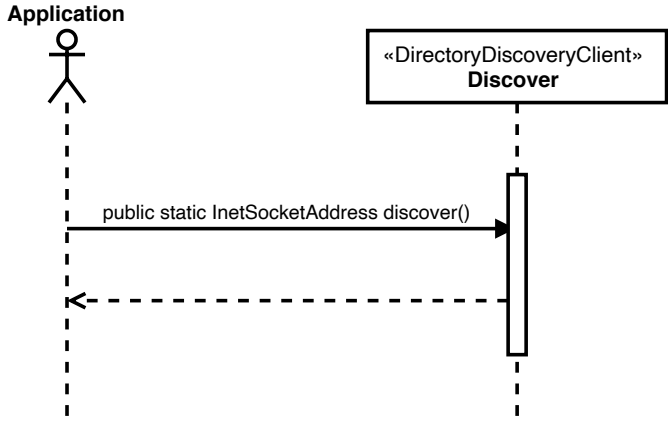


Registering a service

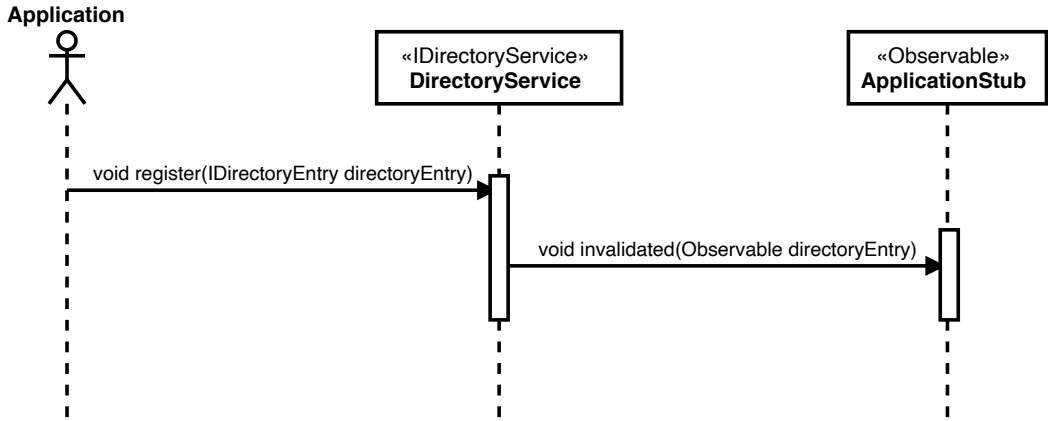


Announce a service

Discover Directory Server

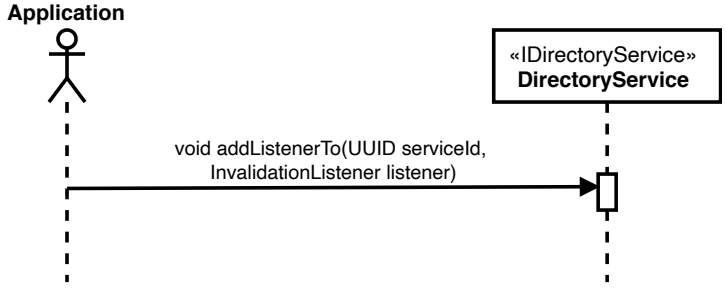


Register a service to be announced

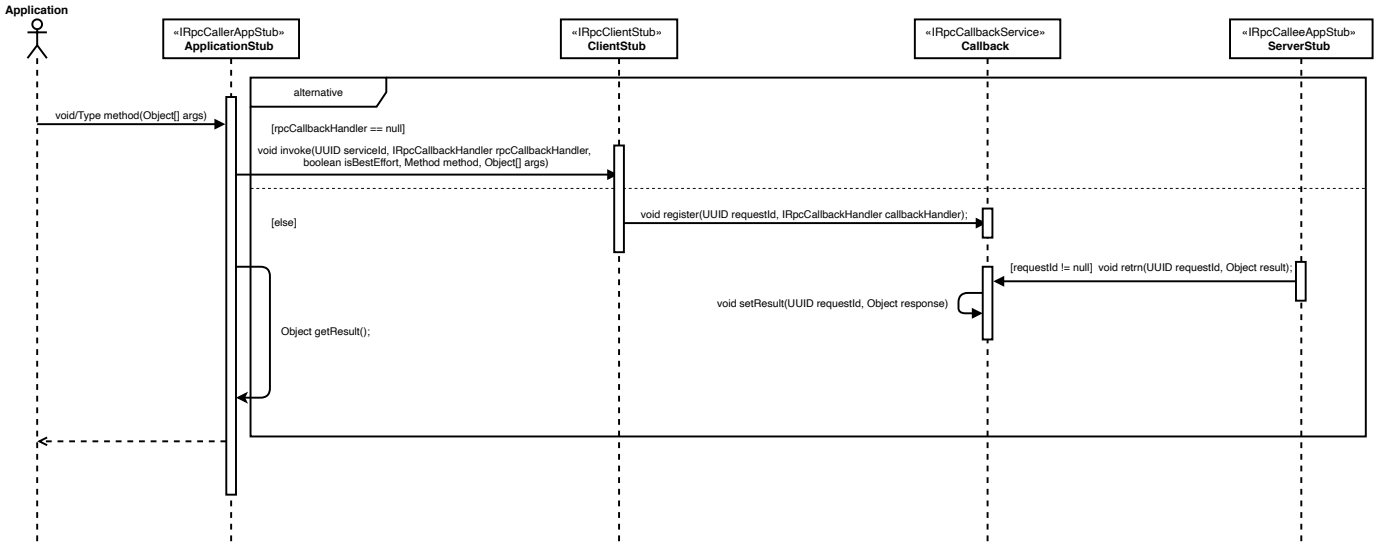


UC02: lookup Services

Subscribe to service group



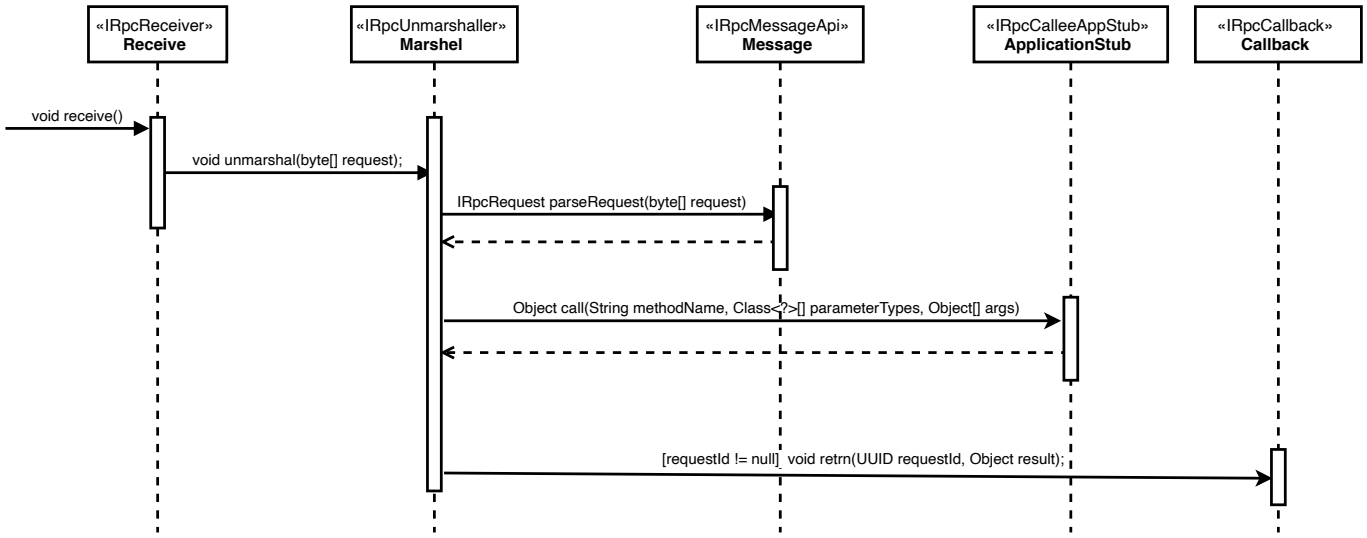
UC03: invoke a function



Level 2

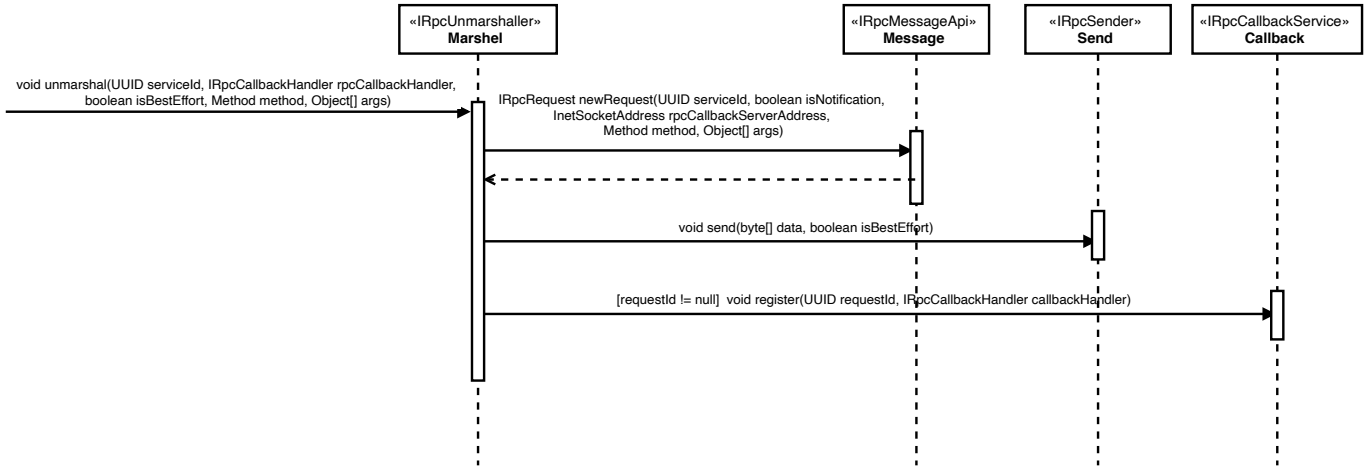
UC01: Offer Services

Receiving calls



UC03: invoke a function

Sending calls



Cross-cutting Concepts

Technical decisions

TD01: observable directory service/entry

in this case the directory service is mainly used as a source of truth for the remote game rooms, therefore implementing it to be observable offer the system a quick and easy way to get the new rooms available without actively pulling or refreshing.

this push approach is also helpful when the state of the room changes weather removed or full which in both cases changes the service reachability and pushes the update immediately to be reflected on all listing nodes

Design Decisions

DD01: Directory server

structured naming or "human-readable" names play no important rule in this system. RemoteRooms are mainly distribute which are grouped under the same service id therefore implementing a very simplified directory server serves the use-cases of this system (the expensive lookup/search and/or the complex mapping of attributes are not a problem for the system use-cases).

> a centralized component is often used to handle initial requests, for example to redirect a client to a replica server, which, in turn, may be part of a peer-to-peer network as is the case in BitTorrent-based systems. Page 102

in this case the directory server serves exactly this purpose and provides the players with the opponent address to initiate the peer to peer "connection"

DD02: Lamport’s logical clocks

> To synchronize logical clocks, Lamport defined a relation called happens- before. The expression a → b is read “event a happens before event b” and means that all processes agree that first event a occurs, then afterward, event b occurs. Page 311

Lamport’s logical clocks fits the use-cases of this system perfectly and were actually indirectly implemented in the game as the application mostly relays on the observable pattern und need to ensure fairness between players a coordination system using the PlayerUpdate version was implemented in the GameUpdater. This implementation can be considered as a basic implementation of the lamport Lamport’s logical clocks

Messaging Protocol

the messaging protocol is highly inspired by this specification of JSON-RPC

Introduction

as a part of the marshaling process The function and its parameters are packed into a message, and sent to the server stub to be unmarshaled again.

this protocol specifies the meta-information agreed on by the marshaller and unmarshaller to transfer the RPC function calls.

A typical message:

```
{
  "id": "Random Request ID",
  "callbackIp": "Callback IP",
  "callbackPort": "Callback Port",
  "method": "Method Name",
  "service": "Service ID",
  "params": [
    {
      "argument": {
        "AttributeX": "Primitive Value X",
        "AttributeY": "Primitive Value Y",
        "type": "Argument Typ"
      },
      "type": "Parameter Type"
    }
  ]
}
```

Request Object

A rpc call is represented by sending a Request object to a Server. The Request object has the following members:

- `id` Random Request UUID
- `callbackIp` IP where to send the result of the invocation
- `callbackPort` Port where to send the result of the invocation
- `service` Service ID
- `method` Method Name
- `params` A list of Parameter Objects (see below)

Notification

A Notification is a Request object without an "id" member. A Request object that is a Notification signifies an async function call

Notifications have no results and therefore requires no callback. omitting the "id" member omits the "callbackIp" and "callbackPort" as well.

Parameter Object

- `type` The Parameter Type
- `argument` Objects that can be broken down into primitive types

Error Object

- `code` Integer Error Code
- `message` String Error Message

Processing Instructions

Processing Instructions used when serializing and deserializing composite data types.

Tron Objects

Default Attributes:

- `type` The Argument Typ

Please note that the differentiation between Parameter Typ and Argument Typ comes from the fact, that the system adheres to the Dependency-Inversion-Principle and multiple implementation might exist.

IRpcCallerAppStub

- `ip` The IP Address of the Application Caller Stub (String)
- `port` The Port of the Application Caller Stub (int)

PlayerUpdate

- `movingDirection` The Name of the Direction the Player is heading (String)
- `dead` weather the Player is dead or not (String)
- `version` Incremented Version Number (int)

DirectoryServiceEntry

- `providerId` The id of the service provider (String)
- `serviceId` The id of the offered service (String)
- `address` The rpc server address (String)
- `port` The rpc server port (int)
- `reachable` weather the service is reachable or not (boolean)

InetAddress

- `address` The IP Address (String)

UUID

- `uuid` The uniq identifier (String)