Autonomous Shuttle Transport System
Case Study

Version 0.4 Beta
Wednesday, 22 January 2003

(Source: SWTPRA-case-study-v0.4b.doc)

Contents

1. Case Study 2
  1.1. Context 2
  1.2. Scenario 2
    1.2.1. The railway network 2
    1.2.2. Shuttles 3
    1.2.3. Orders 3
    1.2.4. Income and expenses 3
  1.3. Required Components 4
  1.4. System Evaluation 4
  1.5. Additional information 5
2. The Shuttle Interface 7
  2.1. Introduction 7
    2.1.1. Architecture 7
  2.2. Interaction Protocols 8
    2.2.1. Shuttle Commands 8
    2.2.2. Broker Interaction 9
    2.2.3. Additional messages 12
    2.2.4. A typical sequence of interactions 14
    2.2.5. Example Statechart 15
3. Visualization Interface 16
  3.1. Architecture 16
  3.2. Interaction Protocols 17
    3.2.1. Initialization 17
    3.2.2. Connection 18
    3.2.3. Simulation 19
  3.3. Overall Protocol 20
  3.4. Summary 20
A Appendix 21
  A.1 Internal & external Message of the Shuttle 21

1. Case Study

1.1. Context

In the context of a series of new research projects at the University of Paderborn ("New rail-technology Paderborn", Graduate School of Dynamic Intelligent Systems, special research area "Self-optimizing Systems in Engineering") a new rail-based transport system is being developed. The system is intended to enable individual traffic of people and goods, which today is mainly conducted by cars and lorries, by autonomously acting shuttles on rail. This autonomy shall eliminate the disadvantages of modern trains concerning individual transport. The autonomous control as well as coordination of the shuttles between each other is one of the problems to be solved during the projects.

Motivated by these project aims, several different demand oriented shuttle controls and visualization software is to be developed. The case study is intended to permit multiple groups to independently design and implement the required control software, along with other building blocks necessary. These building blocks shall be modular by design, be able to cooperate with building blocks from other groups and be interchangeable during run-time of the system. The solutions of the groups can be finally evaluated in form of a simulation of the whole transport system.

A prototype version of this system has already been developed, which the groups are expected to use and integrate their solutions into.

1.2. Scenario

The following simplified system shall form the basis of the case study. The system consists of interconnected stations. Shuttle is being assigned orders to transport goods between certain stations. Successful completion of an order results in a monetary reward for the shuttle involved. In case an order is has not been completed in a given amount of time, a penalty is incurred. New orders are made known to all shuttles, thus all shuttles can make an offer. The shuttle with the best, i.e. lowest offer will receive the assignment. Using the tracks will incur a toll, depending on the distance covered. Maintenance of the shuttles is possible at any station and will cost both time and money.

1.2.1.The railway network

The railway network consists of stations, track and switches. Tracks can be traveled upon in one direction only. A switch is configured as a converging or branching junction depending on the directions of the adjoining tracks. A section of tracks consists of any number of connected tracks and switches. A section of tracks between stations is called a connection.

The direction of a connection is determined by the direction of its constituent sections of tracks. A connection can only be traversed in the predefined direction and changing the direction while underway is not possible. Connection can share sections of tracks and switches, while there must only be one connection between two stations in one direction. See figure 1 for an example of what a railway network could look like.

1.2.1.1. Stations

Any number of shuttles can be present at a station at the same time. The duration of a shuttles stay at a station is not considered maintenance time. Maintenance must be explicitly scheduled.

1.2.1.2. Connections

Connections between stations can be temporarily disrupted. Shuttles currently traveling on a section of tracks at the time of this disruption are not affected by it, and will be able to complete their journey. Shuttles at stations will not be able to use this connection. They can however use a different route. All shuttles will be informed of the disruption and its duration.

1.2.2. Shuttles

Order processing is handled by the shuttles. In the case of a shuttle having accepted more than one order, it can only execute one order at a time. Only when a shuttle has fully completed an order, the next one can be processed. Each shuttle has to have finished at least on order, in order not to be subsequently disqualified.
To complete an order a shuttle has to travel to the start station and then to the destination station. This sequence must be completed in a predefined time span; otherwise a penalty will be levied. (See section 2.4.2) Order-processing begins with the loading at the start station and ends with unloading at the destination. Loading or unloading at other stations is not permitted. A shuttle traveling on a section of tracks can neither change direction nor choose another destination. A travel decision is only possible at a station before the journey has begun.

1.2.3. Orders

Orders are made known to all shuttles by a broker. An order defines start and destination stations as well as the allowed time for completion. The deadline is derived from the time of acceptance of an order and the predefined processing time, which begins at the time of acceptance.
Order assignment follows a strict pattern. Firstly all shuttles are informed of the new order. Within a certain period of time, any shuttle can make an offer, which defines the payment it will receive after successful completion of that order. The shuttle having made the lowest offer will receive the assignment. In the event of two equal offers, the assignment will go to the shuttle that first made to offer. A shuttle can accept more than one offer, but is only able to execute on at a time.

1.2.4. Income and expenses

1.2.4.1. Income

In the beginning every shuttle will receive a fixed starting capital. Afterwards a shuttle's only means of income generation is by successfully completing orders. Payment occurs at the destination station and the profit will be transferred to that shuttle's account.

1.2.4.2. Expenses

There are three different types of costs:

1. Toll for using connections between two stations
   The total cost is the sum of all costs of sections of tracks used. It is payable immediately at the destination
2. Maintenance
   After traveling a certain distance, maintenance has to be carried out. The distance depends on the number of tracks and switches traveled. If a shuttle exceeds this limit, maintenance will be carried out at the next station automatically, and it will not be able to leave the station until maintenance is finished. However, repairs can be scheduled before the distance limit is reached, and carried out at any station. Payment is immediate.

3. Penalties
   If an order is has not been delivered in time, a penalty will be imposed. If a shuttle currently carries out the order, it has to complete it. The transport will always be paid, but the penalty is incurred at the deadline. If a shuttle has not loaded the order at that time, the penalty is doubled and the order is deleted.

If a shuttle is at a station and its account shows a negative balance, it will not be permitted to leave this station, and is retired.

1.3. Required Components

Multiple participants may each represent a company, developing a piece of "intelligent" control software for a shuttle within in the transport system. The aim is for the software to achieve the best possible economical result by observing the shuttle environment and implementing a clever strategy. This implies on the one hand that the control software avoids obvious mistakes as for example traveling to stations with no outgoing tracks, and on the other hand it should take into consideration changing accessibility, impossible time limits or the risk of high penalties.
The groups will also be required to develop a visualization component as well as a topology editor, in order to design and display suitable test networks.

The required components are in particular

1. **Shuttle control**
   The main task will be the implementation of the shuttle control software. It should be designed as a modular system, which is easily extendable. In order to do this, all relevant states of the shuttle will have to be identified and implemented. New requirements and additional states should be appendable without too much difficulty.
2. **Visualization component to display the simulation**
   The visualization must at least be able to display the railway networks and moving shuttles. Additional information may be shown, as far as it is made available by the simulation kernel.
   The information required by the visualization will be made available by a special mechanism of the simulation kernel. This mechanism will have to be analyzed during the re-engineering phase. It enables the visualization component to be exchanges during run-time of the simulation kernel, in order to present the components of each group. (See 3.2)
3. **Topology editor**
   For the generation of example tracks an editor will have to be implemented. The editor should be easy and intuitive to use. Reuse of parts of the visualization component is not forbidden.

During a final contest between the groups and their shuttles it can be determined who developed the most successful shuttle strategy and whose visualization component and editor have been most useful.

1.4. System Evaluation

Every shuttle will be submitted to a trial run, during which its reactions to certain situations will be examined and evaluated. This test is intended to guarantee that the shuttle meets all requirements to qualify for the final contest. This trial does not influence the outcome of the final contest.

During the contest all shuttles will compete on the same railway network. These networks will not be disclosed until that time. All shuttles will start under equal conditions, with the single exception of the starting station, which will be determined at random.
After the start of the simulation the shuttles will follow their winning strategy. They will compete for orders and will try to increase their capital by successful completion of orders. If a shuttle goes bankrupt, it will not take part during the remainder of the contest.
At the end of the tournament the shuttle with the highest capital will be declared winner. As the tournament consists of several rounds, points will be awarded Formula One style. In the event of two shuttles having the same number of points, both will share that place and the next place will be empty. In case of disqualified shuttles, the time of disqualification determines the rank. He, who can compete within in the market the longer, will be assessed the better.

1.5. Additional information

1. Topology elements

1.1 Distance

- In order to facilitate correct depiction of all topology elements, please note the minimum distance between the following elements:

Station	60
Crossing	30
Switch	30
Track	20

All measurements refer to Manhattan-geometry. A station is located within a 60x60 rectangle and thus has a distance to either edge of the rectangle of 30.

1.2 Valid topologies

- Sections of tracks my cross each other. Note that this does not constitute a four-way switch!
- The end coordinate of a track leading to a station must be the same as the coordinate of the station.
- While there must not be tracks leading nowhere, dead ends in the form of stations with no outgoing tracks are possible.
- There is no maximum size for a topology. However, for the tournament the topology will not be larger than 2000 x 2000 units.
- The topology may contain curved corners, although this is not required.
- There is no maximum number of tracks or switches between stations.
- The topology editor has to make sure that a generated topology adheres to all of the above rules. It has to guarantee that the kernel will be able to load and process the topology correctly.

2. Switches

- The lengths of the arms of the switch must not differ by more than a factor of 1.5.
- Switches with more than three in-/outgoing tracks are not possible. The only possible configurations are 2 in / 1 out or 1 in / 2 out.

3. Shuttles

- Shuttle agents may not save any data on the hard drive.
- There may not be any direct communication between shuttle and GUI, other than the "official" messages provided.
- All offers exceeding the transportation costs by a factor of more than ten, will be ignored by the broker agent, in order to avoid that a shuttle always makes very high offers, and gets the assignment when no other shuttle places a bid.
- Every shuttle must be named. This name will be forwarded to the visualization component.
- Shuttles will not know the utilization of certain sections of tracks by other shuttles.
- The shuttle agent may and should consist of more than one class, but may only utilize one thread. Additionally care should be taken not to use too much memory.
- There is a mandatory package hierarchy:

4. Visualization

- The visualization may not consist of more than one client, e.g. one to display the railway network and another for shuttle states.
- The simulation time factor will be made known to the visualization at the beginning of the simulation and every time it is changed.

5. Tournament specifics

- Connections can be assumed to be disrupted only rarely, but when they are they will be for a significant length of time.
- Order availability will vary. However, there will probably be no extreme cases, like no orders at all over long periods, or too many orders at a given time. The system will try to maintain the predefined average number of orders generated.
- The tournament will most likely utilize a topology with about 20-30 stations and one shuttle per group.

2. The Shuttle Interface

2.1. Introduction

The main piece of software to be designed is the shuttle. The simulation software being used provides the software engineer with a protocol which enables the shuttle to communicate with the kernel. The shuttle can issue commands, for example to move to a station, or request information about its environment, like whether a particular connection between two adjacent stations is currently interrupted or not. It is imperative that this interface is understood well, as it represents the only possibility for action.

The shuttle itself should be designed in a modular fashion. It is a good idea to separate distinct functionalities, such as an algorithm to find a path from A to B, and the method by which offers are calculated. This will make possible the implementation of different strategies, in addition to making the code substantially easier to understand and modify.

2.1.1 Architecture

The key to understanding the functionality of the system lies in the way messages are being passed to and from the kernel. In order to be able to simulate several shuttles without one being able to hog all resources, a thread is used for each one. While all method calls in java are by nature synchronous, a mechanism was devised to enable a seemingly asynchronous passing of messages. A message handler is responsible for queuing and delivering all messages, relinquishing control of the thread immediately after receipt of the message object. The following diagram shows how the different components of the kernel are incorporated

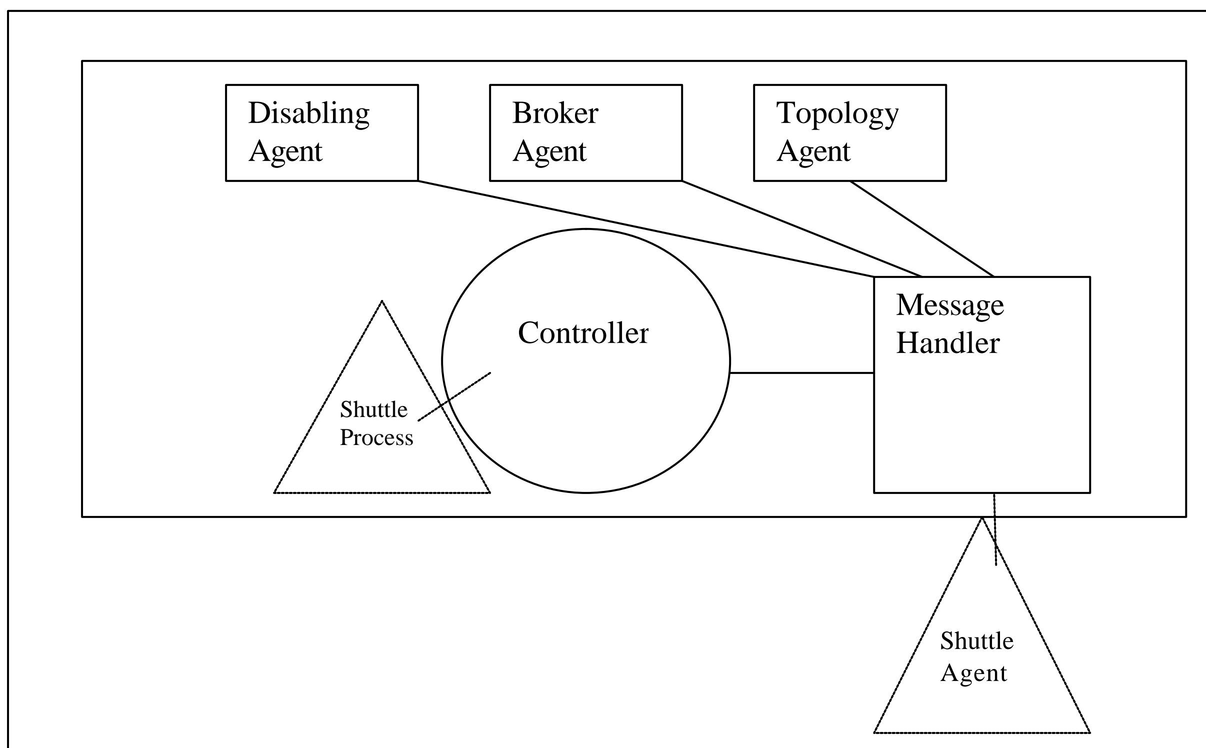


Figure 1.0 Kernel components

Clearly the message handler is the central entity in this diagram. It routes all messages and especially communication with the “outside” class ShuttleAgent, can only be achieved through it.

One can see the three agents on top of the diagram. The disabling agent randomly disables connections between stations. The topology agent is responsible for sending the topology data to all shuttles at the beginning of the simulation. And lastly, and most importantly, there is the broker agent, which controls order creation, assignment and deletion.
The shuttle process can be viewed as the hardware of the shuttle agent, and is thus depicted in a dotted line. It can send messages via an object variable of the class Controller. (See also *2.1 Shuttle commands*)

2.2. *Interaction Protocols*

To understand the following diagrams, one has to keep in mind a few general things about the way kernel and shuttle communicate. As stated above all information is passed on in the form of messages. By doing this, asynchronous communication is simulated. Notice the arrow shape in the diagrams expressing asynchronicity. Furthermore, a delay occures whenever messages are passed to the message handler. A fact that has to be considered, when requesting wake-up calls or calculating internal deadlines.
While we know that the kernel consists of various components and that the message handler passes on all messages, the kernel will be depicted as a single entity. As the messages can by grouped by their content and between whom they are passed, only a few stereotypical ones are explained in the diagrams. A list of all messages can be found in the appendix.

2.2.1.Shuttle Commands

These messages are of a basic command and reply structure. The shuttle agent expresses its wish to perform a certain action, and the shuttle process checks this request, and gives the appropriate response.

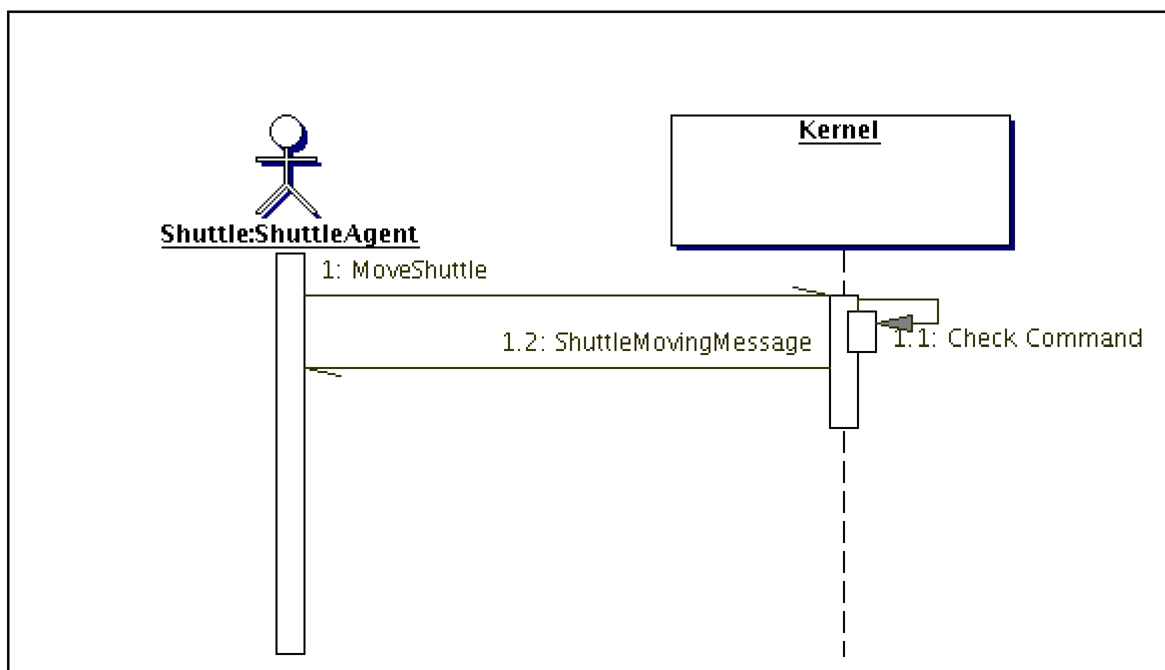


Figure 1 - Shuttle commands – Move Shuttle successful

Here a MoveShuttle command is sent to the kernel. The shuttle process checks, if all requirements have been met to execute this order, i.e. no need to repair, shuttle located at station, etc. Apparently that is the case, and a ShuttleMovingMessage is sent back to the shuttle indicating this.

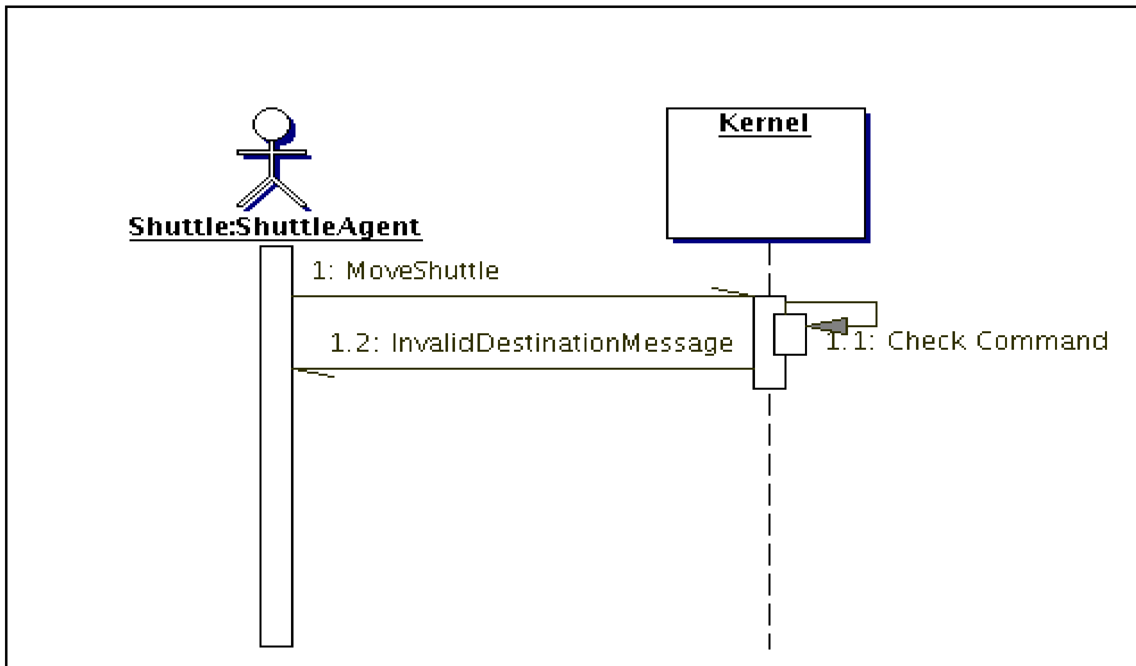


Figure 2 - Shuttle command - MoveShuttle unsuccessful

In this case the command was not successful as the shuttle issued a move order to an invalid destination.
This message-reponse type protocol is repeated for all messages in the shuttle commands group. Each can be sent to the kernel, i.e. the shuttle process, and the shuttle agent will the receive an answer as to whether the command will be executed, or if that was not possible.
To inform the shuttle about its arrival the kernel sends another message. The shuttle can then plan its next move.

2.2.2.Broker Interaction

After looking at an interaction initiated by the shuttle, we now examine one where a kernel component starts the interaction by sending a message to all shuttles saying that a new order is available.

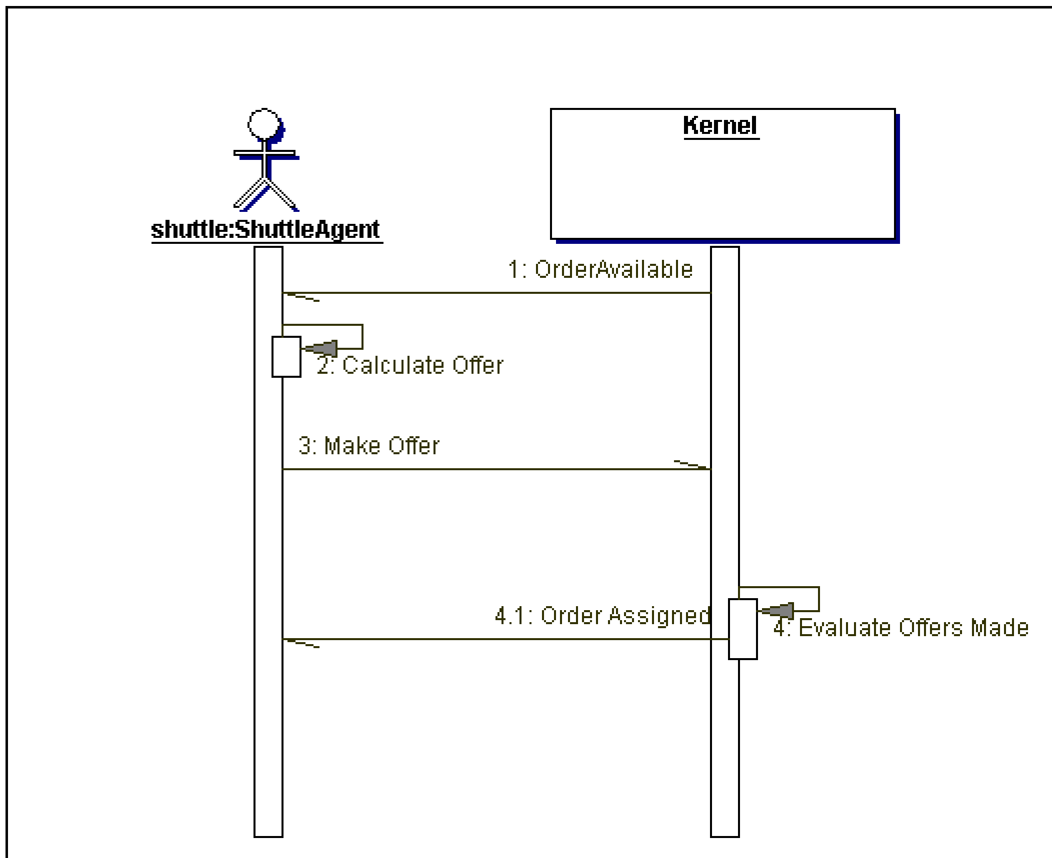


Figure 3 - Order Assignment – Successful Offer

In the diagram above, an order is created by the kernel, and the message OrderAvailable is passed to the shuttle via the message handler, which then calculates an offer amount and responds with a MakeOffer message. Again this is forwarded to the kernel by the message handler. It is important to note that between numbers 1. *Generate Order* and 4. *Evaluate Offers made* all shutlles have the opportunity to make an offer, and the at the end of an arbitrarily chosen, but previously announced time span, the order is assigned to the shuttle having made the lowest offer.

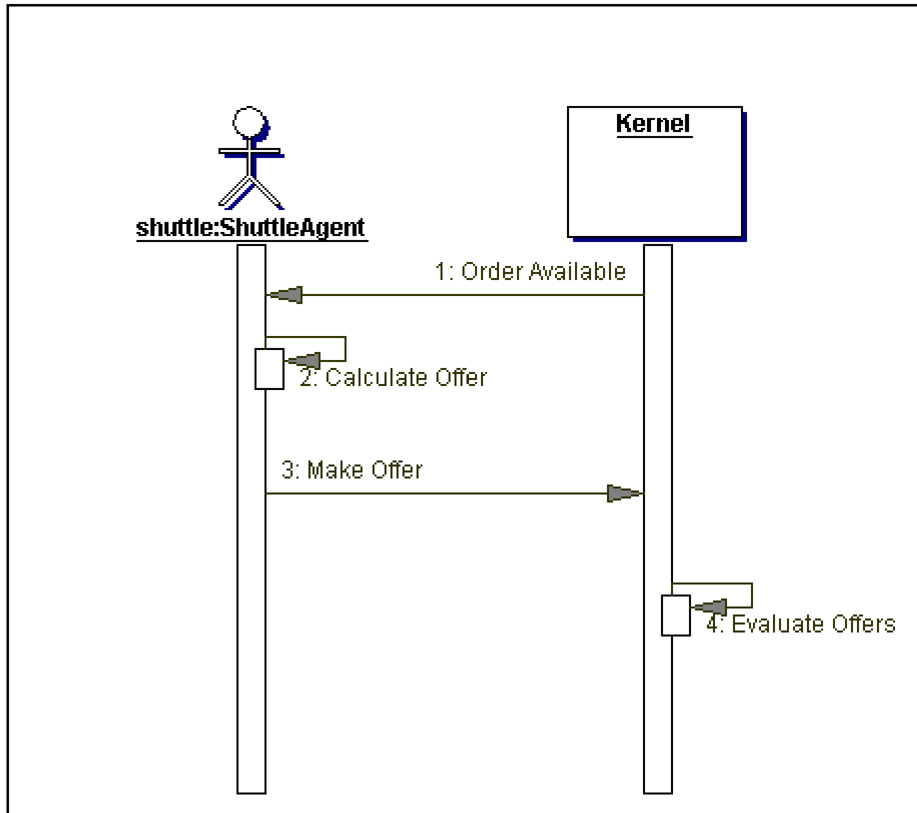


Figure 4 - Order Assignment - Offer unsuccessful

This diagram shows a shuttle making an offer for an order whose offering time has previously expired. Again the message handler serves as an intermediary. After 2. *Evaluate Offers* the kernel will assign the order to a different shuttle, or delete it if no offers were made. Please note that the kernel does not inform the shuttle of the fact that the offer made refers to an order that is no longer available. Neither does the kernel send any messages to let a shuttle know if it has not received an order, even if that offer was made in due time.

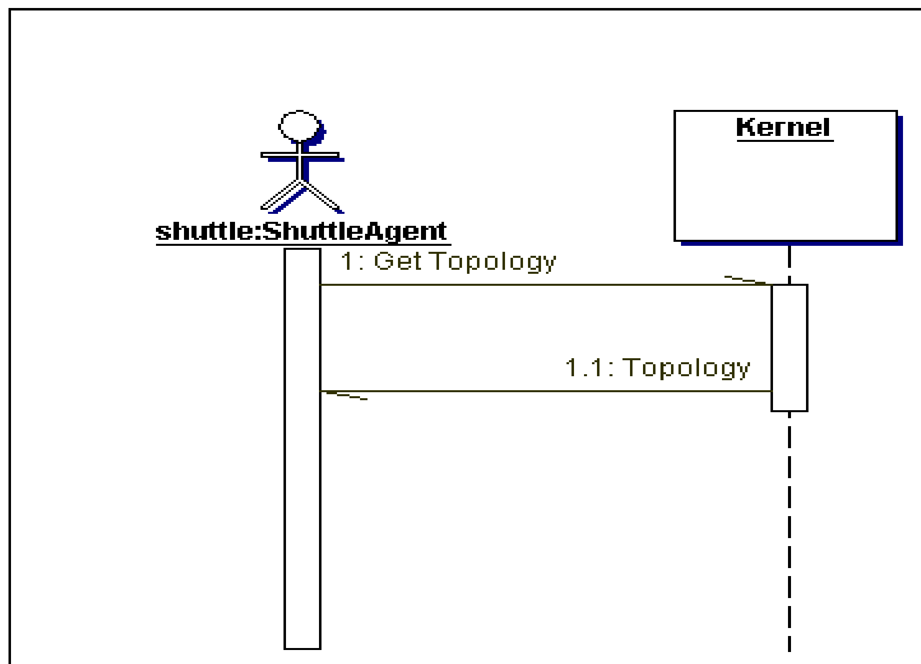


Figure 5 - Receiving Topology

At the beginning of the simulation the shuttle has to be informed about its environment. The topology data represents the most important part. The topology object returned by the kernel contains a collection of TopologyDataObjects, which has to be processed by the shuttle agent.
Additionally, the kernel sends a GameConstantsMessage, to inform all shuttles about things like simulation speed, loading time, etc. This is done automatically and does not have to be requested by the shuttle.

2.2.3. Additional messages

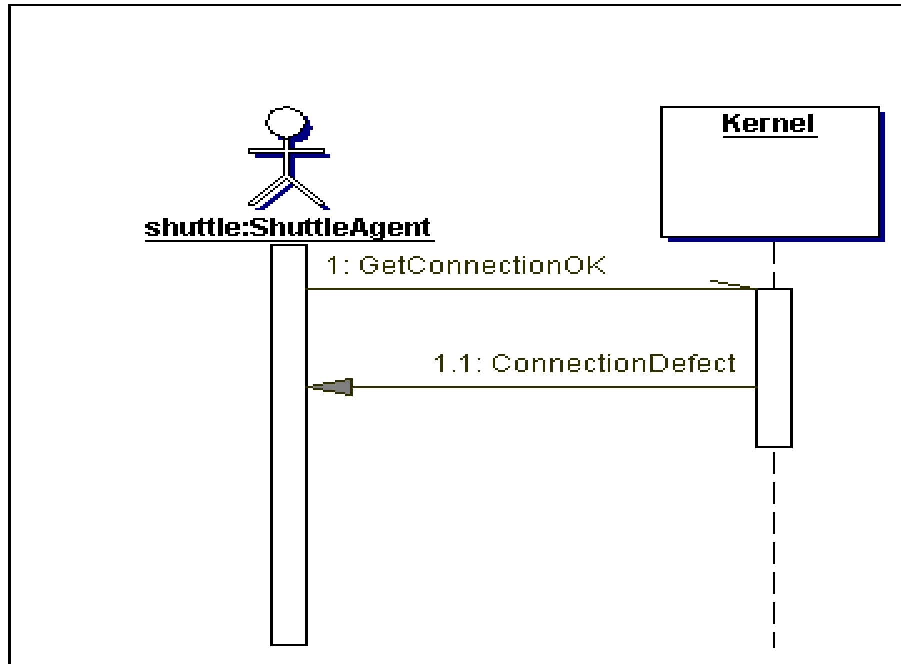


Figure 6 - Checking connection status

There are several pieces of information the shuttle may need during the simulation. It can, for example check whether a particular direct connection between two stations is currently interrupted. The diagram above shows a situation, in which this is not the case. Other such interaction include information about the current simulation time and the shuttle status, which contains data about its location as well as its account balance.

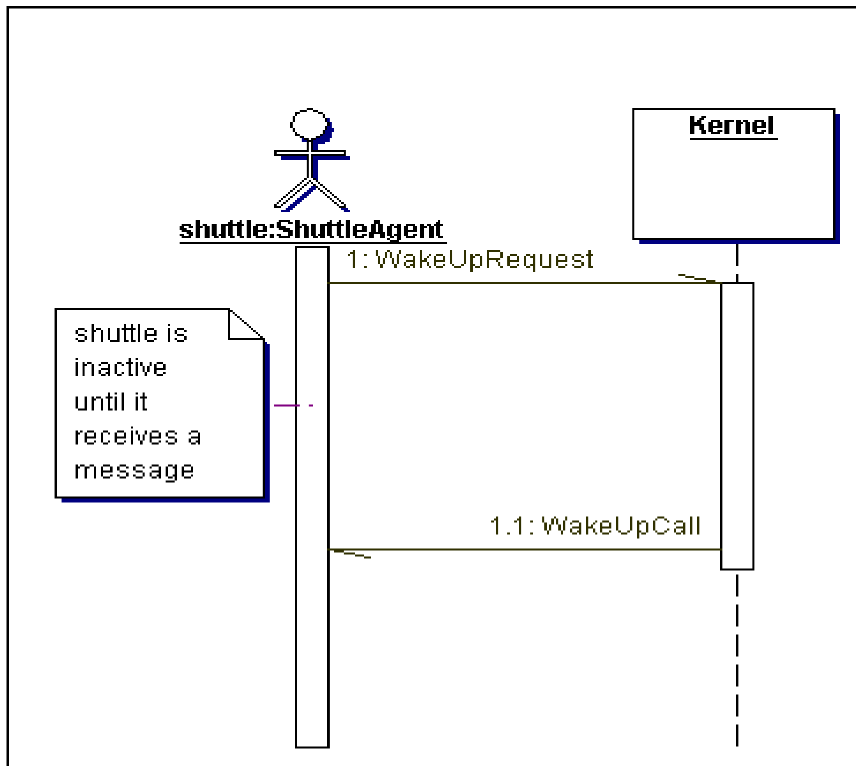


Figure 7 - Requesting wake-up calls

In order to free system resources, when the shuttle is not actively calculating something, the shuttle is able to request wake-up calls from the kernel. This can and should be done, if the shuttle wants to waits for a certain amount of time, after which it will not necessarily be given notice automatically. This could be the case when waiting for a potential order after an offer has been made. You can see how this is used below.

It is important to note that the shuttle will be activated whenever it receives a message, even if the time for the wake-up call has not yet come.

2.2.4.A typical sequence of interactions

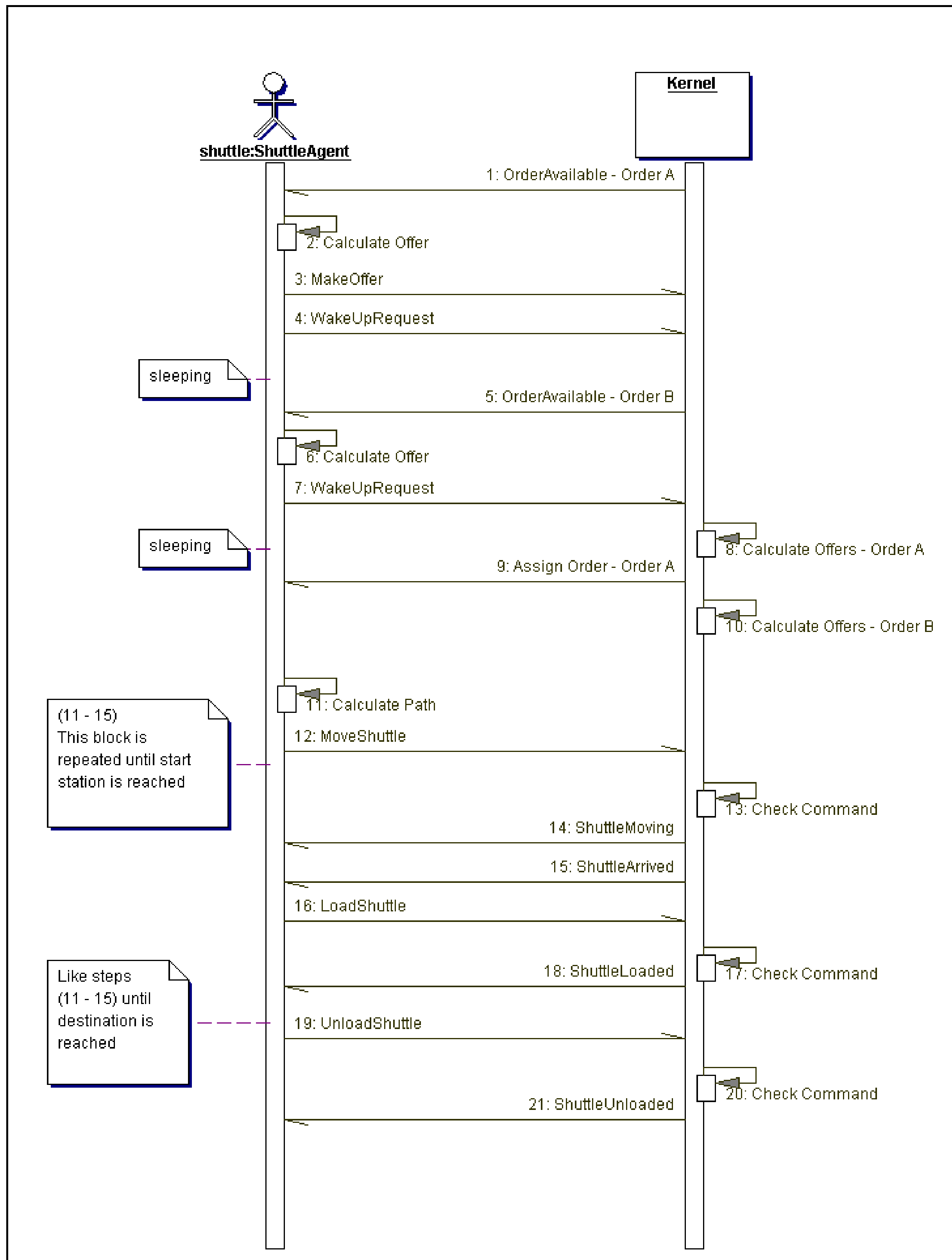


Figure 8 - Sample interaction

The diagram above shows a typical sequence of interactions between shuttle and kernel. It begins with a new order "A", the creation of which is announced to all shuttles. After

deciding that it is worth making an offer, the shuttle goes to sleep, i.e. requests a wake-up call from the kernel. Notice that the shuttle is activated by the advent of order “B”. It decides not to make an offer here, and goes to sleep again. When the time limit during which offers for order “A” are accepted expires, the kernel selects the shuttle which made the lowest valid offer, and informs it about its new assignment. Offers for order “B” are processed by the kernel as well, but the shuttle depicted in the diagram will not be notified, as it did not make an offer for it. The shuttle proceeds to move to the start station, having previously decided upon a route to take. The sequence of move command, move acknowledgement and arrival notification, is repeated until the station where order “A” can be picked up is reached. After successfully loading the order, the shuttle proceeds to the drop-off station the same way as described before. The kernel internally updates the bank balance of the shuttle and deletes order “A”. Depending on the shuttles strategy it may start making bids for new orders or, having been assigned an order already, proceed to the next start station.

2.2.5.Example Statechart

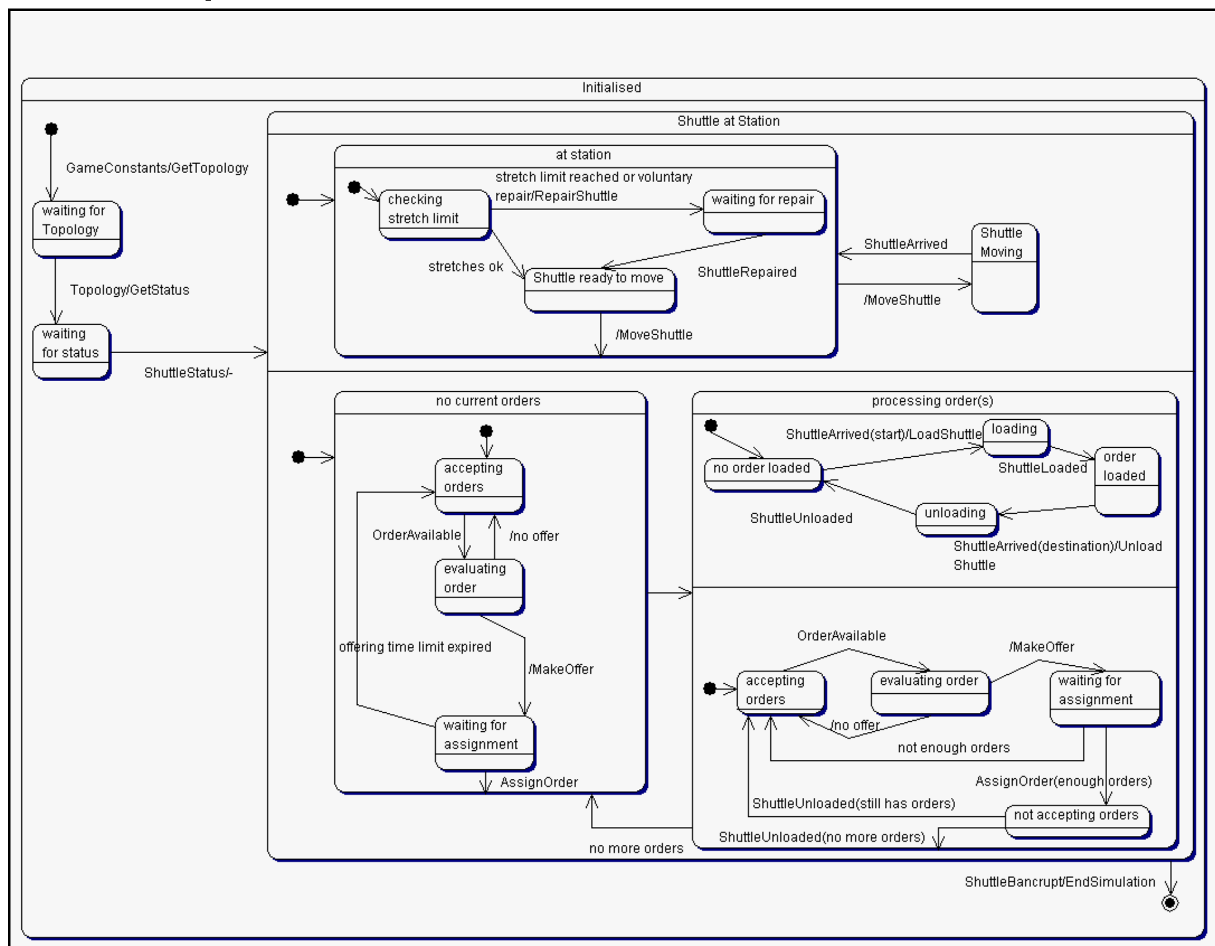


Figure 9 - Statechart with parallel superstates

This statechart shows a possible dependency of events and actions. After having received game constants, topology and its status, the shuttle waits at its start station for new orders. If an order is generated it follows the procedure outlined in *2.2 Broker Interaction*. When an order is assigned to the shuttle, it enters the superstate processing order(s). In parallel, the shuttle begins to move to the order’s start station and so on.

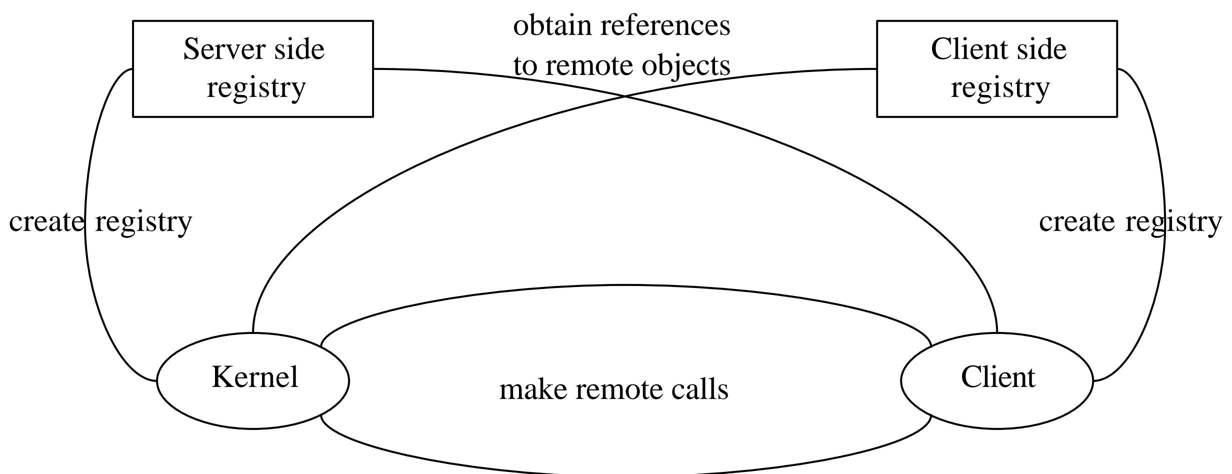
It is important to realize that this is not the only way this can be done. To illustrate this, please note the ability of the shuttle to accept more than one order, although of course, it can only transport one at any given time. A shuttle which only considers newly created orders, when it is idle is also perfectly viable.

3. Visualization Interface

The second main part of the project is the visualization client. The simulation kernel provides an interface that makes possible to visualize a simulation. To develop your own visualization, you have to understand how the kernel and the client can communicate, and what are the most important changes in the kernel, that need to be visualized.

3.1. Architecture

The kernel and the client are two different applications, executed by different Java virtual machines. To visualize a simulation, the client has to react to kernel changes, that means that two virtual machines have to communicate with each other. This communication is realized with the Java Remote Method Invocation (RMI) technology. RMI makes it possible to define objects in a JVM, which are accessible from other JVM's (possibly on other hosts). These objects are called remote objects, and they must implement a remote interface. If an object implements a remote interface, the methods specified in the interface, can be called from other JVM's. Before able to make remote calls, applications have to obtain references to remote objects. References can be read from registries, passed as an attribute or a return value. By developing a simple distributed application, normally one registry is needed to get the first references, the other references can be passed as attributes or return values. The example of the simulation kernel and the visualization client is a bit complicated, registries have to be created on both sides. The client don't passes references to the kernel, so a client side registry have to be created to make it possible for the kernel to make remote calls on the clients.



Java RMI provides a lot of utility classes to create a registries, bind objects to a specified registry and look up remote objects. The most important classes are:

- `LocateRegistry`: This class provides methods to create a remote registry on a particular host that accepts calls on a specific port. The `createRegistry()` method of this class returns an object which implements the Registry interface.
- `Registry`: This interface defines the `bind()` and `rebind()` methods. Remote objects can be bound to a registry by calling one of these methods.
- `Naming`: This class provides methods for obtaining references to remote objects in a remote registry. The `lookup()` method takes a `String` as argument in the form of *rmi://host:port/name*, and returns the reference to the remote object bound with the name to the registry on the specified host.

For more information about the Java RMI technology visit:
http://java.sun.com/products/jdk/rmi/.

3.2. Interaction Protocols

3.2.1 Initialization

When starting the kernel, a registry will be created on the default port (1099). After creating the registry, the kernel binds an instance of the `Visualization` class with the "Visualization" name to the registry.

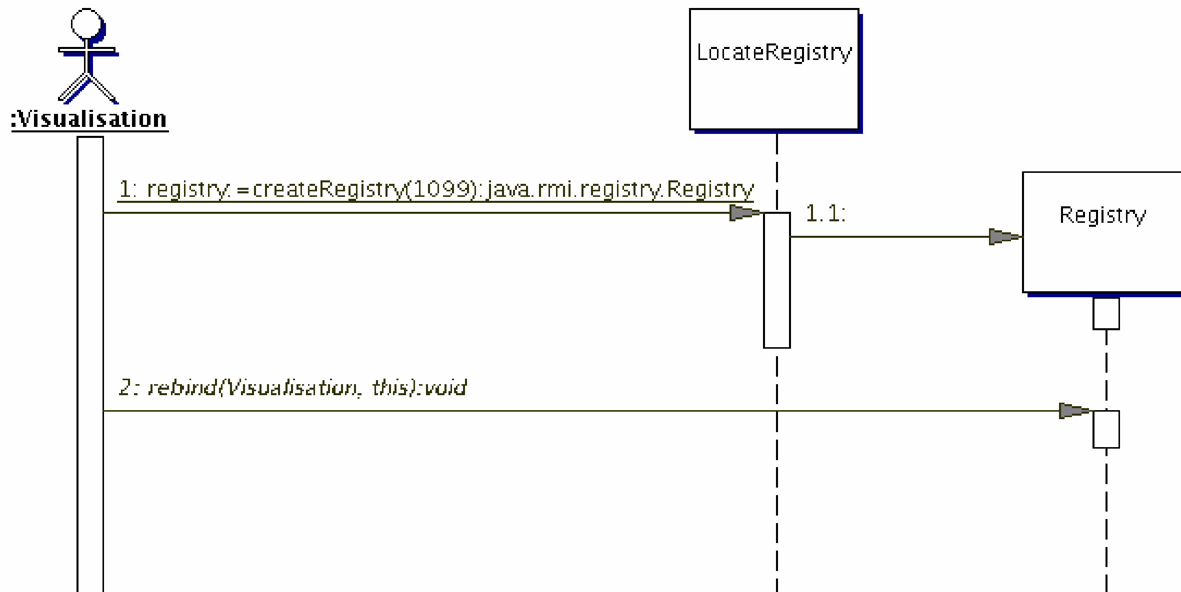


Figure 10 - Kernel init

The client have to build his own registry too, and bind a client object with a unique client name to the registry. This can be done in the same way as the kernel does.

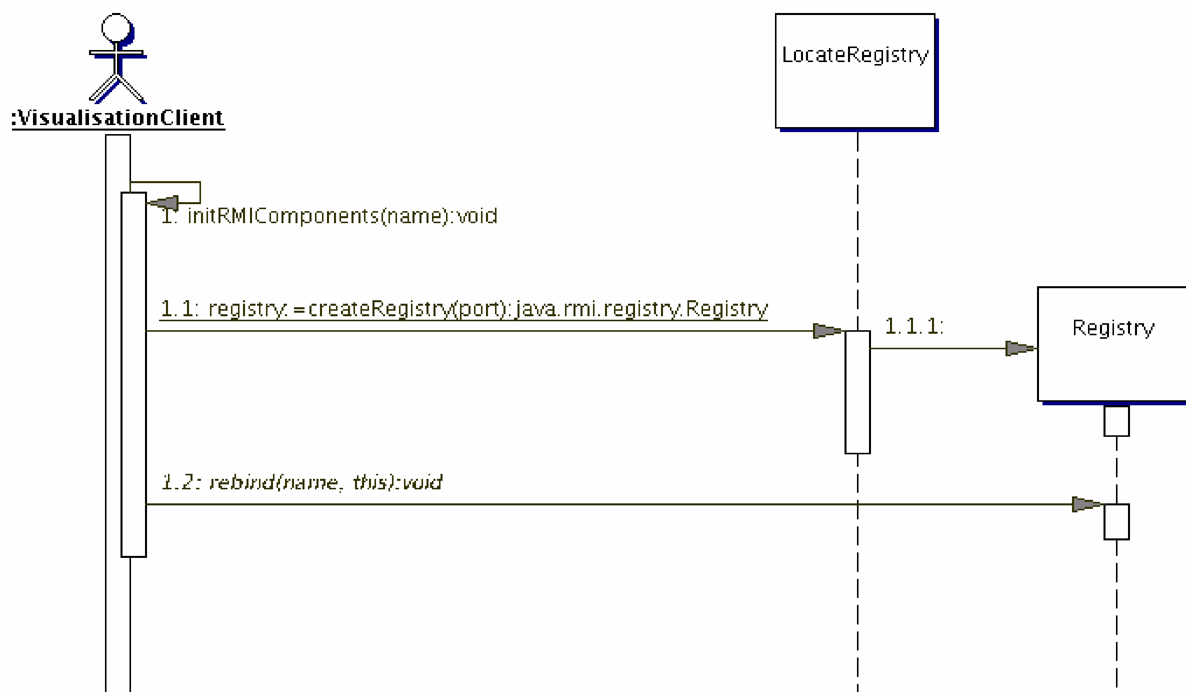


Figure 11 - Client init

3.2.2.Connection

After the remote registries are created on both sides, the client may be able to connect to the kernel. First of all the kernels remote object should be looked up, which is bound to the kernel's registry with the "Visualization" name. Obtaining a reference to this object can be done by calling the `lookup()` method of the Naming class. To establish connection to the kernel, the client must call the `connect()` remote method of the remote object. The connect method has three attributes: client name, client address and client port. The client name must be unique, because the kernel uses a `Hashtable` with the client names as keys to store references to the clients. If the client name is not unique the kernel refuses the connection from the client. The `connectionRefused()` method is called by the kernel, which must be implemented by the client to handle a possibly broken connection. The following figure illustrates a refused connection:

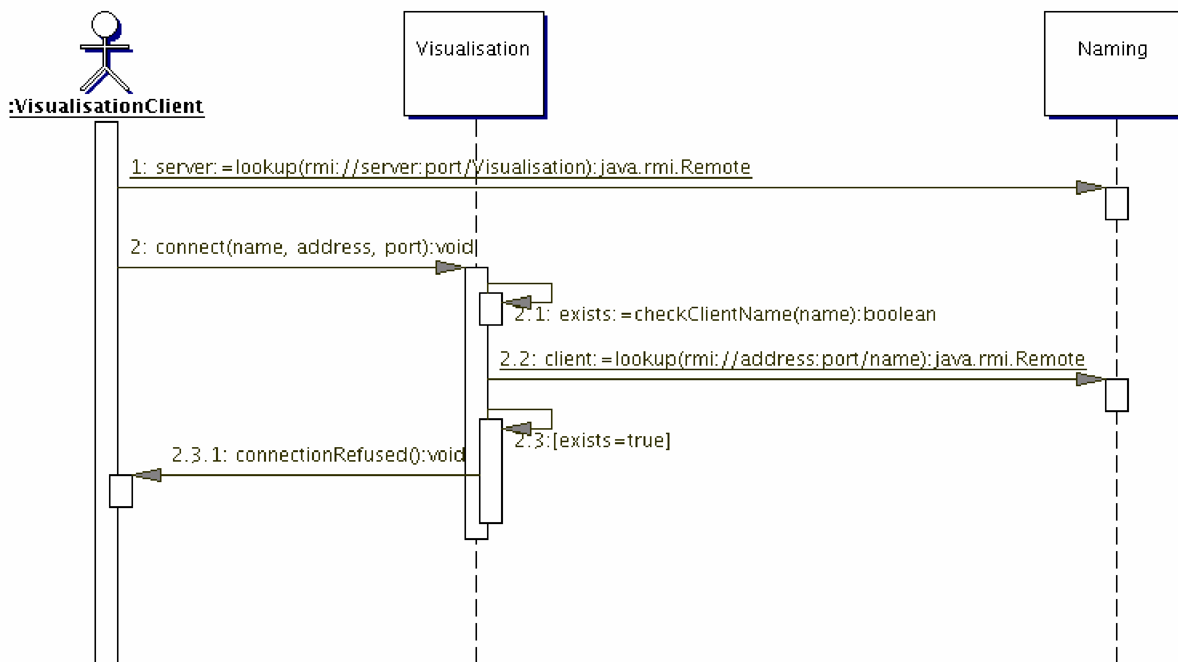


Figure 12 - Client connect with error

If the client name is unique the connection can be established between the kernel and the client. In this case the client will be initialized by calling it's `init()` and `update()` methods (see figure 4). First the `init()` method is called, with the topology (loaded by the kernel) and the init time arguments. After the client got the topology, the kernel calls the `update()` method of the client several times. The `update()` method takes an instance of the `RemoteObj` class and an integer as arguments. The fist argument is rather an instance of a subclass of the `RemoteObj` class, these classes are similar to messages and provide the needed information to visualize every changes in the kernel, the second argument is the time when the "message" was sent to the client. By the initialization the following remote massages will be sent:

- `RemoteShuttleMoved`
- `RemoteOrderCreated`
- `RemoteShuttleStatusChanged`
- `RemoteShuttleDisqualified`
- `RemoteGameConstants`

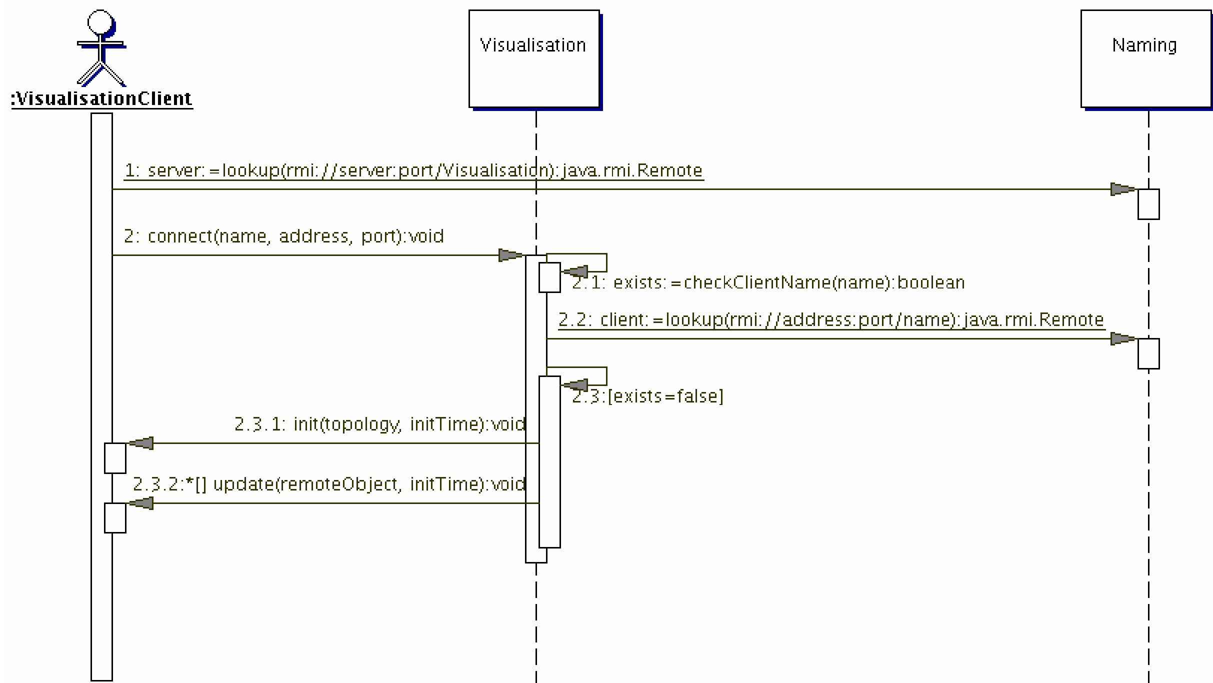


Figure 13 - Client connect without error

3.2.3.Simulation

After a successful connection the `update()` method of the client will be used to send messages during the simulation (see figure 5). The client have to implement this method to handle the remote messages properly. For more information on remote messages see the API documentation of the kernel.

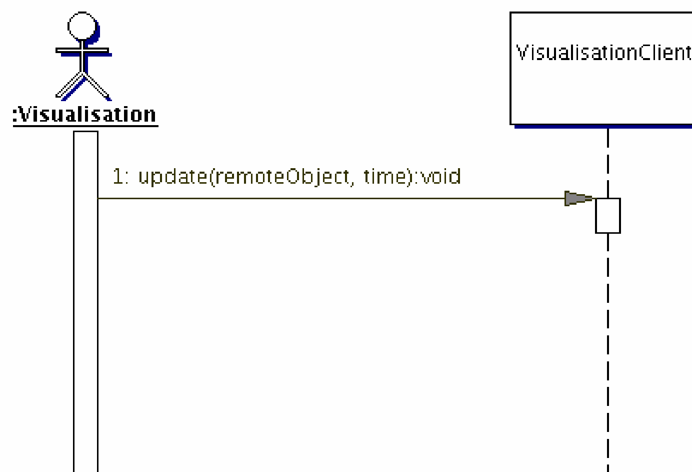


Figure 14 - Client update

The visualization client can connect to the kernel and disconnect from it at any time during the simulation. To disconnect from the kernel the client should call the `disconnect()` remote method. By calling this method the kernel removes the stored references to the client.

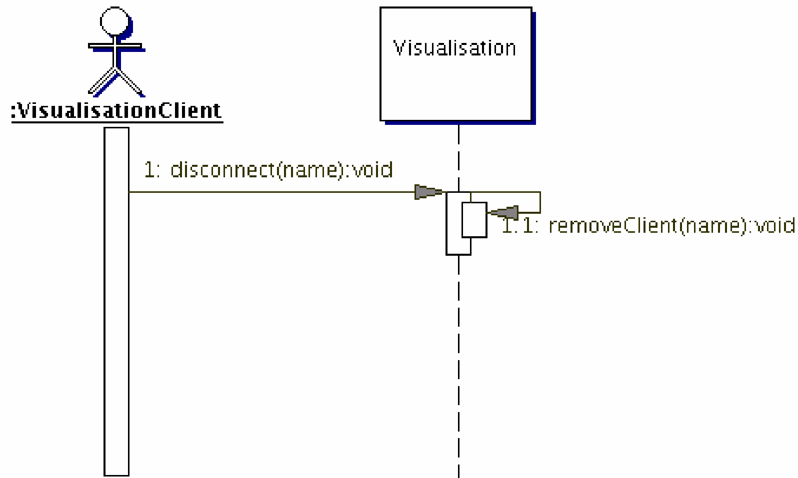
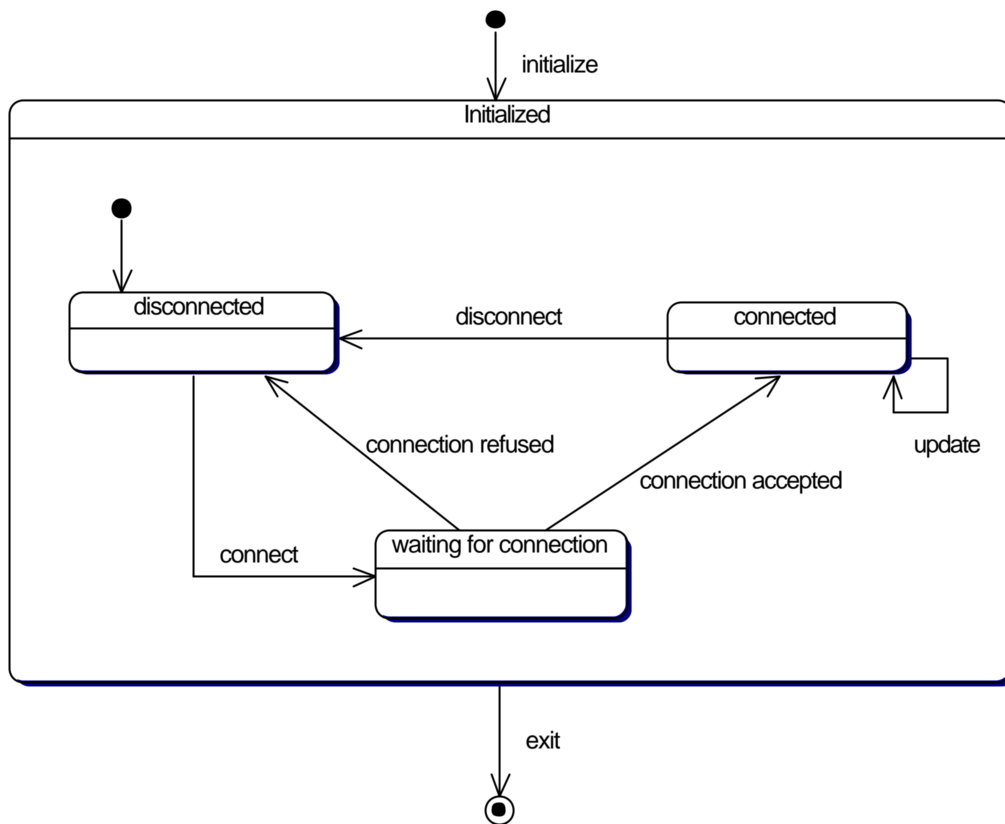


Figure 6 - Disconnect from the kernel

3.3. Overall Protocol

To summarize the actions described by the sequence diagrams, we can make the following state chart:



3.4. Summary

Appendix

A.1 Internal & external Message of the Shuttle

External messages between the ShuttleAgent on the one hand and Controller and BrokerAgents on the other

Messages from ShuttleAgent to Controller

- GetActualSimulationTime
- GetConnectionOk
- WakeUpRequest
- GetShuttleStatus
- ShuttleActionCommand (abstract)
  - MoveShuttle
  - LoadShuttle
  - UnloadShuttleMessage
  - RepairShuttle

Messages from Controller to ShuttleAgents

- GameConstantsMessage
- ProtocolErrorMessage
- ConnectionStatusMessage (to all ShuttleAgents)
- AssignOrder
- ShuttleStatus
- PaidPenaltyMessage
- EndGameMessage (to all ShuttleAgents)
- WakeUpCallMessage

Message generated by ShuttleProcess and sent via Controller object variable

- ShuttleLoaded
- ShuttleUnloadedMessage
- LoadShuttleErrorMessage
- UnLoadShuttleErrorMessage
- InsertedInQueueMessage
- ShuttleMovingMessage
- ShuttleArrivedMessage
- ConnectionDefectMessage
- StationNotReachableMessage
- InvalidDestinationMessage
- MaxStretchLimitReachedMessage
- ShuttleBancruptMessage
- ShuttleDisqualifiedMessage
- ShuttleRepaired

Messages from ShuttleAgent to BrokerAgents

- GetTopology (to TopologyAgent)

- MakeOffer (to BrokerAgent)

Messages from BrokerAgents to ShuttleAgents

- Topology
- OrderAvailable (to all ShuttleAgents)

Messages from BrokerAgent to Controller

- TryToCheatMessage (MessageHandler)
- WakeUpRequest (Disabling & Broker Agents)
- SetConnectionDefect (Disabling Agent)
- NewOrderCreated (BrokerAgent)
- OrderAssignedToShuttle (Broker Agent)
- OrderDeleted (Broker Agent)

Messages from Controller to BrokerAgents

- WakeUpCallMessage