Train Transit System

Martyna Majewska

7866641@studenti.unige.it

Date of project and report delivery: 31.01.2025

1. Details of the proposal

The project proposal focuses on the transit of trains through a network of single-track tunnels located between two stations. It uses the concept of resource allocation and graph traversal.

1.1 Specification of proposal

Transit in this case means a train going from one station to another within a network of one-way tunnels. Each train has a planned, preferred path given as an argument. Tunnels during a given timestamp allow only one train at a time to pass through it in one direction. They will act as shared resources that need to be distributed among the trains, addressing the Multiagent Resource Allocation Problem. The system will simulate real-world constraints, such as adapting train transits to delays or some tunnels being out of service. In addition, some of the trains have special requirements to pass through wider tunnels. The system will act as a MAS implementation.

Agents in the system:

- 6 Train Agents
- 4 Tunnel Agents
- 1 Transit System Agent

1.2 The kind of proposal

Creative project, development and design of a system in the JADE programming language.

1.3 The difficulty of proposal

I would rate my project proposal as **medium** because it involves a system with multiple types of agents, dynamic scheduling through agent communication, and path finding and graph traversal. Idea is based on a concept of shared resource allocation while solving real-time conflicts regarding train delays or tunnels out of service.

2. Introduction

2.1 Problem

This project addresses the challenge of allocating shared resources among agents by assigning time slots for each resource use. The solution involves implementation of agent behaviours for real-time message exchange, allowing for dynamic coordination among competing agents.

The impact of solving such problems extends far beyond theoretical applications. In rail transit systems, intelligent resource allocation can help analyse the use of existing infrastructure: it helps identify and minimize potential bottlenecks in rail and tunnel networks. In addition, identified conflict points may indicate the need to rebuild the rail structure. Agent-based systems enable real-time adaptation to disruptions such as train delays or tunnel disruptions.

The principles applied in this project are not limited to rail networks. Similar models can be applied to traffic management, logistics, and other domains that require coordination of shared resources. As industries increasingly rely on intelligent automation, solving these types of problems will be critical to improving efficient infrastructure management.

2.2 Multiagent Resource Allocation Solution

The system developed in this project is built on the concepts presented in *Dynamic Resource Allocation in MAS*[1], where allocation problem is limited to negotiations and therefore communication between agents. However, the developed system presents a somewhat different approach to resource management. As soon as a train confirms its path defined in the tunnel network, the reservation on tunnels is locked, preventing other trains from overriding this reservation. In addition, if a tunnel does not respond within a predefined timeout period (set by the train), the train cancels its reservation and notifies the tunnel accordingly.

This system addresses the *Multiagent Resource Allocation (MARA)* problem - "*the problem of distributing a number of resources amongst multiple agents*"[2]. In the context of this project, the term of shared "resources" is used to refer to tunnels or platforms and that can only be occupied by a single train agent at a given time interval - its arrival and departure time from a resource.

2.2.1 The Resources

In the scope of this project, resources may be subject to certain constraints, such as being operational only after a specific time slot or being of one of two types: NORMAL or WIDE. This information is crucial for trains, particularly those classified also as wide. These trains are restricted to use tunnels and platforms only of the same type. Furthermore, each resource is assigned a "passing time", which denotes the minimum time required for a train to traverse the node. Delays in rail transit are to be expected. In the event that a train is required to wait for its transit, it will do so in the tunnel rather than on the rails. The presence of a train on the rails can result in significant bottlenecks, as the tunnel will be free and it could accept other trains' reservation proposals. Indeed, the train that follows the first one will not be able to enter one of its desirable rails leading to other tunnels.

2.2.2 Network of Tunnels

The tunnels and platforms can be referred to as the "nodes" of the transit tunnel graph, with the rails between nodes representing the edges of the network. The weight of each edge is the traversal time. The system relies on two graphs to distinguish the possible connections from one station to another. For the sake of this analysis, let's call these stations A and B.

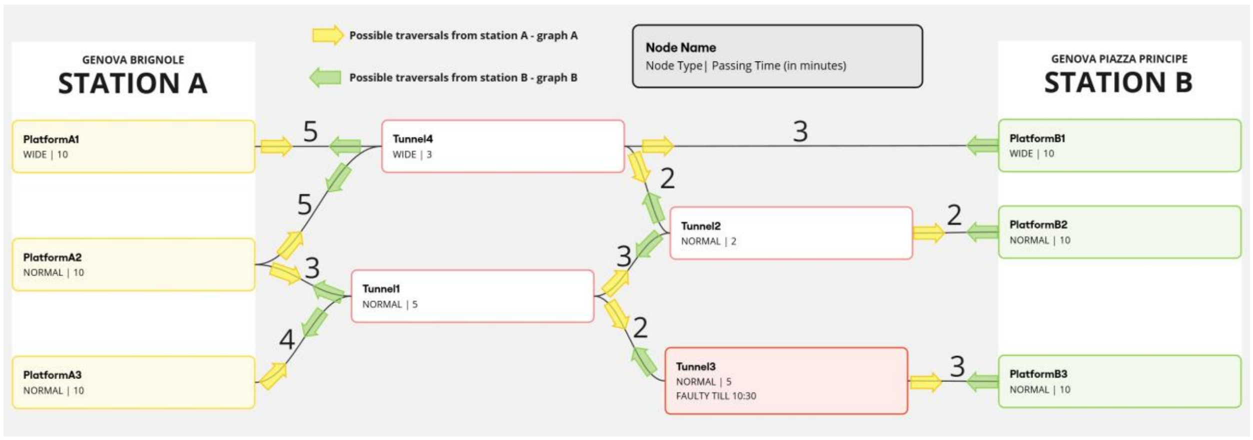


Figure 1: Schema of the graph model currently used in the system

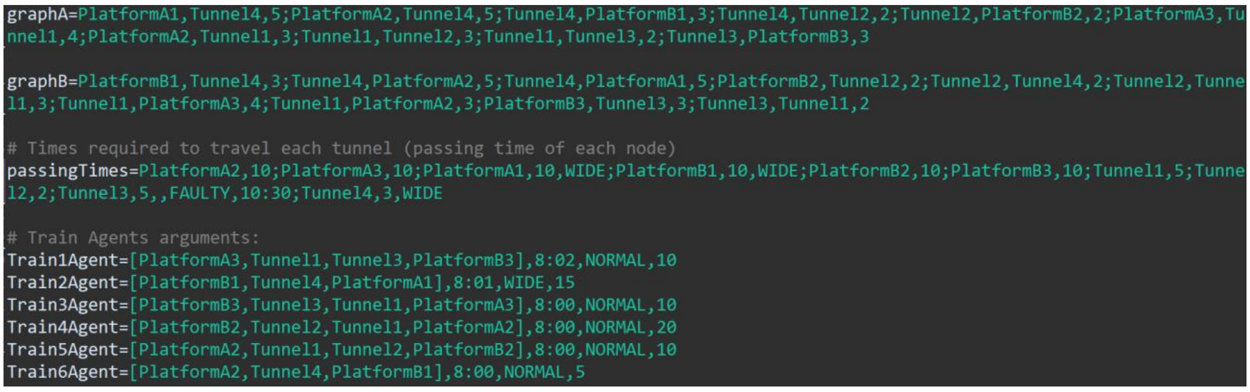
The following schema illustrates the system's current configuration. In the given scenario, if a train begins its journey at station A, platform A1 can only access tunnel 4. Subsequently, from tunnel 4, the train can either proceed directly to station B, platform B1, or traverse to tunnel 2. It is important to note that, from tunnel 2, the train can only reach platform B2. This is due to the fact that there is no benefit in returning to tunnel 1 and then attempting to access Platform B3 via tunnel 3. This is logically impossible, and it creates an assumption in the used system that there is no possibility to go back on rails. The usage of two graphs is only a matter of the different edge directions, respectively, from A and B stations. However, they refer to one and the same node (Tunnel4 in graph A refers to the same node named Tunnel4 in graph B, and together they build one graph of the same nodes). In conclusion, the two directed graphs maintain the train traversal logic and sequence of nodes within the graph.

The final allocation of nodes to trains is determined by the order in which agents request them, with the first agent having priority for reservation. To preserve the agent "competing" logic for resources, the first train that requests a node occupies it. The train imposes a time limit on the collection of answers to its requests, as its primary objective is to be the fastest in securing its path. The system's overall aim is to identify a scheduled plan for each train, with no requirement for optimality. Consequently, the paths retrieved as the solution may not necessarily be the fastest ones.

3. Developed program

3.1 Initial Configuration

In order to start the program, it is necessary to specify the arguments in the *config.properties*[3] file.



```
graphA=PlatformA1,Tunnel4,5;PlatformA2,Tunnel4,5;Tunnel4,PlatformB1,3;Tunnel4,Tunnel2,2;Tunnel2,PlatformB2,2;PlatformA3,Tu
nnel1,4;PlatformA2,Tunnel1,3;Tunnel1,Tunnel2,3;Tunnel1,Tunnel3,2;Tunnel3,PlatformB3,3

graphB=PlatformB1,Tunnel4,3;Tunnel4,PlatformA2,5;Tunnel4,PlatformA1,5;PlatformB2,Tunnel2,2;Tunnel2,Tunnel4,2;Tunnel2,Tunne
l1,3;Tunnel1,PlatformA3,4;Tunnel1,PlatformA2,3;PlatformB3,Tunnel3,3;Tunnel3,Tunnel1,2

# Times required to travel each tunnel (passing time of each node)
passingTimes=PlatformA2,10;PlatformA3,10;PlatformA1,10,WIDE;PlatformB1,10,WIDE;PlatformB2,10;PlatformB3,10;Tunnel1,5;Tunne
l2,2;Tunnel3,5,,FAULTY,10:30;Tunnel4,3,WIDE

# Train Agents arguments:
Train1Agent=[PlatformA3,Tunnel1,Tunnel3,PlatformB3],8:02,NORMAL,10
Train2Agent=[PlatformB1,Tunnel4,PlatformA1],8:01,WIDE,15
Train3Agent=[PlatformB3,Tunnel3,Tunnel1,PlatformA3],8:00,NORMAL,10
Train4Agent=[PlatformB2,Tunnel2,Tunnel1,PlatformA2],8:00,NORMAL,20
Train5Agent=[PlatformA2,Tunnel1,Tunnel2,PlatformB2],8:00,NORMAL,10
Train6Agent=[PlatformA2,Tunnel4,PlatformB1],8:00,NORMAL,5
```

Example 1: Properly specified config.properties file

3.1.1 The Graph Representation

The graph representation previously mentioned, uses two directed graphs to specify possible paths when starting from a given station. The following arguments were passed in the format:

"platform/tunnel, connected to platform/tunnel, travel time in minutes"

with all connections separated by ";".

graphA=PlatformA1,Tunnel4,5; represents the information that there is a connection from PlatformA1 to Tunnel4 in graph A, and the traversal time is equal to 5 minutes. The construction of graph B follows a similar approach, with all edges illustrated in the schema in **Figure 1**.

3.1.2 Nodes' Passing Times

The subsequent argument, labeled "passingTimes", denotes the duration required to traverse each node of the graph. The data is presented in the following format:

"tunnel/platform, time in minutes, node type (e.g. WIDE), whether it is FAULTY, time it will be fixed (hour)"

The last three arguments are not mandatory. In the lack of specification of the WIDE argument, the node is interpreted as being of the NORMAL type. Instance describing the Tunnel3 node:

Tunnel3,5, ,FAULTY,10:30;

3.1.3 Agent Trains’ Arguments

The arguments to be passed to each agent instance must be in the following format:

"list of nodes in traversal order – preferable path to next station, start time of planned journey, train type (WIDE/NORMAL), maximum acceptable delay".

It is necessary to ensure that the name of the train agent is written in the config file in such a way that it begins with the word "Train" to enable proper file read.

Train2Agent=[PlatformB1,Tunnel4,PlatformA1],8:01,WIDE,15

3.2 JADE program launch

In order to allow the system to perform its functions, it is necessary that all start arguments are correctly specified and then passed to JADE (Java Agent DEvelopment Framework). Primarily, the arguments from the configuration file must be represented in a more complex manner, thus providing the essential knowledge for each agent. The manual execution of this task can pose significant challenges; therefore, a separate .java file is employed to ensure the proper formatting of arguments.

The *TrainTransitSystem.java* is responsible for preparing the arguments provided in the configuration file for each agent and then pass them into JADE. The transformation functions include the following:

- *transformNodesTimes*: transforming the "passingTimes" property into a map in which the node name serves as the key and the time is the value. Additionally, it extracts information regarding FAULTY or WIDE nodes.
- *transformTrainArgs*: transforming information regarding trains into a key-value map. The key of this map corresponds to the train name, while the value corresponds to the arguments.
- *generateAllPlannedPaths*: for each train, generate planned path with: *generatePlannedPath* function - according to the static plan of traversal between two stations and start time, arrival and departure times at each node are generated with regard to graph traversal times, values are saved into a map.
- *getNeighbors*: collecting nodes from the side of graph A and graph B. This function is useful for generating TunnelAgent arguments.
- *prepareJadeArgs*: converting all of the information that has been previously generated into a single string - used as the initial arguments of the JADE.
    - TrainTransitAgent receives information about node passing times, all planned paths of trains, and which tunnels are considered as WIDE.
    - Each TunnelAgent receives information regarding its neighboring nodes and the travel time required to access them from stations A and B, and additional information about status of node, indicating whether

it is FAULTY, with an expected time to repair, or FREE if the node can be occupied at any time.

- Each TrainAgent receives the information about its type (WIDE/NORMAL), the maximum acceptable delay, and a previously generated path with exact planned arrival and departure times at each tunnel or platform.

It should be noted that each instance of time in a configuration file was converted into minutes by the *timeToMinutes* function. Arguments that have been prepared in this way can be directly forwarded to JADE.

3.3 Program specifications

The system is governed by a **single transit system agent**, which possesses comprehensive knowledge of the original routes of each train, ensures the completion of all scheduled routes, and is responsible for generating new paths for the train agents. This agent is the only one in the system with the knowledge of the graph's topology, which includes information about all existing edges and nodes, as well as their specifications (e.g. wide tunnels or platforms) and traversal times.

Furthermore, the system is made up of **ten tunnel agents** that function as a shared resource, including four agents for tunnels, three agents for platforms at station A, and three agents for platforms at station B. This exceeds the number specified in the project proposal, as I had not initially expected to include agents for platforms at the stations.

In the exemplary scenario, there are **six train agents competing for access to resources.** Each agent is positioned on a designated platform at a specified time, as indicated by the start time argument. It is crucial to ensure that no agents remain on the same platform during an overlapping time interval. Therefore, the initial information provided in system configuration file arguments must be logical. The objective of each train is to reach the opposing station, preferably on the platform indicated in the original plan. However, if this is not possible, the train will be satisfied with any other platform at the opposing station. It is important to note that trains are agents that actually execute actions, rather than simply responding to the requests of other agents.

3.3.1 TransitSystemAgent.java

The primary function of the Transit System Agent is to manage requests from trains, generating and providing them with new paths derived from a graph. The agent exhibits two distinct behaviours:

1) *HandleIncomingMessages | Cyclic Behaviour*

Behaviour constantly monitoring incoming messages from trains. In response to these messages, the agent is expected to execute specific operations during runtime. The messages include requests for a path or indications that the path scheduling process has been completed by the train. The request for a path is resolved by following the following functions:

- *giveNewPathToTrain:* if it is the first time a specific train requests a new path, the transit system agent generates all possible paths (or up to 10, to avoid an inefficient search) using the Breadth-First Search (BFS)[4] algorithm, taking into account the station from which the train starts. Agent divides the paths into two categories: "direct" paths, which are those that directly reach the exact platform from the train's original path, and "alternative" paths, which are all the others. It filters out the original path that is currently being used and that has already been travelled. Then, the direct paths are added to the final set of paths, while the alternative ones are sorted according to their priority and also added to the final paths.
- *prioritizePaths:* based on calculated "score" of each path, sort them in descending order.
- *calculateScores:* calculating "score" for a given path to determine its value, higher scores are given to paths that originate or terminate at the same node as in the original plan, while lower scores are allocated to paths that require more time to traverse.
- *findAllPaths*: graph traversal algorithm is based on Breadth First Search. This is implemented with the use of the JGraphT Java library[5], and the graph properties *outgoingEdgesOf* and *getEdgeTarget*. It uses a simple queue and adds the neighbours of each node, filtering out nodes that have already been visited. In addition, the nodes are checked for constraints, such as their specific WIDE type. If a direct path to one of the end platforms in opposite station is found, it is added to the solution list and that is the list returned at the end.

Algorithms such as A* and Dijkstra are regarded as inadequate in this context. The program's design prioritizes the discovery solutions, rather than exclusively focusing on the shortest ones.

1) CheckSchedulingEnd | Behaviour

Behaviour designed to respond to any changes and to systematically verify whether all trains have completed their scheduled paths. If train reports the termination of scheduling, it is appended to the *completedTrains* map. If the size of the map is equivalent to the expected number of trains specified in the configuration file, the behaviour forces printing all of the final paths of the trains and marks itself as complete (*trainsScheduled = true*).

3.3.2 TunnelAgent.java

The second type of agent characterized by its external dependence on messages from train agents for the execution of actions is the tunnel agent. Each tunnel agent tracks its current reservations in a list named *nodeAvailability* and the reservations of the rails leading to it from the neighbor nodes in the *neighborRailAvailability* map. It uses *pendingRequests* map to monitor all requests from trains and attempts to give each of them an opportunity to confirm their reservations. A node is capable of processing (waiting for confirmation) for only a single train request at any given moment. The actions described above are all managed by a single behavior:

1) *HandleIncomingMessages | Cyclic Behaviour*

Behaviour involves the constant monitoring of incoming messages from trains or other nodes, updating of the node's state through the implementation of the boolean *waitForConfirmation* and tracking information regarding the currently confirming train. The actions undertaken by the agent are distinguished by the specific command specified within the message content:

RESERVE - request generated by train, handled by functions:

- *handleReservationRequest*: check validity of request, reply to requesting tunnel "OK"; or "NOT AVAILABLE" with the first time slot when it will be free. If the node is available in given arrival and departure time of the train, there is also a need to also check availability of the rail leading to the node in its reservation map.
    - If there exist a rail that is available the response should be "OK",
    - If there is no rail leading to a node that is available in given arrival and departure times, the response should be "NOT AVAILABLE" indicating the first possible arrival time at the node.

  After replying "OK", the sender of the request (i.e., the train) must be marked as the one for which final confirmation message is expected. Furthermore, node immediately mark the planned but not yet confirmed (temporary) reservations in its *neighborRailAvailability* map and the *nodeAvailability* list. Then notifies the neighbour nodes about this update via INFORM ADD message, that will be discussed later.

  If node is currently waiting for the confirmation from another train, all requests that should be replied with an "OK" are added to the *pendingRequests* map.
- *isAvailable:* checking if given arrival and departure time is overlapping with one of already scheduled occupations.
- *calculateWhenFree:* given the arrival and departure time, calculate first, next <u>time instant at which the node will be available.</u>
- *calculateArrivalRailTime:* used to check availability of a rail leading to a specific node within a specified time frame.
    - If train does not require waiting for a rail, the function returns 0.
    - Otherwise, function returns the <u>time when train should arrive at the connecting rail.</u>
- *findNextAvailableTime*: based on the planned time of arrival at the rail, find the first time instant when given rail will be available, whereas returned time should be later than planned arrival at rail.

INFORM ADD – receiving a request from other nodes to mark a given reservation in the node's *neighborRailAvailability* map, handled by single function *markReservationsAddInform.*

INFORM REMOVE – analogous to the previously mentioned command but it's purpose is to remove the reservations marked in the node's *neighborRailAvailability* map, handled by function *markReservationsRemoveInform.*

CONFIRM RESERVATION – command confirming that current reservations are up to date, invokes function:

- *resolveNextRequest:* reset values related to the current reservation (*waitingForConfirmation, confirmingTrain currentRequest*) and process next request from a *pendingRequests* map (if there is any), checking its validity and responding to it by calling again *handleReservationRequest* function.

CANCEL – command intended to cancel requests for reservations of trains that are still in the *pendingRequests* map or one of a current confirming train. This occurs when an "OK" message has been sent but a confirmation message from the train has not yet been received. If the train instead sends CANCEL message it is necessary to remove all temporary reservations and notify other nodes about this update via INFORM REMOVE message. The node suspends its operation using block (2000) to enable the other nodes to update their reservation maps. Thereafter, agent attempts to process the subsequent pending request by invoking the *resolveNextRequest* function.

3.3.3 TrainAgent.java

The objective of the train agent is to find a way to reserve time intervals on each node included in a path. The basic lifecycle of the train agent involves several steps resolved by:

1) *ReservePath | Behaviour*

During this behaviour agent keeps information regarding the current path, the original, planned path (the one provided as the argument), and the calculated delayed paths in a map. Agent also keeps track of requests sent to the nodes. The behaviour is composed of six steps in the reservation process:

Step = 0: sending request with occupation time interval to each node in current path

- *sendRequestForPath:* sending message, to each node in path, with command RESERVE, information from which station the train is going, the specific node at which it will depart, and the planned arrival and departure times. Each sent request is added to *sentRequests* list. Thereafter, train wait for the responses (Step 1).

Step = 1: waiting for tunnel responses with respect to some TIMEOUT

- *waitAndAnalyseTrainResponses*: the basic timeout for the tunnel responses is 5 seconds. During this timeout period, all responses are collected to the *requestResponses* map. If some of the incoming responses are marked with performative REFUSE train immediately marks it by setting *someRequestRejected = true.*

- *analyseTrainResponses:* according to nodes responses different actions might be taken by the train.

After the TIMEOUT:

1. If subset of nodes does not respond, treat them as not accepted. All requests previously sent to tunnels must be cancelled by *sendCANCELMessageToAllTunnels*. Then:

   1.1. If the current reservation is marked as the forced one, the reservation must be forced by selecting the least delayed path from the *delayedPaths* map as the current path (*chooseLeastDelayedPath*) and agent attempt to reserve it once again (Step 0).

   1.2. Otherwise (it is not a forced reservation) agent attempt to obtain the new path from the TransitSystemAgent (Step 3).

2. Otherwise (all of the nodes responded to request):

   2.1. If all nodes accepted its request, as indicated by "OK" messages, send confirmations to them (Step 2).

   2.2. Otherwise, *sendCANCELMessageToAllTunnels* and invoke *resolveNewDelayedPath* function which checks whether total delay of a path updated with new proposed arrival times in nodes is less than maximum acceptable delay. Add updated path to *delayedPaths* map with *totalDelay* as the key.

      2.2.1. If calculated total delay is ACCEPTABLE set delayed path as the current one and make an attempt to reserve it once again (Step 0).

      2.2.2. If delay is NOT ACCEPTABLE, come back to point 1.1., check again whether it is a forced reservation, then proceed to either Step 0 or Step 3.

sendCANCELMessageToAllTunnels: after invoking this function, use block(3000) command to pause the agent for a period of three seconds. This will enable all the nodes to update their reservation maps and notify their neighbours.

Step = 2: sending confirmation to all nodes in the current path

Send message with command **CONFIRM RESERVATION** to all the nodes and end process of reservation (Step 5).

Step = 3: send request to generate new path

With the use of *requestNewPath* function, request for a new path to the TransitSystemAgent, and then wait for its response (Step 4).

Step = 4: waiting for a new path

receiveNewPath function waits for a message from TransitSystemAgent, its content determine the subsequent course of action. The message may contain either the indication "NO PATHS" or the suggested new path. If there are no more paths available for the train, *chooseLeastDelayedPath* function will select the least delayed path as the current one and set *forcedReservation* = *true*. If a new path is received in the message, it will become the current one. In both scenarios, the train is required to redo the entire reservation process (Step 0).

Step = 5: final step, end of scheduling

With the use of *informTransitSystem* function, inform the TransitSystemAgent about success of reservation process by sending message with DONE command. Afterwards setting *currentPathConfirmed* = *true* marking the completion of this behaviour.

3.4 Program outputs

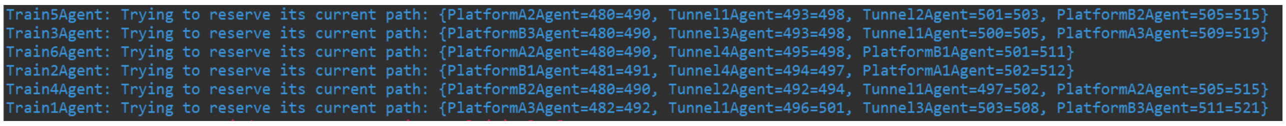
To differentiate between the outputs of the various agents, a color-coding system was employed. The following colors were assigned to each agent type:

- TransitSystemAgent: **YELLOW**
- TunnelAgents: **WHITE**
- TrainAgents: **CYAN**

Additionally, the **OK** messages and **confirmations** were designated **GREEN**, and the **NOT AVAILABLE** messages were labeled **RED**.

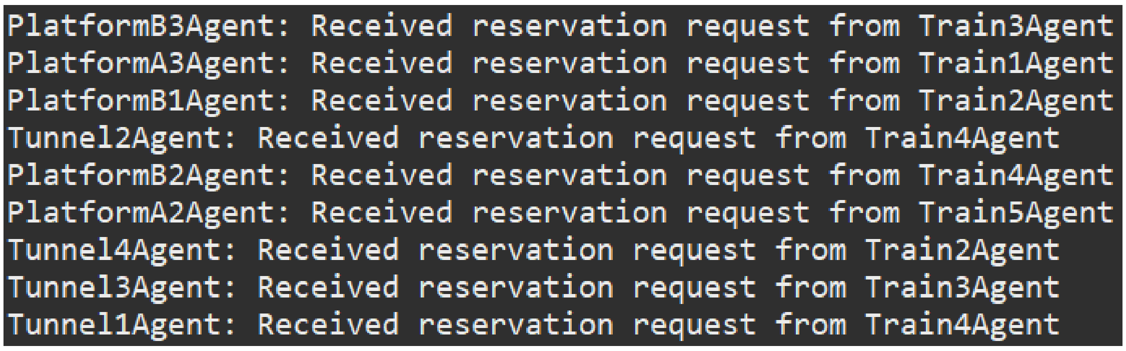
JADE Workflow:

All trains report their current action - reserving current path.

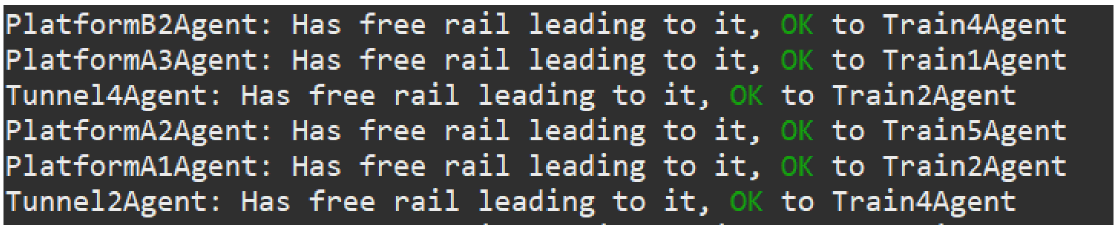


```
Train5Agent: Trying to reserve its current path: {PlatformA2Agent=480=490, Tunnel1Agent=493=498, Tunnel2Agent=501=503, PlatformB2Agent=505=515}
Train3Agent: Trying to reserve its current path: {PlatformB3Agent=480=490, Tunnel3Agent=493=498, Tunnel1Agent=500=505, PlatformA3Agent=509=519}
Train6Agent: Trying to reserve its current path: {PlatformA2Agent=480=490, Tunnel4Agent=495=498, PlatformB1Agent=501=511}
Train2Agent: Trying to reserve its current path: {PlatformB1Agent=481=491, Tunnel4Agent=494=497, PlatformA1Agent=502=512}
Train4Agent: Trying to reserve its current path: {PlatformB2Agent=480=490, Tunnel2Agent=492=494, Tunnel1Agent=497=502, PlatformA2Agent=505=515}
Train1Agent: Trying to reserve its current path: {PlatformA3Agent=482=492, Tunnel1Agent=496=501, Tunnel3Agent=503=508, PlatformB3Agent=511=521}
```

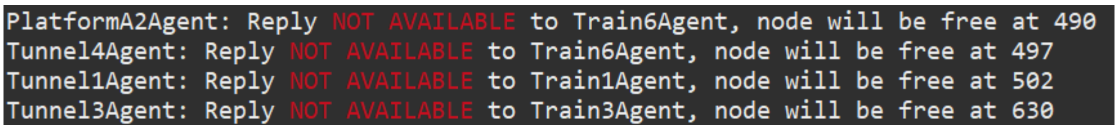
Nodes agents receives the reservation requests from agents, and then reply to it:



```
PlatformB3Agent: Received reservation request from Train3Agent
PlatformA3Agent: Received reservation request from Train1Agent
PlatformB1Agent: Received reservation request from Train2Agent
Tunnel2Agent: Received reservation request from Train4Agent
PlatformB2Agent: Received reservation request from Train4Agent
PlatformA2Agent: Received reservation request from Train5Agent
Tunnel4Agent: Received reservation request from Train2Agent
Tunnel3Agent: Received reservation request from Train3Agent
Tunnel1Agent: Received reservation request from Train4Agent
```

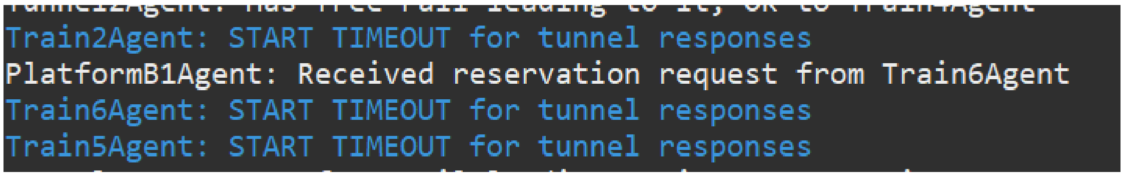


```
PlatformB2Agent: Has free rail leading to it, OK to Train4Agent
PlatformA3Agent: Has free rail leading to it, OK to Train1Agent
Tunnel4Agent: Has free rail leading to it, OK to Train2Agent
PlatformA2Agent: Has free rail leading to it, OK to Train5Agent
PlatformA1Agent: Has free rail leading to it, OK to Train2Agent
Tunnel2Agent: Has free rail leading to it, OK to Train4Agent
```

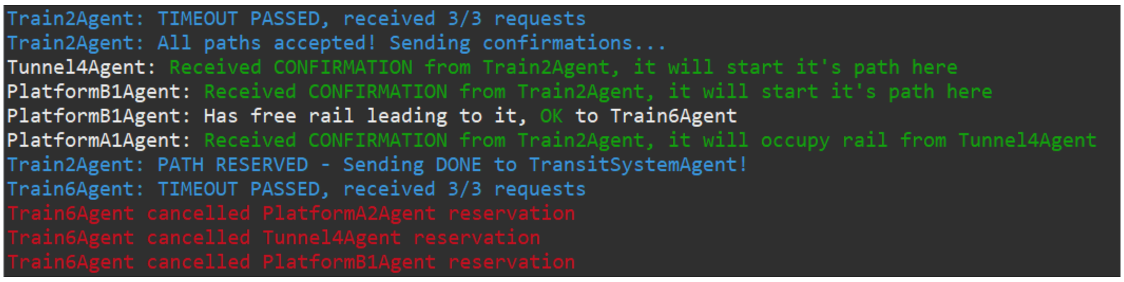


```
PlatformA2Agent: Reply NOT AVAILABLE to Train6Agent, node will be free at 490
Tunnel4Agent: Reply NOT AVAILABLE to Train6Agent, node will be free at 497
Tunnel1Agent: Reply NOT AVAILABLE to Train1Agent, node will be free at 502
Tunnel3Agent: Reply NOT AVAILABLE to Train3Agent, node will be free at 630
```

Trains start timeout for the responses.



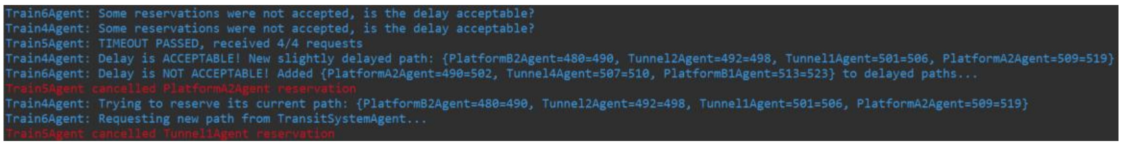
```
Train2Agent: START TIMEOUT for tunnel responses
PlatformB1Agent: Received reservation request from Train6Agent
Train6Agent: START TIMEOUT for tunnel responses
Train5Agent: START TIMEOUT for tunnel responses
```



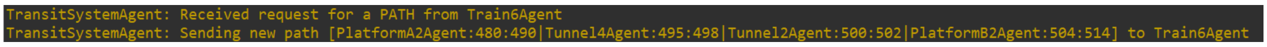
```
Train2Agent: TIMEOUT PASSED, received 3/3 requests
Train2Agent: All paths accepted! Sending confirmations...
Tunnel4Agent: Received CONFIRMATION from Train2Agent, it will start it's path here
PlatformB1Agent: Received CONFIRMATION from Train2Agent, it will start it's path here
PlatformB1Agent: Has free rail leading to it, OK to Train6Agent
PlatformA1Agent: Received CONFIRMATION from Train2Agent, it will occupy rail from Tunnel4Agent
Train2Agent: PATH RESERVED - Sending DONE to TransitSystemAgent!
Train6Agent: TIMEOUT PASSED, received 3/3 requests
Train6Agent cancelled PlatformA2Agent reservation
Train6Agent cancelled Tunnel4Agent reservation
Train6Agent cancelled PlatformB1Agent reservation
```

Timeout passed i.e. Train2Agent received all requests and confirmed its reservation.

Train6Agent cancelled its reservation due to the fact that some reservations were not accepted. Then Train6Agent asks for a new path to TransitSystemAgent:

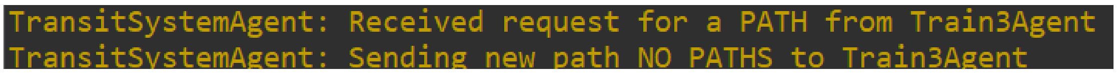


```
Train6Agent: Some reservations were not accepted, is the delay acceptable?
Train4Agent: Some reservations were not accepted, is the delay acceptable?
Train5Agent: TIMEOUT PASSED, received 4/4 requests
Train4Agent: Delay is ACCEPTABLE! New slightly delayed path: {PlatformB2Agent=480=490, Tunnel2Agent=492=498, Tunnel1Agent=501=506, PlatformA2Agent=509=519}
Train6Agent: Delay is NOT ACCEPTABLE! Added {PlatformA2Agent=490=502, Tunnel4Agent=507=510, PlatformB1Agent=513=523} to delayed paths...
Train5Agent cancelled PlatformA2Agent reservation
Train4Agent: Trying to reserve its current path: {PlatformB2Agent=480=490, Tunnel2Agent=492=498, Tunnel1Agent=501=506, PlatformA2Agent=509=519}
Train6Agent: Requesting new path from TransitSystemAgent...
Train5Agent cancelled Tunnel1Agent reservation
```



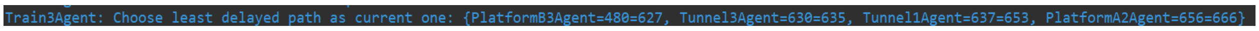
```
TransitSystemAgent: Received request for a PATH from Train6Agent
TransitSystemAgent: Sending new path [PlatformA2Agent:480:490|Tunnel4Agent:495:498|Tunnel2Agent:500:502|PlatformB2Agent:504:514] to Train6Agent
```

The process is repeated until a reasonable path is found, or until no other paths are available:



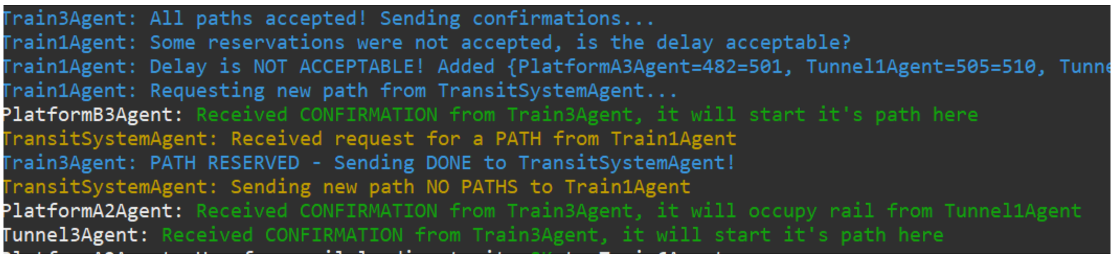
```
TransitSystemAgent: Received request for a PATH from Train3Agent
TransitSystemAgent: Sending new path NO PATHS to Train3Agent
```

Therefore, Train3Agent has selected the path with the least delay as the current one:



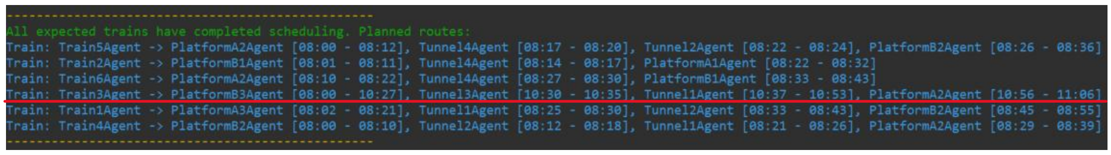
```
Train3Agent: Choose least delayed path as current one: {PlatformB3Agent=480=627, Tunnel3Agent=630=635, Tunnel1Agent=637=653, PlatformA2Agent=656=666}
```

Over time, it will identify the appropriate path and confirm it to the nodes.



```
Train3Agent: All paths accepted! Sending confirmations...
Train1Agent: Some reservations were not accepted, is the delay acceptable?
Train1Agent: Delay is NOT ACCEPTABLE! Added {PlatformA3Agent=482=501, Tunnel1Agent=505=510, Tunne
Train1Agent: Requesting new path from TransitSystemAgent...
PlatformB3Agent: Received CONFIRMATION from Train3Agent, it will start it's path here
TransitSystemAgent: Received request for a PATH from Train1Agent
Train3Agent: PATH RESERVED - Sending DONE to TransitSystemAgent!
TransitSystemAgent: Sending new path NO PATHS to Train1Agent
PlatformA2Agent: Received CONFIRMATION from Train3Agent, it will occupy rail from Tunnel1Agent
Tunnel3Agent: Received CONFIRMATION from Train3Agent, it will start it's path here
```

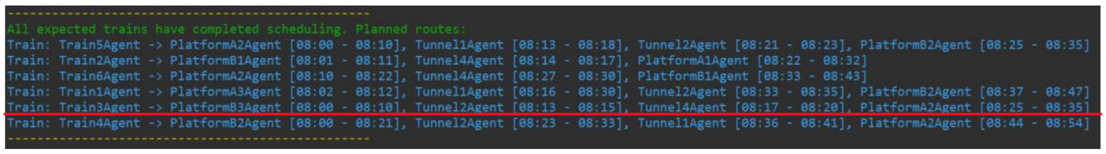
Train3Agent is worthy of consideration. An analysis of the schema presented in **Figure 1.** reveals that, from the platform B3 (the starting point for the agent), there is no alternative route than through the faulty Tunnel3. This route remains inoperative until 10:30 as indicated in the arguments what can be seen on the printed solution of the program:



```
--------------------------------------------------
All expected trains have completed scheduling. Planned routes:
Train: Train5Agent -> PlatformA2Agent [08:00 - 08:12], Tunnel4Agent [08:17 - 08:20], Tunnel2Agent [08:22 - 08:24], PlatformB2Agent [08:26 - 08:36]
Train: Train2Agent -> PlatformB1Agent [08:01 - 08:11], Tunnel4Agent [08:14 - 08:17], PlatformA1Agent [08:22 - 08:32]
Train: Train6Agent -> PlatformA2Agent [08:10 - 08:22], Tunnel4Agent [08:27 - 08:30], PlatformB1Agent [08:33 - 08:43]
Train: Train3Agent -> PlatformB3Agent [08:00 - 10:27], Tunnel3Agent [10:30 - 10:35], Tunnel1Agent [10:37 - 10:53], PlatformA2Agent [10:56 - 11:06]
Train: Train1Agent -> PlatformA3Agent [08:02 - 08:21], Tunnel1Agent [08:25 - 08:30], Tunnel2Agent [08:33 - 08:43], PlatformB2Agent [08:45 - 08:55]
Train: Train4Agent -> PlatformB2Agent [08:00 - 08:10], Tunnel2Agent [08:12 - 08:18], Tunnel1Agent [08:21 - 08:26], PlatformA2Agent [08:29 - 08:39]
--------------------------------------------------
```

Plan 1. Scheduled plan for all trains, Train3Agent waits for Tunnel3 to be repaired.

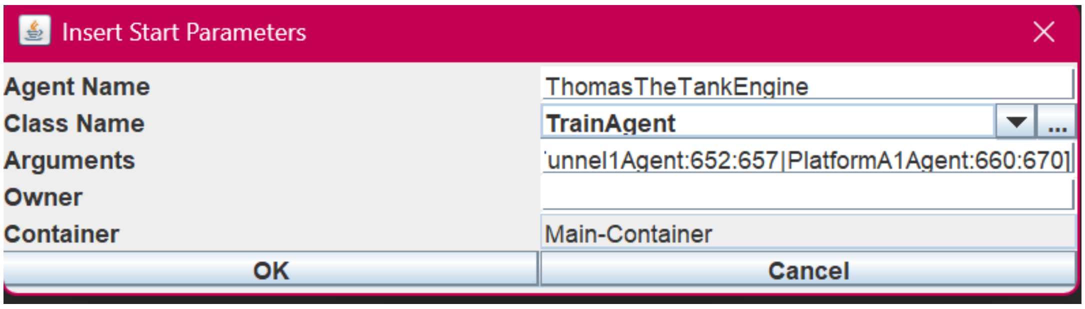
The addition of a single edge `PlatformB3,Tunnel2,3;` that connects PlatformB3 to Tunnel2 in the configuration file in **Example 1,** enables train to use other edge in traversal to opposite station. It results in a more efficient solution for the mentioned Train3Agent:



```
--------------------------------------------------
All expected trains have completed scheduling. Planned routes:
Train: Train5Agent -> PlatformA2Agent [08:00 - 08:10], Tunnel1Agent [08:13 - 08:18], Tunnel2Agent [08:21 - 08:23], PlatformB2Agent [08:25 - 08:35]
Train: Train2Agent -> PlatformB1Agent [08:01 - 08:11], Tunnel4Agent [08:14 - 08:17], PlatformA1Agent [08:22 - 08:32]
Train: Train6Agent -> PlatformA2Agent [08:10 - 08:22], Tunnel4Agent [08:27 - 08:30], PlatformB1Agent [08:33 - 08:43]
Train: Train1Agent -> PlatformA3Agent [08:02 - 08:12], Tunnel1Agent [08:16 - 08:30], Tunnel2Agent [08:33 - 08:35], PlatformB2Agent [08:37 - 08:47]
Train: Train3Agent -> PlatformB3Agent [08:00 - 08:10], Tunnel2Agent [08:13 - 08:15], Tunnel4Agent [08:17 - 08:20], PlatformA2Agent [08:25 - 08:35]
Train: Train4Agent -> PlatformB2Agent [08:00 - 08:21], Tunnel2Agent [08:23 - 08:33], Tunnel1Agent [08:36 - 08:41], PlatformA2Agent [08:44 - 08:54]
--------------------------------------------------
```

Plan 2. Scheduled plan for all trains, Train3Agent do not have to wait for Tunnel3 to be repaired.

As the result program shows schedule of all planned routes for expected trains in the system. Additionally, a possibility to schedule more trains with respect to current plan is available by creating a new TrainAgent:



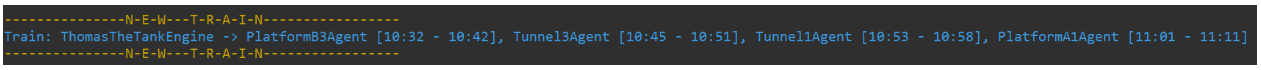
Insert Start Parameters

Agent Name	ThomasTheTankEngine
Class Name	TrainAgent
Arguments	'unnel1Agent:652:657|PlatformA1Agent:660:670]
Owner	
Container	Main-Container

Arguments:

NORMAL,5,[PlatformB3Agent:632:642|Tunnel3Agent:645:650

|Tunnel1Agent:652:657|PlatformA1Agent:660:670]



```
--------------N-E-W---T-R-A-I-N----------------
Train: ThomasTheTankEngine -> PlatformB3Agent [10:32 - 10:42], Tunnel3Agent [10:45 - 10:51], Tunnel1Agent [10:53 - 10:58], PlatformA1Agent [11:01 - 11:11]
--------------N-E-W---T-R-A-I-N----------------
```

The result is the creation of a plan for a train that fits into the currently occupied nodes. In this scenario ThomasTheTankEngine needs to wait for the Tunnel1 as it was already occupied by Train3Agent, according to **Plan 1.**

3.5 Possible Improvements and System Challenges

Expanding the system to include more stations, such as an entire city transit network, and creating separate tunnel graphs for station connections would improve its applicability to real-world rail systems. The system can also provide additional features, such as allowing the reallocation of confirmed reservations to higher-priority trains (e.g. those experiencing delays), according to [1] stealing the reservations of other trains, up to a certain time before the train's scheduled departure. Additionally, the current system assumes that there is only one rail between any two nodes. This may not be the case in reality. The introduction of some of the modifications has the potential to make the system applicable to real-world scenarios within actual rail transit systems.

One of the biggest challenges faced during system development was handling message timeouts. Ideally, tunnels should operate independently, without waiting for train responses, and reservation requests should expire after a set time. However, this approach led to cyclic dependencies—where tunnels waited for responses from trains that were, in turn, waiting for responses from other tunnels—resulting in deadlocks. Due to these issues, I had to abandon this approach. The current implementation instead makes the tunnel or platform status dependent on train responses. If the train does not cancel or does not confirm the reservation, the node will wait forever.

References

[1] Daniela Briola, Viviana Mascardi, Maurizio Martelli, Riccardo Caccia, Carlo Milani: Dynamic Resource Allocation in a MAS: A Case Study from the Industry. WOA 2009: 125-1.

[2] Y. Liu and Y. Mohamed, "Multi-Agent Resource Allocation (MARA) for modeling construction processes," *2008 Winter Simulation Conference*, Miami, FL, USA, 2008, pp. 2361-2369

[3] Java Properties Class https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html

[4] S. Even, *Graph Algorithms*. Rockville, MD: Computer Science Press, 1979.

[5] JGraphT, Java library of graph theory data structures and algorithms https://jgrapht.org/guide/UserOverview

[6] G. Caire, JADE Tutorial, JADE Programming Tutorial for Beginners, 2009 https://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

[7] FIPA ACL Message Structure Specification http://www.fipa.org/specs/fipa00061/SC00061G.pdf

[8] FIPA Communicative Act Library Specification http://www.fipa.org/specs/fipa00037/SC00037J.html