

## **40730 HPC Project**

### **Traffic simulation on a computing grid, using a Cellular Automata approach**

Student: Sean Ryan - Student ID: 02585944

Due date: Friday 30<sup>th</sup> March 2012

## Table of Contents

40730 HPC Project .....	1
Traffic simulation on a computing grid, using a Cellular Automata approach .....	1
Traffic Simulation: Modelling Approach .....	3
Requirement: Distribution of constant work size over a varying number of processes.....	3
Solution: Dynamic distribution of work, using a broker and a peer-to-peer transfer of working regions.....	3
Sequence diagram describing the distributed algorithm for traffic simulation .....	7
Sequence diagram describing the algorithm for peer-to-peer re-allocation of work from an exiting processor process .....	8
Traffic Simulator - design.....	9
Traffic Simulator - components spanning multiple processes.....	9
Traffic Simulator - components within one process .....	10
Implementation.....	11
Requirements for the implementation.....	11
Assumptions .....	11
Alternatives Considered .....	11
Future Considerations .....	13
Performance .....	13
Scalability .....	13
Reliability .....	13
Extending the model .....	14
Improving the monitor .....	14
Cloud Deployment .....	14
Conclusion .....	15
Appendix: Running the simulation .....	16
Installation .....	16
Running the simulation .....	16

## **Traffic Simulation: Modelling Approach**

A cellular automata approach is taken to simulating varying traffic conditions. The traffic is simulated as a set of discrete cells. The model is run one step at a time, and at each step, every cell in the model is processed for one unit of time.

### **Requirement: Distribution of constant work size over a varying number of processes**

The work is distributed dynamically amongst the available processes.

The model has a set of regions, and the size of the model stays constant, since we are interested in modelling the traffic of a particular geographic area (in this case, the Republic of Ireland).

Because the total size of work remains constant, it was necessary to develop an algorithm that could re-distribute portions of work amongst processors, as processors become available.

### **Solution: Dynamic distribution of work, using a broker and a peer-to-peer transfer of working regions**

Each time a new processor joins the system, it contacts the broker, to discover the current set of processors, and to take a new unique processor identifier.

The processor calculates the average number of regions that each processor in the system should have, in order to have a fair distribution of work.

The processor then contacts each existing peer process in turn, asking it for a suitable number of regions of work. Similarly, when a processor leaves the system, it returns its regions of work to the other processors.

In this way, the entire region of work is continually processed, even though the number of available processors changes over time.

### Processes

The following distinct kinds of process were identified, in order to distribute the processing of the traffic model:

Process type	Quantity	Description
nameserver	1	The PyRo nameserver which facilitates locating another process by a well known name.
broker	1	A custom broker which provides information about the computing grid to a new processor, and also allocates a new ID to the new processor
processor	1 to many	<p>A processor process, which works on a portion of the traffic model, in order to simulate traffic.</p> <p>The processor process updates a shared model when local traffic results are available.</p> <p>The processor process communicates with its peer processors, whenever a peer is joining or leaving the computing grid, in order to re-distribute the work.</p>
monitor	1	<p>The monitor is a process to provide the user with an overall view of the traffic model.</p> <p>This process is useful for diagnostics, and also to view the instant and on-going results of the traffic simulation, without having to wait a long time for the simulation to run.</p>

## *Sequence Diagrams*

The following sequence diagrams describe the peer-to-peer algorithm of assigning units of work, and of running the traffic model.

### *Traffic simulation UML sequence diagrams*

It was found very useful, to use a UML sequence diagram to describe the flow of messages (or protocol) when a processor is requesting work from its peers.

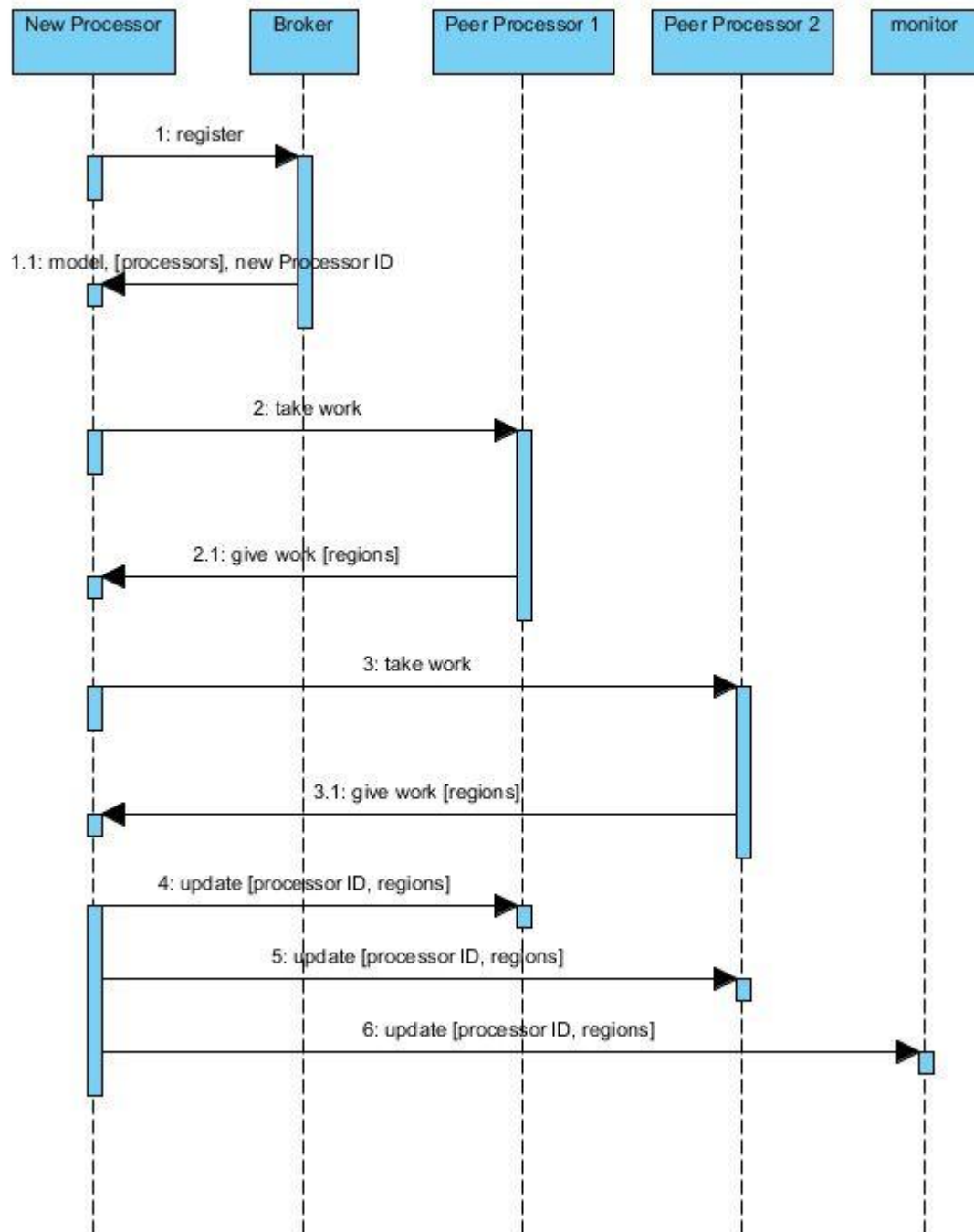
The UML sequence diagram makes the protocol clear to read, and it is easy to adjust the protocol as part of a diagram.

When the sequence diagram is correct, then implementing the protocol is mostly a matter of following the steps of the UML sequence diagram.

note: the term 'processor' is used to describe a process that performs work on the traffic model.

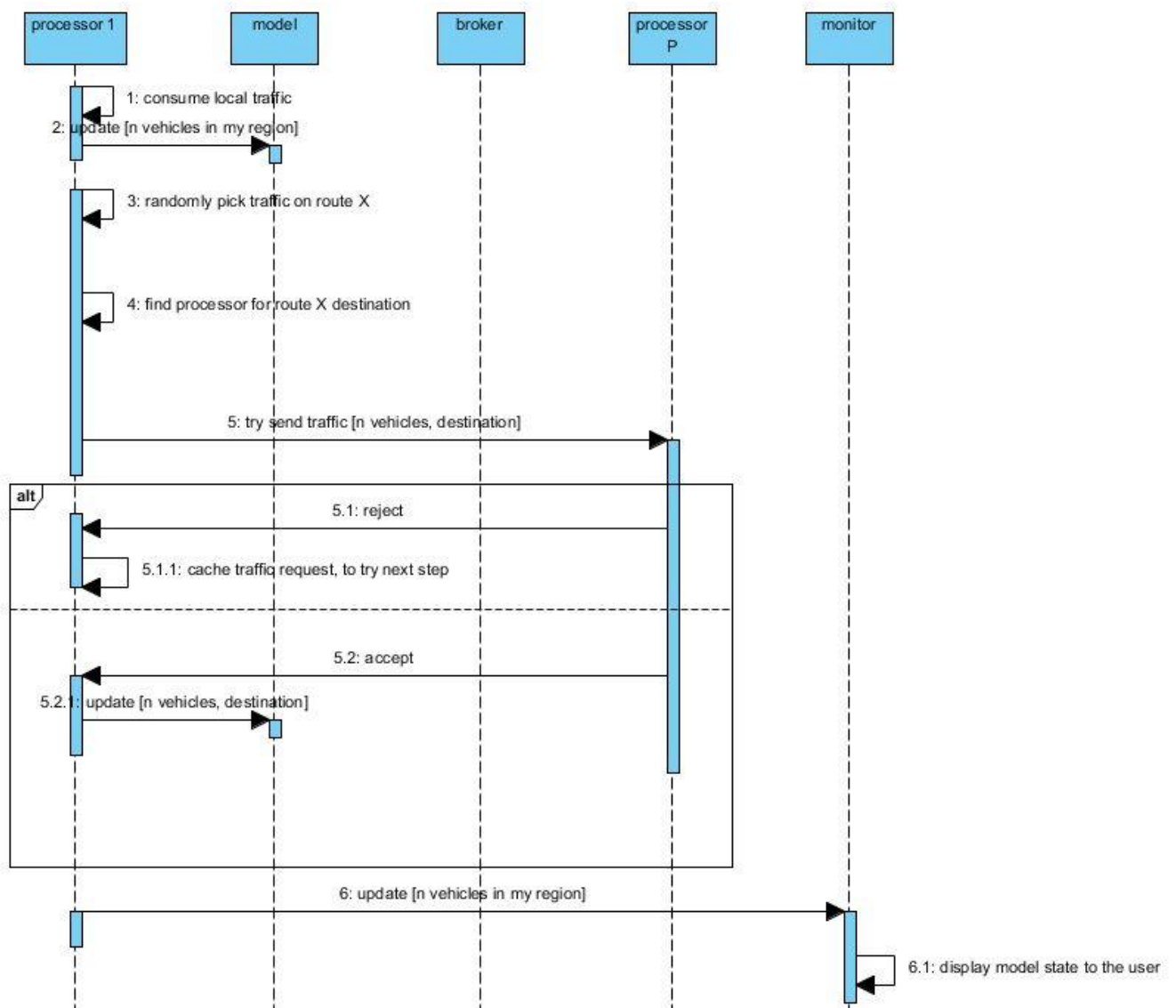
## Sequence diagram describing the algorithm for peer-to-peer distribution of work to a new processor process

Traffic Simulation Sequence Diagram - Peer to peer distribution of work



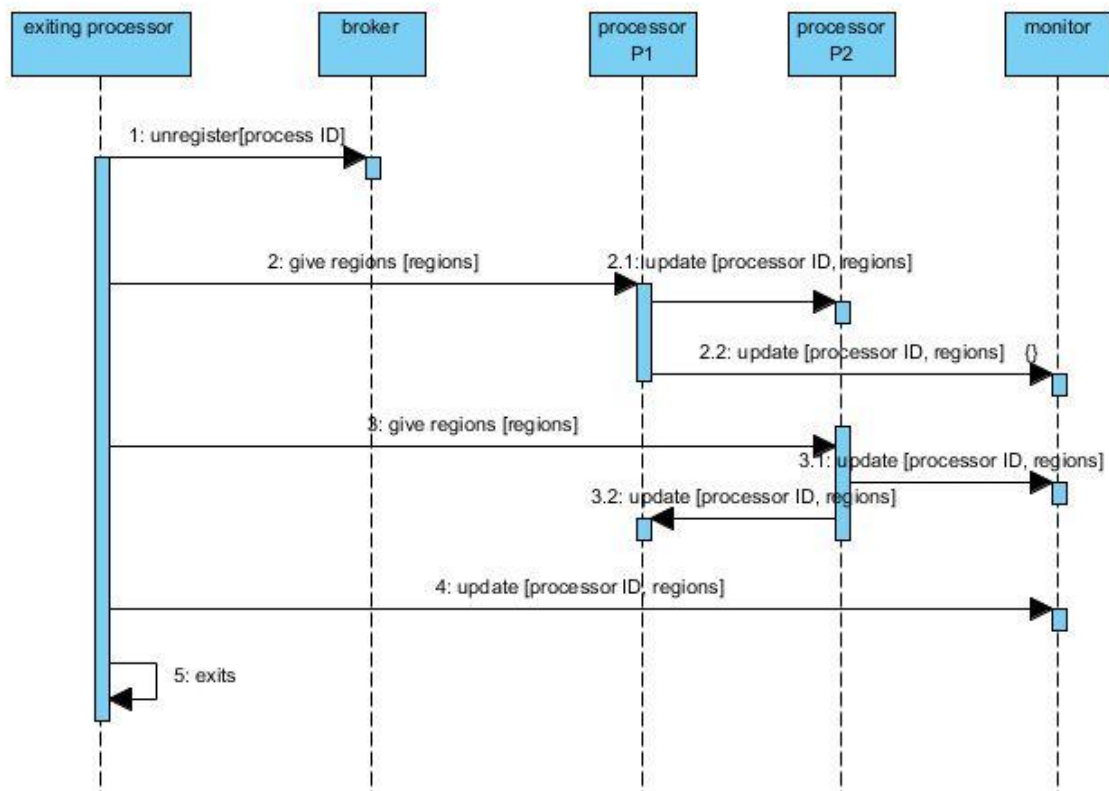
## Sequence diagram describing the distributed algorithm for traffic simulation

HPC Traffic Simulation - Traffic simulation algorithm



## Sequence diagram describing the algorithm for peer-to-peer re-allocation of work from an exiting processor process

Traffic Simulation - processor peer leaving the grid





## Traffic Simulator - design

This section describes the high level design of the traffic simulator.

A cellular automata approach was followed to simulate traffic flow between geographic areas, and at a lower level, between the junctions and motorways of each area.

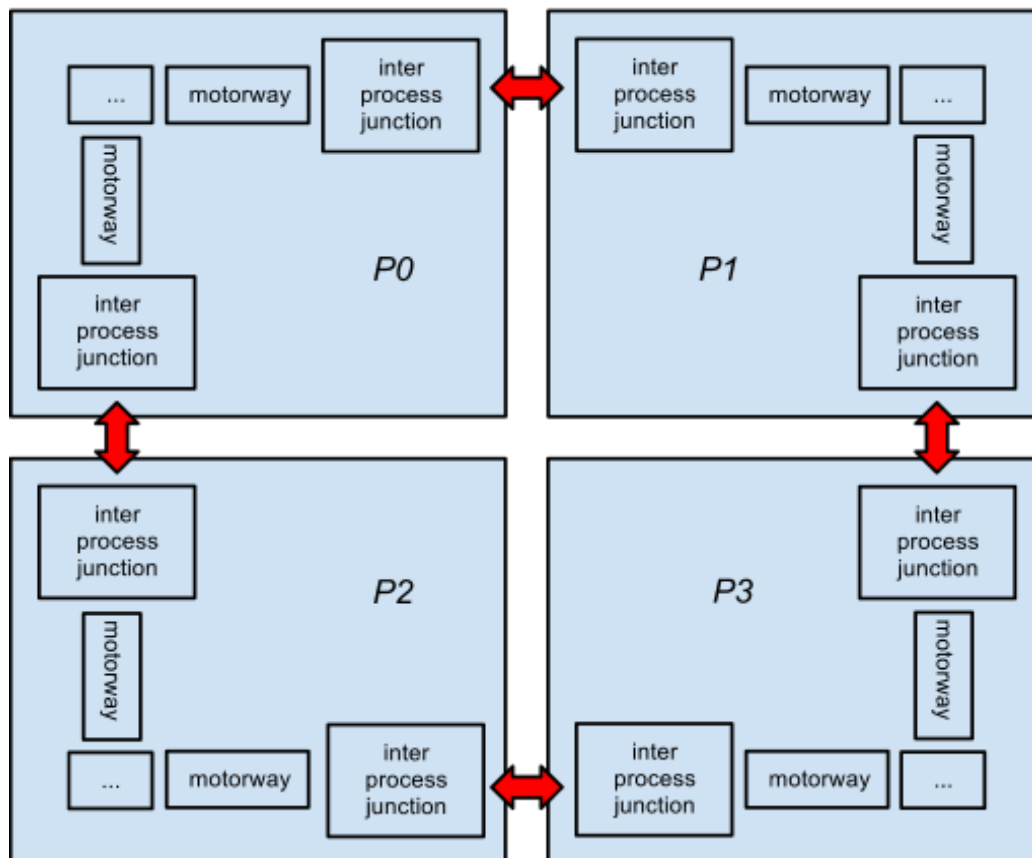
Each unit of the model is considered as a cell that is processed at each step of the model run.

This simple approach allows for a model that can be grown as needed, and can be enhanced to simulate at a lower and lower level, depending on the detail required.

## Traffic Simulator - components spanning multiple processes

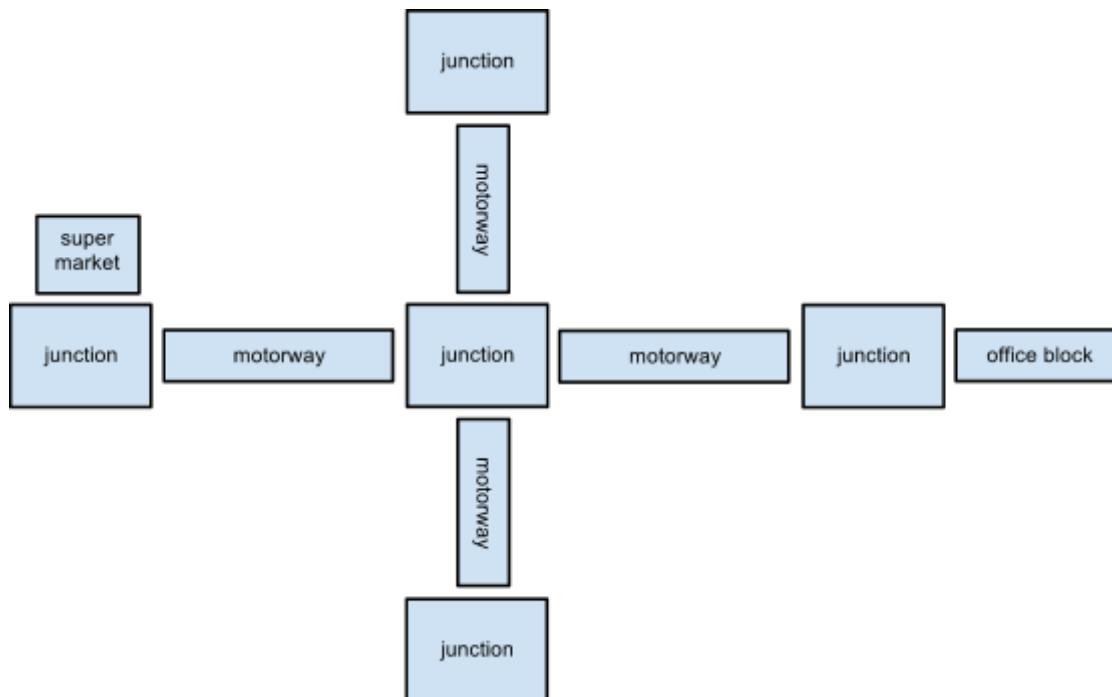
This diagram shows the cells that communicate between processes, at the process boundaries.

### Inter process architecture



## Traffic Simulator - components within one process

This diagram shows a lower level of the model, where cells communicate on one process, simulating a more detailed view of the traffic model.



# Implementation

## Requirements for the implementation

The implementation must fulfill the following requirements:

- OS neutral - the model should be able to run on a hybrid of operating systems, so that it can be distributed on more than one machine
- free of charge software - not tied to licensing - in order to keep costs down, no dependency should be made on OS licenses or other software licences, so that the model can scale cheaply
- rapid application development - the model should be developed in a timely manner
- distributed messaging - the technology must provide a good distributed messaging platform
- high level communication - the technology must provide a high conceptual level of communication, due to the complexity of the work distribution algorithm, and due to the requirement to scale the detail of model, adding more low level cells as needed.
- high level modelling - the technology must allow for high-level modelling of the simulation, to facilitate implementation of the simulation and of the work distribution algorithm

## Assumptions

- Data loss is not critical.

Data loss is not critical, as we are simulating flows of traffic - we are not attempting to solve an exact mathematical problem. We assume that the 'fuzzy' approach and slightly unpredictable outcome typical of cellular automata is suited to the traffic simulation domain.

## Alternatives Considered

### *C++ and open MPI*

#### advantages:

- performance
- cross OS is possible, with careful coding

#### disadvantages:

- low level modelling
- low level message platform

reference: <http://www.open-mpi.org/>

### *.NET and WCF*

#### advantages:

- good performance
- good high level modelling

#### disadvantages:

- WCF is a very large technology space to learn
- dependency on Windows OS which could be expensive to scale

reference: <http://msdn.microsoft.com/en-us/netframework/aa663324>

### *Python and zeromq implementation*

#### **advantages:**

- good performance
- good high level modelling with Python

#### **disadvantages:**

- messaging is low level, not really Object Oriented
- idiosyncratic platform, based on sockets rather than an object oriented or message queue based design

#### **reference:**

<http://www.zeromq.org>

### *Python and pyro (Python Remote Objects)*

#### **advantages:**

- good high level modelling with Python
- object oriented approach to remote communication
- simplicity: rapid development is possible
- OS neutral

#### **disadvantages:**

- performance might not be as good as other alternatives
- in practice, Pyro appears to be very sensitive to differences in OS or in version of Pyro.

reference: <http://irmen.home.xs4all.nl/pyro/>

### *Chosen implementation: Python and pyro (Python Remote Objects)*

The option to use Python and PyRo was chosen, as it facilitated a high level design, and also is quick to develop and to deploy.

#### **A reduced design was implemented**

Note: in order to have a working solution, a stripped down version of the design was implemented, where each process had a single cell region, rather than sub cells for motorways and junctions etc. Note that the object oriented implementation means that each region is encapsulated, and so it would be possible to later add in lower level cells such as motorways and supermarkets etc., as the need arises.

## Future Considerations

The implementation was for demonstration purposes, and there is room for improvement and also for extending the model.

The main areas for future consideration, would be performance, scalability, and extending the model. A set of area for future consideration follows.

### Performance

The performance of the model is limited by the following characteristics of the implementation:

- dependency on one broker

The system has a single broker, which all processes must contact, when joining or leaving the system. This could potentially be a bottleneck, if there is a high turnover of processes

- single shared model

There is a single shared model, which because we are using the PyRo platform, implies that a single process is hosting the model for the entire simulation grid (the broker process). ALL peers communicate with this process when updating or reading the model, and so it is obviously a potential bottleneck if the number of peers is large.

- The design requires ALL peers to communicate with the joining or leaving peer

Each time a process joins or leaves the system, it communicates with ALL of its peers. This would result in a lot of unproductive communication and processing of work distribution, when the number of peers is high.

A better approach would be to only communicate with nearer neighbours. Perhaps having several brokers or 'chief' peers that govern one part of the work area, would mean that joining or leaving processes only need communicate with their peers in that sub-region.

We would require an election algorithm, such as Google Paxos, in order to decide which peer becomes chief of a sub region of the model.

### Scalability

Scalability considerations are very similar to the considerations presented for Performance. In particular, having a single process host the single model, does not scale well. A more scalable implementation would be to partition the mode in a similar way to how work is partitioned by regions.

An alternative approach is to not share the mode, and instead copy the whole model to each new processor as it starts up. The processor would then only communicate between processes that are processing neighbouring regions. This would result in less communication overhead, and so would scale better.

### Reliability

The single model process and the single broker process are single points of failure, which means the system is not particularly robust.

One alternative would be to have a failover model process and broker, that could switch in if the main process fails.

### **Extending the model**

The traffic model could be further extended, adding more detailed and lower level cells. More regions can easily be added, to extend the geographical area that is simulated, and to simulated at a greater scale.

A time server could be added to the system, to signal traffic events, such as rush hour or a natural disaster etc.

### **Improving the monitor**

The monitor process could be enhanced, to provide a richer view of the model.

Because the model represents real geographic regions, with accurate geographic locations, then it would be possible to make the monitor a web service or web server, which would provide a Google maps view of the model, with overlays feeding in from processors as they update traffic conditions.

### **Cloud Deployment**

The current implementation uses the Python language and PyRo, which both are easy to install and run on Ubuntu. It should be possible to deploy and run the processors on Amazon EC2 cloud instances, which means that a great many nodes could be added to the traffic simulation system.

However in practice it may not be straightforward to use PyRo on EC2 and this would require further experimentation.

## Conclusion

The chosen approach of Cellular Automata was very simple and this helped greatly when implementing the distributed peer-to-peer traffic simulation.

Distributed systems are complex, and keeping things simple helps develop a system that works and is implemented in a reasonable timeframe.

Using UML sequence diagrams helped to design a successful peer-to-peer work distribution algorithm, and also helped as a working protocol document when implementing the system.

The implementation approach of using Python and Pyro worked reasonably well, as it helped rapid development in the limited time available.

Deployment issues occurred with Pyro, with problems when the version the Ubuntu OS was not identical on all nodes, and when versions of Pyro itself differed. This does raise questions as to whether Pyro is a robust enough solution for large scale or industrial deployment.

Perhaps a technology based on message queues would be more reliable as the components are less tightly coupled, and so process failure and recovery is possible without disruption to other processes, and also OS version and technology versions may be less problematic (for example Windows and MSMQ works quite well even when the versions are slightly different).

The issue with a message queuing implementation, is that it would be difficult to negotiate the re-distribution of work amongst peers.

However, when one considers that object oriented method calls are conceptually messages being sent between objects, then it should be possible to take the existing object oriented style sequential diagram, and apply it to a message queuing implementation.

Message queues would also decouple the processes, which allows for a more robust system, as for example if a message is sent on a queue to a receiving process, and the receiving process dies and later recovers, the message is not lost, and the sending process is not affected.

## Appendix: Running the simulation

### Installation

Pyro appears to have issues communicating with other versions of Pyro. For example, some versions of Pyro have different communication protocols. Also there are cases where run-time errors occur when running the simulation with differing version of Pyro.

Also there appear to be issues when using later versions of Ubuntu, where the nameserver cannot be located by the remote machine (whether client is run on Windows or on Ubuntu).

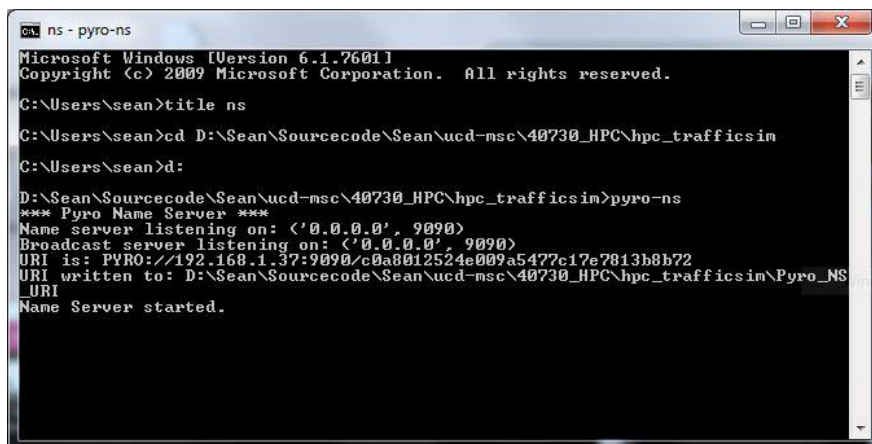
It was found that in practice PyRo only worked well when the following OS version and PyRo versions were used:

- Windows 7 Home Premium (hosting the nameserver, broker and monitor as well as the processors)
- Ubuntu 10.10 (hosting the processors)
- Python 2.7.2
- Pyro 3.9.10

### Running the simulation

For convenience, windows were titled according to the process which they run.

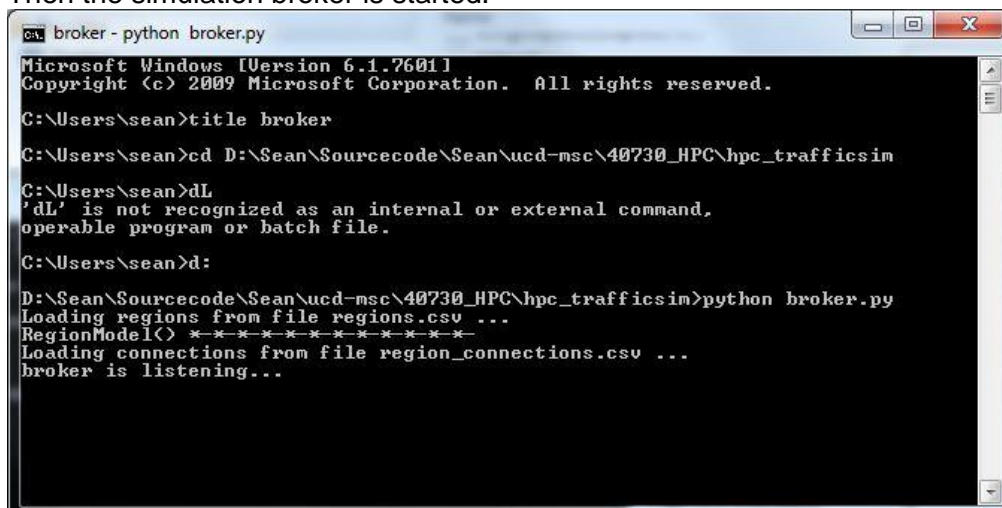
First the pyro nameserver is started on Windows 7:



```
ca. ns - pyro-ns
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\sean>title ns
C:\Users\sean>cd D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficin
C:\Users\sean>d:
D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficin>pyro-ns
*** Pyro Name Server ***
Name server listening on: ('0.0.0.0', 9090)
Broadcast server listening on: ('0.0.0.0', 9090)
URI is: PYRO://192.168.1.37:9090/c0a8012524e009a5477c17e7813b8b72
URI written to: D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficin\Pyro_NS
URI
Name Server started.
```

Then the simulation broker is started:



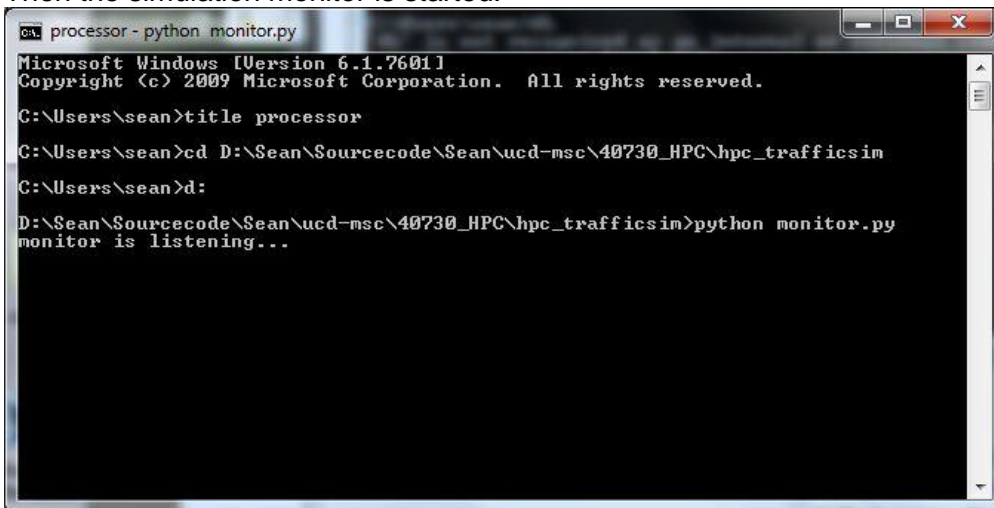
```
ca. broker - python broker.py
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\sean>title broker
C:\Users\sean>cd D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficin
C:\Users\sean>d:
'dL' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\sean>d:
D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficin>python broker.py
Loading regions from file regions.csv ...
RegionModel() *-*-*-*-*
Loading connections from file region_connections.csv ...
broker is listening...
```



Then the simulation monitor is started:

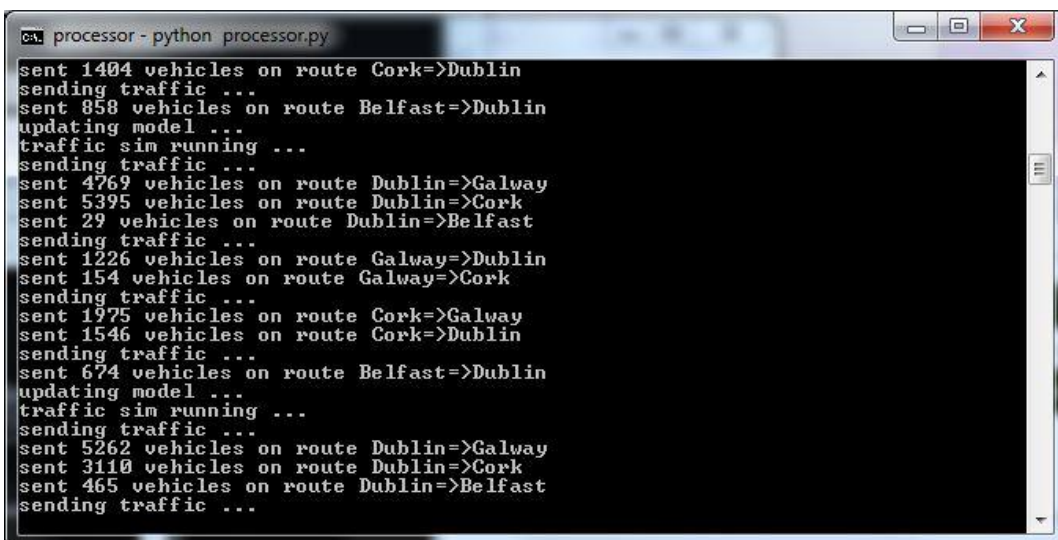


```
ca. processor - python monitor.py
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\sean>title processor
C:\Users\sean>cd D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficsim
C:\Users\sean>d:
D:\Sean\Sourcecode\Sean\ucd-msc\40730_HPC\hpc_trafficsim>python monitor.py
monitor is listening...
```

The first processor is then started.

Because this is the first processor, it takes ALL of the available work, by taking ALL of the work regions.



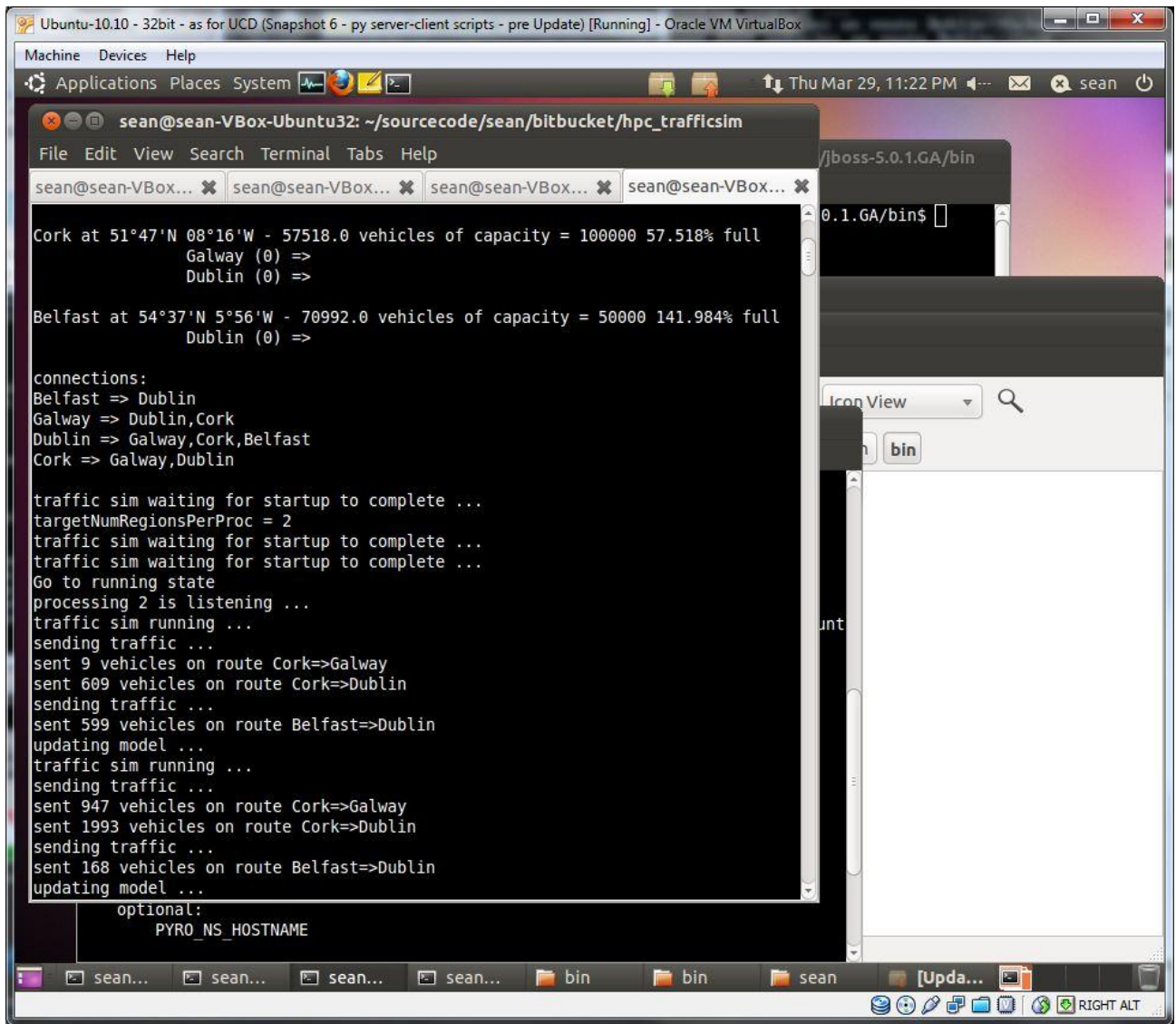
```
ca. processor - python processor.py
sent 1404 vehicles on route Cork=>Dublin
sending traffic ...
sent 858 vehicles on route Belfast=>Dublin
updating model ...
traffic sim running ...
sending traffic ...
sent 4769 vehicles on route Dublin=>Galway
sent 5395 vehicles on route Dublin=>Cork
sent 29 vehicles on route Dublin=>Belfast
sending traffic ...
sent 1226 vehicles on route Galway=>Dublin
sent 154 vehicles on route Galway=>Cork
sending traffic ...
sent 1975 vehicles on route Cork=>Galway
sent 1546 vehicles on route Cork=>Dublin
sending traffic ...
sent 674 vehicles on route Belfast=>Dublin
updating model ...
traffic sim running ...
sending traffic ...
sent 5262 vehicles on route Dublin=>Galway
sent 3110 vehicles on route Dublin=>Cork
sent 465 vehicles on route Dublin=>Belfast
sending traffic ...
```

The distribution of work can be seen in the monitor output.

Additional processors can be added. Each time a processor is added, it contacts the broker in order to locate the other processors.

It then performs the peer-to-peer work distribution algorithm, in order to take work off the existing processors.

Below can be seen the exchange that takes place, when a second processor (running on Ubuntu) is added to the system:



The screenshot shows a terminal window titled "sean@sean-VBox-Ubuntu32: ~/sourcecode/sean/bitbucket/hpc\_trafficsim". The terminal output displays simulation statistics for four regions: Cork, Belfast, Galway, and Dublin. It lists connections between these regions and shows the progress of a traffic simulation, including waiting for startup and sending traffic. The output also includes optional parameters like PYRO\_NS\_HOSTNAME.

```
sean@sean-VBox-Ubuntu32: ~/sourcecode/sean/bitbucket/hpc_trafficsim

Cork at 51°47'N 08°16'W - 57518.0 vehicles of capacity = 100000 57.518% full
  Galway (0) =>
  Dublin (0) =>

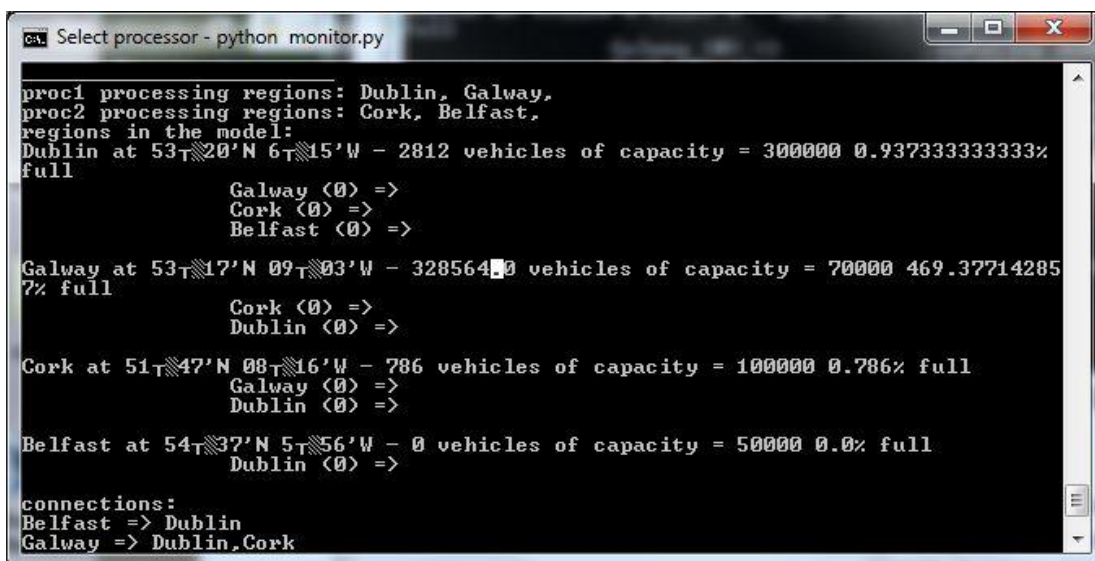
Belfast at 54°37'N 5°56'W - 70992.0 vehicles of capacity = 50000 141.984% full
  Dublin (0) =>

connections:
Belfast => Dublin
Galway => Dublin,Cork
Dublin => Galway,Cork,Belfast
Cork => Galway,Dublin

traffic sim waiting for startup to complete ...
targetNumRegionsPerProc = 2
traffic sim waiting for startup to complete ...
traffic sim waiting for startup to complete ...
Go to running state
processing 2 is listening ...
traffic sim running ...
sending traffic ...
sent 9 vehicles on route Cork=>Galway
sent 609 vehicles on route Cork=>Dublin
sending traffic ...
sent 599 vehicles on route Belfast=>Dublin
updating model ...
traffic sim running ...
sending traffic ...
sent 947 vehicles on route Cork=>Galway
sent 1993 vehicles on route Cork=>Dublin
sending traffic ...
sent 168 vehicles on route Belfast=>Dublin
updating model ...

optional:
  PYRO_NS_HOSTNAME
```

The monitor shows that 2 of the 4 regions have been given by the first processor, to the second processor:



The screenshot shows a terminal window titled "Select processor - python monitor.py". The output displays the assignment of regions to two processors (proc1 and proc2) and the current status of each region, including vehicle counts and capacity percentages.

```
Select processor - python monitor.py

proc1 processing regions: Dublin, Galway,
proc2 processing regions: Cork, Belfast,
regions in the model:
Dublin at 53°20'N 6°15'W - 2812 vehicles of capacity = 300000 0.937333333333% full
  Galway (0) =>
  Cork (0) =>
  Belfast (0) =>

Galway at 53°17'N 09°03'W - 328564.0 vehicles of capacity = 70000 469.377142857% full
  Cork (0) =>
  Dublin (0) =>

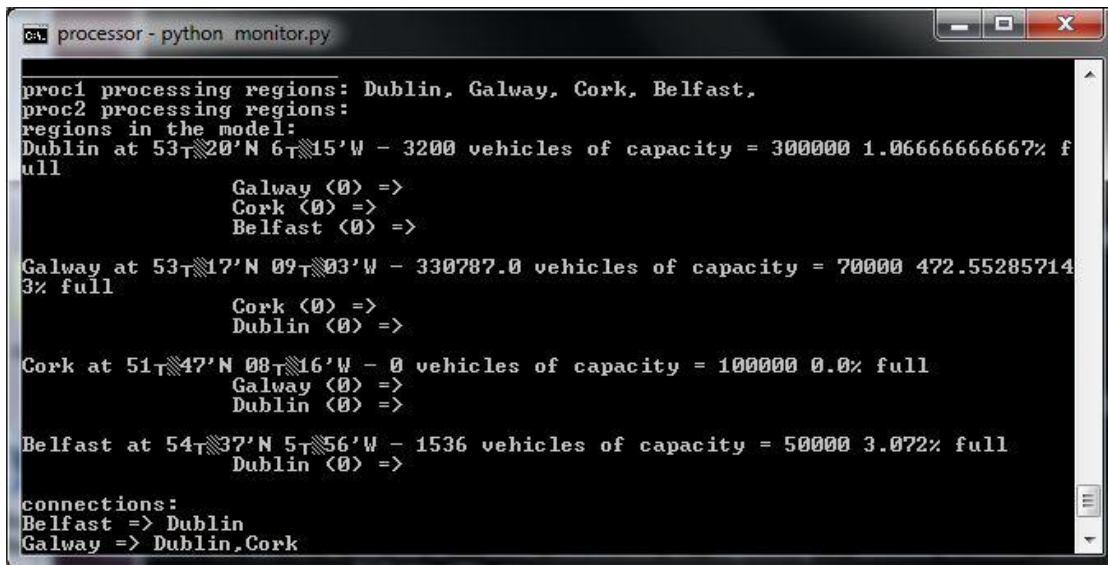
Cork at 51°47'N 08°16'W - 786 vehicles of capacity = 100000 0.786% full
  Galway (0) =>
  Dublin (0) =>

Belfast at 54°37'N 5°56'W - 0 vehicles of capacity = 50000 0.0% full
  Dublin (0) =>

connections:
Belfast => Dublin
Galway => Dublin,Cork
```

When a processor leaves the system, it gives back its work regions to the other processors, so that the full area of work continues to be processed (but at a slower rate, assuming that communication costs did not outweigh the higher number of processes).

When processor 2 exits, it redistributes its work to its peer processors. The monitor shows that all regions are now processed by the first processor once again:



```
processor - python monitor.py

proc1 processing regions: Dublin, Galway, Cork, Belfast,
proc2 processing regions:
regions in the model:
Dublin at 53°20'N 6°15'W - 3200 vehicles of capacity = 300000 1.066666666667% full
    Galway <0> =>
    Cork <0> =>
    Belfast <0> =>

Galway at 53°17'N 09°03'W - 330787.0 vehicles of capacity = 70000 472.552857143% full
    Cork <0> =>
    Dublin <0> =>

Cork at 51°47'N 08°16'W - 0 vehicles of capacity = 100000 0.0% full
    Galway <0> =>
    Dublin <0> =>

Belfast at 54°37'N 5°56'W - 1536 vehicles of capacity = 50000 3.072% full
    Dublin <0> =>

connections:
Belfast => Dublin
Galway => Dublin,Cork
```