Traffic Monitoring System

Altrim Beqiri
Joakim Soderlund

November 8, 2012

Contents

1 Introduction

The Document is organized into the following sections:

- **Runtime View** - Describes the general architecture of the system. The view, aimed at all stakeholders, is meant to provide an overview of the responsibilities of the different processes and bundles within the system. The view also shows where the different elements are to be deployed.
- **Service View** - Describes the different services used in the system and how the services communicate with each other. The view, aimed at developers, is meant to provide details about the services used in the system.
- **Communication View** - Describes the different messages used in the system, how to send messages, and how to receive messages. The view, aimed at developers, is meant to provide details about the communication infrastructure.
- **Organization View** - Describes the organization middleware used in the system. The view, aimed at developers, is meant to provide a details about how organizations merge and split.
- **Usage** - Contains a step-by-step guide for how to run the system.
- **Future Work** - Contains ideas that have not been implemented.
- **Glossary** - Contains the glossary of the document. This section can be found on page 25.

2 Runtime View

2.1 Primary Presentation

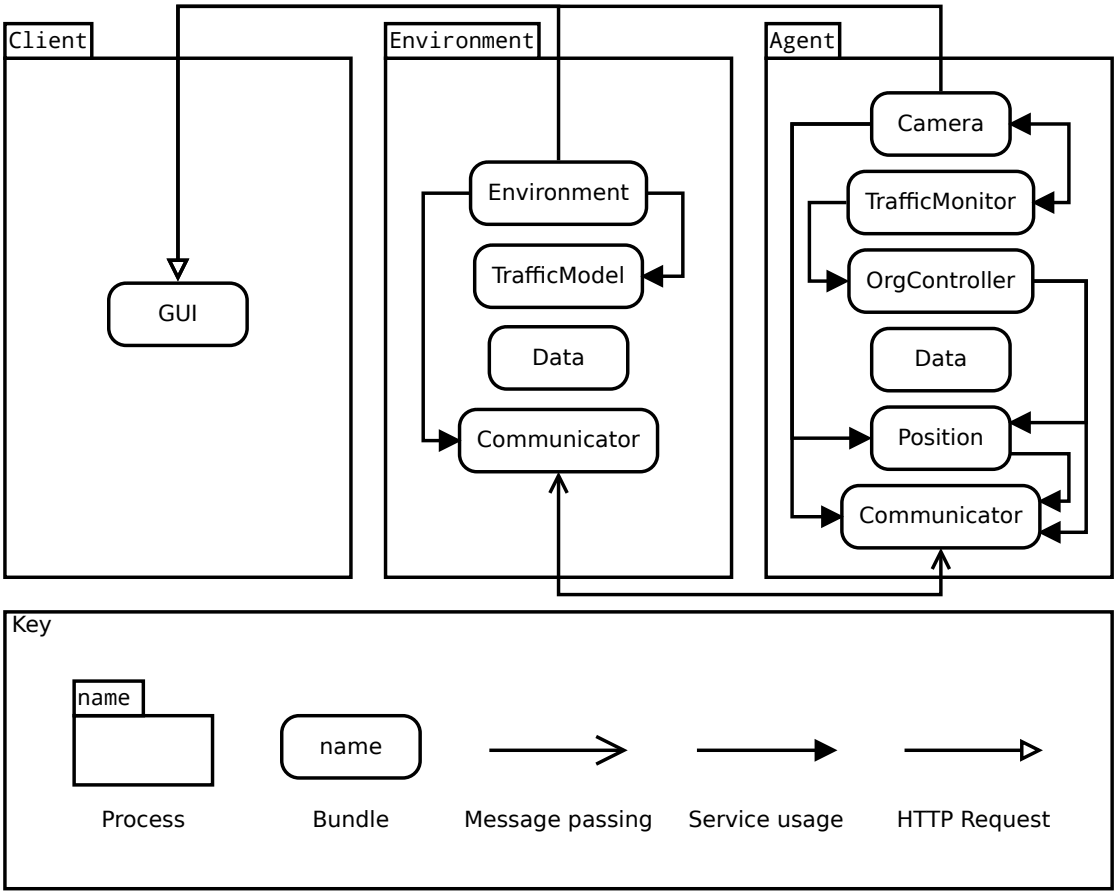

Figure 1: Primary presentation.

This view describes the general architecture of the system. The view, aimed at all stakeholders, is meant to provide an overview of the responsibilities of the different processes and bundles within the system. The view also shows where the different elements are to be deployed.

2.2 Element Catalog

2.2.1 Elements and Their Properties

- **Agent Process** - Process meant to run on the physical node cameras, each monitoring a single road segment. The Agent reports information about traffic congestions to the Client process via HTTP requests. There may be any number of these processes running in the system. Multiple Agents running within the same system will automatically organize themselves into organizations when there is a traffic congestion.

- **Environment Process** - Process simulating a road where road segments have various degrees of traffic. This process provides a GUI for configuring, starting, and stopping the simulation. The Environment reports information about the current degree of traffic on all road segments to the Client process via HTTP requests. There may only be a single instance of this process running within the same system.
- **Client Process** - Process used for monitoring the state of the system. This process provides a GUI displaying both traffic as reported by the Environment process, and traffic congestions as reported by the Agent processes. The process also provides web services which the other processes use to provide the Client with XML-formatted information. There may only be a single instance of this process running within the same system.
- **GUI Bundle** - Contains the GUI and web services for the Client process. The GUI is updated with information from HTTP requests as they are received.
- **Communicator Bundle** - Provides the communication infrastructure for inter-process communication within the system. For more information about this bundle, see section 4.
- **TrafficModel Bundle** - Provides a service for retrieving or setting the degree of traffic at a specific point in time for a specific camera (road segment).
- **Environment Bundle** - Initializes cameras and simulates the road. This Bundle provides a GUI for configuring and starting the simulation. Each time the traffic conditions change this bundle reports the changes to the GUI bundle via HTTP requests. Traffic conditions are also reported to the Agents via the communication infrastructure (section 4) when requested.
- **TrafficMonitor Bundle** - Provides a service for determining whether or not a certain degree of traffic results in a congestion. Information about the current degree of traffic is retrieved via a service provided by the Camera bundle.
- **Camera Bundle** - Provides a service for retrieving the current degree of traffic for the road segment monitored by the Agent Process. The bundle uses a service from the TrafficMonitor bundle in order to see if the road segment is congested or not. The bundle reports information about traffic congestions to the Client process via HTTP Requests.
- **OrgController Bundle** - Provides a service for managing the merging and splitting of organizations within the system. The bundle uses a service from the Position bundle to retrieve the Agents positions in order to manage the organizations. For more information about this bundle, see section 5.
- **Position Bundle** - Provides a service that handles information about the physical positions of Agents.
- **Data Bundle** - Bundle containing various constants and message payloads.

2.2.2 Relations and Their Properties

- **Service Usage** - Used for communication between bundles within a single process. Depicts that a bundle uses OSGi services provided by another bundle. Services are registered and retrieved using OSGi, and then used via regular method calls. For information about the services used within the system, see section 3.

- **Message Passing** - Used for communication between multiple processes. Messages are objects containing addressing information and a payload. These messages are then transferred using raw TCP/IP connections from one process to another. For more information about the messages used within the system, see section 4.
- **HTTP Request** - Used for communication between multiple processes. Similar to message passing, but uses XML-formatted data sent via the HTTP protocol. Used only by the Agent and Environment processes to report traffic degree and congestion information to the Client process.

2.2.3 Element Interfaces

This section only describes the web services and HTTP requests. For information about the services used within the system, see section 3. For more information about the messages used within the system, see section 4.

The GUI bundle provides three web services, both accepting XML in the format shown below, containing any number of camera elements.

```
<model organization="INT">
  <camera id="INT" position="INT" traffic="INT" congestion="BOOL" />
  <camera id="INT" position="INT" traffic="INT" congestion="BOOL" />
  <camera id="INT" position="INT" traffic="INT" congestion="BOOL" />
</model>
```

Figure 2: XML data sent to the Client.

The GUI bundle listens for HTTP POST requests via servlets on `/environment`, `/camera` and `/organization`. By default, the Environment sends requests to `http://localhost:8080/environment`, individual cameras to `http://localhost:8080/camera` and organizations to `http://localhost:8080/organization`.

The `organization` attribute in the model is optional, it is only used when posting to `/organization`. The `traffic` attribute is also optional, while all services can send information about the traffic, services may choose not to provide that information.

2.2.4 Element Behavior

Not applicable.

2.3 Context Diagram

Not applicable.

2.4 Variability Guide

The bundles Environment, TrafficModel, Communicator, Camera, Position and TrafficModel contain interfaces only. While implementations of these bundles are provided with the system, it is possible to replace any implementation of these bundles (element substitution) with another implementation. Note that this requires the replacement implementations to provide services conforming to the interfaces and their documentation.

2.5 Rationale

Not applicable.

3 Service View

3.1 Primary Presentation

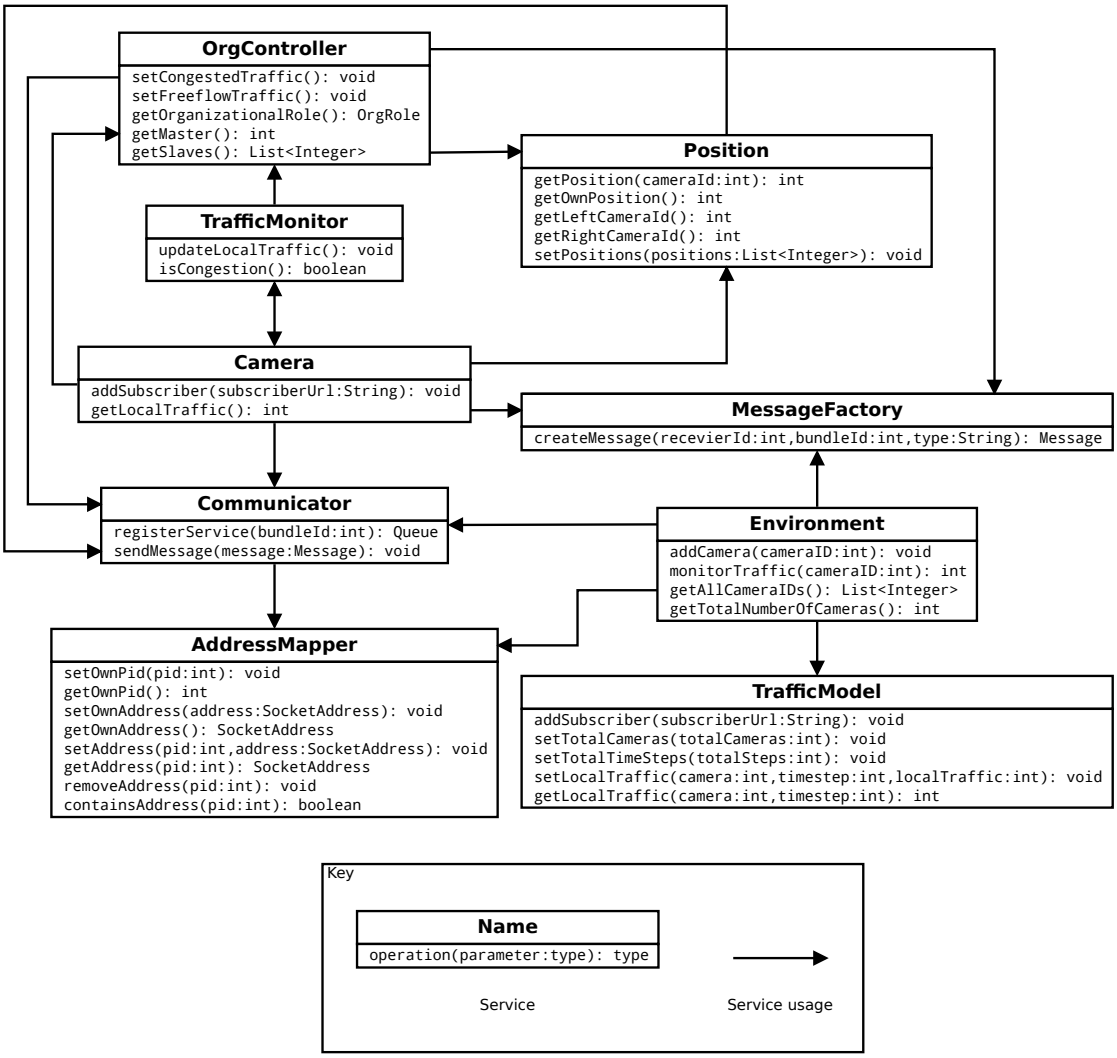

Figure 3: Primary presentation.

This view describes the different services used in the system and how the services communicate with each other. The view, aimed at developers, is meant to provide details about the services used in the system.

3.2 Element Catalog

3.2.1 Elements and Their Properties

- **TrafficMonitor** - Service that decides if the traffic is congested or not.
- **Camera** - Service that returns the local traffic that it observes. It also creates reports that are sent to the client GUI via an HTTP request.
- **TrafficModel** - Service that is used for modeling the traffic. Service for retrieving or setting the degree of traffic at a specific point in time for a specific camera (road segment).
- **Communicator** - Service that provides the communication between the environment and the agents. It uses the AddressMapper service for finding the actual addresses of processes.
- **Environment** - Service that provides information about the monitored traffic area and the deployed cameras. It uses the TrafficModel service to get information about the traffic and the Communicator service for communication between processes.
- **AddressMapper** - Service that provides the real addresses of processes. It maps camera IDs to socket addresses of processes.
- **MessageFactory** - Service that creates instances of messages used for communication between processes.
- **OrgController** - Service that is responsible for controlling the organizations. It handles creating, merging and splitting of organizations. It uses the Position service to retrieve camera positions in order to mange the organizations.
- **Position** - Service that handles the information about the physical position of Agents. Responsible for defining the camera positions which the OrgController uses to manage the organizations.

3.2.2 Relations and Their Properties

- **Service Usage** - Used for communication between the services. Depicts that a service uses methods provided by another service, meaning that the services depend on each other. The communication between services is done via regular method calls.

3.2.3 Element Interfaces

This section describes the optional classes that we use for more convenient way to get the services instances. The AddressMapper, TrafficModel, Communicator, Camera, Position, TrafficMonitor and MessageFactory services provide `Tracker` classes and the `User` interface. The `Tracker` classes are used to set an instance of a created service as soon as it gets registered. The `User` interfaces are used to retrieve instances of services.

```
AddressMapperUser environment = new EnvironmentImpl();
addressMapperTracker = new AddressMapperTracker(context, environment);
addressMapperTracker.open();
```

Figure 4: Example of AddressMapperTracker usage.

In the Environment service we use the `User` interface to retrieve the service as soon as it gets registered.

3.2.4 Element Behavior

Not applicable.

3.3 Context Diagram

Not applicable.

3.4 Variability Guide

The bundles Environment, Communicator, Camera, TrafficMonitor, Position and TrafficModel contain interfaces only. Even though implementations of these bundles are provided with the system. It is possible to replace any implementation of these bundles with another implementation. Note that this requires the replacement implementations to provide services conforming to the interfaces and their documentation. Also, the `Tracker` classes and the `User` interfaces are optional and can be ignored, they are provided only for a more convenient way of retrieving the services.

3.5 Rationale

Not applicable.

4 Communication View

4.1 Primary Presentation

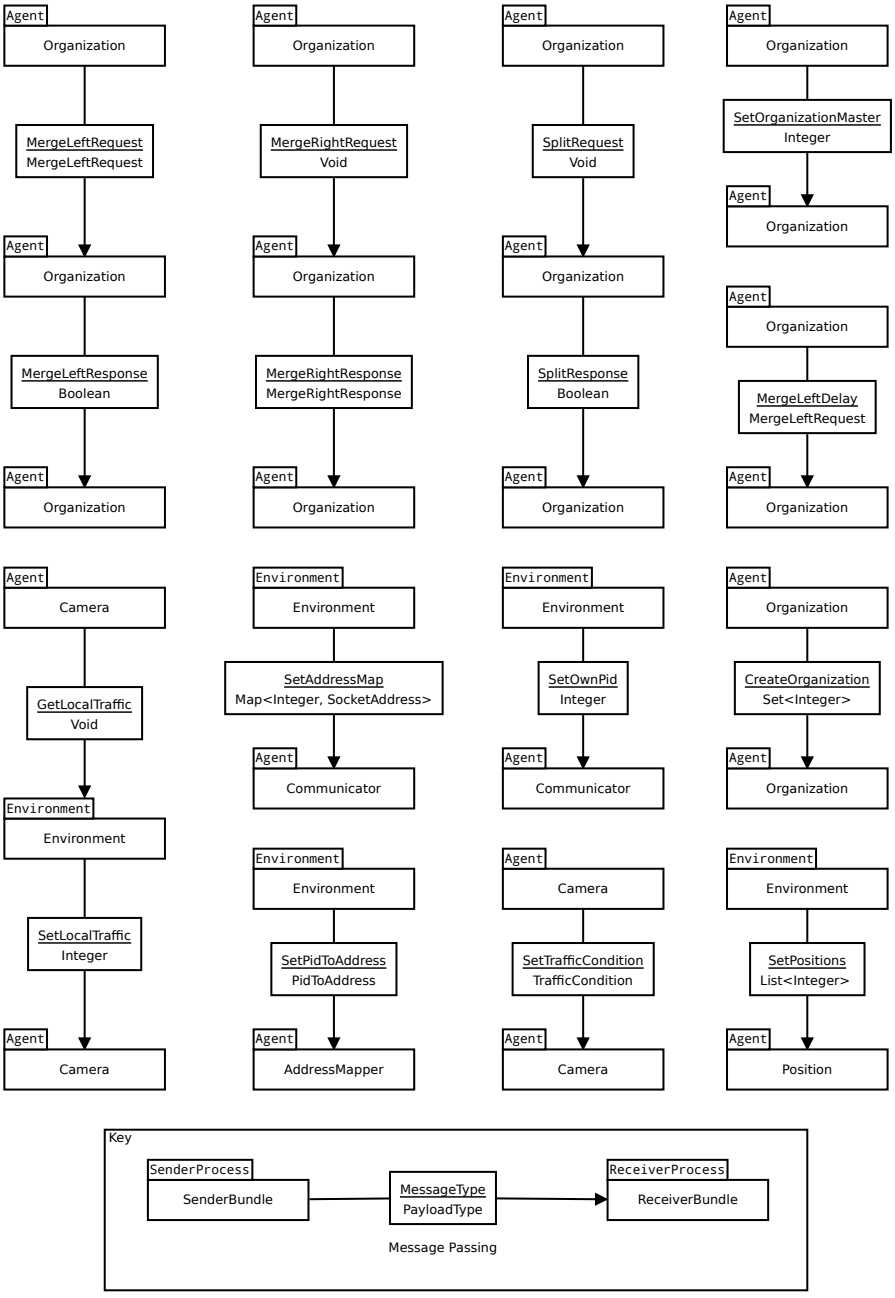

Figure 5: Primary presentation.

This view describes the different messages used in the system, how to send messages, and how to receive messages. The view, aimed at developers, is meant to provide details about the communication infrastructure.

4.2 Element Catalog

4.2.1 Elements and Their Properties

- **GetLocalTraffic** - Requests that the Environment provides the Agent with the current local traffic value.
- **SetLocalTraffic** - This message contains the degree of traffic that an Agent should report for the current timestep.
- **SetAddressMap** - When an Agent joins the simulation it has to know the addresses of the Environment and all other Agents. This message contains those addresses.
- **SetPidToAddress** - When new Agent joins the simulation this message informs all other Agents about the address of the new Agent.
- **SetPositions** - Defines the positions of all Agents within the same system.
- **SetTrafficCondition** - Slave Agents use this message to report their traffic condition to the master of the organization.
- **SetOwnPid** - When an Agent joins the simulation it must have a camera (process) ID. This ID is assigned to each Agent by the Environment. Thus, once the Environment has decided on an ID for an Agent, it is sent to the Agent via this message. Note that the default AddressMapper implementation always listens to these messages regardless of in which process the service runs in.
- **CreateOrganization** - Sent when an Agent's left neighbor splits from an organization, making the receiver the master of a new organization.
- **MergeLeftRequest** - Request sent to the left neighbor of an Agent, asking it to merge.
- **MergeLeftResponse** - Response with a true or false for the requested merging.
- **MergeLeftDelay** - A returned `MergeLeftRequest` which was sent to the wrong Agent due to changes in the organizations.
- **MergeRightRequest** - Request sent to the right neighbor of an Agent, asking it to merge.
- **MergeRightResponse** - Response with a true or false for the requested merging.
- **SetOrganizationMaster** - Informs Agents about their new master.
- **SplitRequest** - Requests an organization split.
- **SplitResponse** - Response with a true or false for the requested splitting.

4.2.2 Relations and Their Properties

- **Message Passing** - A message is an object transferred from one process to another within the system. The message contains addressing information, such as who sent the message and who the intended receiver is. A message also contains a payload which can be any (serializable) object. The payload is attached by the sender and is later unpacked by the receiver.

4.2.3 Element Interfaces

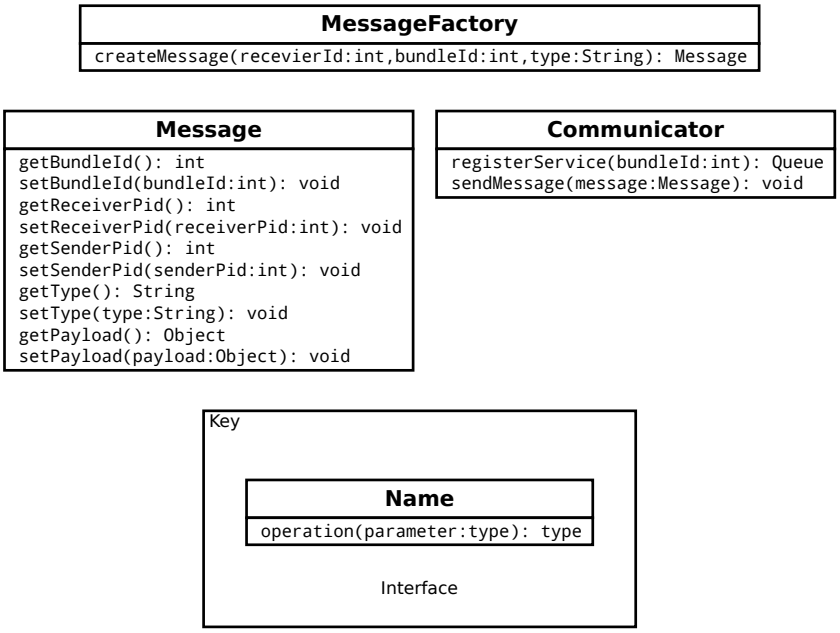

Figure 6: Element interfaces.

To send a message to another process a message must first be created. As the actual implementation of the message is unknown, this must be done via the MessageFactory service. To create a message one needs to know the camera (process) ID of the recipient, which is specified via the `receiverId` parameter. One must also know the bundle ID of the recipient. For the bundle ID we use a constant such as `COMMUNICATOR_BID`, which is by convention the hash code of the interface bundles' name. Note that this could potentially cause problems as the way hash codes for strings are calculated may vary depending on the Java implementation. The last parameter is a string specifying the message type (such as `SetOwnPid`, using the `SET_OWN_PID` constant). Below is some example code for creating and sending a message.

```
Message message = messageFactory.createMessage(
        cameraId,
        TMConstants.COMMUNICATOR_BID,
        TMMessage.SET_OWN_PID
);

message.setPayload(cameraId);
communicator.send(message);
```

Figure 7: Sending a message.

To receive messages one simply needs to register with the Communicator service and start reading messages from the returned Queue. Below is an example on how to read messages *without* using the convenience classes provided by the system. Note that each bundle ID can only register and retrieve its queue once.

```
BlockingQueue<Message> queue = communicator.registerService(
        TMConstants.COMMUNICATOR_BID,
);

while (true) {
        Message message = queue.take();
        if (message.getType().equals("SetOwnPid")) {
                int ownPid = (int) message.getPayload();
                System.out.println("My camera ID is " + ownPid);
        }
}
```

Figure 8: Receiving messages manually.

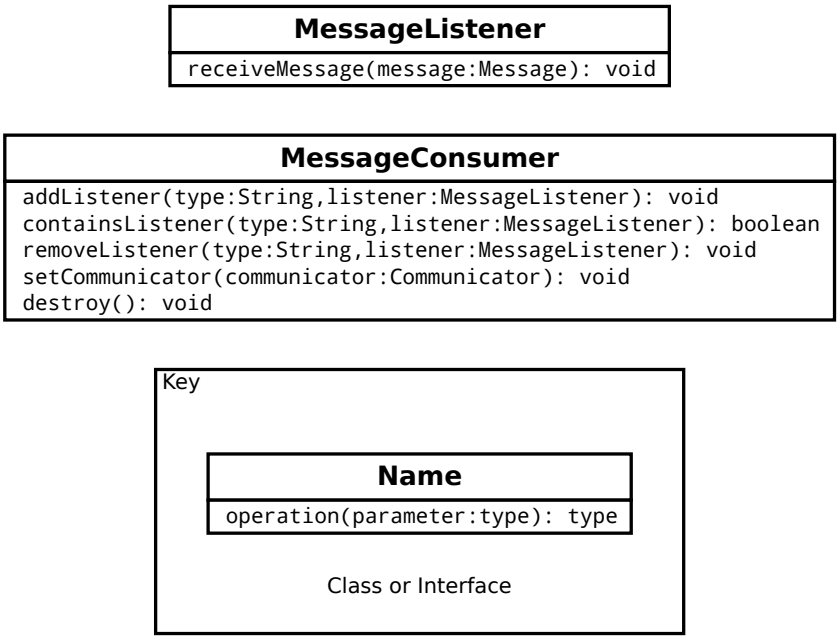

Figure 9: Convenience utilities for the communication infrastructure.

Receiving messages becomes more flexible with the convenience classes in the CommunicatorUtil bundle. CommunicatorUtil includes the MessageConsumer class and MessageListener interface. Classes interested in receiving messages should implement the MessageListener interface. Then, an instance of the listener can added to a MessageConsumer instance along with the desired message type. The MessageConsumer (implements CommunicatorUser, see section 3) will start reading messages from the queue and pass them along to the relevant listeners as soon as it has been given a communicator. It is recommended to use one MessageCommunicator per bundle and make it easily accessible.

```
MessageConsumer consumer = new MessageConsumer(
        TMConstants.COMMUNICATOR_BID,
);

consumer.setCommunicator(communicator);

MessageListener ownPidListener = new MessageListener() {
        public void receiveMessage(Message message) {
                int ownPid = (int) message.getPayload();
                System.out.println("My camera ID is " + ownPid);
        }
}

consumer.addListener("SetOwnPid", ownPidListener);
```

Figure 10: Receiving using CommunicatorUtil.

4.2.4 Element Behavior

Not applicable.

4.3 Context Diagram

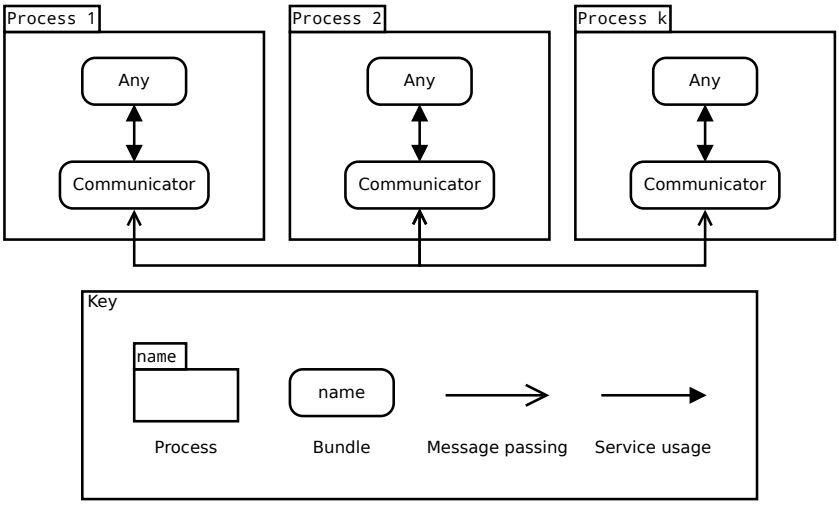

Figure 11: Context diagram.

The communication infrastructure can be used together with any number of processes. It can also be used by any bundle that requires inter-process communication. A message is sent point-to-point, from one process to another. Connections between processes are open only during message transfer.

4.4 Variability Guide

The message types within this system are defined only by their type string. Thus, new types messages can be used simply by specifying a new message type and attaching its payload.

4.5 Rationale

Not applicable.

5 Organization View

5.1 Primary Presentation

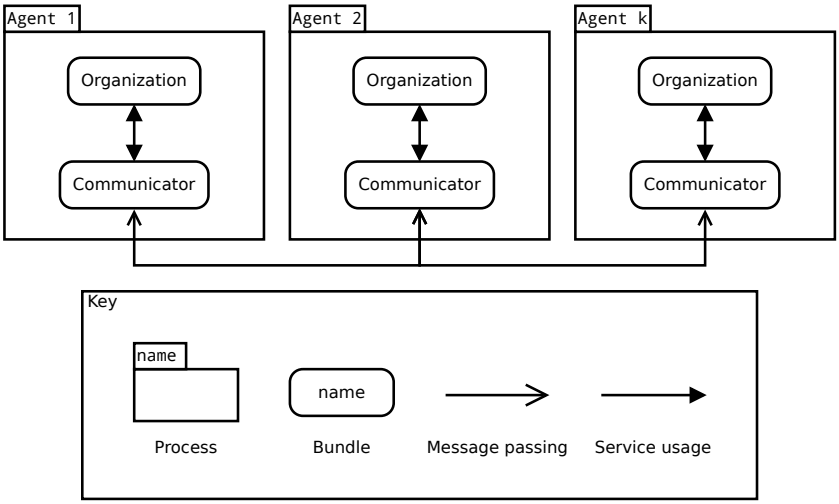

Figure 12: Primary presentation.

This view describes the organization middleware used in the system. The view, aimed at developers, is meant to provide a details about how organizations merge and split.

5.2 Element Catalog

5.2.1 Elements and Their Properties

- **Agent Process** - Process meant to run on the physical node cameras, each monitoring a single road segment. Agent processes are arranged into organizations by the Organization bundle.
- **Communication Bundle** - Provides services used by the Agent processes to communicate with one another.
- **Organization Bundle** - Middleware organizing the Agent processes into organizations.

5.2.2 Relations and Their Properties

- **Service Usage** - Used for communication between bundles within a single process. Depicts that a bundle uses OSGi services provided by another bundle. Services are registered and retrieved using OSGi, and then used via regular method calls. For information about the services used within the system, see section 3.
- **Message Passing** - Used for communication between multiple processes. Messages are objects containing addressing information and a payload. These messages are then transferred using raw TCP/IP connections from one process to another. For more information about the messages used within the system, see section 4.

5.2.3 Element Interfaces

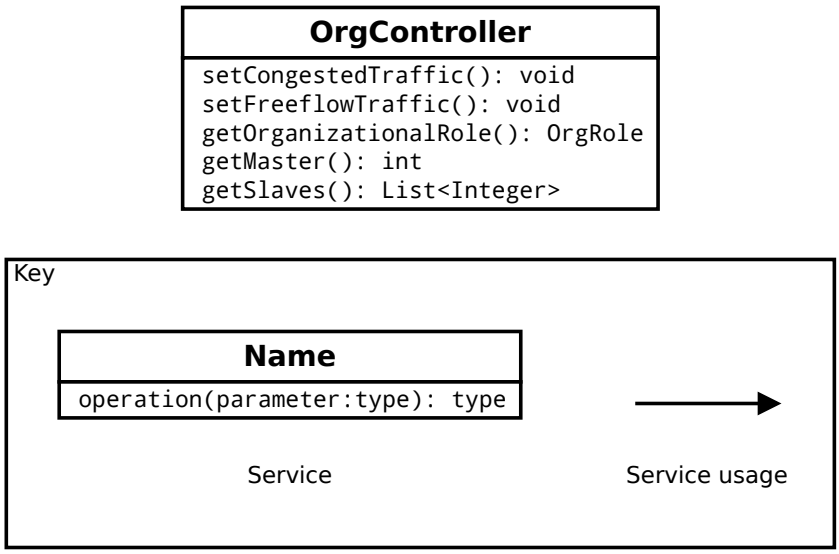

Figure 13: Element interfaces.

In order to properly control the organization middleware, the `setCongestedTraffic` and `setFreeFlowTraffic` methods are to be called each time `LocalTraffic` is updated. This is done by the default `TrafficMonitor` implementation.

5.2.4 Element Behavior

There are several different cases for merging and splitting organizations within the system. This section describes all cases handled by the organization middleware. Note that a master of an organization is always the Agent with the lowest (left-most) position. All cases below are seen from the perspective of a single Agent.

5.2.4.1 Attempt Merge

Occurs when a Agent detects a congestion. The only Agents capable of requesting merges are those with the role `SINGLE`. The reason being that the middleware is designed to make it impossible for two organizations containing multiple Agents to appear next to one another unless they already are in the process of merging.

1. If such an Agent exists, A `MergeRightRequest` is sent to the right neighbor. Set state to `MERGING_RIGHT`.
2. If the `MergeRightResponse` is affirmative, then:
   (a) Set role to `MASTER`.
   (b) The collection of Agents contained in the response is added to the slave set.
   (c) Each slave is sent a `SetOrganizationMaster` message.

3. If such an Agent exists, A `MergeLeftRequest` is sent to the left neighbor. The request contains a collection of Agents which may become slaves of the target Agent. Set state to `MERGING_LEFT`.

4. If the `MergeLeftResponse` is affirmative, then:

   (a) Set role to `SLAVE`.

   (b) Set master to the source of the `MergeLeftResponse`.

   (c) The slave set is cleared.

5. The Agent state is set to `IDLE`.

5.2.4.2 Left Neighbor Requests Merge

Occurs when the left neighbor wishes to merge. Note that we only accept requests when our state is `IDLE`. This is because if we are currently merging, then our left neighbor will soon receive a `MergeLeftRequest` from us.

1. A `MergeRightRequest` is received. If congestion and state `IDLE`:

   (a) Set role to `SLAVE`.

   (b) Respond with an affirmative `MergeRightResponse`, containing a collection of Agents which were member of the old organization.

2. Else:

   (a) Respond with a negative `MergeRightResponse`.

5.2.4.3 Right Neighbor Requests Merge

Occurs when the right neighbor wishes to merge. Note that as only masters of organizations (be it `SINGLE` or `MASTER`) may perform merges. Thus, if a `SLAVE` receives a `MergeLeftRequest` the request will be forwarded to its master.

1. A `MergeLeftRequest` is received. If the request was forwarded by an Agent which is not our slave:

   (a) Respond with a `MergeLeftDelay`.

2. Else if role `SLAVE`:

   (a) Forward `MergeLeftRequest` to master.

3. Else if `MERGING_LEFT`:

   (a) Wait until a `MergeLeftResponse` has been received to our own request, then attempt to handle the `MergeLeftRequest` again.

4. Else if not congestion:

   (a) Respond with an negative `MergeLeftResponse`.

5. Else if congestion:

   (a) Set role to `MASTER`.

(b) The collection of Agents contained in the response is added to the slave set.

(c) Respond with an affirmative `MergeLeftResponse` to the source.

(d) Each slave is sent a `SetOrganizationMaster` message.

5.2.4.4 Master Sent Merge Left Delay

Occurs when our forwarded `MergeLeftRequest` was rejected as our master is no longer an actual master.

1. When a new organization has been created, handle the returned `MergeLeftRequest` like in section 5.2.4.3.

5.2.4.5 Master Set Organization Master

Occurs when a new master has been assigned to an organization and introduces itself to its slaves.

1. A `SetOrganizationMaster` is received.
2. Set role to `SLAVE`.
3. Set master to the source of the `SetOrganizationMaster`.
4. Clear the slave set.

5.2.4.6 Master Sent Create Organization

Occurs when our left neighbor split from the organization, making our Agent the master of a new organization.

1. A `CreateOrganization` message is received.
2. The collection of Agents contained in the message is added to the slave set.
3. If the slave set is empty:
   (a) Set role to `SINGLE`.
4. Else if slave set is not empty:
   (a) Set role to `MASTER`.

5.2.4.7 Slave Sent Split Request

Occurs when a slave wishes to split from an organization. Note that requests for splitting are denied only when we received a request from an Agent which is no longer our slave. This can occur as multiple slaves wishes to split from an organization at the same time.

1. A `SplitRequest` is received.
2. If the request was sent from an Agent in the slave set:
   (a) Send an affirmative `SplitResponse`.
   (b) If any, send a `CreateOrganization` message to the requester's right neighbor, containing a collection of its slaves.

(c) Remove the requester and all Agents to its right from the slave set.

(d) If we no longer have any slaves, set role to `SINGLE`.

3. Else if the request was sent from an Agent not in the slave set:

   (a) Send a negative `SplitResponse`.

5.2.4.8 Attempt Split

For a `MASTER` a split is performed as follows:

1. Set role to `SINGLE`
2. The set of slaves is cleared.
3. If any, send `CreateOrganization` to the right neighbor containing a collection of its slaves.

For a `SLAVE` a split is performed as follows:

1. Sent a `SplitRequest` to master.
2. If the `SplitResponse` is affirmative, then:

   (a) Set role to `SINGLE`.

3. Else if the `SplitResponse` is negative, then:

   (a) When a new organization has been created, attempt to split again.

5.3 Context Diagram

Not applicable.

5.4 Variability Guide

Not applicable.

5.5 Rationale

Not applicable.

6 Usage

This section contains a step-by-step guide for how to run the system with Equinox in Eclipse 4.2 (Juno). The system can be configured by modifying the `TMConstants` interface in the `se.lnu.trafficmonitoring.data` package.

6.1 Run Configurations

1. Import all the bundles into your workspace as plug-ins.
2. Create a new `OSGi Framework` run configuration for the Client process.
   (a) In the Bundles tab, ensure that only the following bundles are selected:
   `se.lnu.trafficmonitoring.gui`
   `org.eclipse.equinox.http.jetty`
   (b) Click on *Add Required Bundles* to add all of the required dependencies.
   (c) In the `Arguments` tab, add `-Dorg.osgi.service.http.port=8080` to the VM arguments.
   (d) In the `Settings` tab, make sure that *Clear the configuration area before launching* is enabled.
   (e) Click *Apply*, your run configuration for the Client process is now complete.
3. Create a new `OSGi Framework` run configuration for the Environment process.
   (a) In the Bundles tab, ensure that only the following bundles are selected:
   `se.lnu.trafficmonitoring.communicator`
   `se.lnu.trafficmonitoring.communicatorimpl`
   `se.lnu.trafficmonitoring.communicatorutil`
   `se.lnu.trafficmonitoring.data`
   `se.lnu.trafficmonitoring.environment`
   `se.lnu.trafficmonitoring.environmentimpl`
   `se.lnu.trafficmonitoring.trafficmodel`
   `se.lnu.trafficmonitoring.trafficmodelimpl`
   (b) Click on *Add Required Bundles* to add all of the required dependencies.
   (c) This step is only relevant if the system is to be run on multiple physical nodes. In the `Arguments` tab, add `-Dse.lnu.trafficmonitoring.communicator.host=HOST` and `se.lnu.trafficmonitoring.communicator.port=PORT` to the VM arguments, replacing `HOST` and `PORT` with where the environment should be available.
   (d) In the *Settings* tab, make sure that *Clear the configuration area before launching* is enabled.
   (e) Click `Apply`, your run configuration for the Environment process is now complete.

6.2 OSGi Console

While the system will work fine with the above run configurations, Eclipse recently changed the way that the Equinox console works. The effect of this change is that an exception concerning the console will be thrown when the system starts. Below are two possible solutions to the problem for both of the run configurations, either solution will work for this system. However, the first solution is recommended as the OSGi console is unnecessary for this system.

1. In the `Arguments` tab, remove the `-console` argument from the program arguments. After that, click on the *Apply* button.
2. In the `Bundles` tab, enable the bundle `org.eclipse.equinox.console`, `org.apache.felix.gogo.command`, `org.apache.felix.gogo.runtime` and `org.apache.felix.gogo.shell`. After that, click on the *Add Required Bundles* button and then on the *Apply* button.

6.3 Agent Processes

The bundles required to run the Agent processes are:

- `se.lnu.trafficmonitoring.communicator`
- `se.lnu.trafficmonitoring.communicatorimpl`
- `se.lnu.trafficmonitoring.communicatorutil`
- `se.lnu.trafficmonitoring.data`
- `se.lnu.trafficmonitoring.orgcontroller`
- `se.lnu.trafficmonitoring.orgcontrollerimpl`
- `se.lnu.trafficmonitoring.position`
- `se.lnu.trafficmonitoring.positionimpl`
- `se.lnu.trafficmonitoring.camera`
- `se.lnu.trafficmonitoring.cameraimpl`
- `se.lnu.trafficmonitoring.trafficmonitor`
- `se.lnu.trafficmonitoring.trafficmonitorimpl`

While it is possible to create and launch Agent processes for the simulated environment via Eclipse, it is not recommended. A complete and easily deployable Agent has been included with the system. To launch agents, the executable `launcher.jar` can be run either manually or via the Environment process GUI. All bundles located in the same directory and starting with the filename `se.lnu.trafficmonitoring` will be automatically installed and started by the launcher. Thus, exporting and using new versions of the bundles with the launcher is easy. The Agent launcher takes a single argument, which is the port number that the Communicator bundle will listen to. The environment will expect the port numbers of the Agents to be $35200 + CameraID$. That is, it expects the first Agent process to listen to `localhost:35200`, the second Agent process to listen on `localhost:35201` and so on. When starting an Agent via Eclipse the port can be set via the VM argument `-Dse.lnu.trafficmonitoring.communicator.port=PORT`. If no port has been specified, a random one will be used.
Before starting the launcher from the Environment GUI, make sure that a `configuration` directory has been created in the directory of the launcher. If there is no such directory, simply start a single instance of the Agent. Once the Agent has started it should have created a `configuration` directory. Afterwards the Agent can be stopped and the Environment launched. The system will not work if the environment attempts to start multiple Agents without a `configuration` directory being present. It may be necessary to recreate the `configuration` directory after bundles have been updated or added.

6.4 Running

1. Launch the Client and Environment run configurations and wait for both processes to fully start.
2. In the Environment GUI, specify the number of cameras to include in the simulation.
3. Select the launcher JAR file via the *Select Launcher* button and use the *Start Processes* button to start the Agent processes. Alternatively, start the Agent processes manually. Make sure all cameras have fully started before proceeding to the next step.
4. Click the *Connect Cameras* button to start the simulation.
5. The simulation can now be monitored using the Client GUI.

6.5 Legend

Figure 14: Client GUI.

The Client GUI contains three different views, each showing a number of colored squares representing road segments. Within each square there is one label showing which Agent that monitors the road segment and another showing the current degree of traffic.

- **Environment** - Shows traffic and congestions as reported by the environment.
  - **Green** - Free-flowing traffic for the road segment.
  - **Yellow** - Traffic congestion for the road segment.
- **Camera** - Shows traffic and congestions as reported by each individual Agent.
  - **Green** - Free-flowing traffic for the road segment.
  - **Yellow** - Traffic congestion for the road segment.
- **Organization** - Shows traffic and congestions as reported by each organization. Agents belonging to the same organization have no borders between their squares. Note that the road wraps around from camera 5 to camera 6.
  - **Green** - Free-flowing traffic for the road segment.
  - **Yellow** - Traffic congestion for the road segment. The Agent is a slave.
  - **Red** - Traffic congestion for the road segment. The Agent is a master.

7 Future Work

- Use events instead of timers to decide when Agents report information to the Client.
- Ensure people suffering from color blindness can use the system properly.
- Ensure that the system works on networks that are not perfect.
- Handle unresponsive Agents in the organization middleware.

8 Glossary

bundle This refers to OSGi bundles. Each bundle concerns one specific subsystem. The bundles together make up a system.

component This is an object within a bundle with a specific responsibility.

congestion A road segment is congested when its traffic degree is higher than a certain threshold. This indicates that the road segment in question is occupied by too many cars for the traffic to flow easily.

message A message is an object transferred from one process to another within the system. The message contains addressing information, such as who sent the message and who the intended receiver is. A message also contains a payload which can be any (serializable) object.

organization A set of Agents collaborating to report congestions.

physical node This is an actual physical machine. It can run multiple processes with different process contexts.

process A process represents a running application in the system. Multiple processes can run on the same physical node.

receiver An object that can receive messages sent by other processes via the communication infrastructure.

road segment A part of a road monitored by a single camera.

sender An object that sends messages to other processes via the communication infrastructure.

service A globally accessible instance of a class (or interface).

servlet A Java class which handles requests according to the Java Servlet API. Upon receiving a request, servlets may either perform some task, return some data, or both.

stakeholder A person with a specific interest in the system. Each person can represent multiple stakeholders and the other way around.

subsystem This term refers to a specific part of the system.

system This term refers to the system as a whole, with all bundles taken into considerations.

traffic degree An integer (between 0 and 9) which indicates how much traffic occupies a certain road segment in relation to how much traffic that road segment can handle.

web service Functionality in a software system which can be accessed remotely via the HTTP protocol.