Session 4: Controlling the Simulation

Adapting the Simulation

Ingo Brinkmeier

Shawn Winter School 2008, 17.-21.11.08, Lübeck, Germany

November 18, 2008

1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

Outline

1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

Tasks (1 of 2)

- Tasks are parametrized commands invoked from the simulation configuration
- Used to
  - manage the simulation
  - running centralized algorithms
  - data-gathering from individual nodes
- Parameters are passed as (*name*, *value*)-pairs
- Access to the whole simulation environment via the *SimulationController*-parameter in their *run*-method
- Tasks are instantiated once when Shawn starts; using the same task-name again invokes the *run()*-method on the same instance

Tasks (2 of 2)

A simulation task must be derived from the following interface:

```
1 class SimulationTask
2 {
3 ...
4 void run(SimulationController&);
5 string name();
6 string description();
7 ...
8 };
```

run() Performs the actual task

name() Returns the unique identifying name of the task

description() Returns human readable description of what the task does

Before running a task it has to be added to the *SimulationTaskKeeper*

```
9  void init_ws08( SimulationControler& sc )
10 {
11     sc.simulation_task_keeper_w.add( new MyTask );
12 }
```

Example Tasks (1 of 2)

1. Configuration

   prepare_world
   : Constructs an empty world with given models

   rect_world/cuboid_world
   : Fills a rectangle/cuboid with random points (nodes)

   simulation
   : Runs the real simulation

2. Data-Gathering and Network status

   add_prestep/add_poststep
   : Adds a task that runs before/after each simulation round

   connectivity
   : Calculates the average node degree

Example Tasks (2 of 2)

3. Miscellaneous

   create_normal/create_uniform
   : Creates a random variable with normal/uniform distribution

   show_tasks verbose=1
   : Lists the implemented tasks and their description

   show_processors/show_edge_models
   : Prints all known processors/edge models i.e. factories

4. Loading and Saving the World

   load_world, save_world, random_seed

Loading and Saving worlds

- By loading and saving worlds it is possible to run simulations in the same scenario

```
save_world file=saved-world.xml snapshot=myID [append=true/false]
load_world file=saved-world.xml snapshot=myID
```

- Always use the parameter *snapshot* to identify the state of the saved/loaded world. The default value is *snapshot=%r*, r standing for the current simulation round
- *append=true* allows to save several snapshots in the same file
- Scenarios are stored in XML files

```
<scenario>
<snapshot id="19">
<node id="v0_W3DGUB-C">
<location x="0" y="0" z="0" />
</node>
[...]
</snapshot>
</scenario>
```

Rerunning the same simulation

- Many parameters in shawn are set using random variables, e.g.
  - Node placement in the world when using tasks like *rect_world*
  - Message transmission in the non-reliable transmission models
  - ...
- Random variables are created using a random seed
  → To reproduce exactly the same behaviour in two simulations the same random seed has to be used
- With the task *random_seed* it is possible to store and load a random seed or to set the random seed to a specific value

```
random_seed action=create filename=file_containing_the_seed
random_seed action=load filename=file_containing_the_seed
random_seed action=set seed=123456789
```

Outline

1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

Tags

- Tags are used to attach *persistent* or volatile data to Nodes, the World or the Simulation Environment
    - A simulation can be stopped and restarted without loosing information
    - Locally stored information can be collected and analyzed by evaluation tasks
    - Tags can be used to exchange data between different algorithms

Tags: Basic Functionality

- Tags decouple state variables from member variables
- Tags use the Handle concept (reference counting)
- There are three different tag types:

  Simple Tags contain a value of a certain type, i.e. double, integer, string, boolean

  Group Tags contain other tags

  Map Tags contain (key, value)-pairs of a certain types
- Persistent tags can be saved to and loaded from XML files using the tasks *save_world* and *load_world* respectively
- The task *tagtest* prints out all the nodes and their attached tags

How to use a Simple Tag (1 of 2)

- Simple Tags have a name and a value

```
std::string& name(void)
ValueType value(void)
void  set_value(const ValueType&)
```

- Create a new Tag by calling the constructor, e.g.

```
IntegerTag(const string& name, int value)
StringTag (const string& name, const string& value)
```

- Tags can be added to or removed from objects derived from the class *TagContainer*, e.g. *Node* or World

```
//overwrites eventually already existing tag with the same name
void add_tag(const TagHandle&)
//tag must be contained in the container!
void remove_tag(const TagHandle&)
```

How to use a Simple Tag (2 of 2)

- Tag access via iterators or the tags name

```
tag_iterator begin_tags(void)
tag_iterator end_tags(void)
//returns NULL if non existent
TagHandle find_tag_w(string& name)
```

- To store a tag with the *save_world* task, the tag has to be persistent

```
bool is_persistent (void)
void set_persistency (bool)
```

Tags: Example 1

```
1 /*Creates a new IntegerTag named tag_name,
2 sets the persistency of the Tag and
3 adds it to the owner()-node of a processor*/
4
5 IntegerTag* intt = new IntegerTag( "tag_name", value );
6 intt->set_persistency( true );
7 owner().add_tag( intt );
```

Tags: Example 2

```
1  /*Writes an integer value to an IntegerTag named tag_name
2  attached to the processors owner node.
3  First checks whether a tag with the given name already
4  exists and whether it is of the right type, otherwise
5  create a new IntegerTag.*/
6
7  TagHandle my_tag=owner().find_tag_w( "tag_name" );
8  if ( my_tag.is_not_null() )
9  {
10     IntegerTag* intt=dybamic_cast<IntegerTag*>( my_tag.get() );
11     if ( intt!=NULL )
12     {
13         intt->set_value( value );
14     }
15 }
16 else
17 {
18     IntegerTag* intt=new IntegerTag( "tag_name", value );
19     owner().add_tag( intt );
20 }
```

Tag Access for SimpleTags

- Note:

  Classes derived from TagContainer (e.g. Nodes, World, etc.) provide the methods

```
1 Type read_simple_tag<Type>( string& name )
2    throw( runtime_error )
```

  and

```
2 void write_simple_tag<Type>( string& name, Type value )
3   throw ( runtime_error )
```

  to read from and write to SimpleTags. These templated methods work for the Types int, double, bool and string.
  Include basic_tags.h to use simple tags.

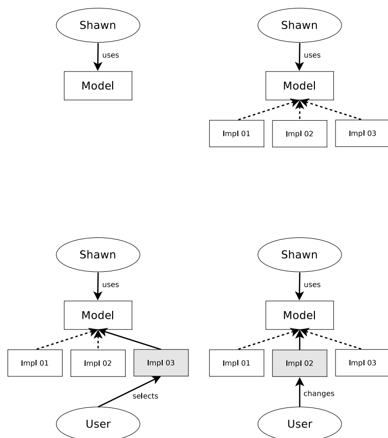
- Example:

```
4 // writes value to the IntegerTag named my_tag
5 // of a processors owner()-Node
6
7 owner().write_tag( "my_tag", value );
```

Outline

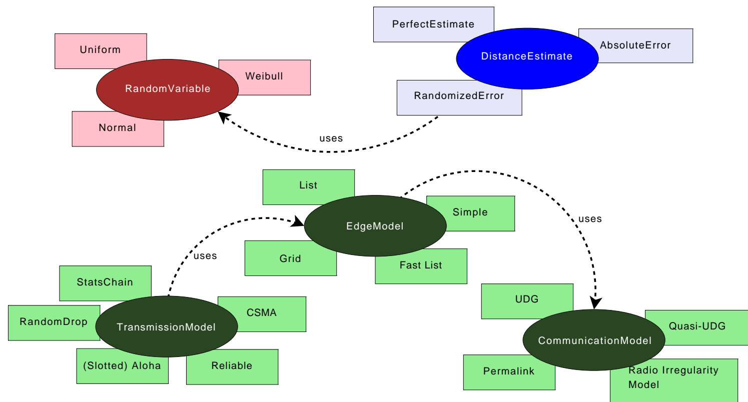
1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

General Concept of Models in Shawn



- General Interface
- Different Implementations
- Free Model Choice
- Flexibility

Available Models



Models: Overview

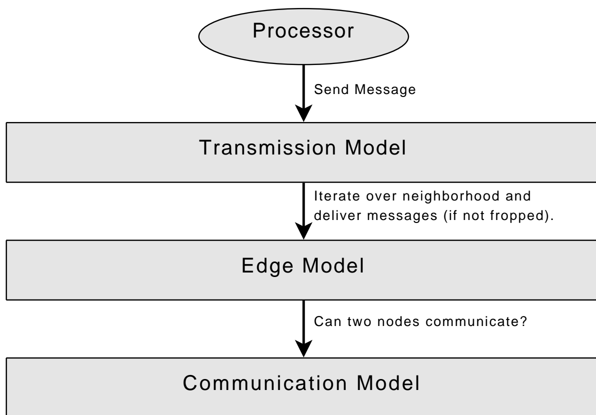
Communication Model
: Determines if two nodes can *in principle* communicate

Edge Model
: Provides access functions to the above defined graph, e.g. to adjacent nodes

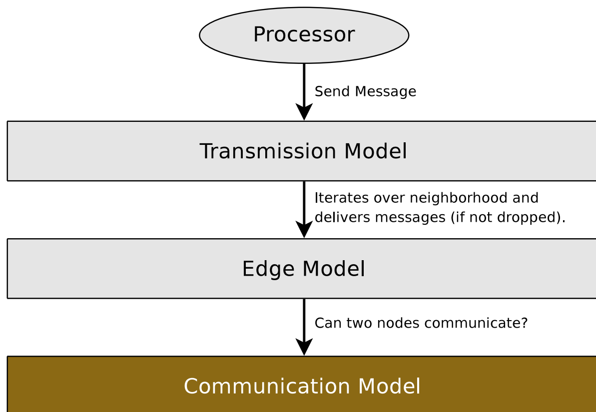
Transmission Model
: Simulates the transmission channel characteristics, i.e. decides for single messages how they are transmitted

Interaction between Models

Message transmission when Processor calls method `send()`.

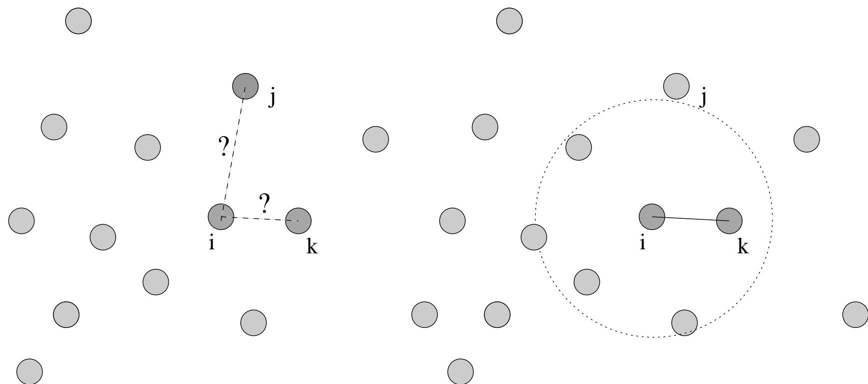


Communication Model



Communication Model

Which nodes can *in principle* communicate?



Basic Functionality

- Just **one** important method

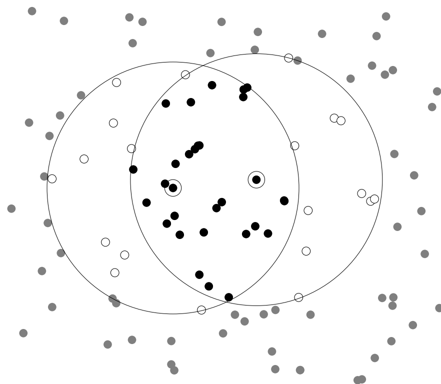
```
1    class CommunicationModel
2    {
3      ...
4      bool can_communicate_uni( \
5        const Node& u, const Node& v ) const throw();
6      ...
7    }
```

- Defines if messages from node *u* can *in principle* reach node *v*

Unit Disk Graph (UDG)

comm_model=disk_graph

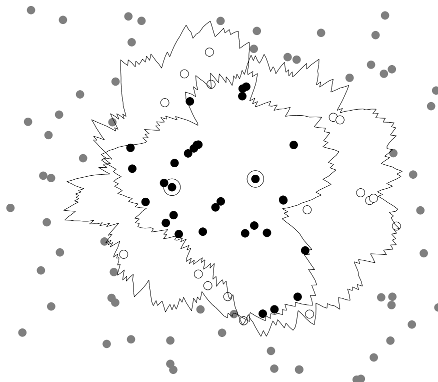
- If the distance between two nodes is less than or equal to a given radius, they can communicate
- Use the parameter `range` in the configuration file to set the communication range



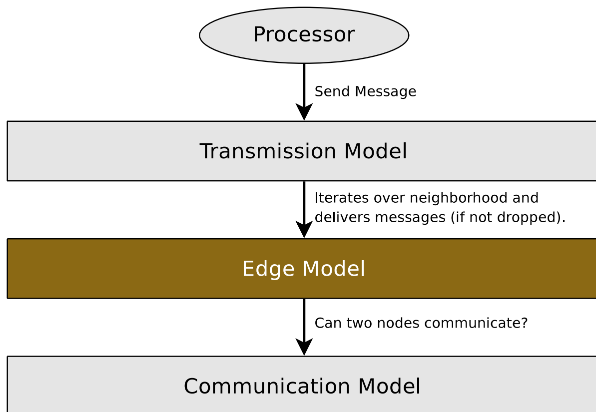
Radio Irregularity Model (RIM)

comm_model=rim

- Models variation in sending power and transmission range depending on the direction of signal propagation
- For parameter descriptions see Doxygen Documentation



Edge Model



Edge Model

Graph Representation of the Network

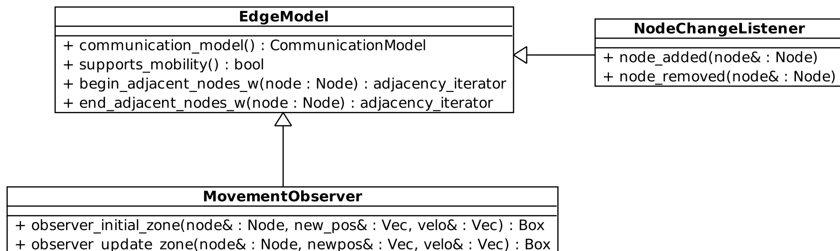
- Access to neighbourhood of each node via iterators
- Network may be mobile
- Usage:

```
1  // Access through the Edge Model
2  for( shawn::EdgeModel::const_adjacency_iterator
3  it = edge_model().begin_adjacent_nodes( node );
4          it != edge_mode().end_adjacent_nodes( node );
5          ++it )
6  {
7      const shawn::Node& neighbour = *it;
8      // do something with the neighbour
9  }
```

```
10 // Access from a node
11 for( shawn::EdgeModel::const_adjacency_iterator
12         it = Node::begin_adjacent_nodes( node );
13         it !=Node::end_adjacent_nodes( node );
14         ++it )
```

Basic Functionality

- Node access via iterators
- Callback when nodes added or removed
- Observe movements if mobility is supported



Simple Edge Model

edge_model=simpel

Functionality:

- Recalculates the neighbourhood of a node on **each** call iterating over **all** nodes in the network
  → Supports mobility
- Returns a node if `can_communicate(...)` returns `true`

Advantages:

- No memory overhead
- Support of Mobility

Disadvantage:

- Not efficient for large networks

List Edge Model

edge_model=list

Functionality:

- Initialization required: **One** initial run over **each** pair of nodes
- Stores neighbourhood of each node in a list
- On call only iterates over this list

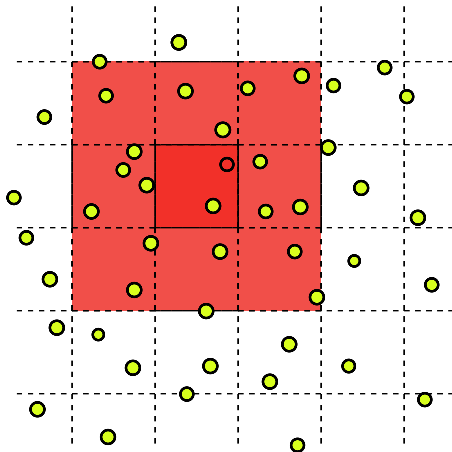
Advantage:

- Very fast, since neighbour list is pre-computed

Disadvantages:

- Initialization takes very long in large networks: $O(n^2)$
- Very memory consuming for large networks
- Mobility is not supported

Grid Edge Model



Grid Edge Model

edge_model=grid

Functionality:

- Model based on a geometric grid that is laid over the network
- Each node is assigned to the grid cell, that covers it
- Only nodes from a nodes own and its neighbouring cells are checked for adjacency
- Grid cell size is set using the transmission range
- On Movement: Node is moved from cell to cell

Advantages:

- Fast, since only a small amount of nodes are checked
- Support of Mobility

Disadvantage:

- Small memory overhead for the grid

Fast List Edge Model

edge_model=fast_list

Functionality:

- Combination of List and Grid Edge Model
- Stores neighbourhood of each node in a list
- Uses a Grid model internally for the initialization
- Initalization: Iterate over all pairs of nodes from neighbouring grid cells instead of all node pairs of the entire network

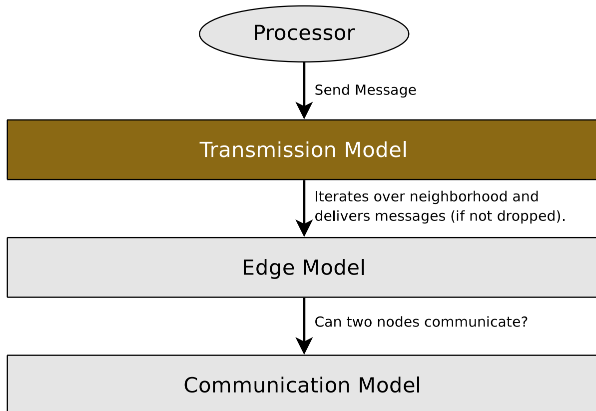
Advantage:

- Initialization phase on average faster than in the *List Model*
  $\rightarrow$ many unnecessary communication checks are avoided

Disadvantages:

- Mobility is not supported
- Slightly bigger memory overhead for the additional *Grid Model*

Transmission Model



Transmission Model

- Used to model transmission channel characteristics
  → Messages can be delayed, dropped or manipulated in other ways
- Nodes pass sent Messages to the Transmission Model by calling its method

```
1    virtual void send_message( MessageInfo& mi );
```

- Transmission Model responsible for the delivery to neighbouring nodes

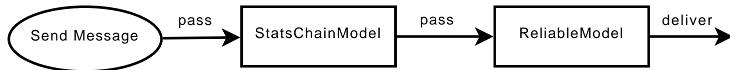
```
2    virtual void deliver_messages();
```

- There are chainable transmission models, that can be combined to form new composed models
- Several implemented models available: *Reliable*, *Random Drop*, *Statistics*

Chainability

- Several Models can be combined to form a chain
- Model must be derived from `ChainabletransmissionModel`
- A chainable model **can not** be the outermost model

```
...
prepare_world edge_model=list comm_model=disk_graph \
  transm_model=stats_chain
chain_transm_model name=reliable
...
```



General Workflow of Message Delivery

1. `Processor::send_message( const MessageHandle& );`
   Passes the Message to its *owner()*-Node that passes it to the World.
2. `World::send_message( Node&, MessageHandle& );`
   Creates the struct MessageInfo containing the message plus related information and passes it to the TransmissionModel.
3. `TransmissionModel::send_message( MessageInfo& );`
   Stores the message temporarily for delivery (if outermost Model), or calls `pass_to_chain()` (if chainable Model).
4. `TransmissionModel::deliver_messages();`
   After the simulation round ended, the World calls this method in the outermost Transmission Model to deliver the stored messages.

Available Implementations (1 of 2)

RandomDropTransmissionModel [transm_model=random_drop_chain]

- Drops messsages with a given probability between 0 and 1
  (Set the parameter `probability` in the configuration file)
- Chainable model
- In case a message is dropped, `pass_to_chain()` is **not** called
- Prints statistics at the end of the simulation

StatsChainTransmissionModel [transm_model=stats_chain]

- Collects statistics of sent messages
- Chainable model
- Prints statistics at the end of the simulation, i.e.
  Number of sent messages divided by type
- Use `dump_stats()` to display the current counters on the console

Available Implementations (2 of 2)

ReliableTransmissionModel [transm_model=reliable]

- Delivers every message
- Must be the outermost model
- If the parameter `immediate_delivery` is `true`, messages are not stored temporarily, but sent immediatly

CsmaTransmissionModel [transm_model=csma]

- Implements CSMA/CA
- Must be the outermost model
- If the transmission medium is blocked, the message is delayed until the end of the current transmission plus a backoff time

... and more available models

Outline

1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

Let's write our own Transmission Model

- Basic Functionality:
  - Messages are delayed with a given probability
  - Delayed messages, are delivered in the following simulation round
  - Implementation as outermost model, i.e. the model is *not* chainable
- We call the model `Ws08DelayedTransmissionModel`
- Create the files
  - `ws08_delayed_transmission_model.{h|cpp}`
  - `ws08_delayed_transmission_model_factory.{h|cpp}`

  in the directory `src/legacyapps/ws08`
  (Use `src/sys/trans_models/reliable_*` as a template)

Adapting ws08_delayed_transmission_model.h

- Derive the model from shawn::TransmissionModel
- Add private member attributes:

```
std::queue<TransmissionModel::MessageInfo*>
      aired_messages_, queued_messages_, delayed_messages_;
double prob_;
```

aired_messages_ will contain the messages that will be transmitted during the call of deliver_messages

delayed_messages_ will contain the delayed messages that will be transmitted in the next simulation round (the next call of deliver_messages

queued_messages_ will contain messages received during the call of deliver_messages that have to be delivered in the next simulation round

Adapting ws08_delayed_transmission_model.cpp (1 of 2)

- `Ws08DelayedTransmissionModel::init()`
  Read out the parameter "delay_prob":

```
1 se.required_double_param("delay_prob", prob_);
```

- `Ws08DelayedTransmissionModel::reset()`
  Copy and paste ReliableTransmissionModel::reset(void), empty all the queues
- `Ws08DelayedTransmissionModel::send_message(...)`

```
2 if (uniform_random_0e_1i() <= prob_)
3   delayed_messages_.push( &mi );
4 else
5   queued_messages_.push( &mi );
```

Adapting ws08_delayed_transmission_model.cpp (2 of 2)

- Ws08DelayedTransmissionModel::deliver_messages()

```
// copy the queued_messages_ for delivery to the aired_messages_
while( !queued_messages_.empty() ){
  aired_messages_.push( queued_messages_.front() );
  queued_messages_.pop();
}
// deliver the messages
while( !aired_messages_.empty() ) {
  MessageInfo* mi = aired_messages_.front();
  aired_messages_.pop();
  deliver_one_message( *mi );
}
// the delayed_messages_ will be delivered in the next round
while( !delayed_messages_.empty() ) {
  aired_messages_.push( delayed_messages_.front() );
  delayed_messages_.pop();
}
```

- deliver_messages() is called once at the beginning of each round. For a more precise timing use the Event Scheduler!

Adding a Factory and adapting the Init-File

- `Ws08DelayedTransmissionModelFactory::create()`

```
1 return new Ws08DelayedTransmissionModel();
```

- `Ws08DelayedTransmissionModelFactory::name()`

```
2 return std::string("ws08_delayed");
```

- Adjust the method `init_ws08(sc)` in `ws08_init.cpp`

```
3 sc.transmission_model_keeper_w().add(
4   new ws08::Ws08DelayedTransmissionModelFactory() );
```

- In the Configuration file use

```
...
prepare_world edge_model=list comm_model=disk_graph \
  transm_model=ws08_delayed
...
```

Outline

1. Tasks
2. Tags
3. Models
4. How to write a Transmission Model
5. Exercises

Exercises for this Session

1. Write a centralised task named "ws08-shortestpath", that attaches two tags to each node:
   - A Tag names "hop_count" containing the number of hops from the node to the gateway
   - A Tag named "predecessor" containing the label of one of the nodes predecessors in the routing tree to the gateway

   Use the Dijkstra algorithm to construct this routing tree.
2. Modify the implemented DelayedTransmissionModel so that messages can be delayed between 1 and max_rounds

Hints

1. Dijkstra algorithm: Given a connected graph $G = (V, E)$, a weight function $d : E \to \mathbb{R}$ and a fixed vertex $s \in V$, find a shortest path from s to each vertex $v \in V$.
   1. Set $i = 0$, $S_0 = \{u_0 = s\}$, $L(u_0) = 0$, and $L(v) = \infty$ for $v \neq u_0$. If $|V| = 1$ then stop, otherwise go to step 2.
   2. For each $v \in V \setminus S_i$, replace $L(v)$ by $\min\{L(v), L(u_i) + d(vu_i)\}$. If $L(v)$ is replaced, put a label $(L(v), u_i)$ on v.
   3. Find a vertex v which minimizes $\{L(v) : v \in V \setminus S_i\}$, say u_{i+1}.
   4. Let $S_{i+1} = S_i \cup \{u_{i+1}\}$.
   5. Replace i by $i + 1$. If $i = |V| - 1$ then stop, otherwise go to step 2.
2.
   - Read in an optional integer parameter "max_rounds"
   - Use a uniform random distribution to select the numer of rounds a delayed message will be delayed
   - Store the delayed messages together with their delay counter in an appropriate data structure and adapt the method `deliver_messages`