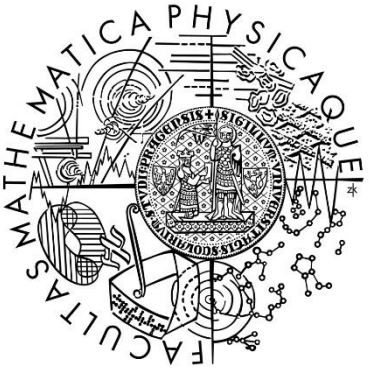
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Ondrej Kováč

Optimization of DEECo gossip-based communication

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2015

I would like to thank to doc. RNDr. Tomáš Bureš, supervisor my thesis, for the opportunity to collaborate on scientific research of cyber-physical system at the Department of Distributed and Dependable Systems. I am especially grateful for the advices and remarks I got during our meetings and for me a brand new look on the design of a distributed system.

I am also grateful to Michal Kit and Vladimír Matěna for our regular meetings and advices introducing me the world of DEECo.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague July 28, 2015

Název práce: Optimization of DEECo gossip-based communication

Autor: Ondrej Kováč

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů, Matematicko-fyzikální fakulta, Univerzita Karlove v Praze, Česká Republika

Abstrakt: Rozšiřování velkého počtu bezdrátových zařízení dalo podnět ke vzniku komponentového modelu DEECo určeného pro aplikace, jejichž neoddělitelnou součástí je mobilita a dynamická kompozice s architekturou vytvářenou za běhu. Velkou výzvou při realizaci takového systému je návrh komunikačního systému založeném na *gossip* protokolu. Takové řešení je zvlášť vhodné pro sítě typu MANET a jeho účelem je zvýšení spolehlivosti. V této práci jsme navrhli optimalizaci protokolu s využitím vlastností sítě s vlastní infrastrukturou. Zachovali jsme při tom *gossip* způsob komunikace bez zavedení centralizovaného prvku. Navrhované zlepšení spočívá ve vytvoření komunikačních skupin zformovaných na úrovni návrhu. Experimenty ukázaly podstatný pokles počtu odeslaných zpráv a celkově snížený čas doručení. Problematiku časování jsme pak zvlášť zpracovali pro MANET sítě a implementovali jsme mechanizmus *pullování*, který významně snížil latenci. Část této práce je věnována formální specifikaci sémantiky za účelem přesného zdůvodnění vlastností systému, čímž jsme také položili základ pro další rozšíření protokolu a budoucí výzkum v oblasti distribuovaných systémů.

Klíčová slova: *gossip*, MANET, distribuovaný systém, *pullování* zpráv, DEECo

Title: Optimization of DEECo gossip-based communication

Author: Ondrej Kováč

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic

Abstract: The spread of wireless devices inspired the creation of a DEECo component model suitable for designing applications with immanent mobility and dynamic composition where the system architecture emerges at runtime. A great challenge in implementation of such a system is the underlying communication mechanism based on gossip protocol in order to achieve resilience and suitability for MANET networks. In this thesis we propose an optimization of the protocol exploiting infrastructure networks, but still preserving the gossip-like communication without a centralized element. The improvement is based on forming communication groups introduced at the design level. The experiments show a substantial decrease in the number of sent messages and a decrease in time of data delivery. The timing aspect of data delivery is further elaborated for MANET networks by implementing a pulling mechanism with significant improvement of the latency. Part of this thesis is dedicated to a formal specification of the system semantic to provide a precise rationale about its properties and laying the ground for further extensions and research.

Keywords: gossip, MANET, distributed system, message pulling, DEECo

Contents

1. Introduction

The number of devices connected to the Internet is growing rapidly. According to Cisco research targeting the forecast of the impact of visual networking application on global networks almost half a billion mobile devices and connections were added in 2014 and the total number exceeded global population. Such a high number of devices and the increase of connectivity gives possibility to arise a new types of distributed applications. The previously futuristic concept *Internet of Things* (IoT) is no longer so unbelievable idea. Multiple devices closely interacting with the physical environment could be able to cooperate together and achieve a common goal which would be otherwise impossible taking into account individual units. A good example of such an application, commonly referred to as cyber-physical system, is so called smart grid [1] automatically adapting the behavior of electrical system to achieve efficiency and reliability.

A special case of cyber physical systems are those with inherent mobility which became more interesting with the rise of popularity of smartphones which possess sufficient computational resources, connectivity mechanisms, storage capability, high-level programming languages support etc. The dynamicity, mobility, great heterogeneity, large-scale and unreliable environment make engineering of such systems a very complex and difficult task. DEECo component model [2] is intended to simplify the development and to give the programmer a possibility to cope with these issues.

DEECo framework is suitable for designing large resilient distributed systems focusing on mobility. At design level it uses components which are composed into dynamic groups in order to cooperate on a common goal but still each component remains an autonomic entity. The decision of a component is made individually on the current belief of the system state resembling the agent-oriented computing. As a result an efficient decentralized execution avoids single point of failure.

One of the challenges in such a dynamic distributed system is the implementation of effective communication. It is necessary to find an optimal way so that the network won't collapse after deployment in the production environment because of the overload, but also that the content will be delivered where it needs to be. At the communication level we would like to adapt to the dynamicity as it is done

at the design level and make the system resistant to node failures or any unexpected communication behavior and also to deal with high mobility of individual nodes. It turns out that it's not possible to rely on the traditional paradigm from the world of computer networking, what makes the problem even more complex. Difficulties which arise are mainly the mobility of the devices and uncertainty of network topology which may rapidly change over time.

Various protocols have been developed for this purpose trying to exploit a particular mobility model or to pretend more solid network structure [3]. DEECo communication on the other hand is based on gossip protocol [4] which eliminates explicit communication channels and provides best-effort resilient communication. The advantages of gossip are mainly the simplicity and easy implementation. The protocol is based on a single rule of publishing local data or data from someone else to anyone who is ready to accept and as such is very resilient to node failures and dynamic topology. On the other hand gossip puts high demands on the transmission capacity and can lead to a network congestion. To make the protocol useful and effective we need to restrict the publishing so that the communication demands will be minimal but the data will be delivered anywhere it is necessary. In gossip we distinguish two types of methods called PUSH and PULL referring to the side initiating the request for data. Both approaches comparing to the other have its advantages and disadvantages and it will be interesting to investigate a synergy between them.

More particularly, we will focus on the following areas:

- DEECo makes the system engineering easier, but on the other hand development of such a framework faces the same difficulty as the programmer implementing a dynamic distributed system. Because of that there is a need for proper verification of the implementation. The design model must be therefore adequately described at the formal level to clearly outline system properties and deduce the derived properties. Our objective is to **extend the DEECo operational semantic defined in [5] by the definition of the communication model currently implemented in jDEECo and provide its documentation.**
- Subsequently our objective is to **extend proposed semantic in order to support efficient gossip communication on infrastructure and**

infrastructure-less networks allowing for PUSH-based and PULL-based communication. The extension will be in a comprehensive form allowing to reason about system properties and laying the ground for the implementation.

- Having the semantic description of proposed concepts we will **provide the implementation in jDEECo** (current implementation of DEECo).
- Finally our objective is to **experimentally evaluate proposed concept on a realistic use case scenario of car sharing**.

This work is further structured as follows: Chapter 2 describes the background necessary to prove our concepts such as DEECo component model, its Java implementation and simulation instruments. In 3 chapter we present an example of distributed application using DEECo. This application will be used to illustrate our concepts. Chapter 4 identifies communication aspects that should be considered and emphasize possible improvements of the communication protocol. Based on this statements Chapter 5 refines DEECo operational semantic to capture the network model as well as our improvements in the protocol. Chapter 6 deals with implementation issues of proposed solution. Experimental demonstration with the use of example application introduces in Chapter 3 is shown and explained in Chapter 7. Chapter 8 discusses typical use-case scenarios and possible future extensions. We mention similar approaches in Chapter 9 and an overall summary is covered by Chapter 10.

2. Background

In this Chapter we will shortly describe tools used to validate our concepts by simulation of an example application in a realistic environment.

2.1. DEECo Component Model

When building an application using DEECo targeting highly dynamic and distributed environment the only important thing is how to structure the software. The runtime of an external framework will take care of the communication and the management services. DEECo component model uses two basic principles – a component and an ensemble. A component is an autonomous unit of deployment developed without awareness of how communication will be performed. The only way of component interaction is through ensembles. An ensemble is a dynamically composed group of components formed by evaluation of component's data. We will further explain both concepts.

A component consists of a knowledge and processes. Knowledge expresses the current component state and contains all component's data in a hierarchical structure with mapping of field identifiers to their possibly structured values. A process is a computational logic which takes knowledge as an input, performs some processing and writes modified knowledge back to the component. It can be either periodic or triggered by a modification of a knowledge field. Every component periodically shares knowledge with other components in the system. Own knowledge is referred to as local while knowledge coming from others is referred to as replica.

An ensemble is a dynamically formed group of components which cooperate to achieve a common goal. In the language of component models the ensemble determine a composition. Membership to an ensemble is evaluated locally based on the local and replica knowledge. A component in the ensemble takes a role of a member or a coordinator which is assigned by periodically checking the membership of local and replica knowledge and vice versa. Complementary to the membership evaluation the ensemble performs an exchange of knowledge between the components forming a group. In particular between the coordinator and the member. Notice that only the local knowledge will be effected by the exchange, because replica will be updated by the remote instance of the ensemble.

Components are not explicitly conscious of an ensemble. Their processes are performed using a local knowledge only, which gets updated whenever the components becomes part of an ensemble. Further the ensembles do not know explicitly about other nodes. They only operate locally on the top of local and replica knowledge which may be modified by an update from the network. These assumptions made the system robust and reliable with respect to the ever changing environment.

2.2. OMNeT++

In this Section we will shortly describe the OMNeT++ simulation framework used in our experiments. OMNeT++ itself only provides a basic platform and instruments for writing simulation but does not contain any simulation components. These are supported by external projects such as INET, MiXiM or Castalia. OMNeT++ delivers a C++ class library with a simulation kernel, utility classes such as random number generator, topology discovery and many more, configurable infrastructure in order to compose simulation components, runtime user interfaces and more. Specifically we are using INET and MiXiM frameworks to simulate a network with wired and wireless communication.

Network models in OMNeT++ are assembled from so called modules. A module is a C++ written class which defines its behavior. Modules can combined together or interconnected with each other by gates. The composition is done by OMNeT++ configuration NED files. We are using modules for emulating the network stack and a module for node mobility.

Communication between OMNeT++ simulation and jDEECo is done by a Java Native Interface wrapper which provides a basic functionality such as running the simulation, sending or broadcasting a packet and setting the position of a particular node in the simulated network especially useful for nodes equipped with a radio device.

2.3. MATSim

MATSim is a framework for running agent based mobility simulation. We are using it to get a realistic imitation of vehicle and person movement. In order to run a MATSim simulation at least 3 configuration files needs to be provided:

- Network – specifies transport infrastructure, in particular nodes and links interconnecting nodes
- Population – plans (activities) of simulated agents
- Configuration – defines simulation parameters, used infrastructure, population, …

Additionally for other types of simulation more configuration files can be provided. For our evaluation we are using a simulation of public transport which requires two additional configuration files:

- Vehicles – defines available types and instances of vehicles
- Schedule – defines plans of public transport vehicles

For the large scaled simulations we are using a tool for generating random population and public transport schedule. The tool was written by us in C#.

Because MATSim is written in Java the integration with jDEECo is quite straightforward. The simulation is managed by a controller instance created and executed in our code. The controller can register several types of listener for instance simulation start, end, or step listener. We are using these to synchronize the simulation with execution of jDEECo processes and ensemble evaluations. MATSim agents' implementations such as the transit driver or passenger are replaced with custom ones so we are able to follow them from our simulation. For more information about the MATSim integration see Chapter 6.

3. Running Example

The main concepts will be illustrated on a realistic example from everyday life. First we describe a possibility for a distributed application and then we propose a solution using DEECo component model. This application will be further used in our experiments.

3.1. Smart Car Sharing

In large cities traveling to work or any other activity using the public transport can consume a serious amount of time. Also public transport is not as flexible as having a car. Many times you have to take a transfer line, the bus does not necessarily stop at your point of interest and you have to adjust your schedule to the timetable of the transport. None of these happen if you have a car. But on the other hand there are other disadvantages such as higher price, necessity of parking lots and from a general point of view more cars in the city increases the traffic congestion. A new concept of transport has been introduced to overcome these drawbacks referred to as *car sharing*. The basic idea of this approach is that if you have a car and you are regularly traveling to work you can share seats with other townsmen who are going to the same destination or to the destination on your way for a little tax. So you decrease your costs and the total amount of cars is decreased as well as there are less unused seats overall. Usually there is a webpage provided by a town or some company which allows to negotiate this agreement of neighbors.

Inspired by the idea of *car sharing* we have proposed an innovative design which will allow to negotiate a seat in the car at real time. Imagine people in the streets having a mobile *Smart Car Sharing* (SCS) application. Those having a car use the application to indicate how many available seats they have and the destination they are going to[1]. Similarly those without a car enter the number of required seats and their destination. The SCS application will then show to the drivers positions of candidates and possibly calculate the appropriate meet point. Equally the pedestrians will see those drivers' positions who meet their requirements. It is expected that the current position will be available to the application through cell phone GPS device. The application can be further extended with public transport buses. The only difference is

[1] With the combination of Google services it won't be even necessary as Google know where you are going to.

that bus driver has given check points and much more seats to offer. Such a service would be especially useful in cities without bus schedule or schedule which can't be properly kept because of difficult transport conditions or high traffic congestion. The user will be able to see online the buses currently passing along.

3.2. DEECo Approach

The very first step when designing a DEECo application is the identification of components and ensembles. Obviously drivers and pedestrians are modeled as components, while the propagation of positions of other persons in the system is captured by an ensemble. Figure 1 shows an example of how such components can be designed. Any component features arbitrary number of roles which act as interfaces in object oriented languages. Lines 1-4 classify a component which is aware of its position in GPS coordinates as well as of some other components positions'. The roles impose requirements on `Driver` component knowledge shown on lines 7-9. The current state is updated by `updatePosition` process (lines 10-14) which every second reads coordinates from GPS device and stores it to the driver position knowledge field. Both `Driver` and `Pedestrian` (lines 16-17) looks exactly the same, they monitor their current position and keep track of positions of interesting components. We have defined them separately because for `Driver` only `Pedestrian` components are interesting and vice versa.

```
1. role PositionAware
2.     position
3. role PositionAggregator
4.     positions
5.
6. component Driver features PositionAware, PositionAggregator
7.     knowledge:
8.          position = GPS(…)
9.          positions = [GPS(…), GPS(…), …]
10.         process updatePosition:
11.                 out position
12.             function:
13.                 position <- Sensor.getPosition()
14.             scheduling: periodic(1s)
15.
16.    component Pedestrian features PositionAware
17.         ... /* same as driver */
```

Figure 1. Example of a component definition for SCS application.

The interaction between driver and pedestrian is expressed by `PositionDistributor` ensemble. Coordinator of this ensemble can be only a component which aggregates positions (line 19) of its members (line 20). The

membership condition defines that driver will only interact with other pedestrians and vice versa (lines 22-24). When the condition is met the coordinator then adds the member's current position to the aggregation list (lines 25-26).

```
18.    ensemble PositionDistributor:
19.         coordinator: PositionAggregator
20.         member: PositionAware
21.         /*…*/
22.         membership:
23.              (coord is Driver && member is Pedestrian) ||
24.                   (coord is Pedestrian && member is Driver)
25.         knowledge exchange:
26.              coord.positions += member.position
```

Figure 2. Example of an ensemble definition for SCS application.

It is expected that some of the deployed applications will have a connection to infrastructure network while others only a radio broadcast medium.

4. Analysis

This Chapter is structured as follows: 4.1 summarizes DEECo computational model as defined in [5] and we introduce necessary notations used in next sections. In the following Chapter 4.2 we identify issues which should to be considered in order to provide an effective communication mechanism. Specifically these are: aspects of the network environment covered by Chapter 4.2.1, specific properties of infrastructure networks described in Chapter 4.2.2, the necessity of guarantee of content delivery described in Chapter 4.2.3 and finally requirements for the semantic refinement included in Chapter 4.2.4.

4.1. DEECo Computational Model

In this Section we will shortly describe DEECo computational semantics as defined in [5]. Notice that we have only included parts relevant to our research. For more detailed information please refer to [5].

The semantics of DEECo system $S = (\mathbb{C}, \mathbb{E})$ where $\mathbb{C}$ is a set of all components and $\mathbb{E}$ is a set of all ensemble definitions is defined by introducing a transition system $A(S) = \prod_{\forall C \in \mathbb{C}} A(C) \times \prod_{\forall E \in \mathbb{E}} A(E)$ constructed as a Cartesian product of component automata $A(C)$ and ensemble automata $A(E)$. Each transition is associated with an action $a \in \mathbb{A}$ and a timestamp $t \in Time$. We will further describe the component automata $A(C)$. Each component C is associated with valuation V_C of its knowledge, belief of valuation $V_C^{C_i}$ of other components $C_i \neq C$ and with a set of queues $\{Q_C^{C_i} | C_i \in \mathbb{C}, C_i \neq C\}$. Each queue is associated with an automaton $A\left(Q_C^{C_i}\right)$ depicted in Figure 3 and is intended to model the latency of belief propagation caused by transmission over network.

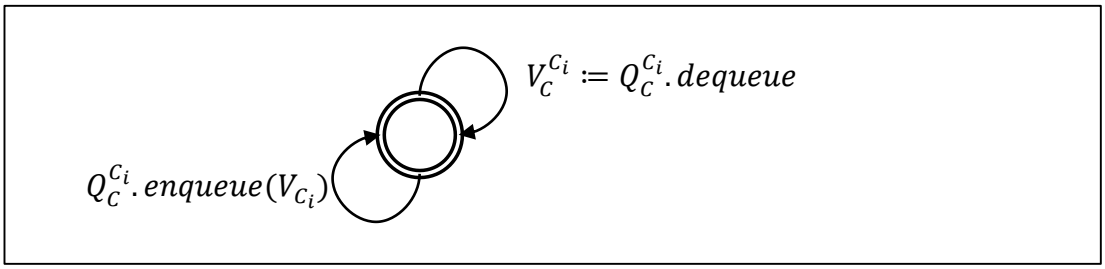


Figure 3. Knowledge valuation queue automaton (Taken from [5]).

The complete automaton $A(C) = \prod_{C_i \in \mathbb{C}} A(Q_C^{C_i}) \times \prod_{p \in P_C} A(p)$, where P_C is a set of all processes of component C and $A(p)$ is an automaton modeling the execution of process p.

Execution trace of $A(S)$ is formally defined as a sequence $T = (a_1, t_1), (a_2, t_2), \ldots$ for which it holds that $\forall a_i \in \mathbb{A}, t_i \in Time: t_i \leq t_{i+1}$ and we denote $\mathbb{T}(S)$ as the set of all traces of system S. In order to reason about knowledge valuation considering timing aspects we introduce a function $V_T^C: Time \rightarrow \mathbb{V}_C$ defined as follows: $t < t_1: V_T^C(t) = I_C$ otherwise V_T^C equals to V_C after all actions $a_i, \ldots, a_m, m = \max(i | t_i \leq t)$. The I_C symbol here refers to the initial knowledge valuation of component C.

The above Section describes a relatively general DEECo operational semantic $\mathcal{D}$ and it is intended to be refined whenever a new feature is implemented. Refinements of semantic $\mathcal{A}$ by semantic $\mathcal{B}$ (denoted as $\mathcal{A} \preccurlyeq \mathcal{B}$) is formally defined as follows: for any system $S = (\mathbb{C}, \mathbb{E})$ and each real-time execution trace $T_{\mathcal{A}} \in \mathbb{T}_{\mathcal{A}}(S)$ featured by semantic $\mathcal{A}$, there exist a trace $T_{\mathcal{B}} \in \mathbb{T}_{\mathcal{B}}(S)$ featured by semantic $\mathcal{B}$ such that for each $C \in \mathbb{C}$ it holds that $V_{T_{\mathcal{A}}}^C = V_{T_{\mathcal{B}}}^C$.

4.2. Shortcomings and Problem Identifications

4.2.1. Communication Aspects

The computational model defined in [5] and shortly described in 4.1 is very general and as such has been intentionally designed to allow new features to be added by semantic refinement. On the other hand it omits several properties necessary for efficient system design especially when deployed in a distributed environment. It does not specify how communication should be performed and the concept of component communicating with any other is rather unrealistic. We need to be able to consider the fact that some component is not reachable at specific time. Also we should take into account different behavior of the communication interface in infrastructure networks or MANET.

Communication between components is based on sending and reception of messages. The content of a message can be of arbitrary size and may exceed the transmission limit given by the physical environment. Therefore the content fragmentation and combination should also be considered. Moreover there is a

significant difference between communication of components deployed on a single node and components on different nodes interconnected by a network.

DEECo communication mechanism based on gossip protocol seems to be suitable because of the unreliable and dynamic environment but may be inefficient or event infeasible when the total number of messages exceeds the network throughput. It is carried out by a regular broadcast of component knowledge data. Any component upon message reception stores it locally and retransmits. Such a solution is also referred to as epidemic protocol based on the similarity with virus spreading in biological terms or flooding. Data dissemination by flooding may lead to network overload and so called *broadcast storm problem* [6]. Drawbacks of flooding are in particular: (i) redundant retransmissions – a node decides to retransmit when neighbors already have the message, (ii) contention – to many broadcasts from nearby hosts (iii) and collisions – more likely to occur.

In a network of n nodes flooding results in n message transmissions and delivers the message to every node. In a realistic production environment the boundary of the network is potentially unrestricted and therefore it is necessary to have a possibility to limit the communication reach to interested components as well as to optimize the communication and to decrease the total number of messages. In order to do that any component upon message reception decides, using local information only, whether to retransmit it or not. Several possible solutions have been introduced such as probabilistic forwarding. A message upon its first reception is retransmitted with probability $p \leq 1$ which is a parameter of the protocol. For a particular network topology there exists a retransmission probability p_{th} such that almost all nodes receives the disseminated information [7]. If $p_{th} < 1$ then gossip algorithm clearly outperforms epidemic spreading. Probabilistic forwarding can be combined with other forwarding schemes for instance counter-based, distance-based, location-based or cluster-based [6] in order to reduce the impact of broadcast storm and to decrease the total number of retransmitted messages. All of these techniques use some information coming from the network to make the decision. An interesting alternative solution could be to consider some of the domain knowledge of a component.

Part of the communication model is already described in [8] but not formally specified. This model presents an optimization in MANET networks, but does not

involve communication in infrastructure networks which are fundamentally different and can be exploited more effectively. The following Chapter analyzes this issue.

4.2.2. Exploiting Infrastructure Networks

The optimization listed in [8] are useful in MANET network where nodes operate on a short-range radio broadcast device and are only capable of communication with near neighbors. If more distant target is addressed the message should be delivered using intermediary nodes passing it from one to another until it reaches the final destination.

When a component is deployed in a network with more reliable infrastructure which is not only capable of broadcasting but also can route a message directly to the receiver the gossip based protocol can be adjusted to improve the performance. We can exploit the routing mechanism so costly in MANET and not only restrict the retransmission. We can directly address the content to groups of interested nodes. The routing mechanism requires the nodes in the network to be addressable. If we are able to identify nodes involved in communication i.e. to form a communication group, we can restrict the broadcast only to the members of the group. In order to implement this solution we need a mechanism which will identify members of those groups and maintain them. Distributed environment puts demand on how communication groups will be established at runtime as there should be no single point of failure.

4.2.3. Increasing Gossip Reliability

As described above gossip dissemination protocols are based on selective retransmission of received messages taking into account only local information. This procedure is referred to as a PUSH mechanism as the data from sender are *pushed* to potential recipients. A feasibility of this approach can be broken when the retransmission decision algorithm releases too many messages. For instance a probability based gossip with threshold 0.9 would retransmit almost all messages and presumably leads to a broadcast storm.

In order to reduce the total number of messages in the network we can decrease the probability. However if the probability is too low the system may suffer from undelivered or too outdated messages. Consider network topology as shown in Figure 4. All nodes $n_i, \dots, n_k, p$ are in the transmission range of s except q which is only in

transmission range of p. Obliviously the probabilistic forwarding should not deliver a message from s to q as p must not decide to retransmit. Similarly in counter-based forwarding with any counter threshold k it is possible that $n_1, \dots, n_k$ will rebroadcast before p and therefore p won't.

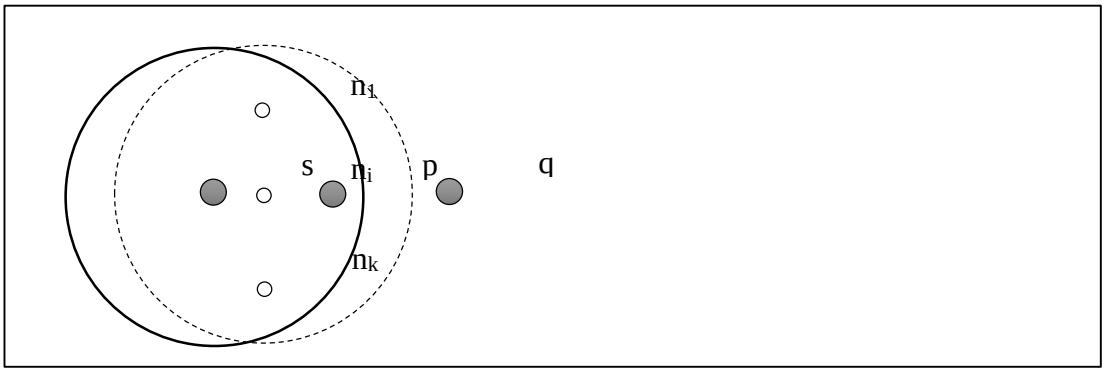


Figure 4. The Necessity of pull request (taken from [9, p. 177]).

Therefore we need a recovery mechanism which explicitly requests the missing message to ensure the reliability of its delivery. This procedure is referred to as PULL method as the data is actively *pulled* by recipients from the sender. There are some well-known pull algorithms [10] basically working in the following way: Every node regularly broadcasts headers of messages it received. It is expected that headers are much smaller than the original message content and that multiple headers can be combined into a single pack. Also message headers are not retransmitted so there is no danger of the broadcast storm. This mechanism allows particular node to realize that some of the messages sent somewhere else in the network are missing and to initiate a PULL request. Thinking more deeply about the possible implementation we must also consider timing aspects. The fact that a message is not delivered does not mean that a PULL request is necessary because it can be delivered in a meantime or the message is so outdated that it is no longer relevant for the application.

While push-based protocol can achieve low latency and high tolerance to communication failures the redundancy and overhead may be unacceptable [11]. On the other hand pull-based protocol can produce lower overhead at the cost of higher latency. Combination of both approaches can lead to a promising solution if it is well balanced. Notice that setting up the protocol for rather pull than push leads to broadcast storm similarly to the probabilistic forwarding with high probability. It is expected that there will be more nodes interested in the sent message and all of them can receive it in one transmission. Letting them to pull the message results in a significant overhead.

As an input parameter we require from the software design process to specify demands on the acceptable latency.

4.2.4. Semantic Requirements

The above proposed modifications should be captured by semantic extension to allow for reasoning about the system and to see how its protocols are designed. There is a big gap between the general semantic and concrete implementation. We need to refine the semantic in such a way that it can be taken by the developer as a specification and implemented. Current DEECo implementation lacks the formal definition which will help to see clearly its properties and deduce its derived properties.

We will describe the semantic refinement by stating rules which must be kept in order to support given features. The rules actually play role of a guard restricting the original semantic.

5. The New DEECo Communication Model

Considering the aspects analyzed in Chapter 4 we propose several DEECo extensions defined by the semantic refinement. This Chapter is structured as follows: 5.1 discusses the communication aspects, in particular multiple components on a single node, network topology and fragmentation of messages. The following Chapter 5.2 takes the communication model, proposes a communication protocol based on gossip together with some optimization i.e. bounding the dissemination and group communication. Another point of view on the gossip communication is taken in Chapter 5.3 considering pull communication as a counterpart of push. Finally Chapter 5.4 gives an informal proof of valid DEECo semantic refinement.

5.1. Communication Aspects

We extend DEECo semantics $\mathcal{D}$ as specified in the DEECo computational model [5] by introducing communication aspects. The semantic refinement is described with rules in the form of a logical formula stating about execution traces of system S.

Taking a general point of view we say that the components in the system communicate by sending messages. Later we assign a specific meaning to particular message such as knowledge valuation or other type of content. Reusing the notation established in [5] we use V_C to refer to a message V sent by component C. But contrary to that notation we use V_C rather for a single instance of a message holding knowledge valuation not necessarily the most current one. This assumption makes difference when taking into account the timing aspects. Knowledge valuation of a component at two instances of time can be the exactly the same but messages sent with these valuations are considered to be different.

Given an execution trace $T = (a_1, t_1), (a_2, t_2), \ldots$ we consider several aspects of the communication among components given by the environment described in the following chapters.

To ease reading, Table 1 and Table 2 summarize the notation defined and used in the following subsections.

Predicate/Function/Set	Description
$Dep \subseteq \mathbb{C} \times \mathbb{C}$	Deployment equivalence relation outlines components deployed on the same node.
$Net: 2^{\mathbb{C}} \times 2^{\mathbb{C}} \times Time \rightarrow Time \cup \{\infty\}$	Network topology function defines connection between nodes and transfer time of a message sent from one to another.
$Pub \subseteq [2^{\mathbb{C}}] \times [2^{\mathbb{C}}] \times \{\bigcup_{C \in \mathbb{C}} \mathbb{V}_C\} \times Time$	Publish predicate defines strategy of messages sending from one node to another at particular time.
$Rt_2 \subseteq [2^{\mathbb{C}}] \times \mathbb{V}_C$	Retransmission predicate defines strategy of given message retransmission by particular node at layer 2.
$Bnd \subseteq \mathbb{V}_{[C]} \times \mathbb{V}_C$	Boundary predicate defines strategy of given message retransmission by particular node taking into account node knowledge.
$Peer_{[C]}: Time \rightarrow [2^{\mathbb{C}}]$	Peers function for particular node defines set of known nodes at specific time.
$Rcp_{[C]}: Time \rightarrow 2^{Peer_{[C]}}$	Recipient function selects recipients of a message at particular time. These recipients are taken from known nodes only.
$Part: \mathbb{V}_C \rightarrow D$	Partitioning function divides knowledge of components into several groups.

Table 1. Recapitulation of predicates and function used for semantic refinement.

Structure	Description
$Q_{[C]}^{[C_i]}$	Node queue encapsulating individual queues between components operating like priority queue with reception time as priority value.
$St_{[C]}$	Node storage encapsulating individual queues between nodes. Allows for communication between components deployed on the same node.
$B_{[C]}$	Node buffer of received messages associated with reception time. Messages are maintained in the buffer for certain period of time.

Table 2. Recapitulation of structures used for semantic refinement.

5.1.1. Components Hosted by Nodes

Components are deployed on nodes which are interconnected by a network infrastructure. Multiple components can be deployed on a single node. Communication model considers these facts because the situation of components on a single node is different from those separated by the network. Single node components **do not communicate** together[2], but they share some local information such as the position in the network topology or received messages. When dealing with component extermination we restrict our considerations to only two cases – either the node is running correctly with all components able to operate or the node is out of execution.

For this purpose we define a deployment relation $Dep \subseteq \mathbb{C} \times \mathbb{C}$, where $Dep(C_i, C_j)$ means that components C_i and C_j are deployed on the same node. The relation is an equivalence and divides the set of components $\mathbb{C}$ into a set of nodes. Notation $[C]_{Dep}$ (shortly $[C]$) is used to refer to a set of components deployed on the same node as C. We also interchange the set $[C]$ with the physical node hosting the components.

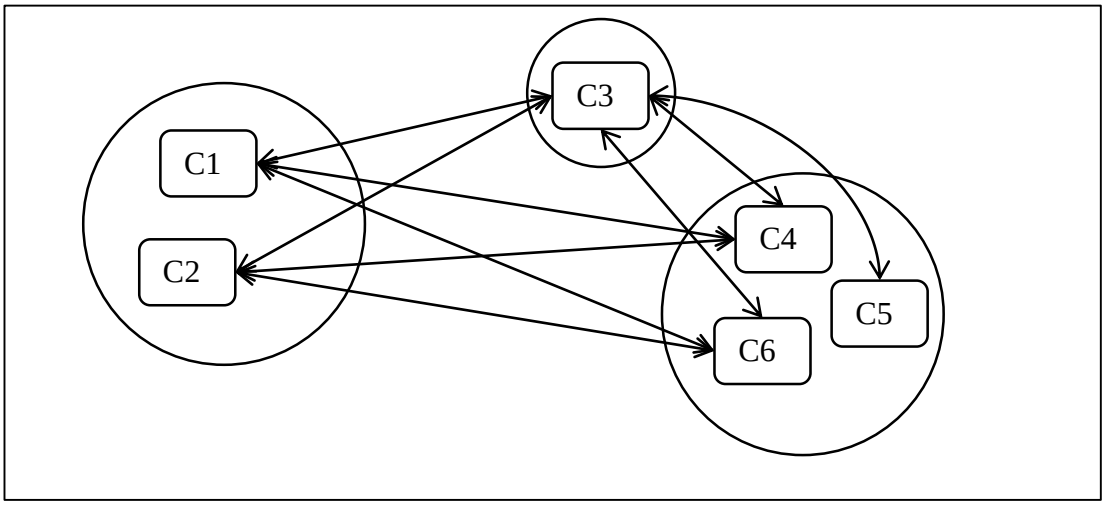


Figure 5. Components deployed on nodes. We have omitted several connections between [C1] and [C4] to preserve the clarity.

Rule 1. No local communication: Components on the same node do not communicate using the queues.

$$\forall (a, t) \in T: a = Q_C^{C_i}.enqueue(V_{C_i}) \Rightarrow C_i \notin [C]$$

[2] Queues associated with these nodes are never used locally by other component on the same node.

Now we consider only communication between nodes. If there is a message coming from node $[C_i]$ to $\left[C_j\right]$ it is available to all components in $\left[C_j\right]$. To simplify the view of the system we define for each node $[C]$ a set of communication queues $\{Q_{[C]}^{[C_i]} | \forall C_i \in \mathbb{C}: C_i \neq C\}$ where each one wraps all queues used to communicate with components on other node. In Figure 5 queue $Q_{[C_1]}^{[C_3]}$ wraps components' queues $Q_{C_1}^{C_3}$ and $Q_{C_2}^{C_3}$. The wrapping queue has the same operations as the original one and calling an operation results in immediate call of the same operation in the underlying queues. Introduction of these modifications give us a single communication channel between nodes. The previous concept of a component hosted by a node and sending a message to other component hosted by a different node can be seen as applications using different ports, but still the message is delivered to the same address. Using Figure 5 as an example sending a message V_{C_3} from $[C_3]$ to $[C_1]$ is handled by $Q_{[C_1]}^{[C_3]}.enqueue\left(V_{C_3}\right)$ and this call results in immediate call of $Q_{C_1}^{C_3}.enqueue\left(V_{C_3}\right)$ and $Q_{C_2}^{C_3}.enqueue\left(V_{C_3}\right)$.

In order to allow reason about the modified version of the system we replace transitions of the queue automata by queue wrappers' operations.

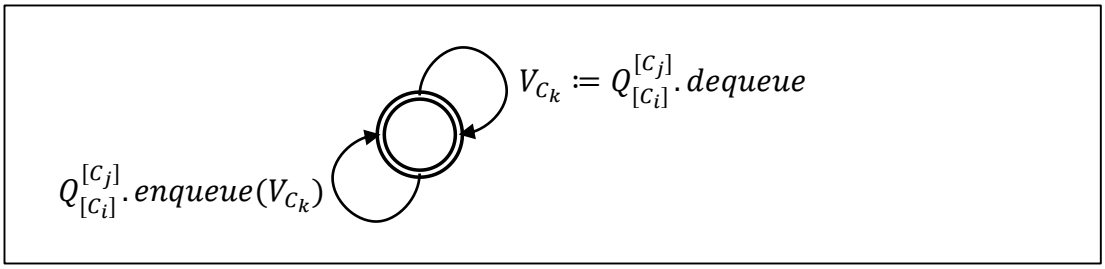


Figure 6. Modified queue automata employing queue wrapper.

Notice that in Figure 6 V_{C_k} is a message sent from component $C_k \in \left[C_j\right]$. This modified system generates a different execution trace $\dot{T} = (a_1, t_1), (a_2, t_2), \ldots$. We use this trace to put constrains on the original trace T.

Rule 2. Shared node interface: a message sent to a node is sent to all components at once hosted by that node and is available to all of them at the same time.

$$\forall (a_x, t) \in \dot{T}: a_x = Q_{[C_i]}^{[C_j]}.enqueue\left(V_{C_k}\right) \Rightarrow$$

$$\Rightarrow \forall C \in [C_i]: \exists \left(a_y, t\right) \in T: a_y = \ Q_C^{C_k}.enqueue\left(V_{C_k}\right)$$

$$\forall (a_x, t) \in \dot{T}: a_x = V_{C_k} \coloneqq Q_{[C_i]}^{[C_j]}.dequeue \Rightarrow$$

$$\Rightarrow \forall C \in [C_i]: \exists (a_y, t) \in T: a_y = V_C^{C_k} \coloneqq Q_C^{C_k}.dequeue$$

The following figure illustrates how individual queues are merged into single node connection in the modified system.

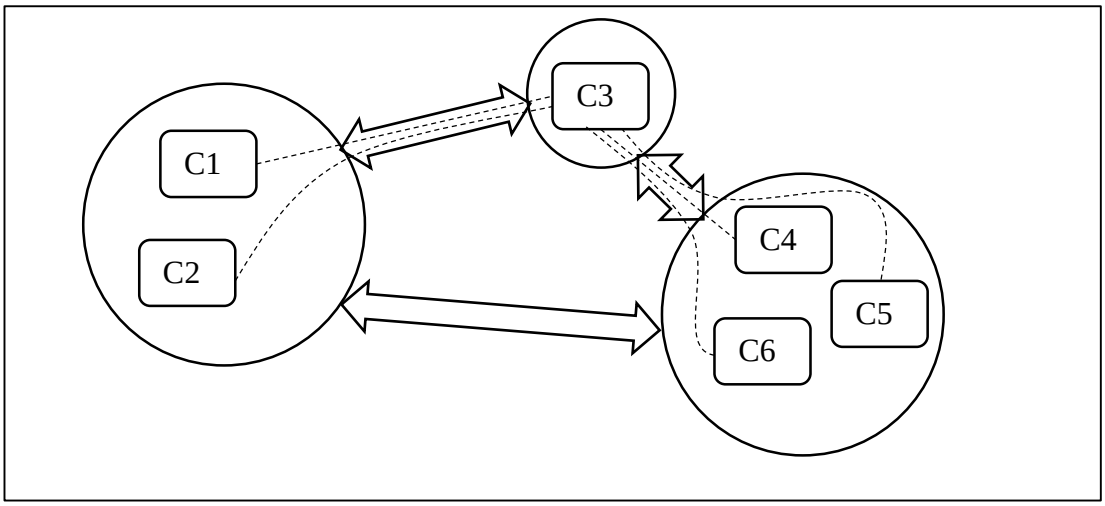


Figure 7 Connections between components are joined into node connections. Connections between components of [C1] and [C4] are skipped for clarity.

5.1.2. Network Topology

As stated in the previous Chapter nodes are interconnected with an infrastructure. The communication model must consider the actual network topology evolving over time and the fact that some nodes could be directly unreachable. In order to capture the network topology we define a function

$$Net: 2^{\mathbb{C}} \times 2^{\mathbb{C}} \times Time \rightarrow Time \cup \{\infty\}$$

The Net function indicates whether two nodes are able to communication at specific instant of time i.e. whether they are currently in the range of the radio broadcast device or linked by wired connection. Here we use the symbol $2^{\mathbb{C}}$ as a set of all subsets of $\mathbb{C}$ where a subset indicates components deployed on one node. Specifically $Net([C_1], [C_2], t_1) = t_2; t_2 \in Time$ means that $[C_1]$ can send a message to $[C_2]$ at time t_1 and $[C_2]$ will receive this message at time t_2. For the Net function it always holds $t_1 \leq t_2$. If $t_2 = \infty$ the message from $[C_1]$ won't be delivered. Now we

refine DEECo semantics to only allow communication when it is supported by the underlying network infrastructure.

Rule 3. Network topology: Communication is only allowed between nodes interconnected by the underlying network infrastructure. More specifically, a message can be sent and received only when the Net function returns valid reception timestamp.

$$\exists (a_x, t_x) \in \dot{T}: a_x = Q_{[C_i]}^{[C_j]}.enqueue(V_{C_k}) \wedge Net([C_i], [C_j], t_x) = t_y \Leftrightarrow$$

$$\Leftrightarrow \exists (a_y, t_y) \in \dot{T}: a_y = V_{C_k} \coloneqq Q_{[C_i]}^{[C_j]}.dequeue$$

From here the internal implementation of node queue will be slightly modified to imitate behavior of a priority queue, where the priority is the result of Net function. This concept allows to model situation where two messages are delivered in a reverse order comparing to the times when sent. Also sending a message to an unreachable node or a lost message will result in actual send action of the node, but the message will stay in the queue forever as the result of Net function is infinity.

We specifically consider deployment on MANET network which are a little bit specific in the way the communication is performed. The content is not directly addressed to the recipient and delivered by the underlying infrastructure. Nodes in MANET network only support message broadcasting i.e. a node is transmitting a message to all nodes within the wireless range of its radio device.

Rule 4. MANET broadcast: In MANET network message is transmitted or retransmitted by broadcasting it within the wireless range of the hosting node.

$$(a_x, t) \in \dot{T}: a_x = Q_{[C_j]}^{[C_i]}.enqueue(V_{C_k}) \Rightarrow$$

$$\Rightarrow \forall C \in \mathbb{C}: Net([C_i], [C], t) \in Time: \exists (a_y, t) \in \dot{T}: a_y = Q_{[C]}^{[C_i]}.enqueue(V_{C_k})$$

5.1.3. Message Fragmentation

The content of a message is potentially unrestricted with respect to the size. Therefore it must be fragmented when exceeds the maximum allowed size that can be transferred over the physical medium. Current DEECo implementation introduces a

two layers mechanism. The layer 1 handles message fragments while the layer 2 combines received fragments into a complete message or divides sent message into fragments. But from the component point of view the message always appears as a single object and it is received only when the entire content has arrived.

A message as defined in [5] is a partial function $V_C: K_C \rightharpoonup D$. In order to allow communication generated by nodes themself we conceptually consider each node to be a DEECo component. Node $[C_i]$ sending a message to $[C_j]$ is then translated into $Q_{[C_j]}^{[C_i]}.enqueue(V_{[C_i]})$ operation. Technically there is no component associated with a node, it can be rather understood as the instance of DEECo framework itself deployed on the node.

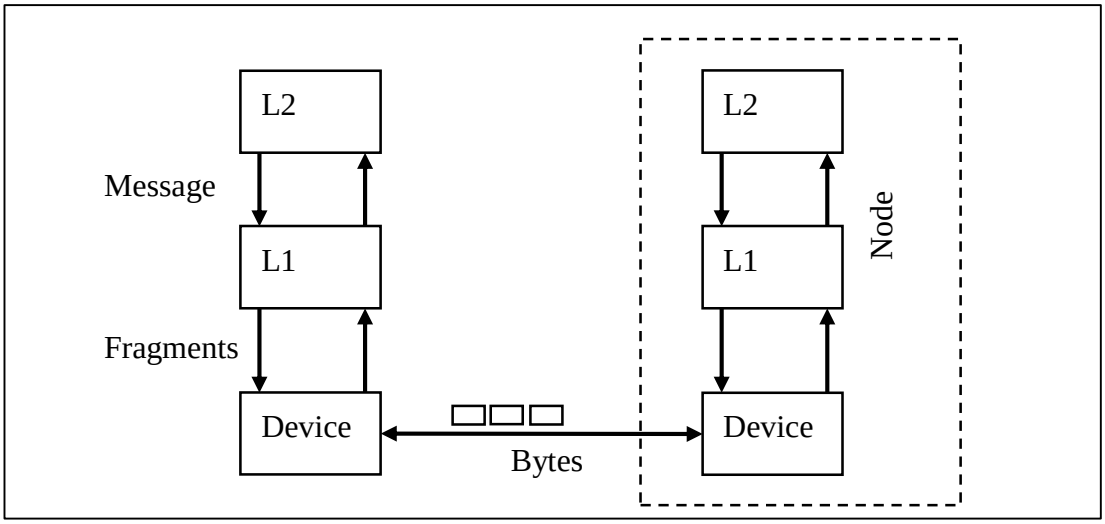


Figure 8. Network layers mechanism implemented by DEECo.

Given a message V_C we introduce a notion of message fragmentation as a set of partial functions $\{F_C^m \mid F_C^m: K_C \rightharpoonup D\}_{m=1}^n$ such that $V_C = \bigcup_{m=1}^n F_C^m$ and $\forall k \neq l: F_C^m \cap F_C^l = \emptyset$. In the queue automata (Figure 6) we replace the notations of a message V_{C_k} by symbol $F_{C_k}^m$ to indicate sending of message fragments. With respect to the knowledge valuation this is a perfectly valid step as $F_{C_k}^m \in \mathbb{V}_{C_k}$. The modified automata is shown on Figure 9. Trace generated by this modification is assigned with symbol $\ddot{T}$.

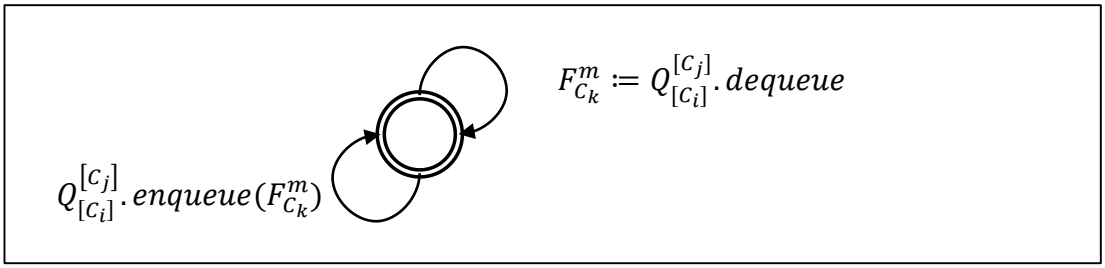


Figure 9. Modified queue automata passing message fragments.

A node can communicate using several type of devices such as radio broadcast device which is unaware of addressing and only allows for sending a message to anyone listening around. Other devices support direct sending of the message to selected nodes by address. Also the decision when to send a message is part of the protocol design. We capture these facts by defining a publish strategy predicate $Pub \subseteq [2^{\mathbb{C}}] \times [2^{\mathbb{C}}] \times \{\bigcup_{C \in \mathbb{C}} \mathbb{V}_C\} \times Time$, where $Pub([C_i], [C_j], V_{C_k}, t)$ indicates node $[C_i]$ sending a message V_{C_k} to node $[C_j]$ starting at time t.

Rule 5. Follow publish strategy: Messages are sent according to the publish strategy Pub keeping the timing requirement.

$$Pub([C_i], [C_j], V_{C_k}, t) \Rightarrow$$

$$\Rightarrow \forall m \in \{1, \dots, n\}: \exists (a_x, t_x) \in \ddot{T}: a_x = Q_{[C_j]}^{[C_i]}.enqueue(F_{C_k}^m) \wedge t_x \geq t$$

5.1.4. Node Storage Wrapper

We encapsulate the concept of queues associated with each node into a concept of node storage $St_{[C]}$ (see Figure 10) similar to hash table having the following operations:

- $St_{[C]}.put(V_{C_k})$ – Operation called by node $[C]$ makes message V_{C_k} available to other nodes which are interested.
- $St_{[C]}.get(C_k)$ – Operation called by node $[C]$ retrieves message coming from component C_k.

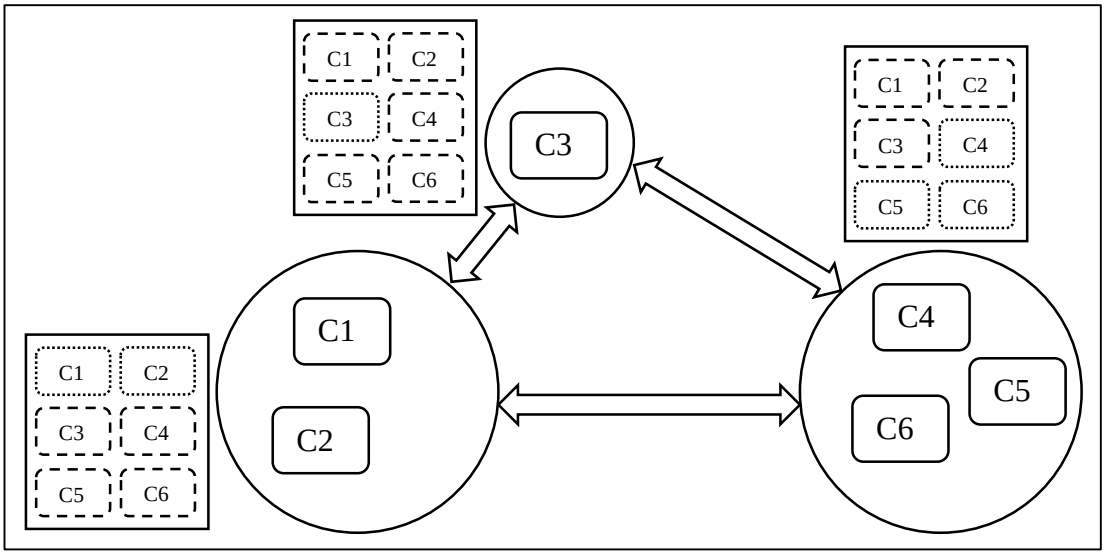


Figure 10. Node storage of individual nodes manages messages of local (dotted line) and replica (dashed line) components.

Conceptually nodes in the system communicate by using the node storage. Upon sending a message V_{C_k} the node $[C_i]$ invokes operation $St_{[C_i]}.put(V_{C_k})$. On the other side node $[C_j]$ when there is a necessity of a message coming from C_i invokes operation $St_{[C_j]}.get(C_i)$.

For the last time we change the notation in the queue automata (Figure 9) because the nodes are no longer communicating using the queues but the node storage.

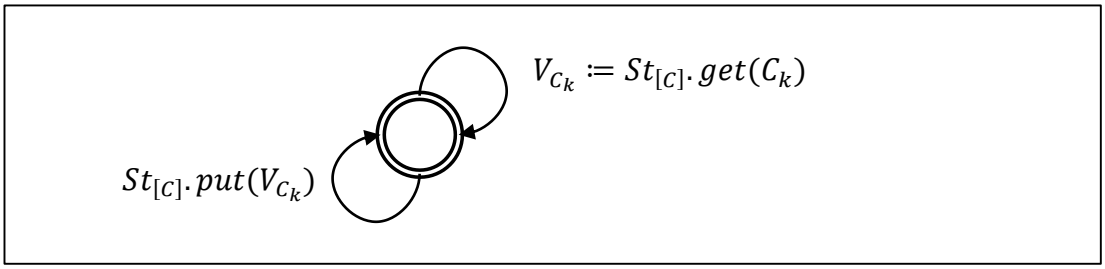


Figure 11. Modified "communication" automata using node storage instead of queue.

We use $\ddot{T}$ symbol to refer to traces generated by this modified system. For given message fragmentation $\{F_{C_k}^m\}_{m=1}^n$ of a message V_{C_k} operation semantics are defined in the following way:

Rule 6. Local message: Node storage wrapper returns immediately message which is locally accessible i.e. coming from component on the local node.

$$\forall C_k \in [C] \Rightarrow St.get_{[C]}(C_k) = V_{C_k}$$

Rule 7. Combine fragments: Central storage wrapper returns the message only after all fragments has been successfully delivered.

$$\exists (a_x, t_x) \in \ddot{T}: a_x = V_{C_k} \coloneqq St_{[C]}.get(C_k) \Rightarrow$$

$$\Rightarrow \forall m \in \{1, \dots, n\}: \exists (a_y, t_y) \in \ddot{T}: t_y \leq t_x, a_y = F_{C_k}^m \coloneqq Q_{[C]}^{[C_k]}.dequeue$$

Rule 7 is only applied to messages coming from component on a remote node. Notice that there is a difference between the messages V_{C_k} in Rule 6 and Rule 7 when interpreted as knowledge valuation. While V_{C_k} in Rule 6 is the current valuation of component C_k, in Rule 7 V_{C_k} is the C belief of knowledge valuation of component C_k.

Rule 8. Follow publish strategy: Messages are disseminated using a predefined strategy Pub which specifies message recipients.

$$\exists C_j \in \mathbb{C}: Pub([C_i], [C_j], V_{C_k}, t) \Rightarrow \exists (a, t) \in \ddot{T}: a = St_{[C_i]}.put(V_{C_k})$$

The $St_{[C_i]}.put(V_{C_k})$ operation does not specify recipients, it only requires the message to be disseminated and the predefined strategy takes care of the rest. This strategy will be defined further.

5.2. Gossiping Components in DEECo

One of the supporting challenges in DEECo design is an efficient and robust data dissemination across a distributed group of nodes. DEECo framework should be able to cope with the difficulty caused by the dynamicity and unreliability of the physical environment. Gossip-based protocol seems to be a promising solution for this complex task [8].

Current DEECo approach uses gossiping [9] to propagate its knowledge to any interested component in the network. A component which is interested in sharing its knowledge data regularly broadcasts on the network. Gossip broadcast protocols can be divided into push based and pull based. Push based gossip forward messages upon the first reception with certain probability. Contrary in pull based protocol the nodes exchange information about received messages. This exchange helps them realize which messages are missing and send an explicit pull request.

Gossip-based dissemination is suitable for MANETs naturally operating on radio broadcast medium. However there are several differences comparing to

communication in infrastructure networks[3]. Nodes in MANET are only capable of communication with immediate neighbors. If more distant nodes are addressed intermediaries[4] are required. On the other hand infrastructure network such as the Internet provide routing mechanism which is very expensive in MANET. Considering these facts, when talking about gossip, we will distinguish between MANET and infrastructure networks.

Resuming mentioned aspects of gossip communication we are going to address four cases:

- Push based gossip in MANET
- Pull based gossip in MANET
- Push based gossip in infrastructure network
- Pull based gossip in infrastructure network

Taking a more general point of view of the system we state that communication is performed by sending messages. Content of these messages is not only the component knowledge, as it is obvious, but also other kind of data such as auxiliary messages supporting the functionality of the protocol. As an example consider ACK packet in the TCP protocol which is necessary to establish a TCP connection, but is not part of the transferred data.

5.2.1. Gossip Communication Model

In a gossip-based protocol all nodes in the system collaborates on the data dissemination by actively retransmitting received message based on some well-defined strategy. The retransmission decision is based on node local information only. Let us define a predicate

$$Rt_2\left([C_i], V_{C_j}\right)$$

which indicates whether a message V_{C_j} should be retransmitted or not by node $[C_i]$. We left the internal definition or Rt_2 predicate unknown by now as there are many possible strategies, but it may be specified in the future. Some of commonly used strategies employ information such as number of received copies of the same

[3] Also referred to as IP-based networks
[4] Relay nodes

message, current GPS position, radio signal strength indicator (RSSI), etc. Current DEECo implementation uses a strategy based on probabilistic retransmission.

Rule 9. Periodic publish: knowledge of each component $C_i \in \mathbb{C}$ is periodically with a period R_K and fixed delay offset o published by the hosting node to other nodes in the system.

$$\exists (a,t) \in \ddot{T}: a = St_{[C]}.put\left(V_{C_i}\right) \Rightarrow t = o + n.R_K, n \in \mathbb{N}_0$$

Rule 10. MANET gossip: In MANET network message fragments are transmitted or retransmitted by broadcasting it within the wireless range of the hosting node.

$$(a_x,t) \in \ddot{T}: a_x = Q_{[C_j]}^{[C_i]}.enqueue\left(F_{C_k}^m\right) \Rightarrow$$

$$\Rightarrow \forall C \in \mathbb{C}: Net([C_i],[C],t) \in Time: \exists \left(a_y,t\right) \in \ddot{T}: a_y = Q_{[C]}^{[C_i]}.enqueue\left(F_{C_k}^m\right)$$

In MANET network message fragments are transmitted by broadcasting it within the wireless range of the node and therefore the publish strategy is determined by the physical environment. It holds:

$$Pub\left([C_i],\left[C_j\right],V_{C_k},t\right) \Rightarrow \forall C \in \mathbb{C}: Net([C_i],[C],t) \in Time: Pub([C_i],[C],V_{C_k},t)$$

Actually even stronger requirement holds. Not only is the message sent to all neighbors of the source node at the same time but the individual fragments at sent at the same time.

In the case of infrastructure networks such as the Internet a broadcast is not possible. Data is rather disseminated by communication with nodes $Rcp_{[C]}(t) \overset{rnd}{\subseteq} Peer_{[C]}(t)$ where $Peer_{[C]}(t)$ is a set of nodes known by $[C]$ at time t and $Rcp_{[C]}(t)$ is a random subset formed by random generator with seed argument t.

Rule 11. IP gossip: In infrastructure network node $[C]$ transmits or retransmits a message to randomly selected subset of known nodes.

$$Pub\left([C_i],\left[C_j\right],V_{C_k},t\right) \Rightarrow$$

$$\Rightarrow \forall C \in Rcp_{[C_i]}(t): Pub([C_i],[C],V_{C_k},t)$$

Notice that the Pub predicate indicates that the message sending process is started at time t, but may take time to be finished. Rule 11 requires a message to be sent to a set of nodes at specific time, but it is actually sent individually starting at the specified time instant.

Rule 12. Follow the retransmission strategy: Each node upon message reception from other node can decide to retransmit it. The decision is based on a strategy involving only node local information and the message being retransmitted.

$$\exists (a_x, t_x) \in \ddot{T}: a_x = V_{C_k} \coloneqq St.get_{[C]}(C_k) \wedge Rt_2([C], V_{C_k}) \Rightarrow$$

$$\Rightarrow \exists (a_y, t_y) \in \ddot{T}: a_y = St.put_{[C]}(V_k), t_y \geq t_x$$

Rule 13. No circular retransmission: In order to prevent from circular retransmission it is forbidden to resend message already retransmitted before.

$$\exists (a_x, t_x) \in \ddot{T}: a_x = St.put_{[C]}(V_k) \Rightarrow$$

$$\Rightarrow \forall (a_y, t_y) \in \ddot{T}: t_x \leq t_y : a_x \neq a_y$$

5.2.2. Bounded Gossip Extension

DEECo component model is suitable for designing systems at architectural level without considering the deployment aspects such as the communication infrastructure, component distribution or number of instances. We are able to reason about the components and ensembles in isolation. On the other hand introducing domain knowledge at the architecture level can significantly improve performance.

Gossip-based protocol successfully meets the requirement for knowledge dissemination but may cause a performance issue. Particularly in realistic environment the reach of the network is potentially unlimited and unrestricted gossip could cause a serious problem. DEECo is extended at the architectural level by the concept of *communication boundary* [8] to permit effective functionality at the communication level – gossip dissemination. The communication boundary specifies whether particular knowledge should be rebroadcasted or not. This mechanism allows to disseminate knowledge data in a specific geographical or other-way specified location.

Nodes included in the boundary represent an over-approximation of nodes interested in disseminated knowledge data i.e. those that satisfies the membership condition.

We denote node knowledge valuation $V_{[C]}$ as the union of knowledge valuation of all components

$$V_{[C]} \coloneqq \bigcup_{C_i \in [C]} V_{C_i}$$

The $\mathbb{V}_{[C]}$ symbol is used for all possible knowledge valuation of component hosted by node $[C]$.

Let us define a predicate $Bnd = \{\mathbb{V}_{[C]} \times \mathbb{V}_{C_k} | C_k, \in \mathbb{C}, C \neq C_k\}$, where $Bnd(V_{[C]}, V_{C_k})$ indicates whether message V_{C_k} received by node $[C]$ could be retransmitted.

Notice that Bnd predicate is a concrete instance of retransmission strategy where the knowledge valuation of all components hosted by the node is considered as the local information the strategy is based on. The specific about the Bnd predicate is that it is a design decision rather than definition of the protocol.

Rule 14. Boundary condition: a knowledge valuation can only be retransmitted while boundary condition Bnd holds.

$$\neg Bnd(V_{[C]}, V_{C_k}) \Rightarrow \neg Rt_2([C], V_{C_k})$$

5.2.3. Communication Boundary Example

In the example described in 3.2 we have defined components aware of their own position in GPS coordinates. Position of each component is disseminated to other components so that the application can display to the user a map with positions of other drivers or pedestrians – actors for short. Developer domain knowledge is that only actor's close enough should be displayed because there is a chance of joining other actor and travel together. Therefore knowledge valuation should not be disseminated too far from its source. This is when the communication boundary is useful.

Figure 12 shows a definition of the ensemble with communication boundary only retransmitting when position of the source component is within 2 kilometers. Notice the relation with Rt_2 predicate specified at design level.

```
18.     ensemble PositionDistributor:
        … /* membeship, knowledge exchange */
27.         communication boundary:
28.             (sender: PositionAware, node: NodeKnowledge)
29.             ∃ comp in node.components:
30.                 comp is PositionAware &&
31.                 |sender.position, comp.position| < 2 km
32.         scheduling: periodic(1s)
```

Figure 12. Example of ensemble extension by communication boundary.

5.2.4. DEECo Groupers

In order to optimize communication in infrastructure network we need to be able to identify components forming a communication group and to prevent from sending knowledge valuation to uninterested nodes.

At the design level this requirement is satisfied by defining a partitioning function $Part: \mathbb{V}_C \rightarrow D$, where D is the domain of knowledge field values. Components with the same $Part(V_C)$ form a group. Communication groups are established by a specialized system component $G \in \mathbb{C}$ called *grouper* which upon knowledge valuation reception computes the value of $Part$ function and places the source component to a particular group. This task is handled by a grouper process and currently formed groups are part of grouper knowledge. This knowledge is then disseminated to other components using regular DEECo approach of knowledge valuation publishing. The grouper can be deployed on multiple nodes together with other components to prevent single point of failure. Each of them is responsible for specific part of the key space i.e. values of $Part$ function.

Grouper knowledge is composed of a set of tables $\{T_1, T_2, \dots, T_s\}$. Each table is a set of nodes' addresses $T_i \subseteq 2^{\mathbb{C}}$. This modification is intended for infrastructure networks where each node possess a unique address.

Rule 15. Partial publish: the knowledge valuation of the grouper is published by part containing individual address tables.

$$(a, t) \in \ddot{T}: a = St_{[G]}.put(V_G) \Rightarrow V_G \in \{T_1, T_2, \dots, T_s\}$$

Rule 16. Publish to groups: knowledge valuation of grouper component is published only on infrastructure network (no MANET) and only to members of particular group.

$$G \in \mathbb{C}, \{T_1, T_2, \dots, T_s\}, Pub([G], [C], T_i, t) \Rightarrow [C] \in T_i$$

Notice that Rule 16 also requires the $[C]$ node to be selected by the random subset.

$$Pub([G],[C],T_i,t) \Rightarrow [C] \in Rcp_{[G]}(t)$$

As a result grouper knowledge valuation is published to a random subset of components specified in the grouper knowledge.

Rule 17. Group join: each node interprets **the last** knowledge valuation of a grouper as the list of known peer nodes – $Peer_{[C]}$. Notice that the node can join multiple groups and therefore we use $\subseteq$ symbol instead of $=$.

$$Peer_{[C]}(t) \subseteq V_G : (a_x, t_x) \in \ddot{T}, a_x = V_G \coloneqq St_{[C]}.get(G), t_x \leq t \wedge$$

$$\wedge\ \forall (a_y, t_y) \in \ddot{T} : t_x < t_y \leq t : a_y \neq V_G \coloneqq St_{[C]}.get(G)$$

5.2.5. Groupers Example

We will now show at the running example how to use the grouper service to optimize the communication. It is possible that some of the drivers or pedestrians would have a more reliable connection to an infrastructure network. Communication groups will be established to connect only drivers and pedestrians marching in a similar direction. Figure 13 shows how groups are established using the information about the route. The pedestrians are only interested in drivers starting and going to the same location as they are.

```
18.    ensemble PositionDistributor:
19.         coordinator: PositionAggregator
20.         member: PositionAware
21.         group: coord => coord.start + coord.dest
22.         membership:
23.              (coord is Driver && member is Pedestrian) ||
24.                   (coord is Pedestrian && member is Driver)
25.         knowledge exchange:
26.              coord.positions += member.position
```

Figure 13. Example of communication groups

Nodes with grouper component require stable infrastructure connection and as the computation is more intensive on that node power consumption is also higher. It is therefore preferable that grouper service can be provided by the public transport buses which may also take part in the dissemination or by some infrastructure established by the town.

Figure 14 shows an example of nodes grouped by common destination. Persons working and therefore traveling to the same building are more likely to communicate with each other.

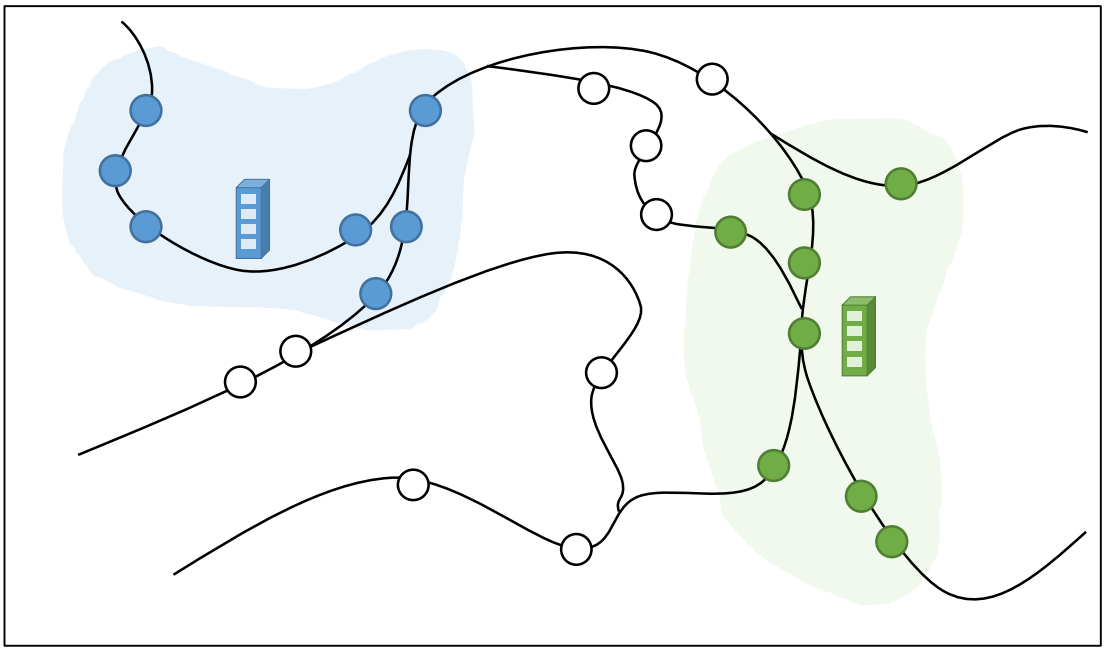


Figure 14. Example of vehicle nodes grouping.

5.3. Pulling Knowledge

Upon reception of a message the node stores its header into a buffer. Content of this buffer is regularly broadcasted to other nodes. Headers received from a neighbor are also put into the buffer and then broadcasted when necessary. Unlike in the case of knowledge valuation the headers are not retransmitted hence the total number of messages is linear with respect to the number of nodes.

Every header has a timeout specified and is removed from the buffer when expired otherwise buffers will grow without limits. It is expected that after this timeout either the message is delivered to all nodes or there are no more interested nodes which are able to receive it.

A PULL request is passed through the network until it reaches the node which has the required message V_C. This node does not retransmit the request any more but it broadcasts the requested message V_C. When some node receives V_C not for the first time it is ignored, because that node already had chance for retransmission. Otherwise V_C is retransmitted using regular gossip rules with probability.

Another timeout parameter needs to be specified for the pull request itself. It is possible that the source of a messages is no longer available and pulling will be stopped.

We have emphasized following protocol parameters which may have a significant impact on the performance or even correct functionality. The first part of Table 3 shows parameters of push-based protocol already implemented in DEECo. Second part summarize parameters identified by our analysis of pull-based extension.

Parameter	Description
PUSH	
Knowledge broadcast period	How often is broadcasted the local component knowledge.
Knowledge rebroadcast probability	Probability of knowledge rebroadcast upon its reception
IP broadcast count	To how many randomly selected known nodes will be sent a message when publishing on infrastructure network.
PULL	
Header broadcast period	How often is broadcasted the content of the headers buffer.
Pull request period	How often the node checks and requests outdated messages by pulling.
Local header timeout	After this timeout a message is considered outdated and can be pulled.
Global header timeout	After this timeout a message is considered either delivered to all interested nodes or the message source is unavailable.

Table 3. Parameters of gossip push-pull protocol.

If a message is outdated it needs to be requested explicitly. Because we have described our solution in a more general way stating that the nodes are communicating by sending messages and not knowledge valuation the principle of message pulling can be then applicable also for layer 1 (see Figure 8) and message fragments can be pulled as well. To allow such a feature we need to propagate in the system which message have been currently sent by which nodes.

On layer 2 the situation is a little bit different when considering the message content as knowledge valuation of a component. Although a component is a source of multiple messages with knowledge valuation during its lifetime from DEECo point of

view only the last one is interesting i.e. the most recent knowledge. Therefore we only need to propagate which components are involved in the communication rather than inform about every message coming from a particular component.

Notice that we only consider MANET network and possible improvements by exploiting infrastructure networks will be mentioned in 8.3.

Each node maintains a buffer $B_{[C]}$ of received messages where each message M is associated with a tuple (l_M, g_M, p_M):

- l_M – Time when the message was received by current node.
- g_M – Maximal known message reception time of M by any node in the system including current node.
- p_M – Flag indicating whether the message M was pulled by any other node.

Typically a message M stored in the buffer will be the knowledge valuation such as V_{C_k}. Expressing that a message is present in the buffer we state $M \in B_{[C]}$. If $M \notin B_{[C]}$ then $l_M = g_M = p_M = \emptyset$ (undefined).

We define following parameters as outlined in Table 3:

- R_K – Period of knowledge publishing.
- R_H – Period of message headers notification.
- R_P – Period of pull request routine.
- T_m – Message timeout. If message is not received after this timeout it is pulled.
- T_p – Pull timeout. If message is not received after this timeout the pulling is stopped.

In the following text we use specific notation for message having a particular content:

- V_H – a message with headers of other messages
- V_P – a pull request message
- V_C – a message with knowledge valuation of component C

The pulling protocol is defined by the following rules:

Rule 18. No header/pull retransmission: A message containing the headers and pull requests must not be retransmitted.

$$\forall C \in \mathbb{C}: \neg Rt_2([C], V_H) \wedge \neg Rt_2([C], V_P)$$

Rule 19. Received message: upon message reception its header is immediately added to the buffer.

$$(a, t) \in \ddot{T}: a_x = V_{C_k} \coloneqq St_{[C]}.get(C_k) \Rightarrow$$

$$\Rightarrow V_{C_k} \in B_{[C]} \wedge l_{C_k} = g_{C_k} = t$$

Rule 20. Publish headers: periodically with a fixed delay offset publish headers of all received messages i.e. the content of the node reception buffer.

$$\exists (a, t) \in \ddot{T}: a = St_{[C]}.put(\{M | M \in B_{[C]}\}) \Rightarrow$$

$$\Rightarrow t = o + n.R_H, n \in \mathbb{N}_0$$

Rule 21. Process headers: received message headers are added to the local reception buffer.

$$(a, t) \in \ddot{T}: a = V_H \coloneqq St_{[C]}.get([C_i]) \Rightarrow$$

$$\Rightarrow \forall M \in V_H: M \in B_{[C]}$$

Rule 22. Pull outdated messages: periodically with fixed delay offset publish headers of outdated messages. Outdated messages are those with expired message timeout.

$$\exists (a, t) \in \ddot{T}: a = St_{[C]}.put(\{M | M \in B_{[C]} \wedge t - l_M > T_m\}) \Rightarrow$$

$$\Rightarrow t = o + n.R_P, n \in \mathbb{N}_0$$

Rule 23. Receive pull request: messages contained in the pull request are marked with the pull flag.

$$(a, t) \in \ddot{T}: a = V_P \coloneqq St_{[C]}.get([C_i]) \Rightarrow$$

$$\Rightarrow \forall M \in V_P: M \in B_{[C]} \wedge p_M$$

Rule 24. Retransmit pulled message: Message containing knowledge requested by the pull request must be retransmitted.

$$(a, t) \in \ddot{T}: a = V_{C_i} \coloneqq St_{[C]}.get(C_i) \wedge p_{C_i} \Rightarrow Rt_2([C], V_{C_i})$$

Rule 25. Up-to-date messages: reception buffer stores headers only for a certain period of time. Old message headers are removed.

$$\forall t \in Time, \forall M \in B_{[C]}: t - g_M \leq T_p$$

5.4. Informal Proof of Semantic Refinement

In the previous Chapters we have defined several rules which determine traces of the refined semantic. We will now prove that the modified semantic is a valid refinement according the definition. Let us denote the new semantic as $\mathcal{C}$ and we will prove $\mathcal{C} \preccurlyeq \mathcal{D}$ (see Chapter 4.1), specifically that for each real-time trace $T_{\mathcal{C}}$ featured by semantic $\mathcal{C}$ there is a trace $T_{\mathcal{D}}$ featured by semantic $\mathcal{D}$ such that $C \in \mathbb{C}: V^C_{T_{\mathcal{C}}} = V^C_{T_{\mathcal{D}}}$.

All rules restrict possible traces and therefore for any trace in $\mathcal{C}$ there is an equivalent trace in $\mathcal{D}$. We divide our rules into several classes of trace restriction and we show that each type does not violate the refinement.

1. Rule stating that if there is some transition (a_x, t_x) then a condition must hold for this transition.
2. Modification of the system automata results in modified traces. Then we give a rule which joins these two types of traces i.e. if there is a transition (a_x, t_x) in the modified trace, than there is a transition $(a_x, t_x)'$ in the original one (or multiple ones). This rule can be applied multiple times, because once modified automata can be modified again.
3. A transition (a_x, t_x) or an arbitrary condition requires other transitions to exist in the same trace.
4. Rules which directly do not effect the traces but rather the behavior of some part of the system such as an auxiliary structure or a set. This rules usually effect rules in class 1 where the condition can be based on the auxiliary structure or set.

The following table shows classification of each of defined rules into one of the above restriction types.

Class	Rules
1	Rule 1, Rule 9, Rule 13, Rule 15, Rule 19, Rule 20, Rule 21, Rule 22, Rule 23, Rule 24
2	Rule 2, Rule 7
3	Rule 3, Rule 4, Rule 5, Rule 8, Rule 10, Rule 12
4	Rule 6, Rule 11, Rule 14, Rule 16, Rule 17, Rule 18, Rule 25

Table 4. Classification of rules according to the restriction of traces.

The rule class 1 only selects subset of traces generated by the original semantic and therefore by using these rules we get a valid refinement. The condition in class 1 rules can be more precisely specified by a rule from class 4, but it does not refine the semantic itself. A little more difficult is the case of class 2 and 3 because the rules require the trace to have particular transitions. So it can happen that in the original semantic there won't be such a trace.

For example Rule 2 requires the presence of $Q_C^{C_k}.enqueue(V_{C_k})$ transition which can be present at any position in the original semantic trace. And this is the case of all rules in class 2 and 3. So we only accept those traces which has the required transition while others will be excluded. These rules therefore only restricts the set of possible traces and the resulting semantic is a valid refinement.

6. Implementation

In this Chapter we describe technical details necessary to implement and evaluate the proposed solution.

6.1. JDEECo Simulation

For the purpose of validation and testing jDEECo provides a simulation infrastructure. The immanent part of it is the DEECo simulation which consists of execution of component processes, ensemble evaluation and knowledge publishing. The core of this simulation framework is a scheduler which enables to register tasks to be executed periodically or sporadically and as such allows to extend the simulation by custom functionality. We have integrated MATSim and OMNeT++[5] simulation in order to achieve realistic behavior of node mobility and communication at the network layer.

JDEECo implementation provides a simple interface for creating plugins. A plugin in this context is a class which can be registered to be automatically created and initialized by the simulation environment. The lifecycle of a plugin is started by the call of `init` method. Each plugin specifies a list of plugin classes it depends on and these are initialized before. During the initialization the instance of currently running simulation is accessible and the plugin is able to deploy components or ensembles or the register new tasks to the scheduler.

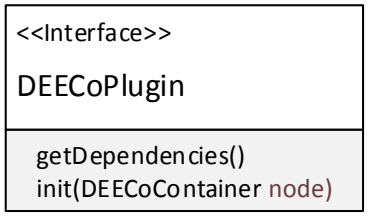


Figure 15. JDEECo plugin interface.

MATSim and OMNeT++ simulation support are implemented as plugins and can be used individually or together. If both simulators are used, they run in a separate threads and at each simulation step synchronized together with data exchange.

In our work MATSim is used to provide a multi-agent mobility simulation. In order to extract information about individual agents during the simulation such as position or speed we initialize the runtime with custom agent factory which for given

[5] Parts of the code were taken from existing jDEECo implementation.

person creates our custom implementation of MATSim agent. A person in this context is an individual from the population with some attributes and plan as specified in the configuration file `population` (Chapter 2.3).

JDEECo simulation uses a timer which provides current simulation time. MATSim plugin extends this timer to provide time according to the MATSim simulation, which is executing on its own. It allows for registering a custom callbacks fired at each simulation step. The timer can use these callbacks to intercept the MATSim simulation and synchronize with DEECo.

Existing jDEECo implementation also contains support for network communication by introducing a two layers mechanism as described in Figure 8. Each layer allows to register a strategy which is invoked any time a packet is received. In particular this is component knowledge on layer 2 or packet fragment on layer 1. Especially for the purpose of the communication protocol implementation we conceptually distinguish two categories of plugins.

- Strategy – plugins which register custom layer strategy. These are used to consume the communication and to perform custom message processing.
- Task – plugins which register custom task to the simulation scheduler. These are used to produce the communication and to send custom messages.

6.2. Gossip Dissemination Protocol

The basic principle of gossip as describe in Chapter 5.2.1 is in short: regularly publish data, retransmit upon reception with certain probability. This protocol is implemented using three plugins depicted on Figure 16. `SendKN`[6] plugin periodically takes knowledge of local node components and publishes them by broadcasting on MANET and/or to selected hosts on IP network. The knowledge is retrieved from `KnowledgeProvider` plugin as a deep copy. On the other side `ReceiveKN` plugin updates replica data of the remote node component. The `GossipRebroadcast` plugin on the same node can decide to rebroadcast or resend to selected hosts.

[6] Plugin class names end with "Plugin" suffix. We have omitted these in the description for short.

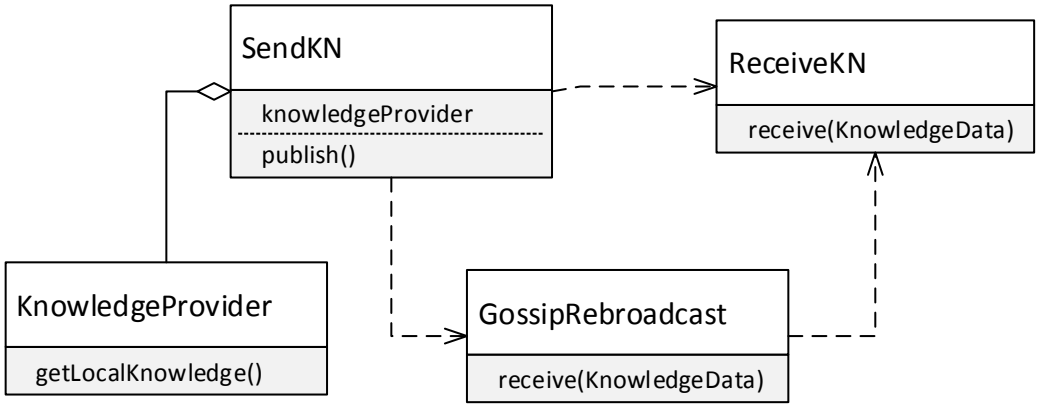


Figure 16. Plugins implementing gossip protocol communication. The arrows indicate knowledge passed over the network.

Especially for infrastructure network `SendKN` plugin uses implementation of `RecipientSelector` interface (see Figure 17) which provides a list of nodes addresses to which given knowledge should be sent. Different implementation of this interface could be used. For instance in the case of regular gossip on infrastructure network we have a `RangeRecipinetSelector` implementation, which selects and return a random set of IP addresses from given range such as 10.0.0.1-10.0.1.100. The same instance is also used by the `GossipRebroadcast` plugin to retrieve a list of addresses to which received knowledge should be resent.

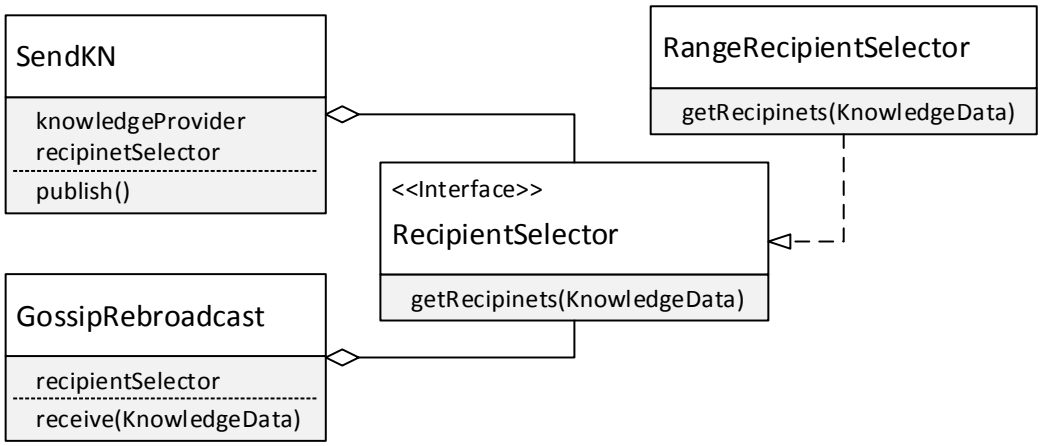


Figure 17. Recipient selector provides a set of hosts for knowledge publishing.

6.3. Grouper Component

Plugin infrastructure allows for deployment of custom components and ensembles. `GrouperServer` converts the hosted node into a grouper service which is implemented using DEECo component `GrouperServerComponent`. The knowledge is partitioned by a function, which is in our case a value of some knowledge field. If multiple groupers are used the domain of the partitioning function is divided between

them similarly to the key space partitioning in distributed hash tables (DHT) and each grouper process only knowledge with partition value belonging to its range. We emphasize that there is no implementation of node joining and leaving as required by the DHT algorithms.

The `processKnowledge` process executes periodically, calculates knowledge partition value and updates the register containing communication groups. For each partition value there is a set of node addresses forming a group. When knowledge of the grouper is published members of a group are assigned to `groupMembers` field.

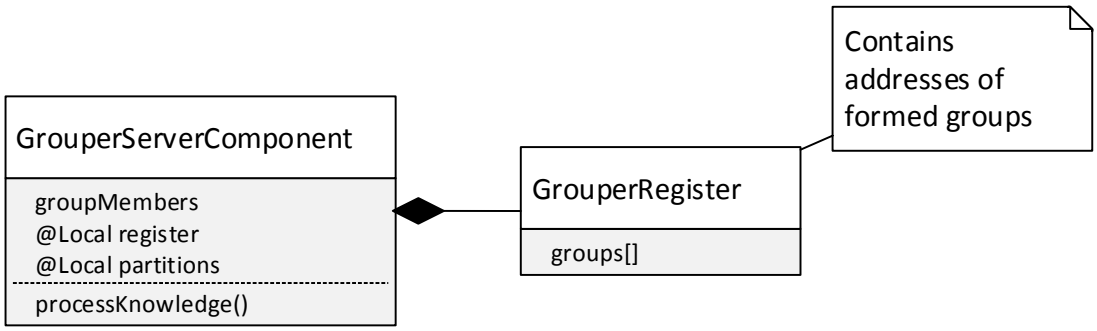


Figure 18. DEECo component implementing grouper service. Fields marked with @Local are excluded from component knowledge.

On the grouper node the `SendKN` plugin is initialized with different implementation of `RecipinetSelector`. `ServerRecipinetSelector` distinguish between grouper knowledge and knowledge of any other component. For grouper knowledge it returns a random subset of the `groupMembers` field, which contains the members of the communication group. For non-grouper knowledge behaves as the previous implementation such as `RangeRecipientSelector`.

On the other side on a non-grouper node, the `ClientRecipientSelector` returns a random subset from local node register. The register can be initialized with the address of a well-known grouper. But during the execution addresses of group members are added, so the node communicates only inside the group. Also this recipient selector prevents from resending knowledge by returning empty set of addresses, because all members of the communication group are aware of it and the content can be addressed to all of them directly from the source node.

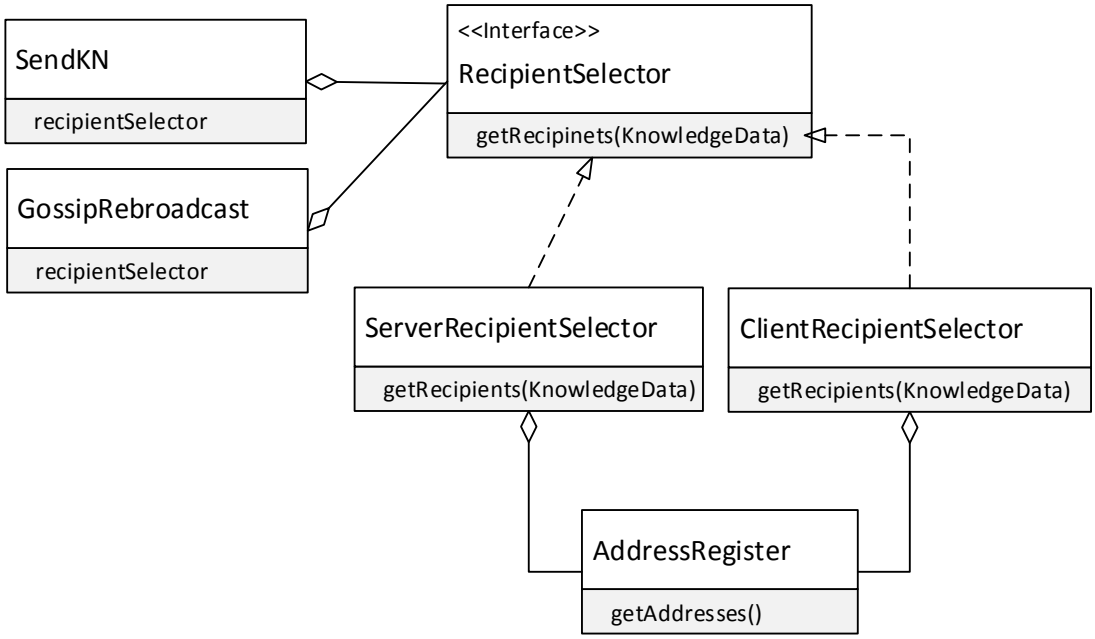


Figure 19. Configuration of recipient selector for the usage with groupers.

The grouper node has a register of known addresses, which is initialized with addresses of other groupers. Whenever the node receives knowledge which partition value is not managed by the current node, the knowledge is resend to addresses from this register i.e. to other grouper. The groupers pointing one to another forms an overlay network which can be customized by the initial addresses in the local registers. In the SCS example we have used a simple ring overlay network as shown in Figure 20.

In order to correctly enable grouper service it is necessary to support communication of grouper component with other nodes in the system. For this purpose we have implemented `GrouperClientEnsemble` and `GrouperClientComponent` deployed on non-grouper nodes only. The ensemble allows only knowledge exchange between server as a coordinator and client as member and the exchange itself copies group members from server to client knowledge. `GrouperClientComponent` has a process which automatically updates the local address register according to its knowledge. All of the above mentioned classes are instantiated by deploying `GrouperServerPlugin` on nodes dedicated to groupers and `GrouperClientPlugin` for the rest of nodes.

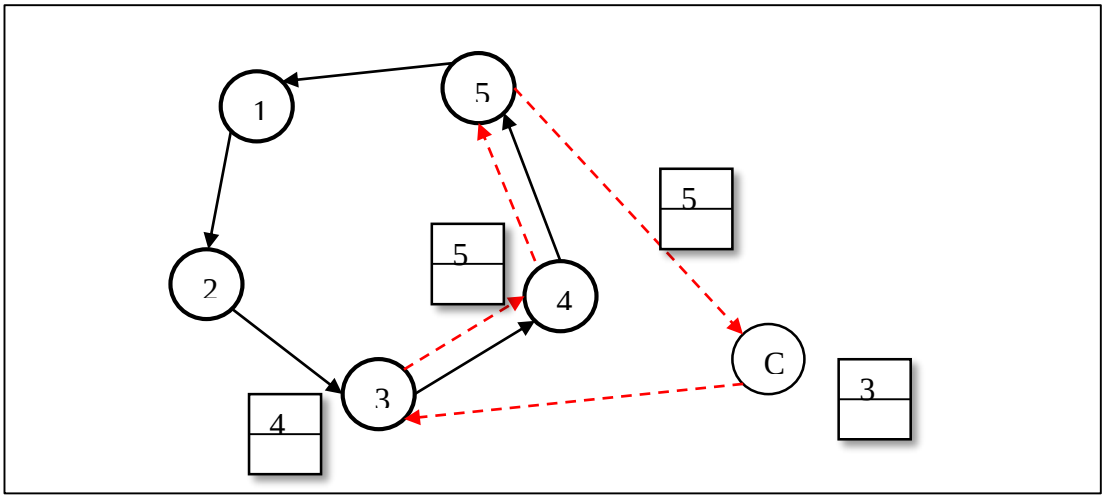


Figure 20. Groupers passing received knowledge using a ring overlay network.

Figure 20 shows an example of possible communication in a network with five groupers forming ring overlay. Component C sends knowledge to the only address in its register which is the grouper 3. Grouper 3 realizes based on the partition value that it is not responsible for this knowledge and resends it to the address from its own register, which is grouper 4. Grouper 4 performs exactly the same logic a passes the knowledge to grouper 5. Grouper 5 is finally takes the knowledge, stores address of component C node to its grouper register and next time it publishes communication groups it also send it to component C node, which updates its address register. Next time component C directly publishes its knowledge to grouper 5 which responds with updated list of group members.

6.4. Gossip PULL concept

Conceptually nodes are communicating by sending messages. Content of a message is the knowledge valuation. The messages are emitted by `SendKN` plugin as explained in 6.2. In order to allow for the pulling mechanism every node periodically broadcasts[7] headers of all received messages. For this purpose we have `ReceptionBuffer` plugin which stores required information. In particular header includes: source component **identifier**, reception time **locally** on the current node, the latest known **global** reception time of any other node and **pulled** flag.

During the system lifetime are sent multiple knowledge valuations of one component. We consider these to be a single message with several versions from which only the latest one is relevant. Therefore upon the next reception of knowledge coming

[7] In this Chapter we will consider only pulling in the MANET network.

from the same component we only update the local reception time and we do not add new record to the buffer. In order to preserve general concept reception buffer also stores information about messages coming from local components. The local and global reception time equals to time when the message was sent by `SendKN` plugin.

The content of `ReceptionBuffer` is regularly published by `SendHD` plugin. Notice that messages in the buffer have a timeout and too old ones are removed before publish. On the other side `ReceiveHD` plugin receives and process the headers by updating the content of its own `ReceptionBuffer`. In particular the global reception time is updated to the maximum value of global reception time on the current node and the remote node. This way the node can realize that there is some message received on other node but never locally. The important detail is that messages emitted by `SendHD` plugin are never rebroadcasted and the total number is linear with respect to the number of nodes.

`SendPL` plugin periodically checks for messages with local reception time beyond the defined timeout and broadcasts headers of these messages as a pull request. `ReceivePL` plugin on the other side receives such messages and sets the pull flag on it in the `ReceptionBuffer`. Messages with pull flag set are rebroadcasted with probability 1. Another possibility is to actively broadcast the messages upon the pull request. This behavior is possible to enable by `SendPulledKN` plugin.

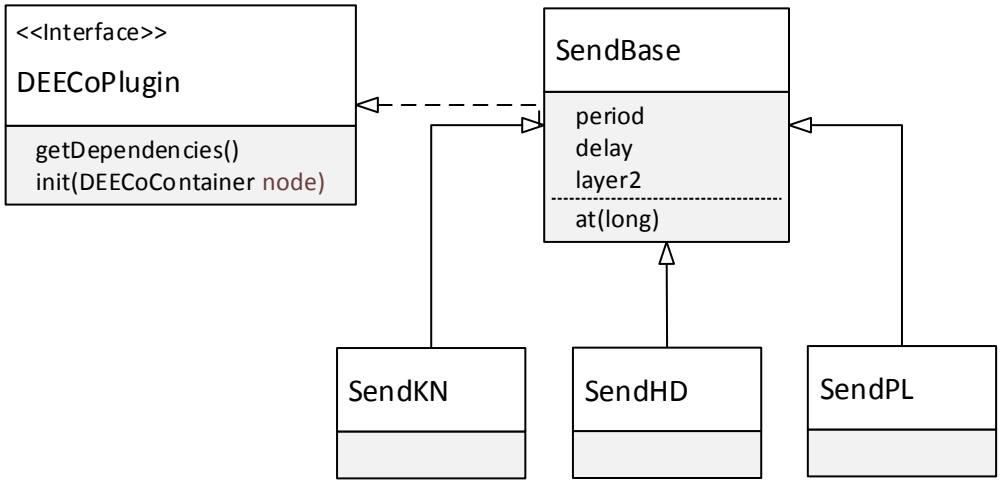


Figure 21. Inheritance hierarchy of plugins sending messages registering a custom task to the scheduler.

The counterpart plugins have a similar inheritance hierarchy but instead of task their register a custom layer 2 strategy. This strategy is called whenever a packet is received from the network.

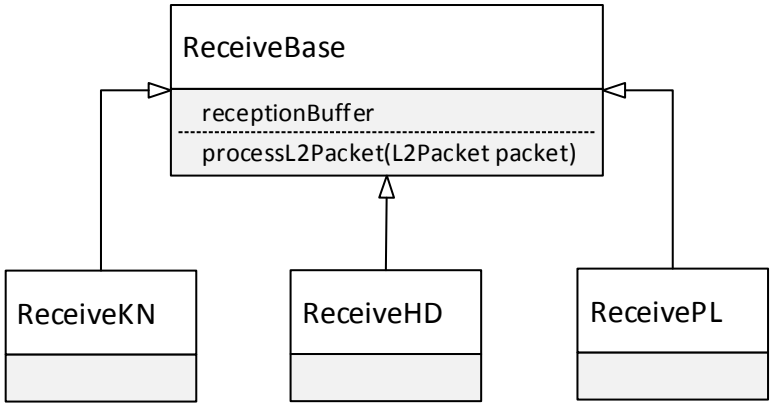


Figure 22. Inheritance hierarchy of plugins receiving messages registering a custom strategy at layer 2.

All of the above described plugins access the `ReceptionBuffer` for information about messages. For instance the `ReceiveKN` plugin updates local reception time whenever a message is received. On the other hand `ReceiveHD` updates global reception time, because these are reception times of messages on other nodes. Contrary to those plugins updating the reception times, `SendPL` uses local reception time to decide which messages are outdated. Complete overview is shown on Figure 23. The dash dot line indicates communication between instances of individual plugins on different nodes.

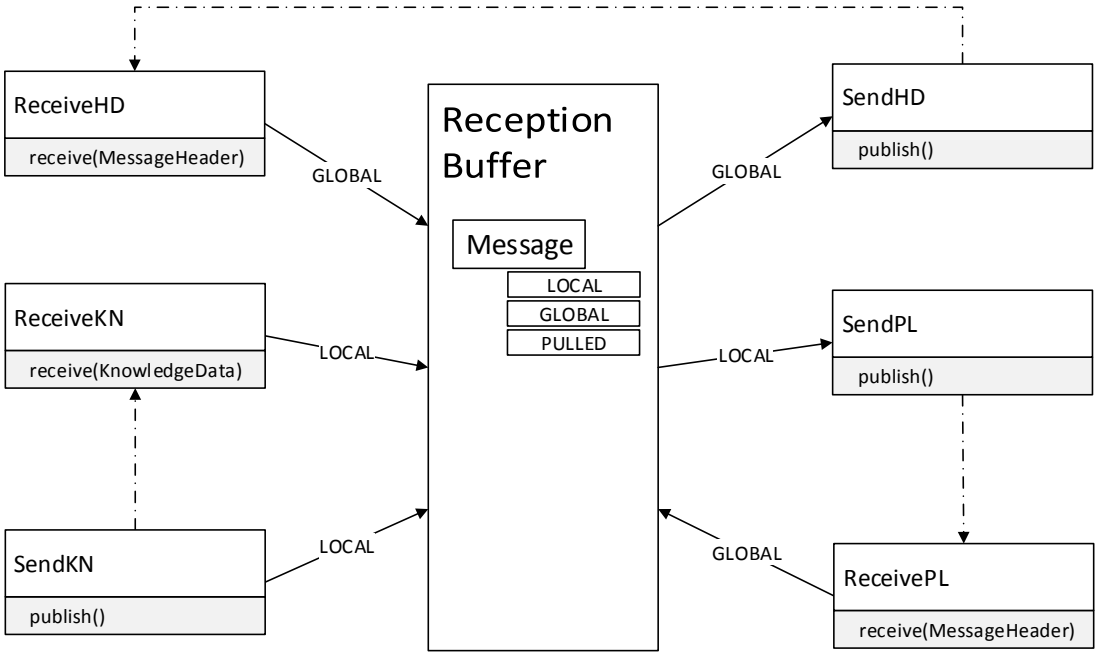


Figure 23. Different plugins accessing reception buffer.

7. Experiments and Evaluation

7.1. Smart Car Sharing

We have been simulating proposed communication mechanisms on a realistic scenario of the center of Berlin. The movement of individual drivers and pedestrians (or agents for short) is performed by MATSim itself. The cars are simulated by transit module with well-defined routes generated by our tool (Chapter 7.1.2). Pedestrians are just regular persons defined in the MATSim population file with prescribed activities. The movement of persons around the city is handled by MATSim and the person uses available public transport to get to the desired destination. The only junction point with DEECo simulation is the reading of actual actors' positions.

Specifically we have run the simulations with the following inputs: the center of Berlin of 8 km perimeter with 100 of car drivers and 100 walkers and the simulation is running for 15 minutes.

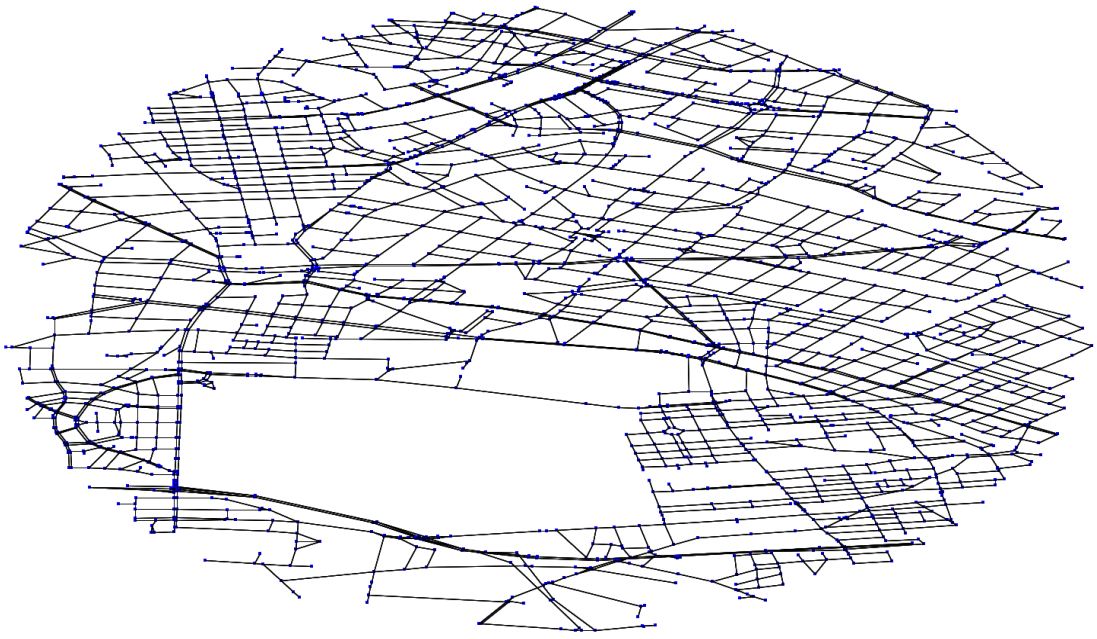


Figure 24. Exported map of the center of Berlin used by the simulation.

In the following Chapters we describe implementation details necessary to perform the simulations and to gather measurement data.

7.1.1. DEECo Application

We have tested implemented features on a realistic scenario described in Chapter 3. The application is decomposed into two components, for driver and passenger, and one ensemble for aggregation of positions. Basically, the functionality

of a driver and a passenger is very similar, both collect positions of actors in the other role and show them on a map. We have implemented this functionality in a parent component `Actor` from which driver and passenger inherit and specify the value of `Role` knowledge field. The base component has a process which regularly obtains current GPS location and updates the `Position` knowledge field. The `PositionAggregator` ensemble on the other hand updates list of currently visible actors. This ensemble only accepts knowledge exchange between `Driver` as a coordinator and `Passenger` as a member or vice versa. During the knowledge exchange the knowledge field `Actors` is updated. Knowledge dissemination of the ensemble is bounded by a two kilometers perimeter as there is no need to display actors too far away. Positions of other actors are removed from the list if they are not updated for a longer period of time, because they disappeared from the system or left the communication boundary and they are no longer interesting. This is acquired by the `updateActors` process of the component.

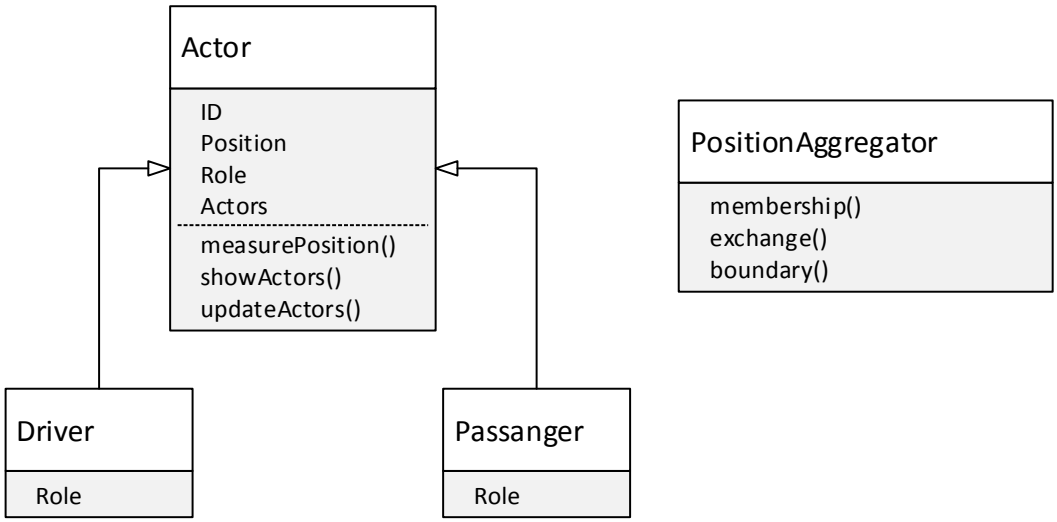


Figure 25. Decomposition of SCS Application into components and ensembles.

7.1.2. MATSim Generator

The concept of SCS application is very similar to public transport. There are cars traveling around the city and passengers using these cars to get to the desired destination. In order to allow for such a simulation we have used MATSim transit module. We generate the schedule of several transit vehicles which simulates the movement of drivers and each node in the MATSim network is a potential transit stop. The passengers are just regular MATSim population with generated schedule. This way we let MATSim to completely simulate behavior of the realistic scenario as

described in Chapter 3.1 – persons are traveling to their destination points using even multiple drivers to get there.

Simulation of smart car sharing application is supported by a generator tools which provides customizable generator of grid network or takes already existing network as an input and creates random population and traffic on it. Specifically this tool generates following MATSim configuration files:

- Grid transport infrastructure – network.xml
- Plans of person activities – population.xml
- Available vehicles used during the simulation – vehicles.xml
- Schedule and routes of the vehicles – schedule.xml
- Input configuration file of MATSim simulator referencing all previously mentioned files – config.xml

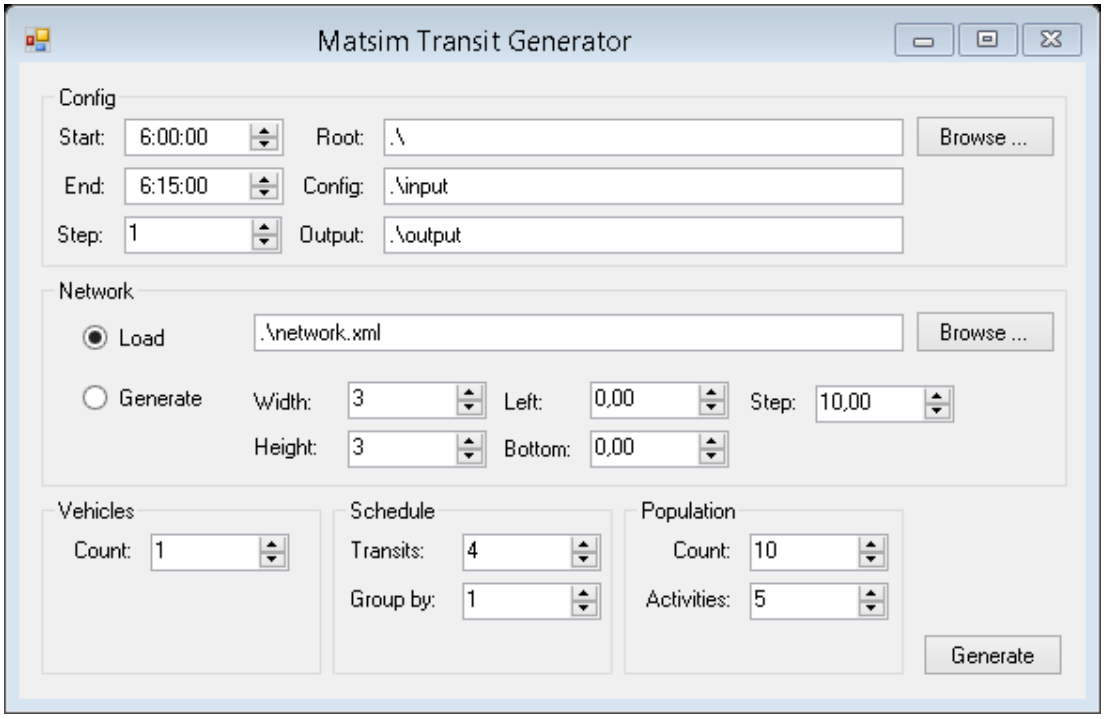


Figure 26. MATSim configuration file generator interface.

7.1.3. Simulation

As described above the implementation contains several plugins which can be configured to provide required platform with specific behavior. As this is a little more complicated task we provide a `SimConfig` class which reads given configuration file and setups the simulation accordingly. The experiments than consists of generating several configuration files with different protocol parameters, communication devices

or simulators and executing the demo application with each of these files. To automate these steps we have implemented a PowerShell script.

The output of the simulation is apparently the communication between nodes. For this purpose `RequestLogger` plugin intercept communication between layer 1 and layer 2 and each sent or received message logs to a file. This file is then processed by a summarization tool which gives us an overall view of the system communication. The `RequestLogger` has the ability to register additional parameters to be logged.

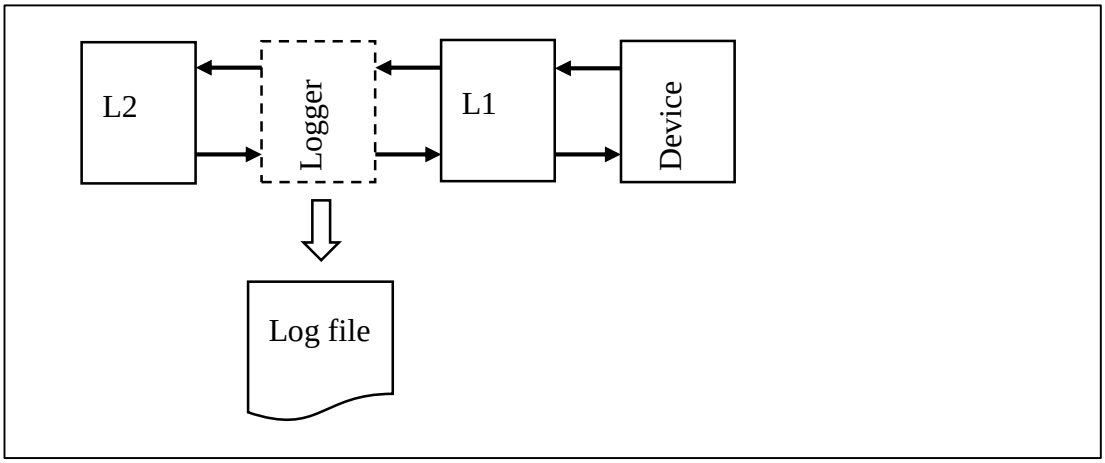


Figure 27. Request logger intercepts communication between layer 1 and layer 2.

7.2. Gossip in MANET

The first experiment consists of nodes equipped only with broadcast device. Every node publishes its knowledge with 5s period and rebroadcasts with certain probability, which is the input parameter of the protocol. Then we run the same simulation using communication boundary. The condition is based on the fact that nodes too far are not interesting – displaying them on the map for the user would not bring any added value. So the knowledge is rebroadcasted only in 0.5 km perimeter. The simulations has been performed with different rebroadcast probabilities. Figure 28 shows comparison of both methods and indicates decreased number of (re)broadcasts necessary to disseminate the knowledge when communication boundary is used.

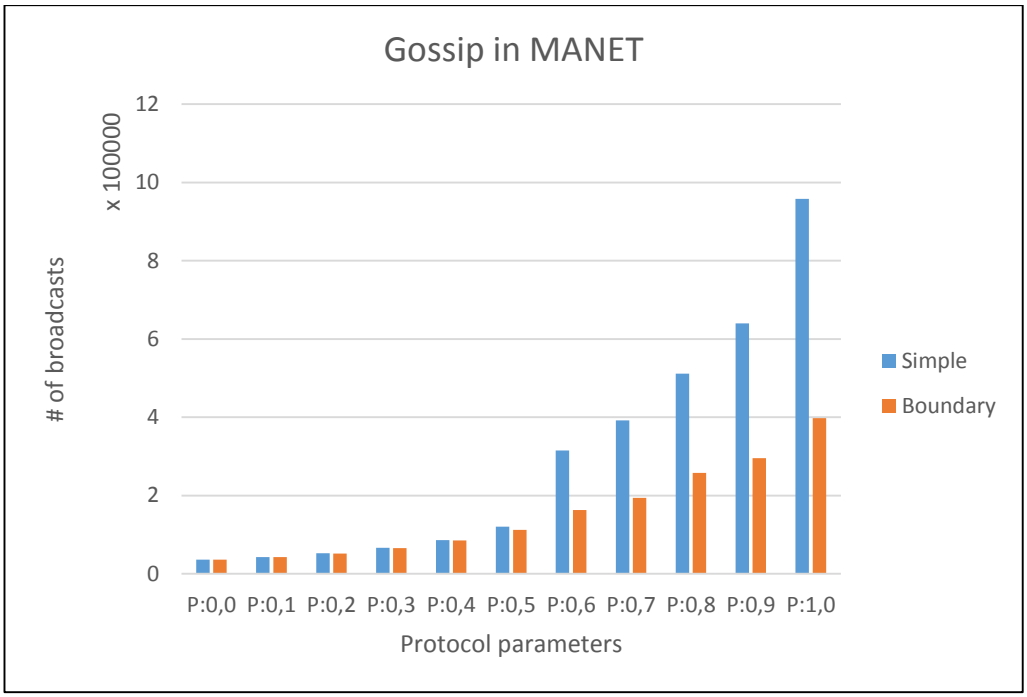


Figure 28. Regular gossip protocol in MANET network comparing to communication boundary. P = rebroadcast probability.

Parameters	Simple	Boundary
P:0,0	36 200	36 200
P:0,1	42 600	42 549
P:0,2	52 522	52 105
P:0,3	66 474	65 458
P:0,4	86 106	85 546
P:0,5	120 407	112 137
P:0,6	314 844	162 826
P:0,7	392 155	193 898
P:0,8	511 723	257 829
P:0,9	639 448	295 620
P:1,0	958 077	397 807

Table 5. Total number of sent messages using Gossip protocol in MANET network.

We have been also investigating how much is the knowledge outdate when delivered to the target node. Figure 29 summarize selected simulations with rebroadcast probability above 0.5 using a box plot and shows that when communication boundary is used the medium age of delivered knowledge is lower. This is caused by the fact that the knowledge in the protocol without boundary is delivered to many other distant nodes after several hops. Using the boundary also

improves the belief inaccuracy. Without it a message can be delivered a long time after is has been sent and the state of that component by that time already changed radically.

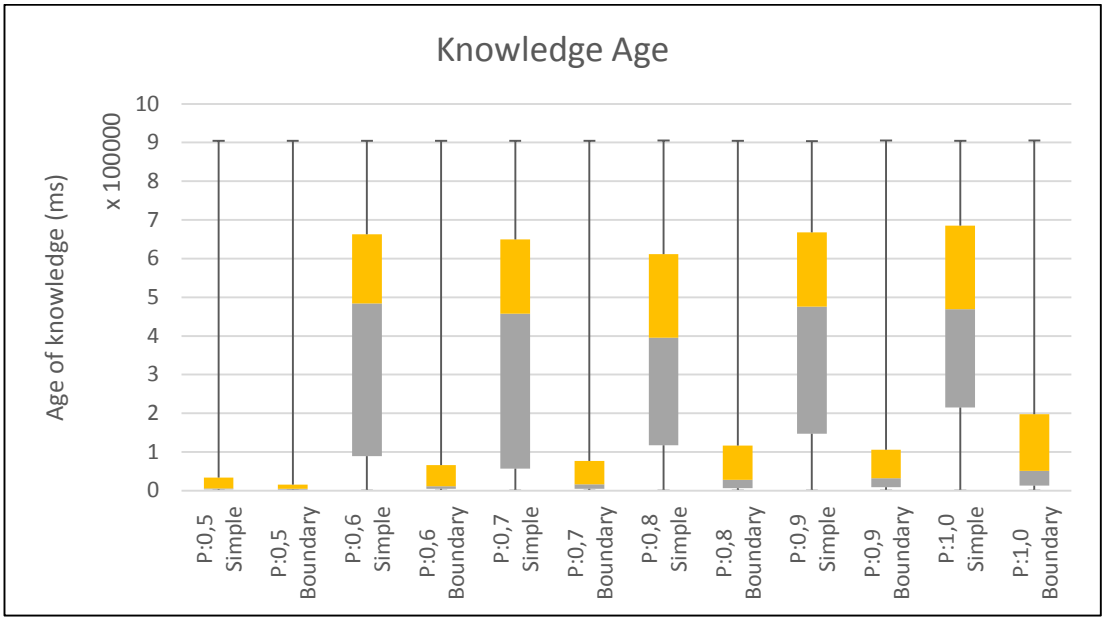


Figure 29. Comparison of knowledge ageing using Gossip protocol in MANET with and without boundary.

We have taken also another point of view on this experiment and we have calculated how many knowledge valuations of distinct components has been delivered to individual nodes. Figure 30 shows that the boundary slightly outperform simple version of the protocol when the probability is higher. This probably caused by too many collisions what results in undelivered (or partially undelivered) messages.

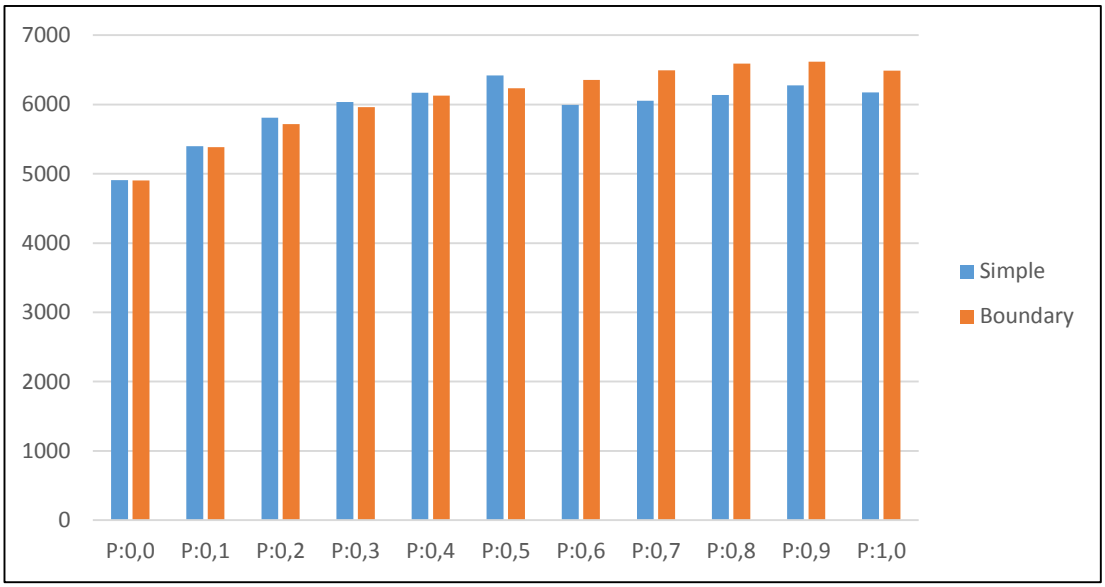


Figure 30. Number of different component's knowledge valuation delivered to individual nodes.

7.3. Gossip in Infrastructure Networks

In the infrastructure network rather than broadcasting, the knowledge is sent to a set of random nodes in order to disseminate the data. Upon reception the knowledge is retransmitted again to random set of nodes with certain probability. We have been simulating the behavior of this protocol changing the probability (0.0 – 0.5) and the size of random nodes set (1 – 3). After that we have exchanged the random set of nodes for the set of nodes provided by the grouper. Figure 31 shows comparison of both approaches and significant decrease of total messages necessary to disseminate the knowledge when using the groupers. In the second case groupers have been deployed to 8 nodes. The number of messages increases drastically with the increase of publish count.

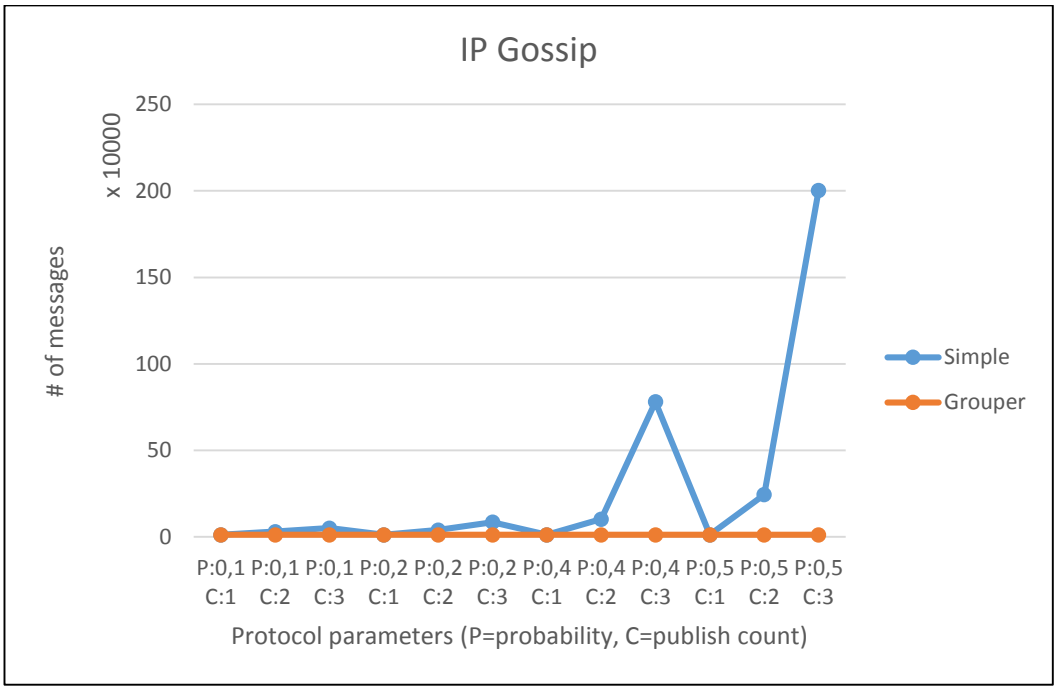


Figure 31. Comparison of regular IP gossip and gossip with groupers.

There is also a positive side effect of the grouper feature. The age of knowledge delivered to individual nodes is smaller as it is delivered to nodes which are interested in it and only to those. Figure 32 shows significant decrease of knowledge age when using the groupers. In the graph the results for simple and grouper version are alternating. It is also visible that by the increase of retransmission probability and publish count the age of knowledge is decreasing in the case of simple protocol version. With input parameters P:0.5 C:3 the age already achieves appropriate values, but the total number of messages is disproportionately high.

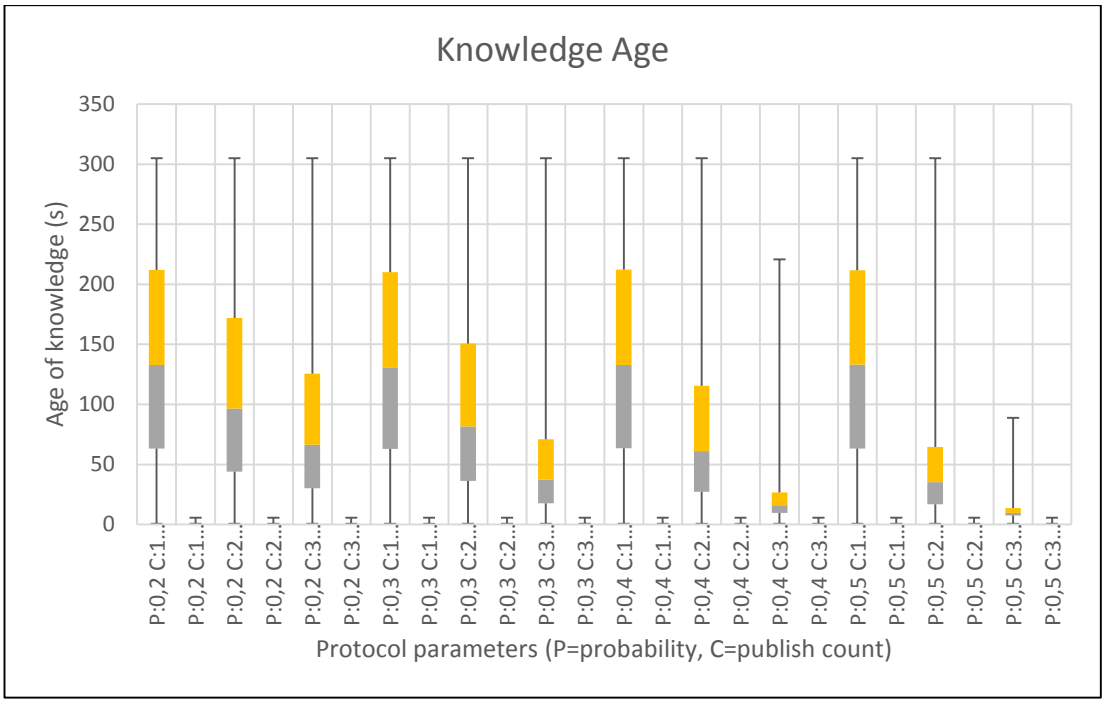


Figure 32. Comparing of knowledge ageing in regular IP gossip and gossip with groupers.

7.4. Pulling in MANET

The pulling mechanism has been simulated with the same inputs as in the experiment of gossip protocol in MANET network, except that the nodes had deployed necessary plugins. Different parameters of the protocols has been tested as outlined in Table 3. Consequently we have been investigating the dependency between individual input parameters considering the age of knowledge delivered to nodes.

For individual configurations of input parameters we have calculated first quartile, median and third quartile. Because there are many of them we report only on those giving reasonable results (this means those whose third quartile of time needed for knowledge delivery is less than 20 seconds). Further, to show necessary detail, we visualize only 1st, 3rd quartile and the median leaving out the minimum and maximum. Selected configuration has been order by median which is in range from 6 to 16 seconds (see Figure 33).

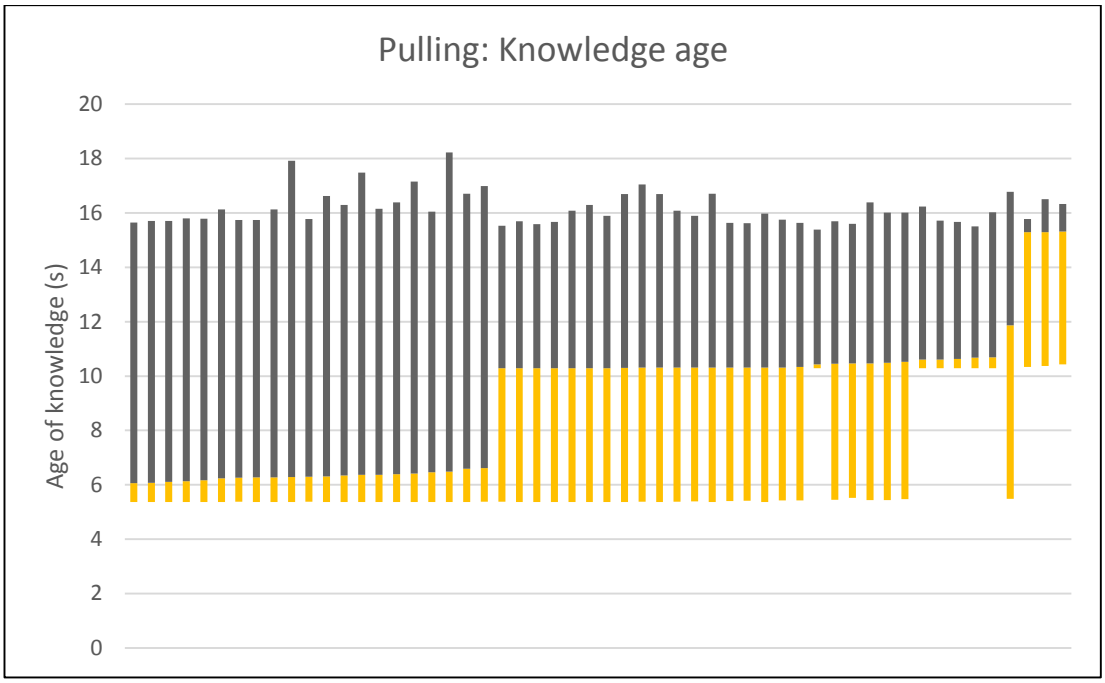


Figure 33. Age of knowledge using various input parameters ordered by median. We have omitted maximum and minimum for clarity.

From these we have taken only those configuration having the median around 6 seconds reasoning that publish period is 5 seconds such a value is still acceptable as delivery latency for the majority of the nodes. All of these configurations have probability parameter set to 0.9. The red line in Figure 34 on secondary axis indicates median of knowledge age without the use of pulling and with the same probability (see Figure 29 in Chapter 7.2). Table 6 summarizes the complete set of input parameters for selected configurations.

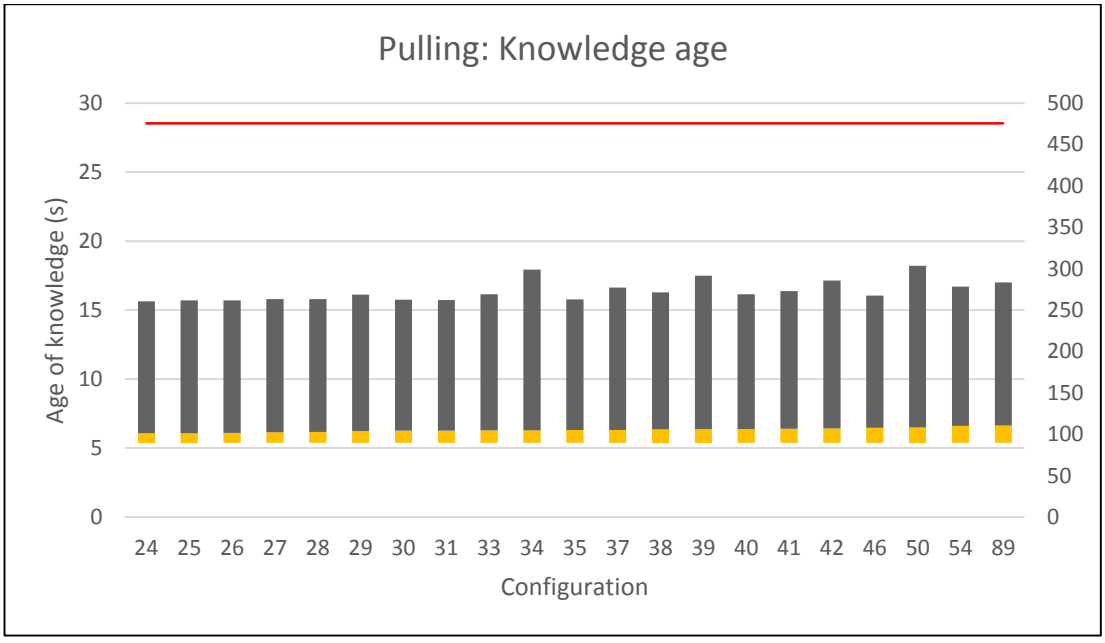


Figure 34. Knowledge age of selected configurations.

Configuration	HD (ms)	PL (ms)	P	L (ms)	G (ms)
24	5000	11000	0.9	9000	13000
25	5000	9000	0.9	9000	13000
26	5000	7000	0.9	9000	13000
27	5000	5000	0.9	9000	13000
28	5000	9000	0.9	11000	17000
29	5000	11000	0.9	13000	21000
30	5000	11000	0.9	11000	17000
31	5000	7000	0.9	11000	17000
33	5000	9000	0.9	13000	21000
34	9000	15000	0.9	11000	17000
35	5000	5000	0.9	11000	17000
37	5000	7000	0.9	13000	21000
38	7000	13000	0.9	11000	17000
39	7000	11000	0.9	13000	21000
40	5000	5000	0.9	13000	21000
41	7000	9000	0.9	11000	17000
42	9000	9000	0.9	11000	17000
46	7000	11000	0.9	11000	17000
50	7000	13000	0.9	13000	21000
54	7000	7000	0.9	11000	17000
89	7000	9000	0.9	13000	21000

Table 6. Input parameters of selected configurations (HD=header publish period, PL=pull request period, P=rebroadcast probability, L=message timeout, G=pulling timeout).

We have compared this approach with previously mentioned regular gossip. The median value for simple version is about 475 second while for the version using the communication boundary the median is 32 seconds. Our approach with properly selected inputs results in median knowledge age of 6 seconds.

Figure 35 show the total number of sent messages for selected configurations. Increasing the publish period of message headers slightly decreases the number of messages without significant impact on the knowledge age. The same argument holds for pulling period. Once the pull request is initiated it is very probable to be delivered and therefore it is not necessary to repeat it more frequently.

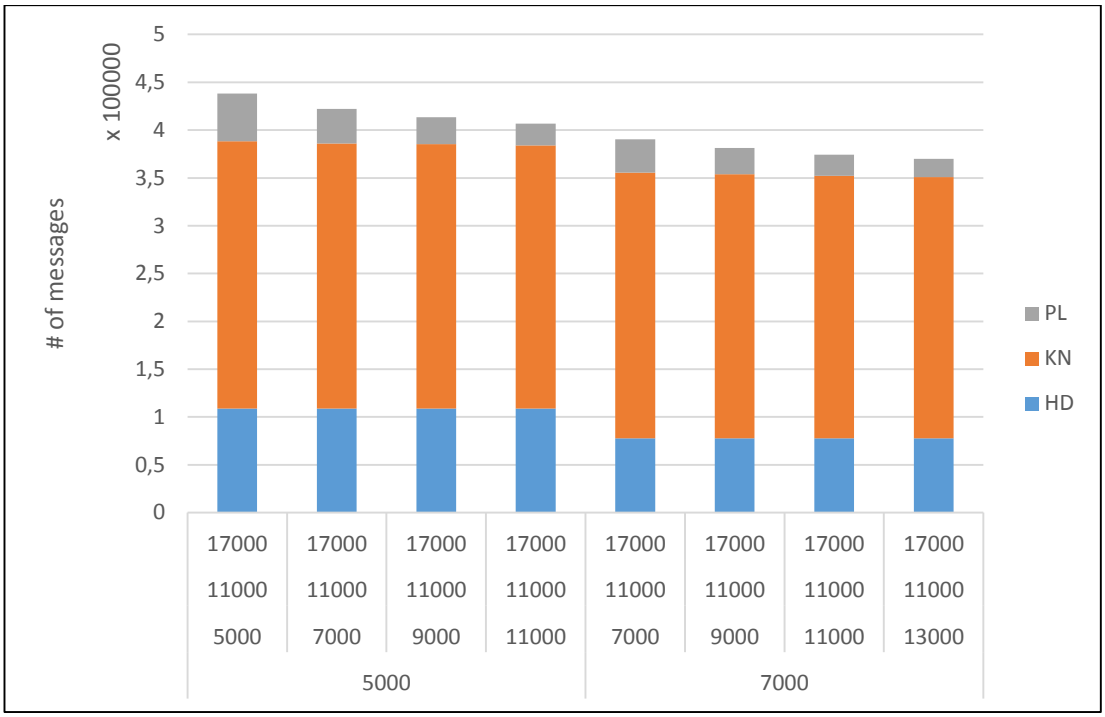


Figure 35. Number of sent messages for selected configurations (Protocol parameters from up down: pull timeout, message timeout, pull request period, message header period).

8. Discussion

8.1. Protocol Parameters

In Chapter 7.4 we have been investigating only reasonable combination of input parameters, holding the following rules:

1. Knowledge broadcast period < Message timeout < Pull timeout
2. Knowledge broadcast period ≤ Header broadcast period ≤ Pull request period

Ad. 1 It is reasonable that the pull request won't timeout before it is even initiated (Local header timeout < Global header timeout). Also the message should not timeout before it is even sent (Knowledge broadcast period < Local header timeout).

Ad. 2 It is not necessary to send notifications about message headers more frequently as the messages itself. The same thing holds for pulling.

The experiments show that it is reasonable to set message timeout approximately twice as publish period because the message can be delivered by the PUSH mechanism while PULL request wastefully is initiated. The message timeout expresses the acceptable latency with the following meaning: it is the half of the knowledge delivery latency. If the message is to be delivered in time after the timeout we need to reserve time for pull request and for the response as well.

If a message is not delivered at least the header should be. Therefore the publish header period should be the same or close to the publish knowledge period if a missing message is about to be discovered. Otherwise the message is pulled too tardily and the age is higher. This is also confirmed by the experiments showing good results only for header publish period set to 5 or 7 seconds.

The pulling period does not influence significantly the delivery latency and for simplicity can be the same as the publish header period. On the other hand setting it a slightly higher decreases the number of messages consumed by pulling as shown in Figure 35.

Summarizing observations mentioned above we give following rules for correct configuration of input parameters.

1. 2x publish knowledge period ≤ message timeout
2. 2x message timeout ≤ acceptable delivery latency

3. publish knowledge period = publish header period ≤ pull request period

Combining the first two rules gives: 4x publish knowledge period ≤ 2x message timeout ≤ acceptable delivery latency. Figure 34 confirms our observation: publish knowledge period is 5 seconds, message timeout is around 10 seconds and the 3rd quartile is less than 20 seconds.

8.2. Behavior in Typical Situations

Except the simulation experiments targeting rather quantitative properties of the implementation such as the number of messages or knowledge age we discuss also the protocol behavior in specific cases. For this purpose we have used sample network topology as shown in Figure 36 where only immediate nodes are accessible to each other. In this network the delivery of message from one nodes to another may require multiple hops. For instance a message from 1 to 6 requires 3 hops.

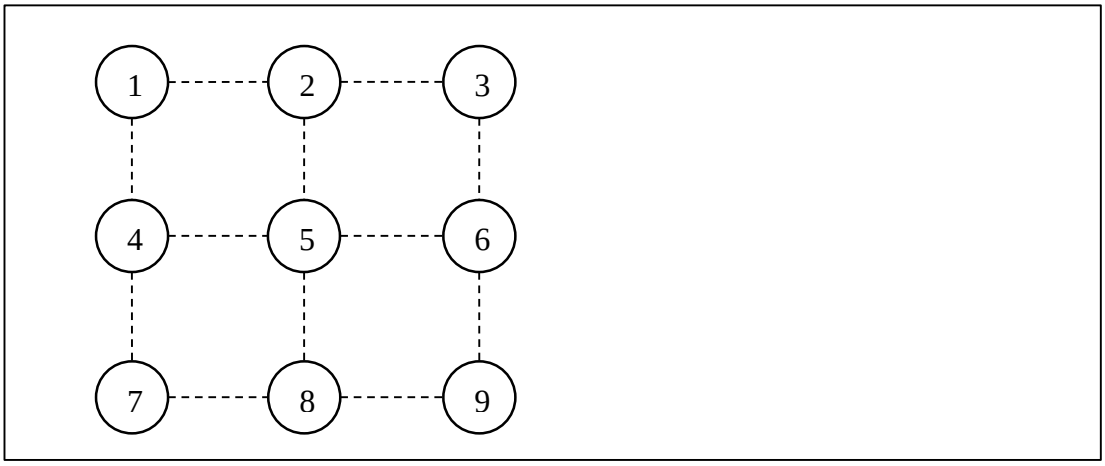


Figure 36. Sample network topology used in the use-case scenarios

8.2.1. Pulling Outdated Knowledge

In this example scenario we show how pulling mechanism is working. The rebroadcast probability is set to zero so the knowledge of components is accessible only to the neighbors. For example knowledge of 1 is published to 2, 4 and 5, but to the rest of nodes is never delivered. We have enabled the pulling with the following parameters:

Parameter	Value
KN[8] period	5000

[8] A message containing component knowledge valuation

Parameter	Value
HD[9] period	5000
PL[10] period	5000
Local timeout	6000
Global timeout	10000

Table 7. Pulling knowledge scenario parameters.

Table 8 shows selected events which occurred during runtime, in particular the delivery of component 1 knowledge to component 9. In this example after 15 seconds the knowledge of component 1 is regularly delivered to component 9 every 10 seconds.

Time (ms)	Event
4782	1 sends KN
4883	2 receives KN from 1
7795	2 sends HD (including KN 1)
7896	3 receives HD from 2 (including KN 1)
8837	3 sends HD (including KN 1)
8938	6 receives HD from 3 (including KN 1)
11585	6 sends PL 1
11686	5 receives PL 1 from 6
14984	5 retransmits KN 1
15186	9 receives KN 1
25186	9 receives KN 1
35186	9 receives KN 1
…	…

Table 8. Events of pulling system.

Even if a message is not delivered to the more distant nodes the immediate neighbors further disseminate the message header. Reception of the header on a node where the message was never delivered results in a pull request and consecutive delivery. In this example we have shown how can be ensured the reliability of message delivery by PULL request when the PUSH dissemination fails to do so.

[9] A message containing headers of received messages
[10] A message requesting particular message to be sent

8.2.2. Node Extermination

As a second example we have consider the node 1 failure after 20 seconds of the system run. Because the knowledge is no longer published the nodes start to pull it. Table 9 shows selected events which occurred during the runtime.

Time (ms)	Event
…	…
21408	9 sends PL 1
21585	3 sends PL 1
21585	6 sends PL 1
24850	8 sends PL 1
28657	2 sends PL 1
29726	4 sends PL 1
29850	8 stops pulling 1
31408	9 stops pulling 1
33657	2 stops pulling 1
…	…

Table 9. Event of pulling system with node failure.

Notice that more distant nodes start to pull the message earlier because it was delivered a longer time ago but then also the immediate neighbors start pulling. Because the source node is no longer available there is no one to respond the request. After a specified timeout the nodes realize that the message probably will not be delivered and stop pulling. In this example we have shown how the protocol overcomes the case of having a header of a nonexistent message.

8.3. Exploiting Infrastructure Networks – PULL-based Communication

The advantage of infrastructure network has been exploited implementing the groupers. It is probable that a promising solution can be also achieved for the pulling mechanism. In this Section we propose some ideas that could be used in the future.

As it is visible on the Figure 32 the grouper acts as the pulling mechanism. Sending to it knowledge of some component has the same result as pulling the knowledge of all other components with some probability. The dissemination of the communication groups is performed by the grouper and can be viewed as the message

headers dissemination. The group members are actually the components involved in the communication.

When implementing the pulling directly without groupers we can exploit the possibility of node addressing and pulled knowledge can be directly send to the requested node and eliminate completely the gossip communication channel. The dissemination could work as in the case of regular gossip together with notification of sent message headers. Each message header has associated the address of the source node. A pull request is then directed to the source node with the address of the requesting node and pulled message is sent directly and immediately. Incoming pulling requests can be exploited to improve the gossip protocol. The dissemination mechanism could prefer to select those nodes from which the most pulling requests are coming. It is expected that these nodes are interested in the disseminated data.

9. Related work

9.1. DHT

Implementation of a distributed key-value storage (DHT) can be decomposed into two main components: a key-space partitioning and an overlay network [12].

Key-space partitioning mechanism assigns range of keys to the involved nodes [12]. Whenever a new value is inserted we find the node responsible for particular key and we store the value on that node. Similarly when searching for a key we find the responsible node and the value is obtained from there. Many systems use some variant of consistent hashing [13] to assign keys to the nodes. So when a new node joins the network only limited number of nodes are affected. Chord [14, p. 230] for instance treats keys as points on a circle. Each node has assigned one point and is responsible for keys between the previous node and itself. Arrival of a new node results only in reorganization between two neighbors.

Whenever performing lookup or insert operation any node in the system must be able to determine which node is responsible for a particular key. Therefore each node maintains a set of links (routing table) to other nodes which form an overlay network. For an illustration purpose consider a simple solution: each node keeps a link to the successor node on the circle. When performing a lookup operation a message with the key is passed around the circle until the responsible node is found.

In order to optimize communication between ensemble components we partition them into related groups. To provide such a service we have introduced a special role of grouper component which performs the grouping and stores the partitions in a distributed storage.

Because key-space partitioning is a complex task and it is not in our interest to come up with a completely new implementation of distributed hash table, we have statically assigned ranges of keys to individual nodes. This solution is however intended for testing purpose only. Because the grouper is a regular component its knowledge can be also target of a partitioning function. Nodes with grouper deployed use the register of known nodes as routing table in DHT pointing to each other forming a circle. The routing tables can be initialized with special purposed grouper designated for grouping other groupers.

The overlay network is build and maintained by a gossip communication and thereby exploiting the jDEECo communication mechanism. This approach is also used by the Amazon Dynamo [15] system utilized to store customer shopping carts. As explained in [16] and [17] gossip protocol seems to be a promising solution for maintaining the overlay network.

A grouper role can be also viewed as a group communication mechanism in a distributed environment except the fact that it does not need to be reliable. A node is subscribed to a group by simply gossiping its knowledge and membership in a group is based on node knowledge.

9.2. Routing

Context aware routing is a technique used in wireless mesh networks or delayed tolerant mobile ad hoc networks. Routing protocols in such networks use various information from the environment (context) in order to discover optimal path from source to the destination or to adapt to the network topology changes. These protocols can be categorized into (i) proactive (table-driven) – each node maintain information about topology (ii) and reactive (on-demand) – path to the destination will be discovered when requested e.g. by flooding message [18]. Various kind of data can be exploited to achieve this goal. [19] propose a solution based on host mobility (it is probable that mobile host meets many neighbors) and previous colocation with the recipient (past colocation indicates meeting in the future).

Geographic routing (geo-routing) for instance relies on the geographical position and instead of fixed address it is able to deliver packets to a node at particular position. This can be achieved by greedy forwarding [20, pp. 3–5] algorithm which forwards sent packet from the source node to the next one which is closer to the destination.

Our gossip communication among the nodes is optimized by exploiting the knowledge data and delivering it only to interested nodes. As in the case of on-demand ad hoc routing protocol. The destination is discovered by flooding the network (broadcasting to any node we know), but maintained by a table-driven routing protocol. Routing information is propagated across the network by the groupers which can be viewed as a global distributed routing table.

Geo-routing can be applied under the following assumptions [20, p. 2]: (i) node can determine its position, (ii) node is aware of neighbor positions, (iii) and destination position is known. In a very simplified way we modify the idea of geo-routing but instead of geographical location we use arbitrary part of node local data. As an example consider Vehicle nodes partitioned by the vehicle type (passenger car, truck, bus …). In the assumptions of geo-routing we replace the position with the vehicle type: (i) whenever a node communicates with other nodes it determines its type, (ii) it is aware of other nodes of same type because of the grouper updates, (iii) so the communication is directed to the well-known nodes with specified vehicle type. Now we are able to send a message to a truck without further knowledge of the current network topology.

Delivery of knowledge data resembles the greedy forwarding. A vehicle truck node gossips its knowledge in the assumption that the recipient is a truck as well. Instead a grouper receives it and realizes that another grouper is responsible for truck vehicles so the knowledge is forwarded. DHT algorithm achieves that the knowledge is finally forwarded to grouper which stores truck vehicle knowledge. From here it can be again forwarded to truck nodes. On the way from the sender the knowledge is permanently approaching to the recipient.

9.3. Stigmergy

Stigmergy is a general mechanism of coordination between multiple agents without the need of planning or direct communication. As a result the agents can implement very simple logic and possess limited resources such as the memory. This mechanism was observed in the behavior of ants laying down pheromones after finding food and returning back to the nest. The designated route attracts more ants which following the path lay down even more pheromones reinforcing the route.

Described mechanism can be successfully used in routing of messages in MANET network. In [21] the authors introduce a STIgmergy base Routing protocol (STIR) targeting delay tolerant networks. Agents in this system create virtual paths between the content publisher and the content consumer similarly to ants searching for the food. This paths are reinforced or modified according to the evolution of the environment such as temporal disconnection, node failure or extermination.

The pulling mechanism of DEECo similarly to STIR publishes the interest of communicating nodes by retransmitting the headers of missing or outdated messages

and leaving behind a virtual path formed by the pull flags. Reaching the source at the producer the requested message is traveling back using the virtual path. The pull flags are temporal and after predefined timeout are released avoiding the message to be delivered where unnecessary.

10. Conclusion

In this work we have considered communication aspects of DEECo component model targeting a distributed environment with nodes mobility and highly dynamic architecture. These aspects has been formally specified by rules restricting rather general semantic of DEECo component model. With this step we have provided a documentation which clearly outlines the environment and system properties at theoretical level. We have shown the usability of this approach by further extension of the refined semantic for new features in the communication protocol. We expect this work will lay a ground for further elaboration of the DEECo component model, especially the communication aspects.

Additionally, we have defined a modification of the communication with optimization in mind with promising results. In particular we have exploited the communication on infrastructure level by introducing a so called grouper service establishing communication group and restricting the range of gossip protocol. The experiments show that a regular gossip results in exponential increase of the number of messages necessary to be sent in order to successful disseminate the data comparing to the grouper service. This solution however does not introduce any centralized element and is based on a well-known principles of distributed hash tables and existing algorithms may be reused for this purpose.

As the last we have considered the timing aspect of data delivery and we have implemented a pulling mechanism in order to actively control the direction of data routing. Our experiments shows that the median of message delivery time is approximately the same as the sending period, which is 70 time smaller than regular gossip in MANET network.

Bibliography

[1] S. Karnouskos, “Cyber-Physical Systems in the SmartGrid,” in *2011 9th IEEE International Conference on Industrial Informatics (INDIN)*, 2011, pp. 20–23.

[2] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECO: An Ensemble-based Component System,” in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, New York, NY, USA, 2013, pp. 81–90.

[3] B. Divecha, A. Abraham, C. Grosan, and S. Sanyal, “Impact of node mobility on MANET routing protocols models,” *J. Digit. Inf. Manag.*, vol. 5, no. 1, pp. 19–24, 2007.

[4] Márk Jelasity, *Gossip*. .

[5] Rima Al Ali, Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil, “DEECo computational model–I.” .

[6] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, “The Broadcast Storm Problem in a Mobile Ad Hoc Network,” in *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, New York, NY, USA, 1999, pp. 151–162.

[7] P. N. Kyasanur, R. R. Choudhury, and I. Gupta, “Smart Gossip: Infusing Adaptivity into Gossiping Protocols for Sensor Networks,” Apr. 2006.

[8] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “Gossiping Components for Cyber-Physical Systems,” in *Software Architecture*, P. Avgeriou and U. Zdun, Eds. Springer International Publishing, 2014, pp. 250–266.

[9] B. Garbinato, “Gossip-Based Dissemination,” in *Middleware for Network Eccentric and Mobile Applications*, .

[10] A. Saidi and M. Mohtashemi, “Minimum-cost First-Push-Then-Pull gossip algorithm,” in *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, 2012, pp. 2554–2559.

[11] P. Felber, A.-M. Kermarrec, L. Leonini, E. Rivière, and S. Voulgaris, “Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams,” *Peer--Peer Netw. Appl.*, vol. 5, no. 1, pp. 74–91, Feb. 2012.

[12] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Looking Up Data in P2P Systems,” *Commun ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003.

[13] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1997, pp. 654–663.

[14] K. Dhara, Y. Guo, M. Kolberg, and X. Wu, “Overview of Structured Peer-to-Peer Overlay Algorithms,” in *Handbook of Peer-to-Peer Networking*, X. Shen, H. Yu, J. Buford, and M. Akon, Eds. Springer US, 2010, pp. 223–256.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s Highly Available Key-value Store,” in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY, USA, 2007, pp. 205–220.

[16] A. Ghodsi, S. Haridi, and H. Weatherspoon, “Exploiting the Synergy Between Gossiping and Structured Overlays,” *SIGOPS Oper Syst Rev*, vol. 41, no. 5, pp. 61–66, Oct. 2007.

[17] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, “Handling Churn in a DHT,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004, pp. 10–10.

[18] Y. Yang, J. Wang, and R. Kravets, “Designing Routing Metrics for Mesh Networks.”

[19] M. Musolesi and C. Mascolo, “CAR: Context-Aware Adaptive Routing for Delay-Tolerant Mobile Networks,” *IEEE Trans. Mob. Comput.*, vol. 8, no. 2, pp. 246–260, Feb. 2009.

[20] B. S. P. Bentham Science Publisher, “Theory and Practice of Geographic Routing,” in *Ad Hoc and Sensor Wireless Networks: Architectures, Algorithms and Protocols*, H. Liu, X. Chu, and Y.-W. Leung, Eds. BENTHAM SCIENCE PUBLISHERS, 2012, pp. 69–88.

[21] A.-D. Nguyen, P. Sénac, and M. Diaz, “STIgmergy Routing (STIR) for Content-Centric Delay-Tolerant Networks,” presented at the LAWDN - Latin-American Workshop on Dynamic Networks, 2010, p. 4 p.

List of Tables

List of Figures

List of Abbreviations

CPS – Cyber-physical system

DEECo – Dependable emergent ensemble of components

DHT – Distributed hash table

GPS – Global positioning system

MANET – Managed ad-hoc network

MATSim – multi-agent transport simulation

NED – Network Description

RSSI – radio signal strength indicator

SCS – Smart Car Sharing

STIR – SIGmergy Routing

TCP – transmission control protocol

Attachments

1. Implementation in Java
2. Configuration files of simulations perform during the experiments
3. Use-case scenario data and configuration files
4. Setup guide for DEECo, OMNeT++ and MATSim
5. Random scenario generator tool
6. Reporting scripts