Implementation and Evaluation of a SwarmLinda System

Technical Report TR-B-08-06

submitted by
Daniel Graff

supervised by
Prof. Dr. Tolksdorf Prof. Dr. Menezes

June 23, 2008

In collaboration with the

Department of Computer Science
Networked Information Systems
Freie Universität Berlin
Berlin, Germany

Department of Computer Science
Laboratory for Bio-Inspired Computing
Florida Institute of Technology
Melbourne, Florida, USA

Abstract. *In the middle 80s David Gelernter from the Yale University developed a programming language called* LINDA. *It is applied in the field of distributed environment development. Moreover,* LINDA *also describes a coordination model which is fully distributed in space and time. Based on its characteristic, although being used in parallel computing, it is restricted to a certain amount of servers while it cannot cope with adaptiveness and scalability in an open environment. As a solution* SWARMLINDA *is proposed based on a decentralized multi-agent system which got its inspiration by nature in the field of swarm intelligence. The ability of the architecture is characterized by a very scalable behavior. The system can grow to enormous size while still be very effective since the principle is based on only local interaction with the surrounding neighborhood. The behavior patterns are observed from natural individuals (aka swarms) like ants, birds, termites and bees. Each agent is characterized by simplicity, dynamism and locality. The research for the report has been performed in evaluating as well as examining properties, problems, behavior pattern and advantages of swarm intelligence used in a* SWARMLINDA *system. In particular, the work discusses involved mechanisms and describes the development of a* SWARMLINDA. *Further on, it defines different metrics being used in the system. A main part is based on the distribution of information objects in a 2D environment by forming clusters which hold similar objects. The cluster itself is defined by a spatial region containing several associative memories. Finally, the report closes by presenting plenty of test runs that have been executed on the* SWARMLINDA *system. In order to rate the performance an evaluation metric has been developed defining the spatial network entropy.*

Contents

1. Introduction 5
   1.1. Historical Background . . . 5
   1.2. Thematic Demarcation . . . 6
   1.3. Motivation . . . 8
   1.4. Objective . . . 16
   1.5. Structuring . . . 16

2. Algorithms In SwarmLinda 18
   2.1. Tuple Distribution . . . 18
      2.1.1. Drop Probability . . . 22
      2.1.2. Path Selection . . . 22
   2.2. Tuple Retrieval . . . 26
   2.3. Tuple Movement . . . 30

3. Development and Implementation 35
   3.1. Used IDE . . . 35
      3.1.1. NetLogo . . . 35
         3.1.1.1. Definition . . . 35
         3.1.1.2. Usage . . . 36
         3.1.1.3. System Architecture . . . 38
         3.1.1.4. Extension-Model . . . 39
      3.1.2. Eclipse . . . 41
         3.1.2.1. Definition . . . 41
         3.1.2.2. Plug-Ins . . . 41
         3.1.2.3. Build System with Ant . . . 42
   3.2. Network Generation Simulator . . . 43
      3.2.1. Generation of Networks . . . 44
      3.2.2. Interaction with Networks . . . 45
      3.2.3. Plotted Graphs . . . 47
   3.3. SwarmLinda Simulator . . . 47
      3.3.1. Visualization . . . 47
         3.3.1.1. Controlling the Simulator . . . 47
         3.3.1.2. Output: Monitors and Plots . . . 52
         3.3.1.3. Additional Test Environment . . . 54
      3.3.2. Implementation . . . 56
         3.3.2.1. Extension . . . 56
         3.3.2.2. Logging . . . 57
         3.3.2.3. Test Environemt . . . 57

4. Improved Metrics in SwarmLinda 59
   4.1. Drop Probability . . . 59
   4.2. Entropy . . . 66

4.3. Pickup Probability 70
4.4. Anti-Overclustering 75

5. Experiments Showing Optimization Results 80
5.1. Tuple Distribution 80
5.2. Tuple Retrieval 85
5.3. Tuple Movement 90
5.4. Comparison of the improved Metrics 93
5.5. Anti-Overclustering and Spatial Clustering 98

6. Conclusion and Future Work 106

References 110

List of Figures 115

List of Tables 117

A. Diagrams 118

1. Introduction

This report evaluates and examines properties, problems, behavior patterns and advantages of swarm intelligence used in a SWARMLINDA system. The following gives an introduction to swarm intelligence extending conventional LINDA systems.

1.1. Historical Background

In order to understand the development of the coordination system one has to look at its origin. In the middle 80s David Gelernter from the Yale University developed a programming language called LINDA introduced in [20]. It is applied in the field of distributed environment development. Moreover, LINDA also describes a coordination model which is fully distributed in space and time. Processes can communicate with each other using an associative memory (tuple space) without knowing any identifying information about each other. A sender A can leave a message in an associative memory, a retriever B can withdraw the message independent in terms of time. The system does not postulate B being a specific retriever but instead B is just one single process from a universal set R defining all processes being involved in the system. One can interpret the associative memory as an abstract space for storing and retrieving arbitrary information objects (tuples).

In the beginning Gelernter proposed the global tuple space as interface between the storage area and the acting processes around. Later he extended the system to multiple tuple space making it more suitable to distributed environments [21]. Therefore LINDA was not only distributed in terms of parallel computing but also in case of spatial usage. Tuples can be spread over the network among several nodes. The system gains more robustness since it avoids having a single point of failure. Partitioning the amount of information objects across the domain leads also to the idea of clustering similar tuples in the same region making the system more effective.

Since distributed computing becomes more and more ubiquitous nowadays and LINDA "is arguably the most important and most successful coordination model ever proposed" [12] it found its place in several applications ranging from parallel computing [19], [40] to mobile systems [36], [39] and finally to peer-to-peer systems [6]. There are also commercial implementations that includes the idea of generative communication used in TSpaces [62], JavaSpaces [18] and GigaSpaces [27].

Later on the idea that objects can retire over time arises. Assume a scenario in which an amount of k tuples are spread among a network containing n nodes. At each time t further m tuples are added. Considering an aging mechanism that tuples are getting older, one can assign an index j to each tuple tu indicating the appearance of tu. Therefore the actual age can be computed by subtracting the appearance time t_{app} from the current time t_{curr}. Shortly speaking, the smaller t_{app} is the older it gets. It is easy to notice that if t_{app} of tuple X is $\frac{1}{2}t_{curr}$ and t_{app} of tuple Y is t_{curr} then Y is exactly twice as old as X. Assume further that the probability of picking up a tuple of kind A - in terms of LINDA tu matches template A - is anti-proportional to its age or, the other way round, correlates

with its freshness. There is no chance of rejuvenation. Let age_1 be the age of a tuple matching template A and age_2 be the average age of a set of tuples also matching template A with $age_1 \gg age_2$. One can associate the age factor with some sort of currentness meaning that recent tuples contain newer information. Since age_1 is already very big compared to similar tuples it is very unlikely that a requester will obtain this tuple. Hence it is appropriate to introduce garbage collection for cleaning up the domain and remove apparently unused objects. An extensive discussion can be found in [32], [29] and [33].

1.2. Thematic Demarcation

It exists several standards for performing communication between two subjects. On a low level the programming language *C* allows inter-process communication (IPC) via pipes [49]. Usually during the runtime of a program a child process will be created using the $fork()$ command. By establishing a pipe between the parent and the child the runtime environment allows a unidirectional communication. This is also called an *anonymous pipe*. The father process uses the input end of the pipe to write a data stream to it. The child on the other side reads the incoming data stream. This variant is commonly used in operating systems. It is byte-oriented and the processes have to be closely related to each other. The *anonymous pipe* exhibits a simplex and reliable FIFO communication channel. Writing and reading commands are always blocking. Both processes have to be on the same local machine.

On the other hand *named pipes* allow a full duplex reliable communication. The processes do not need to be related, they neither have to be on the same machine. The communication can either be byte-oriented or message-oriented. The commands of *named pipes* can also be executed in a non-blocking way.

Another inter-process communication model is *Shared Memory* [49]. The involved processes use a common area in the Random Access Memory (RAM) for information delivery. In detail the initiator has to allocate a certain part of the memory while the retriever has to gain access rights for reading. This model can be applied both to software (communication between processes) and hardware (usage of a typically large block of the RAM that can be accessed by several central processing units (CPUs)) issues. Some graphics card manufacturers also provide shared memory but this does not relate to IPC since the cards use an additional part of the RAM for extending their own local memories.

Similar to IPC using *named pipes* is socket based communication [7]. Processes can also address remote as well as local machines. The communication channel is full duplex. There are different types of socket realizations implementing the reliable, stream based Transmission Control Protocol (TCP) or the packet oriented, unreliable User Datagram Protocol (UDP). Socket communication is usually used for client-server architectures.

A more sophisticated standard is the Common Object Request Broker Architecture (CORBA) [48] developed by the Object Management Group (OMG). CORBA is an object oriented middleware and provides cross platform, language independent interaction. The core of the runtime environment is the Object Request Broker (ORB) which is responsible for locating local and remote objects. CORBA uses Remote Procedure Calls (RPCs) for dis-

tributed interaction. Objects invoke methods from other objects whereas the location does not play a role. The instances can be either contained in the same process, distributed on several processes or spread among different server. Invoking an objects method the ORB performs the actual RPC. The Internet Inter-ORB Protocol (IIOP) that has been published by the OMG assures the communication between different ORBs.

Due to its open and independent behavior CORBA is likely used in distributed systems coping with different languages and operating systems. For it the Interface Definition Language (IDL) defines modules, method signatures, exception handling as well as mappings between language specific data types. Afterwards the IDL-Compiler which comes along with the applied ORB generates stub and skeleton classes which are proxies for the objects performing the RPC.

In the scenario that a process A invokes a method of process B it does not know the physical location of B. Therefore the ORB is responsible for finding B. Each callable CORBA instance has to be registered with a unique identifier at the naming service. The use of a Uniform Resource Identifier (URI) is recommended. During a method invocation the responsible stubs serializes the method name as well as optional parameters and sends it to the remote instance. At its destination the incoming data stream gets deserialized and does the actual method call. Finally it responds to the invoker with optional return arguments.

Another middleware technology is Remote Method Invocation (RMI) [24] which is similar to CORBA but it is restricted to one programming language; it can only be used in Java application. Analogous to CORBA RMI needs a specific compiler for generating stub and skeleton classes. Before invoking remote objects they have to be registered at the RMI registry, preferable with an URI. The RPC takes place by calling the method stub of the remote object, serializing parameters, addressing and sending it to the actual implementation. At its destination the incoming data stream gets deserialized and invokes the method. Optional return arguments will be sent back. All three participants - the caller, the registry and the invoked service - are running in different Java Virtual Machines (JVMs). The Java Remote Method Protocol (JRMP) handles the communication interaction. In order to run code in a non-JVM context the RMI-IIOP (RMI over IIOP) was developed for supporting and addressing CORBA applications.

Another well known communication technology are Web Services which were introduced by the W3C [28]. The basic idea of Web Services is to provide service functions that can be invoked via the World Wide Web (WWW). Similar to CORBA and RMI Web Services also need a Naming Service for registering offered services. The Universal Description Discovery and Integration (UDDI) dictionary contains detailed information encoded in XML. For publishing a service the Web Service Description Language (WSDL) as explained in [13] has to be generated. It contains plenty of information of how to interact with the server, e.g. method signatures with parameters and return values. The service providing node publishes its WSDL at the UDDI. Afterwards service requesters can look up the WSDL in order to contact the provider using SOAP[1].

[1]Originally SOAP was an acronym for Simple Object Access Protocol but since version 1.2 it is not an abbreviation anymore since it is neither simple nor it is only used for object access [60]

However, LINDA does not need a description language for performing inter-process communication. In contrast to web services where the requester is addressing a specific server LINDA deals with openness since a process storing an information object in a particular node does not know neither which other process will potentially pick it up nor whether it will generally picked up by anyone. If this piece of information is not of interest for the environment than it will stay there. There is no direct communication between processes in LINDA, they exchange information via tuples that can be stored and retrieved from tuple spaces. Considering the mechanism of LINDA assigns the system a loose and uncoupled behavior since storing and retrieving processes are independent in terms of time and availability. "LINDA differs from previous interprocess communication models in specifying that messages be added in tuple-structured form to the computation environment, where they exist as named, independent entities until some process chooses to receive them" [20].

1.3. Motivation

In order to construct and maintain huge distributed software systems one has to cope with many issues. As mentioned earlier LINDA is a successful coordination model. Processes communicate using associative memories called tuple spaces. Arbitrary information objects (tuples) can be stored and retrieved from tuple spaces. That is the way how the information is exchanged between the processes - fully distributed in space and time. There are several issues that should be considered carefully:

Scalability: Distributed systems like LINDA can grow to enormous size in terms of servers (nodes) and arbitrary information objects (tuples) stored in it. A plethora of processes are interacting with the system, in detail they store, retrieve and move information objects around. Scalability is today the *sine qua non* of efficient distributed systems [31]. There are several approaches to achieve scalability.

Obreiter and Gräf discussed in [37] a fast way of storing and retrieving tuples by looking at the structure of the tuple. This approach describes the tuple itself as the central point of interest. Applying hash functions each tuple is assigned to a specific server. The return value of the function tells the physical location. Thus tuples will be distributed and organized in tuple spaces based on its structure. However, Rowstron [42], [43] did some implementations exhibiting that tuple spaces should hierarchically organized in order to achieve scalability. Further he proposed local and global tuple spaces.

But in general if one looks at the techniques that were used in most conventional LINDA systems it is noticeable that they are data-oriented. Assume a scenario where a set of servers and an amount of tuples are contained in a network. If one wants to retrieve a specific tuple the most common approach is to ask the available servers based on a multicast whether they hold a matching one (based on a template). Thereupon each requested server will check its tuple space and if it finds one it will lock the tuple and offers it to the inquiring process. Of course, the requester is satisfied by obtaining one tuple whereas the remaining $n - 1$ offered tuples are not of

interest. But in this scenario they are blocked for a certain amount of time until the proceeding of querying is finished. Meanwhile other processes that are performing interactions with the same servers would not find the tuple since they are locked but indeed not used. This behavior is a very severe impact while trying to obtain and maintain scalability.

Finally, constructing huge, scalable distributed systems should postulate some level of fault-tolerance since it is very unlikely to assume a failure-free system [22]. They also should be extreme decentralized and focus on the autonomy of objects since large-scale systems cannot cope with centralization. In order to achieve a fast storage and retrieval time tuples should be kept organized so that similar types are staying at the same location. That can be either the same server or the surrounding neighborhood. On the other hand tuples should stay in proximity to processes requiring them. This behavior postulates some level of mobility. Tuples should be clustered geographically as well as be mobile by staying in proximity to the consumer. Such an organized system would be fault tolerant since its information objects are spread across servers in some neighborhood as well as fast and effective since data can be found easily and retrieved in a short time because they are already close to the querying process.

Adaptiveness: Large scale distributed systems should be adaptive in case of changes in the environment. The fluctuation of tuples can be very strong and thus the environment is changing rapidly. Due to node failures, servers can disappear for a certain amount of time which means that the contained tuples are hidden from the environment and cannot be retrieved or moved until it gets reconnected. A worse scenario is that the server gets totally removed from the system. On the other hand new nodes can be added to the environment without severe effort. Therefore the fluctuation of servers can also very rapid and unpredictable. Large scale distributed systems should cope with all changes in the environment in order to guarantee effectiveness and availability. As mentioned earlier similar information objects should not be stayed at one particular node, they should be clustered among several nodes being in the same neighborhood since the system should avoid a single point of failure. If one or more nodes fail there should not be a bad impact making the system inefficient and slow. Conversely, the system should react to changes and thus maintain its performance. The basic primitive is to use all system resources optimally. Therefore adding new nodes to the system should result in integrating them in order to keep the system always balanced.

Network Unpredictability: Compared to a system that is running on a local machine, distributed systems have to cope with network unpredictability since they are executed in a global context. Even software that is multi threaded or separated into several processes, for instance to take advantage of a cell, multi core or a multi processor computer system, is very reliable concerning its execution and runtime since the inter-process communication exhibits a predictable behavior. In the context of leaving the local machine there are network properties that should be considered. A

major issue is that the Internet is heterogeneous and shows an unpredictable behavior. It is not as easy as one may think to estimate the transmission time of data based on geographically closeness. The Internet is stateless and does not provide quality of service (QoS) [30]. A main issue for engineering distributed systems is to place the information objects in the domain in a way that querying processes get the best possible response times. This should result in a fair load balancing between all the nodes. Considering a distributed system it may not appropriate to determine node neighbors geographically, they should be selected based on latency.

Failures: It is very unlikely to assume a failure free system. Especially in a distributed context one has to face many problems. It is quite important to handle failures effectively. There are fault tolerant coordination models like PLinda [1] based on a transaction mechanism. But as one might think it is very cost intensive to support transaction semantics in a fully distributed system environment since it uses distributed locking mechanisms for keeping the state consistent. Another approach is proposed in [54] analyzing fault-tolerance in LINDA systems. The studies have been taken place in the categories of transactions, mobile coordination, replication and checkpointing. But since "all these mechanisms are rather static in that they assume a fixed location of faults and fixed locations of where faults are managed" [30] the kinds of failures are strict; the system is either consistent or in a state of failure. There is no intermediate level that it is partially inconsistent. However, considering the approach proposed in [44] behaves more flexible since it adds mobile code.

Mobility: Nowadays mobility gets more and more ubiquitous, thus postulating that distributed environments should handle it. However, focusing on LINDA systems one can separate it in three main parts: nodes (tuple spaces), tuples and querying processes. The objective is to find and retrieve tuples in an appropriate time. As mentioned earlier the queried data should stay close to the consumer. But how can this be done? Tuples can be in an active or passive state. Assume a scenario where an amount of tuples are spread among a network and a set of processes are trying to find them. It is very likely that some tuples are already placed well for inquiring processes, however, others do not. As one can think it might be more effective to store the tuples in tuple spaces that are in proximity to the consumers. Therefore tuples should be active and move towards the consuming sources. An unrealistic scenario but easy to understand, is that a network contains two different kinds of tuples. There are two locations - on both sides of the network - that identify the consuming processes which querying one of the two types respectively. It makes sense to move the tuples towards those locations in order to guarantee fast retrieval times as well as establishing some level of order by sorting them by type. Further the system is more balanced since the information is already stored in a suitable place and the effort for finding a tuple tends to go down. Thus system resources would be spared resulting in a high global performance.

There are several miscellaneous models for implementing LINDA. All of them try to solve conventional problems which occur in distributed systems. Although there are

some more sophisticated approaches tending to avoid bottlenecks and establish some level of load balancing and fault tolerance, nevertheless they own a threshold at which the scalability and adaptiveness will suffer from its architecture and system behavior.

Centralization: The first and simplest strategy is the well known client-server model (see Figure 1). This architecture is very easy to implement since there is just one tuple space containing all tuples (see TSpaces [62]). There is no need of synchronization or distributed locking mechanisms. Clients can attach to the server in order to store as well as retrieve tuples. These are the advantages of a centralized organization and it is well fitting in a particular context, when the number of clients are small so that the server can respond to all incoming requests in an appropriate time. But unfortunately, as one may think the server gets easily a bottleneck that is when the number of clients strictly increase. Since we are dealing with open distributed systems it is noticeable that this approach does not fit in the context of adaptiveness and scalability. Further this organization suffers from a single point of failure.

Figure 1: Centralized tuple spaces (according to [30])

Partitioning: According to this strategy (see Figure 2) the data is separated among the nodes in a way that each one contains a subset of all tuples. By physically distributing the amount of tuples will, of course, lead to more productivity but, however, will also result in an unbalanced system. Sorting the tuples by type so that each server holds a specific kind, one does not know in advance whether they are queried equally. It is more likely that one node gets most of the requests while the other ones remain more restfully. Providing parallel query processing and routing it to one of the servers will, of course, increase performance since it supports concurrency. Nevertheless there is still centralization involved that inevitably will result in a bottleneck and so is a weak point of the system. However, Bjornson showed in [4] that applying a carefully chosen hash function will distribute the tuples more efficient among the nodes in order to obtain more balance.

Full replication: This approach replicates all data contained in the tuple space to all nodes (see Figure 3). Therefore each server offers the same information and thus there is a fair load balancing involved in the system since the amount of requests can be redirected to nodes having small workloads. This is an advantage compared to the previous mentioned strategies. Further the approach offers fault tolerance since a node failure does not effect the behavior dramatically and will not lead to a loss of information. Unfortunately, it consumes a lot of system resources to keep the

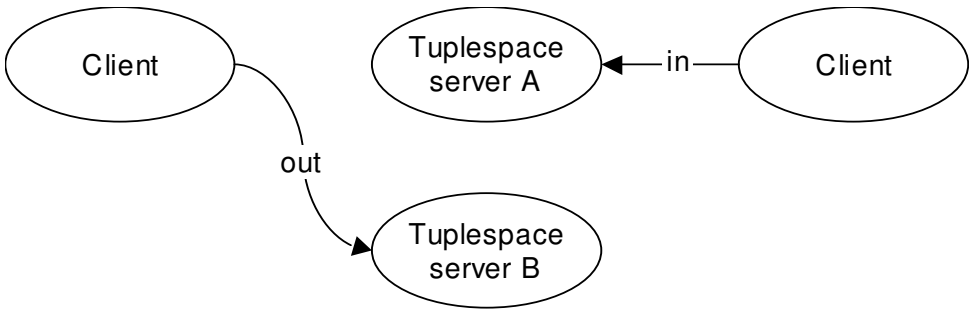


Figure 2: Partitioned tuple spaces (according to [30])

replicas always in a consistent state. Each storage as well as removal of data objects causes the system to react on those changes and perform them on all other nodes. Therefore the update involves distributed locking mechanisms [14] that result in a severe impact on the performance. However, dealing with small applications and a straightforward amount of clients, so that the interactions between the actors in the environment are predictable, the system will be balanced and stable. As soon as one may think of adding arbitrary more clients it is easy to notice that the number of requests will increase inevitably which finally leads to more updates in order to maintain consistency. Conversely, adding a whole server involves complete replication of all data. This approach suffers from too much synchronization and hence cannot cope with scalability and adaptiveness confronted in an open distributed context. In addition one may notice that the storage space is, however, limited and thus a spatial distribution is more suitable.

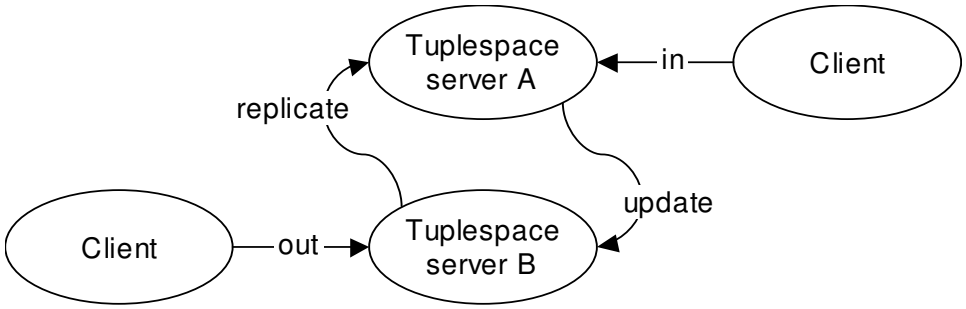


Figure 3: Full replication (according to [30])

Intermediate replication: Finally, the intermediate replication (see Figure 4) is a more sophisticated approach since it profits from previous mentioned models and combines advantages like physical partitioned data while avoiding a single point of failure.

The shape is formed like a grid which consist of in- and outbusses. Data which should be stored in the network gets replicated among all nodes which are in the same outbus. Thus the synchronization only takes place on a few nodes in contrast to the full replication strategy. Analogous, the process of finding and retrieving tuples is only performed on the inbus. Therefore all nodes belonging to this bus needs to be updated in order to be consistent. However, the replication or removal of data includes locking mechanisms but they will be only performed on particular nodes while the remaining ones stay in idle mode waiting for new requests. In fact, this strategy behaves well compared to the previous mentioned ones since it provides concurrent data access, fault-tolerance established by obtaining as many physical data copies as number of inbusses, spatial partitioned data while minimizing the synchronization effort. It is the most general concept; one obtains centralization by reducing the number of nodes exactly to 1 ($outbus = inbus$). On the other hand full replication can be achieved by restricting the outbus to 1 containing n nodes and thus n inbusses. An implementation of this model is proposed in [53]. Although, the architecture is more suitable and seems to be robust, the model gets insufficient by strongly increasing the number of in- and outbusses since the synchronization overhead tends to grow rapidly.

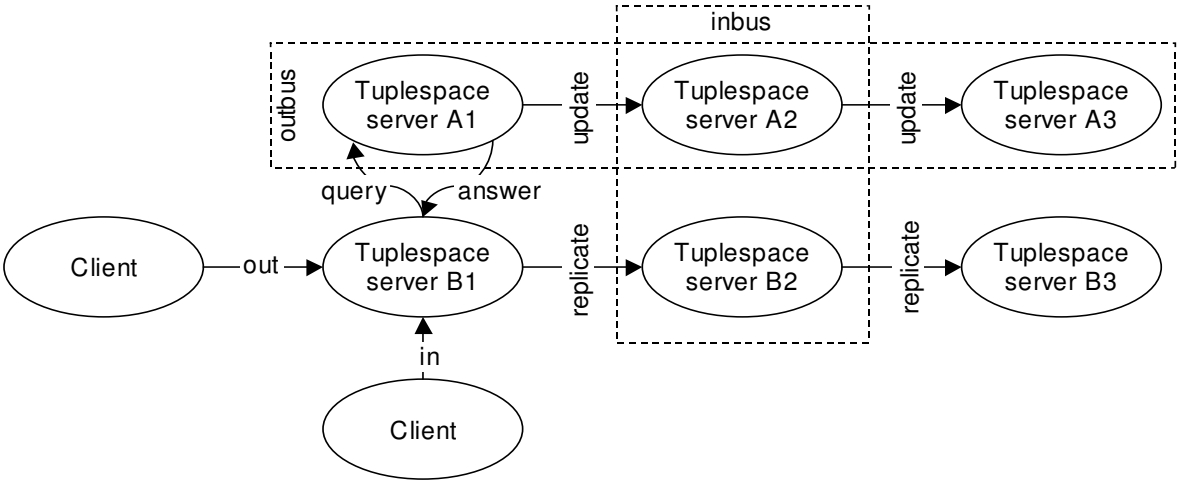


Figure 4: Intermediate replication (according to [30])

None of the aforementioned approaches fit in the context of large-scale distributed system environments since they cannot cope with scalability, flexibility, mobility and adaptiveness. Finally, extensive researches in the field of LINDA systems presented that due to the weakness of these architecture, one has to try a completely different way in order to solve problems and obtain real openness (cf. [31, 30, 52]). Thereupon, researchers got inspired from biology by observing the natural behavior of bees, termites and ant colonies. All of them exhibit a pretty decentralized behavior by interacting very egoistic with the

environment. According to basic studies of ants it is noticeable that the colonies are very scalable. They can grow to enormous size and still behave organized and effective. However, the characteristic of those colonies are not focused on the individual (a single ant), but rather on the collective. One ant may fail, but that is not important for the system as it already involves a specific quota of, somehow, misbehaving ants. The principle of the individual is based on local and easy decisions. There is neither a centralization involved in the system nor a coordinator which tells the ants what to do.

Observing the natural behavior of ants, one may notice that they have a central point - the ant hill - where they meet, where they bring food to and where they stay. From this location they roam around, exploring the environment, detecting food sources and collecting nutrition as well as suitable things, e.g. for building and expanding their nest. Ants are very productive; once in their lifetime they obtain a specific position in their community in order to serve and maintain the collective. The type of jobs reach from collecting, hunting, cleaning to caretaking ants to name just a few. While the first two mentioned take place outside the last two ones occur inside the ant hill.

However, the intelligence of an ant is rather pure and thus its complexity is small. But exactly this behavior requires only little computation and therefore results in fast decision-makings. Conversely, studying the organization of a whole ant colony, one may notice that they act very collective; they are able to solve complex problems. Each ant has its specific function in the society. That is the reason why they can grow to enormous size and still be very effective. Researchers adapted these behavior and applied it to the field of computer science. One can find several approaches using the so called *swarm intelligence* in [5, 16, 25, 41] which belongs to a special part of artificial intelligence. In contrast to conventional LINDA systems exhibiting the typical data-oriented view SWARMLINDA presents a radically new concept: one can understand the domain as a two dimensional space which consists of servers (tuple spaces), connections between them (links) and the individuals (ants). There are two types, the tuple- and template-ants. The first one carries information that should be stored somewhere in the network. The other one carries a template and is looking for matching tuples contained in the tuple spaces. They find their way using pheromones which can be tracked by the ants. Tuple-ants leave a specific scent on their way based of the type of the carried tuple. Therefore template-ants can track these trails and follow the tuple-ant. With a high probability it moves in a direction where it will find matching tuples. On the other hand tuple-ants can also track these pheromone trail since they are interested in collecting and clustering same information in the same region. Followers are reinforcing existing routes by increasing the amount of pheromones. In particular, there is a probability involved in the system with which they decide to follow an already taken path or if they try to explore a new route. However, this behavior leads to finding several paths from a source to the sink and thus optimal trails will be found. The ants are tending to use the shortest path and hence reinforcing this trail. Compared to the nature, if there are obstacles in their way, the ants are tending to find new routs to their ant hill (sink) by exploring alternative paths. Applied to computer science the individuals jink to another neighbor if their main route is unavailable, e.g. due to a broken network cable or a server failure. The system is based on easy goals (collecting items, drop them in an appropriate environment so that it fits in

the context of existing objects around and find and retrieve them). This simple routines result in a, surprisingly, very fault tolerant behavior. Swarm-like approaches are based on (cf. [31, 12, 52]):

Simplicity: Since the main objective of the extension of conventional LINDA systems and the introduction of swarm intelligence is to cope with scalability and thus allowing, at each time, to add as many nodes as desired without impacting the system performance, one basic requirement is to keep the system simple. The acting processes (ants) should be very simple in their behavior; they only have a few simple routines for collecting as well as retrieving tuples including mechanisms for finding optimal trails. The combination of a large amount of individuals performing simple tasks by showing an extreme selfish (decentralized) behavior leads to the emergence of complex situation solving possibilities while minimizing resource usage.

Dynamism: A required property of large-scale open distributed systems is dynamism since the environment is changing rapidly. Information objects can move inside the network, so that one cannot predict in advance the location of a specific tuple tu at a given time. If tu is at time t_1 at node N_1, it may be at N_2 at t_2. Therefore the tuple finding mechanism postulates some level of dynamism to find the tuple, in any case, independent on the location where it currently stays. If a tuple moves to another location template-ants should not need more time to perceive the tuple, in average the search duration should be take less since the movement tends to improve the system organization and clustering without indicating more computation overhead.

Locality: The basic principle of SWARMLINDA is locality. The individuals - independent on their task - roaming around in the network due to accomplishing their own objective. At each current location they interact with the local environment, i.e. the tuple space and the surrounding neighbors. Their objective is to finish its task quickly and successfully. However, if the current tuple space does not contain appropriate information they observe the surrounding neighborhood and continue their search in the direction in which they assume the highest probability of accomplishing its task. The decision to take a particular path is chosen probabilistically.

However, based on the mentioned principles pheromone trails disappear over time since tuples can be shifted inside the network which invalidates its current position. With the dwindling of the scent the system is able to find new routes and shrinking the importance of already taken trails. In fact, recent strong used trails are still more preferred by ants than rare used trails.

The swarming approach satisfies all the requirements for large-scale distributed systems while conventional LINDA systems fail due to its inflexible behavior. SWARMLINDA is scalable since the individuals does not care about the amount of nodes; they only interact with their local environment. In case of node failures or changes due to the environment the system does not need to perform a global update; instead it continues running and is not affected in its behavior since it adapts local changes immediately.

1.4. Objective

This report is about to evaluate the proposed algorithms and system behavior of SWARMLINDA. In order to perform the required tests the report is based on a simulator which comprises the first part of the work. The implementation is done in NetLogo[2] [61] completed with additional Java-Code which extends the simulator and is integrated in the NetLogo source code.

The work is based on proposals and research inspired by [12, 31, 30, 54, 52, 10, 11, 9, 8]. The report shows evaluations from existing algorithms and presents new adapted algorithms in order to improve the system performance. In particular, the report focuses on establishing a fast clustering while minimizing the system entropy (degree of order in the network). The tuples should be spread among the nodes in the network, so that homogeneous clusters arise while achieving heterogeneity between the nodes. The tuple types should be geographically distributed.

However, the report also evaluates the basic idea of SWARMLINDA. It examines the postulated behavior of adaptiveness and scalability while operating fully decentralized. Distributing and retrieving of tuples are only done by simple local interactions. Trails to sources are found using pheromones. The report shows how the tuple finding and distributing mechanisms proceed under certain circumstances.

The report shows some behavior that was expected in advance due to the characteristics of swarm intelligence. With it there is one part of the project confirming foreseen issues while the other one comes along with significant characteristics that appeared during the execution.

1.5. Structuring

The report starts with the introduction which gives the reader detailed information about LINDA and SWARMLINDA systems. In particular it provides the historical background of LINDA systems. It exhibits the basis idea of the coordination model and presents a plethora of approaches ranging from a single tuple space, to multiple tuple spaces applying the different distribution mechanisms (see section 1.3). After describing the scalability problems of conventional LINDA systems a motivation is given introducing and justifying swarm intelligence. Finally, it defines the objective of the report and explains the part which should be examined.

Section 2 describes in detail the algorithms which take place in a SWARMLINDA system. In particular, it deals with tuple distribution, retrieval and movement.

Section 3 covers all implementation and development details. It describes the used IDEs[3] in which the simulator is implemented; NetLogo which also provides the runtime environment for the simulation itself and Eclipse for developing extensions for the simulator in Java. Further it describes the simulator for generating scale-free networks based

[2] NetLogo is a cross-platform multi-agent programmable modeling environment which is implemented in Java

[3] Integrated Development Environment

on the approach of Barabasi [3]. The generated networks can be exported and imported in the SWARMLINDA simulator and thus defines the environment.

Before evaluating the proposed algorithms of SWARMLINDA section 4 deals with improving metrics and algorithms. At first, the work focuses on modifying the drop probability for a tuple carried by an ant in order to achieve a better clustering, i.e. increasing the level of homogeneity of tuple spaces. Further on, this section handles the modification of the spatial entropy of the network by introducing a weighting factor. This creates a more realistic scenario in contrast to applying the unified entropy. Finally, the section closes with the modification of the pickup probability, applied in the tuple movement phase, and the anti-overclustering strategy introducing a threshold and therefore setting an upper limit for the amount of tuples in a tuple space.

Afterwards section 5 analyzes and exploits the collected data from the test runs. It starts with the development of the network, i.e. the training effect, for tuple storage and retrieval in dependence on time. In detail, it shows the improvement of finding routes from sources to sinks since pheromone trails emerge from ants already taken certain paths. After some time the reinforcement of often used routes result in finding an optimal path between two locations. The individuals are able to track specific type of scents for moving towards a certain direction and thus the time or amount of hops needed for arriving at a particular node is minimized. Moreover, the metrics discussed in section 4 are compared in their original and modified form. The result points out why the changes behave more appropriate than its original counterpart. Finally, it exhibits the difference between applying some sort of seeding in the beginning of the training of the network or starting with an initial empty one. In this scenario there are no pheromones, tuples or ants in the test environment.

In the end, section 6 summarizes the main aspects of the report and gives a conclusion of the work. It presents the basic issues of the research. It closes with ideas for future work that can be done in this specific part of swarm intelligence.

2. Algorithms In SWARMLINDA

In the previous section the development from conventional LINDA to SWARMLINDA systems is described. It is given a historical evolution as well as severe problems under which LINDA is suffering from its architecture. SWARMLINDA is proposed in order to cope with the mentioned issues and promises solutions for large-scale distributed systems.

This section deals with the applied algorithms in SWARMLINDA. First, the distribution of tuples (storage) is listed. Further on, it continues with tuple retrieval and finally closes with tuple movement. Before describing the algorithms it may be necessary to introduce some abstraction terms for avoiding confusion:

Individuals These are the active entities, mapped to ants, which roam around in the network performing some tasks. The individuals are the executing instances of the algorithms.

Environment The world in which the individuals live, where they have their ant hill (tuple spaces) and the roads (links) connecting several molds is called the environment. Ants are getting born in some ant hill as well as die somewhere in the environment.

State The state describes a particular situation (snapshot) of the environment of the system. This involves locations of the ants (including their moving direction, their cargo and their age), number of nodes, number of links (including the kind of connectivity) and number of tuples and their storage location. The individuals can change the state by modifying the environment, e.g. convey some tuples to other places, consumption by a requester, etc.

2.1. Tuple Distribution

This section deals with the distribution of tuples among the nodes in the network. Looking at the previous mentioned models (see section 1.3) designed in order to improve and extend conventional LINDA systems it is obvious that there is always a compromise between performance, scalability and synchronization. Adding more servers to the network comes along with an increase of locking and synchronization mechanisms. Centralization is the only approach which is easy to maintain but does not scale well with a rising number of clients. However, the usage of hash functions tells the respective system where to physically store the information objects. Although hashing is a fast and well developed technique it does not cope with an increase of servers; it will suffer from performing rehashing in order to integrate new system resources. But large-scale distributed systems postulate openness. Servers should be added on the fly while on the other hand they can disappear for some amount of time due to node failures or unavailabilities based on network problems (bad connectivity).

The environment is dynamic. The state is changing very often. Thus the system should handle addition and removal of system resources (nodes) easily. On the other hand the information objects are also dynamic. They are not assigned to a particular location in advance; they find their place during system execution. If they do not feel comfortable

in a specific location they are able to get picked up by some ant process and transfered to another node. This is the world of a SWARMLINDA environment. There is no static behavior like in LINDA. There is no synchronization or locking mechanisms involved. There is no central instance that is responsible for taking care that new system resources should be covered by replicating or partitioning an amount of tuples there. The active individuals (the collective) assures that integration. There is no particular ant which deals with the job; instead the community is responsible and therefore there might be one or several arbitrary individuals which explore the new resource more coincidently rather than purposely.

Biological ants show a typical behavior in collecting and storing food. They keep their items sorted by type. For instance they bring different materials for expanding their nest; they also store food, larvae, eggs, etc. Each similar item stays at the same location while different types are kept separate, thus forming clusters. This organization refers to *brood sorting* [15] used by ants. This process is applicable independent neither on the amount of individuals around nor on the amount of items already brought and stored in the ant hill. Therefore it is very scalable and is not bound to a central coordinator.

This based swarm behavior is adapted to SWARMLINDA. The system consists of two different ant types: tuple- ant template-ants. The first mentioned one are the individuals that operate here. They carry a specific tuple and trying to store it somewhere in the network where it fits appropriately. The *out*-primitive is the command for instructing an ant to store a tuple in the environment and thus initiates tuple distribution. For implementation issues, the command may have the method signature $out(pTuple)$. Each client which is connected to the SWARMLINDA network is able to invoke the command. Abstractly, only a connection object is needed implementing the method. The parameter *pTuple* is a local instantiated object holding arbitrary values. This information object is an n-tuple of the form $(v_1, v_2, ..., v_n)$ with $\forall v_i \; \exists u \in U$. U is the basic universe containing all data types. It is important to be aware of the ordering of the tuple since $(a, b) \neq (b, a)$. They are not commutative; there is also no symmetry.

As mentioned earlier (see section 1.3) ants track pheromones in order to find tuples or to store them at a location where similar ones are around. Based on the characteristics of the tuple, e.g. the number of parameters and their data types, it indicates a specific kind of tuple. In other words it matches a certain template. For instance the tuples (1, 2) and $(12, 43)$ are similar since both of them match the template (*Integer*, *Integer*). Hence a match is based on the same number of parameters as well as their data types, showing the same ordering. (*Integer*, *Integer*, *Integer*) is therefore a different template and does not match. There is no commutative behavior; (3.4, "*This is a string*") and ("*This is another one*", 10.75) is thus matching template (*Double*, *String*) and (*String*, *Double*). In fact, all four mentioned templates are dissimilar. The pheromones for tracking tuples are based on the composition of the templates. Thus a scent trail indicates an occurrence of similar tuples.

Let us assume a client is connected to a server s and executes an *out* command in order to store a tuple tu in the SWARMLINDA network. Generally speaking, each *out* involves the following steps:

1. Technically the ant gets born on the server *s* at which the client performs its request. At its birth the tuple is already assigned to the ant. Based on the matching template the ant has a specific scent. In order to mark its trail the ant drops some pheromones on its current location. Thus surrounding ants will be attracted and are able to persecute the *out* performing ant. The idea is not only to cluster the information objects, on a lower level the ants should be already clustered in advance in the same spatial region and hence the probability of emerging tuple-clusters is more likely.

2. The ant looks at its current tuple space whether to store the carried tuple. Since the individuals follow the principles of locality and simplicity the decision is made of the concentration that means the constitution of templates at the local node (see Equation 4, page 22). Swarm based systems deal with probabilities. Thus the behavior is predictable but in a specific situation, one cannot say for sure how it ends up. However, the probability rises if the concentration of tuples matching the same template as the one carried is high. With it, the ant forces the environment to stay sorted and ordered in terms of clustering. This leads to a balanced system and reduces the effort of retrieving and storing tuples.

3. If the ant successfully drops a tuple at the current tuple space it spreads pheromones on the node and all surrounding neighbors in order to reinforce the location. The directly connected servers also notice the storage and update their scent tables. This circumstance reflects the biological situation since natural ants are able to track pheromones over a certain distance. It is not necessary that the ants should stay exactly side by side. In contrast to the born or movement situation where they drop pheromones only locally, the 'mission accomplished' case should be higher weighted. In fact, the tuple seems to fit in the context and therefore the locality might be also of interest for surrounding ants. After finishing its task there is no further need for this ant and thus it dies.

4. In case that the ant decides against storing the tuple it starts moving and heads for the next tuple space. Before setting itself in motion it has to choose one of the adjacent links connecting the surrounding neighbors. The principle is to follow already formed trails that has been taken by other individuals. Therefore the ant scans its neighborhood and decides stochastically which link it should take. The probability to move to a particular node is proportional to the amount of accumulated pheromones. Of course, only the scent which is similar to the tuple carried by the ant is of interest for the path selection. (see Equation 5, page 25).

5. Once the ant chose a particular link and went to the connecting node it gets older. In detail, there is an aging mechanism involved in the system. At the time the ant gets born a time-to-live (ttl) parameter is assigned to its properties. It limits the lifetime of the individual. Like biological ants they get born, start a 'career' in obtaining a specific position in the collective and performing their job until they die. Analogous the individuals behave the same way. Its age is an assurance mechanism that they get 'replaced' sometime by new ones. In particular, with a restriction in their lifetime

the ants are forced to drop their carried tuple at last when they die. Each time when an ant moves to another node it looses one of its life-points and thus gets older. Afterwards it continues the iteration with item 2.

Following the aforementioned steps it is likely to obtain a state of distribution of tuples among the nodes as shown in Figure 5. Since all actions which have been taken place in the scenario of a SWARMLINDA system are probabilistic and there is a stochastical influence, one cannot predict in advance a deterministic ending. But Figure 5 indicates one possible state of the environment. There is no guarantee that a new launch of the scenario will somehow generate exactly the same result. But it is very likely that it will look very similar. The small quadratic shapes represent nodes containing tuple spaces connected by links. Same colors indicate tuples matching the same template. Thus a region of same colors represent clusters. Similar tuples stay in proximity (surrounding neighborhood).

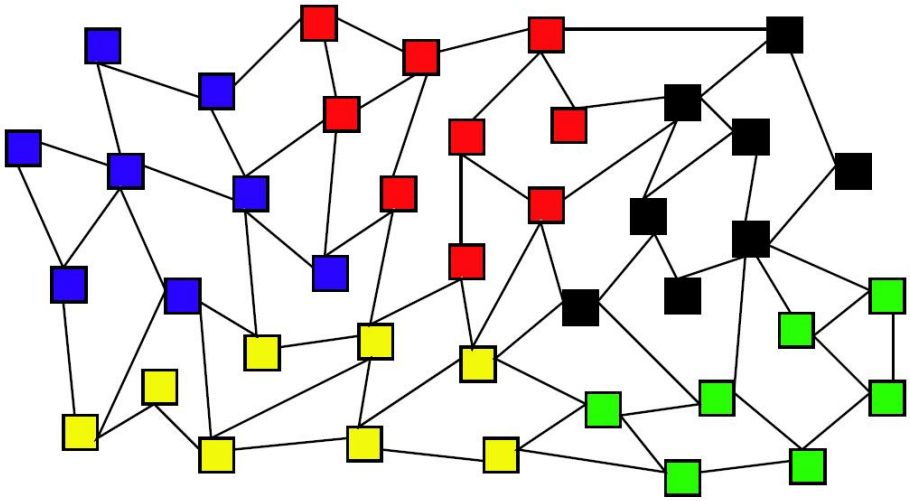


Figure 5: Idealistic distribution of tuples forming homogeneous clusters in a P2P SWARMLINDA network (adapted from [12])

However, the amount of pheromones on each node disappears over time in order to maintain adaptiveness. Tuples can be moved around in the network (see section 2.3, page 30) due to form new clusters as well as achieve more homogeneity inside the cluster by transferring misplaced tuples to other locations. If the scent trails are static the system will get lost in chaos since misleading trails would confuse the individuals. Thus it is necessary to introduce a disappearing of pheromones like in real nature given by an evaporation-rate $(1-\rho)$, with $0 \leq \rho \leq 1$ (see Equation 1):

$$Ph_i(t+1) = Ph_i(t)(1-\rho) \tag{1}$$

At each time $(t+1)$ each node i has a percental decrease of pheromones given by the evaporation-rate.

2.1.1. Drop Probability

As already mentioned the decision of storing a tuple in a tuple space depends on the concentration. That is the amount of tuples similar to the one carried by the ant. In order to classify an information object as similar one needs a similarity function. In this case the function is defined as follows:

$$sim(t_c, t_s) = \begin{cases} 1 & if\ template(t_c) = template(t_s) \\ 0 & otherwise \end{cases} \tag{2}$$

The similarity is determined by comparing the template of the carried tuple t_c and the stored tuple t_s which is located inside the tuple space TS. If the templates are not equivalent then the tuples are different. Applying the function for computing the concentration C (cf. [10]) involves Equation 2:

$$C = \sum_{\forall t_s \in TS} sim(t_c, t_s) \tag{3}$$

The probability of dropping a tuple P_{drop} at a tuple space is then defined as:

$$P_{drop} = \left(\frac{C}{C+K}\right)^2 \tag{4}$$

K represents the ttl value. It is obvious that the probability of dropping the tuple rises if the age of the ant increases and with it the ttl value shrinks. One can associate this behavior with the property that the older an individual gets the more it wants to finish its task and get retired. In biology muscles get weak and thus load seems to be heavier. Finally, the ant is able to dispose the tuple when its age tends to get maximized and thus the ttl value reaches 0. In this case Equation 4 returns 1 and the tuple gets stored automatically. Focusing on the concentration C it is easy to notice that P_{drop} increases while C grows by holding K fixed. The likelihood of a tuple to get stored, thus, correlates with C. Tuples get attracted by those tuple space since they already form a big cluster.

2.1.2. Path Selection

If the ant decides against storing the tuple it continues moving to the next node. In order to establish clustering of tuples one may notice that it makes sense to let the ant head in the direction where it is very likely that similar individuals are around. That is achieved by selecting the node containing the largest amount of pheromones which is similar to its own. Figure 6 shows the natural behavior of ants which are looking for food or trying to store some items in ant hills. The circle shapes indicate nodes containing tuple spaces which are connected by links. The crossing ants represent the individuals which are heading towards a node. On the left side of the network one may notice a huge amount of several ant types. The colors indicate different kinds of tuples or templates. This can be either tuple- or template-ants. It does not matter to which one they belong to because both of them trying to find locations with similar tuples around. Tuple-ants try to store their carried tuple; however, template-ants trying to retrieve. As already mentioned colored

individuals are interested in the same kind of tuple. The black nodes are neutral. They do not contain any types of tuples, but they may contain pheromones. The node color is an indicator for emerging clusters only. The ants on the left side are coming from miscellaneous servers. They are not grouped yet, which may infer to the following assumptions:

- They have just been born in the network on one of the previous servers, respectively.
- The places they visited before were not attractive; since attractiveness depends on their task, they could not track suited pheromone trails.
- A small fraction of them may scout-ants; they explore the environment. By doing this they may find new suitable spots.
- There was some level of pheromones but they were born geographically isolated from each other (e.g. in border regions). Further on, the emerging cluster are located in the center of the network. Therefore the ants have to move towards this spot by following scent trails. They come from different directions and finally meet in the center.

But to abstract away from the circumstances of their arrangement, one should pay more attention to the right side that are the links connecting the central node with the colored ones. Due to pheromone trails the individuals get separated from each other and form new groups. They get attracted from arising clusters which are represented by the colored nodes. However, a SWARMLINDA system deals with probabilities; all actions which have been taken place are based on certain random factors. In fact, it is possible that an ant heads towards a node which is not appropriate for its purposes. But that does not impact the system. Contrariwise, it enables a new route to the sink which may lead to take a few hops more. It provides an alternative path. This involves some level of load balancing as well as establishing fault tolerance. If the main route becomes a bottleneck or disappears due to a failure, the ants tending to use the alternative path as shown in Figure 7.

The graph in Figure 7(a) shows active individuals carrying objects from node *B* (food source) to node *G* (ant hill). As one may see, most of the ants moving over node *D* that is the shortest path to the sink based on hop counts. It is also noticeable that some ants do not follow the emerging ant trail, they taking alternative routes and thus exploring the surrounding neighborhood. In fact, these individuals may take longer in terms of time for arriving at the ant hill. Conversely, if the amount of ants on one link exceeds a certain threshold they can cause a congestion since the network traffic rises rapidly. In this case, although it is the shortest path, they require more time for arriving at the sink. The scenario is equal to a road which is used by too many cars. At a certain level it will result into a traffic jam. In order to find a remedy, it is necessary to build beltways for redirecting a partition of the traffic. Thus it makes sense to balance the amount of ants by spatial separation.

By doing so the individuals are aware of alternative routes. Therefore they will take the additional paths for heading towards their destination. Anyway, if the main ant trail becomes blocked due to a broken network cable the ants are able to use the alternative

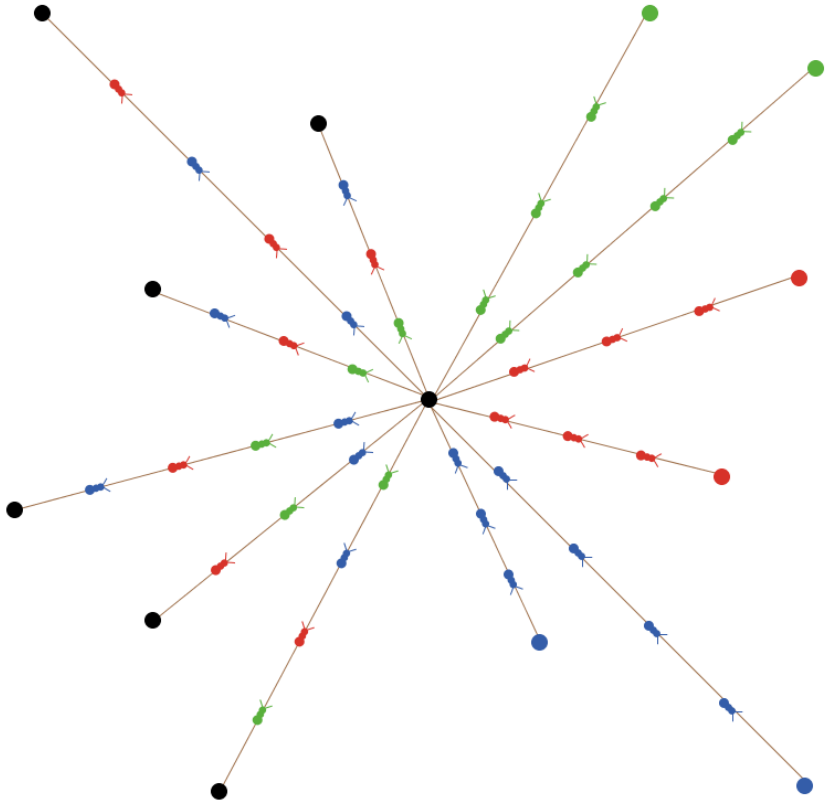
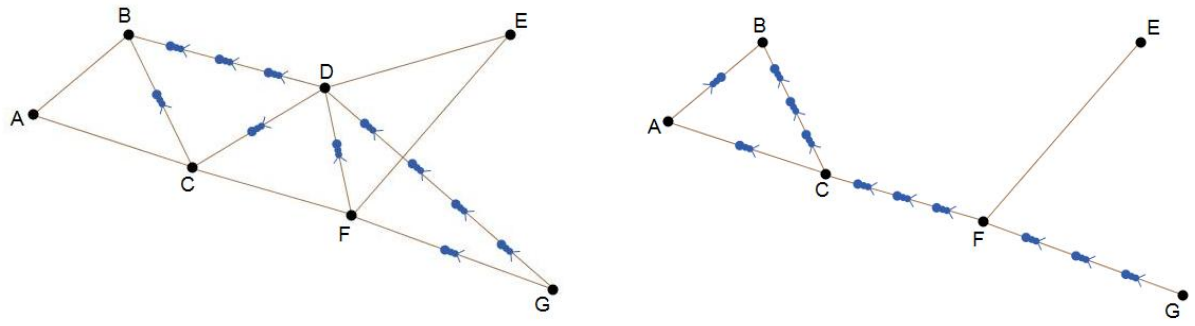


Figure 6: Self-organization of roaming ants in a SWARMLINDA network achieved by tracking pheromones

route. Even worse, when a complete server failure appears like in Figure 7(b) then all connecting links are also lost. Node *D* disappears but it is obvious that the individuals tracking pheromones on node *C* as well as *F* since these servers have already been visited by former ants as in Figure 7(a). Now, most of the individuals taking this route and forming a new ant trail which is under the given circumstances the shortest path from *B* to *G*. In fact, a few ants exploring again the neighborhood as they did before the node disappeared. If the connection from *B* to *C* breaks down then the ants are forced to take the route via node *A*. That is the last link that can disappear otherwise the graph is not coherent anymore and thus some nodes may not be reached by ants. However, since this example is only a scenario for demonstrating the path finding, a real SWARMLINDA environment usually contains much more nodes and connectivity.



(a) Roaming ants carrying tuples from node B (source) to node G (sink). Shortest path (number of hops) is chosen via node D.

(b) Alternative routing over node C and F due to a failure of D.

Figure 7: Path finding using pheromones. Node failures and resulting disappearance of connections does not affect the routing reliability and robustness of a SWARMLINDA system.

The selection of the next node is determined by applying Equation 5. As mentioned earlier the path selection is based on a probabilistic function. The probability of taking the link from node i to j is given as follows: the numerator is the summation of the concentration C and the amount of pheromones Ph of j divided by the sum of C and Ph of all nodes n in the neighborhood NH of i. Ph represents the kind of pheromone which matches the tuple carried by the respective ant. Usually, nodes containing a huge amount of tuples also hold significant pheromone trails since it is very likely that these nodes are involved in many interactions with tuple- or template-ants. However, it is also possible that a specific node l is not visited for some time. If there is neither a consumer nor a producer of a specific tuple type which is located at l, there would not be any much ants traveling to l. In the scenario that after a certain time a consumer wants to retrieve a tuple located at l it may be hard to find it since the pheromone trail already disappeared. In this case the ants, nevertheless, get attracted from l since it contains a huge amount of matching tuples. On the other hand one can assume that a node contains a big amount of pheromones but it does not contain tuples of interest. In this case it is likely that a surrounding node contains tuples of interest, so that ants get attracted as well by being in the proximity. However, if the neighbors neither contain any tuples of interest nor any pheromones the node selection for the next step is performed randomly. There is also a mechanism which prohibits the ants to move backward that is the direction where they came from. An exception occurs when this is the only possibility.

$$P_{ij} = \frac{C_j + Ph_j}{\sum_{\forall n \in NH(i)} (C_n + Ph_n)} \tag{5}$$

2.2. Tuple Retrieval

This section deals with the retrieval of tuples in the SWARMLINDA network. In contrast to conventional LINDA systems tuples cannot be found using hash functions. Remember, LINDA advocates to sense tuples by hashing: $node\_id = H(tuple)$. As shown in Figure 8 a tuple is found by applying a hash function. The resulting value tells the physical storage location (node_id). Analogous to tuple distribution this mechanism suffers under performing rehashing while servers are added on the fly as well as get removed. On the other hand the function delivers a unique ID and thus it is hard to form clusters so that similar tuples can stay in proximity to each other.

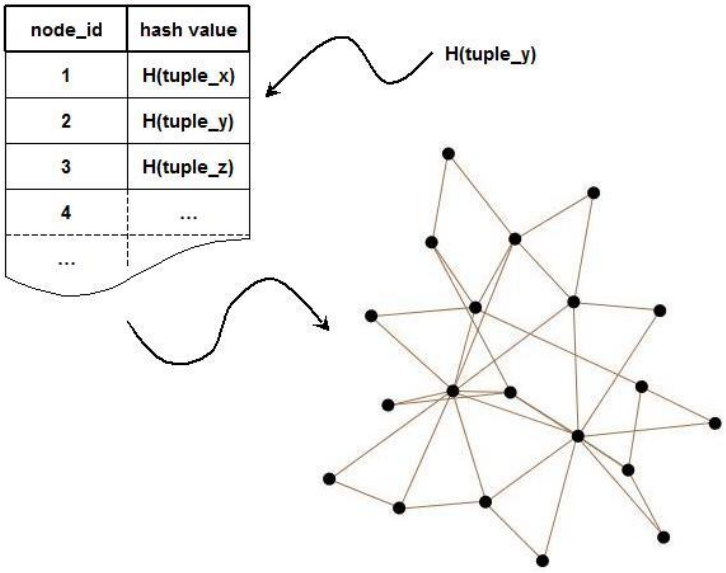


Figure 8: Retrieval of tuples via hashing in LINDA

SWARMLINDA instead does not use look-up tables in order to find and retrieve tuples. Clients are able to connect somewhere with the SWARMLINDA network. They submit a request and query a specific kind of tuple. The server to which the client is connected to instantiates a mobile agent (ant) analogous as described in section 2.1 (page 18). In particular, the request to the server invokes the SWARMLINDA *in*- or *rd*-primitive. Although they are doing almost the same, the semantic is slightly different. While *in* retrieves the actual information object by removing this item from the respective tuple space and thus changes the SWARMLINDA environment, *rd* only takes a copy and returns it to the requester. For implementation issues the method signature may look like $in(pTemplate)$ or $rd(pTemplate)$. Similar to the *out*-primitive only a connection object is required for performing the command. The parameter $pTemplate$ is a local instantiated object containing the search criteria. The structure of the template is similar to the one of the tuple since both of them appear in the form $(v_1, v_2, ..., v_n)$ with $\forall v_i \; \exists u \in U$. But tuples are forced to have concrete values from U.

Templates, however, can use some abstraction level, they can use data types as values. This allows the requester for searching a specific tuple like (1, "*This is a string*") as well as (*Integer*, *String*). Also mixed forms are possible like (7.5, *String*). Analogous to tuple distribution the template objects are not commutative as well as symmetric, $(a, b) \neq (b, a)$. By submitting a request with template parameters like (19.5, *Double*, *Integer*) the following tuples would match although they are different: (19.5, 7.8, 10), (19.5, 34.1, 3). The tuple which appears first to the searching ant is chosen for retrieval. During looking for tuples the individuals orientate themselves by tracking pheromones from previous crossed ants. While tuple distribution uses tuple-ants as active instances, tuple retrieval focuses on template-ants as the operating actors. They carry a template and trying to find a matching tuple. The server to which the client is connected to represents the ant hill. On implementation level the ant hill is a factory object which is able to instantiate the ants.

The routing mechanism in order to sense tuples is similar to the one mentioned in section 2.1 (page 18). However, additionally to tuple distribution which is a one way approach - tuple-ants heading from the source to the sink - tuple retrieval, in contrast, deals with finding a return path back to the ant hill (start and instantiation location) where they came from. Since the client wants to obtain the tuple the ant is responsible for delivery. In order to guarantee that the individual is able to find the way back it has a memory. So it keeps the visited nodes in mind while it performs its task. In [30] is shown that ants own a small memory and only remember their last few hops and continue finding their ant hills (start locations) by tracking the significant scent of the ant hill. In contrast, this study is based on keeping each visited node in mind. Once an ant is on their return path it remembers all hops back to its birth place. It avoids tracking scents of the respective ant hill. As mentioned in section 1.5 (page 16) the simulation is based on networks which has been generated according to the approach of Barabasi [3]. Those networks often referred to as social networks lead to a small-world phenomenon [34]. The characteristic of the arrangement of edges and vertices shows the six degrees of separation [23]. That means that each node can be reached by any other node in the network by using six hops in average. According to this principle the template-ants remember each visited node.

Let us assume a client is connected to a server *s* and executes an *in* command in order to retrieve a tuple *tu* from the SWARMLINDA network (see Figure 9, page 29). Generally speaking, each *in* as well as *rd* involves the following steps:

1. Technically the ant gets born on the server *s* at which the client performs its request. The submitted query parameters are transformed to a template which is assigned to the ant at its birth. Based on the template the ant has a specific scent. At its instantiation the ant obtains an initial empty memory.

2. In order to remember the visited nodes it adds its current location to its memory. The ant looks at its current location whether the tuple space contains a tuple which matches the template it is carrying. The best case appears when the ant finds an appropriate tuple at the first server it visits. It would not need to roam around in the network and has immediately done its task. This optimizes the request by minimizing the query time. On the other hand it saves bandwidth of the network.

3. If the ant successfully found a tuple it depends on the executed command how to proceed. While *in* removes the chosen tuple from the tuple space, *rd* only takes a copy and leaves the actual tuple at its place. However, after obtaining a suitable tuple the ant finds its way back to the ant hill by using its memory. Once it starts its return path it drops pheromones on each node. Therefore a successful trail is marked for finding similar tuples like tu.

4. If the ant cannot find an appropriate tuple it moves on and heads towards the next tuple space. The movement phase is similar to the one in tuple distribution. Before the ant sets itself in motion it has to choose one of the adjacent links connecting the surrounding neighbors. The principle is to follow already formed trails that has been taken by other individuals. Therefore the ant scans its neighborhood and decides stochastically which link it should take. The probability to move to a particular node is proportional to the amount of accumulated pheromones. Of course, only the scent which is similar to the template carried by the ant is of interest for the path selection. (see Equation 5, page 25).

5. Once the ant chose a particular link and went to the connecting node it gets older. Similar to tuple distribution its ttl value decreases by one. If the ant cannot find a matching tuple within its given time it dies in the network and the request fails. In this case one can assume that there is no suitable information. The limitation of its search avoids to much traffic in the system. Otherwise they would consume to much system resources. Additionally, it makes no sense after some time to continue the search. The probability that there is an appropriate tuple somewhere tends to get very small. However, if the ant is still alive it continues the iteration with item 2.

Focusing again on the diverse implications of the two retrieval mechanisms leads to the following: the semantic varies since after performing an *in* related ant processes searching for the same tuple may not find it here. Instead they have to move to another location and thus influencing the system slightly different. They remove a tuple elsewhere and drop scent on other nodes. There might be an assumption that *rd* behave better since the information is still in the network and thus can be found by other processes as well. But, conversely, if one thinks about freshness of information objects *in* balances the system that in average the tuples cannot get old since they already have been removed. Therefore fresh information objects can be spread in the network. By searching tuples it is very likely that these objects get retrieved. The SWARMLINDA environment is affected and changed to a diverse state. One cannot say in general, if the modification behave better or worse. There is no abstract evaluation for the commands. However, it would make sense to review *in* and *rd* dependent on the context concerning a specific postulated system behavior.

As mentioned before the ant drops some pheromones on the visited nodes after it successfully found a tuple. In order to express the direction which has been taken by the ant the amount of pheromones differ for each involved node. The memory of the ants is a list: $memory(n_1, n_2, ..., n_k)$. The visited nodes are added at the end. Therefore n_k is the node where the tuple has been found while n_1 is the ant hill. The amount of pheromones

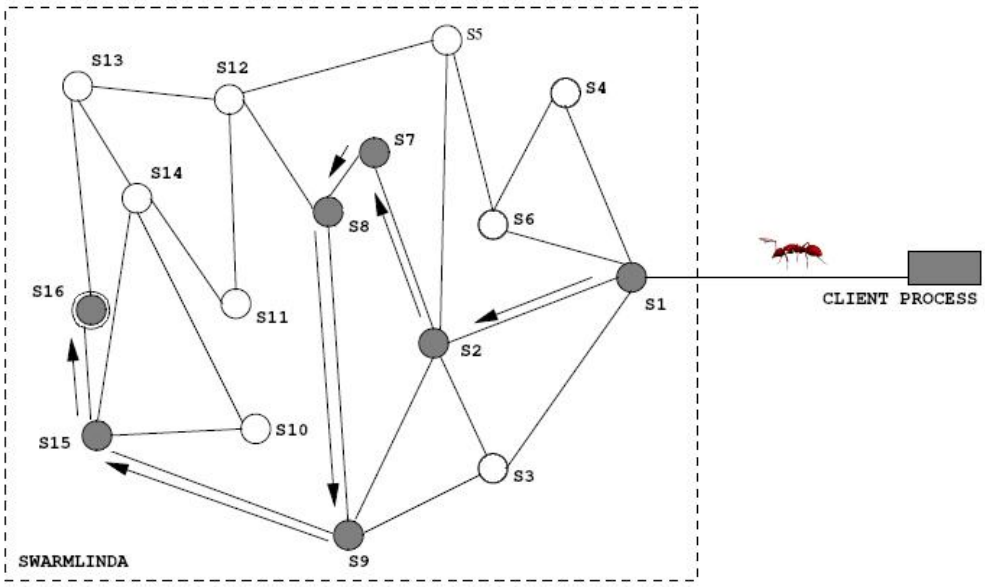


Figure 9: Retrieval of tuples by tracking pheromones in SWARMLINDA (adapted from [30])

which should be dropped on the nodes depends on the compression coefficient $(1 - \rho)$, with $0 \leq \rho \leq 1$ (see Equation 6). It defines the level of decrease of pheromones for the respective nodes.

$$\begin{aligned} Ph_{drop}(n_k) &= 1 \\ Ph_{drop}(n_{i-1}) &= Ph_{drop}(n_i)(1 - \rho) \end{aligned} \tag{6}$$

The intention for introducing the compression coefficient is to emphasize the proximity to the source of successful matched tuples. Figure 9 shows the marked ant trail from the ant hill S_1, to which the client is connected, to the destination node S_{16}. By reaching S_{16} the ants memory comprises the following nodes: $memory = (S_1, S_2, S_7, S_8, S_9, S_{15}, S_{16})$. Afterwards it picks up the tuple and returns to its ant hill and leaves some amount of pheromones according to Equation 6 at the nodes it remembers. Assume that the scent tables of the involved nodes are empty. Finally, they get updated as shown in Table 1.

node	S_1	S_2	S_7	S_8	S_9	S_{15}	S_{16}
scent value	.53	.59	.66	.73	.81	.9	1

Table 1: Pheromone distribution among the visited nodes, with ρ = .1

This mechanism of distributing the amount of pheromones among the nodes increases the probability that ongoing and prospective ants will find the new discovered tuple space more easily. Assume a new client process which is connected to S_3, performs an *in* and looks for similar tuples as the one in the aforementioned scenario. According to

the principles of SWARMLINDA's *in* primitive the ant gets born on S_3. It cannot find a matching tuple, thus it scans its neighborhood. Looking at the scent table for node S_1, S_2 and S_9 (see Table 1) the ant has to decide where to go since all neighbors contain pheromones. Although the individual may go to S_1 or S_2 it is more likely that it will continue its search by heading towards S_9 since the amount of pheromones is even larger. Applying Equation 5 (page 25) the probabilities for moving towards to one of the three nodes is shown in Table 2.

node	S_1	S_2	S_9
%	27	31	42

Table 2: Probability distribution between S_1, S_2 and S_9 applying Equation 5 (page 25)

Anyway, assume that the ant decides against moving to S_9 and instead follows the trail to S_1 or S_2. In fact, first the ant dissociates itself from its destination. But this does not lead to a bad impact since it is very likely that the ant will move from S_1 to S_2 in the next step since there is no other scent in the neighborhood of S_1. Arriving at S_2 the ant has the possibility to head for S_7 or S_9. Again, it is more likely to choose S_9. Although it took a detour over S_1 and S_2 the ant may go on to S_9 and thus balances it since it does not move over S_7 and S_8.

However, the worst case for the ant is the trail $[S_3, S_1, S_2, S_7, S_8, S_9, S_{15}, S_{16}]$. In fact, that would even take longer than the previous ant needed. But after the ant finds its tuple it reinforces again the trail. Thus the scent tables get updated as shown in Table 3. The evaporation of pheromones are left out for simplifying the scenario. However, after some ants appeared in the network and were looking for similar tuples the trail will get more significant and thus the probability of taking the shortest path gets more likely.

node	S_3	S_1	S_2	S_7	S_8	S_9	S_{15}	S_{16}
scent value	0.48	1.06	1.18	1.32	1.46	1.62	1.8	2

Table 3: Update of the pheromone distribution among the visited nodes, with $\rho = .1$

2.3. Tuple Movement

This section deals with the movement of tuples in the SWARMLINDA network. The aforementioned mechanisms shows SWARMLINDA's way of distribution as well as retrieval of tuples. The algorithms for performing *out*-, *in*- and *rd*-commands are represented in detail. Both mechanisms behave in a non-deterministic way and thus one cannot predict in advance the exact distribution of tuples among the nodes as well as the spatial formation of clusters. Analogous to that it is not clear from which node a template-ant will bring a tuple. Several executions, although containing the same request, may lead the ant to different nodes. One can also not foresee in which geographical region clusters will arise. However, based on the system behavior of a SWARMLINDA certain characteristics are predictable, e.g. formation of clusters generally, emergence of pheromone trails, etc.

Assume the following scenario: a SWARMLINDA system is initially empty. The environment comprises nodes and links connecting it. The formed graph is coherent. Now, several *out*- and *in*-commands are executed. After some time we take a snapshot of the system at runtime (see Figure 10).

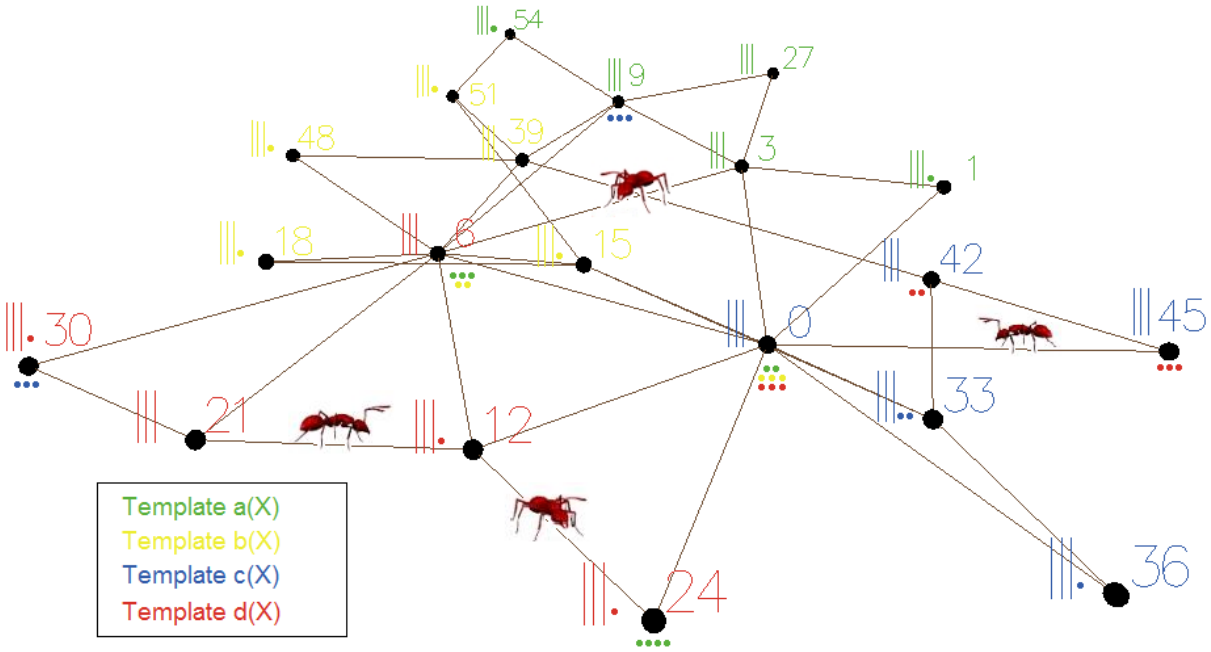


Figure 10: Snapshot of a SWARMLINDA system at runtime

Like in earlier scenarios the color expresses the kind of tuple (template) which is stored at the respective tuple space. The number which is assigned to the tuple spaces is the node ID. Its color represents the majority of tuples matching the same template. Close to it the amount of tuples is represented by the vertical bars (|) and dots (.). Each bar counts for ten tuples while the dots represent a single tuple. Most of the nodes are single colored. In that case the tuple space contains only tuples matching the same template. Thus the tuple space is totally homogeneous. Conversely, some of the nodes are heterogeneous since they contain tuples matching different templates, e.g. 0, 6, 9, 24, 30, 42, 45. The circumstances for the current distribution involves the following:

- In the beginning ($t = 0$) the system was initially empty. Without performing seeding - putting some tuples directly on nodes for setting up the system - there is neither scent nor tuples in the network. Thus the first ants are not able to orientate themselves. Therefore they roam around in the environment, their routing is based on random walks and the probability of dropping a tuple tends to be infinitesimal ($\lim(P_{drop} \rightarrow 0)$). It is very likely that they finally drop their tuple because they are running out of time, so they die. As time passes by pheromone trails emerge and

some tuple spaces get more attractive to some ants since they contain more similar tuples. This is the result of the scout ants. Ongoing and prospective ants are more able to scan the environment and transport their tuple to a suitable location. Also template-ants perform better since they can track pheromones now. In fact, this behavior results in a state where some tuple may be misplaced since they do not fit in the context.

- The nodes in the middle of the network (0 and 6) are the so called hubs[4]. They form the backbone. The characteristic of hubs is the connectivity since they own the largest amount of links. If they break down the system receives a severe impact. On the other hand, if one of the spokes[5] gets shut down it would only slightly affect the system, if any. In this case if node 0 and 6 are shut down the network gets separated into two isolated parts. Thus the graph is not coherent anymore.

  However, based on the fact that hubs owns many links the probability that they get visited by ants is, in fact, larger in contrast to spokes. The probability of visiting a certain node is proportional to its amount of links. This theory is valid as long as no pheromones are in the network. If the ants start to spread their scent among the nodes the probability of visiting a node gets more influenced by the respective amount of pheromones. Thus the importance of multiple connectivity shrinks. But, nevertheless, they maintain more relevance than light connected nodes. In Figure 10 (page 31) one can see that node 0 and 6 indicates more heterogeneity than the remaining ones. Therefore, one can assume that, especially in the beginning, the probability of visiting those nodes by ants was even higher than the others.

- One may notice that the distribution of tuples, that is the amount of tuples stored in the respective tuple spaces, is almost equal. This phenomenon appears because of the appliance of the overclustering-avoidance strategy which is described in section 4.4 (page 75).

The situation that there are somehow heterogeneous clusters does not show an advantage. It is very likely that tuple- as well as template-ants get attracted by nodes that contain many tuples matching the same template. Conversely, the tuples which are in the minority and occur some kind of misplaced are not of interest for most ants. Thereby the pheromone trails leading to those tuples evaporates fast over time since it does not get reinforced much. It is, in fact, possible that they rest there for a long time. Since unused system resources does not result in an improvement it may be better to move them to a location where they fit in the context. The transportation of the tuples, so that they feel more comfortable in their new neighborhood, increases the probability that they can be found by ants. Thereby more clusters achieve homogeneity. Pheromone trails get marked more significantly. In total, the system performance gets increased by adequately using the available resources.

[4] Hubs are located in the center of a network and connecting almost spokes, thus they own many links.

[5] In most common cases spokes are located in border regions containing less links and are connected to each other via hubs.

The tuple movement mechanism is based on a specific kind of ant that performs the transportation. One can interpret this individual as a combination of tuple- and template-ants. This new kind is called cleaning-ant since it tries to establish a higher level of order. Thus its task is to reorganize the environment. That means it collects tuples from nodes and conveys them to a different location. It tries to find a more suitable place for tuples. The tuple movement process involves the following steps:

1. The cleaning-ant which is the active individual during tuple movement gets born on a server s. At its instantiation time the ant is neutral. In contrast to tuple distribution and retrieval the ant does neither carry a tuple nor a template. Thus it does not emit pheromones since it is scentless. It only gets an ttl parameter assigned in order to limit the activity of the ant.

2. Since the objective of tuple movement is to establish a higher level of order the individual looks at its current node in order to find a tuple which may not feel comfortable at this location. The decision for picking up a tuple is based on two factors: the local node entropy and a fitness value. The system entropy defines the level of order in the network. High entropy values show a chaotic organization of tuple while low entropies indicate order. However, the node entropy is part of the system entropy and defines the level of order on the layer of tuple spaces. Section 4.2 (page 66) deals with the system entropy in detail. On the other hand the fitness value indicates how suited is a tuple at its current location. This involves the ratio between groups of tuples at the local tuple space and its neighbors. For instance, looking at Figure 10 (page 31) node 0 contains all four kinds of tuples indicated by the respective colors. The tuple space is comprised of the following groups: 2 tuples of template $a(X)$, 3 of template $b(X)$, 30 of template $c(X)$ and 3 of template $d(X)$. Therefore $c(X)$ claims the majority and is directly attached to three further nodes also owning the majority of $c(X)$ (node 33, 36 and 45). The group of $a(X)$, $b(X)$ and $d(X)$, respectively, seems to be misplaced and thus should be moved to another tuple space. It is obvious that there are two nodes claiming the majority of $a(X)$ (node 1 and 3) which are directly connected to 0. The minority of $a(X)$ tuples should be transferred to those nodes. Analogous $b(X)$ tuples should be moved to node 15 while $d(X)$ tuples can be moved to node 6, 12 or 24.

3. If the ant does not find a misplaced tuple it heads for the next node by randomly choosing one from its neighborhood. Since it carries neither a tuple nor template it is scentless and thereby unable to track pheromones. After arriving at its selected node it continues the iteration with item 2.

4. Finally, if the ant finds a misplaced tuple it picks it up by removing it from the tuple space. This involves the same semantic as the *in*-primitive. By doing so, the ant adopts the scent of the tuple and mutates to a tuple-ant by following the *out*-primitive. During the metamorphosis its ttl value gets refreshed by assigning it the same value as usual for tuple-ants. One can see the metamorphosis as a rejuvenation of its physical strength.

However, the following steps are the same as in tuple distribution (section 2.1, page 18) by starting at item 2 since the ant is already born. According to the *out*-primitive the ant dies after successfully finishing its task or by reaching its maximum age.

After describing tuple movement one can raise the question what mechanism triggers the process of initiating tuple movement. Observing the natural behavior of ant colonies there are some types staying always in proximity to the ant hill. Some of them only work inside the mold while others protect it from enemies. They are moving outside but stay close. Back to the issue of tuple movement one can think of an ant which observes the items brought to the ant hill. Thus it is aware of the fractions of different tuple types. Since they know their neighborhood very well and so the distribution of tuples there, they are able to decide whether it makes sense or is necessary to move tuples towards another mold.

This observing ant can be interpret as a time controlled instance which gets activated after a specific time. If the observer decides that a transport is useful it causes a certain amount of cleaning-ants to get instantiated for reorganization issues. However, the amount of cleaning-ants can be determined by applying a function dependent on the level of order. Afterwards the observer can sleep for a certain amount of time st_{sleep}. The duration depends on the action the ant took. If the ant decides against triggering cleaning-ants the timespan gets increased by some Δr while activating the cleaning-ants decreases st_{sleep} by some Δr. Equation 7 shows the computation for the ongoing sleep time, with $0 \leq \rho \leq 1$:

$$st_{sleep}(t) = \begin{cases} st_{sleep}(t-1)(1-\rho) & if\ cleaning-ants\ triggered \\ st_{sleep}(t-1)(1+\rho) & otherwise \end{cases} \tag{7}$$

3. Development and Implementation

This section deals with the development of a SWARMLINDA system and handles implementation details. At first, the used IDEs[6] are presented. This involves the issues for choosing these development environments. Further on, the simulator for network generation as well as the SWARMLINDA simulator are described in detail. This includes class diagrams and the structure of the system architecture.

3.1. Used IDE

In the following the NetLogo [61] and Eclipse IDE are presented. Both were used for developing the SWARMLINDA system.

3.1.1. NetLogo

3.1.1.1. Definition

NetLogo is a cross-platform multi-agent programmable modeling environment which is entirely written in Java[7]. The development of the simulators are based on the NetLogo version 3.1.4 which was the current version at development time. This version is implemented in the Java version 1.4.1. Thus it does not provide the feature of Java generics which has been introduced with the 5.0 (1.5.0) version. For the implementation of the simulators the JDK[8] version 1.5.0_11 was used. Due to updates, the final executions of the simulator were done in the JRE[9] 1.6.0_03.

One may ask of the derivation of the name: The part 'Logo' is chosen because NetLogo is a dialect of the Logo programming language [57] which is used for functional programming. The first part 'Net' is chosen to "evoke the decentralized, interconnected nature of the phenomena you can model with NetLogo, including network phenomena" [35]. It also refers to HubNet, the "multiuser participatory simulation environment included in NetLogo" [35]. NetLogo is a project developed at the Northwestern University in Evanston (Illinois) under the supervision of U. Wilensky in 1999.

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It can comprise thousands of agents which operate fully independently in the environment. Agents are threads performing some tasks. The developer can assign different instructions to the agents. During the runtime of a simulation the user is able to send additional commands to a single agent, a group of agents as well as to the whole population. The intention is to observe complex phenomena in collective behavior although each agent operates fully autonomous. This also refers to swarm behavior. Since

[6] Integrated Development Environments

[7] Java is an object-oriented programming language originally developed by Sun Microsystems and released in 1995 as a core component of Sun's Java platform

[8] Java Development Kit is a product of Sun Microsystems and has been released under the GNU General Public License (GPL) as OpenJDK

[9] The Java Runtime Environment is the software used to run any applications and is part of the JDK

NetLogo is entirely written in Java it is platform independent. Therefore it can be installed on Windows, Linux and Mac OS X operating systems.

NetLogo is also a programming language which comes along with the development environment. It defines a set of built-in primitives which can be used for creating, destroying as well as interacting with agents.

3.1.1.2. Usage

NetLogo is a clearly arranged and profitable system simulation tool. It provides a spatial separation between the controlling and the visualization area. Therefore the parameter configuration and adjustment is isolated from the actual simulation. Figure 11 shows the typical NetLogo view. The window on the left represents the actual development environment. The small size of the window is arranged in order to fit in the figure. However, the IDE provides a usual menu which can be used for navigation issues, configuration as well as support for the implementation. It is the first panel arranged in top of the application. Below, one can find the selection tabs for switching between the interface (current selection), the information guide and the implementation section. The navigation tabs are followed by the design panel. It provides several different components for the creation, adjustment and arrangement of control elements such as buttons, sliders, switches, etc. Additionally the widgets can be attached to global system variables that can be accessed by the application. That is feasible for sliders as well as drop down menus. On the other hand implemented procedures can be assigned to buttons. A complete listing of control elements supported by NetLogo can be found in Table 4.

The center location provides much space for placing the widgets. The control area is the main part for setting up and adjusting the configuration parameters for the actual simulation. The simulation can be interrupted, stopped as well as reconfigured. In Figure 11 one can see the *go* and *setup* button. NetLogo is based on the convention that each simulation shall have these control elements. While *setup* postulates a call of the identical procedure setting up some sort of system variables, *go* starts the simulation. On the bottom right of the 'go' button one can see two contrary arrows indicating a recursive sign. This is NetLogo's emblem for a forever button. The concept behaves like `while(true){ /*do something ..*/}`. Forever buttons can be interrupted by disabling the widget or the usage of the *halt* control sequence which comes along with NetLogo.

The command center is connected to the bottom of the control area. It is divided into the output area and the input line. NetLogo's term for agents is turtles. They can be further grouped by assigning them to a specific breed, e.g. ants, termites, bees, etc. Looking at the command center one may notice that the interaction engine is in observer mode. This forces NetLogo to accept any instruction statement for the total environment. Beside turtles it is possible to address patches. They form the terrain which is 2D and was later extended to 3D. The command center can also be set into turtles as well as patches mode. This enables direct and exclusive communication with those parts of NetLogo. Every executed command and response from the system will be written in the output area.

The right window of Figure 11 (page 37) represents the visualization area. The image shows a 3D terrain consisting of nodes connected by edges. The degree of each node,

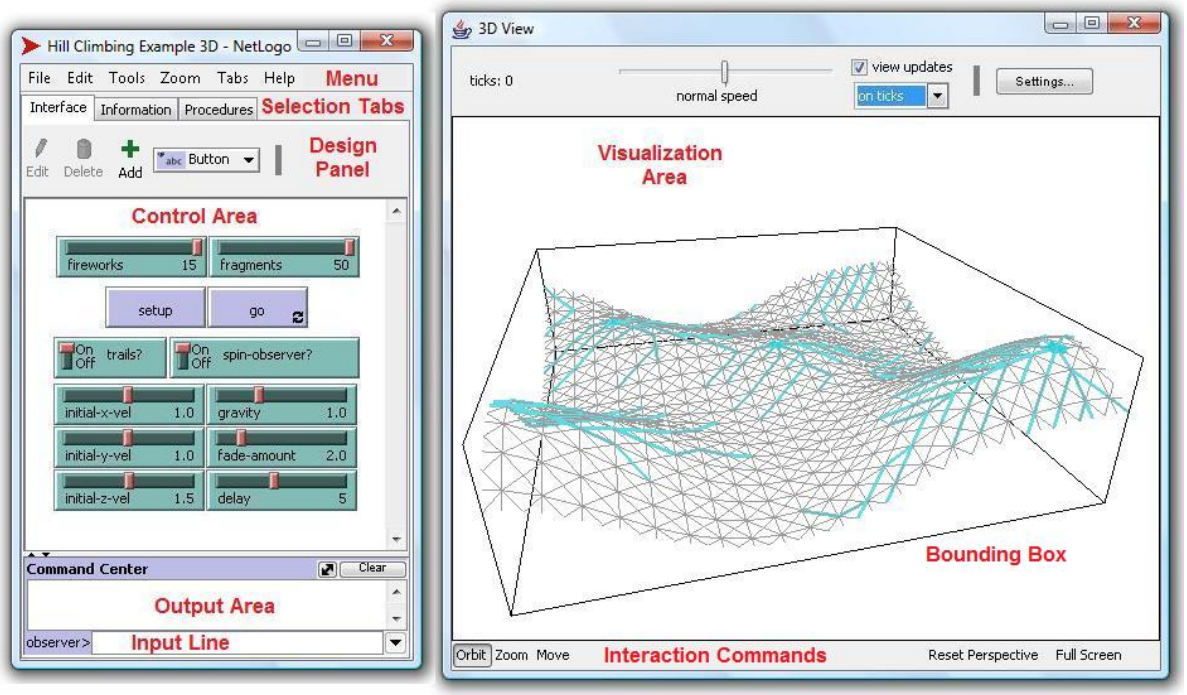


Figure 11: NetLogo controlling area (left) and visualization window (right)

which counts the number of connected links, is equal as well as the arrangement of the links. This pattern forms a grid which shows the topology of mountains. The marked spots are trails left from ants climbing uphill. All meet at the top of the mountain. The environment is captured in the bounding box. Panels for interaction are arranged at the top as well as at the bottom of the visualization area. Therefore the model can be scaled, rotated and moved.

By changing the selection tab to information one will find plenty of information which explain and describe the current model. Commonly issues for the development and detailed instructions of how to setup as well as configuring the model are exhibit. It also

Control element	Description
Slider	Allows to select a numerical value within a given range.
Button	Allows to execute procedures.
Chooser	Allows to choose a discrete value from a list.
Switch	Enables\ disables a global boolean value.
Monitor	Monitors a system variable.
Plot	Creates a plot which can be used for drawing graphs.
Output	Creates an additional output area (max. 1).

Table 4: Control elements in NetLogo

gives advices of the kind of phenomenon which can be observed. Sometimes some hints are provided in order to exploit a specific system configuration which emphasizes the system behavior the most. It is also possible to add proposal for extending the simulation.

Finally, the implementation selection tab provides a development environment for implementing source code in NetLogo. In fact, the IDE is not comparable to Eclipse, IntelliJ IDEA or NetBeans but, nevertheless, provides syntax highlighting as well as fast procedure access. Similar to VBA[10] NetLogo does not support explicit assignments of data types. A variable gets a type assigned automatically by initializing it. The IDE also includes a verification tool which validates the current source code according to the NetLogo syntax. In case of violation the developer gets an error message.

3.1.1.3. System Architecture

This section deals with the system architecture of NetLogo. Unfortunately, the source code is not published yet. From the current point of view it is not predictable whether the supervisors of NetLogo from the Northwestern University will release the implementation details someday as an open source project or, at least, provide the source code. But, based on the FAQs[11] of the NetLogo homepage, they are looking forward to "eventually releasing the source under an open source license" [35]. Even for private issues, like this report, the source code would be advantageous. Nevertheless, they support the NetLogo API[12].

In fact, they allow the access of NetLogo code on Java level. By using the extension mechanism as described in section 3.1.1.4 it is feasible to interact with the NetLogo system core. Due to this openness one can take advantage of it by implementing the own Java classes. Thus it is possible to introduce new semantics and primitives in NetLogo. It is also possible to share the generated models of NetLogo with the user community. Models from other contributors can be downloaded from the NetLogo website and be integrated without severe effort.

Figure 12 shows the hierarchy and inheritance of some NetLogo classes in the UML[13] diagram. As one may see the `Agent` class is the super class (from the `org.nlogo.agent` package) for the drawn NetLogo instances. It is subclassed from the `Observable` class which comes along with the `java.util` package and is part of the JDK. `Observable` classes represent data objects in the model-view paradigm. They are used in an application to allow other components of the system to monitor them. In detail, all instances which have been registered for observing an `Observable` object *obs* gets notified by calling their `update` method if *obs* has been changed.

One may associate the `Agent` class with a mobile agent performing some tasks. To avoid confusions, the `Agent` class is an abstract base class which provides common parameters as well as methods. The `Turtle` class implements mobile agents like the roaming ants used

[10] Visual Basic for Applications
[11] Frequently Asked Questions
[12] Application Programming Interface
[13] Unified Modeling Language

in SWARMLINDA. The `Link` class is used for indicating a connection between two objects of class `Turtle`. The most common usage for links is to connect nodes. They are modeled using the `Turtle` class. The `Patch` class determines the space, i.e. the terrain in which the simulation takes place. Finally, the `Observer` class is the monitoring instance.

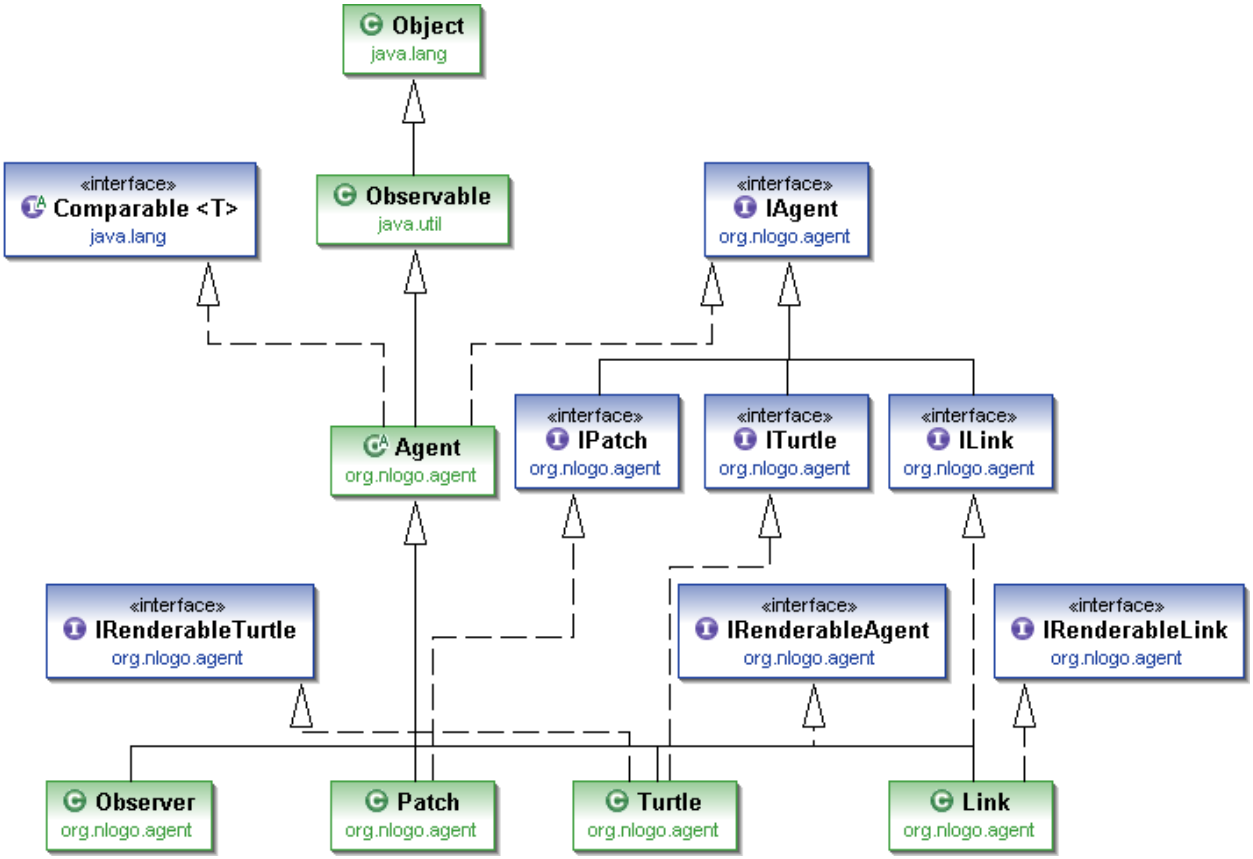


Figure 12: Part of the NetLogo system architecture modeled in UML

3.1.1.4. Extension-Model

As mentioned earlier NetLogo provides an extension API for integrating Java source code. For the inclusion it is required to compile the source files and compress the resulting class files into a jar archive. NetLogo postulates the archive containing a manifest file which consists of the parameters as listed in Table 5.

Figure 13 shows the architecture of the NetLogo extension API. NetLogo comes along with some built-in primitives. That are specific functions aggregating some Java code, e.g. creating and destroying agents. They can be distinguished in two categories: reporters and commands. Commands define a specific sequence of instructions that need to be executed. Reporters, additionally, return a value after processing. This structure is displayed in Figure 13 by looking at the arrangement of the interfaces: the `Reporter` and `Command` extends the `Primitive` interface. They are implemented in the identical abstract default

Parameter name	Description
Manifest-Version	Represents the manifest version.
Class-Path	Defines the classpath containing the compiled class files.
Extension-Name	Defines a character name identifying the extension package.
Class-Manager	Refers to the manager class which is responsible for providing the extension in NetLogo.
NetLogo-Version	Defines the version of NetLogo in which the extension is used.

Table 5: Manifest parameters for NetLogo extensions

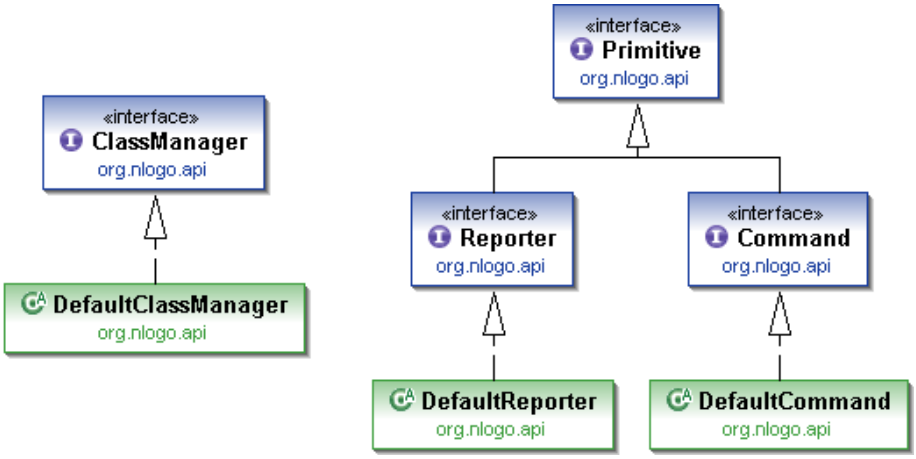


Figure 13: Architecture of the NetLogo extension API in UML

class, respectively. If one wants to create a new primitive for NetLogo it is necessary to sublcass either the `DefaultReporter` class or the `DefaultCommand` class and implement the method `report` or `perform`.

```
1  public class SampleClassManager extends DefaultClassManager {
2
3    /* Declaration of variables and methods */
4
5    public void load(PrimitiveManager pPrimitiveManager) {
6      pPrimitiveManager.addPrimitive("first-primitive",
7                                     new FirstInstructionSet());
8      pPrimitiveManager.addPrimitive("second-primitive",
9                                     new SecondInstructionSet());
10   }
11 }
```

Listing 1: Source code of the `SampleClassManager`

After finishing the implementation of the new derived class one has to tell NetLogo how to associate a new primitive with its implementation. The `DefaultClassManager` im-

plementing the `ClassManager` interface is responsible for assuring the correct code execution by invoking a primitive. The interconnection is performed like in Listing 1.

3.1.2. Eclipse

3.1.2.1. Definition

"Eclipse is an open-source software framework written primarily in Java" [58]. Eclipse is mostly used as IDE for software development in Java. However, it supports several other languages as C and C++. Eclipse is a cross-platform software environment released under the Eclipse Public License developed by the Eclipse Foundation. Its latest release is the version 3.3.1.1 (October 23, 2007). Eclipse provides a mechanism for installing a plethora of plug-ins (see section 3.1.2.2). It comes along with features like in-time compilation, code completion and supports template usage. However, the version used for the development of NetLogo extensions was 3.2.

3.1.2.2. Plug-Ins

There are plenty of different plug-ins for Eclipse. For the development it was useful to integrate the eUML2 as well as the Checkstyle plug-in.

The eUML2 plug-in is a modeling tool for Eclipse for package, class as well as sequence diagrams. It has been developed by Soyatec, an open solution company, and has been released in 2006. The eUML2 tool "is built on top of the UML2 framework of Eclipse as the UML metamodel, which is in fact the best open source implementation of the latest UML2.1 specification" [47]. The version supports the OMG XMI[14] storage format, which allows the model exchange with other UML metamodels. The eUML2 studio edition is comprised of the following four parts:

eUML2 Modeler: Supports the modeling of class and sequence diagrams in order to facilitate developers to design their code.

eDepend: This tool is an advanced dependency viewer. Developers get a quick overview of dependencies of their code.

eEMF Modeler: Allows the developer to design EMF[15] models in an easy way.

eDatabase: Also known as eclipseDatabase, eDatabase is a general database Eclipse toolset. It supports graphically design of the database, i.e. creation, modification and access.

The eUML2 designer is a powerful tool for generating UML diagrams on a high level. The diagrams can be exported in different graphic formats. The plug-in supports vector-based as well as raster-based graphic formats:

- Vector graphics

[14] XML Metadata Interchange
[15] Eclipse Modeling Framework

  - Scalable Vector Graphics (SVG)
  - Windows Metafile (WMF)
- Raster graphics
  - Portable Network Graphics (PNG)
  - Joint Photographic Experts Group (JPEG)

The diagrams in this report have been generated and exported using the eUML2 plug-in installed in Eclipse.

The second plug-in, Checkstyle, is a tool that keeps developers aware to adhere specific code standards. The plug-in has been written by Oliver Burn. Checkstyle is very effective by integrating it into the build process of the source code. It comes along with a command line tool and an ant task. As described in section 3.1.2.3 Ant is used for performing the build process and thus Checkstyle is invoked by calling an ant target. Invoking Checkstyle it inspects the written source code and points out items that deviate from a defined set of coding rules [45]. It starts examining the code by executing the `check` target.

3.1.2.3. Build System with Ant

The build process for the source code is performed with the Apache Ant software [2]. Ant is an in Java developed tool for generating applications based on source files. The plug-in is an open source software project and has been released under the Apache Software License. It is comparable to the well known program 'make' that also builds applications based on source files. The first version of Ant was developed by James Duncan Davidson in 1999 and was part of the Jakarta-Project. Ant is an acronym and stands for "Another Neat Tool" [59].

The main difference between Ant and make is that Ant uses an XML file to describe the build process whereas make uses its proprietary makefile format. Commonly, the input file for Ant is named `build.xml`. This file contains all the instructions for Ant of how to build the application or a part of it. Ant is also very suitable to integrate JUnit tests into the build process and thus enabling test-driven development.

However, as mentioned in section 3.1.1.4 (page 39) and presented in detail in section 3.3.2.1 (page 56) the SWARMLINDA simulator uses the extension model for integrating Java code. Ant is responsible in order to compile and build the source files and plug it into NetLogo system. As shown in Listing 2 it is sufficient to run the `createJar` target for executing the whole build process.

The `createJar` target depends on the `incrementVersionNumber` target. Therefore it has to be executed first. The build process includes a versioning system that assigns each generated output a unique version combined with a timestamp. After integrating the generated output into NetLogo it is able to show the version and the timestamp of the build process. The class `FileManager` is responsible for the file management. It administrates the already built versions and knows the prospective one for the ongoing project. Afterwards the `FileManager` writes the version information inside the current project.

```
<project name="NetLogoExtensions" basedir=".">
  <!-- some global properties & targets -->

  <target name="createJar" depends="incrementVersionNumber"
                           description="Creates a jarfile for NetLogo">
    <java fork="true" classname="tools.JarBuilder">
      <classpath path="\${build}"/>
    </java>
  </target>

  <!-- some global properties & targets -->
</project>
```

Listing 2: Ant target invoking the build process

Finally, the `createJar` target is invoked by Ant and instantiates the `JarBuilder`. This is a helper class for creating a jar[16] file. Conform to the J2EE[17] structuring conventions all source files contained in the `src` package are compiled to the `build` package without loosing the internal package structuring. Once, the compilation is done the `JarBuilder` helps to create the jar file. In order to compress the required files into the archive the `JarBuilder` executes a batch file by starting the command line using Java's `Runtime` class.

The batch file assures the integration of the required files contained in the packages: `etc`, `tools`, `primitives`, `interfaces`, `logging` and `lib`. It also includes the manifest which is required by NetLogo for loading the extension. After the jar file is created the `JarBuilder` is responsible for moving it to NetLogo's extension directory. Finally, NetLogo can be started and scans its extension directory. By executing NetLogo code it starts loading extension files dynamically.

Figure 14 shows the architecture of the `tools` package which is used by Ant. However, this package is also included in the jar file since there is a mechanism involved in a primitive that also needs file management 3.3.2.1 (page 56).

3.2. Network Generation Simulator

This section deals with the generation of scale-free networks based on the approach of Barabasi [3]. The generated networks can be exported and imported in the SWARMLINDA simulator. During the generation one has to cope with two issues: first, the arrangement of connectivity between the nodes and second the spatial arrangement of nodes. There shall be only a few nodes exhibiting a high degree of connectivity while the majority remains less connected. Spatial arrangement postulates that the nodes are spread among the 2D terrain that they cover most of the space. They shall be equally distributed.

[16] Java Archive
[17] Java 2 Enterprise Edition

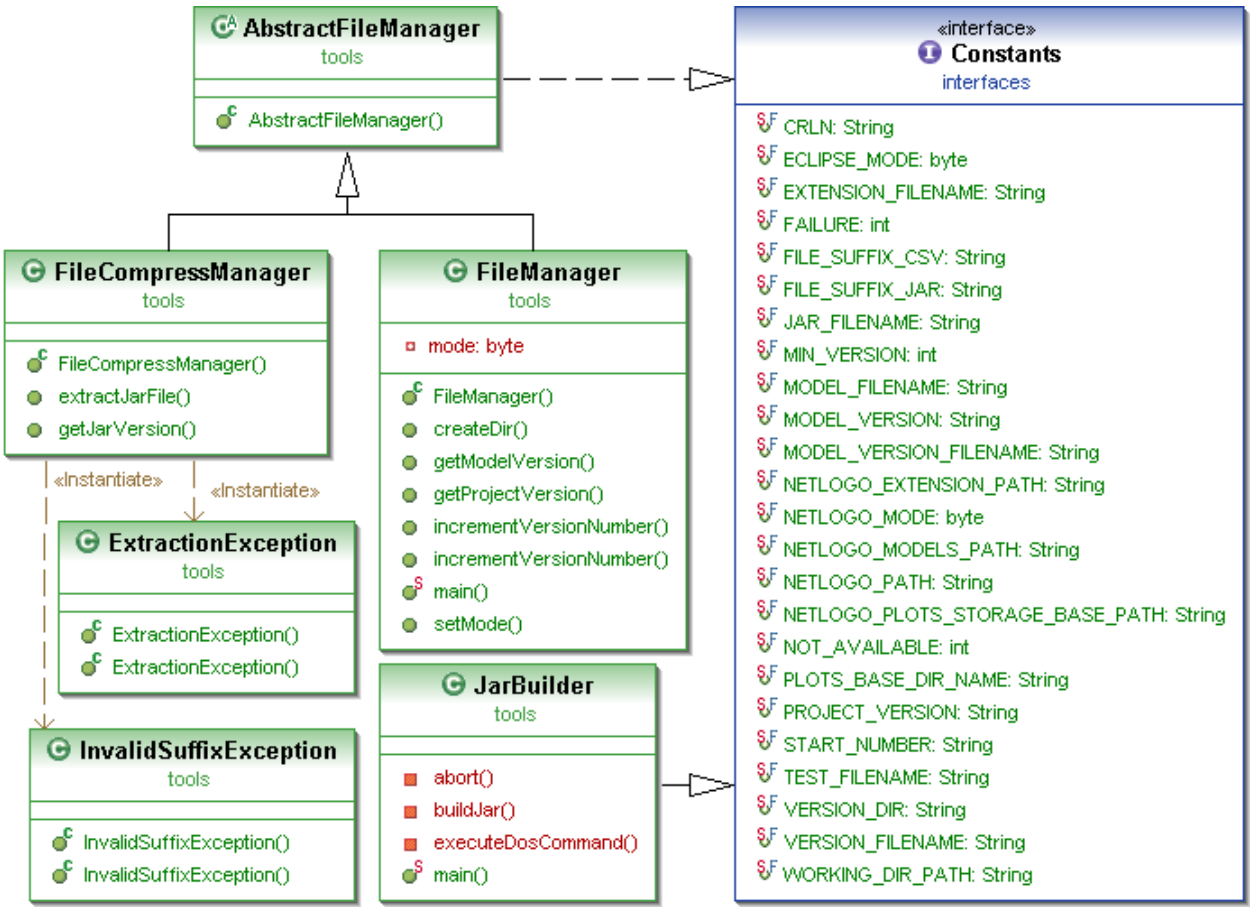


Figure 14: Architecture of the `tools` package

3.2.1. Generation of Networks

Figure 15 shows a network containing 178 nodes that has been generated using the simulator. On the top left one can see the five basic control elements for generating the networks. First, the *setup* button has to be pressed in order to initiate some system parameters and also reset the plots. Further it kills all kinds of breeds, that are the acting turtles, in the simulation area. Afterwards, it sets up two nodes and connects them with an edge. After the nodes got connected the position of the second node gets modified so that both nodes stay in proximity to each other. This is done by randomly choosing an angle between 0 and 360 degrees and then let the second node move 8 steps off the first node's position in the direction of the chosen angle.

In order to put more nodes in the network one can press the *go* or *go-once* button. The *go-once* control element forces the simulator to create one additional node, connect it to two different nodes and finally arranges the new node so that it stays in proximity to both nodes. This mechanism forces the amount of nodes to "explore" the terrain. Increasing the number of nodes they tend to stay in proximity but also try to keep an equal distance to

each other. Therefore the resulting network is concentrated but grows in all directions. However, while *go-once* adds exactly one node and two links to the environment, the *go* control element is a forever-button. That means that it continues adding nodes and edges. It is realized by a `while(true){..}` control sequence. In order to force NetLogo to stop the sequence it is necessary to press the *go* button again.

Looking at Figure 15 one may notice that the generated network is comprised of a few well connected nodes while the majority remains less connected. The shape of the nodes determines the connectivity. The larger the diameter of the circles the more it is linked to others. The topology follows a scale-free power-law distribution [3]. That means that complex networks exhibit the behavior that already well connected nodes are more attractive for new connectivities. The higher the degree of links of a node the higher is the probability of getting connected to a new node that has been added to the environment. Thus the probability $P(n)$ of connecting a new node to one which is already contained in the network follows $P(n) \sim D(n)$, with $D(n)$ determining the degree of node n.

By activating the *redo layout* button one can try to stretch the nodes. Sometimes it is even possible to rearrange a nodes position when it was not well placed in the previous steps. By pressing *resize nodes* one can choose between the *connectivity* and *normal mode*. In *normal mode* all nodes own the same diameter while *connectivity mode* indicates different sizes as shown in Figure 15.

The *network-diffusion* slider controls the diffusion of nodes. Low values result in best performance since the addition of nodes is very fast. High values indicates low performance since the arrangement of nodes is computational intensive. It is advisable not to turn the value under a certain threshold ($\sim$30) since the ratio of computation time and distribution tends to get insufficient. The two switches *show-node-information* and *show-ant-information* enables or disables labels of turtles in the view showing their identification number and some additional information, e.g. nodes appends the degree.

With *export world* it is possible to export the whole world that is the environment composed of all variables in the system. They are stored as csv[18] file. On the other hand *import world* imports a world from a csv file. Additionally, *export plots* supports the export of the drawn graphs. The format is csv, too. Afterwards they can be used as input for an external visualization tool like Excel.

3.2.2. Interaction with Networks

The network generation simulator can be used in two modes: automatic or manual generation. Also a hybrid approach is possible. In order to generate the network fully manually it is recommended to activate the *clear all* button. This removes all entities in the environment. Afterwards one can determine the location of a new node by adjusting the two sliders for the *x-* and *y-coordinate*. By pressing the *create node* button a new node is placed at the given location. Meanwhile the ID of the new node is written to the output area. The visualization area is constructed as a symmetric coordination system ranging from $[-\varphi, \varphi]$ in y-axis and $[-\gamma, \gamma]$ in x-axis.

[18]Comma Separated Values

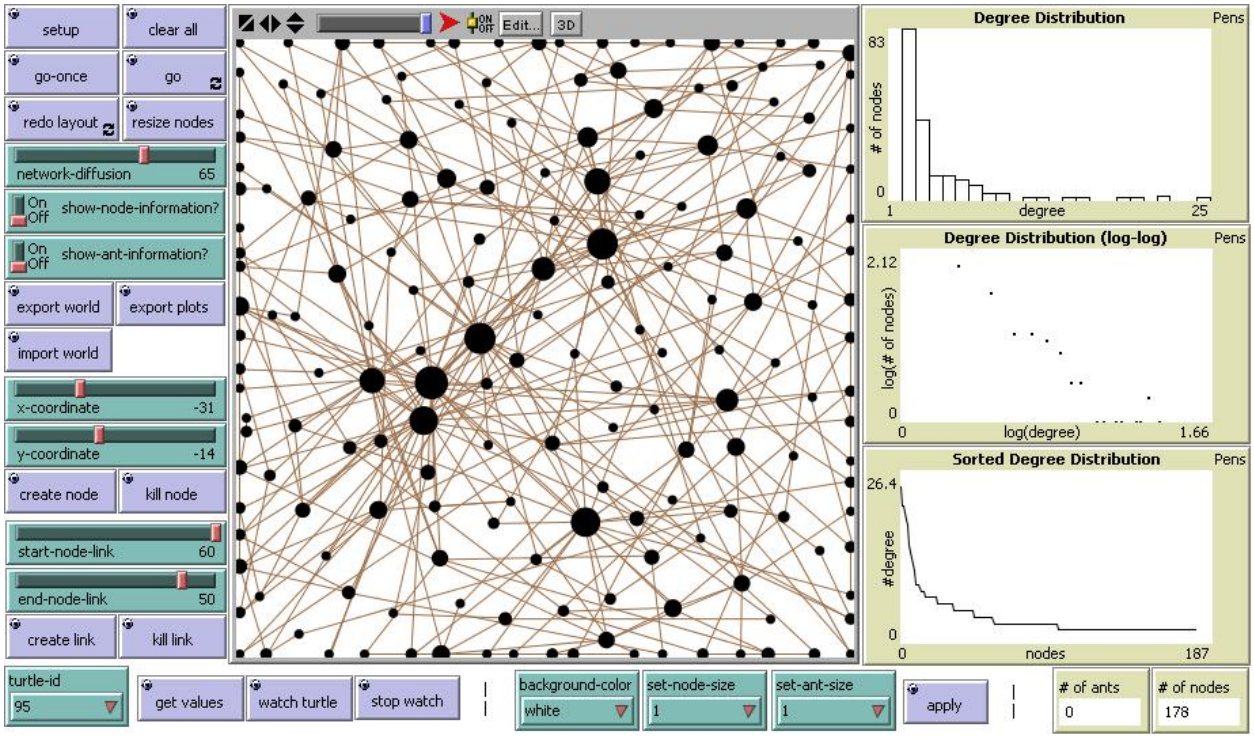


Figure 15: Network Generation Simulator

In order to create a link one has to select a start as well as an end node by adjusting the respective sliders. Pressing the *create link* control element the link will be drawn in the visualization area and its ID is printed to the output area. For removing nodes as well as links one has to pick a turtle ID from the chooser. Pressing the respective *kill node* or *kill link* button the turtle dies and disappears from the visualization area. However if one selects an ID from the chooser and activates the *get values* control element the sliders above are set to the values of the respective turtle values.

Looking at the visualization area of Figure 15 it is obvious that it may be hard to find a specific turtle within an amount of 178 nodes and 353 links. By pressing the *watch turtle* control element the turtle which owns the selected turtle ID gets highlighted. This facilitates finding a turtle. Pressing the *stop watch* button the highlighted turtle gets released. It is also possible to combine both approaches. The network can be generated hybrid by adding nodes and links automatically and manually.

With the choosers *background-color*, *set-node-size* and *set-ant-size* one can modify these properties. Finally, the *apply* button has to be pressed in order to tell NetLogo to adapt to the changes. The two monitors on the right bottom exhibit the amount of ants and nodes in the environment.

3.2.3. Plotted Graphs

The right side of the simulator shows three graphs arranged vertically. All of them have in common that they show a specific view of the degree of nodes in the network. The purpose is to observe and evaluate the approach of Barabasi [3] that networks contain less hubs but a plethora of nodes that are less connected. There are other studies which also emphasize this phenomenon of complex networks like in biology where a large network is formed by the nervous system. The vertices represent the nerve cells which are connected by axons [26]. On a social level vertices can be seen as human beings while the links represent the social interactions between them [55]. In computer science the most arguably representation for complex networks is the world wide web. Vertices can be interpret as HTML documents pointing to other documents and on the other hand are referenced by documents [38].

In the visualization area of Figure 15 (page 46) one can interpret the well connected nodes (big vertices) as backbone of the network. In any case, if they break down the nodes tend to get less connected. The top right plot shows the dependency between the degree of a node and the amount of nodes owning this degree. It is obvious that there are only a few vertices holding a high degree of connectivity. They form the backbone. In contrasts there are plenty of nodes owning a very small degree. They exhibit only two or three links. Finally, there are the "intermediate" nodes holding around four to 20 links. The curvature follows an exponential decrease.

The second plot shows a logarithmic scaled graph while the third plot exhibits the dependency between a specific node and its degree. The x-axis indicates sorted node IDs. Each time a vertex is added to the environment it gets an ID assigned. The IDs starting from 0 are chosen in ascending order. Therefore the plot shows a time dependent graph. Following the curvature from left to right the nodes are added more recently. In fact, the probability that an older vertex owns more edges than a newer is higher. Basically, the probability has to be higher since former nodes obtain more chances to receive a new connection than recently added ones. Conform to the approach of Barabasi the first created vertices are designated to obtain the most links since they already own the highest degree from the beginning.

3.3. SwarmLinda Simulator

This section deals with the development and implementation of the SwarmLinda simulator. The implementation is comprised of an extension written in Java and the actual NetLogo model.

3.3.1. Visualization

3.3.1.1. Controlling the Simulator

On the left side of Figure 16 one can see the basic operations. That is:

Import world imports an existing network generated with the network simulator from

section 3.2 (page 43). The file format must be a csv file. The import includes initiation of the required parameters contained in the model.

Setup clears the whole environment. It requires importing a network; if this is not done, yet, the import cannot be started. This is assured due to an internal state mechanism that guarantees the correct order of executions. Further it initiates several parameters (e.g. pheromones-lists, tuple spaces).

Import snapshot behaves similar to *import world* except that the state mechanism works different. While *import world* forces the user to start with an initial empty network *import snapshot* allows loading a specific state of a simulation. Thus it is possible to compare different test runs based on the same snapshot of the system. Additionally, *import snapshot* sets up the plots and draws some curves representing the current system state.

List primitives prints all added primitives to the NetLogo system formatted in the output area. The primitives itself and the related description is loaded dynamically from the Java classes via the Java reflection mechanism as described in 3.3.2.1 (page 56).

Export plots exports all drawn plots to a csv file specified by the user. The data contained in the output file can be used for external tools for further processing.

Apply layout applies the layout changes due to modifications of the graphical properties. The control elements for the configuration are located on the opposite site of the visualization area.

Save state saves the current state of the model environment. This includes all parameters contained in the model. The data is automatically saved in a temporary csv file.

Restore state contrariwise analogous, restores the saved state by loading the temporary file and overwrite the existing parameter values.

Below the basic operation buttons one can find some configuration control elements. The settings are exclusively for the primitives executions. The primitives trigger the system to perform some tasks. However, the configuration of the primitives only effects the start sequence but not the system behavior. In detail, the system behavior remains, in fact, the same but different configurations may lead to different system states after executing the primitive. The configuration parameters are:

Num-ants sets the amount of ants that shall be created during one of the primitive executions. They get born inside the network on a node that is specified with *chosen-node*.

Set-age sets the age or to be more specific the *ttl* value for the ants that shall be instantiated. As one may see in Figure 16 the *ttl* value is set to 20. This is a suitable value for this specific network since the size is relatively small. Therefore it is recommended

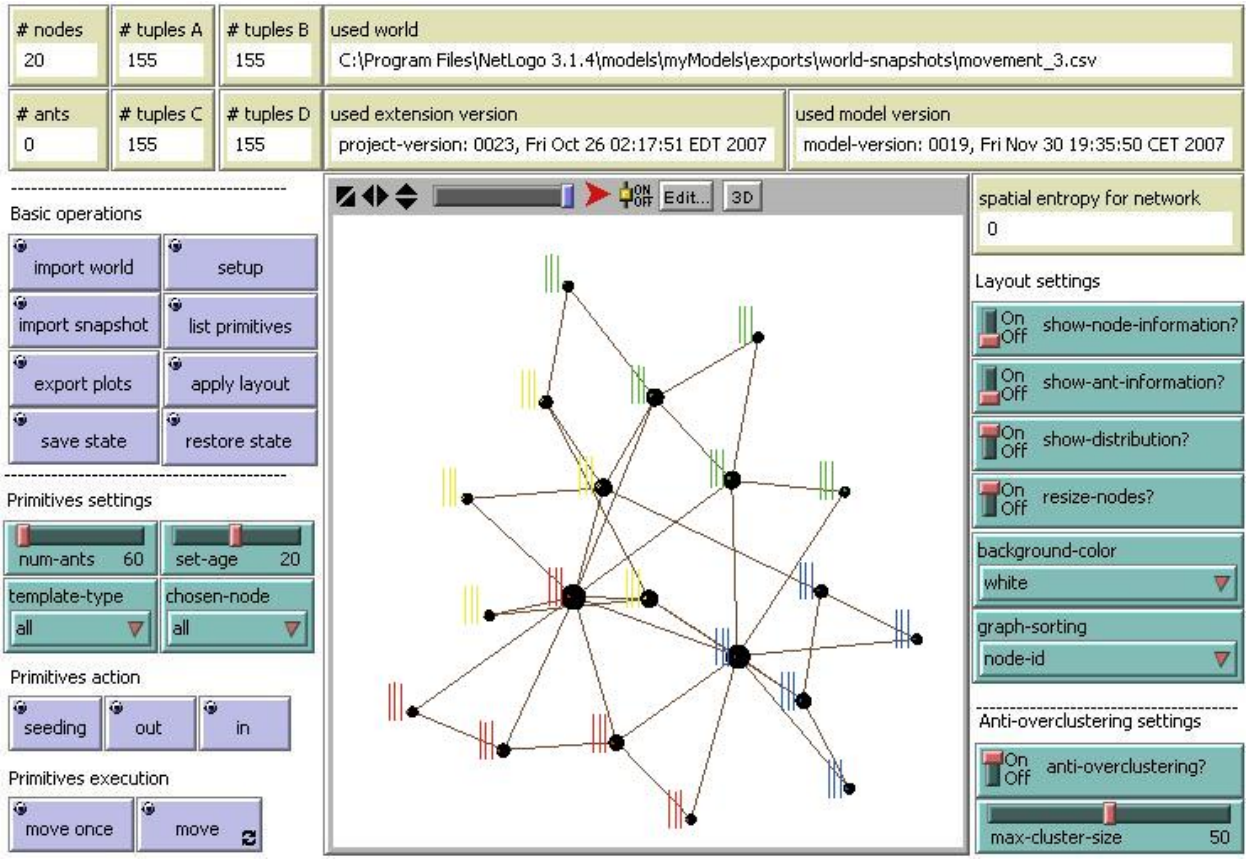


Figure 16: Control elements and visualization area of the SWARMLINDA Simulator

to adapt the *ttl* value dependent on the network size. That gives the ants the possibility to explore the whole network. Limiting the value to a very small one the ants tend to stay in proximity to their birth location. Adjusting the *ttl* value very small in a big network it is very likely that the network gets partitioned into several regions whereas each region contains its local clusters.

Template-type sets the template which is involved in the next primitive execution. This version of the simulator supports four different template types:

- $a(X) = X(v_1, v_2)$ with $v_1 \in String \wedge v_2 \in String$
- $b(X) = X(v_1, v_2)$ with $v_1 \in String \wedge v_2 \in Integer$
- $c(X) = X(v_1, v_2)$ with $v_1 \in Integer \wedge v_2 \in String$
- $d(X) = X(v_1, v_2)$ with $v_1 \in Integer \wedge v_2 \in Integer$

Each template has a distinctive scent and thus can be tracked by ants. However, the simulator can be extended by adding new template types without severe effort. To avoid confusions the templates are neither symmetric nor commutative. As one may see in Figure 16 the current template selection shows “all”. That means that

all from the simulator supported template types are involved in the next primitive execution.

Chosen-node specifies the node at which ants shall be born while triggering a primitive. The current selections is set to "all" which means that the ants get instantiated randomly in the network. However, adjusting this parameter to specific nodes one can force to form clusters in a specific region or observe how ants behave when placing them at the farest node from their cluster. Therefore one can evaluate the path selection done by ants in order to examine whether they take the shortest path or taking a detour.

Below the primitive settings one can see the primitive actions. That are the different commands which trigger the algorithms explained in section 2 (page 18) except the seeding.

Out places the amount of ants specified with *num-ants* on the node determined with *chosen-node*. If *chosen-node* is set to "all" each ant is put by random on a node. Thus the set of ants are separated among the nodes. Each ant carries the tuple which has been specified with '*template-type*. In fact, the tuple parameters are filled with concrete values. If *template-type* is set to "all" then each ant gets a tuple assigned matching one of the available templates. The selection of a template is deterministic. The implementation tries to assign the templates equally to the amount of ants. If $(\textit{num-ants} \mod 4) = 0$ then the size of the formed groups of different ants are equal. Otherwise some groups contain one individual less.

Seeding Seeding does the same as *out* except that the ants get killed after they have been placed. The idea of seeding is to augment the environment with some tuples and therefore with pheromones. This mechanism supports prospective ants to potentially find pheromones and also help emerging of clusters. It is not necessary to perform seeding for the simulation. One can start immediately with *out* primitives. However, seeding can also be applied later. This function enables the observation of how much ants get attracted if seeds have been placed belatedly somewhere in the environment.

In behaves similar to *out*. Ants get placed in the network, too, but instead of assigning them a tuple they carry the chosen template. Additionally, in contrast to *out*-, *in*-ants obtain a memory which they use in order to keep the trail they took in mind.

Finally, after performing the *out*- or *in*-primitive the actual algorithm has to be triggered by using the primitives execution control elements.

Move once forces all ants in the environment to successively perform the respective algorithms for tuple distribution or retrieval in dependency on the previous chosen action. *Move once* indicates that the algorithm is performed only once. By iteratively pressing the *move once* button one can observe each step the ants took.

Move aggregates the process of pressing *move once* until the ants finish their task. The forever-button stops immediately if the last ant is done.

On the right side of the simulator in Figure 16 (page 49) one can see control elements for setting the layout as well as anti-overclustering parameters. Anti-overclustering is a mechanism used in SWARMLINDA for assuring a more balanced distribution of tuples among the nodes. Imagine a scenario where, although tuples get grouped in specific regions and form clusters, the size of the tuple spaces may vary extremely. Anti-overclustering avoids high concentrations on particular nodes by introducing a threshold value. There are two methods for determining the maximum limit:

- The threshold is static and hard encoded for a specific simulation
- The threshold is dynamic and depends on parameters of the environment

The anti-overclustering strategy is described in detail in section 4.4 (page 75). However, the control elements work as follows:

Anti-overclustering switches the strategy on or off. If the mechanism is turned off the SWARMLINDA algorithms are performed as describes in section 2 (page 18). Mostly, this results in clusters exhibiting one big cluster while the remaining ones stay almost empty. Turning the strategy on effects the *out*-primitive. The probability of dropping a tuple gets influenced by the anti-overclustering value in dependence on the concentration.

Max-cluster-size regulates the maximum amount of tuples that can be stored at a tuple space. This is a static mechanism that is suitable for specific test cases.

In the middle of the right side one can see the control elements configuring the layout. All made changes are applied to the system by pressing the *apply layout* button on the left. However, the parameters influence the layout as follows:

Show-node-information enables the visualization of node specific information like its ID and degree. The information is printed close to the respective node.

Show-ant-information enables the visualization of ant specific information like its ID. The information is printed in proximity to the ants current location.

Show-distribution enables the visualization of the majority of tuples matching the same template for each node. The color indicates the template type. In order to maintain a clear view only one type is shown at each tuple space. It also would be no outstanding advantage to indicate all types since a characteristic of the system is to stay apart from dissimilar tuple spaces. The amount of tuples is represented analogous to the network generation simulator: the vertical bars (|) indicate ten, the dots (.) one tuple, respectively.

Resize-nodes enables the visualization of stretching or compressing the diameter of the vertices. The size indicates the connectivity. The higher the degree of a node the bigger is its diameter.

Background-color sets the color of the terrain and also adapts the front color, i.e. nodes and links.

Graph-sorting allows the drawings of the graphs shown in Figure 17 (page 53) in two modes. Currently the plots are sorted in ascending order based on the node ID. This mode enables a good comparison between the four graphs for tuple distribution. One can observe the emergence of clusters while the template types separate their positions apart from the others. However, the second mode arranges the graphs in descending order based on the amount of tuples contained in the respective tuple spaces.

3.3.1.2. Output: Monitors and Plots

All monitors are arranged in the top of Figure 16 (page 49). The fields show the following values:

 # nodes shows the total amount of nodes currently in the network.

 # ants shows the total amount of ants currently in the network.

 # tuples A counts exclusively the number of tuples in tuple spaces that match template $a(X)$.

 # tuples B counts exclusively the number of tuples in tuple spaces that match template $b(X)$.

 # tuples C counts exclusively the number of tuples in tuple spaces that match template $c(X)$.

 # tuples D counts exclusively the number of tuples in tuple spaces that match template $d(X)$.

Used world indicates the absolute path of the imported world.

Used extension version shows the current loaded NetLogo extension version.

Used model version shows the current loaded NetLogo model version.

Spatial entropy for network exhibits the current level of order in the network. The entropy is defined in [0,1], whereas 0 indicates absolute order while 1 stands for total chaos.

Figure 17 (page 53) shows the observation area containing plots and some monitors for following the system behavior. The purpose for the elements is listed below:

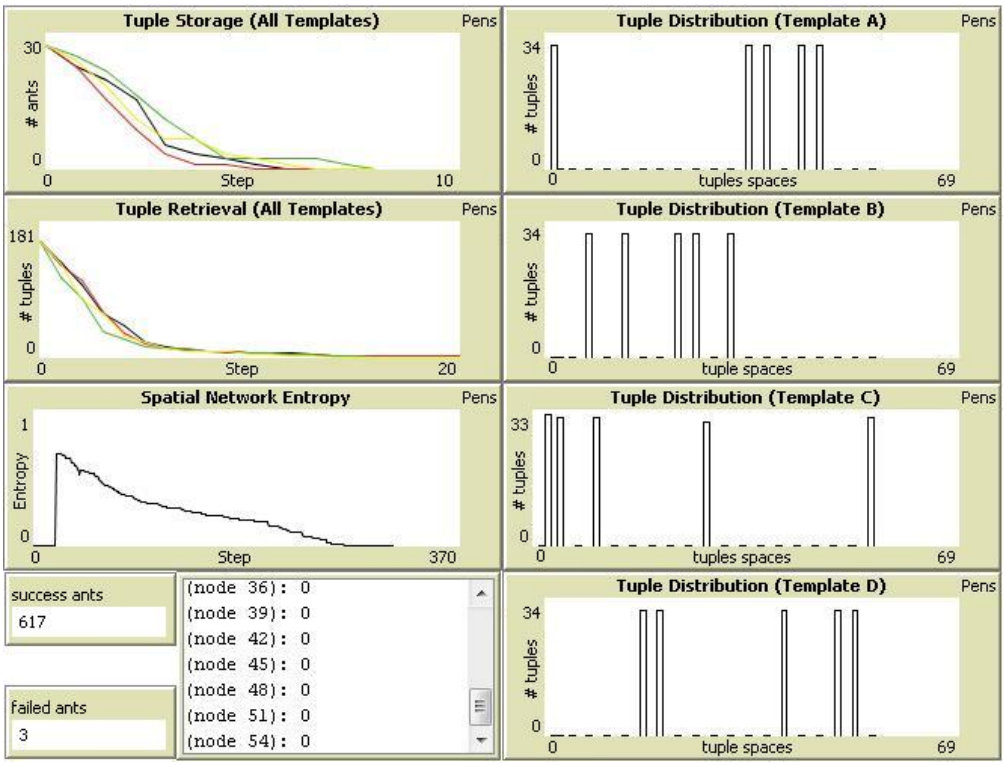


Figure 17: Plotting area of the SWARMLINDA Simulator

Tuple Storage shows the amount of steps it took for storing all tuples that were carried by ants in the network. This plot is generated due to an execution of the *out*-primitive. In the beginning ($step = 0$) the graph exhibits around 30 ants that have been instantiated in the environment. Each step comprises one iteration of the *out*-algorithm. The more steep the slope of the curve is the faster is the storage of tuples since ants disappear after accomplishing their tasks. In fact, it is desired that the curvature follows an exponential decrease. The plot shows four curves in different colors representing the template types.

Tuple Retrieval shows the amount of steps it took for retrieving a specific number of tuples from the environment. In contrast to *tuple storage* this graph is generated by an execution of the *in*-primitive. In the beginning ($step = 0$) the graph shows around 180 template-ants of each kind. Each step indicates an iteration of the *in*-algorithm. Analogous to *tuple storage* it is desired to achieve an exponential decrease of the curve. Likewise, the four colors indicate the different template types.

Spatial Network Entropy is an indicator for the level of order in the network. The plot shows the development of the entropy. It is desirable to keep the entropic value always as low as possible. Looking at the curvature one can see different phases of the system. The around 350 steps involve setting up the environment and activities

of first ants (very high entropy). The system is unorganized. Afterwards, the curve secedes due to a self-organization of tuples. They get separated in the environment and form cluster. The last phase shows tuple movement in order to achieve an improved distribution of tuples among the nodes.

Tuple Distribution (Template A) shows the amount of tuples at the respective nodes of template type $a(X)$. The value on the horizontal axis indicates the node ID.

Tuple Distribution (Template B) shows the amount of tuples at the respective nodes of template type $b(X)$. The value on the horizontal axis indicates the node ID.

Tuple Distribution (Template C) shows the amount of tuples at the respective nodes of template type $c(X)$. The value on the horizontal axis indicates the node ID.

Tuple Distribution (Template D) shows the amount of tuples at the respective nodes of template type $d(X)$. The value on the horizontal axis indicates the node ID.

Success ants monitors ants that successfully accomplished their tasks. This allows the observation of the system performance and is useful for the test runs.

Failed ants monitors ants that failed finishing their tasks. This allows the observation of the system performance and is useful for the test runs.

Output area is a second general output area and is used for printing the node entropy values.

3.3.1.3. Additional Test Environment

Figure 18 shows the additional test environment which is comprised of the simulation of node failures and tuple movement. Node failures can be forced intentionally either by using the manual mode or by applying the automatic execution. Either way it makes sense to save the current state of the environment before performing the tests. Afterwards, one can execute a test without failure and save the results of the test run. After restoring the state one can perform some node failures and then run the same tests again. Both results can be compared in order to see the impact of node disappearances. However, pressing the *kill node* button selects randomly one of the nodes for shut down. The vertex itself and all connecting edges get removed. The node ID is listed below in the *killed nodes* monitor. It is possible to kill as much nodes as placed in the network.

Below the manual execution mode one can find the automatic configuration environment. The workflow is described in detail in section 3.3.2.3 (page 57). The test environment comprises the following elements:

Num-tests sets the number of tests that shall be performed sequently. Before each test run the environment is saved and gets restored after the test run. This assures always working on the same world.

Num-nodes-to-kill sets the number of nodes that shall be shut down for the test run.

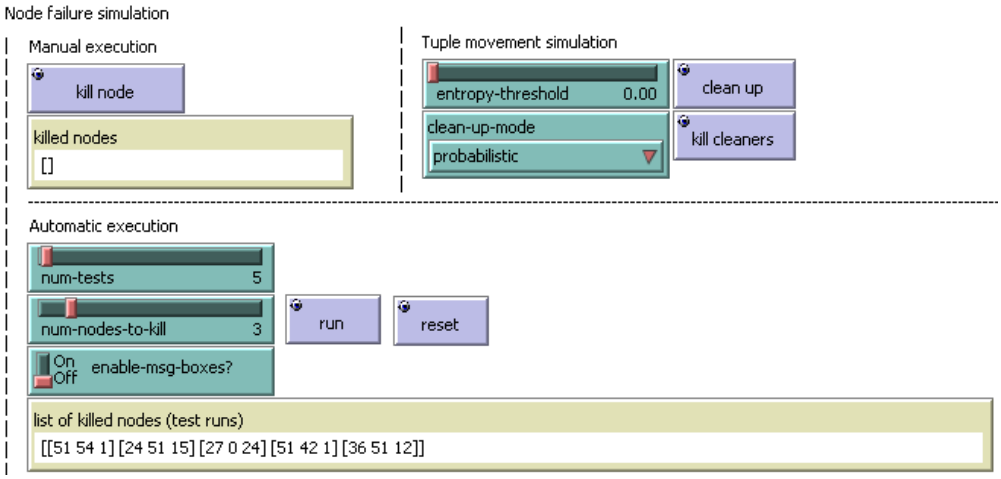


Figure 18: Node failure simulation and tuple movement configuration section of the SWARMLINDA Simulator

Enable-msg-boxes enables resp. disables message boxes during the test runs. In order to perform a fast run it is recommended to disable the message boxes.

List-of-killed-nodes shows a list of sublists containing all nodes that have been shut down during the respective test runs. Each sublist is semantically a set indicating that each element is unique in it. In fact, it makes no sense to shut down the same server twice in one test run. However, the list containing the sublists is also a set. That postulates that each element has to be unique, too. Thus permutations of the elements in the sublists are forbidden. Only real combinations are allowed.

Run starts the test runs by applying the chosen configuration. The progress of each test run is printed to the primary output area.

Reset resets the monitor *list-of-killed-nodes* in order to allow new test runs.

Beside the test environment one can see the configuration for tuple movement. The control elements are defined as follows:

Entropy-threshold sets the maximal entropy threshold. This only takes effect if the *clean-up-mode* is set to deterministic. In this case the cleaning ants are trying to achieve this entropy level. In detail, they try to push the entropy below this level. Once they accomplish their objective they stop immediately but remain in the network.

Clean-up-mode sets the mode for cleaning up the environment. There is a deterministic as well as a probabilistic approach.

Clean up instantiates as much cleanup-ants in the environment as specified with *num-ants*. By pushing the *move once* or *move* button the tuple movement algorithm which is described in section 2 (page 18) is performed.

Kill cleaners removes all cleaning ants from the environment.

3.3.2. Implementation

3.3.2.1. Extension

This section deals with the implementation of the NetLogo extension written in Java and used in the simulator. During the development it was necessary to add some primitives in order to facilitate the programming part by aggregating a set of instructions. This also produces smaller and more structured NetLogo source code. On the other hand some functions are simply not available in the NetLogo primitive dictionary. Java as a powerful programming language provides solutions for all requirements. The added primitives are listed below:

Primitive	Description	Implemented class
Real-random-int	Generates a random integer value between 0 and the input parameter	`RandomInt`
Real-random-long	Generates a random long value between 0 and the input parameter	`RandomLong`
Real-random-float	Generates a random float value between 0 and the input parameter	`RandomFloat`
Real-random-double	Generates a random double value between 0 and the input parameter	`RandomDouble`
Get-extension-version	Returns the currently loaded NetLogo extension version	`Version`
Get-model-version	Returns the currently used NetLogo model version	`Model-Version`
Get-next-filename	Returns a unique filename for automatic plot export	`FilenameGeneration`
Get-primitives	Lists all implemented primitives in the output area	`PrimitiveLister`
Contains-set?	Tests if the given set is contained in a given super set	`ContainsSetValidator`
Create-dir	Creates the given abstract path	`DirCreator`
Min-int	Returns min-int, -2^{31}	`IntegerMinValue`
Max-int	Returns max-int, $2^{31} - 1$	`IntegerMaxValue`
Min-long	Returns min-long, -2^{63}	`LongMinValue`
Max-long	Returns max-long, $2^{63} - 1$	`LongMaxValue`
Min-float	Returns min-float, 2^{-149}	`FloatMinValue`
Max-float	Returns max-float, $(2 - 2^{-23}) \cdot 2^{127}$	`FloatMaxValue`
Min-double	Returns min-double, 2^{-1074}	`DoubleMinValue`
Max-double	Returns max-double, $(2 - 2^{-52}) \cdot 2^{1023}$	`DoubleMaxValue`
First-n-integers	Lists the first $n - 1$ integers	`IntegerList`

Table 6: Added primitives in NetLogo

The NetLogo extension version is directly loaded from the jar file. In contrast the model version which has been used for the simulator development is contained in the model directory. Each time the NetLogo model gets modified the versioning mechanism gets notified and increases the version number. The primitive `get-next-filename` has been implemented in order to obtain a unique file name and storage location for exporting specific plots. The primitive is used in the automatic test environment 3.3.2.3 (page 57). `Get-primitives` lists all developed primitives in NetLogo's output area. As a coding standard if one wants to add a new primitive the `SwarmLindaExtension` class postulates an identifier (primitive name), a textual description and a resource (functionality containing class). All three attributes are mandatory. Invoking `get-primitives` accesses the stored information via Java reflection[19]. A class diagram is appended in section A. Figure 45 (page 119) shows the arrangement as well as inheritance of classes.

3.3.2.2. Logging

Based on inheritance all classes implementing the primitives own a logging instance. In order to enable logging during the runtime of the simulator the Apache log4j [17] utility was used. It is a Java-based tool originally written by Ceki Gülcü and is now a project of the Apache Software Foundation [56]. The tool is platform independent and allows different logging levels. All log messages are written to a specific port (here: 4445) on localhost. The visualization of logging entries is performed by the software tool lumbermill. "Lumbermill is a visual log processing and distribution center for Log4j and (in 2.0) java.util.logging (JSR47). It is a Swing/GUI standalone application that supports viewing and archiving of log events." [46].

3.3.2.3. Test Environemt

The test environment is a suitable part of the simulator for observing characteristics of node failures. Section 3.3.1.3 (page 54) introduces the graphical aspects of the test environment. However, technically it works as follows:

1. The current state of the world is stored in a temporary file.
2. It continues looping over the actual test implementation as much as specified by the amount of tests determined with the *num-tests* slider. According to the configuration of *num-nodes-to-kill* the system selects randomly the adjusted amount of nodes and removes them from the terrain. In detail, it chooses the nodes and puts them into a set. Afterwards it checks whether the given combination or a permutation is already contained in the *list-of-list-of-killed-servers* that comprises all sets of removed nodes that has been shut down in previous test iterations. In the scenario that the set is contained in the list the algorithm has to drop the current selection and looks for a new one. The new added primitive *contains-set?* is responsible for the check. However, in the beginning the *list-of-list-of-killed-servers* is empty. Hence, the new set is accepted anyway. Finally, it adds the set to the list *list-of-list-of-killed-servers*. Meanwhile the system shuts down the selected nodes.

[19] Reflection is a relatively advanced feature [51] that enables access of class information during runtime

3. The current implementation uses the test simulator for observing the behavior of template-ants. Therefore only the *in*-algorithm is applied. The next step performs the *in*-command that is the same as pressing the *in* button for ant instantiation in the network which is followed by the execution of the *move-forever* button. This applies the whole *in*-algorithm.

4. After the execution the world and all plots are stored for further evaluation. For the storage the system uses the relative to the simulation file related folder *export\plots*. Depending on the configuration the system looks for a folder that indicates the kind of test, e.g. setting *num-nodes-to-kill* to three the folder is named "three node failures". If the folder does not exist the system creates it. The actual export files follow the format *test_<serial-number>.csv*. Each of the files can be post-processed with external software tools.

5. In order to guarantee that the next test run takes place under same conditions the environment is restored from the temporary storage file. Already performed test runs are listed in the monitor below the configuration parameters (see Figure 18, page 55).

4. Improved Metrics in SWARMLINDA

This section deals with the improvements as well as new definitions of formulas that are involved in the implemented SWARMLINDA system. Section 2 (page 18) discusses in detail basic formulas as well as algorithms. Section 5 (page 80) evaluates basic algorithms and applied formulas and points out the improvements that are explained in this section. However, the development of improvements of the system has been performed by adjusting formula parameters and exploit results shown by the simulator.

4.1. Drop Probability

SWARMLINDA's way to distribute tuples among the nodes is performed by executing the *out*-primitive. This invokes the algorithm for tuple distribution that has been discussed in section 2.1 (page 18).

Remember, tuple distribution deals with the ability of storing information objects in a domain that is the network. The network comprises an arbitrary amount of nodes connected by links. SWARMLINDA is characterized by a self-organization mechanism that is responsible for arranging its content. The basic principles of such a system postulate the following properties:

Semi-uniform distribution: all system resources shall be used equally. That requires that the amount of information objects are distributed so that each node obtains a certain number of tuples of the original set. Let Γ be the base set of information objects that shall be distributed. Each node shall obtain a set γ that is a real subset of Γ, $\gamma \subset \Gamma$. Ideally, the set γ is defined as follows: $\gamma_{opt} = \frac{\Gamma}{|N|}$, with N containing all nodes in the network. According to a scenario under real conditions it is unlikely that the partitions are of equal size. Thus γ varies around some $\Delta\tau$: $\gamma_{real} = \gamma_{opt} \pm \Delta\tau$.

Homogeneous cluster: the emerging cluster shall be as homogeneous as possible. According to the brood sorting done by biological ants the system shall organize their objects by type. Let χ be a homogeneity indicator of a node n resp. tuple space, with $\chi \in [0, 1]$. 0 indicates heterogeneity while 1 represents homogeneity. It is desirable to measure χ tending to approximate 1.

Heterogeneous environment: the environment that is the network is separated into multiple regions. Each region is comprised of an amount of nodes. While the regions shall also be as homogeneous as possible SWARMLINDA postulates the network to be heterogeneous. Let $\Re$ denote a set of nodes that form a specific region and let $\Re_\chi$ be its homogeneity value defined as

$$\Re_\chi = \frac{1}{|\Re|} \sum_{\forall n \in \Re} \chi_n \tag{8}$$

then the homogeneity value for the network denoted by $\aleph_\chi$ shall be minimized, $\aleph_\chi \to 0$. $\aleph_\chi$ is defined as follows:

$$\aleph_\chi = \frac{1}{|N|} \sum_{\forall \Re_\chi \in \aleph} (1 - \Re_\chi) \, |\Re| \tag{9}$$

The formula treats each member $\Re$ of $\aleph$ based on its cardinal number. Hence, big regions have a strong influence of the homogeneity value of the whole network.

Proximity: SWARMLINDA postulates that similar information stay in proximity. That is the basic for forming regions. One can create a region by grouping nodes that contain similar information with the restriction that all nodes have at least one connecting link to one of the group members. Thus the evolved network graph of the region needs to be coherent.

Load balancing: the aforementioned characteristics result in load balancing. If all resources are used and the amount of tuples is distributed by spatial separation based on the template type, it assures load balancing since ants roam to different locations.

However, the *out*-algorithm consists essentially of three parts: the movement-, the decision- and the aging-phase. They are executed successively. Once a tuple-ant is in the decision-phase it decides probabilistically whether to drop its tuple at the current location or not. According to the approach of Casadei *et al.* defined in [10] Equation 11 computes the probability of dropping a tuple at a node.

$$C_{orig} = \sum_{\forall t_s \in TS} sim(t_c, t_s) \tag{10}$$

$$P_{drop}^{orig} = \left(\frac{C_{orig}}{C_{orig} + K} \right)^2 \tag{11}$$

The concentration C_{orig} is defined as the amount of tuples contained in a tuple space that matches a common template that also matches the tuple the ant carries. Although the formula for the drop probability P_{drop}^{orig} achieves clustering it tends more to gain a bad homogeneity value. This occurs because C_{orig} counts only similar objects in the local tuple space. It does not take into consideration the amount of different tuple types that are stored at the respective node. It would be more meaningful to set the amount of similar tuples in contrast to the differing ones and let this ratio influence the drop probability.

Figure 19 shows the comparison between the original computation of the concentration C_{orig} and the, in order to improve the system performance, called modified concentration C_{mod}. However, Figure 19(a), 19(b) and 19(c) exhibit different concentration levels based on different system environments. The environment is defined by the amount of tuples and their template type in a specific node. All plots have been generated using the same tuple space TS but at different time intervals. Each graph exhibits the concentration as

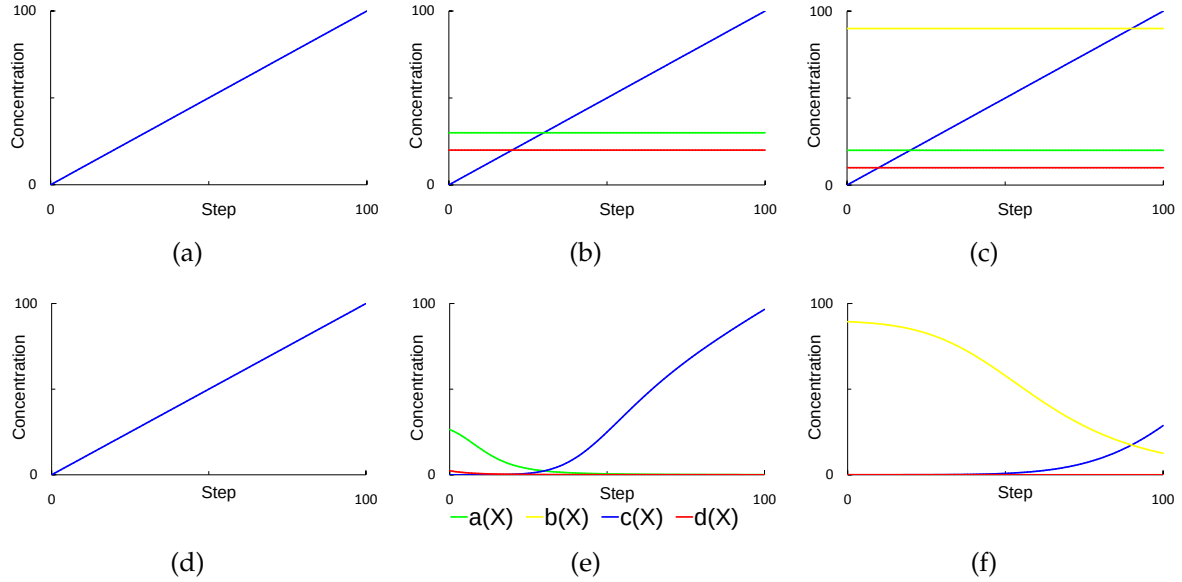


Figure 19: Comparison between C_{orig} represented by the upper three graphs and C_{mod} represented by the lower three graphs. It is shown the dependency between homogeneous and heterogeneous cluster structures. The graphs are arranged in three columns and are based on the scenarios shown in Table 7.

a function of time. The horizontal axis is labeled with *Step* that is a unit of time introduced in the SWARMLINDA simulator. In order to avoid confusions *Step* is a quasi-time dimension. It is more related to a time interval in which one iteration of the mentioned algorithms from section 2 (page 18) is performed. Hence, it is very likely that each ongoing step modifies the state of the environment. In fact, this simply appears by letting ants roam in the environment and dropping pheromones. One may notice that each graph starts with $Step = 0$. This suggests that *Step* is not a global time. Indeed, there is another variable holding the global time but for the test scenarios it is more suitable to define a local time introduced by *Step*.

	$\|A\|$	$\|B\|$	$\|C\|$	$\|D\|$	K
Scenario_1	–	–	Var	–	10
Scenario_2	30	–	Var	20	10
Scenario_3	20	90	Var	10	10

Table 7: Configuration of the system environment for the test runs

The shown graphs are based on the same principle: they show the concentration of the respective tuple types at a specific node. Again, the color indicates the template. Further on, in all examples the amount of tuples matching template $a(X)$, $b(X)$ and $d(X)$ respectively are static. While time passes by there is no removal or addition of those tuples. Let A be the set of tuples matching template $a(X)$; B, C and D respectively for $b(X)$, $c(X)$ and $d(X)$, with $A, B, C, D \subset T$. Their size is restricted within the following range: $|A|$, $|B|$,

$|C|, |D| \in [0, 100]$. Thereby T is the set of all tuples at TS and is defined as the set union: $T = A \cup B \cup C \cup D$. Each set is a disjoint subset of T.

However, $|C|$ is increasing at each step by one. In the beginning C is empty. Setting up the tests $|A|$, $|B|$ and $|D|$ obtain different values but they do not change while a test is running. In particular, the tests show the development of the concentration under certain circumstances.

Figure 19(a) shows a scenario with initially no tuples in TS at time $Step = 0$. The test run exhibits the progress of the concentration while tuples are added to TS. The tuple space obtains a maximum homogeneity value ($\chi = 1$) since there are no other tuple types around. One can observe a linear increase of the concentration value. The scenario in Figure 19(b) follows also the rule that tuples matching $c(X)$ are added while holding A and D constant during the run. The test is set up by augmenting TS with $a(X)$ and $d(X)$ tuples so that $|A| = 30$ and $|D| = 20$. C_{orig} is constant for $a(X)$ and $d(X)$ tuples thus other tuple types do not influence the concentration. On the other hand C_{orig} for $c(X)$ tuples is also not influenced by the presence of dissimilar information objects. Finally, Figure 19(c) extends the previous scenario by including all four templates. The test is set up by augmenting TS with $a(X)$, $b(X)$ and $d(X)$ tuples with $|A| = 20$, $|B| = 90$ and $|D| = 10$. Analogous to the previous runs C_{orig} for $c(X)$ tuples is not influenced by the appearance of other types while the concentration remains static for $a(X)$, $b(X)$ and $d(X)$ tuples. Since the concentration plays a basic role in order to compute the probability of dropping tuples at nodes it shall be defined in a way that it reflects the homogeneity of tuple spaces. The scenario in Figure 19(c) is even worse since in the end ($Step \approx 90$) C_{orig} for $b(X)$ and $c(X)$ is very high. Therefore the probability of dropping a tuple matching either templates is also high. This results in a very heterogeneous structure ($\chi \to 0$).

In order to claim homogeneous structures it is required to let the constitution influence the concentration value. Therefore Equation 13 postulates C depending on χ. This is the basic for the global aim: $\aleph_\chi \to 0$. Remember, that $\aleph_\chi$ shall be heterogeneous in contrast to $\Re_\chi$ that shall be homogeneous.

$$F_{sig}(x) = \frac{1}{1 + e^{-(20x-10)}} \tag{12}$$

$$C_{mod} = F_{sig}\left(\frac{\gamma_{ij}}{\gamma_i}\right)\gamma_{ij} \tag{13}$$

$$P_{drop}^{mod} = \left(\frac{C_{mod}}{C_{mod} + K}\right)^2 \tag{14}$$

Equation 13 incorporates the presence of dissimilar tuples by including the ratio between γ_{ij} that is the number of tuples on node i matching template j and γ_i defining the total amount of tuples at i. The ratio itself is not sufficient in order to obtain the postulated homogeneity. Therefore a sigmoid function is adapted to fit in the context by stretching resp. compressing the ratio value (Equation 12). This reinforcement function is required to achieve a high attractiveness for tuples that own the majority and shall maintain it. In contrast tuples being in the minority shall avoid this tuple space so they gain a small

value. Looking at Figure 20 it shows the adapted sigmoid function in order to translate the actual ratio. The multiplication with γ_{ij} in Equation 13 is a weighting factor that scales the concentration. It shall emphasize the importance of that template.

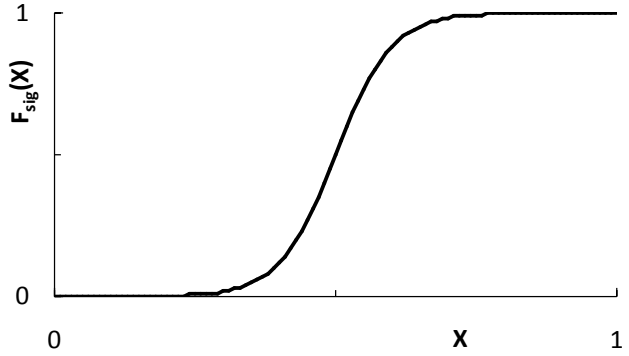


Figure 20: Sigmoid curve defined by Equation 12

The structure of the formula for computing the drop probability remains unchanged. Of course, P_{drop}^{mod} (Equation 14) stays in dependency on C_{mod} while P_{drop}^{orig} (Equation 11, page 60) is based on C_{orig}. However, applying the new computation of the concentration leads to more homogeneous clusters. Figure 19(d), 19(e) and 19(f) (page 61) show three scenarios applying C_{mod}. The scenarios are set up equal to the ones shown in Figure 19(a), 19(b) and 19(c) respectively. The comparison of two graphs arranged in a column exhibits the different development of the concentration. The graph in Figure 19(d) behaves analogous to the one in Figure 19(a). If there are no dissimilar tuples around C_{mod} approximates C_{orig}. The graph of Figure 19(e) behaves totally different to its above located counterpart: In the beginning (until $Step \approx 30$) template $a(X)$ holds the majority of tuples on that node. Hence its concentration value is the highest but as one may see it is decreasing due to a constant increase of tuple type $c(X)$. At $Step = 30$ the amount of $a(X)$ and $c(X)$ tuples is equal and consequently its concentration value, too. In fact, both are very low since it is vague whom of them shall obtain the tuple space by accumulating tuples of its type. In the following ($Step > 30$) one can see a steep increase of C_{mod} for $c(X)$ tuples while the curve for $a(X)$ tuples declines to 0 making it very unlikely to drop further tuples. At the bottom of the graph one can see the curve for $d(X)$ tuples. Since there are only 20 tuples of that type the concentration is very low in the beginning and declines over time to 0, too.

Figure 19(f) is based on the same scenario as its above suited counterpart. Similar to Figure 19(c) the concentration for $b(X)$ tuples starts with the value ($Step = 0$, $C = 90$). While the curve computed with C_{orig} is a constant horizontal line C_{mod} forces the value to decrease. The slope decreases with the number of steps and reaches its minimum at around $Step = 60$. It continues by increasing the slope and declining C_{mod}. The shown curvature is based again on a static increase of $c(X)$ tuples. Hence, one can see an increase of the curve for $c(X)$. Both curves intersect at $Step = 90$ since they show an equal amount of tuples. Since there are only 20 $a(X)$ and 10 $d(X)$ tuples in the scenario one can hardly

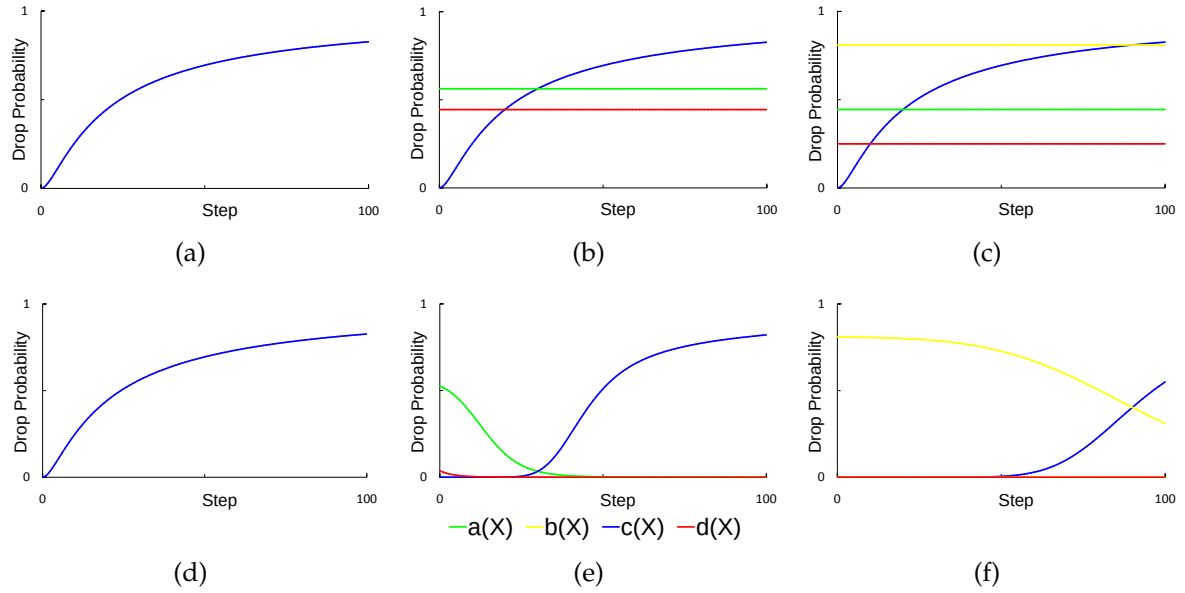


Figure 21: Comparison between P^{orig}_{drop} represented by the upper three graphs and P^{mod}_{drop} represented by the lower three graphs. The charts show the dependency on the different concentration formulas. The graphs are arranged in three columns and are based on the scenarios shown in Table 7 (page 61).

see their curves. Their concentration value is infinitesimal and thus approximates the horizontal axis. In the end ($Step > 90$) it is more likely that $c(X)$ tuples get stored at the node.

Figure 19 shows the adaptive behavior of C_{mod} while the constitution of the tuple space is changing. This results in more homogeneous cluster structures. The concentration is always dependent on similar and dissimilar tuples and regulates itself.

According to both concentration formulas the drop probability is affected. Although it is influenced by K the concentration has a strong effect of P_{drop}. However, Figure 21 shows the comparison between the two drop probabilities P^{orig}_{drop} and P^{mod}_{drop} which are based on the two aforementioned concentration formulas. The scenarios for the graphs are exactly the same as in Figure 19 (page 61). In particular, the drop probabilities have been computed using the presented concentration values and with $K = 10$.

Figure 21(a) and 21(d) are equal. In fact, they have to be since their concentration value is identical. The graphs exhibit an increase of the drop probabilities while $c(X)$ tuples get incremented with each step. The curve approximates the highest probability but since K is set to 10 the curvature flattens in the upper section.

Figure 21(b) and 21(e) show the behavior of the development of the drop probability while other tuple types are stored at the tuple space. According to their concentration value the curves in Figure 21(b) suffer from ignoring other types. Around $Step = 30$ $a(X)$ as well as $d(X)$ tuples form the minority but still hold a relatively high probability value. In average there is a 50 % chance that further tuples of these types get stored at

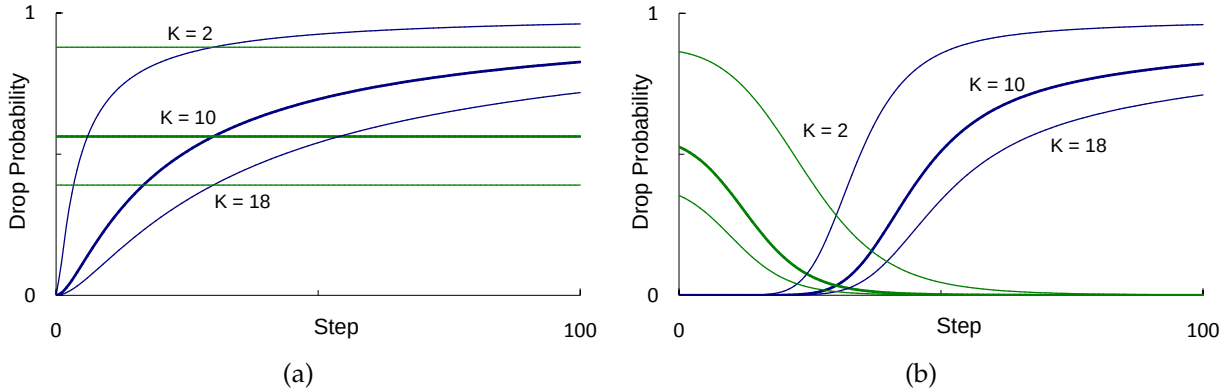


Figure 22: Comparison between P^{orig}_{drop} represented by the graph on the left and P^{mod}_{drop} represented by the graph on the right. The charts indicate drop probabilities based on different K values. The two graphs are based on the second scenario shown in Table 7 (page 61).

the tuple space. Additionally tuples of type $c(X)$ which claim the majority reach higher values. In particular, the given constellation allows three types of tuples to be stored at that node. Therefore the main objective ($\aleph_\chi \rightarrow 0$) cannot be achieved since it is unlikely that homogeneous regions emerge under the given conditions.

Figure 21(e) indicates the adaptive behavior of the drop probability. The tuple space holds only a sparse set of $d(X)$ tuples but a few more $a(X)$ objects. This forces P^{mod}_{drop} to approximate 0 while the probability value for $a(X)$ tuples claims around 50 %. Due to an increase of $c(X)$ tuples the curve representing the majority type declines rapidly and intersets the $c(X)$ curve at $Step = 30$. At this time the amounts of both types are equal: $|A| = |C|$. In the following one can see a steep increase of the slope of the $c(X)$ curve. This behavior results in homogeneous structures.

Analogous Figure 21(c) shows an even worse scenario since it includes all four templates. For all types the drop probability is relatively high. Although $d(X)$ tuples own the absolute minority ($|D| = 10$) they obtain a probability value of around 25 % constantly. The values of P^{orig}_{drop} for the other types are even higher, except for $c(X)$ tuples while $Step < 10$. In contrast Figure 21(f) shows a clear separation of drop probabilities for the respective templates. At $Step = 90$ one can see the intersection of the curve indicating the substitution of the major type.

The shown graphs in Figure 21 (page 64) are based on a fixed K value ($K = 10$). However, the explained behavior as well as the mentioned problems are independent on K. One can interpret K as a stretch value that allows the modification of the drop probability within a certain $\Delta\tau$. Figure 22(b) compares two graphs taken from Figure 21 by applying different K values. In particular, the plots from Figure 21(b) and 21(e) are reduced to template $a(X)$ and $c(X)$ while $d(X)$ has been removed from the scenario.

While Figure 22(a) has been generated using C_{orig} Figure 22(b) is based on C_{mod}. Each chart exhibits two template types and three curves respectively with K set to 2, 10 and 18.

It is noticeable that the curvatures remain almost the same. Again the approach shown in Figure 22(a) suffers from its ignorant behavior independent on K. It is even worse if K reaches small values. The curve computed with $K = 2$ indicates a situation that it is very likely that almost every tuple gets stored. That circumstance will inevitably result in a mixture of different templates. In contrast the graphs generated with P_{drop}^{mod} (Figure 22(b)) express again a very adaptive behavior. In case that other tuple types are accumulated at the tuple space the probability value tends to go down if a certain threshold is reached. Independently on K there is always a suitable ratio between the drop probabilities for the respective tuple types.

4.2. Entropy

The previous subsection discusses in detail how to achieve a good clustering. The presented formulas exhibit the adaptive behavior of the drop probability. This results in more homogeneous cluster structures by reducing mixed tuple constitutions at tuple spaces. However, this section deals with an evaluation metric represented by the spatial entropy reviewing the drop probability. Moreover, the entropy value is not only a rate for P_{drop}, it is more an indicator for the level of order in the network.

In general, it is necessary to evaluate a system in order to classify its performance. The used algorithms and system behavior itself is rated applying the metric of spatial entropy introduced by Casadei *et al.* [10]. It determines the level of organization in the network.

Equation 15 exhibits according to Casadei *et al.* [10] the entropic value for template j at node i. Moreover, $\frac{\gamma_{ij}}{\gamma_i}$ defines the fraction of similar tuples in contrast to the total amount of stored tuples. Figure 23 shows the development of the entropy by constantly increasing γ_{ij} while γ_i is kept fix. The semantic indicates that due to an increase of γ_{ij} other tuple types γ_{ik}, with $j \neq k$ disappear on the tuple space. Hence, n_χ tends to approximate 1 meaning that the constitution gets more homogeneous and results in a good entropic value ($H_{ij} \to 0$). In fact, the entropy gets worse if γ_{ij} and γ_{ik} are almost equal. In that case the constitution is very heterogeneous, $n_\chi \to 0$. However, one can estimate the codomain of H_{ij} given by $0 \leq H_{ij} \leq \frac{1}{|T|} \log_2 |T|$, with $|T|$ indicating the number of templates.

$$H_{ij} = \frac{\gamma_{ij}}{\gamma_i} \log_2 \frac{\gamma_i}{\gamma_{ij}} \tag{15}$$

In order to compute the entropy of a node the sum over all template entropies H_{ij} has to be calculated. Equation 16 exhibits the calculation of H_i that is the entropic node value. The division by $\log_2 |T|$ normalizes the node entropy within the range: $0 \leq H_i \leq 1$. However, the spatial network entropy is calculated by summing the local entropies of each node (tuple space) and dividing it by the amount of nodes in the network (see Equation (17)). The entropy level ranges from 0 (for complete order) to 1 (for total chaos). The amount of information objects of the same kind in a node determines the entropy. A large amount of tuples of the same kind means low entropy values.

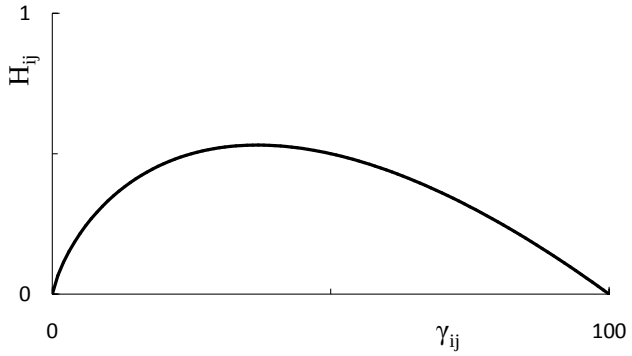


Figure 23: Entropy curve defined by Equation 15 with $\gamma_i = 100$ (page 66)

$$H_i = \frac{\sum_{\forall j \in T} H_{ij}}{\log_2 |T|} \tag{16}$$

$$H_{orig} = \frac{\sum_{\forall i \in N} H_i}{|N|} \tag{17}$$

In a system with complete order the information objects are distributed among the nodes in a way that clusters of similar objects are formed. However, when we look at the entropy described in [10] at a specific point in time it provides us with the information about the current level of order in the network but it does not take into consideration the actual distribution of tuples among the nodes.

Let us consider a scenario to analyze the entropy calculation as proposed by Casadei *et al.* Given a network of 20 nodes with 200 tuples in it. We have four groups of templates each of them containing 50 tuples. If we put all 200 tuples on one specific node while keeping the remaining 19 ones empty, we will get an entropy value of 0.05. Although this entropy appears to be good, it is not very representative of the actual state because the idea of a good entropy should be that the network is ordered meaning that we find clusters among the nodes containing similar information. But in this case we will find one node containing all objects while the other ones are empty. One can say that we got four local clusters but they are not distributed in the network. To make matters worse, the computation of the local entropy of the node that contains all the tuples results in the worst value for the entropy since the amount of tuples matching the different templates are equal, but the remaining nodes get an entropy value of 0 meaning that they are considered to have a complete order only because there are no tuples in them. The conclusion here is that the entropy calculation of Casadei *et al.* can degenerate to something that is not representative of the network.

Considering the aforementioned problem leads to the idea of calculating a weighted entropy by assigning a weighting factor to each local node entropy. Equation (17) shows the original formula for calculating the entropy as proposed in [10]. The idea is to sum the local entropies of the respective nodes (H_i) and divide it by the number of nodes in

the network ($|N|$). The approach is based on the theory that each node gets the same importance independent on the distribution of tuples.

In contrast Equation (18) postulates that each node gets a weighting value γ_i attached that indicates the importance of that node in the network. The weighting factor is proportional to the local amount of stored tuples.

$$H_{mod} = \frac{\sum_{\forall i \in N} (\gamma_i H_i)}{\Gamma} \tag{18}$$

Going back to the scenario described earlier in this section we will get an entropy value of 1 by applying Equation (18) compared to 0.05 using Equation (17). The new value signifies that the system is totally chaotic in terms of spreading tuples across the network and clustering similar tuples on a node and surrounding neighborhood while separating different objects in other regions in the network ($\Re_\chi \to 1; \aleph_\chi \to 0$).

As a (quasi-) minor issue we can also say that the original entropy calculation may accidently identify a situation of a single point of failure (since all tuples are stored in the same tuple space) as good. In the described scenario, if the node containing all the tuples is shutdown or disappears, all of the tuples are unavailable for a specific amount of time.

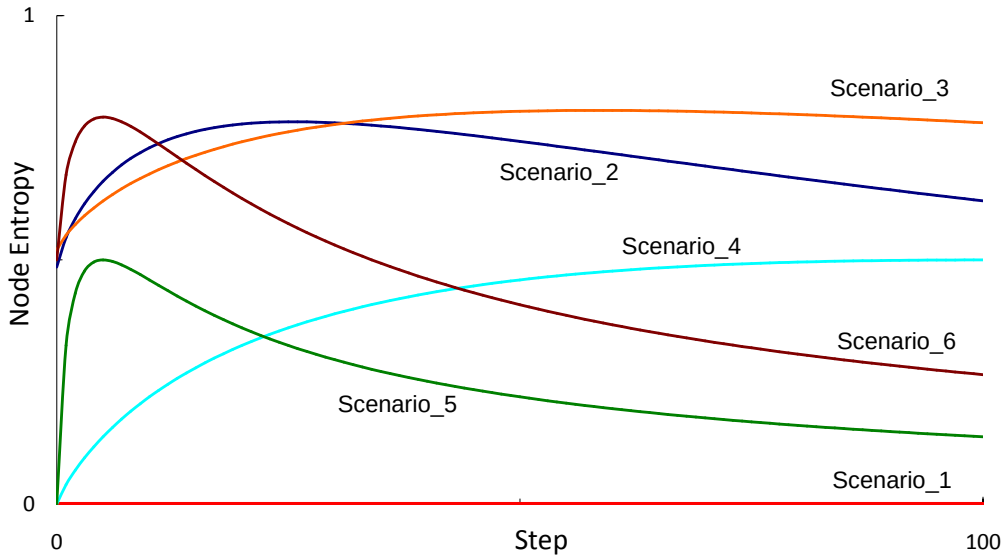


Figure 24: Node entropy curves for the respective scenarios according to Table 7 (page 61) and Table 8 (page 69) defined by Equation 16 (page 67).

Figure 24 shows according to the aforementioned scenarios (see Table 7, page 61) the development of the respective node entropies. Additionally, scenario 4 - 6 is included in the graph given by Table 8. However, since *Scenario_1* contains only $c(X)$ tuples the entropy remains constantly by 0. Thus indicating complete order. *Scenario_2* and *Scenario_3* exhibit an average value of H_i in the beginning followed by a rapid increase since the tuple space is augmented by $c(X)$ tuples. Both scenarios are set up with a basic amount

of tuples from different types. Therefore the situation tends to get chaotic. Finally the entropy decreases because $c(X)$ tuples claim the majority. The more steep slope of *Scenario_2* compared to *Scenario_3* is based on the more high-contrasted constitution of tuple types. The fraction of $c(X)$ tuples compared to dissimilar ones in *Scenario_3* is less significant than in *Scenario_2*. Hence, the entropy declines more slowly.

	$\|A\|$	$\|B\|$	$\|C\|$	$\|D\|$	K
Scenario_4	–	100	Var	–	10
Scenario_5	–	5	Var	–	10
Scenario_6	5	5	Var	–	10

Table 8: Configuration of the system environment for the test runs

However, scenario 4 - 6 shows different behaviors of the entropy. While *Scenario_4* exhibits a constantly increase of H_i due to a high amount of dissimilar tuples and finally reaches its maximum at $Step = 100$, *Scenario_5* and *Scenario_6* indicate a fast rush of the entropy followed by a light slower decrease. This phenomenon appears since the amounts of other tuples within the tuple space are significantly small.

	<*node*>	
<*scenario*>	$a(X)$	$b(X)$
	$c(X)$	$d(X)$

	Node_1		**Node_2**		**Node_3**		**Node_4**		**Node_5**		**Node_6**	
Scenario_7	25	25	30	70	100	–	5	15	5	5	20	30
	25	25	–	–	–	–	80	–	10	80	40	10
Scenario_8	1	1	500	–	200	–	5	–	1000	–	400	–
	1	1	1	1	–	–	3	2	10	–	–	–
Scenario_9	1	–	300	–	100	–	10	–	13	–	–	200
	–	–	250	50	50	80	–	–	–	–	150	250

Table 9: Scenarios of tuple distribution in a six node comprising network

Table 9 shows three additional scenarios comprising six nodes that are contained in a network. Each tuple space has a specific constitution of tuple types given by the table. Further on, Table 10 exhibits the corresponding entropic node values and finally computes the spatial network entropy according to Equation 17 and Equation 18 denoted by *H_orig* and *H_mod* respectively. One may notice that *Scenario_7* is comprised of equal sized nodes. Therefore the spatial node entropy is equal applying both formulas. However, Table 10 shows the entropic node values as a function of the constitution of tuple types.

It is noticeable that in *Scenario_8 H_mod* is smaller than *H_orig* due to a very homogeneous cluster structure within huge tuple spaces. Only *Node_1* and *Node_4* indicate bad entropic values. Since these tuple spaces contain a very small amount compared to Γ the

	H_1	H_2	H_3	H_4	H_5	H_6	H_orig	H_mod
Scenario_7	1	0.44	0	0.44	0.51	0.92	0.553	0.553
Scenario_8	1	0.02	0	0.74	0.04	0	0.301	0.029
Scenario_9	0	0.66	0.77	0	0	0.78	0.368	0.715

Table 10: Entropy calculation based on the tuple distribution given by Table 9

influence is thereby also very small. This results in very small entropic values applying Equation 18 while Equation 17 exhibits approximately a ten times higher value. Contrariwise, *Scenario_9* shows a situation where *H_orig* obtains a lower value than *H_mod* due to different cluster constitutions. *Node_2*, *Node_3* and *Node_6* own bad entropic node values while the remaining ones indicate total order. Since Equation 17 computes the arithmetic mean value of all node entropies the worst value which *H_orig* may adopt would be 0.5 since half of the network seems to be sorted. In contrast *H_mod* exhibits a higher value because *Node_2*, *Node_3* and *Node_6* contain the majority of tuples according to Γ. Based on this behavior Equation 18 computes a more realistic value of the entropy compared to Equation 17.

4.3. Pickup Probability

Section 4.1 (page 59) discusses equations for achieving a good distribution level of tuples among the nodes. The tuples shall be organized so that homogeneous regions appear; different object types shall be kept separated. Consecutively, section 4.2 (page 66) explains evaluation metrics in order to rate a given distribution of tuples. However, independently on the current state it is very unlikely that under real-world[20] conditions the distribution is perfect ($H = 0$). Therefore, there are always tuples that do not fit in the context where they stay since the SWARMLINDA system is developed on a probabilistic approach. According to this scenario the best case takes place if a template-ant passes by and picks it up due to a request that is based on tuple retrieval (see section 2.2, page 26).

Unfortunately, tuples are hardly found by template-ants if they are some sort of misplaced in a region since the pheromone trails may already evaporated. Thus there may be not much ants that transport or look for such a tuple: neither template- nor tuple-ants. It is more likely that they get locally lost in a region if the homogeneity value reaches a certain threshold. As one may think isolated tuples are not an advantage of the system since they represent resources that tend to get useless. This phenomenon may appear in several regions over time. In order to avoid locally lost objects and assure the most possible availability of resources this section deals with the movement of tuples. The idea is simple: tuples that do not fit in a certain environment shall be transferred to a - in most cases surrounding - region characterized by a suitable context. This mechanism shall guarantee more homogeneity in regions resulting in a lower entropy and finally achieves an improved performance level. Tuple storage as well as retrieval increase its efficiency

[20] A real-world scenario may contain a huge amount of nodes, tuples and different templates. In contrast one can set up an artificial-world scenario characterized by less nodes, tuples and templates.

since they get applied on a (quasi-)ordered network. Tuple movement has to cope with the following tasks:

Tuple space selection describes the process of exploring a tuple space that seems less organized. Its structure is characterized by a heterogeneous constitution of tuples.

Template selection defines the mechanism of selecting a tuple type that seems misplaced at its location. Hence it is advisable to migrate the tuple since it has a severe impact of the node entropy.

Pickup and movement deals with the removal of the selected tuple and transports it to a more appropriate region. This combines two advantages. On the one hand the entropy of the selected tuple space will increase since one misplaced tuple got removed. On the other hand the tuple space to which the ant brought the tuple may also increase its entropy since the idea is to store the tuple in an appropriate environment. This results in an improved constitution on both nodes.

Equation 22 defines the normalized probability for template j within tuple space i. It is necessary to compute $P^{norm}_{pickup}(i,j)$ for each template $j \in T$ to select a tuple type. The formula also involves a certain probability that no tuple is selected for pickup. The exponent k serves as a controlling parameter which adjusts the range of not picking up a tuple.

However, Equation 21 calculates a probability value for a template j within tuple space i that indicates the likelihood of picking up this specific template. The codomain ranges from 0 (for the tuple shall stay there) to 1 (for pick it up). The formula is comprised of the node entropy H_i and a fitness value of the respective template. The node entropy is the first part that flows into the formula. It expresses the level of organization of the node. If H_i is very high the tuple space is in a very chaotic state and thus shall be cleaned up. On the other hand if H_i approximates a low value the organization seems quite appropriate so that it is unlikely that a tuple gets selected for movement.

The second part is the fitness value of a particular template. $RF_{template}(i,j)$ as defined in Equation 19 expresses the relative frequency of template j within node i and its neighborhood NH. $RF_{template}(i,j)$ is an indicator of the fraction of tuples matching template j on i represented by γ_{ij} in contrast to the surrounding nodes denoted by $NH(i)$. Therefore, if the value approximates 0 the local amount of tuples matching j is relatively very small in contrast to its neighborhood. In this case it may be appropriate to move those tuples to one of the neighbors indicating an even higher value. On the other hand if $RF_{template}(i,j)$ tends to reach 1 the neighbors own less tuples of type j and thus it may not be useful to migrate a tuple.

$$RF_{template}(i,j) = \frac{\gamma_{ij}}{\gamma_{ij} + \sum_{\forall n \in NH(i)} \gamma_{nj}} \tag{19}$$

$$RF_{total}(i) = \frac{\gamma_i}{\gamma_i + \sum_{\forall n \in NH(i)} \gamma_n} \tag{20}$$

$$P_{pickup}(i,j) = \begin{cases} RF_{total}(i) \left(\frac{H_i + F_{sig}\left(1 - RF_{template}(i,j)\right)}{2} \right) & if\ \gamma_{ij} > 0 \\ 0 & otherwise \end{cases} \tag{21}$$

$$P^{norm}_{pickup}(i,j) = \frac{P_{pickup}(i,j)^k}{\sum_{\forall t \in T_i} P_{pickup}(i,t)}\ ;\ with\ k > 0 \tag{22}$$

In order to arrange H_i and $RF_{template}(i,j)$ equally it is necessary to compute the complement of $RF_{template}(i,j)$. The result is applied to a sigmoid function defined in Equation 12 (page 62) that has also been used for the computation of the drop probability described in section 4.1 (page 59). Again the function stretches or compresses the input value in order to reinforce the tendency and thus the exigence of tuple movement. Since both summands, H_i and $RF_{template}(i,j)$ own the same codomain mapped to [0,1] they obtain equal influence. Finally, a weighting factor represented by $RF_{total}(i)$ is introduced in order to express the importance of that node in contrast to its neighborhood.

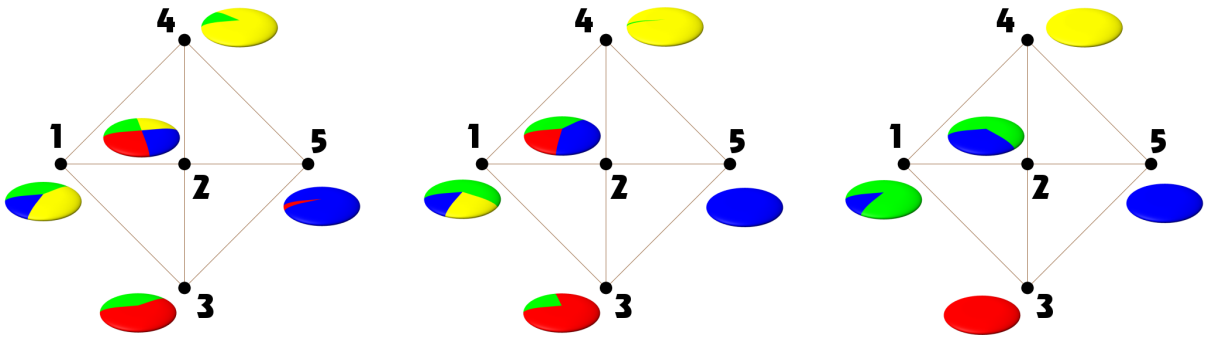
	<node>	
<step>	$a(X)$	$b(X)$
	$c(X)$	$d(X)$

	Node_1		**Node_2**		**Node_3**		**Node_4**		**Node_5**	
Step = 0	50	50	2	2	15	–	10	60	–	–
	20	–	2	2	–	20	–	–	100	4
Step = 400	66	23	2	–	8	–	1	89	–	–
	17	–	2	1	–	25	–	–	103	–
Step = 800	75	–	2	–	–	–	–	112	–	–
	12	–	1	–	–	26	–	–	109	–

Table 11: Constitutions of tuple spaces according to Figure 25

In contrast to $RF_{template}(i,j)$ Equation 20 computes the relative frequency of the total amount of tuples within tuple space i in contrast to its neighborhood NH. The ratio indicates an importance value that is normalized in [0,1]. While small values express unimportance high values indicate that this tuple space is huge compared to the surrounding ones. Hence, it shall be treated more meaningful. In general, the more tuples a tuple space contains the more it influences the entropy. Therefore, there shall be a higher probability that huge tuple spaces are cleaned up with more priority than those who hold only a few tuples.

In the scenario exhibited in Figure 25 one can see a network comprised of five nodes. Each node owns a pie chart that tells the current fractions of templates. The constitution of the tuple distribution is shown in Table 11 for each node that is contained in the network. The three rows indicate the state of the environment at $Step = 0$ resp. 400 and 800. According to Equation 22 the probability of withdrawing a tuple depends on the node entropy and the relative frequencies for the specific template as well as for the size



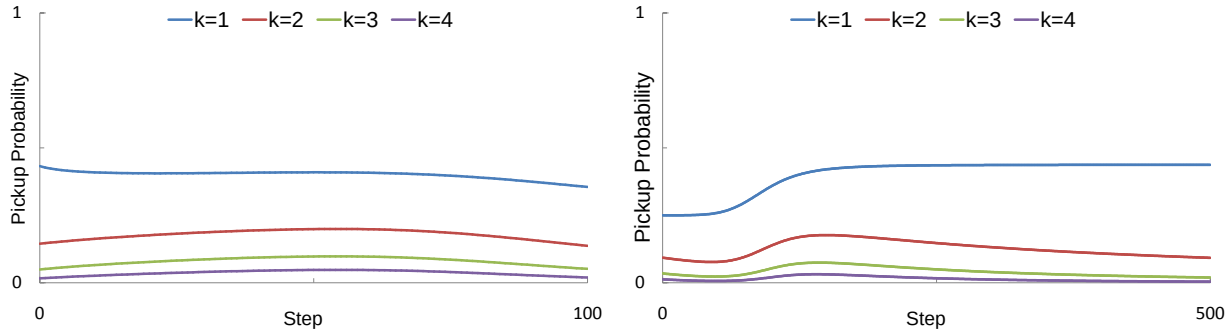
(a) $Step = 0; H_{mod} = 0.437$ (b) $Step = 400; H_{mod} = 0.27$ (c) $Step = 800; H_{mod} = 0.078$

Figure 25: Emergence of homogeneous cluster structures due to tuple movement

of the tuple space. This balances the pickup probability so that the constitution of a tuple space does not change rapidly. Although *Node_2* contains less tuples in contrast to the surrounding ones its constitution changes approximately with the same speed as the other nodes in the network. In order to decline the system entropy it is necessary that the cleaning ants prefer huge tuple spaces for tuple movement.

The mentioned scenario involves only tuple movement, i.e. there is no removal or addition of tuples included. Figure 25(a), Figure 25(b) and Figure 25(c) show different organization levels based on the specific amount of time that has been passed by represented by step. The computation of the system entropy follows the modified approach presented in Equation 18 (page 68). One can see that the level of organization increases over time. The tuple movement process effectuates the emergence of homogeneous cluster structures as postulated in Equation 8 (page 59) resulting in a heterogeneous network situation postulated in Equation 9 (page 59).

However, it is required to compute $P_{pickup}(i, j)$ for all templates $j \in T$ that are located at the current tuple space. The separate pickup probabilities indicate the necessity of withdrawing the respective tuple types in order to achieve higher homogeneity. However, in order to select a tuple the values have to be normalized. In its original form they can be hardly compared. The exponent parameter k controlls the behavior of $P^{norm}_{pickup}(i, j)$ since it scales the individual pickup probabilities. Setting $k = 1$ results in separate normalized probabilities with the characteristic that the sum is 1: $\sum_{\forall j \in T_i} P^{norm}_{pickup}(i, j) = 1$. Each tuple type obtains a normalized probability fraction that is proportional to $P_{pickup}(i, j)$. For $k > 1$ indicates that there is a certain probability dependent on k so that no tuple gets picked up. In order to stay conform to SWARMLINDA's characteristic of non-determinism it is advisable to choose $k > 1$. The current implementation adjusts $k = 2$. According to Figure 26 one can see that the gap between picking up and leaving the tuple at its current location increases while k adopts higher values. One can understand k as a factor restricting the average amount of tuples that shall be moved. In a real-world scenario with a huge amount of servers and tuples a low value of k will result in high dynamism but also



(a) Interior influence on $P^{norm}_{pickup}(i,j)$

(b) Exterior influence on $P^{norm}_{pickup}(i,j)$

Figure 26: Development of the pickup probability in dependence on k

in large network traffics. Therefore the load will inevitably increase and may impact the system performance since a request executed by a client may take longer than with lower network load. So it is a compromise between dynamism on the one side and avoiding high system overhead on the other side.

	Node_1		**Node_2**	
Scenario_10	20	30	–	–
	X	10	100	–
Scenario_11	20	30	–	–
	100	10	X	–

Table 12: Constitutions of tuple spaces according to Figure 26

Figure 26 compares the influence of internal and external tuple space structures. Both plots are of the point of view of $c(X)$ tuples. Table 12 shows the scenario in which the graph is plotted. In *Scenario_10* the X indicates that at each step one tuple of type $c(X)$ is added to *Node_1*. In the beginning X is set to 0. Based on *Node_2* the probability tends to be higher ($k = 1$) in the beginning according Figure 26(a). That is because locally (*Node_1*) the amount of $c(X)$ tuples in contrast to *Node_2* is low. They tend to move to the bigger tuple space represented by *Node_2*. With a further increase of $c(X)$ tuples the probability declines. In case that $k > 1$ one can see that the curve reaches its maximum at around $Step = 50$ and declines afterwards. From $Step = 0$ the probability is increasing since due to an increae of $c(X)$ tuples the entropy tends to get worse. On the other hand the tuple space gets more importance since it is growing. As a third factor *Node_2* attracts $c(X)$ tuples because of its relatively huge amount. At $Step = 50$ the entropy tends to get better since the constitution gets more homogeneous because $c(X)$ tuples on *Node_1* claim the majority. Second, the difference between the amounts of $c(X)$ tuples get smaller. Thus *Node_2* looses attractiveness.

Figure 26(b) shows the exterior influence on $P^{norm}_{pickup}(i,j)$. In the beginning ($Step < 60$) the probability is decreasing. *Node_2* is not an attractor since it has no tuples. After a few

steps it gains a little bit attractiveness but on the other side the weighting factor $RF_{total}(i)$ for *Node_1* declines since there are tuples in the neighborhood. At $Step >= 60$ *Node_2* reaches a threshold so that its attractiveness value increases in a way that the pickup probability for $c(X)$ tuples adopts a higher value. This is the result of the relatively strong increase of the slope. But at around $Step = 160$ the curve reaches its maximum and starts declining again. The size of tuples of *Node_2* - due to an increment of $c(X)$ tuples per step - is equal to the size of *Node_1*. Hence the weighting factor $RF_{total}(i)$ for *Node_1* compared to *Node_2* is smaller now. Therefore according to Figure 26 the probability is always based on internal and external cluster structures.

4.4. Anti-Overclustering

In section 4.1 (page 59) the modified mechanism of dropping tuples at a node is presented. It is explained in detail how the improvement produces higher homogeneous cluster structures. In section 4.2 (page 66) an evaluation metric is introduced that rates the performance of the system, i.e. the distribution of tuples among the nodes and therefore the homogeneity of tuple spaces. The entropy defines the level of organization of the network. Afterwards, section 4.3 (page 70) suggests a mechanism for selecting and withdrawing tuples within tuple spaces that seem to be misplaced. The process is part of the tuple movement algorithm. The idea is to migrate tuples between nodes so that the system entropy tends to adopt smaller values and thus indicates a higher level of organization. Finally, this section deals with the idea of avoiding overclustering of tuple spaces and achieve a more equal distribution. However, the idea of overclustering avoidance was first introduced by Casadei *et al.* in [10], but this approach is developed independently.

The concept of anti-overlcustering is based on avoiding too dense cluster. One can interpret this mechanism as a spatial separation of data, but the divided partitions shall stay in geographical proximity to each other. This enables the formation of regions holding similar information objects. In case of an unavailability of one of the nodes there are still the neighbors that can be requested. This approach is a support to fault tolerance of the system. On the other side a more equal distributed system leads to an improved load balancing between the nodes. This guarantees a higher performance of the entire system.

In order to force tuple-ants to drop tuples in the direct neighborhood of huge tuple spaces (avoid too dense clusters) it is necessary to calculate a fitness value. However, again there is no determinism for that behavior and of course, the drop locations depend on the context. The ant shall only drop the tuple if it fits in the environment. Equation 25 postulates spatial clustering. At this level there is no overclustering avoidance. Instead it supports spreading the tuples over a region with the restriction of not violating the rule of maintaining homogeneity. P^{sc}_{drop} which introduces spatial clustering is an alternative formula of computing the drop probability to Equation 13 (page 62). The main difference is that P^{sc}_{drop} additionally computes a density factor that indicates the level of fitness of a tuple in a particular region. The concentration C_{sc} is given by Equation 24 that involves the density value denoted by D_{ij}.

Equation 23 defines the density for template j for tuple space i as the ratio of the sum

of tuples matching template j within i and its neighborhood divided by the sum of all tuples within i and its neighborhood. The ratio is mapped to a codomain in [0,1]. If this values tend to get big (approximates 1) the density of tuple matching j seems very high. Therefore, the likelihood shall increase due to obtain homogeneous regions. In contrast, if D_{ij} declines and approximates 0 the set of tuples matching j seems very sparse in this region. Hence, it may appropriate not to store the tuple in this location.

$$D_{ij} = \frac{\gamma_{ij} + \sum_{\forall n \in NH(i)} \gamma_{nj}}{\gamma_i + \sum_{\forall n \in NH(i)} \gamma_n} \tag{23}$$

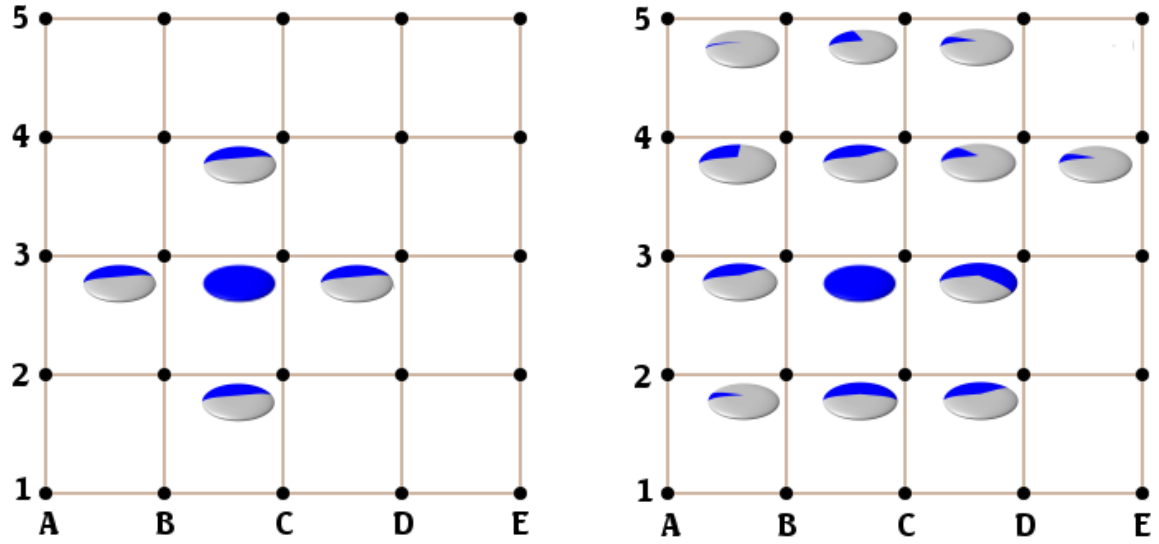
$$C_{sc} = F_{sig}\left(\frac{\frac{\gamma_{ij}}{\gamma_i} + D_{ij}}{2}\right)\gamma_{ij} \tag{24}$$

$$P^{sc}_{drop} = \left(\frac{C_{sc}}{C_{sc} + K}\right)^2 \tag{25}$$

Figure 27 shows the effect of spatial clustering applying Equation 25. The scenario comprises a 25 node containing network. The environment consists only of $c(X)$ tuples. Figure 27(a) exhibits an initial state ($Step = 0$) with an amount of 120 tuples stored in the network. In the terrain one can see five nodes that stay in proximity and hold all the tuples. The distribution of tuples among the nodes is indicated by the pie charts that are beneath the respective nodes. While the blue fraction represents the proportion of $c(X)$ tuples in contrast to the biggest tuple space the gray part exhibits the set difference. For instance, the tuple space $C3$ presents with 40 tuples the biggest node. In contrast, tuple space $B3$, $C2$, $C4$ and $D3$ own 20 tuples respectively. Hence, their pie chart holds proportionally a 50 % blue resp. gray fraction.

After 150 steps the amount of $c(X)$ tuples has increased to $\Gamma_C = 275$. In Figure 27(b) one can see the effect of spatial clustering. Compared to Figure 27(a) the tuples are more spread among the nodes. The region has extended by occupying tuple spaces in the direct neighborhood. In general, as a characteristic it is noticeable that the higher the distance to the core node ($C3$) is the less tuples indicated by the pie charts can be found. Since this strategy is based on achieving a higher spread of tuples and not avoid overclustering the core node also gains tuples. As one may notice the relation between γ_{B3} and γ_{C3} has changed between $Step = 0$ and $Step = 150$. This is due to a higher increase of tuples on $C3$ while $B3$ obtains less tuples.

In order to avoid overclustering it is necessary to constrain the amount of tuples within a tuple space by introducing a maximal threshold value represented by *max-size*. Nevertheless, *max-size* is not a hard threshold, it is more an indicator that influences the drop probability. Equation 27 defines the drop probability in consideration of avoiding overclustering. It is optional to use P^{sc}_{drop}; alternatively one can replace it with P^{mod}_{drop}. In this case the spatial clustering is disabled. The formulas can be combined independently. However, P^{aoc}_{drop} depends on an external control parameter *max-size* that has to be set manually. It controls the decrease of the drop probability. Equation 26 defines Ψ_i that is a scaling factor that adjusts P^{aoc}_{drop}.



(a) $Step = 0; \Gamma_C = 120$ (b) $Step = 150; \Gamma_C = 275$

Figure 27: A 25 nodes comprising scenario with Spatial Clustering

$$\Psi_i = F_{sig}\left(1 - \frac{\gamma_i}{2 \cdot max\text{-}size}\right) \tag{26}$$

$$P^{aoc}_{drop} = \Psi_i P^{sc}_{drop} \tag{27}$$

The decline of the drop probability according to Equation 26 is shown in Figure 28. The curvature exhibits the smooth decrease of the stretch factor Ψ_i depending on the amount of local tuples (γ_i) and the configured threshold given by *max-size*. If γ_i reaches *max-size* the drop probability declines to half of its original value ($\Psi_i = 0.5$). However, if P^{sc}_{drop} approximates 1 P^{aoc}_{drop} obtains a value around 0.5. Therefore the probability is not very low. But since P^{sc}_{drop} is very high, the tuple seems to fit in the environment. Thus there shall be at least a certain probability that allows that the tuple gets stored. Finally, if γ_i continues increasing Ψ_i declines exponentially and adopts 0 very fast. Hence P^{aoc}_{drop} also declines strongly and finally adopts 0.

Figure 29 presents the effect of overclustering avoidance. In particular, the development is shown that approaching tuples are stored in the direct neighborhood of huge tuple spaces. Figure 29(a) exhibits a scenario of five nodes and an amount of 100 tuples ($Step = 0$). The node n_2 owns the most tuples. In contrast, n_3 and n_4 obtain half while n_1 and n_5 get a quarter of the amount of tuples stored in n_2.

21 steps later one can see in Figure 29(b) that the amount of tuples has increased by 30 ($\Gamma_C = 130$). It is noticeable that n_3 and n_4 receives most of the tuples so that its amount raises around 50 %. Node n_1 and n_5 obtains a lighter increase. At $Step = 67$ more tuples have been added to the scenario so that it ends with $\Gamma_C = 190$. Figure 29(c) shows that

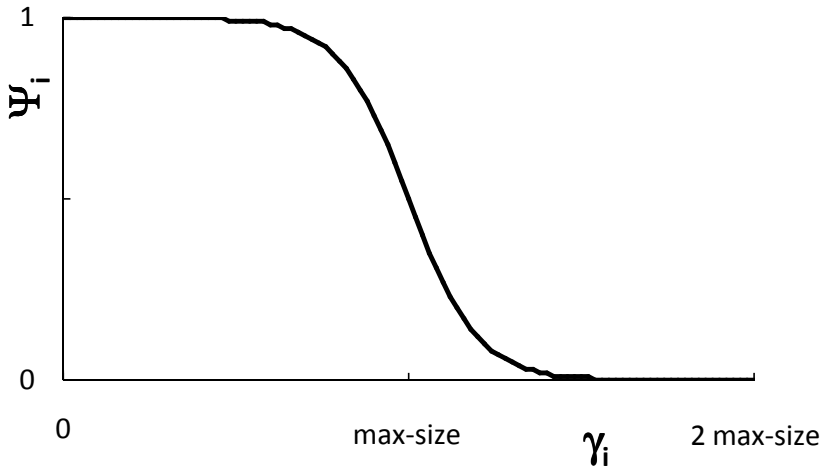


Figure 28: Development of Ψ_i in dependence on γ_i and *max-size* based on Equation 26

all tuple spaces indicates a high filling. The difference of the amount of tuples among the nodes tends to get very small. Compared to Figure 29(a) the tuples are almost equally distributed. Therefore the systems results in a higher balance.

However, a better spatial clustering can be achieved by modifying the concentration C_{sc} defined in Equation 24 (page 76) as wells as P^{sc}_{drop} defined in Equation 25 (page 76).

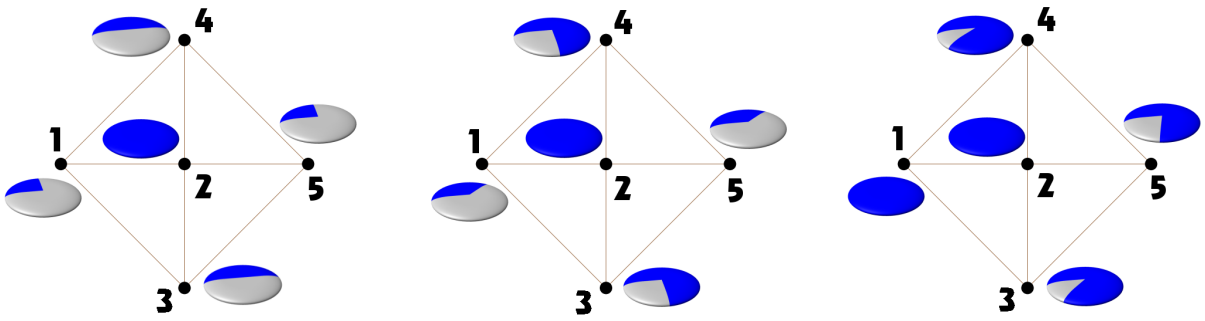
$$C_{sc^*} = F_{sig}\left(\frac{\frac{\gamma_{ij}}{\gamma_i} + D_{ij}}{2}\right) \tag{28}$$

$$P^{sc^*}_{drop} = \left(\frac{C_{sc^*}}{C_{sc^*} + \frac{K}{K_{fix}}}\right)^2 \tag{29}$$

$$P^{aoc^*}_{drop} = \Psi_i P^{sc^*}_{drop} \tag{30}$$

In contrast to C_{sc} the modified formula C_{sc^*} defined in Equation 28 is not based on the weighting factor γ_{ij}. This normalizes C_{sc^*} in a codomain of [0,1]. While low values of C_{sc^*} characterize an inappropriate location high values indicate a fitness of the tuple in the environment. By avoiding the multiplication with γ_{ij} the concentration value is simple independent on the amount of similar stored tuples. In fact, there is a certain independence on the amount of tuples but only in relation to dissimilar tuples stored at node n_i and its neighborhood $NH(i)$. Thus the density plays an important role.

In contrast to P^{sc}_{drop} the modified formula $P^{sc^*}_{drop}$ defined in Equation 29 is not simply dependent on the age factor K. It is a function based on the ratio of K divided by its



(a) $Step = 0; \Gamma_C = 100$ (b) $Step = 21; \Gamma_C = 130$ (c) $Step = 67; \Gamma_C = 190$

Figure 29: A five nodes comprising scenario with Overclustering Avoidance

initial value K_{fix} which is set to the first value that K adopts. Shortly speaking, it is the maximum age an ant get reach in a given system configuration. Thus the division by K_{fix} normalizes the ratio in [0,1]. Therefore, all parameters used in $P^{sc^*}_{drop}$ are normalized in the same codomain. This combination achieves a much better clustering than using P^{sc}_{drop} which is shown in section 5.5 (page 98).

Finally, the drop probability which avoids over-clustering P^{aoc}_{drop} defined in Equation 27 (page 77) may also be improved since it depends on P^{sc}_{drop}. Equation 30 introduces $P^{aoc^*}_{drop}$ as the extended formula. In fact, the over-clustering itself is not affected no matter which drop probability is used (P^{orig}_{drop}, P^{mod}_{drop}, P^{sc}_{drop}, $P^{sc^*}_{drop}$) since Ψ_i defined in Equation 26 (page 77) independently regulates the probability level. But in order to guarantee a better spatial clustering in combination with the over-clustering avoidance it is necessary to compute $P^{aoc^*}_{drop}$ based on $P^{sc^*}_{drop}$.

5. Experiments Showing Optimization Results

This section deals with the evaluation of the implemented SWARMLINDA system. Several experiments have been performed on the simulator, introduced in section 3.3 (page 47), in order to examine the system behavior and the characteristics of a fully decentralized application. Section 4 (page 59) describes existing formulas taken from literature and introduces modified as well as new equations making the system more effective. While the comparisons between the different approaches have already been pointed out, this section is based on the results of several test executions of scenarios that comprise a whole system environment. The simulator runs by including all presented formulas and shows the quality of collaboration of the individual approaches. However, all test runs have been performed on the topology which is given by Figure 32 (page 84).

5.1. Tuple Distribution

This subsection presents results for tuple distribution. The following shows the behavior of the applied algorithm in different system environments. The idea of tuple distribution is to spread the tuples among the nodes in the network so that similar tuples stay either at the same location or in the direct neighborhood. This process results in forming homogeneous cluster structures. The grouping of tuples is based on its type represented by a template. The characteristics of tuple distribution - invoking SWARMLINDA's *out*-primitive - including the algorithm has been described in detail in section 2.1 (page 18). While this explanation is oriented more theoretically, section 3.3 (page 47) shows technical details and exhibits how to perform a simulation with tuple distribution. Finally, section 4.1 (page 59) presents the concrete formulas which has been used in order to distribute the tuples. While section 5.4 (page 93) discusses the difference between the two mentioned approaches, this section deals with simulations based on P_{drop}^{mod} (Equation 14, page 62).

Figure 30 (page 82) shows the training of the system environment at different points in time. Table 13 summarizes the configuration of the system parameters for the respective test runs. As aforementioned Figure 32 (page 84) shows the used topology exhibiting the environment. However, the configuration table defines the following parameters: *#nodes* and *#ants* indicate the number of nodes and ants that are in the network before the test run is started. Since *topology* already defines the arrangement of nodes and links *#nodes* is given implicitly. The parameters *#tuples a(X)* resp. *#tuples b(X)*, *#tuples c(X)* and *#tuples d(X)* exhibit the amount of tuples of the given types that have already been stored in the network. As a result of the arrangement of these system parameters the value *entropy* indicates the level of order in the scenario. Afterwards *seeding* indicates the type of placed seeds before the test runs. The parameters *aoc*[21] and *sc*[22] set the clustering mode. With *num-ants* the amount of ants which get a *ttl*-value of *age* assigned is given that will be instantiated for the test-run. While *template-type* sets the type of tuple that the ants shall carry, *chosen-node* indicates the node at which the ants shall be born. In the scenarios

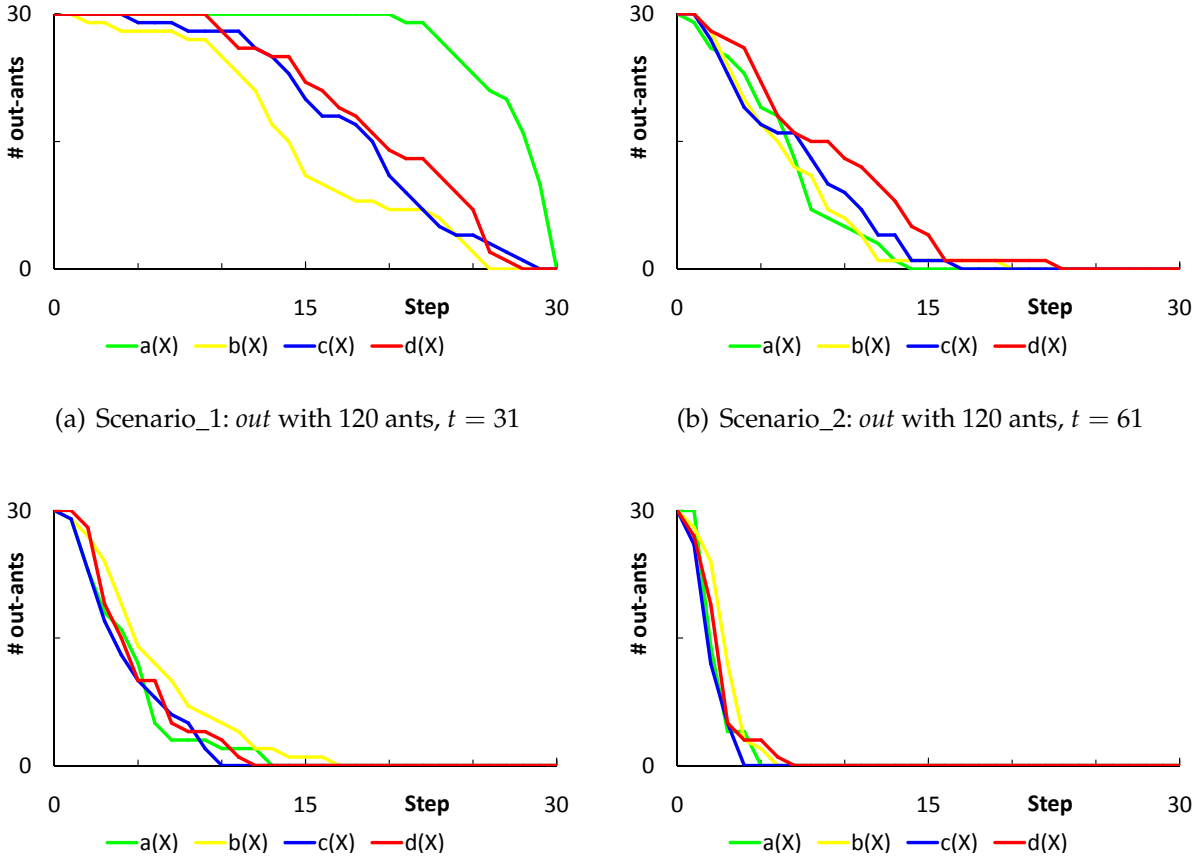
[21] Anti-Overclustering
[22] Spatial Clustering

template-type is determined as *all*. This means that 30 ants respectively for each template get instantiated. The node is also set to *all* exhibiting that the location of birth is chosen randomly. The parameter *time* is given implicitly based on the actions that have been executed before. Since the following scenarios are performed sequentially the time is increasing consequentially.

	Scenario_1	**Scenario_2**	**Scenario_3**	**Scenario_4**
#nodes	20	20	20	20
#ants	–	–	–	–
#tuples $a(X)$	30	60	90	750
#tuples $b(X)$	30	60	90	750
#tuples $c(X)$	30	60	90	750
#tuples $d(X)$	30	60	90	750
entropy	0.7323	0.5171	0.3995	0.0778
seeding	–	–	–	–
aoc	–	–	–	–
sc	–	–	–	–
num-ants	120	120	120	120
age	30	30	30	30
template-type	all	all	all	all
chosen-node	all	all	all	all
time	31	61	91	751

Table 13: Configuration of the system environment for the test runs

Figure 30(a) shows an *out* with 120 ants at time $t = 31$. The graph exhibits the dependence on the amount of steps that are required to distribute the tuples among the nodes. The vertical axis (# *out-ants*) indicates the amount of ants carrying tuples matching the respective templates. Each time an ant drops a tuple at a tuple space it dies afterwards and hence the curve declines. However, until $Step \approx 10$ the curves behave almost the same. They are approximately parallel to the *Step*-axis. Afterwards one may notice a relatively steep decrease of the curves except the green one. This indicates that ants carrying tuples matching templates of type $b(X)$, $c(X)$ and $d(X)$ successfully found tuple spaces for storing. Conversely, ants carrying $a(X)$ tuples seem to have orientation problems. It takes around 22 steps until the first few ants found appropriate tuple spaces. It is noticeable that most of the $a(X)$ ants finally drop their tuple since their *ttl*-value have been expired and thus they are forced to store it anyway. But in general, it takes a relatively long time to distribute all tuples among the nodes. One may notice that the scenario starts at time $t = 31$. Therefore, there were 30 previous steps that have already been passed. It is not recommended to show the graph of the actual 'first' scenario that appeared before *Scenario_1* since all curves are parallel to the *Step*-axis until $Step \approx 29$ and finally decline strongly in one step to 0. This phenomenon is normal because since there are no tuples stored in the environment as well as scents there is no attraction for the *out*-ants. This results commonly in a worse entropy. As one may notice *Scenario_1* starts with an entropic value that



(a) Scenario_1: *out* with 120 ants, $t = 31$

(b) Scenario_2: *out* with 120 ants, $t = 61$

(c) Scenario_3: *out* with 120 ants, $t = 91$

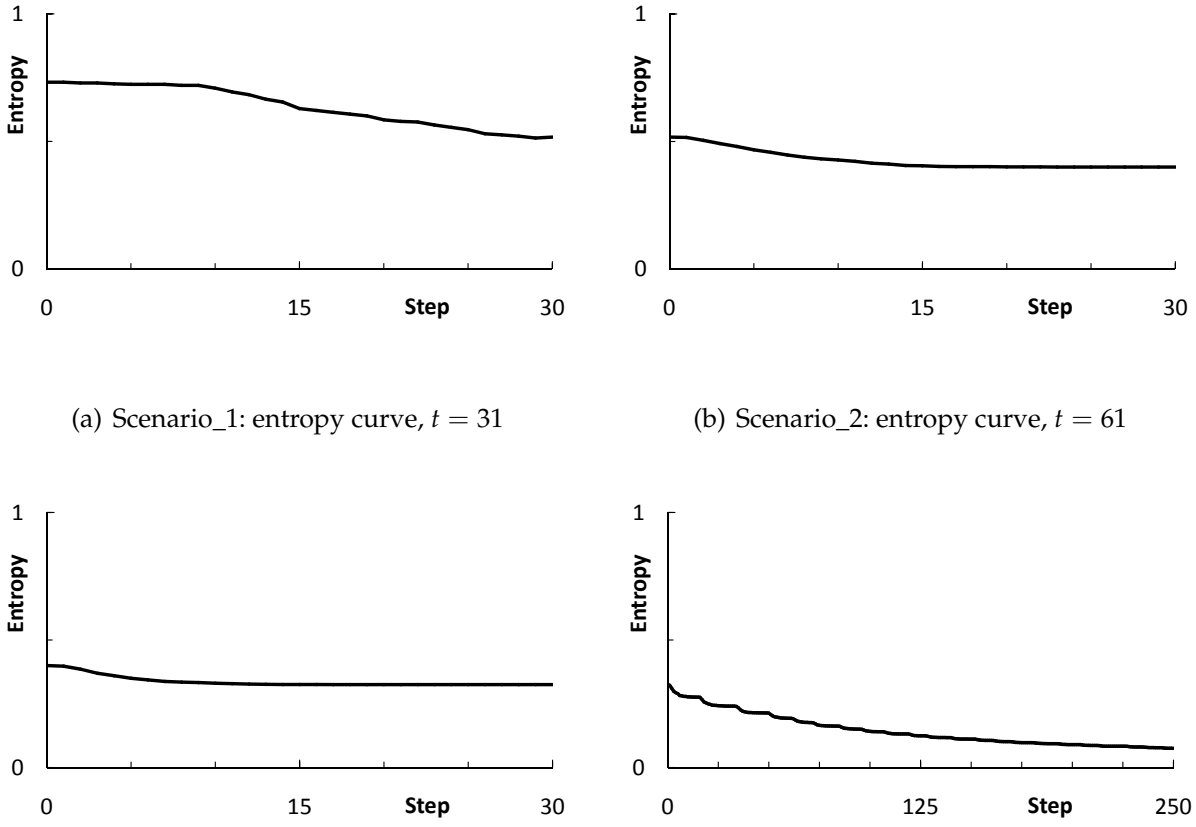
(d) Scenario_4: *out* with 120 ants, $t = 751$

Figure 30: Training effect of the system by executing *out*-primitives at different points in time

indicates a chaotic state. With that *Scenario_1* starts where the actual 'first' scenario ends.

Figure 31 shows, parallel to Figure 30, the development of the spatial network entropy as steps passed by. Therefore the respective graphs from both figures are correlated. In particular, the graph of Figure 31(a) shows the entropy according to the development of the graph in Figure 30(a). Analogous the entropy is approximately constant until $Step \approx 10$ since no tuples get stored during this time. Afterwards the curve declines due to a distribution of tuples that form cluster structure.

The scenario shown in Figure 30(b) starts at time $t = 61$ which is the end of *Scenario_1*. Looking at the different curves it is noticeable that they behave totally different due to an improvement of scents and emerging clusters. Based on the training effect the ongoing ants are able to track pheromone trails more significantly that have been formed in *Scenario_1*. Therefore, they find suitable tuple spaces in shorter time. By comparing the graphs of Figure 30(a) and 30(b) at $Step = 15$ it is obvious that according to *Scenario_1* not half of the tuples have been stored while *Scenario_2* is almost done with storing all



(a) Scenario_1: entropy curve, $t = 31$

(b) Scenario_2: entropy curve, $t = 61$

(c) Scenario_3: entropy curve, $t = 91$

(d) Scenario_4: entropy curve, $t = 751$

Figure 31: Development of the entropy during the test runs of the *out*-primitives

tuples. Also the curvatures exhibit an even stronger decrease of the slope. Although $a(X)$ tuples had a bad start according to *Scenario_1* the progress in *Scenario_2* is very significant since they behave even better than the other ones. Finally, at $Step \approx 23$ the last ant drops its tuple. According to the mentioned behavior the entropy curve shown in Figure 31(b) declines constantly until $Step \approx 15$ due to the strong distribution of tuples. Afterwards the curve is more a parallel to the *Step*-axis.

Figure 30(c) which shows the results of *Scenario_3* takes place directly after *Scenario_2*. It is noticeable that based on a further improved training effect the decrease of the curves get stronger. The curves exhibit an exponential decline. While at $Step \approx 5$ not half of the amount of tuples in *Scenario_2* have been stored *Scenario_3* indicates a success rate of around 66 %. This means that there is approximately $\frac{1}{3}$ tuples left that have to be dropped. At $Step \approx 12$ almost all tuples have been stored. This is on average three steps earlier than in *Scenario_2*. Finally, at $Step = 16$ the last tuple gets stored. According to the given storage distribution the entropy, shown in Figure 31(c), decreases strongly up to $Step \approx 5$. Afterwards, it keeps on declining slowly until $Step \approx 10$.

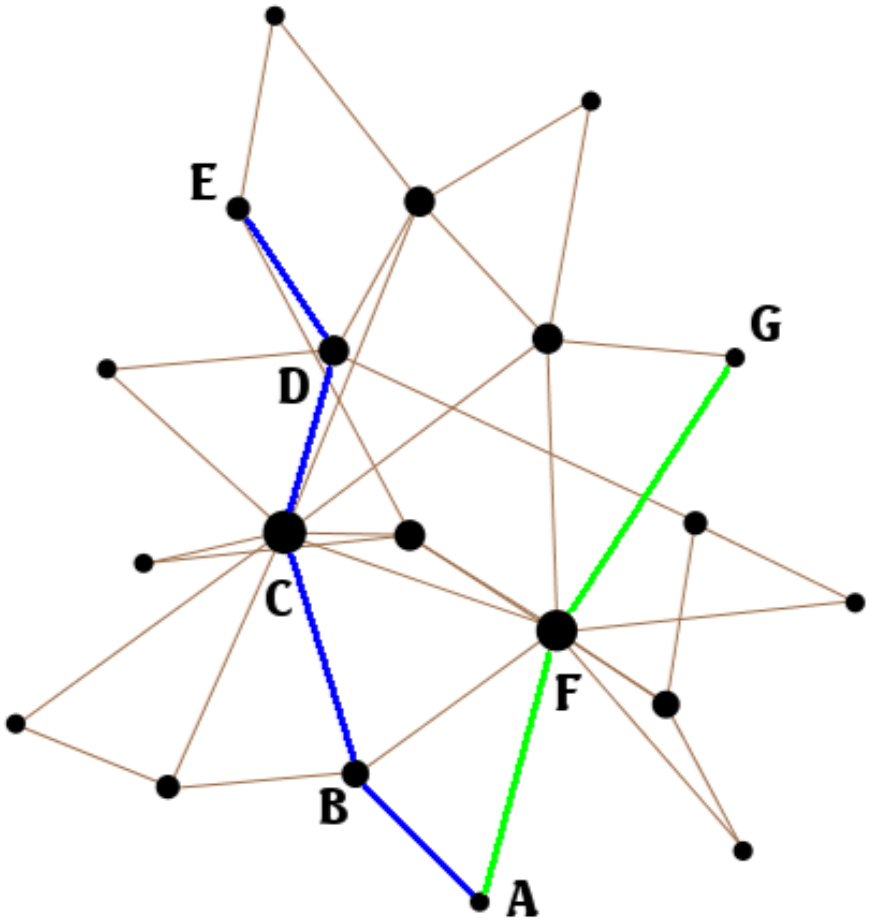


Figure 32: Network topology with two marked paths: $\overline{AE}$ (blue) and $\overline{AG}$ (green)

While *Scenario_1* to *Scenario_3* are directly linked to each other *Scenario_4* has been executed later. Between *Scenario_3* and *Scenario_4* there have been several other *out*-primitives performed. Therefore, the system reaches a higher training level. In the graph of Figure 30(d) one can see that the exponential decline is stronger compared to the graph shown in Figure 30(c). At $Step \approx 5$ almost all tuples have been stored. Compared to the graph of *Scenario_3* it took around 12 steps. However, the last tuple gets stored at $Step = 7$. In contrast to the other entropy graphs Figure 31(d) shows the development of the system entropy starting from the end of *Scenario_3* (Figure 31(c)) and ranges to the end of the current scenario. Following the curvature one may see a very compressed form of the curve from Figure 31(b) or 31(c) that repeats itself several times. During each execution of the *out*-primitive the $\Delta\tau$ - indicating the range of the entropy between $Step = 0$ and $Step = 30$ - declines. Finally, $\Delta\tau$ approximates a very small value so that it seems that the different curvatures merge into one line that decreases constantly.

However, recapitulated Figure 30 shows a constant improvement of the system. Due to the training effect the ants are able to track pheromones, follow emerging trails and find suitable tuple spaces in shorter time. The scenarios exhibit that the more the system gets

trained - i.e. putting a lot of tuples in it - it gets more effective due to an emergence of more significant trails and more homogeneous cluster structures. Analogous, Figure 31 shows a very improved development of the system entropy. In the beginning it indicates an almost chaotic state and ends with a very organized system. Thus the tuples get in fact clustered among the nodes.

Even though the network contains only 20 nodes it is not as trivial as one may think of to let an ant route from its current location to a specific one. For instance, Figure 32 shows as aforementioned the arrangement of nodes and links in which the different scenarios have been executed. Based on the given topology according to Barabasi's approach the nodes are linked following a scale-free power-law distribution (cf. [3]).

Node	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Connectivity	2	4	10	5	3	9	2
Probability	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{10}$	$\frac{1}{5}$	$\frac{1}{3}$	$\frac{1}{9}$	$\frac{1}{2}$

Table 14: Node connectivity and path probability for network topology *type_a* according to Figure 32

Assume a scenario without scents. An ant stays currently at node A and wants to move to E (see Figure 32) since there are similar tuples around. Assume further that the shortest route is given by $R = \overline{ABCDE}$. Table 14 shows the degree of each node and the probability value that an ant takes a specific outgoing link. Then the probability for taking path $\overline{ABCDE}$ is calculated as follows: $\prod_{\forall r \in R \setminus \{E\}} P_R = \frac{1}{400} = 0.0025$. Thus, the probability for taking the path R is only 0.25 %.

Assume, alternatively that the ant which stays currently at node A wants to go to node G by taking the shortest route given by $Q = \overline{AFG}$. According to Table 14 the probability is calculated as follows: $\prod_{\forall q \in Q \setminus \{G\}} P_Q = \frac{1}{18} = 0.0\overline{5}$. The probability for taking path Q is with 5 % significantly higher, but the route given by Q is only via one hop indicated by F. Thus the nodes stay in proximity. Considering the fact that the system needs only a few steps (see Figure 30(d), page 82) in order to assign all tuples to suitable tuple spaces it would take much longer if there are no scents involved in the system.

5.2. Tuple Retrieval

While the previous subsection deals with the evaluation of tuple distribution represented by SWARMLINDA's *out*-primitive this subsection focuses on tuple retrieval. This invokes SWARMLINDA's *in*-primitive as described extensively in section 2.2 (page 26). While this explanation is oriented more theoretically section 3.3 (page 47) shows technical details and exhibits how to perform a simulation executing tuple retrieval.

Figure 33 (page 87) shows different scenarios performing tuple retrieval. Table 15 summarizes the configuration of the system parameters for the respective test runs. *Scenario_5*, *Scenario_6* and *Scenario_7* are comprised of the same system parameters as *Scenario_1*, *Scenario_2* and *Scenario_3*. Thus the respective simulation takes place under the same conditions. Therefore it is possible to compare *Scenario_1* with *Scenario_5*, *Scenario_2* with *Sce-*

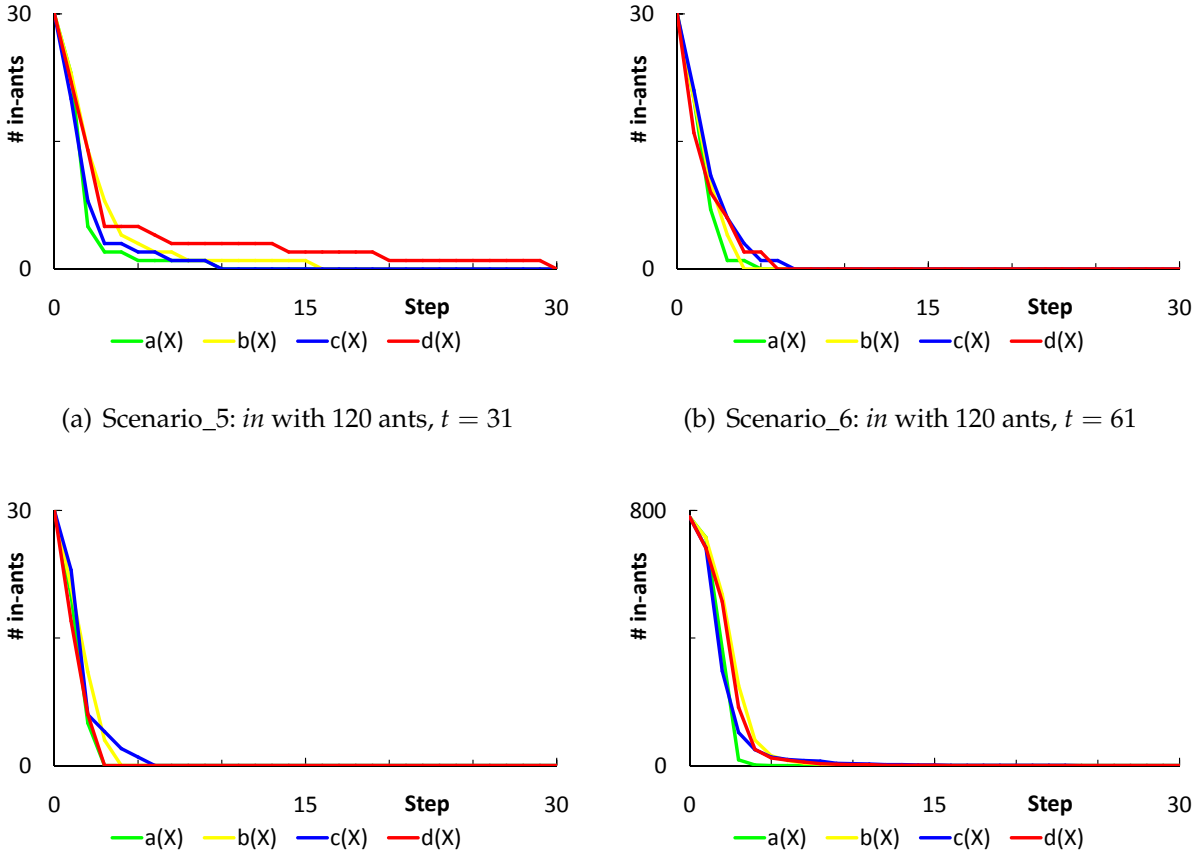
nario_6 and *Scenario_3* with *Scenario_7*. The test run for *Scenario_8* has been performed directly after *Scenario_4*.

	Scenario_5	**Scenario_6**	**Scenario_7**	**Scenario_8**
#nodes	20	20	20	20
#ants	–	–	–	–
#tuples $a(X)$	30	60	90	780
#tuples $b(X)$	30	60	90	780
#tuples $c(X)$	30	60	90	780
#tuples $d(X)$	30	60	90	780
entropy	0.7323	0.5171	0.3995	0.0753
seeding	–	–	–	–
aoc	–	–	–	–
sc	–	–	–	–
num-ants	120	120	120	3120
age	30	30	30	30
template-type	all	all	all	all
chosen-node	all	all	all	all
time	31	61	91	781

Table 15: Configuration of the system environment for the test runs

However, Figure 33(a) shows an *in* with 120 ants at time $t = 31$. Analogous to tuple distribution a decrease of the curve in the plots exhibits success. In particular, the template-ant found a matching tuple and brought it back to the requesting node. It is noticeable that at $Step \approx 4$ almost all ants already found a matching tuple. There are in average around 5 ants of each type left that are still looking for one. Compared to Figure 30(a) (page 82) the *out* takes much longer in order to store the amount of tuples carried by ants than retrieving tuples that match the templates the *in*-ants carry. Since both scenarios have been executed under the same conditions the training of the network remains the same. In fact, a high influence of the drop probability P_{drop} - independently which modification is used - discussed in section 4.1 (page 59) is the concentration C. C itself - again, independently on the used modification - is oriented based on the amount of similar tuples within a tuple space. Since, *Scenario_1* contains less tuples it will take long to find appropriate tuple spaces unless K tends to adopt smaller values. In contrast, tuple retrieval does not postulate a huge amount of tuples in order to accelerate the actual retrieval process. Once a tuple is found that matches a given template it will be withdrawn immediately from the tuple spaces. Since the network is already augmented with pheromones the ants find tuples very fast. Nevertheless, a few ants require much more time to detect suitable tuples indicated by the red and yellow curves. Especially, the tracking of $d(X)$ tuples seems to be very exhausting.

Figure 34 shows, analogously to Figure 31 (page 83), the development of the entropy for the respective test runs given by Figure 33. However, Figure 34(a) exhibits the entropy of *Scenario_5*. In the beginning until $Step \approx 5$ the entropy is declining strongly. This is due



(a) Scenario_5: *in* with 120 ants, $t = 31$

(b) Scenario_6: *in* with 120 ants, $t = 61$

(c) Scenario_7: *in* with 120 ants, $t = 91$

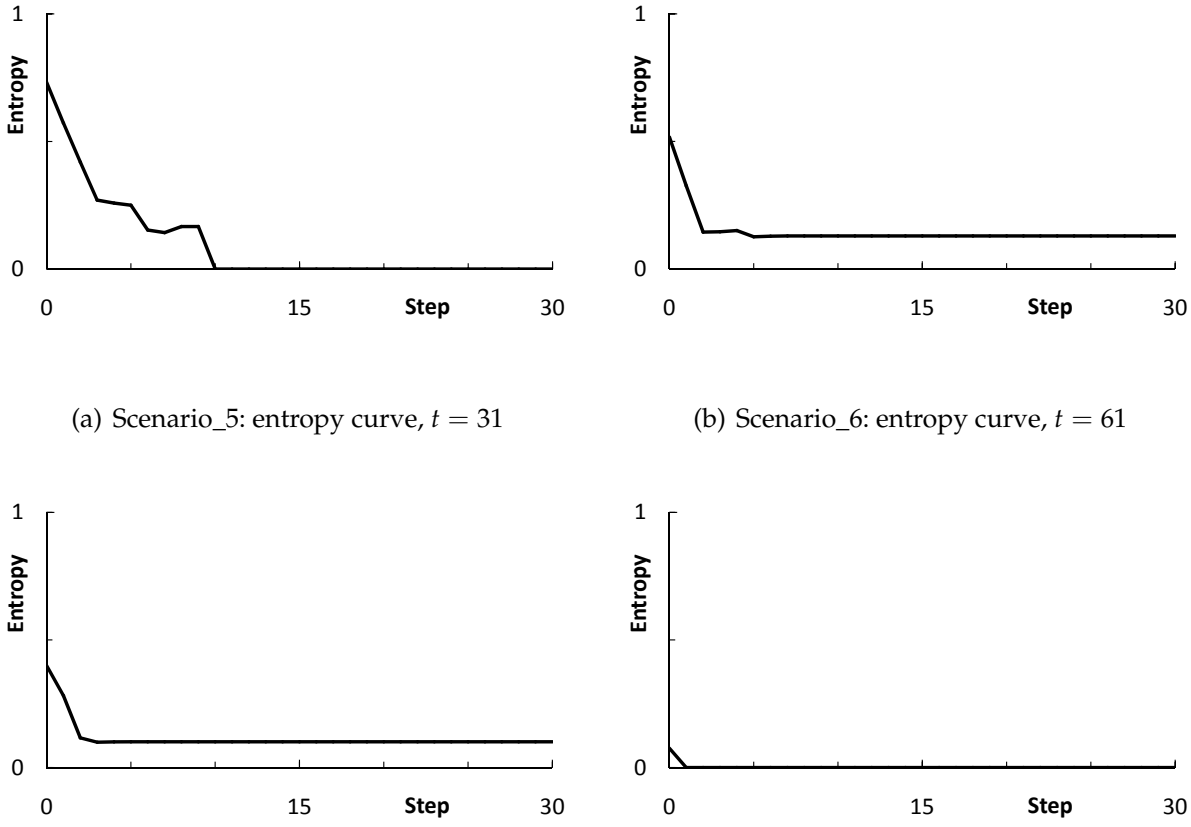
(d) Scenario_8: *in* with 3120 ants, $t = 781$

Figure 33: Training effect of the system by executing *in*-primitives at different points in time

to an increase of homogeneity within tuple spaces. The withdrawing of tuples leads also to the phenomenon of supporting the process of achieving a higher level of organization in the network. The, in the beginning, some sort of misplaced tuples get retrieved. On the other hand the already formed clusters avoid an increase of heterogeneity since the drop probability for dissimilar tuples tends to decline strongly as shown in section 4.1 (page 59). Therefore the combination of *in*- and *out*-primitives achieves an even higher level of organization.

However, the entropy in Figure 34(a) indicates an increase for $7 \leq Step \leq 9$. This occurs for instance if an ant withdraws a tuple from a tuple space so that the amount of tuples of different types approximate each other. Assume there is one $a(X)$ tuple and two $b(X)$ tuples. If the ant withdraws one $b(X)$ tuple the entropy gets worse since $|A| = |B|$. At $Step \geq 10$ the entropy reaches 0. That means that the scenario does not contain any heterogeneous tuple spaces.

Figure 33(b) and 33(c) show test runs executing the *in*-primitive with 120 tuples each



(a) Scenario_5: entropy curve, $t = 31$

(b) Scenario_6: entropy curve, $t = 61$

(c) Scenario_7: entropy curve, $t = 91$

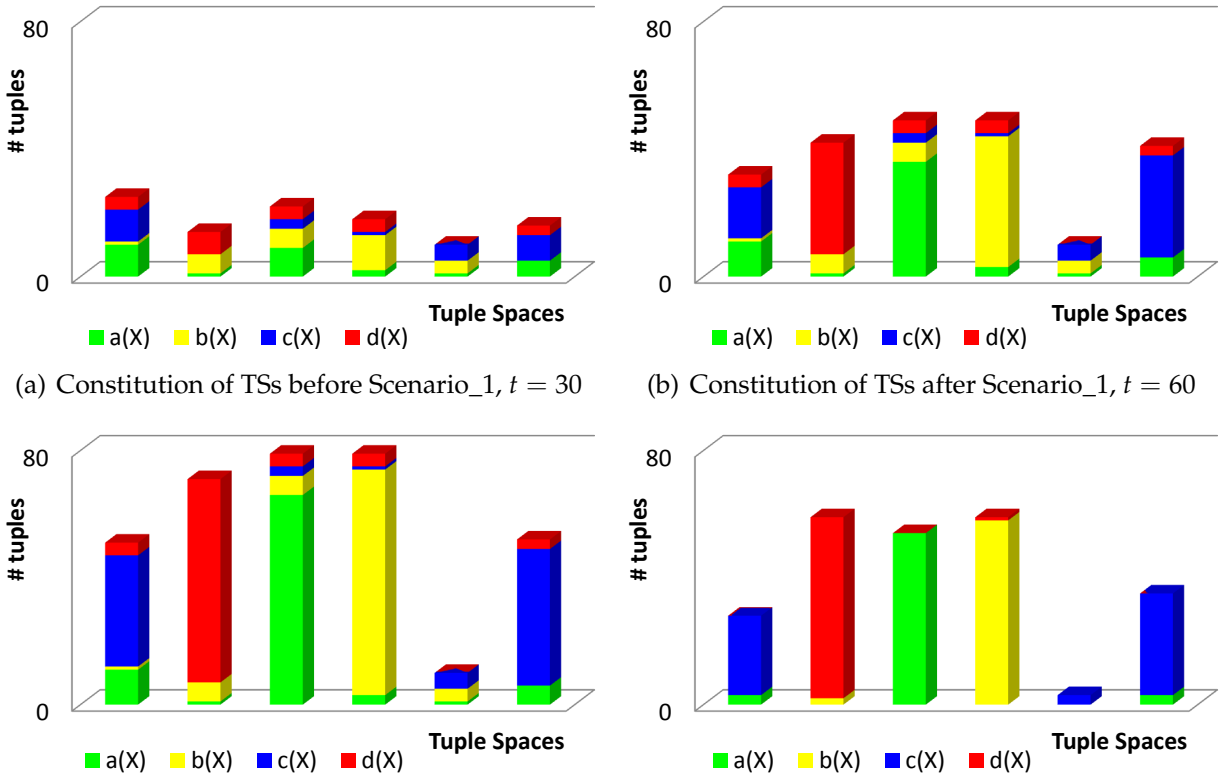
(d) Scenario_8: entropy curve, $t = 781$

Figure 34: Development of the entropy during the test runs of the *in*-primitives

at time $t = 61$, $t = 91$ respectively. The plots vary slightly. Both graphs exhibit an exponential decline and thus ants find matching tuples very fast. The average success time - indicating at which most of the ants successfully found a tuple - is around $Step = 4$ for *Scenario_6* and around $Step = 3$ for *Scenario_7*. Comparing Figure 33(a), 33(b) and 33(c) one can see a continuous improvement of the training of the network since the retrieval time decreases.

Figure 34(b) and 34(c) show the respective network entropies for *Scenario_6* as well as *Scenario_7*. Following the curvatures one may notice that they behave almost the same. They indicate a strong decline in the beginning ($Step \approx 3$) and continue as a parallel to the *Step* axis ($Step \geq 10$).

In contrast to Figure 33(a), 33(b) and 33(c), Figure 33(d) shows an execution of the *in*-primitive at $t = 781$ performing an *all-in* that is characterized by retrieving all tuples stored in the network. According to Table 15 (page 86) the environment contains 780 tuples of each type ($\Gamma = 3120$). Based on the well trained network the *in* takes again around four steps for retrieving most of the tuples. At $Step \approx 8$ all tuples have been retrieved.



(a) Constitution of TSs before Scenario_1, $t = 30$

(b) Constitution of TSs after Scenario_1, $t = 60$

(c) Constitution of TSs after Scenario_2, $t = 90$

(d) Constitution of TSs after Scenario_7, $t = 120$

Figure 35: Development of the constitutions of tuple spaces during the test runs of *out*- and *in*-primitives

Therefore the network is empty. Although there is a huge amount of template-ants the performance of the system does not suffer under an impact. Instead it shows the same robust performance as in Figure 33(c). Since the system is already in a very organized state and all tuples get retrieved Figure 34(d) shows the entropy for *Scenario_8*. The value declines immediately to 0.

Figure 35 shows the constitutions for a set of tuple spaces contained in the network. Figure 35(a) indicates an almost chaotic state that has already been explained in *Scenario_1*. As expected the different constitutions within the tuple spaces are very heterogeneous. This appears because the network was empty so the ants had no orientation where to drop the tuples. Looking at Figure 35(b) and 35(c) one may notice the development of the constitutions of the respective tuple spaces while tuples have been added to the environment by executing *out*-primitives. Therefore the distribution of tuples result in an improved entropy due to a higher level of organization. Finally, Figure 35(d) shows an execution of the *in*-primitive according to *Scenario_7*. This erases almost all heterogeneous tuple space structures. Therefore the combination of *out*- and *in*-primitives result in an improvement of the system by forcing homogeneity and avoiding heterogeneity.

5.3. Tuple Movement

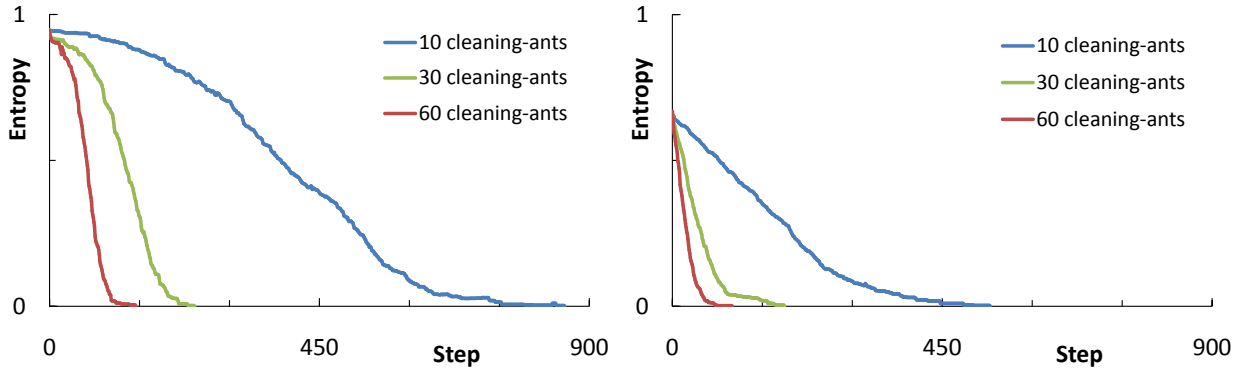
The previous subsections present evaluations of the algorithms for tuple distribution and retrieval. It is shown that both mechanisms are able to balance the system in order to achieve good entropic values. Nevertheless, it is, in fact, possible that specific constellations of tuples in the environment confuse the system in a way that it might be hard to achieve a high level of clustering. Besides, there may be some tuples that are less requested - or even less found - but seem to be misplaced. In order to improve the probability of retrieving them anyway it may be helpful to migrate them to a different location.

	Scenario_9	**Scenario_10**	**Scenario_11**
#tuples $a(X)$	100	100	100
#tuples $b(X)$	100	100	100
#tuples $c(X)$	100	100	100
#tuples $d(X)$	100	100	100
entropy	0.9466	0.9466	0.9466
seeding	400 seeds randomly		
aoc	–	–	–
sc	–	–	–
num-ants	10	30	60
age	*out*-phase = 20		
chosen-node	all	all	all

Table 16: Configuration of the system environment for the test runs

However, this section presents evaluations for tuple movement. Figure 36 shows the results of tuple movement according to the scenarios given in Table 16. The scenarios are based on an initial empty environment. At $Step = 0$ a seeding with 400 seeds has been performed randomly. As shown in Table 16 there are 100 tuples each placed on random nodes in the network. The distribution follows approximately an equipartition that results in a very bad entropy since the environment is characterized by heterogeneity. The scenarios starting at $Step = 1$, thus the environment is untrained. As explained in section 2.3 (page 30) cleaning-ants do not participate in the aging mechanism. Once they pick up a tuple they metamorphose to conventional *out*-ants and get an age assigned which is set to 20. The test runs are based on the convention that the *out*-ant does not die after it completes its task. Instead, it metamorphoses back to a cleaning-ant. Therefore, the individuals are active all the time. The scenarios indicate how fast the system can be organized in dependence on the amount of used cleaning-ants.

Figure 36(a) exhibits the amount of steps needed in order to achieve a fully organized system given the respective amount of ants given by *Scenario_9*, *Scenario_10* and *Scenario_11*. All three curves have in common that they can be divided into three parts: the first part is characterized by an increasing negative slope and finally converges to a static section. In this period the system state is almost chaotic. There are neither cluster of tuples nor pheromones that indicate a successful path. Hence, the cleaning-ants roaming approximately blind through the network trying to collect tuples and sense scents.



(a) Cleaning in an untrained network

(b) Cleaning in a trained network

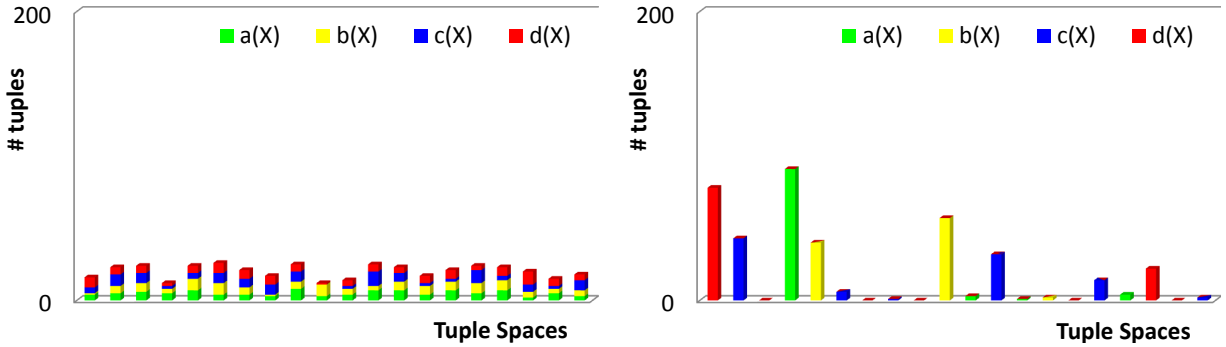
Figure 36: Development of the entropy in dependence on the amount of cleaning ants during tuple movement

Slowly they find pheromone trails that have been emerged more by coincidence. However, by tracking those paths ants carrying similar tuples tend to head towards the same direction. The end of phase 1 is determined by some basic trails that have been establish including small clusters.

	Scenario_12	**Scenario_13**	**Scenario_14**
#tuples $a(X)$	200	200	200
#tuples $b(X)$	200	200	200
#tuples $c(X)$	200	200	200
#tuples $d(X)$	200	200	200
entropy	0.66	0.66	0.66
seeding	400 seeds randomly		
aoc	–	–	–
sc	–	–	–
num-ants	10	30	60
age	*out*-phase = 20		
chosen-node	all	all	all

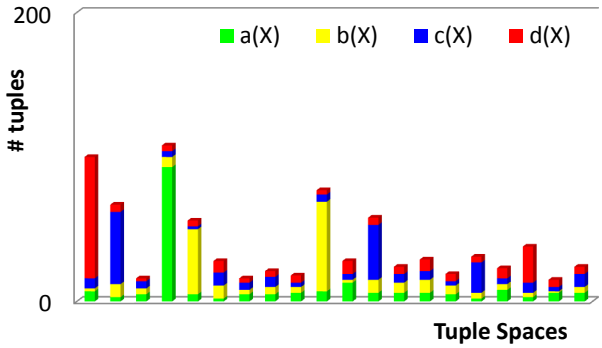
Table 17: Configuration of the system environment for the test runs

The second part is characterized by an almost static slope. In this period the ants moving tuples around and distribute them to appropriate tuple spaces. This phase is defined by a strong decrease of the entropy and exhibits the most effective execution of tuple movement. In the following the negative slope declines again indicating the beginning of part three. The environment is approximately sorted. There are only a few tuples left that need to be migrated. With it the performance of tuple movement slows down. In general, phase two is the main period in which most of the entropy declines. As mentioned the mechanism of tuple performance reaches its maximal load. Nevertheless, all three phases require approximately the same amount of time to be executed. The number of cleaning-

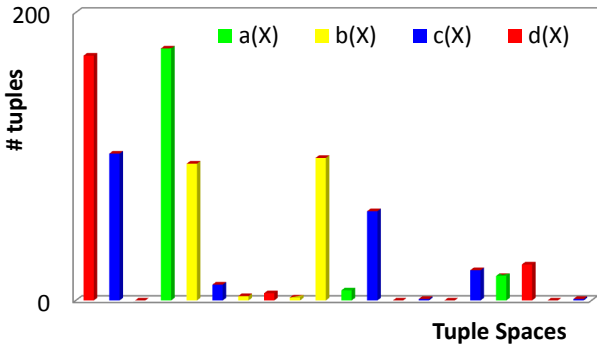


(a) Untrained network with seeding

(b) Clustering by tuple movement



(c) Trained network with seeding



(d) Reinforcement of clustering

Figure 37: Self-organization of tuples in an untrained and trained network during tuple movement

ants can be interpret as a scaling factor that regulates the rate of self-organization. In fact, more ants achieve a specific level of organization in less time. But there is a threshold at which the ants cannot increase a given performance. As shown in Figure 36(a) and 36(b) the amount of required steps is not proportional to the amount of deployed ants.

Figure 36(b) is based on the scenarios given in Table 17. These scenarios have been performed in the state of the end of the scenarios given by Table 16. Additionally, a new seeding has been executed so that the ordered network gets confused again. Analogous to the first seeding the second one follows approximately an equipartition. Hence, the system entropy claims a value of 0.66.

Since the environment is already trained one may notice that the redistribution of the new seeds performs much faster than indicated in Figure 36(a). Additionally, the first phase does not take place since the ants are not required to orientate themselves. They start determined migrating the tuples to already formed clusters. Since *Scenario_12*, *Scenario_13* and *Scenario_14* exhibit that it is sufficient to perform phase two and three by avoiding phase one the redistribution of tuples is accomplished around 33 % faster.

According to the mentioned scenarios Figure 37 shows the distribution of tuples among the nodes in the network at specific points in time. Figure 37(a) exhibits the state of the environment given by *Scenario_9*, *Scenario_10* and *Scenario_11* before the cleaning-ants have

been created. The columns indicate the respective constitutions of tuple spaces given by the colors that represent the templates. The state exhibits an approximately equipartition that results in a chaotic entropic level.

Figure 37(b) shows the state after the first cleaning process. The state is totally ordered indicated by fully homogeneous cluster structures. Thereupon, Figure 37(c) exhibits the state given by *Scenario_12*, *Scenario_13* and *Scenario_14* before the cleaning-ants have been created. By applying an uniformly distributed seeding with 400 tuples on the environment given by Figure 37(b) leads to the environment shown in Figure 37(c).

Finally, Figure 37(d) represents the state of the distribution of tuples after the second cleaning process. With approximately 66% of the required time of the previous cleaning process the second one achieves again a situation characterized by homogeneous cluster structures. This execution reinforces the first clustering procedure.

5.4. Comparison of the improved Metrics

Section 5.1 (page 80) presents evaluations for distributing tuples among the nodes. In the following (secion 5.2, page 85) the results of tuple retrieval has been discussed in order to find specific tuples in the environment. Finally, the evaluation of tuple movement (section 5.3, page 90) has shown characteristics of achieving a high level of organization by separating dissimilar tuple types while concentrating similar ones. All tests for these mechanisms have been performed based on the formulas given by:

- P_{drop}^{mod} for computing the drop probability (Equation 14, page 62)
- H_{mod} for computing the spatial network entropy (Equation 18, page 68)
- P_{pickup}^{norm} for computing the pickup probability (Equation 22, page 72)
- P_{ij} for computing the path selection (Equation 5, page 25)

However, this section deals with the comparison of the original drop probability P_{drop}^{orig} (Equation 11, page 60) and entropy calculation H_{orig} (Equation 17, page 67) proposed by Casadei *et al.* in [10] in contrast to the in this report developed modified drop probability P_{drop}^{mod} as well as the modified entropy computation H_{mod}.

Based on the studies presented in [22] Figure 38 (page 95) shows the comparison of the four different scenarios defined in Table 18. All scenarios start with an initial empty network that is comprised of 20 nodes. Each listed scenario is composed of a different combination of the presented equations. The parameter *num-ants* is set to 1000. But this amount is not instantiated at once in the environment. The goal was to be a little closer to a real system. The creation of 1000 tuples at the same time, although not impossible, is unlikely. Hence, the instantiations of the tuples follow the given intervals:

- 40 tuple-ants (10 per type) were instantiated at 5 times.
- 200 tuple-ants (50 per type) were instantiated at 4 times.

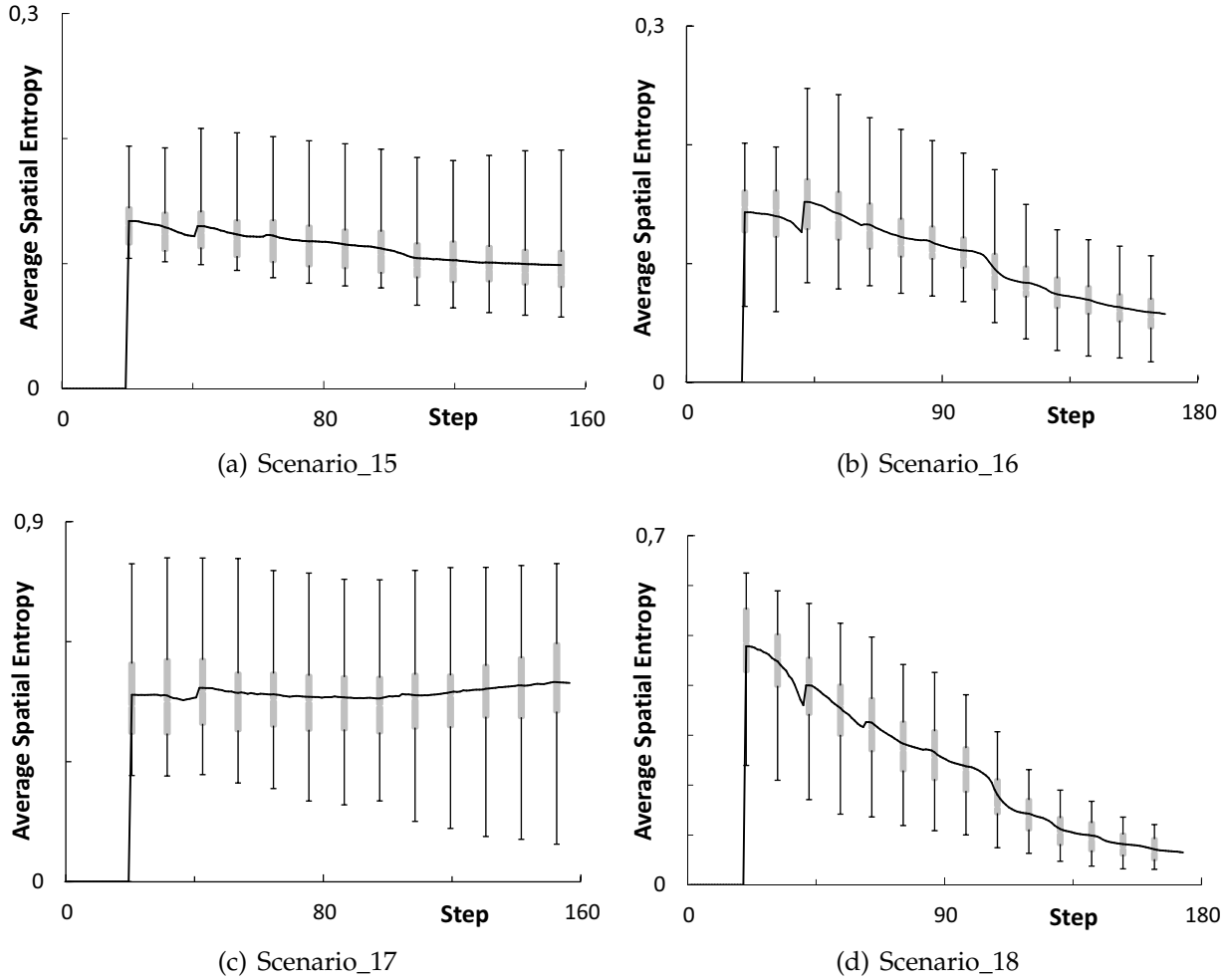
The commands were executed iteratively meaning that the next one waits until all of the previous ants have been finished with their tasks. Therefore in total there are 9 iterations that have been executed. Figure 38 shows the average spatial entropy of the system while performing the listed commands. The curves exhibit the average of 20 test runs on the simulator. The whisker chart shows the minimum and maximum and with it the range of the measured data points. The box plots - the drawn gray areas in proximity to the curve - show the lower and upper quartile containing 50 % of the data.

All graphs have a common characteristic: in the first 19 steps the entropy value is 0 and thus the curve lays on the x-axis followed by a sudden jump to a relatively high value. For $19 \leq Step \leq 20$ the curve is approximately perpendicular to the x-axis. The phenomenon occurs since the environment is initially empty ($H = 0$). The ants roaming around without dropping tuples. Finally, at $Step = 20$ they die and hence leave their carried tuple at the current location. Usually this results in bad entropic values since their orientation cannot be suitable. Nevertheless, based on their initial random walks they drop pheromones on their way. So surrounding ants are able to follow similar tuple-ants by tracking scents. At least, this behavior enables a coarse formation of similar ants.

	Scenario_15	**Scenario_16**	**Scenario_17**	**Scenario_18**
primitive	*out*	*out*	*out*	*out*
#tuples $a(X)$	–	–	–	–
#tuples $b(X)$	–	–	–	–
#tuples $c(X)$	–	–	–	–
#tuples $d(X)$	–	–	–	–
entropy	0	0	0	0
seeding	–	–	–	–
aoc	–	–	–	–
sc	–	–	–	–
num-ants	1000	1000	1000	1000
age	20	20	20	20
chosen-node	all	all	all	all
formula P_{drop}	P_{drop}^{orig}	P_{drop}^{mod}	P_{drop}^{orig}	P_{drop}^{mod}
formula H	H_{orig}	H_{orig}	H_{mod}	H_{mod}

Table 18: Configuration of the system environment for the test runs

However, Figure 38(a) and 38(b) show the comparison of P_{drop}^{orig} and P_{drop}^{mod} by measuring the entropy according to H_{orig}. At $Step \approx 20$ both curves indicate a similar entropy value. It is noticeable that the curvature in Figure 38(b) applying P_{drop}^{mod} follows a steeper decline compared to Figure 38(a). Finally, the last data point of the graph in Figure 38(b) exhibits around half the value of the data point of the plot indicated in Figure 38(a). By applying P_{drop}^{mod} the curve is decreasing constantly stronger than using P_{drop}^{orig}. This behavior exhibits an improved self-organization that is based on a more sophisticated clustering. Tuples get more attracted to tuple spaces that offer most homogeneous cluster structures as well as a



(a) Scenario_15 (b) Scenario_16

(c) Scenario_17 (d) Scenario_18

Figure 38: Development of the average entropy values during the execution of 20 test runs based on the scenarios shown in Table 18 (Figure 38(a) and 38(d) adapted from [22])

certain amount of tuples that have already been stored. On the other hand P_{drop}^{mod} generally avoids the emergence of heterogeneity within tuple spaces while P_{drop}^{orig} does not consider the actual constitution of tuple spaces causing in possible heterogeneous clusters.

Figure 38(c) and 38(d) show the comparison of P_{drop}^{orig} and P_{drop}^{mod} by measuring the entropy according to H_{mod}. At $Step \approx 20$ both curves indicate a similar entropy value. A direct comparison between the four curves plotted in the respective graphs of Figure 38 are also available in Figure 39 since the curves have been scaled equally. In order to avoid confusions one may notice that the graphs in Figure 38 indicate individual scaled axes. This is due to an improved visualization of the deviation of the different data points from the mean value. By applying whisker charts and box plots they show the entropic range as well as the deviated proportions of the entropy for the test runs.

However, it is noticeable that the curvature in Figure 38(d) applying P_{drop}^{mod} exhibits a totally different shape which shows a very steep decline of the entropy. In contrast Figure 38(c) indicates a slight decrease of the entropy followed by an increase again. Finally, the curve ends with an even higher entropic value as shows in the beginning ($Step = 20$).

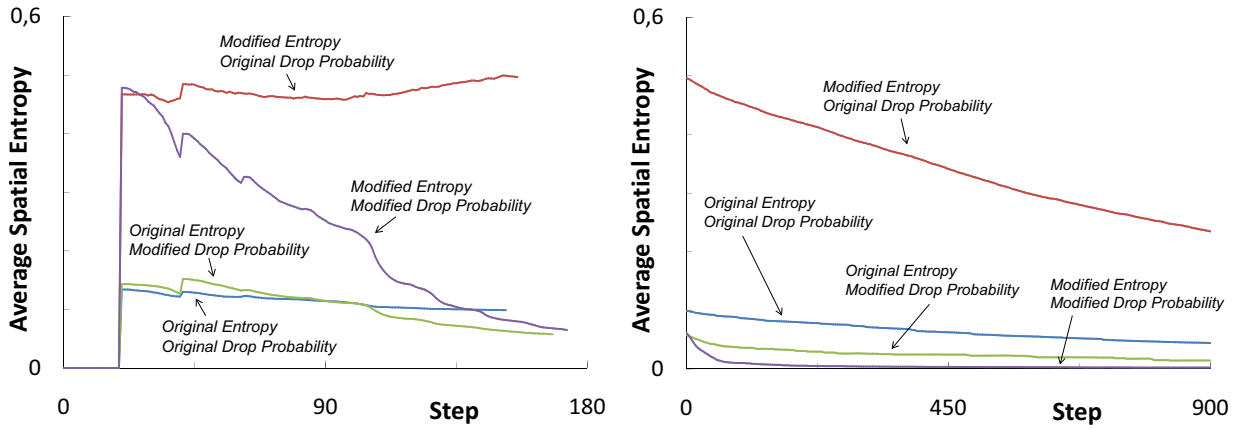
Figure 38(a) and 38(c) as well as Figure 38(b) and 38(d) can be compared since they show the different developments of the entropy applying H_{orig} indicated by the upper graphs and H_{mod} indicated by the lower ones. As described in section 4.2 (page 66) H_{mod} has been introduced since it reflects the actual spatial system entropy by weighting the different node entropies by accounting for their total amount of tuples. In contrast H_{orig} computes the uniform system entropy by treating each node equal. In fact, that causes in an even lower system entropy established by a certain amount of nodes that do not contain tuples. Since empty tuple spaces contribute an entropic node value of 0 to the system entropy it tends to adopt lower values in general. But this value is not a representative one for the global system as already mentioned and extensively explained in section 4.2 (page 66).

However, Figure 38(d) indicates a relatively bigger range between the minimum and maximum values of the whisker in the beginning because the initial distribution of tuples in the first iteration is almost randomly. But during the simulation one can see a steep decrease of the entropy, the quartiles and the range of the whisker which demonstrates a clear (self-)organization. The entropy tends to go down very fast and finally converges.

	Scenario_19	**Scenario_20**	**Scenario_21**	**Scenario_22**
primitive	*tuple movement*			
#tuples $a(X)$	250	250	250	250
#tuples $b(X)$	250	250	250	250
#tuples $c(X)$	250	250	250	250
#tuples $d(X)$	250	250	250	250
entropy	0.0988	0.0575	0.4967	0.0649
seeding	–	–	–	–
aoc	–	–	–	–
sc	–	–	–	–
num-ants	10	10	10	10
age	20	20	20	20
chosen-node	all	all	all	all
formula P_{drop}	P_{drop}^{orig}	P_{drop}^{mod}	P_{drop}^{orig}	P_{drop}^{mod}
formula H	H_{orig}	H_{orig}	H_{mod}	H_{mod}

Table 19: Configuration of the system environment for the test runs

Figure 39 compares the development of the spatial system entropies during the execution of the 20 test runs. Observing the lower two curves indicating the original and modified drop probability evaluated by the original entropy one can see the approximately same behavior of the curve in the beginning. At $Step \approx 90$ both curves intersect due to an equal entropic value. In the following the curve exhibiting the modified drop probability



(a) Scenarios according to Table 18

(b) Scenarios according to Table 19

Figure 39: Comparison of the average entropy values during the execution of 20 test runs based on the scenarios shown in Table 18 and Table 19 (adapted from [22])

remains continuously beneath the original drop probability curve. The entropy of the last data point shows around half the value.

Comparing the curves representing the original and modified drop probability by applying the modified entropy it is easy to notice that the curves behave totally different. While the graph based on the original drop probability rises a little in total the curve following the modified drop probability declines strongly, intersects the curve based on P_{drop}^{orig} and H_{orig} and finally approaches the curve calculated by P_{drop}^{mod} and H_{orig}.

It is noticeable that the system takes a little longer while distributing the tuples among the nodes by applying the modified drop probability. But if one draws a line perpendicular to the x-axis at the last common data point ($Step \approx 160$) it is obvious that this does not affect the entropy. The level of organization remains unchanged.

Figure 39(b) shows the development of the average entropy based on 20 test runs according to Table 19. The graph shows the effect of redistributing the amount of tuples in the environment while performing tuple movement. Each curve type is directly connected to the curve of the graph from Figure 39(a) since the test runs according to Table 19 have been executed directly after the test runs according to Table 18.

The curve representing the modified drop probability and entropy requires around 150 steps in order to achieve an entropic value that approximates 0. Thus the system claims full organization. As described in section 5.3 (page 90) the tuple movement takes place in phase three that is characterized by an almost organized system. The absolute value of the negative slope decreases and finally converges.

The curves applying P_{drop}^{mod} and H_{orig} as well as P_{drop}^{orig} and H_{orig} show an almost similar behavior. They are declining slowly but constantly towards 0. But as one may notice in the beginning $Step < 150$ the curve representing P_{drop}^{mod} and H_{orig} shows a similar behavior as the curve exhibiting P_{drop}^{mod} and H_{mod}. The curve also shows the end of phase three of

tuple movement except that the slope is not as steep as applying P_{drop}^{mod} and H_{mod}.

Based on achieving an organized state by applying P_{drop}^{mod} and H_{mod} at $Step \approx 100$ the system stays calm for the further 800 steps. In contrast to the other approaches the system saves resources. This minimizes communication overhead in networks and bandwidth. In real systems *out*- and *in*-commands as well as tuple movement run concurrently so that the system is well organized after a short time and can handle new commands easily resulting in an improved robustness. Applying P_{drop}^{orig} and H_{mod} (since H_{mod} is a more realistic evaluation metric and indicates the actual level of organization) it will take a long time forcing the entropy to go down. Even the 900 steps are not sufficient enough since the entropy ends with a value around 0.23. One the other hand, one can see the effectiveness of tuple movement resulting in a new distribution of tuples among the nodes no matter how the entropy is calculated.

5.5. Anti-Overclustering and Spatial Clustering

The previous section discusses and compares the original and modified drop probability as well as the entropy calculation. It shows and gives reasons why P_{drop}^{mod} results in an improved homogeneity of cluster structures while H_{mod} rates a given scenario in a more realistic way than its counterpart H_{orig}.

However, this section deals with avoidance of overclustering and distribution of tuples in order to form spatial cluster in networks. The idea is to effectively use the given system resources by causing the system to fairly balance the amount of tuples among the nodes. Although P_{drop}^{mod} achieves good qualities by aiming homogeneous cluster structures it does not distribute the tuples equally across the nodes resulting in spatial clusters. As postulated in section 4.1 (page 59) the given amount of tuples shall be distributed equally in the system. This includes that there shall be neither peaks indicating a huge amount of tuples within single tuple spaces nor many unused nodes that provide free system resources but are not involved since they obtain no tuples. The following scenarios show test runs that are exclusively based on the formulas given by:

- $P_{drop}^{aoc^*}$ for computing the drop probability with *aoc* and *sc* (Equation 30, page 78)
- H_{mod} for computing the spatial network entropy (Equation 18, page 68)
- P_{pickup}^{norm} for computing the pickup probability (Equation 22, page 72)
- P_{ij} for computing the path selection (Equation 5, page 25)

Table 20 shows the configuration of scenarios that indicate the manual interaction with the system while it is running. Thus the scenarios are linked. Figure 40 (page 100) exhibits the distribution of tuples among the nodes and hence the constitution of tuple spaces as a result of test runs applying the given scenario configuration. Since the objective is to

	Scenario_23	**Scenario_24**	**Scenario_25**	**Scenario_26**
primitive	*seeding*	*out*	*out*	*tuple movement*
#tuples $a(X)$	0	40	190	240
#tuples $b(X)$	0	40	190	240
#tuples $c(X)$	0	40	190	240
#tuples $d(X)$	0	40	190	240
entropy	0	0	0.0539	0.0709
aoc	according to $P_{drop}^{aoc^*}$			
sc	according to $P_{drop}^{sc^*}$			
max-size	50	50	50	50
num-ants	160	600	200	10
age	20	20	20	20
chosen-node	E, I, P, R	all	all	all

Table 20: Configuration of the system environment for the test runs

avoid over-clustering the parameter *max-size*[23] is required in order to adjust the threshold as an orientation for tuple-ants.

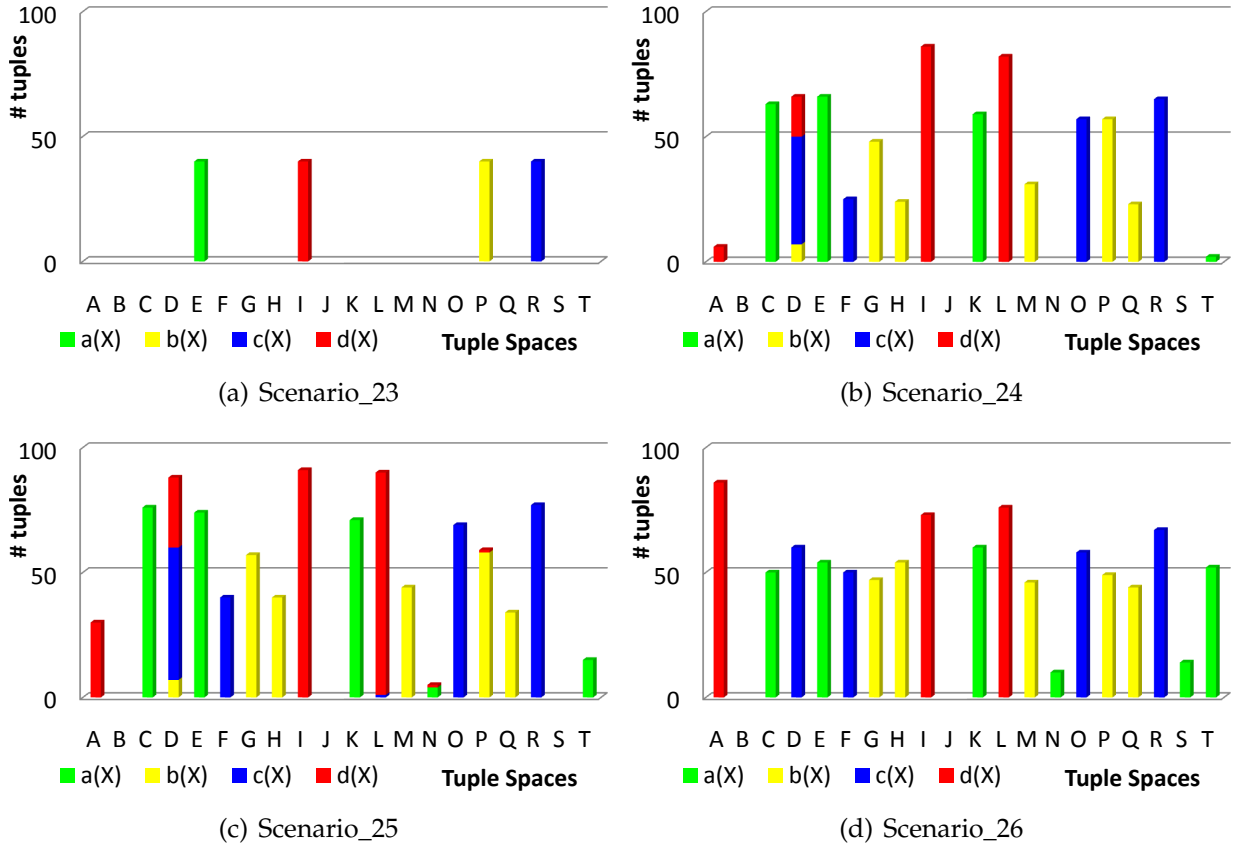
Scenario_23 describes the seeding of 160 tuples in an empty network on the nodes given by *chosen-node*. As an orientation one can see the result of seeding in Figure 40(a) based on the node identifier. The geographical arrangement of nodes and links and the final system state after the test runs given by Table 20 is shown in Figure 43(b) (page 105). The different colored nodes indicate the respective type of cluster.

The idea of *Scenario_23* is to observe the system behavior by setting those seeds on particular nodes in the network. The seeds are supposed to attract similar tuples and force them to move towards their locations. Therefore, it shall facilitate the system to form spatial clusters surrounding the seeded nodes.

Scenario_24 performs an *out* command with 150 tuples of each type, in total 600 tuples. The result is shown in Figure 40(b). By looking at the development of the distribution of $a(X)$ tuples indicated by the green columns one may notice that TS C, E and K are the main nodes containing $a(X)$ tuples. They are almost of equal size. Since E was the seeded node with 40 tuples it is noticeable that most of the tuple-ants dropped their tuple at C resp. K. Since *max-size* is set to 50 the ants avoid overclustering E. On the other hand if one looks at Figure 43(b) it is obvious that the mentioned three tuple spaces form a geometric triangle. Therefore the drop probabilities get reinforced among themselves since $P_{drop}^{sc^*}$ is based on spatial clustering and claims higher values if the connected neighborhood contains many similar tuples while indicating less dissimilar ones causing in a high relative frequency.

Since K does not have any neighbor holding dissimilar tuples the drop probability claims the maximal value but is, of course, still dependent on the age factor given by the ratio $\frac{K}{K_{fix}}$. C also reaches a relatively high value since it has E and K as neighbors

[23] defines the preferable cluster size that has been discussed in section 4.4 (page 75)



(a) Scenario_23

(b) Scenario_24

(c) Scenario_25

(d) Scenario_26

Figure 40: Distribution of tuples among the nodes as a result of the scenarios defined in Table 20

while A contains less dissimilar tuples and B is empty. Thus B does not affect the drop probability. Only D exhibits a high amount of dissimilar tuples. Finally, T also owns a few $a(X)$ tuples since it is the neighbor of E while S is empty.

The second seeded node is I holding $d(X)$ tuples. It is noticeable that the respective tuple-ants seems to have problems during the distribution. In fact, L was occupied in an early state and therefore tuples got spread over L and I forming a spatial cluster. Since D would complete a geometric triangle tuple-ants tend to adopt the tuple space. But by looking at its constitution tuple-ants carrying tuples of type $b(X)$ and $c(X)$ also try to conquer the node. Therefore tuple-ants of type $d(X)$ may not successful at D. Since the drop probability for L and I keeps on decreasing the ants tend to explore its neighborhood in order to look for suitable tuple spaces. Finally A was chosen - maybe more by accident - but, in fact, was empty. Thus the ants tend to adopt the node.

P is the third seeded tuple space holding tuples of type $b(X)$ and grows approximately to the chosen value represented by *max-size*. Although M and Q do not form a triangle they, nevertheless, stay in proximity to P and hence indicate a spatial cluster. H and G form an additional spatial cluster that is isolated from M, P and Q. However, since D would be required to complete a geometric triangle the tuple-ants try to obtain it. The

	Scenario_27	**Scenario_28**	**Scenario_29**	**Scenario_30**
primitive	*seeding*	*out*	*out*	*out*
#tuples $a(X)$	0	60	100	180
#tuples $b(X)$	0	60	100	144
#tuples $c(X)$	0	60	100	141
#tuples $d(X)$	0	60	100	180
entropy	0	0	0	0.0052
aoc	according to $P_{drop}^{aoc^*}$			
sc	according to $P_{drop}^{sc^*}$			
max-size	30	30	30	30
num-ants	240	160	320	200
age	20	20	20	20
chosen-node	$\forall n \in V$	all	all	all

Table 21: Configuration of the system environment for the test runs

result is analogous to the attempt of ants carrying $d(X)$ tuples.

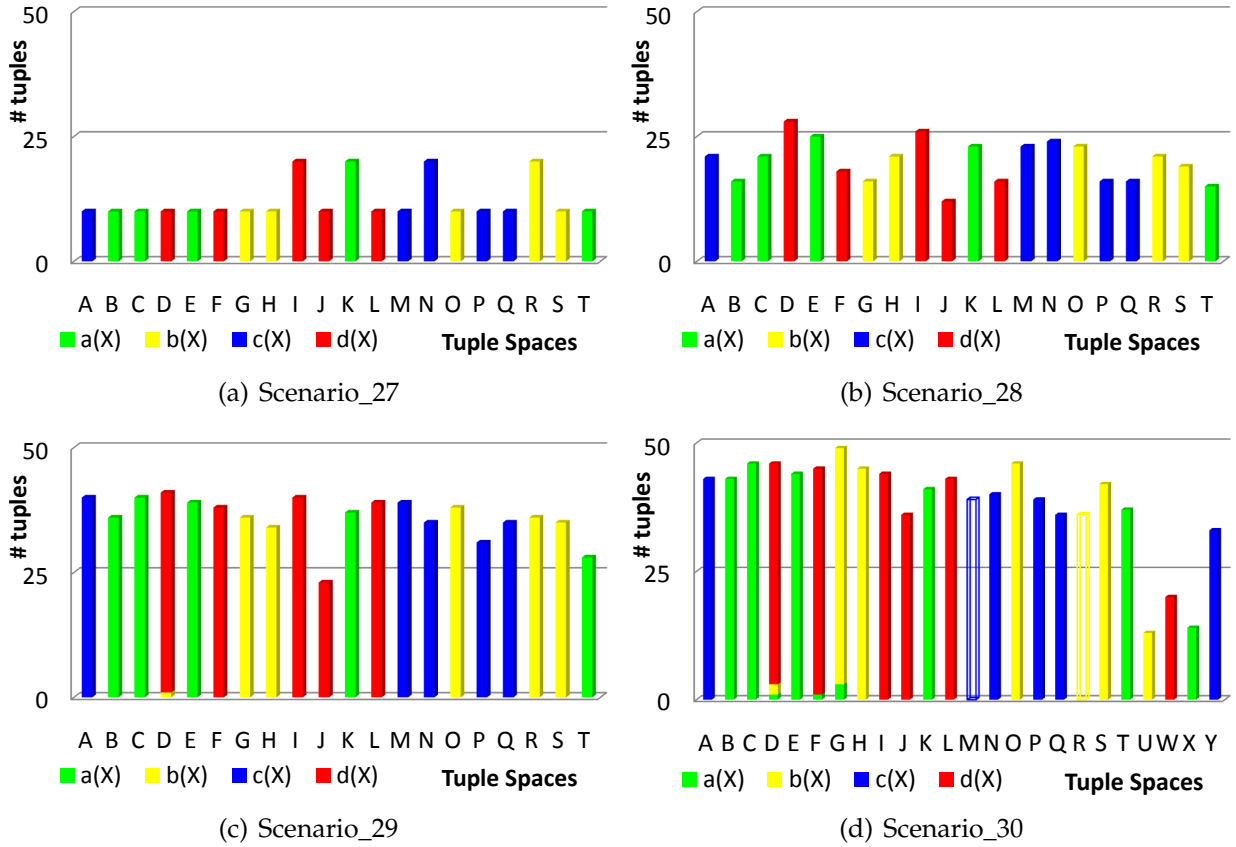
Finally, R is the fourth seeded tuple space owning tuples of type $c(X)$. It is surrounded by D and O indicating a geometric triangle. Additionally F is also occupied by $c(X)$ tuples and thus the four tuple spaces form a spatial cluster. However, the entropy was 0 before the execution of the scenario and finally increases to 0.0539 which is quite low and hence indicates a good clustering.

Scenario_25 shows again an *out* command but the amount of tuples is reduced to 200. The results of the distribution of the tuples are represented in Figure 40(c). It is noticeable that C, E and K holding $a(X)$ tuples grow to a limited size and finally the ants looking for additional tuple spaces in order to store the tuples. Since T is appropriate because it stays in the neighborhood of E it gains tuples. One may notice a small amount of tuples at N. This occurs more by accident based on ants that have been instantiated in that area and could not find suitable path to the upper location of the network graph.

The situation for $b(X)$ tuples is almost the same except that the occupied tuple spaces grow according to the size of the similar *out*-ants. Compared to the distribution indicated by Figure 40(b) the current one tends to be more equal based on the size of tuple spaces. The situation for $c(X)$ tuples is also very similar to the one indicated by Figure 40(b). Finally, $d(X)$ tuples got more attracted by A and D since I and L already contain plenty of tuples and thus may get overclustered. The entropy raises to a value of 0.0709 which is, in fact, higher than in *Scenario_24* but still exhibits a well organized state.

Scenario_26 performs tuple movement with 10 ants to the given environment. Figure 40(d) shows the result of tuple movement. The entropy falls back to 0 and thus indicates complete organization. The final distribution is also shown in Figure 43(b) (page 105) which also tells the spatial locations of the tuples spaces. It is noticeable that the average size of the tuple space approximates each other so that no tuple space is overclustered. In particular all tuple spaces indicate complete homogeneous constitutions.

Table 21 describes the configuration of further scenarios. In contrast to the previous test



(a) Scenario_27

(b) Scenario_28

(c) Scenario_29

(d) Scenario_30

Figure 41: Distribution of tuples among the nodes as a result of the scenarios defined in Table 21

runs *max-size* is reduced to 30 while the seeding is increased to 240 seeds that get placed on every node in the network (V denotes the set of vertices, *Scenario_27*). The distribution is shown in Figure 41(a). The geographic arrangement is exhibited in Figure 43(c) (page 105). The seeding already involves the equal formation of spatial cluster. There is one centroid of each cluster indicating twice the amount of tuples as the surrounding tuple spaces.

Scenario_28 is characterized by an *out* with 160 ants. The distribution of tuples shown in Figure 41(b) exhibits an almost equal distribution of the tuples to the respective tuple spaces. The result is an entropy value of 0 which indicates a perfect organization. Further on, no tuple space is overclustered and the average size is approximately equal.

In the following *Scenario_29* doubles the amount of *out*-ants. The result is shown in Figure 41(c) as well as in Figure 43(c) (page 105). The clustering is again characterized by an approximately uniform distribution, there is no violation of the cluster size. Only the entropy increases a little to 0.0047 due to a rising drop probability at D for $b(X)$ tuples since this tuple space is surrounded by four nodes containing $b(X)$ tuples. Therefore one can recapitulate the situation that based on a perfect distribution in the beginning given by the small seeding which already forms equal sized spatial clusters the result is also

	Scenario_31	**Scenario_32**	**Scenario_33**
primitive	*out*	*out*	*out*
#tuples $a(X)$	0	200	260
#tuples $b(X)$	0	200	260
#tuples $c(X)$	0	200	260
#tuples $d(X)$	0	200	260
entropy	0	0.1272	0.1501
aoc	according to $P_{drop}^{aoc^*}$		
sc	according to $P_{drop}^{sc^*}$		
max-size	40	40	40
num-ants	800	240	10
age	20	20	20
chosen-node	all	all	all

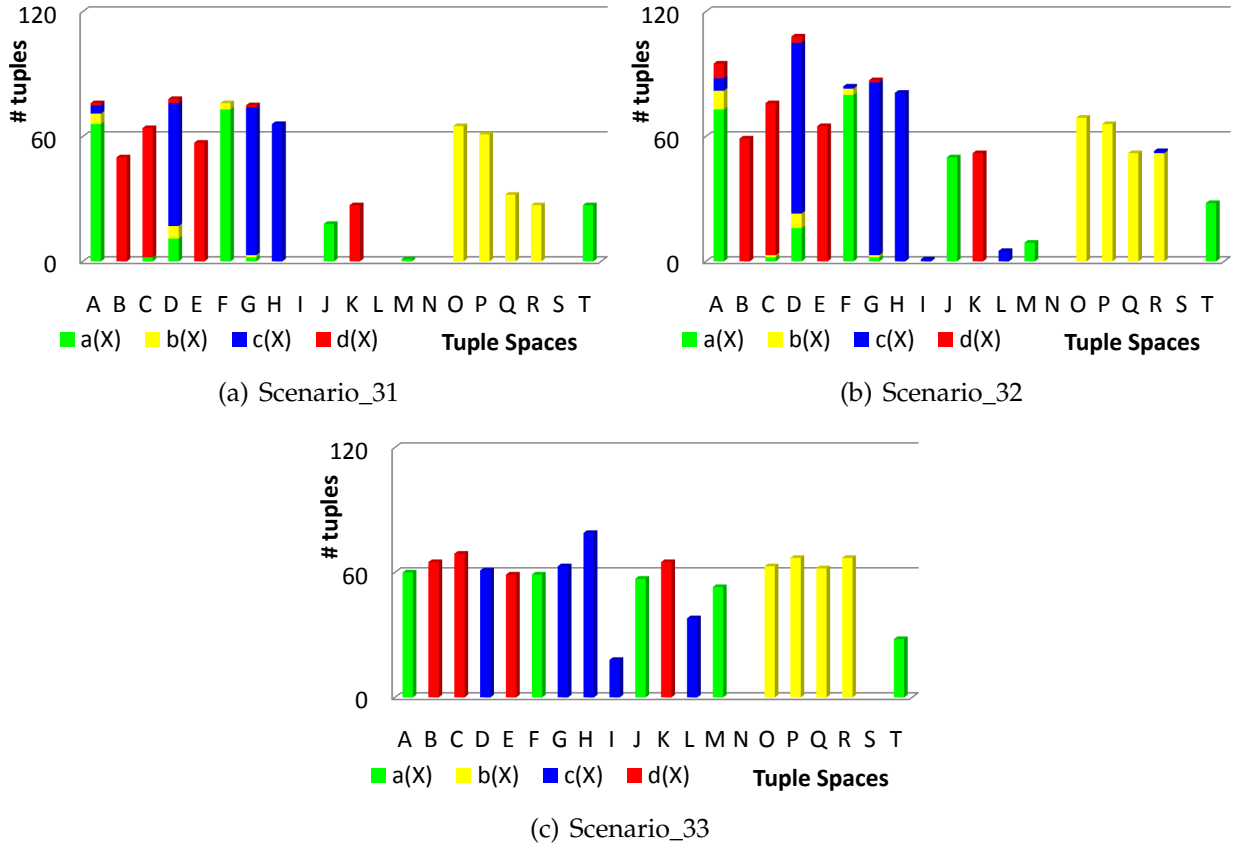
Table 22: Configuration of the system environment for the test runs

defined by a perfect organization.

In order to simulate the removal as well as addition of nodes on the fly M and R have been shut down while there have four additional nodes been created. The modification of the topology is presented in Figure 43(d) (page 105). U has been placed near to the old position of R. W stays in the proximity of L and I. X is located between B and K and finally Y has been placed between N and Q. The removal of tuple spaces occur in a loss of some tuples of type $b(X)$ and $c(X)$ that had been stored at M resp. R. As a result the entropy is also affected and increases marginally to 0.0052. The disappearance of tuples is also listed in *Scenario_30*. Given the new topology an *out* is performed with 200 ants. The idea of this test run is to observe how the system react due to a loss of two tuple spaces and its connections as well as the addition of the four nodes. The result is listed in Figure 41(d) showing the new distribution of tuples. M and R are shown as transparent columns bordered by the respective color. This indicates that these tuple spaces do not exist anymore but they express the relation of how much tuple have been lost in contrast to the surrounding nodes. However, the scenario shows that the new added nodes get occupied by different tuple-ants and thus are integrated in the respective spatial cluster. In fact, most of the tuples got clustered among the new tuple spaces in order not to violate the overclustering rule. As one may see in the column diagram some tuples, nevertheless, got stored on tuple spaces that result in an increase of the entropy which adopts a value of 0.06 in the end. The final distribution is given in Figure 43(d) (page 105).

Table 22 describes the configuration of further scenarios. In contrast to the previous test runs *max-size* is set to 40 and the seeding is turned off in order to observe the behavior of clustering without giving the system any external support.

Scenario_31 shows an amount of *out*-ants which is set to 800. Since the scenario should be a little more closer to real-world scenario the amount is not instantiated at once. At eight different points in time there have been 100 ants instantiated each. The result of the clustering is shown in Figure 42(a). Although there has been no seeding performed the



(a) Scenario_31

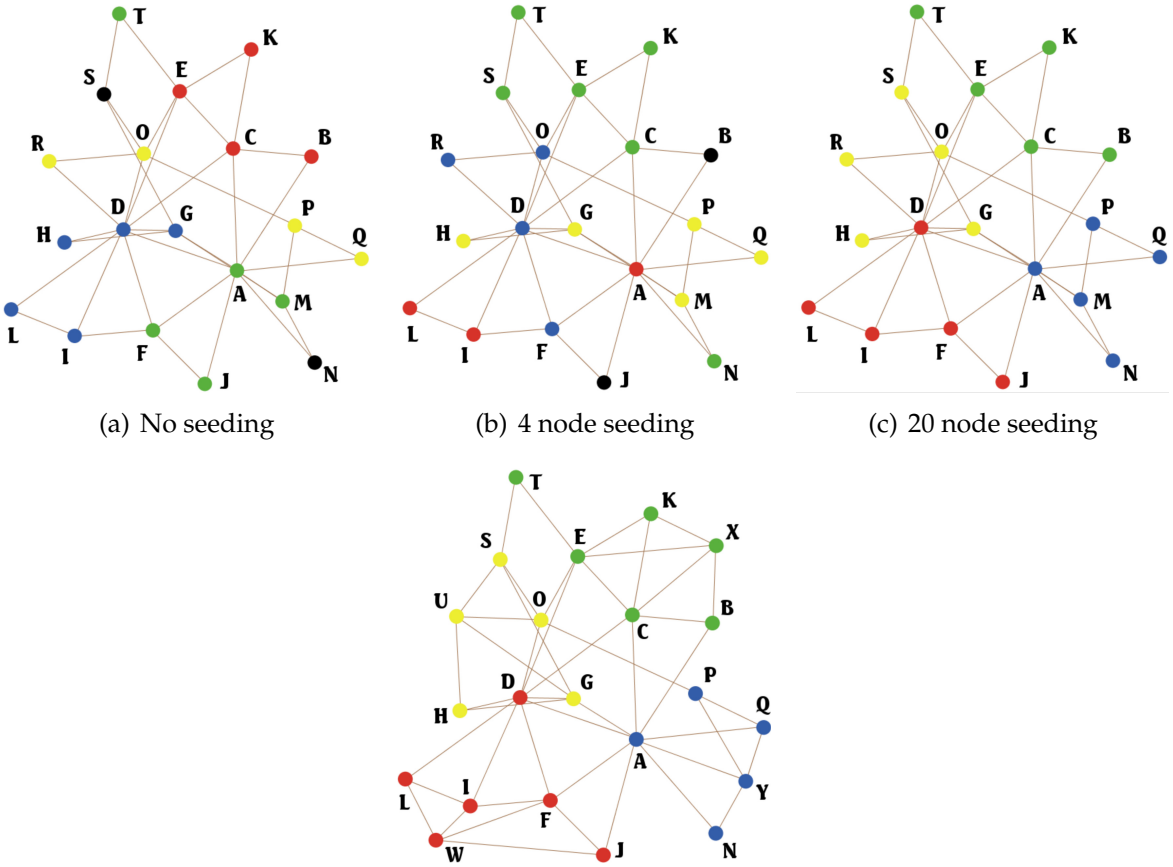
(b) Scenario_32

(c) Scenario_33

Figure 42: Distribution of tuples among the nodes as a result of the scenarios defined in Table 22

formation as well as the homogeneity of spatial clusters have been developed well. First, $a(X)$ tuples have formed a spatial cluster based on A, F, J and M. There is one single node cluster that have been emerged at T and is thus isolated. Further on, $b(X)$ tuples have occupied the tuple spaces O, P, Q and R and hence forming one spatial cluster. It divides the network graph into two separate parts. In the following, $c(X)$ tuples claim D, H as well as G while $d(X)$ tuples capture B, C, E as well as K and form spatial clusters respectively. The entropy is with 0.1272 higher than in the previous scenarios but based on an initial empty network the clustering is quite good. In general the entropic value exhibits a well organized state.

Scenario_32 results in a more balanced situation which is shown in Figure 42(b). Especially, $b(X)$, $c(X)$ and $d(X)$ tuples exhibit an approximately equal distribution. Thus they avoid overclustering. Further on, it is interesting that $c(X)$ tuples adopt I as well as L in order to extend the spatial cluster. The respective ants get forced to behave like that since three tuple space are insufficient by not violating the overclustering rule. Therefore they expand the cluster to five nodes that are coherent. The remaining tuple types do not capture any more tuple spaces since they already obtain at least four tuple spaces and thus it is sufficient in order to distribute the tuple without violating the restriction of



(a) No seeding (b) 4 node seeding (c) 20 node seeding

(d) Topology modification

Figure 43: Network graphs that exhibit the distribution of tuples among the nodes and thus indicate the spatial clusters. Additionally they show the topology in which the scenarios have been executed.

overclustering.

Scenario_33 shows the behavior of tuple movement applied to the environment that has been generated after *Scenario_32*. The final distribution is shown in Figure 42(c) as well as in Figure 43(a) (page 105). The tuple movement is performed with ten cleaning-ants at an entropic level of 0.1501. Finally, the tuple movement pushes the entropy down to 0 and thus the environment is totally organized. The distribution of tuples is again approximately equal and hence avoids overclustering. The average spatial cluster size - that defines the number of nodes contained in a spatial cluster - is 4.5.

6. Conclusion and Future Work

As a recapitulation the report introduces SWARMLINDA as an extension for conventional LINDA systems since they suffer from their architecture and cannot cope with openness and scalability as well as adaptiveness, flexibility and mobility. Section 1 (page 5) introduces the field of interest and reflects common properties of LINDA systems exhibiting the inability of growing to enormous size. In the following SWARMLINDA based on decentralized, autonomous agents involving swarm intelligence which is a particular part of artificial intelligence represents a solution applied to scale-free networks.

Thereupon, section 2 (page 18) describes the basic idea and mechanisms of SWARMLINDA as well as the respective algorithms of which the system consists of. In particular, it presents tuple distribution, tuple retrieval as well as tuple movement as the main mechanisms that perform the processes of storing and finding information objects in the environment. Finally, tuple movement supports the system to maintain a balanced state causing in effectively using its resources.

The development and implementation details are given in section 3 (page 35). It explains the simulator for network generation as well as the SWARMLINDA simulator. Further on, the system architecture is presented as well as the used technologies, applied plug-ins and OTS[24] software.

Section 4 (page 59) defines metrics that are included in the respective algorithms for distributing, retrieving and migrating tuples as well as evaluating the performance of the state of the system environment. It postulates specific behaviors like homogeneous spatial clusters. The section also discusses and proposes improved metrics making the system more effective. It also gives evidence for the improved behavior. The final contribution is to avoid big tuple spaces by tending to balance the size of tuple spaces in the network. On the other hand the network shall be partitioned into different spatial regions that form a cluster and thus holding similar tuples. This results in a more fault tolerant as well as improved load balanced system since the resources are used equally.

Finally, section 5 (page 80) performs several test runs on the simulator using the proposed metrics and evaluates as well as interprets the results. All introduced mechanisms are evaluated and presented.

The report shows that the idea of SWARMLINDA based on decentralized autonomous agents that operate in the environment performing different tasks, in fact, behaves as proposed. Although each agent is completely independent on others and is only processing a simple set of routines the total amount exhibits an impressive collective behavior. The test runs clarify the effectiveness of the system in dependence on its degree of training.

The system is totally independent on external influences and does not postulate any manual instructions of getting started. Beside the initiation of the system there is also no interaction required in the prospective executions. Based on its self-organization which has been presented in the test runs the amount of information objects get automatically assigned to tuple spaces. The result of the distribution is characterized by a division of the domain in separate parts forming spatial regions that contain similar tuple types. The

[24] off-the-shelf

constitution of a region indicates high homogeneity while the partitions among themselves exhibit heterogeneous structures.

Therefore the system is able based on an initial empty state to distribute and organize a base set of information objects in the network analogous to brood sorting used by ants. The emergence of spatial clusters and reinforcement of pheromone trails result in an improved routing of ants from their current start locations to their individual destinations. In addition, the retrieval process gets improved as well since the ants also track the scent marking trails. Although, based on SWARMLINDA's non-determinism, one cannot predict in advance the exact distribution of tuples among the nodes. But one can notice the phenomenon of formation of spatial clusters.

In order to support the system to maintain its organization the requirement of tuple movement has been pointed out. By redistributing some misplaced tuples assure the degree of order. Further it forces the environment that similar types stay at the same location. This results in an easier way of continuing the clustering as well as retrieving the tuples.

The system is also able to cope with the postulated issues:

Scalability The system is scalable since the addition of new tuple spaces does not impact the system in order to reorganize itself like as required in LINDA which is based on hashing and thus requires a rehashing of its content. The decentralized and autonomous behavior of SWARMLINDA agents result in an exploration of the new added terrain followed by the emergence of new spatial clusters in that environment. Of course, if a huge amount of new tuple space is added to the system it will not redistribute its content in order to form bigger spatial clusters. It is more likely depending on the concrete topology that more than one spatial cluster of a type will be created. The tests in this report comprises a small number of nodes been added to the environment. As a reaction the system integrates the new nodes and expands the current spatial clusters involving the new nodes.

Adaptiveness A SWARMLINDA system is also characterized by an adaptive behavior since it reacts of removal or addition of nodes and modifications of the environment quickly and without suffering under disorientation. The tests have shown that a removal as well as an addition causes in integration of the new tuple spaces without noticing any severe impact. Of course, each system has its limitations: if one shuts down as much hub nodes resulting in an incoherent network graph, some nodes get isolated. This inevitably leads to an unavailability of the information stored there. But this is no big phenomenon if one thinks for instance of the arguably most well known scenario: the World Wide Web. Since the topology also follows a scale-free power-law distribution - which is the base of the executed test scenarios in this report - it is, however, fault tolerant but also vulnerable. If backbone server disappear from the network - caused by (Distributed) Denial of Service ((D)DoS) attacks - it will cause in an impact and may isolate several subordinated nodes.

Fault tolerance A SWARMLINDA system also copes with fault tolerance. As mentioned the removal or unavailability of tuple spaces do not cause the system of suffering under a severe impact. The basic mechanisms are not affected by node failures. In

fact, if a spatial region holding a specific tuple type gets isolated from the network and in a scenario that a requester wants to obtain such a tuple the template-ants may not find any.

This report contributes a sufficient amount of different tests evaluating the proposed behavior of SWARMLINDA. Further on, it presents improved metrics resulting in a more effective system. It shows the development of different metrics in order to cluster tuples on nodes and in spatial regions with respect to keeping homogeneity. It also introduces an evaluation metric defined by the spatial entropy that has been extended from Casadei *et al.* Further on, it shows mechanisms to equally distribute the amount of tuples among the nodes and hence avoid overclustering and establish a well balanced system. Finally, it presents a metric for the routing of ants.

Nevertheless, there are still plenty of issues that are worth to be researched. An inspiration for future works is given as follows:

Dynamic node failures In order to be a little closer to a real-world scenario it is suitable to integrate a dynamic node failure system. As a configuration one can adjust the percentage of failures (1 %, 5 %, ...) in the network. During the runtime of the system some nodes according to the configuration disappear for a certain amount of time and then reappear. Therefore, the affected amount of tuples is not lost. They are only not available at the specific point in time. Thereupon, it may of interest to perform different test scenarios based on different configurations of dynamic node failures. Further on, it may appropriate to find out at which degree of node failure the system suffers under an over proportional - meaning very high - impact. This may an interesting contribution to the limitation of fault tolerance.

Connection latency Again, in order to be a little closer to a real-world scenario it is appropriate to assign links a cost value that reflects the network latency between nodes. Therefore, the spatial clustering may not result in an emergence of regions comprising servers that stay in geographical proximity but rather tend to form regions based on shortest latency.

Physical extension Since the presented simulator is based on threads that although perform code concurrently they are not distributed among several machines. Therefore it may interesting to extend the simulation based on local running threads to real distributed processes that run on several machines. It is possible to integrate this mechanism in the current simulator. The simulator itself - in particular, the visualization area - runs on one machine in order to observe the system behavior. Each node which is shown in the simulator represents a real machine connected in a real network. Thus on each machine there is a specific application running that accepts communication on a specific port. Thereupon, ants are really roaming between the separate machines by traversing the physical network. This would show a little more realistic results. The aforementioned extension ideas can be included in this approach and thus are very close to a real-world scenario. In order to measure

system performance of the separate machines it is suitable to use JConsole[25] which comes along with JDK 1.5. "It uses the extensive JMX[26] instrumentation of the Java virtual machine to provide information on performance and resource consumption of applications running on the Java platform" [50].

[25]JConsole is a JMX-compliant monitoring tool
[26]Java Management Extensions

References

[1] B. Anderson and D. Shasha. Persistent linda: Linda + transactions + query processing. In J.P. Banatre and D. Le Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 57 in LNCS, pages 93–109. Springer, 1991. http://citeseer.ist.psu.edu/anderson91persistent.html.

[2] Apache. The Apache Ant Project, visited: January 28, 2008. http://ant.apache.org/index.html.

[3] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. Science, 286:509–512, October 1999.

[4] Robert Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University Department of Computer Science, 1992. Technical Report 931.

[5] E. Bonabeau and G. Theraulaz. Swarm intelligence: From natural to artificial systems. Oxford Press, 1999.

[6] Nadia Busi, Alberto Montresor, and Gianluigi Zavattaro. Data-driven coordination in peer-to-peer information systems. *International Journal of Cooperative Information Systems*, 13(1):63–89, March 2004.

[7] Kenneth L. Calvert and Michael J. Donahoo. *TCP/IP Sockets in Java. Practical Guide for Programmers.* Morgan Kaufmann, 2001. ISBN:1558606858.

[8] Scott Camazine, Nigel Franks, James Sneyd, Eric Bonabeau, Jean-Louis Deneubourg, and Guy Theraula. *Self-Organization in Biological Systems*. Princeton University Press, Princeton, NJ, USA, 2001. ISBN:0691012113.

[9] Matteo Casadei, Luca Gardelli, and Mirko Viroli. Simulating emergent properties of coordination in maude: the collective sort case. *Electron. Notes Theor. Computer Science*, 175(2):59–80, 2007. Elsevier Science Publishers B. V., Amsterdam, The Netherlands. ISSN:1571-0661. http://dx.doi.org/10.1016/j.entcs.2007.05.022.

[10] Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. Self-organized over-clustering avoidance in tuple-space systems. 2007.

[11] Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. A self-organizing approach to tuple distribution in large-scale tuple-space systems. 2007.

[12] Ahmed Charles, Ronaldo Menezes, and Robert Tolksdorf. On the implementation of swarmlinda. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 297–298, New York, NY, USA, 2004. ACM Press. http://portal.acm.org/citation.cfm?id=986607.

[13] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language, 2007. http://www.w3.org/TR/wsdl20/.

[14] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Strategies and protocols for highly parallel linda servers. *Softw. Pract. Exper.*, 28(14):1493–1517, 1998. John Wiley & Sons, Inc., New York, NY, USA. ISSN:0038-0644.

[15] J.-L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien. The dynamic of collective sorting robot-like ants and ant-like robots. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pages 356–365, Cambridge, MA, 1991. MIT Press.

[16] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernatics*, 26(1):29–41, 1996.

[17] Apache Software Foundation. Logging services, visited: February 3, 2008. http://logging.apache.org/log4j/.

[18] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, 1999. ISBN:0201309556.

[19] O. Galibert. YLC, A C++ Linda system on top of PVM. *Lecture Notes in Computer Science*, 1332, 1997.

[20] David Gelernter. Generative communication in Linda. *ACM transactions in Programming Languages and Systems*, 7(1):80–112, 1985. http://doi.acm.org/10.1145/2363.2433.

[21] David Gelernter. Multiple tuple spaces in linda. In *Proceedings of the Parallel Architectures and Languages Europe*, volume II: Parallel Languages, pages 20–27, London, UK, 1989. Springer-Verlag. ISBN:3-540-51285-3.

[22] Daniel Graff, Ronaldo Menezes, and Robert Tolksdorf. On the performance of swarm-based tuple organization in linda systems. 2007.

[23] J. Guare. *Six Degrees of Separation: A play*. Vintage Books, New York, 1990.

[24] Oliver Haase. *Kommunikation in verteilten Anwendungen. Einführung in Sockets, Java RMI, CORBA und Jini.* Oldenbourg, 2001. ISBN:3486257382.

[25] J. Kennedy and R. C. Eberhart. Swarm intelligence. Morgan Kaufmann, 2001.

[26] C. Koch and G. Laurent. Complexity and the nervous system. *Science*, 284(5411):96–98, April 1999.

[27] G. T. Ltd. Gigaspaces platform, 2002. http://www.gigaspaces.com.

[28] Ingo Melzer. *Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis*. Spektrum Akademischer Verlag, 2007. ISBN:3827418852.

[29] Ronaldo Menezes. Ligia: Incorporating garbage collection in a Java based Linda-like run-time system. In Raimundo J. Macedo, Alcides Calsavara, and Robert C. Burnett, editors, *Proc. of the 2nd Workshop on Distributed Systems (WOSID'98)*, pages 81–88, Curitiba, Paraná, Brazil, 1998. `http://citeseer.ist.psu.edu/menezes98ligia.html`.

[30] Ronaldo Menezes and Robert Tolksdorf. Adaptiveness in linda-based coordination models. In *Proc. of the 1st International Workshop on Engineering Self-Organising Applications*, Melbourne, Australia, 2003.

[31] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable linda-systems based on swarms. Technical report, Florida Institute of Technology, Melbourne, Florida, 2003.

[32] Ronaldo Menezes and Alan Wood. Garbage collection in Linda using tuple monitoring and process registration. In *Proc. of the 10th International Conference on Parallel and Distributed Computing and Systems*, pages 490–495, Las Vegas, Nevada, USA, 1998. Acta Press. `http://citeseer.ist.psu.edu/menezes98garbage.html`.

[33] Ronaldo Menezes and Alan Wood. Distributed Garbage Collection of Tuple Space in Open Linda Coordination Systems. In *Proc. of the 14th International Symposium on Computer and Information Sciences*, pages 957–965, Kusadasi, Turkey, 1999. `http://citeseer.ist.psu.edu/menezes99distributed.html`.

[34] S. Milgram. The small world. *Psychol. Today 2, 60*, 1967. Ablex, Norwood, NJ.

[35] NetLogo. User manual, Decemer 5, 2007. `http://ccl.northwestern.edu/netlogo/faq.html`.

[36] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. ISSN:0098-5589.

[37] P. Obreiter and G. Gräf. Towards scalability in tuple spaces. In *Proceedings of the 2002 Symposium on Applied Computing*, pages 344–350, 2002.

[38] Members of the Clever project. Hypersearching the web. *Scientific American*, 280(6), June 1999.

[39] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.

[40] J. Pinakis and C. McDonald. The Inclusion of Linda Tuple Space Operations in a Pascal-based Concurrent Language. In J. Gupta and J. Lions, editors, *Proceedings 14th Australian Comp. Science Conf.*, Kensington, Australia, 1991.

[41] Mitchel Resnick. Turtles, termites, and traffic jams. MIT Press, 1994.

[42] A. Rowstron. WCL: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.

[43] A. Rowstron and A. Wood. Bonita: a set of tuple spaces primitives for distributed co-ordination. In *Proc. HICSS30, Sw Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[44] Antony Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based coordination languages. In P. Ciancarini and P. Wolf, editors, *Coordiantion Languages and Models (Coordination '99)*, pages 196–210. Springer Verlag, 1999.

[45] Sourceforge. Eclipse Checkstyle Plug-in, visited: January 27, 2008. `http://eclipse-cs.sourceforge.net/`.

[46] Sourceforge.net. Lumbermill - log4j/jsr47 gui, visited: February 3, 2008. `http://sourceforge.net/projects/lumbermill/`.

[47] Soyatec. eUML2, visited: January 27, 2008. `http://www.soyatec.com/euml2/history/`.

[48] Josef Stepisnik. *Distributed Object-Oriented Architectures. Sockets, Java RMI and CORBA.* Diplomica, 2007. ISBN:3836650339.

[49] W. Richard Stevens. *Advanced Programming in the UNIX® Environment*. Addison Wesley Professional, 1992. ISBN:0-201-56317-7.

[50] Sun. Jconsole, visited: February 24, 2008. `http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html`.

[51] Sun. The reflection api, visited: February 3, 2008. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

[52] R. Tolksdorf and R. Menezes. Using swarm intelligence in linda systems. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *Proc. of the 4th International Workshop Engineering Societies in the Agents World*, London, UK, 2003. `http://citeseer.ist.psu.edu/tolksdorf03using.html`.

[53] Robert Tolksdorf. Laura — A service-based coordination language. *Science of Computer Programming*, 31(2–3):359–381, 1998. `http://citeseer.ist.psu.edu/tolksdorf98laura.html`.

[54] Robert Tolksdorf and Antony I. T. Rowstron. Evaluating fault tolerance methods for large-scale linda-like systems. In Hamid R. Arabnia, editor, *PDPTA*. CSREA Press, 2000. `http://dblp.uni-trier.de/db/conf/pdpta/pdpta2000.html`.

[55] S. Wasserman and K. Faust. Social network analysis. Cambridge University Press, 1994.

[56] Wikipedia. Log4j, visited: February 3, 2008. http://en.wikipedia.org/wiki/Log4j.

[57] Wikipedia. Logo (programming language), visited: January 23, 2008. http://en.wikipedia.org/wiki/Logo_(programming_language).

[58] Wikipedia. Eclipse (software), visited: January 27, 2008. http://en.wikipedia.org/wiki/Java_eclipse.

[59] Wikipedia. Apache Ant, visited: January 28, 2008. http://en.wikipedia.org/wiki/Apache_Ant.

[60] Wikipedia. SOAP, visited: January 30, 2008. http://de.wikipedia.org/wiki/SOAP.

[61] Uri Wilensky. *NetLogo*. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, Illinois, 1999. http://ccl.northwestern.edu/netlogo/.

[62] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

List of Figures

1. Centralized tuple spaces . . . 11
2. Partitioned tuple spaces . . . 12
3. Full replication . . . 12
4. Intermediate replication . . . 13
5. Idealistic distribution of tuples in SWARMLINDA . . . 21
6. Self-organization of ants by tracking pheromones . . . 24
7. Routing based on pheromones . . . 25
8. Retrieval of tuples via hashing in LINDA . . . 26
9. Retrieval of tuples by tracking pheromones in SWARMLINDA . . . 29
10. Snapshot of a SWARMLINDA system at runtime . . . 31
11. NetLogo controlling interface . . . 37
12. Part of the NetLogo system architecture modeled in UML . . . 39
13. Architecture of the NetLogo extension API in UML . . . 40
14. Architecture of the `tools` package . . . 44
15. Network Generation Simulator . . . 46
16. Control elements and visualization area of the SWARMLINDA Simulator . . . 49
17. Plotting area of the SWARMLINDA Simulator . . . 53
18. Test section of the SWARMLINDA Simulator . . . 55
19. Comparison between C_{orig} and C_{mod} . . . 61
20. Sigmoid curve defined by Equation 12 . . . 63
21. Comparison between P_{drop}^{orig} and P_{drop}^{mod} in dependence on $c(X)$ . . . 64
22. Comparison between P_{drop}^{orig} and P_{drop}^{mod} in dependence on K . . . 65
23. Entropy curve defined by Equation 15 with $\gamma_i = 100$ . . . 67
24. Comparison of node entropies . . . 68
25. Emergence of homogeneous cluster structures due to tuple movement . . . 73
26. Development of the pickup probability in dependence on k . . . 74
27. A 25 nodes comprising scenario with Spatial Clustering . . . 77
28. Development of Ψ_i in dependence on γ_i and *max-size* . . . 78
29. A five nodes comprising scenario with Overclustering Avoidance . . . 79
30. Training effect of the system by executing *out*-primitives . . . 82
31. Development of the entropy during the test runs of the *out*-primitives . . . 83
32. Network topology with two marked paths . . . 84
33. Training effect of the system by executing *in*-primitives . . . 87
34. Development of the entropy during the test runs of the *in*-primitives . . . 88
35. Development of the constitutions of tuple spaces . . . 89
36. Entropy development depending on the amount of cleaning ants . . . 91
37. Self-organization of tuples during tuple movement . . . 92
38. Average entropy development of scenarios defined in Table 18 . . . 95
39. Comparison of the average entropies based on Table 18 and 19 . . . 97
40. Distribution of tuples with *aoc* and *sc* based on a four node seeding . . . 100
41. Distribution of tuples with *aoc* and *sc* based on a 20 node seeding . . . 102

42. Distribution of tuples with *aoc* and *sc* without seeding 104
43. Network graphs showing the distribution of tuples based on *aoc* and *sc* . . . 105
44. Package diagram of the extension classes for the SWARMLINDA Simulator . 118
45. UML diagram of the extension classes for the SWARMLINDA Simulator . . . 119

List of Tables

1. Pheromone distribution among the visited nodes 29
2. Probability distribution between S_1, S_2 and S_9 30
3. Update of the pheromone distribution among the visited nodes 30
4. Control elements in NetLogo 37
5. Manifest parameters for NetLogo extensions 40
6. Added primitives in NetLogo 56
7. Metrics: Definition of *Scenario*_1 − *Scenario*_3 61
8. Metrics: Definition of *Scenario*_4 − *Scenario*_6 69
9. Scenarios of tuple distribution in a six node comprising network 69
10. Entropy calculation based on the tuple distribution given by Table 9 70
11. Constitutions of tuple spaces according to Figure 25 72
12. Constitutions of tuple spaces according to Figure 26 74
13. Evaluation: Definition of *Scenario*_1 − *Scenario*_4 81
14. Example: Node connectivity and path probability 85
15. Evaluation: Definition of *Scenario*_5 − *Scenario*_8 86
16. Evaluation: Definition of *Scenario*_9 − *Scenario*_11 90
17. Evaluation: Definition of *Scenario*_12 − *Scenario*_14 91
18. Evaluation: Definition of *Scenario*_15 − *Scenario*_18 94
19. Evaluation: Definition of *Scenario*_19 − *Scenario*_22 96
20. Evaluation: Definition of *Scenario*_23 − *Scenario*_26 99
21. Evaluation: Definition of *Scenario*_27 − *Scenario*_30 101
22. Evaluation: Definition of *Scenario*_31 − *Scenario*_33 103

A. Diagrams

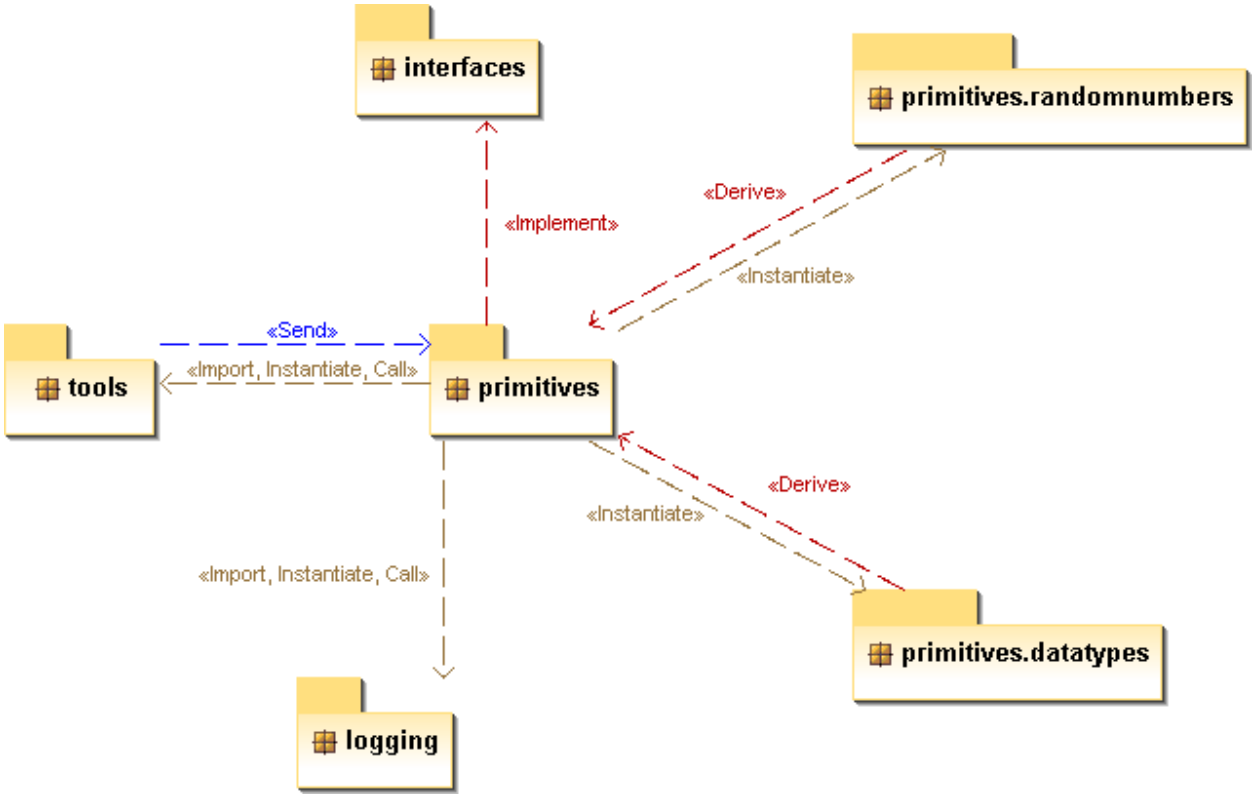


Figure 44: Package diagram of the extension classes for the SWARMLINDA Simulator

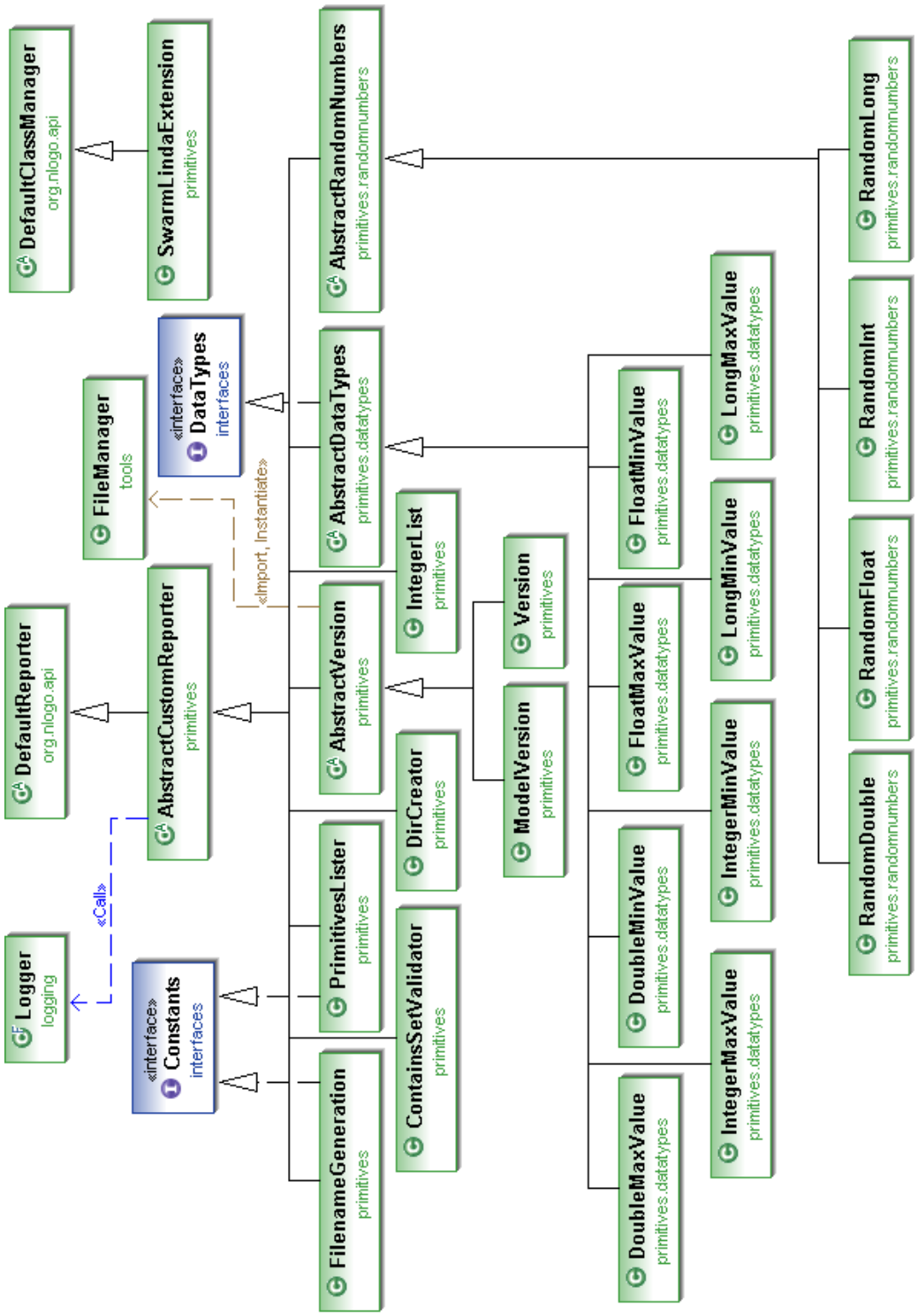


Figure 45: UML diagram of the extension classes for the SWARMLINDA Simulator