Bringing Autonomic Services to Life

Deliverable D3.3

Software Implementation of Modules for Adaptive (T3.1) Aggregation and Unit Differentiation (T3.2)

Status and Version:	Final Version	
Date of issue:	29.03.2008	
Distribution:	Public	
Author(s):	Name	Partner
	Elisabetta Di Nitto	DEI
	Daniel Dubois	DEI
	Raffaela Mirandola	DEI
Checked by:	F. Zambonelli, A. Manzalini	UNIMORE, TI

Abstract

This document constitutes the textual part of Deliverable D3.3.

D3.3 software, and other related documentations (e.g. Readme), are available at:
https://152.66.87.177/repositories/cascadas/trunk/wp3/D3.3

Table of Contents

1 Introduction

The purpose of this document is to give a formalization of a standard self-aggregation algorithm interface. This interface has been used in a distributed framework that is able to simulate these algorithms in a distributed setting. To accomplish this goal we have used a simplified ACE Model, however a new more general purpose interface is under development to be integrated more easily in WP1 toolkit. The final aim of the simulation framework is not only to provide an environment to develop and test new algorithms, but also to build a knowledge base that can be used by ACEs in choosing the best clustering strategy for their particular situation.

2 Self-Aggregation as an ACE Functionality

In the context of the WP1 Toolkit the aim of this work is to provide self-aggregation algorithms as functionalities that can be used by ACEs in order to create cluster of nodes of compatible types. The notion of compatibility is defined as equality in the case of normal clustering and inequality in the case of reverse-clustering. The type is application specific and can be defined as a common goal, a common functionality or any other characteristic of an ACE.

ACE-to-Algorithm communication is achieved by initializing the algorithm with a reference to the local ACE, so the algorithm can access basic ACE methods, like methods to send messages to other ACEs, to send replies to received message, and to retrieve replies to their own messages.

Algorithms running on different ACEs can communicate by using a message event protocol in which each event message contains the source ACE, the destination ACE, the command name and its parameters. Each algorithm is registered on the ACE as a listener for event messages, therefore as soon as a message is received from another ACE an event is fired on the algorithm.

Figure 1 shows how the communication works among algorithms of different nodes: the algorithm asks its ACE to send a message using the communicator object that is part of the ACE, then the receiving ACE will fire the message event of its own algorithm. A message can also be replied to provide a synchronous communication mechanism, in this case the reply can be retrieved directly from the ACE after the message has been sent.

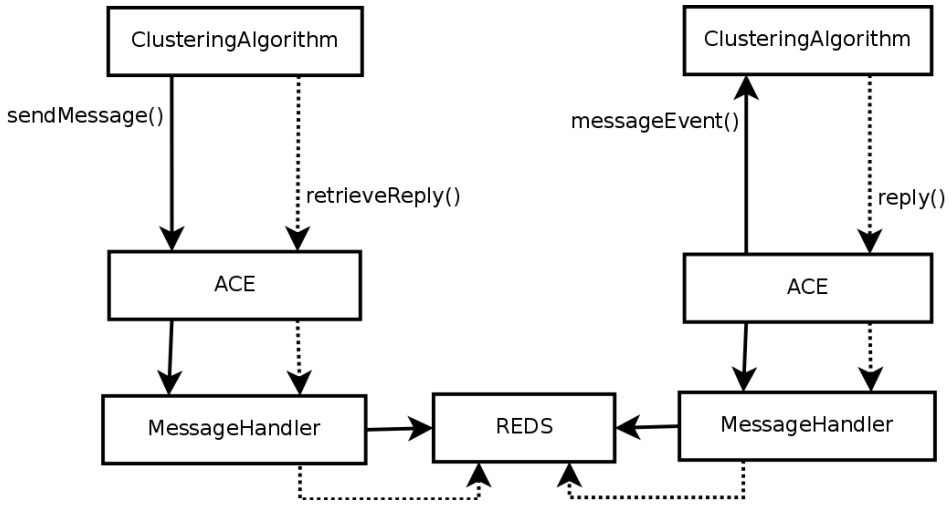


Figure 1: Communication schema for Algorithm-to-Algorithm communication

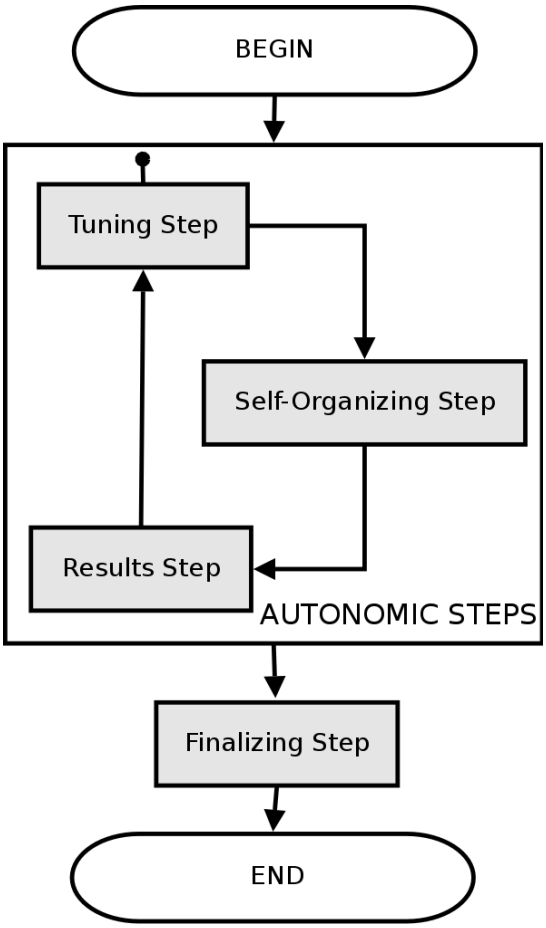
In Table 1 it is possible to see the methods of the *ClusteringAlgorithm* interface with a brief explanation. The protocol used for exchanging messages among them is left to the implementation.

`initializeAlgorithm( instanceId : String, aceInterface : AceCoreInterface )`	Initializes the algorithm. "*instanceId*" is an identification for the algorithm instance inside the node. "*aceInterface*" is used to communicate with the ACE.
`finalizeAlgorithm()`	Terminates the algorithm.
`messageEvent(message : AceMessage)`	Notifies the algorithm that a new event message has been received.
`isCompleted() : Boolean`	Checks if the algorithm is completed and can be safely removed from the ACE.
`setOption(name : String, value : Object)`	Sets an implementation dependent algorithm options.
`getOption(name : String) : Object`	Gets implementation dependent algorithm options.
`setNeighbors(neighbors : AceList)`	Sets the list of neighbor nodes.
`getNeighbors() : AceList`	Gets the list of neighbor nodes.
`getInstanceId() : String`	Get the instance identification: it is used to distinguish different instances of the same algorithm inside the node.

Table 1: ClusteringAlgorithm interface[1]

[1] For information regarding other classes and interfaces please refer to the JavaDOC documentation that comes with this document.

The life cycle of the algorithm that implements the *ClusteringAlgorithm* interface should respect this schema:



Initialization Step

The algorithm is instantiated and it receives information about the *local node*, an *ID* that identifies the current instance of the algorithm (to solve ambiguity when there are parallel executions of the same algorithm in the same ACE) and an interface used to communicate with its local ACE.

Tuning Step

In this step some algorithm proprieties can be optionally set by who is loading the algorithm.

Results Step

Optional step. The algorithm report its results to the local node or on a remote node (useful to perform algorithm analysis).

Self-Organizing Step

The algorithm participates to the self-organization of the system by sending and receiving event messages.

Finalizing Step

In this step the algorithm terminates cleanly its execution, stops its threads and marks itself as completed. Only a completed algorithm can be safely unloaded.

3 Clustering Algorithms

This section presents several clustering algorithms. The first of them is the original (Saffre, et al., 2006) algorithm that has already been presented in the D3.1 deliverable, while the others are variants of it that in certain situations can behave better than the original version. Details on those variants can be found in (Dubois, 2007).

3.1 Original Clustering Algorithm

The purpose of this algorithm is to cluster a network of ACE nodes according to a particular criteria. In our simulator we have nodes characterized by an *ID* and a *Type*, therefore the clustering algorithm will add links between nodes with the same type and removes links between nodes with different types. This algorithm has two different implementations: *Passive* and *Active* (known also as *On Demand*). The first uses a protocol with less messages, but needs an upper bound on the number of maximum neighbors to avoid reaching scale-free topologies, the seconds exchanges more messages, but, for the reasons explained in D3.1 WP3 deliverable, does not need an upper bound.

Passive Clustering

1. a random node elects itself as the matchmaker node;
2. the matchmaker node chooses two neighbors that have the same type;
3. the matchmaker makes the two chosen neighbors establish a new link;
4. the matchmaker removes a link between itself and one of the chosen neighbors.

Active Clustering

1. a random node elects itself as the initiator node;
2. the initiator node elects a matchmaker node among its neighbors;
3. the matchmaker node chooses one neighbor that shares the same type with the initiator node;
4. the matchmaker makes the initiator and the chosen neighbor establish a new link;
5. the matchmaker removes a link between itself and the chosen neighbor.

Passive Original Algorithm:

```
matchmaker = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if (matchmaker has two neighbors n1 and n2 such that n1.type == n2.type) then
    add link between n1 and n2
    remove link between matchmaker and n1
  fi
od
```

Active Original Algorithm:

```
initiator = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if (initiator has a neighbor n1) then
    matchmaker = n1
    if ((matchmaker has a neighbor n2 such that n2.type == initiator.type) AND
        (n2 != initiator)) then
      add a link between initiator and n2
      remove a link between matchmaker and n2
    fi
  fi
od
```

3.2 Fast Clustering

This algorithm is similar to original Clustering, but enforces the following additional constraint: the matchmaker cannot choose neighbors that share the same type of it. This additional constraint makes impossible to remove links between nodes of the same type. The advantage of this algorithm is that it improves the overall clustering of the network in the fastest way, the disadvantage is that it can get stuck in local clustering maximums because it is too constrained.

Passive Fast Algorithm:

```
matchmaker = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if ((matchmaker has two neighbors n1 and n2 such that n1.type == n2.type) AND
      (matchmaker.type != n1.type)) then
    add link between n1 and n2
    remove link between matchmaker and n1
  fi
od
```

Active Fast Algorithm:

```
initiator = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if (initiator has a neighbor n1) then
    matchmaker = n1
    if ((matchmaker has a neighbor n2 such that n2.type == initiator.type) AND
        (n2 != initiator) AND matchmaker.type != n2.type) then
      add a link between initiator and n2
      remove a link between matchmaker and n2
    fi
  fi
od
```

3.3 Accurate Clustering

This algorithm is based on the original Clustering, but relaxes some of its constraints: the matchmaker is now able to create links between neighbors of different type unless it does not have to disconnect a node with the same type. This additional constraint adds "noisy" links to the network, but does not increase the number of links between nodes of different type. The advantage of this algorithm is that it does not get stuck in local clustering maximums, however the additional noise slows it down, therefore it requires more messages to converge.

Passive Accurate Algorithm:

```
matchmaker = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if ((matchmaker has two neighbors n1 and n2) AND (matchmaker.type != n1.type)) then
    add link between n1 and n2
    remove link between matchmaker and n1
  fi
od
```

Active Fast Algorithm:

```
initiator = LOCALNODE
for i=1 to NUM_ITERATIONS
do
  if (initiator has a neighbor n1) then
    matchmaker = n1
    if ((matchmaker has a neighbor n2 such that n2.type != matchmaker.type) AND
        (n2 != initiator)) then
      add a link between initiator and n2
      remove a link between matchmaker and n2
    fi
  fi
od
```

3.4 Adaptive Clustering

This algorithm is modelled as a FSM in which the node starts behaving as the most constrained algorithm (*Fast Clustering*) until that algorithm becomes stuck. In such event the algorithm switches to a medium constrained algorithm (*Original Clustering*) and, as soon as it becomes stuck again, it switches to the less constrained one (*Accurate Clustering*). If the node earns a new link then the algorithm is changed again to the most constrained. This FSM can be seen in Figure 2 and the meaning of the transitions is described below:

- *Failure*: triggered when an algorithm is not able to iterate because its constraints does not make possible to choose valid neighbors;
- *Success*: triggered when an algorithm is able to complete an iteration;
- *New Neighbors*: triggered when a new neighbor has been added.

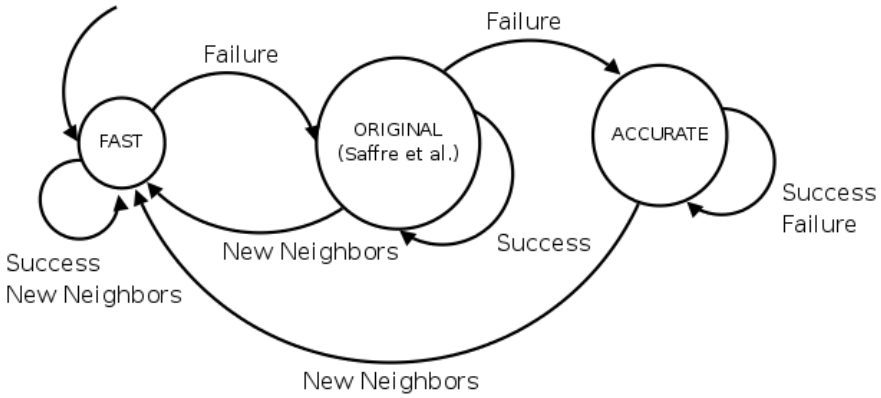


Figure 2: Adaptive Clustering FSM

4 Simulation Framework

The purpose of this framework is to simulate and run simplified ACEs in different nodes of an existing network with the purpose of running self-organizing clustering algorithms. The main requirements of this framework are:

- transparency to the network and to the middleware used to run it;
- unique standard interface for algorithms to provide generality and possibility of reuse;
- autonomic clustering algorithms and performance indicators are fully modularized.

4.1 Architecture

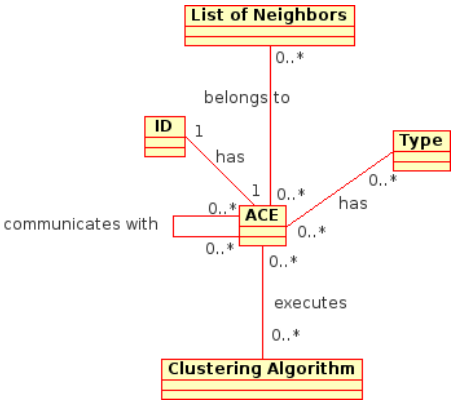


Figure 3: Simulation Framework High-Level Conceptual Model

In Figure 3 it is possible to see the main components of the simplified ACE conceptual model that we used.

ACE

This component represents the node of the system, identified by an *ID* and a *type*. It can be conceptually seen as an active component equipped with all the instruments to communicate with other nodes and to start/stop algorithms. In the real implementation (as shown in Figure 4) it is implemented by classes **AceCore** and **AceNode**.

ACETYPE

This component simply represents the type of an ACE. Its implementation is strictly application dependent.

ID

This component identifies univocally the current ACE.

GROUP/LIST OF NEIGHBORS

This component specifies the belonging of the current ACE to one or more groups. This concept can be used to allow ACE-to-ACE cooperation in cooperative self-organization algorithms.

CLUSTERING ALGORITHM

This component is completely abstract and can represent any kind of self-organization algorithm of the simulated system. It is able to modify the state of the ACE and to communicate to other ACEs.

4.2 Implementation

After explaining the high-level model we propose a possible implementation of it. The implementation we are going to discuss has been developed using a software life-cycle based on prototyping and the JUnit Framework for testing and validation.

The UML class diagram in Figure 4 introduces the architecture that will be explained in the following paragraphs.

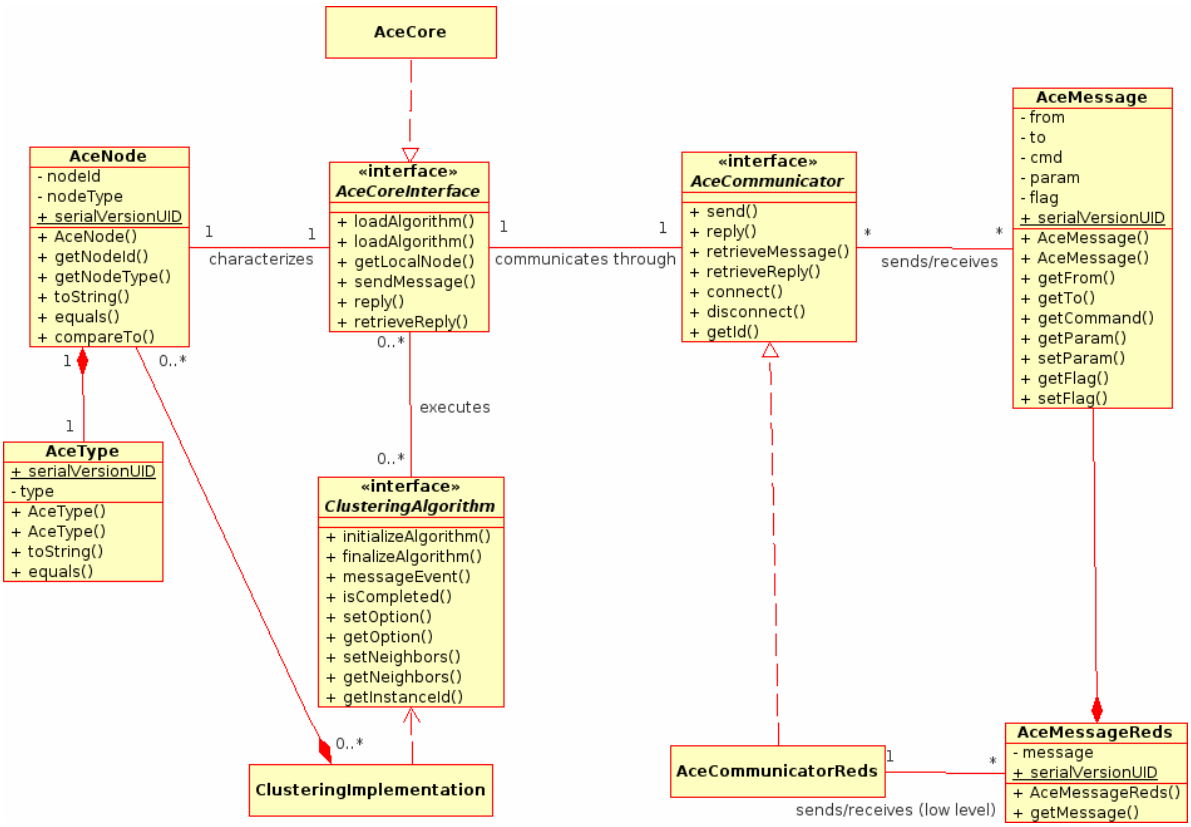


Figure 4: UML Class Diagram of Simulator Architecture

AceCoreInterface

The purpose of this interface is to act as an aggregation point for the node itself, the algorithms, and the communicator. It does not implement any algorithm, but it simply creates the basement for them. The final aim is to easily replace this component with the WP1 ACE prototype.

The most important functions provided by this component are:

1. starts a connection to the middleware using the communicator specified in the constructor;
2. instantiates a new **AceNode** using the **AceType** specified in the constructor;
3. starts the first algorithm[2] specified in the constructor;
4. starts a thread pool in order to receive message events from the middleware without blocking;
5. gently shuts down the node and disconnects the middleware when all algorithms are completed.

AceNode

This class is a simple immutable class representing an ACE. It is used in all aggregation lists, for example to provide the neighbor lists of a node and statistical information about the current state and every node transition.

It does not contain any runnable code except redefined *compareTo* and *equals* methods. The final purpose is having a single object that can identify univocally an ACE.

AceType

This is a class that represents the type of an ACE. The type can be, for example, one of the following information:

- capabilities of the ACE;
- goals of the ACE;
- any kind of structured information.

For simplicity our prototype uses the AceType mainly to represent strings and numbers, but any type of information can be used. From a high-level point of view the AceType can be seen also as part of the Knowledge Base of the ACE that can be accessed by every algorithm.

AceCommunicator Interface

This interface provides all the high level communication methods needed by an ACE to allow ACE-to-ACE communication. Its purpose is to provide methods for the following communication-related actions:

- *connecting/disconnecting* to/from the middleware;
- *sending* a message to the middleware;
- *replying* to a previously received message (synchronous messaging);

[2]At least one algorithm must be specified in order to keep the AceCore running. In fact it terminates when all loaded algorithms are completed.

- *retrieving* new messages and replies.

The *ID* information that appears in the interface represents the node identifier used inside the middleware, which can be, for example, the host-name of the local machine.

AceMessage Class

The **AceMessage** class is not middleware-dependent: the reason for this choice is that every middleware has already its internal message format and that the **AceCommunicator** implementing classes can act also as message translators.

Messages contain the following information:

- *Source*: the *ID* of the sending ACE (it is the *ID* used inside **AceNode** and not the *ID* used inside the middleware);
- *Destination*: the *ID* of the destination ACE or the special *ID* of the broadcast address;
- *Command*: an algorithm-dependent command;
- *Parameter*: a command-dependent parameter: it can be any serializable object;
- *Flag*: additional optional information.

AceRepliableObject Class

In the previous paragraph we have seen that a node is not only able to send messages, but it is also able to reply to existing ones. To enable this mechanism a new **AceRepliableObject** class has been introduced: it is a container for all middleware-dependent data required to take advantage of the reply mechanism (if available). The task of creating instances of this class is delegated to the **AceCommunicator** middleware-dependent implementing class.

4.3 Distributing Issues

The sequence diagrams in Figure 5 and Figure 6 show new protocol implementations of the clustering algorithms for the simulation framework, these new protocols have only four message exchanges for the passive version and five message exchanges for the active version. Nevertheless using the algorithms may not be as simple as it seems: an autonomic distributed environment introduces problems that have not been investigated yet. These problems are:

- when to start an algorithm iteration;
- what to do in the event of a failure;
- how to prevent synchronization issues.

Synchronization issues can be easily avoided by adding a lock flag to each node: when an initiator starts its iteration, it will lock itself and related nodes in order to protect from unwanted commands coming from other nodes. In fact, a locked node will refuse all commands issued by nodes that are not involved in the iteration.

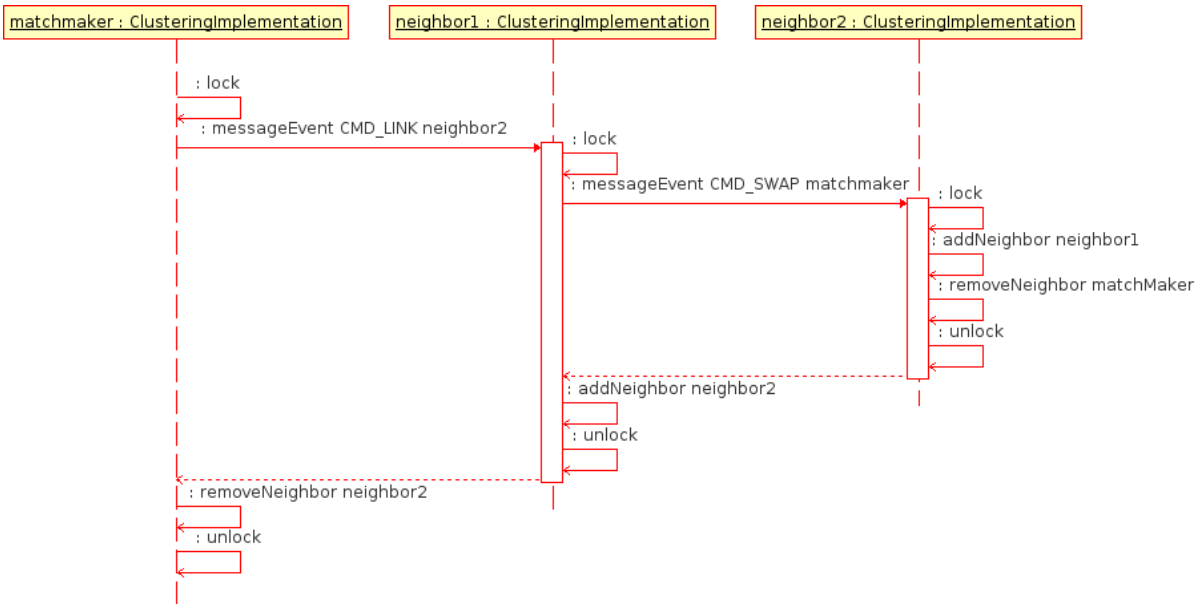


Figure 5: Passive Clustering Sequence Diagram

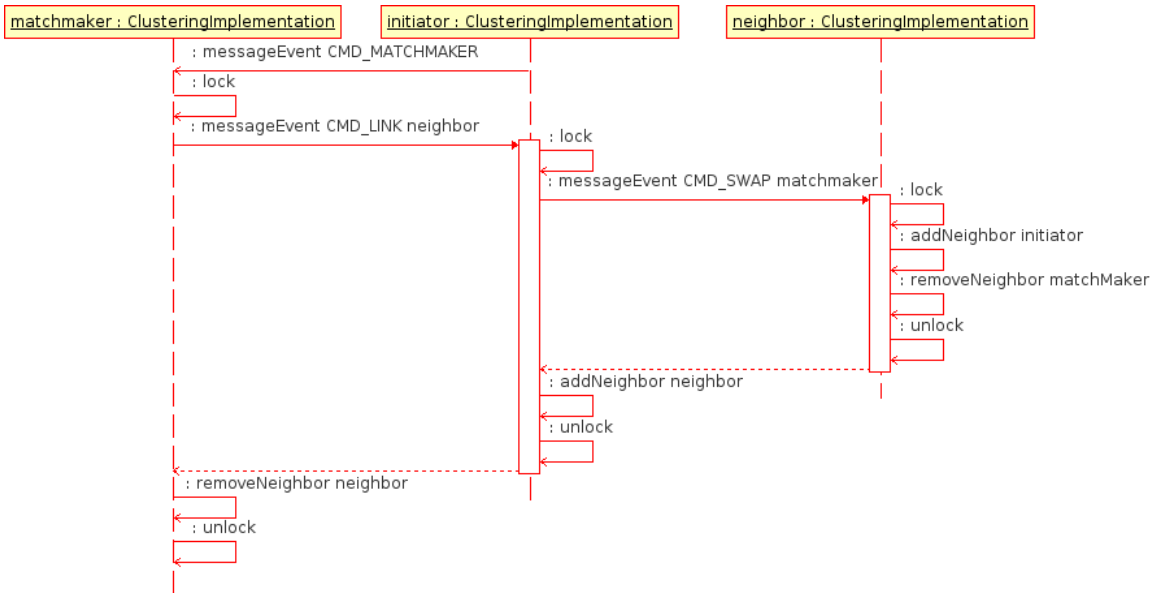


Figure 6: Active "On Demand" Clustering Sequence Diagram

To handle problems like dying nodes, impossible iterations, and message losses, all critical messages exchanged by the nodes must be confirmed with a reply. In the eventuality of a timeout no changes are performed.

Another problem that has been encountered was the *temporizing problem*: this problem consists in choosing the right amount of time to wait between starting a new iteration of the algorithm. To solve this problem we forced each node to stay idle for a random amount of time between an iteration and the subsequent one.

4.4 Manager Node

Until now we have seen the architecture of the framework and examples of algorithms, but we did not mention anything about what components are dedicated to the management of the simulation and the gathering of the results. To fulfill the remaining requirements of the framework the concept of *Manager* should be introduced.

4.4.1 Clustering Manager Overview

The Manager is that piece of software that instructs all other nodes to start a particular algorithm, it also provides remote initialization functions to generate the initial status of the whole environment and finally it can analyze and store the data produced during the simulation.

In Figure 7 we will provide an example of manager architecture that can be suitable for the clustering algorithms: the *Clustering Manager*.

The architecture displayed in the class diagram of Figure 7 is divided into three main macro-components:

- Manager;
- Topologies;
- Analyzers.

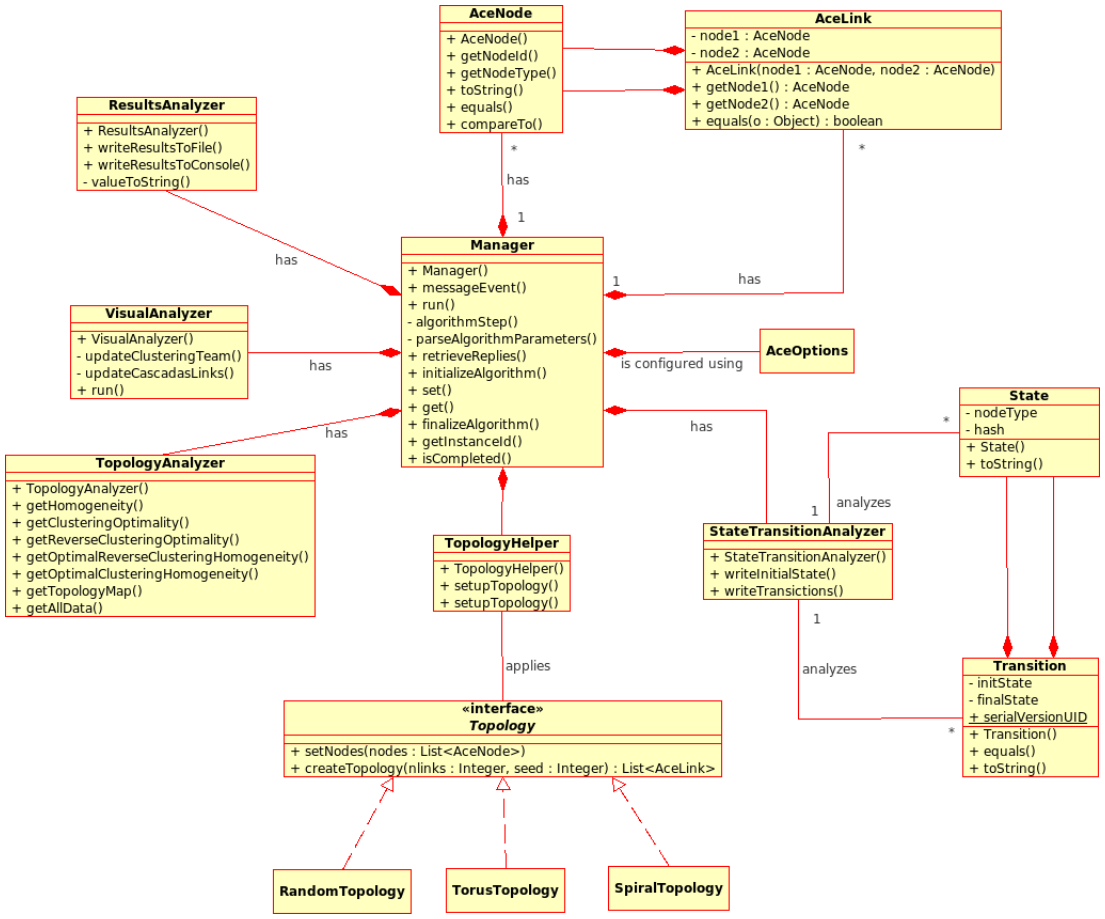


Figure 7: Clustering Manager class diagram

Manager Class

The Manager class implements the ClusteringAlgorithm interface and it is seen as a clustering algorithm. During the simulations it performs the following tasks:

1. it sends to the broadcast address a discover request: all the nodes will reply with a serialized version of their **AceNode** class;
2. it sends to all discovered nodes a message event that instructs them to load the specified algorithm (**Clustering**) in order to be ready for its execution;
3. it loads a topology algorithm using the **TopologyHelper** class;
4. it starts the topology algorithm in order to create an initial list of neighbors in every node;
5. it starts at regular intervals all the required steps of the clustering algorithms until the value of *homogeneity* converges to a fixed value;
6. it monitors the events that are being executed in remote nodes in order to keep track of changes in the network topology;
7. it finally prints the results of the simulation.

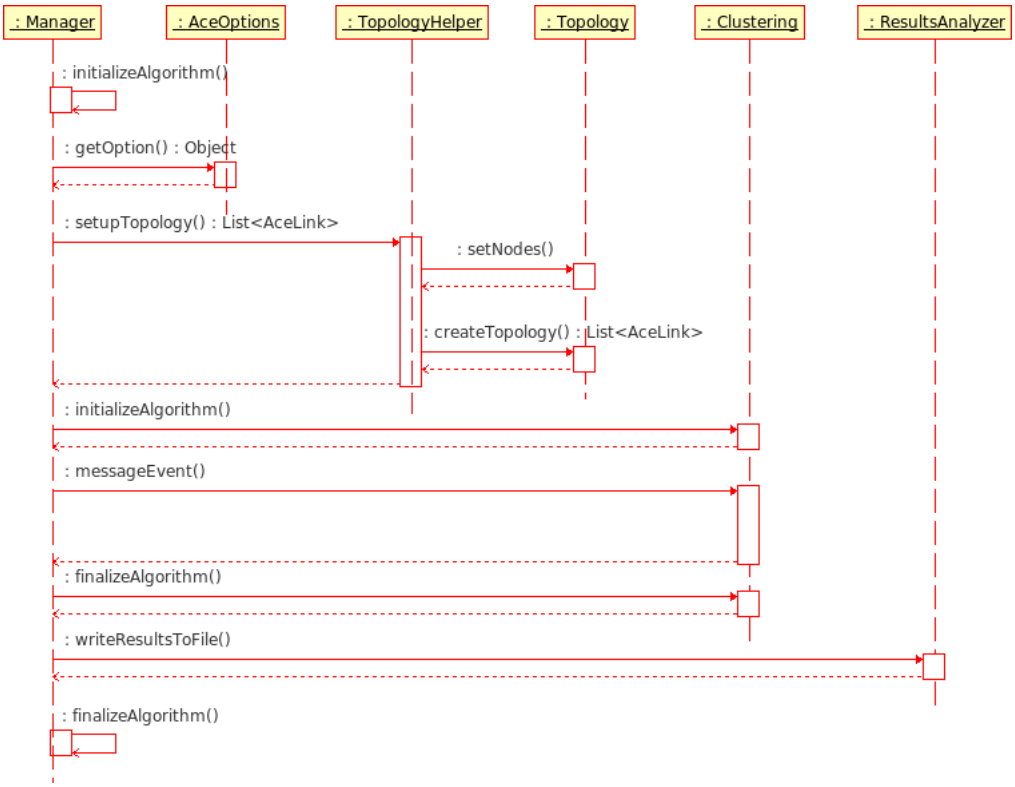
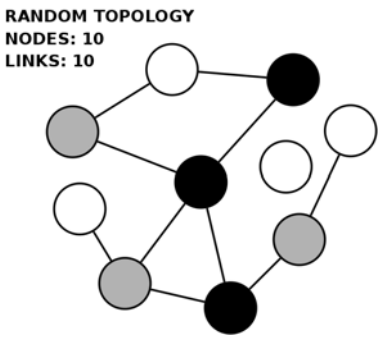


Figure 8: Manager Sequence Diagram

4.4.2 Topologies

A topology algorithm is defined as a function that creates a list of links from a list of nodes. The **Topology** interface is used in the clustering manager to implement topology algorithms that will be used to initialize the network. As can be seen in the sequence diagram of Figure 8 the manager node will use the **TopologyHelper** class as a wrapper for loading and using the topologies.

Random Topology



This is the simplest topology. This algorithm creates random links between the nodes making it possible to have various types of topologies like, for example, partitioned graphs and singletons.

Torus Topology

The torus topology algorithm will create links in order to form a main donut-like loop. If the number of links is greater or equal to the number of nodes, the additional links are created in such a way that the single loop becomes a chain-shaped loop.

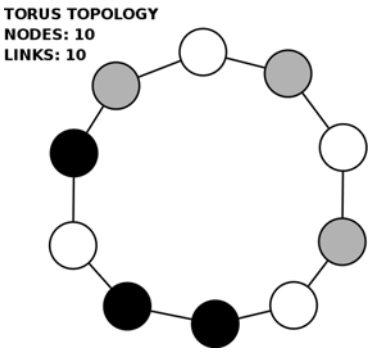


Figure 9: Torus Topology Example

Spiral Topology

The spiral topology algorithm is very similar to the torus one, but it forms a "broken" loop. All nodes can be seen as part of a spiral, that can become a chain-shaped spiral if the total number of links is greater than the number of nodes.

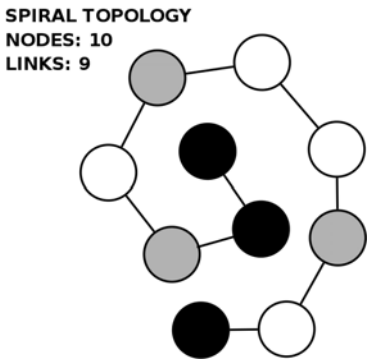


Figure 10: Spiral Topology Example

Scale-free Topology

The Scale-free topology (Barabasi, et al., 1999) is a topology similar to real-world networks in which there are supernodes (or hubs) and normal nodes. This network topology is characterized by the following relation: $P(k) \sim k^{-\gamma}$ where *P*(*k*) is the probability

of a generic node to be linked to *k* neighbors and γ is a generic constant that is between 2 and 3 for most real networks.

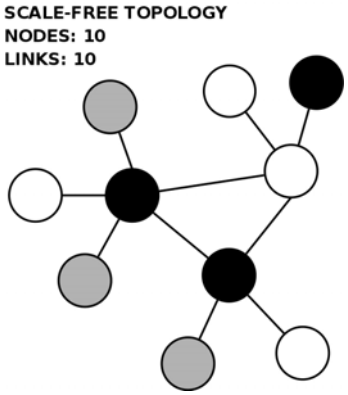


Figure 11: Scale-Free Topology Example

4.4.3 Analyzers

The analyzers are the last components of the manager. Their purpose is to manage all the data that the manager receives from the other running algorithms. This data is pretty useless if it is not organized, structured and presented to the final user, since having "good" data is the primary aim of the entire simulation process. In the following paragraph we will point out the most important details about the clustering manager analyzers.

Topology Analyzer

This analyzer will use the updated network topology to calculate topology-related performance indexes like homogeneity and optimality.

Results Analyzer

This analyzer is used to select the performance indexes to show when the simulation is completed. The same information will be used to create a CSV file ready to be analyzed using the favorite spreadsheet or database application in order to perform some data mining and produce graphical results.

5 Simulations

5.1 Input Parameters

In order to use a systematic approach we need to identify which input data we can use to start different simulations. To perform this work we have to think about the eventual characteristics and needs of possible real systems that will host the algorithms. According to the clustering example we can argue that:

- initial network topologies are important;
- the number of nodes in the network is important;
- the number of types, and therefore the initial homogeneity of the system is important.

All these important characteristics are parameterized in the simulation framework and can therefore be used as possible input data for observing the algorithms behaviors. **Table 2** shows possible values for the input parameters.

INPUT PARAMETERS	EXAMPLE VALUES
Time unit	5 seconds
Maximum execution time	100 seconds
Number of types	2, 4, 6, 8, 10, 12, 16
Number of nodes	100
Number of links	150, 200, 250
Topology algorithm	random, torus, spiral, scale-free
Topology initialization	random, deterministic seed
Max number of neighbor	6, unlimited
Number of tests	20

Table 2: Input Parameters

5.2 Performance Indexes

Homogeneity

This performance index represents the level of the network clustering related to nodes having neighbors of the same type (Saffre, et al., 2006). This index is a number between 0 and 1: lower values mean a reverse-clustered network, while higher values mean a clustered network.

The formula that calculates this index is the following:

$$h = \frac{\sum_{i=1}^{N} v(node_i)}{N}$$

Where *N* is the total number of nodes, *v(x)* is the total number of nodes of the same type linked to *x* and *L* is the total number of links in the system.

Optimality

This index has the same purposes of the homogeneity index, it is algorithm-related and it is a number between 0 and 1. 0 means that the algorithm is far from its goal, 1 means that it has completely reached its goal (Dubois, 2007).

The formula that calculates this index for the clustering problem is the following:

$$optimality_{Clustering} = \frac{h}{h_{optimalClustering}}$$

Where *h* is homogeneity, while $h_{optimalClustering}$ is the homogeneity created by an optimal clustering algorithm to the same topology.

Unlike the homogeneity, this index can always reach its upper bound even if the homogeneity of the network cannot go beyond a particular value. An optimality of 100% means that the algorithm behaves like the optimal algorithm.

$$optimality_{ReverseClustering} = \frac{1-h}{1-h_{optimal\,ReverseClustering}}$$

Even for reverse clustering the optimality meaning remains the same: bounded between 0 and 1 with 1 as its goal value.

Total Messages

This index simply measures the total number of messages that have been sent by all the nodes of the system. Sometimes an algorithm with a good optimality may use too many network resources: this can be quite bad in production environments based on wireless networks, where the data flow should be kept as low as possible.

Useful Messages

This index provides information about the network load. It is calculated in the following way:

$$usefulmessages = \frac{successfulmessages}{totalmessages}$$

Where *totalmessages* is the performance index previously introduced, while *successfulmessages* is the total number of messages and replies that have been completed without errors.

This index is bounded by 0 and 1 and it represents the quantity of useful information that is being transmitted by the network. Lower values mean that the algorithm is not behaving efficiently because a lot of junk-messages are flooding the underlying middleware.

Time of Convergence

Another important index is the time of convergence. It measures the elapsed time between the algorithm execution and its convergence. We assume that a clustering algorithm is convergent when the homogeneity of the network does not change over 3% for 30 consecutive simulation seconds.

Useful Iterations

This indicator is similar to the useful messages one, with the difference that it counts iterations instead of messages. This index is very correlated to the messages one, however it has been useful during the optimization phase of the algorithms to decide the optimum number of iterations that provides a good compromise between speed and iteration failures.

Links-variance

This index gives information about how the node degree differs among the nodes.

$$linksv = \frac{\sum_{i=1}^{N}(d_i - \frac{2N}{L})^2}{N}$$

Where N is the total number of nodes, L is the total number of links, and d_i is the number of neighbors of node i. A high value of this index means that there are many super nodes, a value of zero means that all the nodes have the same number of neighbors.

Domains number

This indicator gives the number of groups of interconnected nodes with the same type, where the group must have at least two nodes (Saffre, et al., 2006).

Average/variance of domains size

These indicators give the mean and the variance for the size of all domains.

5.3 Execution

This section explains how to use the framework to start different types of simulations.

The class used to start the simulator is the **Main** class found in the manager package. The typical operations of the class are the following:

- starting one of more middleware brokers (REDS) and relative connections between them in order to constitute a homogeneous network of brokers;
- instantiation of one or more **AceCore** classes representing the "active" nodes of the system;
- for each node of the system a new **AceCommunicator** is created to connect the node to the respective broker;
- another **AceCore** that runs the clustering manager algorithm is connected to the broker: it will start, manage and terminate the simulation process;
- according to the events received from the manager all the nodes start executing the clustering algorithm until they receive a stop event command;
- finally the manager prints the results and the application is terminated.

The execution steps done by the main class are exemplified in the sequence diagram in Figure 12.

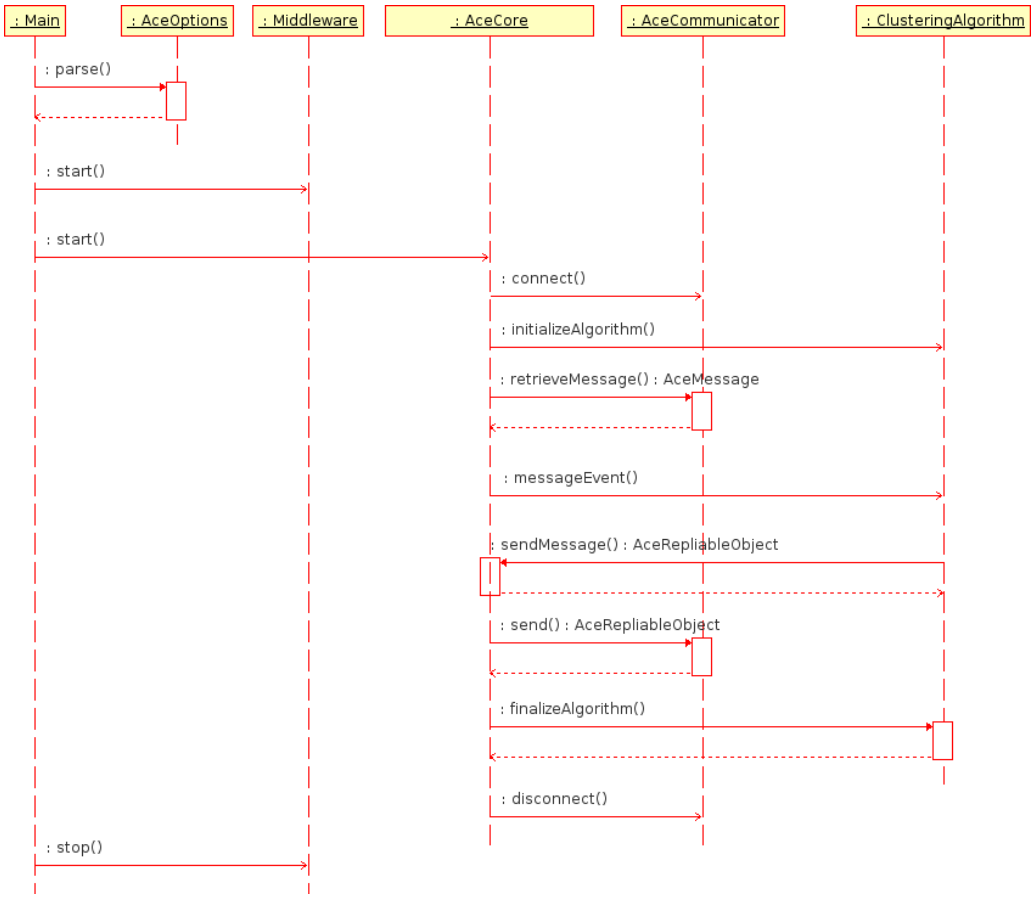


Figure 12: Startup Sequence Diagram

5.3.1 Simulator Frontend Usage

The command line frontend offers the possibility to configure the manager algorithm according to particular parameters. It allows to select which algorithm must be executed and which performance indexes should be generated.

Looking at Table 3 it is possible to visualize a list of commands with relative descriptions. From the above commands it is evident that a simulation can be started using three different modes:

1. using only a local broker, specifying only the local port where the new middleware instance will be started (using options `-startreds true` and `-port`);
2. using only a remote broker, specifying the address and the port of the remote broker where all the local nodes will be connected to (using options `-startreds false`, `-address` and `-port`);
3. using both a local and a remote broker, specifying the local port for the new local middleware instance and the URL of the remote broker that is going to be connected to the local one (using options `-startreds true`, `-port` and `-neighbor`).

Option	Description
-address <...>	set the address of the broker
-algorithm <...>	set the name of the algorithm to start (only Clustering available)
-algparams <...>	set the parameters for the algorithm See Table 4 for additional information about these parameters
-debug <true/false>	show debug messages
-duration <number>	set the duration for an algorithm step in ms
-help	print usage information
-initialfile <....>	write the initial topology to the specified file
-linksnumber <number>	set the number of links
-neighbor <...>	set the URL of a neighbor broker
-nodesname <name>	set the name prefix of the local nodes
-port <number>	set the port of the broker
-results <...>	select the output results for the algorithm
-resultsfile <...>	write the simulation results to a CSV file
-seed <number>	set the seed used by the random number generator
-startmanager <true/false>	enable or disable the manager algorithm
-startreds <true/false>	start a new REDS instance
-steps <number>	set the maximum number of algorithm steps
-topology <...>	set the initial topology (RandomTopology, TorusTopology, SpiralTopology, ScaleFreeTopology)
-transitionsfile <...>	write transitions to the specified file
-typesnumber <number>	set the number of different types
-visual <true/false>	display the algorithm in a GUI

Table 3: Command-line Options

Clustering Algorithm parameter	Type of value	Example of values
active	Boolean	True: active mode False: passive mode
reverse	Boolean	True: reverse clustering False: normal clustering
optimization	String	"saffre": Original (Saffre, et al., 2006) algorithm "fast": Fast algorithm "accurate": Accurate algorithm "adaptive": Adaptive algorithm
maxneighbors	Integer	<=0: unlimited neighbors >0: limited neighbors

Table 4: Clustering Algorithm Parameters

In all three situations only one simulator should enable the manager algorithm, with the additional requirements that it must be the last node to connect to the network (all the simulators except the one that runs the manager algorithm should be started using `-startmanager false` option). In the eventuality that more than one simulator are connecting from the same machine, it is necessary to modify the name of the nodes using `-nodesname` option in order to prevent name collisions.

5.3.2 Single Broker Execution Example

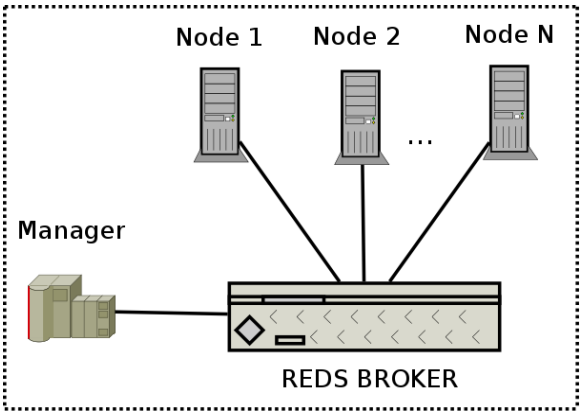


Figure 13: Single Broker Scenario

In this paragraph a single-broker simulation with the following characteristics will be presented:

- 100 total nodes divided into 3 different types with 150 links and distributed using a random topology;
- the selected algorithm is the adaptive active reverse clustering;
- the simulation will keep running until homogeneity convergence is reached.

The command that will run the described configuration is the following:

```
./asf -nodesnumber 100 -linksnumber 150 -duration 10000 -typesnumber 3
      -algorithm Clustering -algparams active=true,mode=accurate,reverse=true
      -topology RandomTopology
       -results homogeneity,roptimality,messages,usefulmessages
```

After some time it is possible to read an output like the following one:

```
*** ACE SIMULATION FRAMEWORK CONFIGURATION ***
    Start Local REDS: enabled
        REDS Address: 127.0.0.1
           REDS Port: 2000
   Nodes Name Prefix: node
     Number of nodes: 100
```

```
     Number of types: 3
Debug Logging Status: disabled
  Start Configurator: enabled
     Number of links: 150
       Topology Name: RandomTopology
             Results: homogeneity,roptimality,messages,usefulmessages
      Algorithm Name: Clustering
Algorithm Parameters: active=true,mode=adaptive,reverse=true
 Initialization seed: 1627062523
       Step Duration: 10000 ms
 Max Number of Steps: unlimited
  Initial State File: initial.txt
   Transitions File: transitions.txt
    Results CSV File: results.csv
[REDS] Broker started
[MAIN] Loading 100 nodes
[MANAGER] Loading Clustering class in all nodes.
[MANAGER] Applying RandomTopology topology to 100 nodes. Max 150 links.
[MANAGER] Starting Clustering algorithm (max unlimited steps of 10000 ms)
*** ALGORITHM RESULTS ***
   nodesnumber : 100
   linksnumber : 150
   typesnumber : 3
      topology : RandomTopology
          seed : 1627062523
     algorithm : Clustering
     algparams : active=true,mode=adaptive,reverse=true
          time : 0
   homogeneity : 0.32666666666666666
   roptimality : 0.6733333333333333
      messages : 0
usefulmessages : 0
... we skip some intermediate steps here ...
*** ALGORITHM RESULTS ***
   nodesnumber : 100
   linksnumber : 150
   typesnumber : 3
      topology : RandomTopology
          seed : 1627062523
     algorithm : Clustering
     algparams : active=true,mode=adaptive,reverse=true
          time : 40000
   homogeneity : 0.006622516556291391
   roptimality : 0.9933774834437086
      messages : 2182
usefulmessages : 0.7089825847846013
*** CONVERGENCE REACHED!  ***
*** End of Statistics ***
```

5.3.3 Multiple Brokers Execution Example

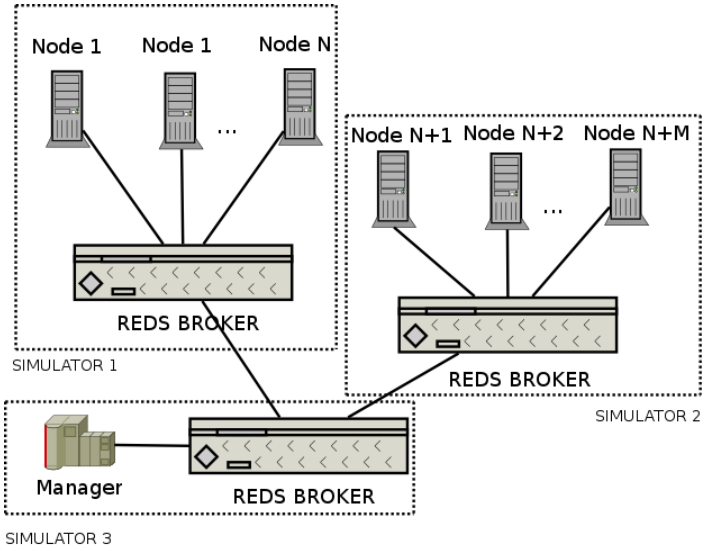


Figure 14: Multiple Brokers Scenario

In this simulation example we will consider a network composed by three different brokers. Every broker will be started in a distinct simulator using a different machine, this way we will achieve the goal of having a more realistic situation for the simulations.

The parameters of the following simulation are the following:

- 150 total nodes (50 for each simulator/broker) divided into 2 types and 200 links distributed over a spiral topology;
- the algorithm used is the fast passive clustering.

This time we will start three instances of the simulator running the following commands on three different machines.

Machine 1 (10.69.0.72):

```
./asf -nodesnumber 50 -typesnumber 2 -port 2000
      -startreds true -startmanager false
```

Machine 2 (10.69.0.73):

```
./asf -nodesnumber 50 -typesnumber 2 -port 2000
      -startreds true -startmanager false -neighbor reds-tcp:10.69.0.72:2000
```

Machine 3 (10.69.0.74):

```
./asf -nodesnumber 50 -linksnumber 200 -duration 10000 -typesnumber 2
-algorithm Clustering -algparams active=false,mode=fast,reverse=false
-topology SpiralTopology -results homogeneity,optimality,messages,usefulmessages
-starred true -startmanager true -neighbor reds-tcp:10.69.0.73:2000
```

As we have seen, only *Machine 3* is actually executing the manager algorithm and, after some time, it is the only machine to produce some output:

```
*** ALGORITHM RESULTS ***
    nodesnumber : 150
    linksnumber : 200
    typesnumber : 2
       topology : SpiralTopology
           seed : 1627062523
      algorithm : Clustering
      algparams : active=false,mode=fast,reverse=false
           time : 0
    homogeneity : 0.28
     optimality : 0.28140703517587945
       messages : 0
usefulmessages : 0
... we skip some intermediate steps here ...
*** ALGORITHM RESULTS ***
    nodesnumber : 150
    linksnumber : 200
    typesnumber : 2
       topology : SpiralTopology
           seed : 1627062523
      algorithm : Clustering
      algparams : active=false,mode=fast,reverse=false
           time : 30000
    homogeneity : 0.5778894472361809
     optimality : 0.586734693877551
       messages : 12058
usefulmessages : 0.02628960026538398
*** CONVERGENCE REACHED!  ***
*** End of Statistics ***
```

5.3.4 Interpretation of Results

The report that the algorithm prints after each execution step is a collection of the input and output parameters of the simulation. These parameters can be customized by using the `-results` command line option. In Table 5 we can see all the results supported by the simulation framework including simulation parameters and performance indexes.

Result name	Input/output	Meaning
nodesnumber	input	Total number of nodes
linksnumber	input	Total number of links
typesnumber	input	Total number of types
topology	input	Name of the topology algorithm
topologyparams	input	Parameters for the topology algorithm

seed	input	Seed used by random number generator
algorithm	input	Name of the self-organization algorithm
algparams	input	Parameters for the self-organization algorithm
time	output	Elapsed time
homogeneity	output	Homogeneity performance index
linksvariance	output	Variance of the number of neighbor among all nodes
domainsnumber	output	Number of domains
domainsaveragesize	output	Average size among all domains
domainssizevariance	output	Variance of the size among all domains
singletonsnumber	output	Number of singletons
optimality	output	Optimality performance index
roptimality	output	Reverse Optimality performance index
topologymap	output	Current Topology Map with nodes, types and links
iterations	output	Total iterations executed
failediterations	output	Total number of failed iterations
usefuliterations	output	Percentage of useful iterations
messages	output	Total number of send messages
failuremessages	output	Total number of messages involved in failed iterations
usefulmessages	output	Percentage of useful messages

Table 5: Possible simulation results

6 Bibliography

[1] **Barabasi Albert-Laszlo and Albert Reka** Emergence of Scaling in Random Networks [Rivista]. - [s.l.] : Science, 1999. - 509-512 : Vol. 286.

[2] **Dubois Daniele Joseph** Design, Development, and Simulation of Self-Organization Algorithms for Autonomic Systems [Online]. - Politecnico di Milano, Master Thesis, 20 April 2007. - `http://www.elet.polimi.it/upload/dinitto/papers/dubois.pdf`.

[3] **Saffre Fabrice [et al.]** Aggregation Algorithms, Overlay Dynamics and Implications for Self-Organised Distributed Systems. - [s.l.] : CASCADAS Project, 2006. - Vol. D3.1.