



**BISON**  
**IST-2001-38923**

*Biology-Inspired techniques for  
Self Organization in dynamic Networks*

**Implementation of basic services in AHN, P2P and Grid  
networks**

**Deliverable Number:** D06  
**Delivery Date:** December 2004  
**Classification:** Public  
**Contact Authors:** Gianni Di Caro, Frederick Ducatelle, Poul Heegaard,  
Mark Jelasity, Roberto Montemanni, Alberto Montresor  
**Document Version:** Final (February 22, 2005)  
**Contract Start Date:** 1 January 2003  
**Duration:** 36 months  
**Project Coordinator:** Università di Bologna (Italy)  
**Partners:** Telenor ASA (Norway),  
Technische Universität Dresden (Germany),  
IDSIA (Switzerland)

**Project funded by the  
European Commission under the  
Information Society Technologies  
Programme of the 5<sup>th</sup> Framework  
(1998-2002)**



### **Abstract**

This document is a companion document of Deliverable D07. In D07 we report the description and the evaluation of the algorithms and protocols developed for the basic functions (routing, topology management, collective computations and monitoring) in dynamic networks. On the other hand, in this document, we describe the characteristics of the software implementation of these same algorithms and protocols. In particular, we discuss the format of the output data and how to assign the parameters to set the network scenarios and control the behavior of the algorithms.

## Contents

<b>Introduction</b>	<b>5</b>
<b>I Routing in mobile ad hoc networks</b>	<b>7</b>
<b>1 Software implementation</b>	<b>7</b>
1.1 Utility and state functions . . . . .	8
<b>2 Usage of the algorithm</b>	<b>11</b>
2.1 Input parameters . . . . .	11
2.2 Output of the algorithm . . . . .	14
2.3 Source code . . . . .	15
<b>II Topology management</b>	<b>21</b>
<b>3 Peer sampling service in overlay networks</b>	<b>21</b>
3.1 Configuring the protocol-instances . . . . .	21
3.2 Configuring the scenarios . . . . .	21
3.3 Configuring the observers . . . . .	23
<b>4 Minimum power connectivity in wireless networks</b>	<b>25</b>
4.1 Minimum Power Topology . . . . .	25
4.2 Minimum Power Broadcast . . . . .	30
<b>III Collective computations</b>	<b>33</b>
<b>5 Aggregation in overlay networks</b>	<b>33</b>
5.1 Implementation in Peersim . . . . .	33
5.2 Implementation on Planet-Lab . . . . .	35
<b>IV Path Management and Monitoring in dynamic networks</b>	<b>37</b>
<b>6 The simulation model for path monitoring and management</b>	<b>37</b>
6.1 Simulation model general outline . . . . .	37

6.2	Input data . . . . .	40
6.3	Output from the simulator . . . . .	41
6.4	Compiling and starting the simulator . . . . .	42
6.5	Source code and parameter files . . . . .	42
<b>References</b>		<b>43</b>

## Introduction

This document contains descriptions of the software developed in the BISON project for basic functions in dynamic networks. These basic functions were first proposed in Deliverable D01 and later revised in D05. They consist of routing, topology management, collective computation and monitoring.

This Deliverable should be read in conjunction with Deliverables D05 and D07. In Deliverable D05, models were proposed for each of the basic services. Deliverable D07 contains descriptions of the algorithms which were implemented based on these original models, as well as reports of the evaluation of these algorithms according to the guidelines of Deliverable D04.

The current deliverable is focused on the practical implementation details. For each of the basic services, there is a description of the developed software and of how it should be used. In particular, a detailed overview of the inputs needed and the outputs provided by the programs is given. For each program there is also a reference to on-line resources where the software can be found.

The rest of this document is organized as follows. Part I describes the software developed for routing in mobile ad hoc networks. Part II is dedicated to topology management. This function was explored both for mobile ad hoc networks and overlay networks. Part III is about collective computation protocols for overlay networks. Part IV, finally, addresses the monitoring function in generic dynamic networks.



## Part I

# Routing in mobile ad hoc networks

In this section we describe the implementation details of the routing algorithms for mobile ad hoc networks (MANETs) described in Deliverable D07 [7]. Hereafter, we refer to these algorithms with the common name of *AntHocNet*.

## 1 Software implementation

AntHocNet has been implemented using *QualNet Developer* [16] version 3.7, a commercial network simulation package that provides a comprehensive environment for designing protocols, creating and animating experiments, and analyzing the results of those experiments. The characteristics of QualNet have been described in Deliverable D12-D13 [14].

We implemented our AntHocNet as a *network layer protocol*. The network layer is responsible for data forwarding and queuing/scheduling. The Internet Protocol (IP) resides at this layer and is responsible for packet forwarding. Routing protocols can be implemented at this layer in order to manage/update the routing tables used by the IP for data forwarding.

In QualNet, every protocol must implement three functions to interface to the Simulator: Initialization, Event dispatcher, and Finalization. Since we have used the graphical Designer tool to implement AntHocNet in the form of a *finite state machine*, the Initialization and Finalization functions have been implemented in the form of state code, while the code for the Dispatcher, that regulates state transitions, has been automatically generated by QualNet. The files `network/ants01.c` and `network/ants01.h` in the QualNet directory tree contain the full code of our protocol. It amounts to about 6500 lines of ANSI C code (including also a few lines of comments). QualNet is available on both Windows and Linux platforms and we could use it on both platforms (Windows XP and Linux Red Hat 9.0) without the need for any modification at our code when migrating from one operating system to the other.

When a protocol is written with the help of the Designer tool, the actual file which is written through the graphical interface is an XML file (`Ants01.xml` in our case) contained in the directory `gui/models` that is also the repository for both the C code and the directives for the graphical display of the finite state machine. The diagram in Figure 1 shows the AntHocNet's finite state machine. The code is composed of a total of 25 states, 82 utility functions and 2 functions to initialize and printout statistics. We tried to make the code as modular as possible without sacrificing performance. In addition to these functions that compose the protocol, we have also added 4 functions at the MAC layer (files `addons/seq/mac_802_11.c` and `addons/seq/mac_802_11.h`) in order to notify the routing protocol about dropped packets and received acknowledgments and to calculate some time averages.

Apart from some minor adaptations, we did not modify QualNet's code for what concerns the management and simulation of the node mobility and the physical, MAC, transport and application layers for MANETs. Each one of these models has a set of parameters whose values can be assigned at running time. However, in our experiments we have mostly used the QualNet's default values. Some of these parameters and their values are shown in subsection 2.1.

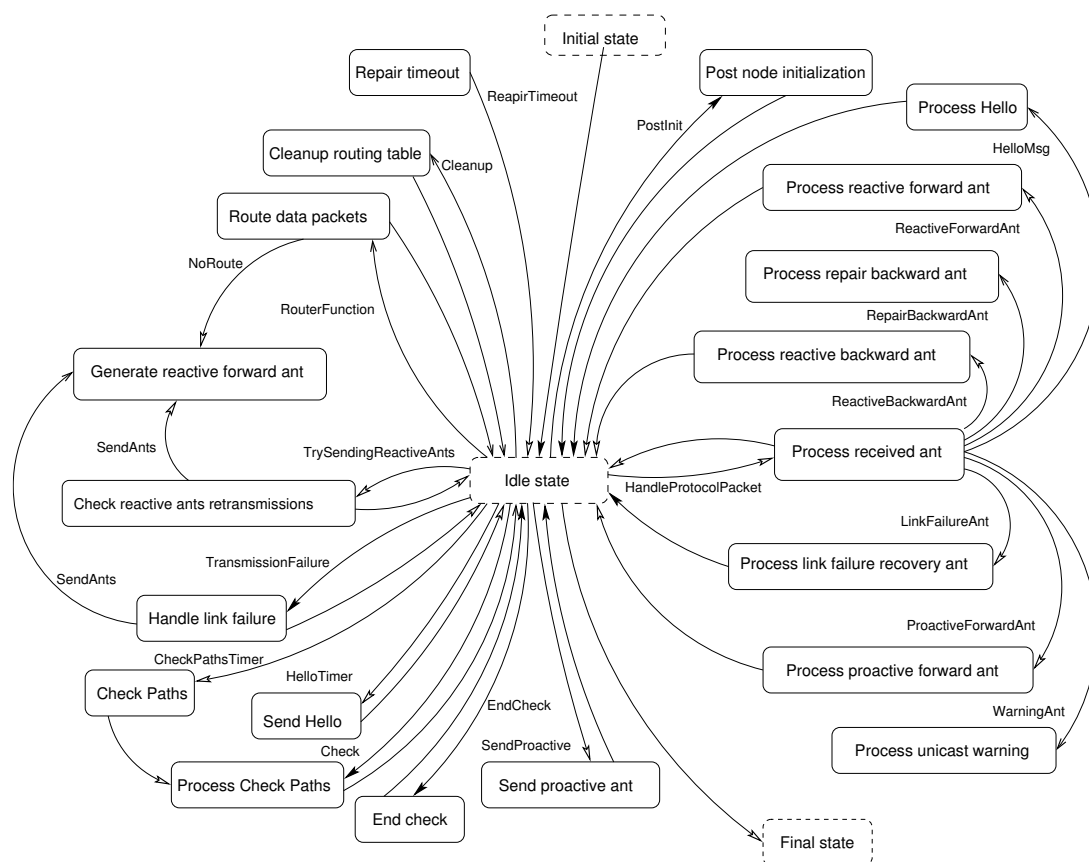


Figure 1: Finite state machine diagram representing the global behavior of AntHocNet at each network node. The labels associated to the arcs represent the conditions or the events which trigger state transitions. Arcs without label mean that after the execution of the state actions the state transition associated to the arc is automatically executed.

## 1.1 Utility and state functions

In this subsection we report a list of the utility functions used in the states' code of AntHocNet. The states with their transition structure have been reported in Figure 1. Most of the states have a self-explaining name. To each state, in the files `ants01.c` and `ants01.h` there is a corresponding function whose name is `enter` followed by the name of the protocol, which is `ANTS01`, and by the name of the state without spaces (e.g., `enterANTS01ProcessReceivedAnt`).

The behavior of the algorithm inside most of the states shown in Figure 1 is discussed Deliverable D07 [7]. A few states have been introduced only for monitoring purposes. This is the case of the states named `Check Paths`, `Process Check Path Ant`, `End Check Paths`, `Check Positions`. These states serve for the purpose of observing the *pheromone distribution* over the network for a specific source-destination pair. At this aim a sort of virtual ant (called *CheckPathAnt*) is periodically sent for each active session. The virtual ant does not consume network bandwidth and is sent from the routing layer of one node to the routing layer of a neighbor without being actually processed across the layers like a normal packet. At each

node it records the pheromone distribution for a certain destination. These data are saved on a file and input offline to a Perl script and then to a C++ program that we have written in order to provide a graphical animation of the evolution of the pheromone distribution. Figure 2 shows the screen shot of the output generated by these animation programs.

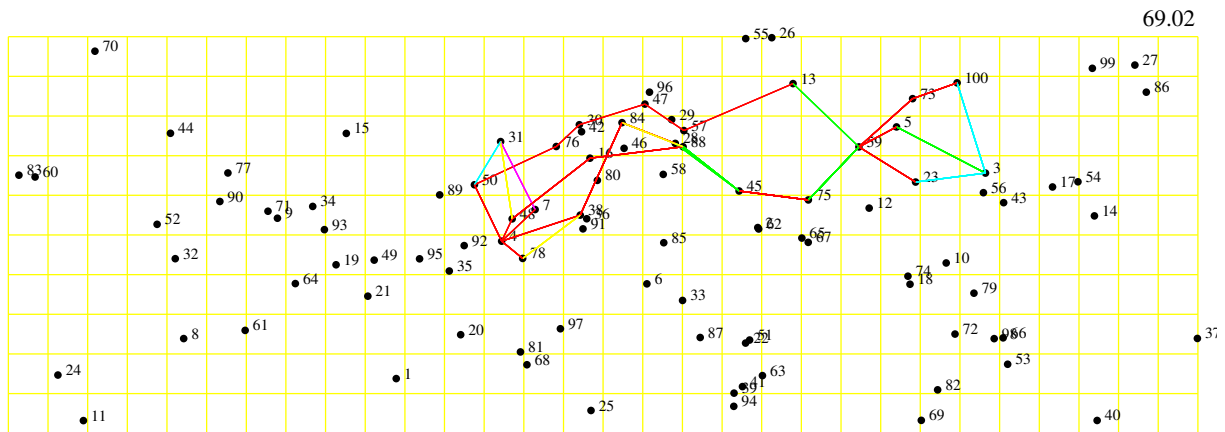


Figure 2: Screen shot of the output of our animation program showing the pheromone distribution at a certain time of the simulation (69.02 seconds in this case, as reported in the upper right corner) for the route from node 3 to 4. Different colors indicate different ranges of pheromone values, that is, different values of the estimated goodness of a next hop.

Table 1 reports the alphabetically ordered list of the utility functions (except for a few functions used only for tracing purposes). Given the rather large number of functions, we only report the name of the function and a short description of its purpose. In the actual code every function has the prefix `ANTS01`, which is the name of the protocol, but in the list we dropped this. We applied the same naming convention also to all the data structures that we have defined. In this way we could make a clear distinction among functions and structures developed for one version of the protocol but that have also been used in successive versions.

Most of the functions take as input, in addition to function-specific parameters, a pointer to the `ANTS01Data` structure which is the main data structure holding data and pointers to all the other data structures used at each node by the protocol. Each node holds a separate copy of the `ANTS01Data` (this is the normal way of proceeding in QualNet, for instance the routing protocol AODV has an `AODVData` structure, which holds all protocol-specific data). The `ANTS01Data` structure holds the pointers to the two central data structures of the protocol: the routing table and the neighbor table. Both are organized as ordered lists (based on the node address they refer to). The routing table contains a list of known destinations and for each destination a list of known next hops for the destination and the related pheromone values and relevant statistics. In order to speedup several operations (e.g., remove a next hop after a few missing hello messages), the list of the known next hops is also maintained in a separate neighbor list. Each entry in this list contains the list of the destinations that are known to be reachable through that next hop. The two tables contain pointers to each other in order to speed up their reciprocal access. Figure 3 shows the `ANTS01Data` structure. Most of the entries have self-explaining names.

Table 1: List of the main utility functions implemented in AntHocNet.

Function name	Purpose
BestPheromone	Return the best pheromone value for a specific destination
BestPheromoneForChooseHelloEntries	Select the next hops with the best pheromone to include in the hello msg
BetterBootstrappedPheromoneAvailable	Check for a destination if bootstrapped pheromone is better than regular one
BuildBackwardAntPacket	Build the packet data structure for a backward ant
BuildForwardAntPacket	Build the packet data structure for a forward ant
BuildLinkFailureAntPacket	Build the packet data structure for a link failure ant
BuildProactiveAntPacket	Build the packet data structure for a proactive ant
BuildRepairAntPacket	Build the packet data structure for a repair ant
BuildUnicastWarning	Build the packet data structure for a unicast warning
CalculatePheromone	Calculate the pheromone for the path followed by an ant
CalculatePheromoneOneHop	Calculate the pheromone value for a single hop path
CheckIfForwardAntAlreadySeen	Check if an ant from the same generation has already been seen
ChooseHelloEntries	Choose the entries to include in the hello message
CreateBackwardAnt	Create a backward ant
CreateForwardAnt	Create a forward ant
CreateNTNbrEntry	Create an entry in the neighbor table
CreateNTNbrDestEntry	Create in a destination entry in a neighbor table's entry
CreateRTDestEntry	Create a new destination entry in the routing table
CreateRTDestNextHopEntry	Create a next hop entry for a destination entry in the routing table
DecideIfProcessFAnt	Check if a forward ant should be further processed and forwarded
DecideIfProcessFAntAtDest	Check if a forward ant should be processed and returned at destination
DiscretizePheromone	Discretize a real-valued pheromone value
DeleteBackwardAnt	Delete a backward ant
DeleteForwardAnt	Delete a forward ant
EncapsulatePacketInMessage	Encapsulate a data packet in a message data structure
GetBestNextHopData	Get the next hop with the best pheromone value to forward a data packet
GetBestNextHopProactive	Get the next hop with the best pheromone value for a proactive ant
GetNextHopData	Get data information from the routing table for a specific next hop
GetNextHopProactive	Get the next hop to forward a proactive ant
GetVirtualSendTime	Calculate the estimate of the time needed to send an ant to a neighbor
GetVirtualSendTimeLocal	As the previous one but using only local data to calculate the estimate
InitiateForwardAnts	Initialize a new generation of reactive or proactive forward ants
InitiateRepairForwardAnts	Initialize a new generation of repair forward ants
InsertAndCheckBuffers	Put a packet into a waiting buffer and check for packets to send
ListCreateEntry	Create a new entry into a list structure holding nodes
ListInit	Create a new list structure holding nodes
ListInsert	Insert a new entry into a list structure
ListPrint	Print the contents of a list structure
ListRemoveEntry	Remove an entry from a list structure
ListSearch	Search for an entry within a list structure
ListSearchFrom	Search for an entry within a list structure starting from a specific entry
MACAckReceived	Notification from the MAC layer that a packet has been successfully sent
MACLayerStatusEventHandler	Notification from the MAC layer of a failed transmission
MessageBufferInsert	Insert a new packet into a waiting buffer
MessageDebufferDestWithoutSend	Dequeue and discard packets from a waiting buffer
MessageDebufferDestWithSend	Dequeue and send packets from a waiting buffer
NTRemoveEntrySimple	Remove an entry from the neighbor table
NTRemoveNbrSplitArrays	Remove neighbor data after receiving a link failure ant and forward it
PacketInLoop	Check if a packet is looping
ProcessAntAtDest	Process a forward or backward ant at destination
RefreshTimeStampsAfterHello	After receiving hello refresh related entries in routing and neighbor tables
RemoveEmptyBufferDest	Before removing a destination from routing table delete all its pending events
ResendOrBufferPacket	After removing a next hop check if the packet can be rerouted, if not buffer it
RTRemoveEmptyDest	Remove from the routing table an entry without next hops
RTRemoveNextHop	Remove a next hop for a destination and calculate the pheromone loss
RTRemoveNextHopSimple	Remove a next hop for a destination in the routing table
SendSimpleLinkFailureNotification	Send a link failure notification ant
TotalPheromone	Calculate the total sum of the pheromone available for a certain destination
RTUpdateAfterFailureMsg	Update the routing table after a failure notification
RTUpdateBootstrapping	Update bootstrapped pheromone after receiving hello message
UnpackForwardAntPacket	Unpack a forward ant packet

Table 1: (continued)

UpdateRTBackward	Update routing table after receiving a backward ant
UpdateRTDestBoth	Update all the time stamps associated to the utilization of a destination
UpdateRTDestLastDataTransported	Update the time stamp for a destination being used to forward data
UpdateRTDestLastUsed	Update the last used time stamp for a destination in the routing table
UpdateRTDestWaitingSince	Update the time stamp for the node waiting for ants to return
UpdateRTEntry	Update an entry in the routing table
UpdateRTForward	Update the routing table for the direction a forward ant is coming from
UpdateNTTable	Update the neighbor table

---

## 2 Usage of the algorithm

### 2.1 Input parameters

A network scenario can be set using either the graphical interface or a *configuration file*. In this section we only cover the aspects regarding the use of a configuration file.

The definition of a network scenario implies the specification of: (i) protocol stack and related parameters for each single node, (ii) mobility/wired model, (iii) characteristics of the node area, (iv) radio propagation and antenna models, (v) input traffic patterns, (vi) simulation time, (vii) random seeds, (viii) statistics to record, (ix) visual features, if a graphical interface is used. While it is mandatory to make explicit choices for all protocols and models, on the other hand, in most of the cases the QualNet's default values for the parameters used by these protocols/models constitute a reasonable choice. Therefore, if not strictly necessary, it is possible to rely on most of these default values.

In the configuration file, which is passed as input to the QualNet executable from the command line, selections and assignments are made according to the following general format:

[Qualifier] VARIABLE [Instance] VALUE

The qualifier field is used to specify a specific node that the parameter's value will be valid for. The instance field allows multiple values of the parameter to be specified in the experiment (no correlation to specific nodes). Both fields are optional and the instance takes precedence. The value specified for a variable can take several forms: string, integer, double, float, and clocktype. The QualNet API supplies a function for reading each variable format from the configuration file. If a new protocol has been designed using the graphical interface of the Developer tool, as is the case for AntHocNet, the input variables are defined through the graphical interface and are automatically read from the configuration file without the need to explicitly write a parsing function. Figure 5 shows a typical configuration file that we have used in our experiments. The figure's caption points out the meaning of some of the choices defined in the configuration. Most of the variables have self-explaining names. The implicit convention in assigning variable names consists in starting with the keyword that identifies the protocol or the class of models the variable refers to.

Usually, an input traffic file has also to be specified (APP-CONFIG-FILE variable). The traffic generators currently available in QualNet are CBR, FTP, FTP/GENERIC, HTTP, LOOKUP,

```

typedef struct struct_ANTS01_str {
    NodeAddress          myAddress;
    ANTS01List          routingTable;
    ANTS01List          neighbourTable;
    ANTS01StatsType     stats;
    ANTS01List          messageBuffer;
    ANTS01List          relayMessageBuffer;
    int                 state;
    unsigned short      seed[3];
    int                 initStats;
    int                 printStats;
    boolean             statsPrinted;
    clocktype          reactiveAntsRetryInterval;
    int                 reactiveAntsNumberOfAttempts;
    double              lostPheromoneTreshold;
    unsigned short      antGenerationNumber;
    double              dataRoutingExponent;
    clocktype          jitterRange;
    ANTS01List          currentConnectionsInfo;
    int                 proactiveAntCounter;
    double              proactiveAntBroadcastProbability;
    double              proactiveAntSurvivalProbability;
    double              proactiveAntRoutingExponent;
    clocktype          RTCleanupInterval;
    int                 repairMaxBroadcasting;
    clocktype          repairTimePerHop;
    double              acceptFAntHopsUnseenTreshold;
    double              acceptFAntDelayUnseenTreshold;
    double              acceptFAntDelayTresholdAtDest;
    double              acceptFAntHopsTresholdAtDest;
    int                 proactiveMaxBroadcasting;
    double              discountHops;
    double              discountTripTime;
    double              discountPheromone;
    double              timeHopsWeight;
    double              repairTimeFactor;
    int                 tempPacketCounter;
    clocktype          helloInterval;
    int                 allowedHelloLoss;
    clocktype          maxBufferTime;
    clocktype          checkPathsInterval;
    double              acceptFAntDelayUnseenTresholdAtDest;
    double              acceptFAntHopsUnseenTresholdAtDest;
    double              acceptFAntHopsTreshold;
    double              acceptFAntDelayTreshold;
    int                 maxHelloEntries;
    ANTS01HelloEntry*  tempHelloEntryArray;
    double              proactiveTriggerPercentage;
    int                 reactiveBroadcastsMin;
    int                 reactiveBroadcastsInt;
    int                 reactiveBroadcastsMax;
    ANTS01MsgId        msgIds[ANTS01_NUMBER_OF_MSG_IDS];
    short              msgIdsIndex;
    ANTS01DataPtrTempData tempData;
    int                 maxTtl;
    int                 maxDataTtl;
    FILE*              delaysFp;
    FILE*              positionsFp;
} ANTS01Data;

```

Figure 3: ANTS01Data: the main data structure of the protocol. A separate copy of this data structure is held at each network node.

MCBR, TELNET, TRAFFIC-GEN, TRAFFIC-TRACE, VBR, and VOIP. Figure 4 shows a simple example of such a file that defines 4 CBR sessions. The general format of an entry for CBR traffic session is:

```
CBR <src> <dest> <items to send> <item size>
    <interval> <start time> <end time> [tos] [RSVP-TE]
```

where, <src> is the client node, <dest> is the server node, <items to send> is how many application layer items to send, <item size> is size of each application layer item, <interval> is the interdeparture time between the application layer items, <start time> is when to start CBR during the simulation, <end time> is when to terminate CBR during the simulation, [tos] (optional) is the contents of the TOS field of the IP header, and [RSVP-TE] (optional) specifies that CBR will use RSVP-TE to send. Since each specific traffic generator has its own specific parameters, also the formats are slightly different. However, they all look similar to the CBR example just explained. Figure 4 shows the case of a simple traffic file in which four CBR sessions are specified.

```
CBR 19 17 12000 512 5S 70S 100S
CBR 11 29 10000 512 2.5S 82.49S 199S
CBR 22 1 15000 256 0.8S 91.39S 248S
CBR 15 18 10000 512 1.1S 107.8S 274S
```

Figure 4: A simple example of an input traffic file. In this case there are only 4 CBR sessions. The format is explained in the text.

AntHocNet's parameters start with the ANTS01 keyword, that identifies the protocol inside QualNet. As it is quite common for routing protocols in MANETs, there is a rather large number of parameters that can be assigned. In order to guarantee maximal flexibility and study the behavior of the algorithm under several possible conditions, we have defined a tunable input variable for each parameter used in the algorithm. In Figure 5 only a subset of the main parameters are reported. ANTS01-HELLO-INTERVAL defines the periodicity to send hello messages. ANTS01-PACKET-BUFFER-SIZE sets the size (in number of packet) of the buffers for packets waiting for a route to be found, the ANTS01-ACCEPT-FANT-HOPS-THRESHOLD-\* parameters define the thresholds to accept a forward ant during its journey toward the destination and forward it further. These parameters are used to regulate the multipath behavior during the reactive path discovery phase. For instance, if ANTS01-ACCEPT-FANT-HOPS-THRESHOLD-AT-DEST is set to 1.0, this means that if the number of hops of a forward ant is greater than 1.0 times the number of hops of the minimal number hops reported by the ants from the same generation that have passed so far through that same node, then the ant is no further forwarded. Otherwise it is forwarded (if also some other additional conditions are met, depending on the type of forward ant). ANTS01-\*-ROUTING-EXPONENT defines the value of the parameters that define the level of spreading (data or ants) across the available paths. ANTS01-REACTIVE-ANTS-RETRY-INTERVAL sets the time (in seconds) to retry to send reactive forward ants after no ant came back. The parameters referring to the proactive ants broadcast probability, survival and counter were used in the earlier versions of the algorithm to regulate the generation and propagation of the proactive ants. Also the pheromone decrease rare and decrease interval parameters are not in use anymore in the most recent version of the algorithm. They were used

to implement automatic pheromone evaporation typical of ACO algorithms. The parameters `ANTS01-REPAIR-TIME-*` regulates the time to wait for a local repair ant to come back with a new route. `ANTS01-RT-CLEANUP-INTERVAL` sets the periodicity to clean up routing tables from destinations and next hop that have not been used or heard from in the recent past. `ANTS01-MAX-BUFFER-TIME` defines the maximum number of seconds a packet can stay in the buffer waiting for a route before being discarded. `ANTS01-JITTER-RANGE` sets the maximum time jitter that is used to desynchronize node broadcasts, in order to reduce the possibility of collisions (this is a common practice in MANETs). `ANTS01-MAX-TTL` is the equivalent of the TTL parameter used in most routing algorithms: if an ant or a packet has traveled for a number of hops greater than the value associated to this parameter, it is killed.

In addition to these parameters, it is also possible to assign a value to the parameters related to the exponential averages used in AntHocNet.

## 2.2 Output of the algorithm

In QualNet, each protocol data structure (e.g., `ANTS01Data` for AntHocNet) holds a *statistic structure* (the `ANTS01StatsType stats` entry in Figure 3) that the designer can manipulate in order to add statistical variable to monitor and display. When the graphical interface of the Designer tool is used, it is rather straightforward to add statistical variables. At the end of the simulation, the `Finalization` function provides to organize and print out the value of the statistics for each single node. The typical `.stat` file resulting after running a simulation looks like the one in Figure 6. From the input configuration file it is possible to select the protocols for which statistics should be saved and printed out. QualNet provides also graphical tools to plot the results collected in the statistics file. Clearly, from these *cumulative statistics* it is straightforward to get figures of merit like delivery ratio, end-to-end packet delays, and average delay jitter.

If the simulation is run through the graphical `Animator` interface, it is also possible to define and display *dynamic statistics*, that is, the intermediate values of statistics throughout time. However, since for efficiency reasons we don't usually use the `Animator` to run simulations, we added a software module that saves on a file detailed statistics for each received packet. In this way we can offline analyze the performance of the algorithm at different time scales using our custom perl scripts. This is important to assess BISON-specific properties like adaptivity and robustness.

We defined a number of cumulative statistical variables in order to get a detailed picture of the behavior of the algorithm and of its performance and efficiency (the adopted metrics have been also discussed in general terms in Deliverable D04 [3]). In particular, we monitor (for each different type of ants): the total number of ants generated, their total number of hops (i.e., transmissions), the number of ants incurring in loops, the number of routes successfully repaired by the local repair mechanism, the number of times forward ants have to be resent because no ants were coming back, the number of ants and packets killed because of exceeded TTL, the number of forward ants killed on their way toward the destination, the average hops number incurred by packets and ants, the number of packets dropped because of no pheromone/route, the number of proactive ants that could find their way toward the destination, the total number of link failures. Since QualNet reports statistics per node, we made a Perl script that parses the

file with the statistics and outputs cumulative statistics for the whole network. The final output from this Perl script looks like the one reported in Figure 7. In addition to this we have already mentioned the fact that we can graphically analyze the evolution of the pheromone distribution throughout the network (see Figure 2) and we have also developed some scripts to follow the status of the network queues and of other variables of interest.

### 2.3 Source code

The complete source code of the AntHocNet protocol is available at the BISON's code repository at the address: <http://www.cs.unibo.it/bison/code/AntHocNet.tgz>. The tar-zipped archive contains: (i) the C source code of AntHocNet (files `ants01.c`, `ants01.h`, `ants01constants.h`), (ii) the corresponding XML file (`Ants01.xml`) automatically generated by QualNet's `Designer` tool, (iii) a few Perl scripts that allow the user to parse the output and generate data files to plot, (iv) few examples of input configuration files. A `README` file explains the organization and the use of the software. Clearly, since AntHocNet has been developed using QualNet, it is necessary to have installed QualNet in order to be able to actually compile and run the software.

```

EXPERIMENT-NAME AntHocNet_20CBRsessions
SIMULATION-TIME 900S
SEED 20
PARTITION-SCHEME AUTO
COORDINATE-SYSTEM CARTESIAN
TERRAIN-DIMENSIONS (3000,1000)
DUMMY-ALTITUDES ( 0, 0 )
TERRAIN-DATA-BOUNDARY-CHECK YES
DUMMY-NUMBER-OF-NODES 100
NODE-PLACEMENT RANDOM
MOBILITY RANDOM-WAYPOINT
MOBILITY-WP-PAUSE 30S
MOBILITY-WP-MIN-SPEED 0
MOBILITY-WP-MAX-SPEED 20
MOBILITY-POSITION-GRANULARITY 0.5
MOBILITY-GROUND-NODE YES
PROPAGATION-CHANNEL-FREQUENCY 2400000000
PROPAGATION-MODEL STATISTICAL
PROPAGATION-LIMIT -111.0
PROPAGATION-PATHLOSS-MODEL TWO-RAY
PROPAGATION-SHADOWING-SIGMA 0.0
PROPAGATION-FADING-MODEL NONE
PHY-MODEL PHY802.11b
PHY802.11-AUTO-RATE-FALLBACK NO
PHY802.11-DATA-RATE 2000000
PHY802.11b-TX-POWER-DBPSK 8.0
PHY802.11b-TX-POWER-DQPSK 8.0
PHY802.11b-TX-POWER-CCK-6 8.0
PHY802.11b-TX-POWER-CCK11 8.0
PHY802.11b-RX-SENSITIVITY-DBPSK -94.0
PHY802.11b-RX-SENSITIVITY-DQPSK -91.0
PHY802.11b-RX-SENSITIVITY-CCK-6 -87.0
PHY802.11b-RX-SENSITIVITY-CCK11 -83.0
PHY802.11b-RX-THRESHOLD-DBPSK -84.0
PHY802.11b-RX-THRESHOLD-DQPSK -81.0
PHY802.11b-RX-THRESHOLD-CCK-6 -77.0
PHY802.11b-RX-THRESHOLD-CCK11 -73.0
PHY802.11-ESTIMATED-DIRECTIONAL-ANTENNA-GAIN 15.0
PHY-RX-MODEL PHY802.11b
PHY-LISTENABLE-CHANNEL-MASK 1
PHY-LISTENING-CHANNEL-MASK 1
PHY-TEMPERATURE 290.0
PHY-NOISE-FACTOR 10.0
ANTENNA-MODEL OMNIDIRECTIONAL
ANTENNA-GAIN 0.0
ANTENNA-HEIGHT 1.5
MAC-PROTOCOL MAC802.11
MAC-802.11-DIRECTIONAL-ANTENNA-MODE NO
MAC-802.11-SHORT-PACKET-TRANSMIT-LIMIT 7
MAC-802.11-LONG-PACKET-TRANSMIT-LIMIT 4
MAC-802.11-RTS-THRESHOLD 0
MAC-PROPAGATION-DELAY 1US
PROMISCUOUS-MODE YES
NETWORK-PROTOCOL IP
IP-QUEUE-NUM-PRIORITIES 3
IP-QUEUE-PRIORITY-INPUT-QUEUE-SIZE 50000
IP-QUEUE-PRIORITY-QUEUE-SIZE 50000
IP-QUEUE-TYPE FIFO
IP-QUEUE-SCHEDULER STRICT-PRIORITY
ROUTING-PROTOCOL ANTS01
ANTS01-HELLO-INTERVAL 1
ANTS01-PACKET-BUFFER-SIZE 100
ANTS01-ACCEPT-FANT-HOPS-THRESHOLD 0.9
ANTS01-ACCEPT-FANT-DELAY-THRESHOLD 0.9
ANTS01-ACCEPT-FANT-HOPS-THRESHOLD-AT-DEST 0.9
ANTS01-ACCEPT-FANT-DELAY-THRESHOLD-AT-DEST 0.9
ANTS01-ACCEPT-FANT-HOPS-UNSEEN-THRESHOLD 2.0
ANTS01-ACCEPT-FANT-DELAY-UNSEEN-THRESHOLD 2.0
ANTS01-ACCEPT-FANT-HOPS-UNSEEN-THRESHOLD-AT-DEST 2.0
ANTS01-ACCEPT-FANT-DELAY-UNSEEN-THRESHOLD-AT-DEST 2.0
ANTS01-HOP-COUNT-STAT-WINDOW-SIZE 10
ANTS01-DATA-ROUTING-EXPONENT 2
ANTS01-REACTIVE-ANTS-RETRY-INTERVAL 0.5
ANTS01-JITTER-RANGE 0.005
ANTS01-PROACTIVE-ANT-BROADCAST-PROBABILITY 0.1
ANTS01-PROACTIVE-ANT-SURVIVAL-PROBABILITY 0.5
ANTS01-PROACTIVE-ANT-COUNTER 5
ANTS01-PHEROMONE-DECREASE-RATE 1.0
ANTS01-PHEROMONE-DECREASE-INTERVAL 10000
ANTS01-PROACTIVE-ANT-ROUTING-EXPONENT 1
ANTS01-PROACTIVE-MAX-BROADCASTING 2
ANTS01-REPAIR-TIME-PER-HOP 0.05
ANTS01-REPAIR-TIME-FACTOR 5
ANTS01-REPAIR-MAX-BROADCASTING 2
ANTS01-RT-CLEANUP-INTERVAL 5
ANTS01-MAX-BUFFER-TIME 1
ANTS01-MAX-TTL 16
ALGORITHM-SEEDS 25
STATIC-ROUTE NO
DEFAULT-ROUTE NO
TCP LITE
TCP-USE-RFC1323 NO
TCP-DELAY-ACKS YES
TCP-DELAY-SHORT-PACKETS-ACKS NO
TCP-USE-NAGLE-ALGORITHM YES
TCP-USE-KEEPALIVE-PROBES YES
TCP-USE-PUSH YES
TCP-MSS 512
TCP-SEND-BUFFER 16384
TCP-RECEIVE-BUFFER 16384
APP-CONFIG-FILE tests/CBRsessions.app
PACKET-TRACE NO
ACCESS-LIST-TRACE NO
APPLICATION-STATISTICS YES
TCP-STATISTICS NO
UDP-STATISTICS YES
ROUTING-STATISTICS YES
NETWORK-LAYER-STATISTICS YES
QUEUE-STATISTICS YES
SCHEDULER-STATISTICS YES
MAC-LAYER-STATISTICS YES
PHY-LAYER-STATISTICS YES
MOBILITY-STATISTICS YES
NODE-NAME Default
USE-NODE-ICON NO
PARTITION 0
SUBNET N8-192.0.0.0 { 1 thru 100 } Default
IP-FORWARDING NO
[1 thru 100] IP-FORWARDING YES

```

Figure 5: Example of QualNet input configuration file. The routing protocol is AntHocNet, the file `tests/20CBRsessions.app` (e.g., see Figure 4) specifies the characteristics of the input traffic for the 100 nodes network. Simulation time is 900 seconds and the node area is  $3000 \times 1000m^2$ . Nodes move according to a random waypoint model and the 802.11b is the chosen model at the physical and MAC layers. Statistics are collected for all layers.

## Implementation of basic services (Final)

```

1,      , [0],      Physical,      802.11,Signals transmitted = 9986
1,      , [0],      Physical,      802.11,Signals received and forwarded to MAC = 54334
1,      , [0],      Physical,      802.11,Signals locked on by PHY = 55744
1,      , [0],      Physical,      802.11,Signals received but with errors = 1410
1,      , [0],      Physical,      802.11,Energy consumption (in mWhr) = 225.148
1,      , [0],      MAC,          802.11MAC,Packets from network = 8683
1,      , [0],      MAC,          802.11MAC,UNICAST packets sent to channel = 583
1,      , [0],      MAC,          802.11MAC,BROADCAST packets sent to channel = 8089
1,      , [0],      MAC,          802.11MAC,UNICAST packets received clearly = 292
1,      , [0],      MAC,          802.11MAC,BROADCAST packets received clearly = 39811
1,      , [0],      MAC,          802.11DCF,Unicasts sent = 583
1,      , [0],      MAC,          802.11DCF,Broadcasts sent = 8089
1,      , [0],      MAC,          802.11DCF,Unicasts received = 292
1,      , [0],      MAC,          802.11DCF,Broadcasts received = 39811
1,      , [0],      MAC,          802.11DCF,CTS packets sent = 296
1,      , [0],      MAC,          802.11DCF,RTS packets sent = 722
1,      , [0],      MAC,          802.11DCF,ACK packets sent = 293
1,      , [0],      MAC,          802.11DCF,RTS retransmissions due to timeout = 136
1,      , [0],      MAC,          802.11DCF,Packet retransmissions due to ACK timeout = 3
1,      , [0],      MAC,          802.11DCF,Packet drops due to retransmission limit = 11
1,      ,      ,      Network,      ANTS01,Total Reactive Ants Generated = 881
1,      ,      ,      Network,      ANTS01,Total Reactive BAnts Received = 35
1,      ,      ,      Network,      ANTS01,Total Looping Ants = 13007
1,      ,      ,      Network,      ANTS01,Total Forwarded Ants = 6312
1,      ,      ,      Network,      ANTS01,Total Sending Attempts for Forward Ants = 578
1,      ,      ,      Network,      ANTS01,Total Link Failure Notifications Generated = 11
1,      ,      ,      Network,      ANTS01,Total Proactive Ants Generated = 49
1,      ,      ,      Network,      ANTS01,Total Repair Ants Generated = 0
1,      ,      ,      Network,      ANTS01,Total Routes Repaired = 0
1,      ,      ,      Network,      ANTS01,Total Routes Unrepaired = 0
1,      ,      ,      Network,      ANTS01,Total Critical Link Failures = 0
1,      ,      ,      Network,      ANTS01,Total Reactive Inter Unaccepted Ants = 15505
1,      ,      ,      Network,      ANTS01,Total Reactive Inter Unaccepted Hops = 120237
1,      ,      ,      Network,      ANTS01,Total Reactive Dest Unaccepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Reactive Dest Unaccepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Reactive Accepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Reactive Accepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Proactive Inter Unaccepted Ants = 34
1,      ,      ,      Network,      ANTS01,Total Proactive Inter Unaccepted Hops = 197
1,      ,      ,      Network,      ANTS01,Total Proactive Dest Unaccepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Proactive Dest Unaccepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Proactive Accepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Proactive Accepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Repair Inter Unaccepted Ants = 55
1,      ,      ,      Network,      ANTS01,Total Repair Inter Unaccepted Hops = 110
1,      ,      ,      Network,      ANTS01,Total Repair Dest Unaccepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Repair Dest Unaccepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Repair Accepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Repair Accepted Hops = 0
1,      ,      ,      Network,      ANTS01,Total Looping Hops = 65112
1,      ,      ,      Network,      ANTS01,Drop Data Packets No Pheromone = 0
1,      ,      ,      Network,      ANTS01,Total Packets Delivered = 0
1,      ,      ,      Network,      ANTS01,Total Hops Delivered = 0
1,      ,      ,      Network,      ANTS01,Average Hops Number = 0.000000
1,      ,      ,      Network,      ANTS01,Total Ttl Killed Ants = 341
1,      ,      ,      Network,      ANTS01,Update Exploratory High Prob = 43
1,      ,      ,      Network,      ANTS01,Update Exploratory Low Prob = 1
1,      ,      ,      Network,      ANTS01,No Update Exploratory Low Prob = 1
1,      ,      ,      Network,      ANTS01,Exploratory BAnts Received = 34

```

Figure 6: Excerpt from an example of an output file generated after running AntHocNet. In this case statistics are reported for all layers, but it is possible to select a different behavior from the input configuration file. The first column reports the node identifier to which the statistics refer to. The full output file would report the statistics for all network nodes.

```

1,      ,      ,      Network,      ANTS01,Probing BAnts Received = 14
1,      ,      ,      Network,      ANTS01,Total Probing Accepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Probing Dest Unaccepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Exploratory Accepted Ants = 0
1,      ,      ,      Network,      ANTS01,Total Exploratory Dest Unaccepted Ants = 0
1,      ,      ,      Network,      ANTS01,Exploratory Broadcast = 12
1,      ,      ,      Network,      ANTS01,Exploratory Unicast = 56
1,      ,      ,      Network,      IP,ipInReceives = 40103
1,      ,      ,      Network,      IP,ipInHdrErrors = 0
1,      ,      ,      Network,      IP,ipInForwardDatagrams = 133
1,      ,      ,      Network,      IP,ipInDelivers = 39970
1,      ,      ,      Network,      IP,ipOutRequests = 8683
1,      ,      ,      Network,      IP,ipOutDiscards = 0
1,      ,      ,      Network,      IP,ipOutNoRoutes = 0
1,      ,      ,      Network,      IP,ipReasmReqds = 0
1,      ,      ,      Network,      IP,ipReasmOKs = 0
1,      ,      ,      Network,      IP,ipReasmFails = 0
1,      ,      ,      Network,      IP,ipFragOKs = 0
1,      ,      ,      Network,      IP,ipInDelivers TTL sum = 159440
1,      ,      ,      Network,      IP,ipInDelivers TTL-based average hop count = 60.01
1,      ,      ,      Network,      IP,ipControlCallback = 0
1,      ,      ,      Network,      IP,ipDataCallback = 0
1,      192.0.0.1, [0], Network,      StrictPrio,Packets Queued = 8228
1,      192.0.0.1, [0], Network,      StrictPrio,Packets Dequeued = 8228
1,      192.0.0.1, [0], Network,      StrictPrio,Packets Dropped = 0
1,      192.0.0.1, [1], Network,      StrictPrio,Packets Queued = 455
1,      192.0.0.1, [1], Network,      StrictPrio,Packets Dequeued = 455
1,      192.0.0.1, [1], Network,      StrictPrio,Packets Dropped = 0
1,      192.0.0.1, [2], Network,      StrictPrio,Packets Queued = 0
1,      192.0.0.1, [2], Network,      StrictPrio,Packets Dequeued = 0
1,      192.0.0.1, [2], Network,      StrictPrio,Packets Dropped = 0
1,      192.0.0.1, [0], Network,      FIFO,Total Packets Queued = 8228
1,      192.0.0.1, [0], Network,      FIFO,Total Packets Dequeued = 8228
1,      192.0.0.1, [0], Network,      FIFO,Total Packets Dropped = 0
1,      192.0.0.1, [0], Network,      FIFO,Average Queue Length (bytes) = 54.789114
1,      192.0.0.1, [0], Network,      FIFO,Average Time In Queue = 0.000307354
1,      192.0.0.1, [0], Network,      FIFO,Longest Time in Queue = 0.090538004
1,      192.0.0.1, [0], Network,      FIFO,Peak Queue Size (bytes) = 512
1,      192.0.0.1, [1], Network,      FIFO,Total Packets Queued = 455
1,      192.0.0.1, [1], Network,      FIFO,Total Packets Dequeued = 455
1,      192.0.0.1, [1], Network,      FIFO,Total Packets Dropped = 0
1,      192.0.0.1, [1], Network,      FIFO,Average Queue Length (bytes) = 90.856353
1,      192.0.0.1, [1], Network,      FIFO,Average Time In Queue = 0.000303691
1,      192.0.0.1, [1], Network,      FIFO,Longest Time in Queue = 0.018846186
1,      192.0.0.1, [1], Network,      FIFO,Peak Queue Size (bytes) = 184
1,      192.0.0.1, [2], Network,      FIFO,Total Packets Queued = 0
1,      192.0.0.1, [2], Network,      FIFO,Total Packets Dequeued = 0
1,      192.0.0.1, [2], Network,      FIFO,Total Packets Dropped = 0
1,      192.0.0.1, [2], Network,      FIFO,Average Queue Length (bytes) = 0.000000
1,      192.0.0.1, [2], Network,      FIFO,Peak Queue Size (bytes) = 0
1,      ,      ,      Transport,      UDP,Packets from Application Layer = 890
1,      ,      ,      Transport,      UDP,Packets to Application Layer = 0
1,      , [1024], Application, CBR Client,Server Address = 192.0.0.2
1,      , [1024], Application, CBR Client,First Packet Sent at (s) = 10.010000000
1,      , [1024], Application, CBR Client,Last Packet Sent at (s) = 899.010000000
1,      , [1024], Application, CBR Client,Session Status = Closed
1,      , [1024], Application, CBR Client,Total Bytes Sent = 56960
1,      , [1024], Application, CBR Client,Total Packets Sent = 890
1,      , [1024], Application, CBR Client,Throughput (bits/s) = 512

```

Figure 6: (continued)

## Implementation of basic services (Final)

---

```
Delivery ratio: 0.86 (13891/16217)
Average end-to-end delay: 0.04
Average jitter: 0.38
Overhead: control forwarded versus data delivered: 22.4
Overhead: control and data forwarded versus data delivered: 27.8
Total number of link failures: 3384
Mac layer link failures: 10
Average number of hops: 5.00
Total repaired routes: 921
Total unrepaired routes: 550
Total number of link failures involving path failures: 1471
Mac layer retry ratio: 0.19
Total reactive ants generated: 2700
Total reactive forward ants resend attempts: 1193
Total reactive inter unaccepted ants: 808572
Total reactive inter unaccepted hops: 4351772
Total reactive dest unaccepted ants: 3814
Total reactive dest unaccepted hops: 33116
Total reactive accepted ants: 986
Total reactive accepted hops: 7572
Total reactive ants killed for ttl: 69
Total proactive ants generated: 2571
Total proactive inter unaccepted ants: 0
Total proactive inter unaccepted hops: 0
Total proactive dest unaccepted ants: 0
Total proactive dest unaccepted hops: 0
Total proactive accepted ants: 1649
Total proactive accepted hops: 9605
Total proactive dead end ants: 145
Total proactive dead end hops: 568
Total proactive ant send failures: 595
Total proactive looping ants: 182
Total proactive looping hops: 1037
Total exploratory accepted ants: 0
Total exploratory dest unaccepted ants: 0
Total exploratory ant unicasts: 0
Total exploratory ant broadcasts: 0
Total probing accepted ants: 0
Total probing dest unaccepted ants: 0
Total updates with exploratory ants over low probability paths: 0
Total no updates with exploratory ants over low probability paths: 0
Total updates with exploratory ants over high probability paths: 0
Total number of exploratory backward ants received at source: 0
Total number of probing backward ants received at source: 0
Total repair ants generated: 1471
Total repair inter unaccepted ants: 87241
Total repair inter unaccepted hops: 176904
Total repair dest unaccepted ants: 1793
Total repair dest unaccepted hops: 4806
Total repair accepted ants: 1262
Total repair accepted hops: 5382
Total looping ants: 131125
Total looping hops: 730761
Total number of data packets dropped for lack of pheromone: 280
Total number of data packets dropped for max ttl: 280
Fraction of data packets dropped for loops: 280
```

Figure 7: Final output for AntHocNet. The network-wide statistics for a number of variables of interest in terms of performance, efficiency and micro behavior of the protocol are reported.



## Part II

# Topology management

## 3 Peer sampling service in overlay networks

The algorithm of the random overlay management protocol, and the peer sampling service built on top of it can be found in deliverable D07. The algorithms were implemented in Peer-Sim [15], the lightweight simulator developed by the project. Deliverable D12-13 describes the simulator in detail. The URL <http://peersim.sf.net/code/middleware04-exp.tar.gz> contains the implementation of the protocol and the configuration files used to generate the evaluation results, and instructions on how to run the simulations. In the following we explain the configuration files and the output in more detail.

### 3.1 Configuring the protocol-instances

In deliverable D07 we have described a protocol space which contained several possibilities for instantiation. In particular, three parameters were examined: peer selection (rand, head, tail), view selection (rand, head, tail) and the communication model (push, pull, push-pull).

The implementation of the protocol framework is contained in class `OrderedDpvem`. This component can be added to a `peersim` simulation by defining it as a protocol:

```
protocol.0 dpvem.OrderedDpvem
protocol.0.c 30
protocol.0.order hops
protocol.0.peerselect rand
protocol.0.viewselect rand
protocol.0.push
protocol.0.pull
```

The configuration above adds a protocol with ID “0” to the simulation. Parameter `c` defines the view size, parameter `order` defines the way the age of the items in the view is measured. In the presented evaluation results, this was fixed to be the hop-count of the given item in the network, as configured above.

The instances of the protocol framework can be configured using parameter `peerselect` (peer selection), `viewselect` (view selection) and parameters `push` and `pull`: for instance, for “push” only parameter `push` has to be defined, for “push-pull” both `push` and `pull`, etc. Accordingly, the example above defines protocol (rand,rand,pushpull).

### 3.2 Configuring the scenarios

First of all, some technical parameters, and some parameters common to all scenarios have to be fixed:

```
random.seed 1234569890
simulation.shuffle
simulation.cycles 300
```

Parameter `shuffle` results in randomization in the order of activating the nodes in the network in each cycle. The given random seed was used throughout the experiments. Parameter `cycles` defines the number of cycles to be executed.

Let us turn now to the bootstrapping scenarios. We have defined three of them: starting with a random network, starting with a regular lattice and growing the network in a way that new nodes connect to the oldest node. To define the random network scenario, the following, rather self-explanatory, initializer has to be added to the configuration:

```
overlay.size 10000

init.0 WireRegularRandom
init.0.degree 30
init.0.protocol 0
```

which adds an initializer with ID “0” to the simulation that adds random links to the topology, and set network size to be 10,000. In order to define the lattice scenario, this initializer has to be replaced by

```
overlay.size 10000

init.0 WireRingLattice
init.0.k 30
init.0.protocol 0
```

Finally, to configure the growing scenario, the following has to be set:

```
overlay.size 1
overlay.maxSize 10000

dynamics.0 peersim.dynamics.GrowingNetwork
dynamics.0.add 500
dynamics.0.init.0 WireStar
dynamics.0.init.0.protocol 0
```

which results in a network initially consisting of 1 node, and growing until reaching 10,000 nodes. The “dynamics” component will add 500 new nodes in each cycle, initializing each of them with one link, pointing to the initial node.

Finally, to configure the catastrophic failure scenario, we have to add the node-removal dynamics component:

```
dynamics.0 peersim.dynamics.GrowingNetwork
dynamics.0.add -50
dynamics.0.percentage
dynamics.0.at 300
```

which removes 50% of the nodes at cycle 300. In order to be able to observe what happens afterwards, the number of cycles have to be obviously increased to eg 500.

### 3.3 Configuring the observers

We have defined several figures of merit in deliverable D04 and presented their evaluation in D07. Here we describe how to observe these figures of merit during a simulation. The observers that continuously run during the simulation are the following:

```
observer.0 peersim.reports.DegreeStats
observer.0.protocol 0
observer.0.method stats

observer.1 peersim.reports.ConnectivityObserver
observer.1.protocol 0

observer.2 peersim.reports.GraphStats
observer.2.protocol 0
observer.2.nc 100
observer.2.nl 100
```

They each result in one line for each cycle of the simulation, so the output looks like this:

```
observer.0: 38.0 82.0 10000 59.8228 30.233823542354177 1 1
observer.1: {-1=10000}
observer.2: 0.005750531432776238 2.663166316631663
observer.0: 35.0 92.0 10000 58.1144 56.82899553955354 1 1
observer.1: {-1=10000}
observer.2: 0.02968675820786031 2.7704770477047704
...
```

Observer 0 prints statistics about the degree distribution of the undirected version of the connectivity graph. Its output is generated by `peersim.utils.IncrementalStats` and accordingly, the numbers from left to right are the minimum, maximum, sample-size, average, variance, number of samples that are equal to the minimal, and maximal value.

Observer 1 examines the connectivity of the graph. Its output is a list of clusters in the form of `clusterID = clusterSize`. Cluster ID is arbitrary, although if there is a cluster with an ID different from -1, then the graph is not strongly connected. In the given example the graph is strongly connected.

Observer 2 outputs the clustering coefficient and the average path length, in this order. The number of samples to approximate the average are given by parameters `nc` and `n1`, respectively.

To perform the random node removal experiments, and examine self-healing afterwards, we need to specify the following observer:

```
observer.3 peersim.reports.RandRemoval
observer.3.protocol 0
observer.3.n 100
observer.3.at 500
```

which will perform the random node removal experiment. Parameter `n` defines the number of runs to perform to collect statistics. The experiment is done by removing all the nodes in a random order, in the following way: in the first step the network size is reduced to be 50%, and subsequently the size is reduced by 1% of the original size in each step, resulting in 50 steps altogether. For each step, the observer outputs one line containing the size of the largest cluster in the remaining network and the number of clusters. These numbers are averages over the experiments so need not be integers. This observer has to be run at the end of a simulation.

Finally, to get information on the histogram of the degree distribution, one can use the same observer component as observer 0 above, only the parameter `method` has to have the value `hist`. The histogram will be printed as a result.

## 4 Minimum power connectivity in wireless networks

In this section we describe the implementation of some minimal power topology algorithms and protocol we developed. For a detailed description of the algorithms we refer the reader to Deliverable D07 [7] and to the articles indicated separately for each algorithm presented.

In our work we consider wireless networks where individual nodes are equipped with omnidirectional antennae. Typically these nodes are also equipped with limited capacity batteries and have a restricted communication radius. Topology control is one of the most fundamental and critical issues in multi-hop wireless networks which directly affect the network performance. In wireless sensor networks, topology control essentially involves choosing the right set of transmitter power to maintain adequate network connectivity. Incorrectly designed topologies can lead to higher end-to-end delays and reduced throughput in error-prone channels. In energy-constrained networks where replacement or periodic maintenance of node batteries is not feasible, the issue is all the more critical since it directly impacts the network lifetime.

Unlike in wired networks, where a transmission from  $i$  to  $m$  generally reaches only node  $m$ , in wireless sensor networks it is possible to reach several nodes with a single transmission (this is the so-called *wireless multi-cast advantage*, see Wieselthier et al. [18]). This property is used to minimize the total transmission power required to connect all the nodes of the network.

In order to represent the problem in mathematical terms, a model for signal propagation has to be selected. We adopt the model presented in Rappaport [17]. According to this model, signal power falls as  $\frac{1}{d^\kappa}$ , where  $d$  is the distance from the transmitter to the receiver and  $\kappa$  is a environment-dependent coefficient, typically between 2 and 4. Under this model, and adopting the convention that every node has the same transmission efficiency and the same detection sensitivity threshold, the power requirement for supporting a link from node  $i$  to node  $j$ , separated by a distance  $d_{ij}$ , is then given by

$$p_{ij} = (d_{ij})^\kappa \quad (1)$$

Technological constraints on minimum and maximum transmission powers of each node are usually present. In particular they state that for each node  $i$ , its transmission power must be within the interval  $[P_i^{min}, P_i^{max}]$ .

We worked on two different problems related to power control in wireless networks. These two problems are defined in Sections 4.1 and 4.2 respectively, together with the methods we developed for them. Implementation details are also presented separately for the two problems.

The source code of the algorithms is available at the BISON's code repository at the address: <http://www.cs.unibo.it/bison/code/MinPowerConnectivity.tar.gz>. The archive contains a README file that explains the organization of the files and how to use the code.

### 4.1 Minimum Power Topology

The minimum power topology *MPT* problem can be formally described as follows. Given the set  $V$  of the nodes of the network, a *range assignment* is a function  $r : V \rightarrow \mathcal{R}^+$ . A *bidirectional link* between nodes  $i$  and  $j$  is said to be established under the range assignment  $r$  if  $r(i) \geq p_{ij}$

and  $r(j) \geq p_{ij}$ . Let now  $B(r)$  denote the set of all bidirectional links established under the range assignment  $r$ . *MPT* is the problem of finding a range assignment  $r$  minimizing  $\sum_{i \in V} r(i)$ , subject to constraints on minimum and maximum transmission powers and to the constraint that the graph  $(V, B(r))$  must be connected.

Starting from a distributed protocol presented in Glauche et al. [8], we developed a new power-aware protocol. A description of the new protocol, LPMT, can be found in Montemanni and Gambardella [12] and in Deliverable D07 [7].

The protocol has been simulated in ANSI C, using the same protocol simulation technique already adopted by Glauche et al. [8] (the protocol we will compare with).

Figure 8 illustrates the algorithmic implementation of the distributed rules in more detail. Initially, all nodes come with a minimum transmission power  $P_i = P_{min}$  and an empty neighborhood list  $\mathcal{N}_i = \emptyset$  (with the respective list of required transmission powers  $\mathcal{I}_i$  empty as well). All of them start in the receive mode. Then, at random, one of the nodes switches into the discovery mode. By subsequently sending Ask4Info messages and receiving replies, the picked node increases its power until it has discovered enough neighbors to guarantee connectivity with high probability. At this point it uses the collected information to set up the optimization problem *IP* (see below) and solves it.

Once *IP* has been solved, the optimal solution of *IP* is distributed to the set of neighbors that sent their information in order to set up problem *IP*. The head node can now set up its new transmission power  $P_i$ , its set of neighbors  $\mathcal{N}_i$  with the respective required transmission powers  $\mathcal{I}_i$ .

The other nodes will use the information received to set up their power and their new neighbor lists. The node returns then into the receive mode.

For simplicity we assume that only one node at a time is in the discovery mode; furthermore, we assume the maximum transmission power  $P_{max}$  to be sufficiently large, so that each node is able to discover at least  $n_{gb_{min}}$  neighbors.

In the receive mode a node listens to incoming Req4Info messages. Upon receipt of such a message, the node first checks whether it already belongs to the incoming neighborhood list. If yes, the requesting node has already asked before with a smaller discovery power and there is no need for the receiving node to react. Otherwise, it updates its transmission power to  $\max(P_i, P_j)$ . Then it sends back information about its neighbors and the respective transmission powers required to reach them. The node then waits for the head node  $j$  to solve *IP* and collects the results. These results are used to update transmission power  $P_i$ , the set of neighbors  $\mathcal{N}_i$  and the respective transmission powers  $\mathcal{I}_i$ .

The simulation is carried out by our main procedure **Simulate**, which receives in **input** the following data, as a list from the command line:

- $\kappa$ : it regulates signal propagation, according to equation (1);
- Network type: it defines the distribution of the nodes over the network area. The following three options are possible (we refer the reader to Glauche et al. [8] for a detailed description of these networks):
  - Homogeneous: each of the points is given a random position  $(x, y)$ . A typical real-

```

LMPT( )
 $P_i := P_i^{min};$ 
 $\mathcal{N}_i := \emptyset;$ 
ReceiveMode( );
DiscoveryMode( );
ReceiveMode( );

DiscoveryMode( )
 $P_i^{disc} := P_i^{min};$ 
 $\mathcal{N}_i^{disc} := \emptyset;$ 
 $\mathcal{I}_i := \emptyset;$ 
While (  $P_i^{disc} \leq P_i^{max}$  and  $|\mathcal{N}_i^{disc}| < n_{gb_{min}}$  )
     $P_i^{disc} := P_i^{disc}(1 + \Delta);$ 
    Ask4Info( $i, P_i^{disc}, \mathcal{N}_i^{disc}$ );
    ( $j_1, \text{info}_{j_1}, i$ ), ... := ReceiveInfo( );
     $\mathcal{N}_i^{disc} := \mathcal{N}_i^{disc} \cup \{j_1, \dots\};$ 
    Update  $\mathcal{I}_i$  according to  $\text{info}_{j_1}, \dots;$ 
EndWhile
Create  $IP$  according to  $\mathcal{I}_i;$ 
 $Sol :=$  Optimal solution of  $IP;$ 
SendSol( $i, \mathcal{N}_i^{disc}, sol$ );
Set  $P_i, \mathcal{N}_i$  and  $\mathcal{I}_i$  according to  $sol;$ 

ReceiveMode( )
( $j, P_j, \mathcal{N}_j$ ) := ReceiveReq4Info( );
If (  $i \notin \mathcal{N}_j$  )
     $P_i := \max(P_i, P_j);$ 
     $\text{info}_i :=$  combination of  $\mathcal{N}_i$  and  $\mathcal{I}_i;$ 
    SendInfo( $i, \text{info}_i, j$ );
     $sol :=$  ReceiveSol( );
    Set  $P_i, \mathcal{N}_i$  and  $\mathcal{I}_i$  according to  $sol;$ 
EndIf
    
```

Figure 8: Pseudo-code for the Local Minimum Power Topology (LMPT) protocol.

ization is illustrated in Figure 9. By definition this type of network does not show generic clustering;

- Multifractal: to construct simple clustered point patterns it is possible to employ a binary multiplicative branching process. The nonuniform probability measure supported on the network area is constructed by iteration: at first the parent square is divided into four offspring squares with area  $\frac{1}{4}$ . Two randomly chosen offsprings get a fraction  $\frac{(1+\beta)}{4}$  of the parent probability mass (1 in the beginning), whereas the remaining two get a fraction  $\frac{(1-\beta)}{4}$ . In the next iteration step each offspring square follows the same probabilistic branching rule and non-uniformly redistributes its

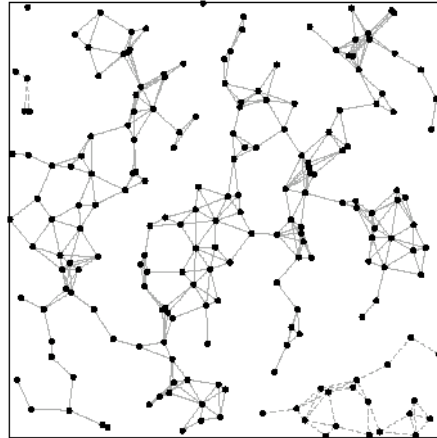


Figure 9: Realization of a homogeneous network.

probability mass onto its own four offsprings. After  $j$  iteration steps the probability has been non-uniformly subdivided onto  $4^j$  subsquares with area  $\frac{1}{4^j}$ , where  $\binom{j}{i} 2^i$  of these subsquares ( $0 \leq i \leq j$ ) come with probability  $[\frac{(1+\beta)}{4}]^i [\frac{(1-\beta)}{4}]^{j-i}$ . One after the other each of the points to be distributed is given an independent and uniform random number, which, given some probability-mass-weighted ordering of the  $4^j$  subsquares, corresponds to exactly one subsquare, onto the particle is deposited and randomly placed inside. One such realization of a point pattern is shown in Figure 10. The hierarchical clustering of points is due to the hierarchical branching structure of the iteration process;

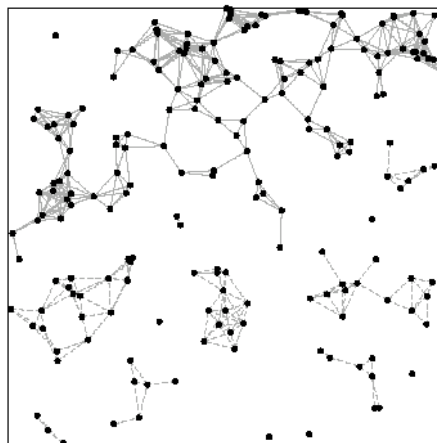


Figure 10: Realization of a multifractal network.

- Manhattan:  $N_x$  and  $N_y$  streets are equidistantly placed parallel to the  $x$ - and  $y$ -axis, respectively. One after the other each of the  $N$  points to be fitted is randomly placed onto one randomly chosen street. Fig. 1c gives an illustration of one realization;

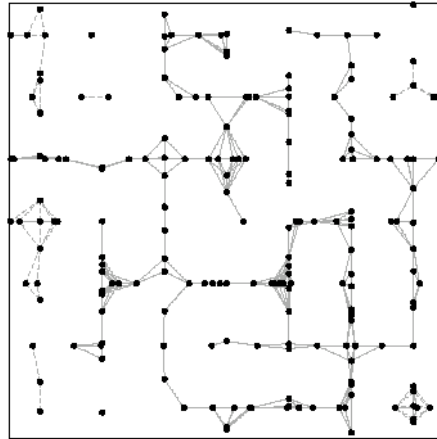


Figure 11: Realization of a Manhattan network.

- $ngb_{min}$ : this parameter regulates the minimum number of neighbors of each node forced to exist by the protocol. This parameter is originally (in Glauche et al. [8]) network dependent as soon as we look for the smallest possible value that guarantees connectivity. It is however possible to select a conservative value (e.g. 20) that guarantees connectivity almost for sure for every possible practical network. Our protocol will guarantee that the overall energy consumption is minimized, independently from this value, i.e. we could even transform it from a parameter to a protocol constant. We do not do this since we profit of it to make the execution of the protocol faster;
- Random seed: it is used to make independent, reproducible runs of the protocol.

The procedure `Simulate` starts by randomly creating the network under analysis according to the input parameters. Then all the nodes starts executing the protocol summarized in Figure 8, until all of them has defined their final transmission power (i.e. they passed through the discovery mode). At this point the procedure exits, after having provided the output. The **output** of the simulation procedure is a list of assignments according to the format defined in Figure 12, where for each node there is the transmitting power emerged from the running of the protocol together with a list of neighbors. This list is important because the phenomenon of hidden nodes, i.e. two nodes that could mutually reach each other but that don't see each other, could emerge while running the protocol. This problem is common to many distributed protocols (see for example Glauche et al. [8]) and could influence real connectivity.

Node	Power	List of neighbors
1	0.12	(2, 3, 5, ...12)
2	3.25	(1, 3, 5, 6, ...16)
3	2.78	(1, 2, 6, ...11)
...		
N	0.13	(12, 13, 14, ...18)

Figure 12: Output of the simulation of protocol LMPT (procedure Simula).

## 4.2 Minimum Power Broadcast

The minimum power broadcast (MPB) problem treated here is very similar to the problem described in Section 4.1. The difference is that here we elect a node  $s$  as source of the message we want to broadcast, and we do not require bidirectional links. We assume a fixed  $N$ -node network with a specified source node which has to broadcast a message to all other nodes in the network.

We consider a centralized implementation where the broadcasting tree is built at the source node, where complete knowledge of the locations of all nodes in the network is assumed.

The interested reader can refer to Montemanni et al. [13] for a description of the simulated annealing algorithm we developed. Information can be found also in Deliverable D06 [7].

The simulator has been implemented in ANSI C. A pseudo-code of the simulated annealing algorithm is presented in Figure 13.

The simulation is carried out by our main procedure **Simulate**, which receives in **input** the following data, as a list from the command line:

- $\kappa$ : it regulates signal propagation, according to equation (1);
- Network size: it defines the number of nodes of the network. This parameter will be used to read from file the data of the networks, that are the same used in Das et al. [4] and [5], and has been provided by Arindam K. Das.
- Network index: is used to identify the network among the fifty available for each network size considered;
- Simulated annealing parameters: they have to be introduced in the order, and with the meaning specified in Table 2;
- Random seed: it is used to make independent, reproducible runs of the algorithm.

The procedure Simulate starts by reading the network used within the simulation from file, according to the size specified as a parameter, and according to the network index specified again as a parameter. The input file has the format specified in Table 14, where for each node  $x$  and  $y$  coordinated are specified on a  $5 \times 5$  grid. Then the algorithm Simulated Annealing (see

```

SimulatedAnnealing()
 $S_O := \text{BIP}();$ 
For  $i := 1$  to  $N$ ;
    If ( $\text{rand}(0,1) < p_p$ )
        increase the power of  $i$  in solution  $S_O$ ;
 $BestS := S_O$ ;
 $t := t_{init}$ ;
 $C := 0$ ;
While ( $t < T_t$ )
    If( $C > C_T$ )
         $t := \alpha t$ ;
     $S_N := S_O$ ;
     $C := C + 1$ ;
     $i :=$  random transmitting node of  $S_N$ ;
    decrease the power of  $i$  in  $S_N$ 
        (node  $j$  is not reached anymore by  $i$ );
    If ( $S_N$  is a feasible solution)
         $S_O := S_N$ ;
    Else
         $SubT :=$  subtree of  $S_N$  not containing  $j$ ;
        If ( $\text{rand}(0,1) < p_r$ )
             $k :=$  random node in  $SubT$ ;
        Else
             $k :=$  node in  $SubT$  which reconnects
                solution  $S_N$  with the minimum
                increase in power;
        If( $\text{rand}(0,1) < e^{-\frac{\text{Cost}(S_N) - \text{Cost}(S_O)}{t}}$ )
             $S_O := S_N$ ;
    If (  $\text{Cost}(S_N) < \text{Cost}(BestS)$  )
         $BestS := S_N$ ;
         $C := 0$ ;
Return  $BestS$ ;

```

Figure 13: Simulated annealing algorithm for the *MPB* problem.

Table 2: Parameters of the Simulated Annealing algorithm.

Parameter	Meaning
$p_p$	Perturbation probability (initial solution)
$p_r$	Random selection probability for reconnection
$t_{init}$	Initial temperature
$C_T$	Iteration interval for temperature update
$\alpha$	Annealing parameter
$T_t$	Stopping criterion (temperature threshold)

Figure 13) is executed, with the parameter specified in input for Simulate. After the Simulated Annealing algorithm is finished, Simulate concludes, after having provided the output. The **output** of the simulation procedure is a list of assignments, according to the format defined in

Node	x	y
1	1.13	2.34
2	3.25	3.23
3	2.78	4.98
...		
N	0.13	4.88

Figure 14: Input of the Simulated Annealing simulation.

Figure 15, where for each node there is the transmitting power emerged from the running of the algorithm.

Node	Power
1	0.12
2	3.25
3	2.78
...	
N	0.13

Figure 15: Output of the simulation of Simulated Annealing simulation.

## Part III

# Collective computations

## 5 Aggregation in overlay networks

In Deliverable D07, we describe a decentralized protocol for distributed aggregation in large-scale overlay networks. Aggregation is a common name for a set of functions that provide a summary of some global system property. Examples of aggregation functions include network size, total free storage, maximum load, average uptime, location and intensity of hotspots, etc.

Our aggregation protocol has been implemented in two forms: first, through the Peersim simulator, to evaluate its robustness and its scalability; second, on Planet-Lab, to validate both the analytical and simulation results. The next sections provide the implementation details of both implementations.

### 5.1 Implementation in Peersim

The simulation software used to produce the figures of Deliverable D07 can be downloaded at the following address: <http://peersim.sourceforge.net/code/aggregation.tgz>.

The archive is composed by:

- `peersim-0.3.jar`, the version of Peersim that has been used to produce the results;
- `jep-2.24.jar`, a Java API for parsing and evaluating mathematical expressions, needed by the Peersim core (<http://www.singularsys.com/jep/>);
- the `src/` directory, containing the source code of the protocol;
- the `config/` directory, containing the configuration files;
- `build.xml`, an Ant file for building/executing the code (<http://ant.apache.org/>);
- `README`, a file containing instructions for compiling and executing experiments.

The `config` directory contains the configuration files that have been used to generate the Deliverable D07. Here, we describe one of these files in details, breaking it in small chunks. The goal of this specific experiment is to evaluate the behavior of our aggregation protocol in the presence of message losses. The size of the network is fixed to 100.000 nodes:

```
overlay.size 100000
overlay.node peersim.core.GeneralNode
```

As described in Section 3.2, some parameters generic to all simulations have to be fixed:

```
random.seed 1234567891
simulation.shuffle
```

This particular experiment runs for 50 epochs, whose length is 30 cycles. At the end of each epoch the values to be aggregated are re-initialized:

```
simulation.cycles EPOCH*50
EPOCH 30
```

Two protocols are configured: `Simplenewscast`, that maintains a random connected network among the nodes in the overlay, and `AverageGeneralAP`, the actual aggregation protocol:

```
DEGREE 30

protocol.0 newscast.Simple{\sf newscast}
protocol.0.cache DEGREE

protocol.1 aggregation.general.AverageGeneralAP
protocol.1.linkable 0
```

The cache parameter, which specifies the number of node identifiers maintained by `newscast` nodes, exchanged at each cycle, is initialized to `DEGREE=30`. The aggregation protocol uses the `newscast` protocol for its own gossip.

At the beginning of each epoch, the value to be aggregated stored at each node is re-initialized. The initial distribution is peak-shaped, where one nodes holds the value 1, while all the other hold a value 0. In this way, the resulting average is  $1/N$ , with  $N$  being the size of the network. This value can be used to obtain an approximation of the network size.

```
dynamics.0 peersim.vector.PeakDistribution
dynamics.0.protocol 1
dynamics.0.value 1
dynamics.0.step EPOCH
```

Initially, the caches of `newscast` nodes are initialized randomly. During the protocol execution, a connected and random network will be proactively maintained by the `newscast` protocol itself.

```
init.0 peersim.dynamics.WireRegularRandom
init.0.protocol 0
init.0.degree DEGREE
```

The experiment log is printed by the particular observer classes included in the configuration file; for this specific example, the `CountingObserver` class prints information about the current state of the simulation, including the current time, the variance observed among nodes, the convergence rate, the current average, the number of nodes that are currently working, etc.

```
observer.0 aggregation.CountingObserver
observer.0.protocol 1
observer.0.epoch EPOCH
observer.0.partial
```

The following is an example of log output:

```
observer.0 EXP 1.0    TIME 30 VAR 1.6244920869591612E-6
  RED 0.09616993154798233 RATE 0.9224278142384115 AVG 2960.0
  MAX 2147483647 MIN 80.0 CNT 148 UPNODES 5098 SIZE 10000
observer.0 EXP 1.0    TIME 60 VAR 8.488827437732943E-19
  RED 3.075519789232789E-13 RATE 0.37029590956128294 AVG 5098.0248
  MAX 5098 MIN 5097.928202820037 CNT 5024 UPNODES 5797 SIZE 10000
observer.0 EXP 1.0    TIME 90 VAR 2.057728788803435E-20
  RED 7.93942626403941E-15 RATE 0.326426818122002 AVG 5780.6134
  MAX 5780 MIN 5780.593573229793 CNT 5635 UPNODES 5794 SIZE 10000
observer.0 EXP 1.0    TIME 120 VAR 1.6659069374057485E-20
  RED 6.8561891301068315E-15 RATE 0.32477983863462656 AVG 5770.1751
  MAX 5770 MIN 5770.1595643896035 CNT 5627 UPNODES 5815 SIZE 10000
observer.0 EXP 1.0    TIME 150 VAR 1.2810332723171075E-20
  RED 5.404956808379313E-15 RATE 0.32212714108732765 AVG 5830.1898
  MAX 5830 MIN 5830.175282347604 CNT 5648 UPNODES 5827 SIZE 10000
```

The final part of this configuration file specifies the windows of message loss probability that should be simulated. This is done in two parts: through a range configuration line, the value of parameter `FAIL` is made variable between 0.00 and 0.50, with increments of 0.01. This variable is then used in the configuration of the aggregation protocol, specifying the failure probability of each message sending:

```
range.0 FAIL;0.00:0.50|0.01
protocol.1.failure.asymmetric FAIL
```

## 5.2 Implementation on Planet-Lab

In order to validate our analytical and simulation results, we implemented the `COUNT` protocol and deployed it on PlanetLab [1]. PlanetLab is an open, globally distributed platform for developing, deploying and accessing planetary-scale network services. At the time of this writing, more than 170 academic institutions and industrial research labs are members of the PlanetLab consortium, providing more than 400 nodes for experimentation.

The Planet-lab implementation has been written in C. The communication mechanism is based on UDP; this choice is motivated by the fact that in a overlay network based on a `newscast`, interactions between nodes are short-lived, so establishing a TCP connection is relatively expensive.



## Part IV

# Path Management and Monitoring in dynamic networks

## 6 The simulation model for path monitoring and management

In the following sections we discuss the implementation details of our algorithms for path monitoring and management in dynamic networks.

On the basis of the models presented in Part V, Section 6 (“Performance monitoring of routing stability in dynamic networks”) of Deliverable D05 [6], we have implemented the *CE ants* algorithm for path monitoring and management in highly dynamic networks. This algorithm is described in detail in Part IV of Deliverable D07 [7], which also includes extensive evaluation results of the algorithm. In the following sections we describe the implementation characteristics of the CE ants algorithm and those of the simulation environment that we have specifically developed for it.

### 6.1 Simulation model general outline

The simulator is implemented in DEMOS [2] using SIMULA [10]. SIMULA is an object oriented simulation programming language, while DEMOS (Discrete Event Modeling on SIMULA) is a class library that implements basic discrete event simulation concepts. The simulator is implemented using a process-oriented approach where the key elements are described by entities and the cooperation, or competition, between entities by resources, e.g. as token/semaphores. In Figure 16 the simulation outline is given. The ant nest consists of two entities, an ant generator and a receiver entity. The ants are data structures passed on from node to node entities until they are too old or have reached the destination (completed their task). Ants are then returned to the receiver. The traffic load and link and node status are controlled by separate entities. The forwarding of data structures between the node entities are controlled by a semaphore resource (token passing). Observe that the ant’s logic is inside the node entities and not in the ant packets. In the following the entities are described in more detail.

**The nodes.** The nodes model the user terminals (e.g. in ad-hoc networks), routers in wired or wireless networks, or peering points in an overlaid network (e.g. peer-to-peer). A node has the following features:

- forwarding of ants:
  - searching for the destination according to the routing probabilities - uniformly distributed while ants are exploring, and according to normalized pheromone values if not.
  - backtracking on the reversed path and updates the corresponding pheromone values



- accumulated travel time to estimate end-to-end delay when destination address is reached or to discard packet if TTL is expired.
- accumulated route probability
- control variable, i.e. temperature  $\gamma$

**The ant nest.** The ant nest consists of two types of entities where the search for a destination is originated and the updating ant is terminated.:

**ant generators** - responsible for emitting ants that are searching for a destination node.

**ant receivers** - that receive and update statistic (e.g. score value and  $\gamma$ -temperature) of the route found by the ant.

**Link status** (Topology changes) The networks structure or topology may change while the ants are searching for routes from source to destination. New nodes may appear, existing nodes may disappear, and old node may reappear.

**Node status** (Topology changes) Same as link status. A node failure is modeled by introducing failure in all links connected to this node and deleting all pheromone values.

**Traffic load** In addition to structural dynamics, changes in the traffic load will have a significant influence on the performance of the ant-based routing algorithm. The most realistic approach is to generate data traffic by sending data packets similar to the ants. However this will impose a heavy burden on the simulator and will reduce the performance dramatically. Since it is only necessary to model the influence of data traffic on the convergence and performance of the routing algorithm and not details in the data traffic itself it is possible to take an alternative approach. This idea is to model each node as an  $M/M/N_e$ -system, and then randomly split this into  $M/M/1$ -system for each of the  $N_e$  interfaces. In Figure 17 this is illustrated. Traffic is offered to node  $i$  according to a Poisson process with intensity  $\lambda_i$ . These data packets are routed to one of the  $N_e$  outgoing interfaces according to the routing probabilities  $p_{ij}^{(l)}$  where they receive a negative exponentially distributed service time with intensity  $\mu_{ij} = 1/c_{ij}$  where  $c_{ij}$  is the link rate of interface  $j$  of node  $i$ . Then, instead of modeling each data packet arrival epoch, only the ant arrival epochs are studied as an embedded Markov process. It is known from Burke's theorem [9] that in an  $M/M/N_e$ -system the departure process is Poisson. This means that the ant packets arrive each node along the route according to a Poisson process. Furthermore, it is known that the arrival, departure and a random observation process sees the same distribution of the number of customers in the system [11] and hence also the same waiting time distribution.

Under these assumptions the time through node  $i$  for an ant is the service time plus the random queuing delay sampled from the waiting time distribution for an  $M/M/1$ -system [11]. The delay distribution for interface  $j$  in node  $i$  is:

$$f_{ij}(t) = (\mu_{ij} - \lambda_i p_{ij}) e^{-(\mu_{ij} - \lambda_i p_{ij})t} \quad (2)$$

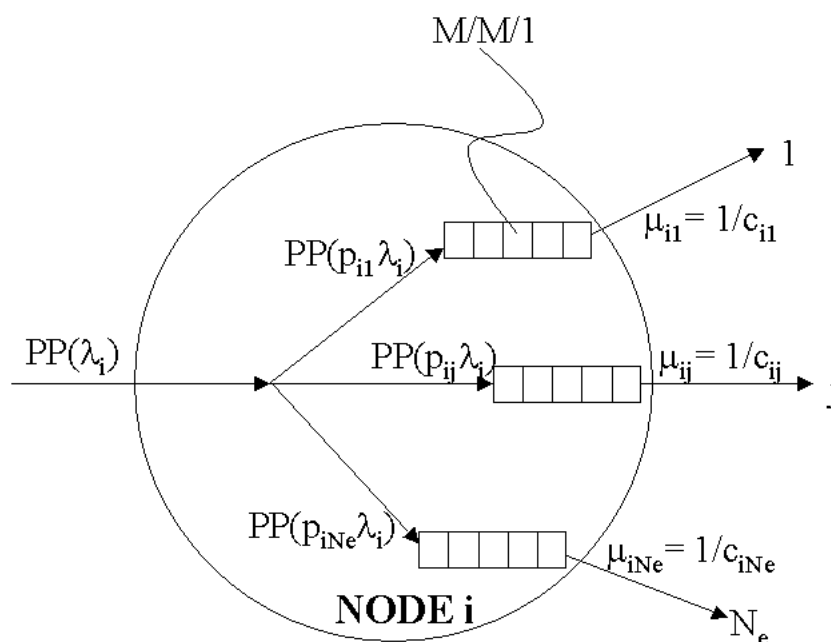


Figure 17: Simplified traffic load model

where  $\mu_{ij} = 1/c_{ij}$  is the service intensity of interface  $i$ ,  $\lambda_i$  is the arrival intensity to node  $i$ , and  $p_{ij}$  is the routing probability.

This means that when an ant arrive a node, it is routed according to the routing probabilities to interface  $i$ , and then a random delay  $D_{ij}$  is sampled from the distribution given in Eq.2.

$$D_{ij} = -\log U / (\mu_{ij} - \lambda_i p_{ij}) \quad (3)$$

where  $U \in Unif(0, 1)$  is a uniformly distributed random variate between 0 and 1.

The model's limiting assumptions are that the traffic load in each node is independent the other nodes and that the data traffic arrives according to a Poisson process.

## 6.2 Input data

The simulator take two sets of input

- Input to the CE ants algorithm (CE ant specific, run time)
- Input to describe to scenario (network topology, paths to be established, dynamics: when and what)

### Input parameters

The input to the algorithm is given in a file called “parameters” with the following format:

```
Filename:  ``parameters`` (example)
10        ! seed to the random generator;
0         ! run AntNet if 1 (no longer in use);
1         ! run CE ants if 1 (will always be ran anyway);
0         ! run original Schoonderwoerd ant algorithm (no longer in use);
200       ! number of exploration ants;
0         ! simulation time before dynamics phase;
10000    ! number of events (if simulation time=0) before dynamics phase;
0.001    ! nu - memory parameter of AntNet (no longer in use)
0.98     ! beta - the ``memory`` parameter of autoreg. ;
0.01     ! rho
0.25     ! rho2 (no longer in use);
0.01     ! Elimit (no longer in use);
0.1      ! probability of exploration ants;
0        ! search for primary and backup paths (no longer in use);
0        ! detailed data traffic (no longer in use);
0.05     ! probability of killing ants;
```

### Input scenario

The scenario is described in a file called “scenario” with the following format:

```
Filename:  ``scenario`` (example)
10        ! maximum number of edges per node;
60        ! maximum X-dimension (no longer in use);
43        ! maximum Y-dimension (no longer in use);
200       ! ``TTL`` - maximum number of hops from source to destination;
21        ! maximum number of paths to be established;
10        ! the number of nodes in the network;
1 24 15 4 3 5 7 9 12 ! format: nodeID, Xpos, Ypos, #dest, destID1, ... , nodeName
.....
9 1 40 4 1 3 5 7 52 ! ex: node 9, Xpos=12, Ypos=43, 4 dest.: from 9->1, 9->3,...;
10 12 43 0 51      ! ex: node 10, Xpos=12, Ypos=43, ... , nodeName=51
4 2 1 3 1 5 1 9 1 ! format: #neighbors: toNode linkCapacity...
.....
3 2 1 9 1 6 1     ! ex: 3 neighbors toNode=2, linkCapacity=1, ...
7                ! number of events in sequence;
! [1|2|3 <time > <number of events> ; 1=load change, 2=link failure, 3=link repair;
! ex:
1 50000000 0      ! 1=change load, sim. time > 0 ignore number of events;
0.90            ! load level;
2 50000000 0      ! 2=link failure, sim. time > 0 ignore number of events;
1 5 6            !
3 50000000 0      ! 3=link restoration, sim. time > 0 ignore number of events;
1 5 6
```

## 6.3 Output from the simulator

During the simulation events are written to a trace file called “trace-CE- $\$seed$ ” where  $\$seed$  is the seed used in this experiment. Events are ants received by the receiver entity in the ant nest, change in traffic load, link failure and repair. The ants produce a record with the following format:

```
Filename:  ``trace-CE-10`` (example)
! token, l=ID, SEQno, RECno, time, cost, temp, temp-all, path prob, tempdiff, path (seq
of nodes);
token 13 35379 957018 353.411337 4.61855 0.22166 0.22211 0.99233820 0.00205844 7> 8>
4> 2> 1
token 1 35314 957077 353.432507 3.55380 0.20129 0.20173 1.00000000 0.00216726 1> 2>
4> 3
.....
```

The trace files from repeated simulations are post-processed and filtered and merged by Perl and awk scripts and piped into gnuplot to produce appropriate graphics.

## 6.4 Compiling and starting the simulator

The simulator is just a simulation program written in a simulation language called SIMULA [10]. It is compiled by cim (<http://www.ifi.uio.no/~cim/cim.html>), which translates the SIMULA source code to c and compiles it using the c-compiler on the machine. The simulator applies a SIMULA extension (class library) called DEMOS (Discrete Event Modelling on SIMULA) [2]. The simulator is called "Bison-router.sim" and the executable file "Bison-router". The simulator expect to find two file (or links to them) "parameters" and "scenario" described above in the working directory

```
compile :  %> cim -m100 Bison-router
run      :  %> ./Bison-routerls
```

## 6.5 Source code and parameter files

The source code used in most simulations in Part IV of Deliverable D07 [7] can be found at <http://www.cs.unibo.it/bison/code/monitoring.tar.gz>. This catalog contains:

- `Bison-router.sim` - the source code implementing the model described in Section 6.1.
- `parameters` - the parameters listed in Section 6.2.
- `scenario` - the simulation scenario file from Section 6.2.

## References

- [1] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI'04)*, pages 253–266. USENIX, 2004.
- [2] G.M. Birtwistle. *A system for Discrete Event Modelling on Simula*. MacMillan, 1979.
- [3] G. Canright, A. Deutsch, G. Di Caro, F. Ducatelle, N. Ganguly, P. Heegarden, and M. Jelasity. Evaluation plan. *Internal Deliverable D04 of FET Project BISON (IST-2001-38923)*, 2004.
- [4] A.K. Das, R.J. Marks II, M. El-Sharkawi, P. Arabshahi, and A. Gray. The minimum power broadcast problem in wireless networks: an ant colony system approach. In *Proceedings of the IEEE Workshop on Wireless Communications and Networking*, 2002.
- [5] A.K. Das, R.J. Marks, M. El-Sharkawi, P. Arabshahi, and A. Gray. *r-shrink*: A heuristic for improving minimum power broadcast trees in wireless networks. In *Proceedings of the IEEE Globecom 2003 Conference, San Francisco, CA, December 1-5, 2003*.
- [6] G. Di Caro, F. Ducatelle, N. Ganguly, P. Heegarden, M. Jelasity, R. Montemanni, and A. Montresor. Models for basic services in ad-hoc, peer-to-peer and grid networks. *Internal Deliverable D05 of FET Project BISON (IST-2001-38923)*, 2003.
- [7] G. Di Caro, F. Ducatelle, P. Heegarden, M. Jelasity, R. Montemanni, and A. Montresor. Evaluation of basic services in ad-hoc, peer-to-peer and grid networks. *Internal Deliverable D07 of FET Project BISON (IST-2001-38923)*, 2004.
- [8] I. Glauche, W. Krause, R. Sollacher, and M. Greiner. Continuum percolation of wireless ad hoc communication networks. *Physica A*, 325:577–600, 2003.
- [9] J. F. Hayes. *Modeling and analysis of communication theory*. Plenum press, New York, USA, 1984.
- [10] B. Kirkerud. *Object-oriented programming with SIMULA*. Addison Wesley, 1989.
- [11] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. John Wiley and Sons, New York, USA, 1975.
- [12] R. Montemanni and L.M. Gambardella. Power-aware distributed protocol for a connectivity problem in wireless sensor networks. *Submitted for publication*, 2004.
- [13] R. Montemanni, L.M. Gambardella, and A.K. Das. The minimum power broadcast problem in wireless networks: a simulated annealing approach. In *Proceedings of the IEEE Wireless Communication & Networking Conference (WCNC 2005)*, 2005, to appear.
- [14] A. Montresor, G. Di Caro, and F. Ducatelle. Simulation environment. *Internal Deliverable D12-D13 of FET Project BISON (IST-2001-38923)*, 2004.
- [15] PeerSim. <http://peersim.sourceforge.net/>.

- [16] QualNet Simulator, Version 3.6. Scalable Network Technologies, Inc., Culver City, CA, USA, 2003. <http://stargate.ornl.gov/trb/tft.html>.
- [17] T. Rappaport. *Wireless Communications: Principles and Practices*. Prentice Hall, 1996.
- [18] J. Wieselthier, G. Nguyen, and A. Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *Proceedings of the IEEE Infocom 2000 Conference*, pages 585–594, 2000.