BISON
IST-2001-38923

Biology-Inspired techniques for
Self Organization in dynamic Networks

Implementation for Advanced Services in AHN, P2P Networks

Deliverable Number:	D09
Delivery Date:	January 2005
Classification:	Public
Contact Authors:	Andreas Deutsch, Niloy Ganguly, Tore Urnes, Geoffrey Canright, Márk Jelasity
Document Version:	Final (February 23, 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 5th Framework
(1998-2002)

Abstract

This document describes the implementation of the models that have been developed so far in BISON in relationship to the management of advanced functions in dynamic networks. This is an accompanying document to D10. In this deliverable we provide a description of the software packages produced to implement the schemes discussed and evaluated in D10. The schemes are an immune-inspired search and distributed content algorithm, chemotaxis-based load balancing and structured topology management using concepts of adhesion. Moreover, detailed instruction of how to run the software is also provided. The software packages are available at the BISON website.

Contents

1 Introduction **4**

2 Search and distributed content management using concepts from natural immune systems **4**

2.1 Implementation details (1) - disearch (Dresden immune search) package 4

2.1.1 Building up the p2p network 5

2.1.2 Experiments 7

2.1.3 Running the p2p network at each time step 11

2.1.4 Input 13

2.1.5 File arrangement 14

2.1.6 Running and compiling the files 15

2.2 Implementation details (2) - peersim-based isearch package 15

2.2.1 Configuration file 16

2.2.2 Protocol 17

2.2.3 Initializers 18

2.2.4 Observers 18

2.2.5 Basic data structure 19

2.2.6 Software package details 20

3 Chemotaxis-inspired load balancing **20**

3.1 The software 20

3.2 Network topologies 21

3.3 Initializing load distributions 21

3.4 Running simulations 22

3.5 Simple batch processing 24

4 Overlay topology construction **25**

4.1 Configuring the protocol-instances 25

4.2 Configuring the scenarios 26

4.3 Configuring the observers 27

5 Conclusion **27**

1 Introduction

This deliverable deals with the implementation of advanced functions. It is an accompanying document to D10 and D08, where we have explained the schemes and reported their experimental results.

This deliverable presents implementation details of all the three different works performed by the consortium in this direction. The works are namely (a) search and distributed content management using an immune system-based concept, (b) load balancing using the concept of chemotaxis and (c) adhesion-inspired structured topology management.

Since all the works are building services over an overlay network, most of the works have been carried out on the peersim software. The work of Dresden (search and distributed content), which started before the development of the peersim software, has however been developed on basis of equivalent home-grown software - *disearch*. The codes have then been ported to peersim and are also described.

We elaborate the implementation details one by one.

2 Search and distributed content management using concepts from natural immune systems

This scheme has been implemented on two different platforms. We first discuss the simulation software which has been initially built in Dresden. Next, we discuss its porting to peersim which was done to maintain homogeneity in implementation among various consortium partners.

2.1 Implementation details (1) - disearch (Dresden immune search) package

This section details the implementation of the scheme. The scheme is implemented in the Linux environment, although technically it should run in any other environment (with minor modifications). The language used to implement is C.

The program consists of two major modules.

- Model building. Here we build up the desired *p2p* network. Model building involves three major tasks, namely A. topology determination, B. data and query distribution, and C. determining the sequence of updating.
- Experiments. The experiments *coverage* and *time-step* implement different types of search algorithms[1].

Fig. 1 shows the general flow diagram of the program and the position of the different modules in the flow.

[1]Kindly note, in this section 'algorithm' essentially refers to the message-forwarding algorithms like random-walk, proliferation-mutation etc.

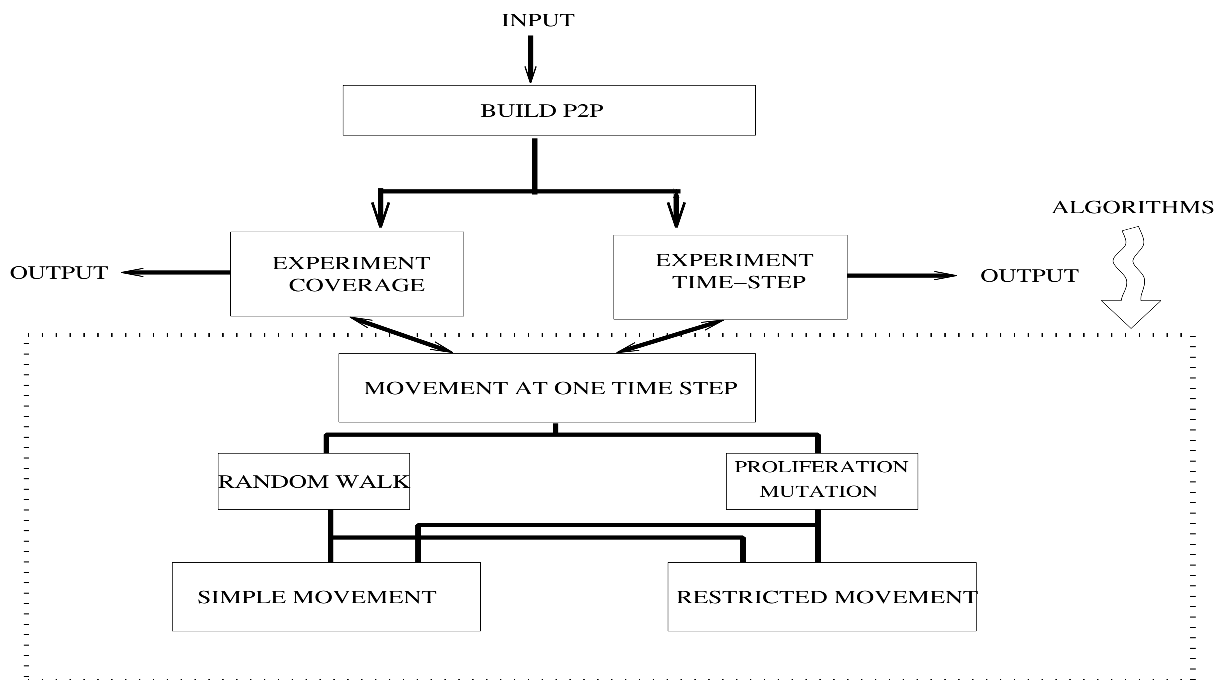

Figure 1: General flow diagram of the program and the position of the different modules in the flow.

- In the beginning the *p*2*p* network is instantiated.
- The desired experimental setup (either coverage or time-step) is chosen.
- The algorithms (random walk or proliferation-mutation) are executed.
- Note, the two experimental setups only differ in terms of the exit condition and output requirements. However, they execute the same algorithms. So, the program design is modularized in terms of one time step.
- The algorithms form two broad categories - the restricted version and the simple version.

The principal modules are now discussed one by one.

2.1.1 Building up the p2p network

Building the *p*2*p* network involves construction of the network topology, formation of the data and query distributions, and determination of the sequence in which nodes operate. Therefore, the module implements the following functions.

1. Forming the topology from a given connection list among peers.
2. Constructing content (data) for peers.
3. Building query list.

4. Setting update sequence.

1. Forming the topology

The most important task pertaining to topology formation is the assignment of neighbors of each peer. While assigning the neighbors, it should be assured that the resulting network is connected. A network is connected when there is at least a path connecting two randomly chosen nodes. The neighborhood assignment is obtained from a provided list of connections among peers. This list is generated by external *topology generators* such as BRITE [3] or Inet [2] and also internal grid *topology generators*. The function is defined as below.

```
GenerateNeighborhood(list,p2p)
Input : List of connections among peers. It is a file
                  of the format described in  Fig. 2
Output : Peers containing neighborhood information.
```

The file structure of the topology generator is discussed through *Fig. 2*. The lines starting with '#' are lines which do not appear in the files but are added for the purpose of explanation.

```
1  102
# <Line No.> <Total number of connections in the network>
2  4 19
# 4 19 means that there is a connection between peer
# number 4 and peer number 19
3  3 200
    .
    .
    .
102 120 19
103 12 197
# Note GenerateNeighborhood reads in only unique relations, that is,
# if there is information about connection between 4 19,
# then GenerateNeighborhood automatically doesn't read in 19 4.
```

Figure 2: Structure of a topology file

2. Constructing contents of the peers

The second function takes care of building the data base of the peers. We have considered two types of data representation, *simple* and more *realistic*. We have also assured that the distribution of tokens (keywords) in both of them follows Zipf's law.

The simple data representation is accomplished by the following functions.

```
InsertSimpleData(noOfbits,zipfExponent,p2p)
Input  : The length of each token in terms of number of bits
```

```
         The value of Zipf's exponent
Output : Peers with data.
```

The realistic data formation is accomplished by the following module:

```
InsertComplexData(noOfKeywords,nodeKeyword,zipfExponentp2p)
Input  : The number of unique keywords
         The average number of keywords at each node
         The value of Zipf's exponent
Output : Peers with data.
```

3. Building query list

The number of queries depends upon the number of times a particular experiment is intended to be repeated. Since the tokens of the query maintain Zipf's distribution, it is necessary to prepare the complete query list before the start of the experiment. Similar to the data distribution, here we also have to generate simple as well as realistic query lists. The query lists are generated by the following function.

```
GenerateQueryList(numberOfsearch,type,zipfExponent,queryOutput)
Input  : Number of times an experiment is repeated
         Simple or realistic
         The value of Zipf's exponent
Output : The total query list.
```

4. Setting update sequence

As mentioned above, each single node operates once in one time step, with the order of node updates following a random sequence. The initial random sequence is generated to start the network by the following function. We start with an array which has the node numbers written in order. The function *Shuffle* randomly selects n pairs of data and swaps them; n is the number of nodes. Therefore,

```
Shuffle(RandomArray)
Input : RandomArray
Output: RandomArray shuffled.
```

Once the peer-to-peer model is set up, the experiments are then performed. The experimental module is elaborated next.

2.1.2 Experiments

We perform two types of experiments - coverage and time-step. The basic structure of the two experiments is similar, only with varying exit conditions. The structure is elaborated below.

```
Experiment
{
InitializeSearch
for (each time-step; until exit condition not fulfilled)
          {
          Run the p2p network for one time step
          Collect related information
          }
}
```

The experiments are repeated for a specified number of times; we collect the information averaged over several runs. We next discuss the repetition number, exit conditions for each algorithm, and output derived from each experiment.

Coverage experiment

The coverage experiment is repeated $\mathcal{K}$ times. Each experiment runs until all the peers are not visited by the message packets. The output information consists of

1. Percentage of network covered.
2. Number of time steps.
3. Number of search items found.
4. Number of messages produced.
5. Number of operations performed. /* No of messages produced and no. of operations performed differ in any restricted version as we consider no of operations = no of messages produced + no of lookaheads performed to check whether a peer has already been visited or not. */

An example showing the output is as follows.

```
Input File Name = ../input/Rand.inp
RPM
Percent -  Turn Search Messages Operations
 21   -    19   155396    77     111
 32   -    21   231442   104     167
 43   -    23   307027   131     235
 53   -    24   381431   156     316
 63   -    25   454515   182     418
 73   -    26   526942   208     560
 83   -    28   592819   236     760
 91   -    29   657841   271   1128
100   -    41   716397   462   5579
```

The output information is the result derived from averaging the results of $\mathcal{K}$ runs.

The functions describing the coverage experiments are noted one by one. The *ExptRepeatCoverage* function performs repetition of *ExptCoverage* $\mathcal{K}$ times.

```
ExptRepeatCoverage(p2p,K,algorithm,message,
                   nodeOutput,queryOutput,exptOutput)
{
Input : The Network
        Number of times search has to be repeated
        Algorithm which has to be run
        Number of messages to be used
        Sequence of the nodes in which the search has to be initiated
        The exact query in those nodes.
Output: The output is stored in the array exptOutput. /*The output
        comprises of the fields described above.*/

for(i = 1 to K)
        {
        ExptCoverage(p2p,algorithm,message,node,query,exptResult)
        exptOutput = Avg(exptOutput,exptResult)
        }
}
```

The *ExptRepeatCoverage* calls the function *ExptCoverage*, where *ExptCoverage* is described below.

```
ExptCoverage(p2p,algorithm,message,node,query,exptResult)
{
Input :  The Network
         Algorithm which has to be run
         Number of messages to be used
         Initializing node
         Query to be processed
Output : The output is stored in the array exptOutput
         It has the same format as exptOutput.

Initialize(p2p,message,node,query,algorithm)
While(all nodes not visited by message packets)
      RunP2p(p2p,algorithm,exptResult)
}
```

The experiment *time-step* is described next.

Time-step experiment

The experiment is repeated for (say) $\mathcal{P}$ generations, where one generation is defined as a sequence of $\mathcal{Q}$ searches. That is, essentially the search is repeated $\mathcal{P} \times \mathcal{Q}$ times. Each experiment runs for a pre-defined number ($\mathcal{N}$) of time steps. We obtain output for each generation; this is of the following form.

1. Generation number.
2. Number of search items found.
3. Number of messages used.

An example showing the output is the following:

```
Input File Name = ../input/Rand1.inp
RPM
Gn. No. SearchItems  MessageUsed
   1   88.22          73.84
   2  116.66          85.38
   3   55.15         58.59
   4  144.93          96.54
   5   70.15          61.18
   6   71.56          68.76
   7  144.79         101.62
   8  156.57         105.19
   9   58.68          62.05
  10   89.62         69.66
```

The *ExptRepeatTimeStep* function takes care of the repetition of *ExptTimeStep* for $\mathcal{P} \times \mathcal{Q}$ number of times.

```
ExptRepeatTimeStep(p2p,P,Q,N,message,algorithm,nodeOutput,
                   queryOutput,exptOutput)
{
Input :  The Network
         Number of generations search has to be repeated
         Number of searches comprising a generation
         Number of time steps to be run
         Number of messages to be used
         Algorithm which has to be run
         Sequence of the nodes in which search has to be initiated
         Exact query in those nodes
Output : The output is stored in the 2-dimensional array exptOutput.
         Each row of exptOutput consists of the three fields
         - generation no., no. of search items, no. of messages used.
```

```
for(j = 1 to  P)
{
         for(i = 1 to Q)
         {
         ExptTimeStep(p2p,N,algorithm,node,query,exptResult)
         exptOutput[i] = Avg(exptOutput[i],exptResult)
         }
}
}
```

The *ExptRepeatTimeStep* calls the function *ExptTimeStep,* where *ExptTimeStep* is described below.

```
ExptTimeStep(p2p,N,algorithm,message,node,query,exptResult)
{
Input : The Network
        Number of time steps to be run
        Algorithm which has to be run
        Number of messages to be used
        Initializing node
        Query to be processed
Output :The output is stored in the array exptResult.

Initialize(p2p,message,node,query,algorithm)
for(N number of time steps)
        RunP2p(p2p,algorithm,exptResult)
}
```

We now elaborate the functions performed at each time step.

2.1.3 Running the p2p network at each time step

At each time step, all the nodes in the network check for incoming messages and perform actions upon them. The actions performed by them can be divided into three steps - (a) checking similarity between message and data; (b) forwarding the message; and (c) topology evolution. The sequence in which the nodes perform is random. The function *RunP2p* is noted below.

```
RunP2p(p2p,algorithm,exptResult)
{
Input :  The Network
         Algorithm which has to be run
Output : The output is stored in the array exptResult.

Shuffle(RandomArray)
for(each node)
```

```
        {
        k = CheckSimilarity
        N_new = DetemineNoMessage
        FowardMessage
        }
}
```

Each of the three functions are next explained one by one.

1. Similarity checking

The similarity measure differs for the two types of data distribution we have taken into consideration. So CheckSimilaritySimple returns either 0 or 1, while CheckSimilarityReal returns the actual number of keywords matched. We note the input/output of the functions below.

```
k = CheckSimilaritySimple(node.message, node.content, noOfSimilarBits)
Input :  The first message in the node's message queue
         The node's content
Output : No. of bits similar between node.message and node.content.
         /*this output is required as an input for the next function*/
         k - indicating similarity (0/1)
```

For the realistic data type, the function is:

```
k = CheckSimilarityRealistic(node.message,node.content)
Input :  The first message in the node's message queue
         The node's content
Output : k - Number of keywords found similar.
```

2. Determining the number of packets to be forwarded

The number of message packets which will be forwarded to the neighborhood depends upon the types of algorithms - random walk, or affinity-driven proliferation-mutation - we run.

```
int DetermineNoMessage(algorithm,similarity measures)
{
return N_new
}
```

Once the number N_{new} of new packets is determined, the packets are forwarded through the function described next.

3. Forwarding the message packets

Forwarding the message packets broadly falls under two categories - restricted movement, and simple or unrestricted movement. The message forward function is noted.

```
ForwardMessage(node,N_new,category,p2p)
Input : The node which forwards the message
        Number of messages to be forwarded
        The type of forwarding scheme
        (restricted, unrestricted, high-degree etc.)
Output : The forwarded message in p2p network.
{
If (category = 1)
        MessageForwardRestricted(node,N_new,category,p2p)
If (category = 2)
        MessageForwardUnrestricted(node,N_new,category,p2p)
}
```

This concludes our description of the implementation of immune-based search algorithm. We now discuss the input function.

2.1.4 Input

The input consists of various parameters which we note one by one.

1. Topology file - The file which contains the information on the topology of the peer to peer network. Fig. 2 shows the details of the file.
2. Number of nodes
3. Action Type
    - <TimeStep> / <Coverage>; <How Many Times> <Generation Length>
4. Type of data/query distribution
    - <Simple> / <Realistic>; <no. of bits> / <no of keywords>
5. <Proliferation constant> <Mutation constant>
6. <Algorithms to be performed>

We elaborate the input file configuration with the following example file. The lines starting with '#' are lines which are not in the input file, but are used here only for the purpose of explanation.

```
inet.txt
#a powerlaw graph as input.

10000
#no. of nodes

1 1000 0
#1 - indicates experiment coverage;
#1000 - indicates no.of times it is repeated;
#0 - is a dummy parameter, it is there to maintain equivalence
#with time-step experiment which requires two parameters.

1 2000
#data type - realistic and no of keywords used - 2000

0.5 0.0
#<proliferation constant>; <mutation constant>

1 0 1 0
# <RPM  executed> <PM not executed> <RW executed> <RRW not executed>
```

2.1.5 File arrangement

The files are arranged in several directories. We specify the directories and the files inside the directories one by one.

- immuneSearch/hfile
  - initial_population.h - functions to build the topology.
  - Information.h - functions to build the contents of the network and query set.
  - Movement.h - functions to guide the movement at each generation.
  - Experiment.h - functions defining the experiments.
  - small_function.h - miscellaneous functions.
  - header.h - information regarding various variables.
- immuneSearch/bin
  - immune.c - the main file to run the program.
  - makefile.
- immuneSearch/graphs - files containing topology information of different graphs.
- immuneSearch/input - files containing different possible input files.

Besides these files, there is a file grid.c which generates the grid topology in the directory TopologyGenerator/GRID.

2.1.6 Running and compiling the files

Compiling the main file immune.c can be done just by running the make file. Similarly, the files in topology generators - convert.c and grid.c - can be compiled with the following respective commands.

```
gcc grid.c -lm -o grid.out
```

Running the main program is done from the directory immuneSearch/bin with the command

```
immunesearch.out inputfilename
```

The other two executable files convertbrite.out and grid.out are run with the following commands, respectively:

```
grid.out neighbor(4/8) length breadth outputfilename
```

The equivalent implementation of *disearch* package in peersim is next discussed.

2.2 Implementation details (2) - peersim-based isearch package

As mentioned in the previous section, the implementation of the search algorithm can be roughly divided into the following steps.

- Initialization of the *p*2*p* network.
- Determination of the desired experiment (either coverage or time-step). This includes both the output as well as the exit condition.
- The algorithm(s) to be executed.

The peersim simulation environment comprises of cycles where at each cycle *protocols* are run and *observers* collect information after every cycle. Moreover, there are *initializers* to initiate the protocols. The protocols, initializers and observers are used to simulate the above mentioned steps of the search algorithm. The *initializers* are used to instantiate the *p*2*p* network. [Note, here we are reporting only the implementation of simple data/query types.] Since peersim is cycle based and runs for a specified number of cycles, the experiments (coverage or time-step) are not distinguished here. Rather, coverage is run for an arbitrary large number of time steps, so that it is ensured that all nodes are covered. The *observer* consequently outputs the same output for both experiments. Later the output is further analyzed in an excel worksheet to derive the final results.

We outline the implementation of *initializers*, *protocols* and *observers* and also the basic data structure. However, to provide the idea, how our code interacts with the peersim core code, we at first explain the input peersim configuration files which are customized to suit the purposes of the immune search scheme.

2.2.1 Configuration file

To run a search algorithm, a suitable peersim configuration file is needed. In the following we present the configuration file and go on explaining the inputs. (Note : if a line starts with #, it is a comment line)

```
 1 # PEERSIM EXAMPLE iSEARCH
 2 random.seed 1264967890
 3 simulation.cycles 21
 4 simulation.shuffle
 5
 6 overlay.size 10000
 7
 8 protocol.0 example.isearch.RProliProtocol
 9 protocol.0.ttl 50
10 protocol.0.walkers 10
11
12 init.0 peersim.dynamics.AnyTopology
13 init.0.protocol 0
14 init.0.filename TestP4_sorted
15
16 init.1 example.isearch.SearchDataInitializer
17 init.1.protocol 0
18 init.1.max_queries 100
19 init.1.data_zipf 1.0
20 init.1.query_zipf 1.0
21 init.1.keywordsWidth 10
22
23 observer.0 example.isearch.SearchObserver
24 observer.0.protocol 0
25 observer.0.verbosity 0
```

Line 2 to 6 - Global peersim configuration
Lines from 2 to 6 regard the global peersim configuration, such as the seed for the random number generator, the number of simulation cycles to perform, the node shuffle switch (to avoid picking nodes in the same order at each cycle) and the network size. Line 3 showing the number of simulation cycles to be performed is not needed for running the search algorithm, it is just a part of the historical code of peersim.

Line 8 to 10 - Protocol
The protocol actually represents different algorithms (random walk, proliferation etc.) we run. Here, in the configuration file, we define the random walk protocol (**lines 8 to 10**). The `ttl` parameter defines the number of time steps a particular search item will run. Since, in peersim, the two experiments, *time-step* and *coverage*, are not separately executed; here for running the experiment coverage, the ttl is to be set to an arbitrary large value, so that all the peers are

ultimately covered. The `walkers` define the number of message packages initially used to start an experiment.

Line 12 to 21 - Initializers
The first initializer (**lines from 12 to 14**) defines the topology. The topology can be simply read from a file containing information of connections between nodes (similar to that explained through Fig. 2). The topology file is named in the 14th line.

The second initializer (**lines from 16 to 21**) defines the way key repository and the query distribution are to be initialized at each node. The parameters are **max_queries** - the number of query operations performed for experiment; **data_zipf** and **query_zipf** - the Zipf's coefficient for data and query, respectively; **keywordsWidth** - the width of the token used for query. (Since we are working with simple data sets, each data or token is k bits wide and we have 2^k unique tokens.)

Line 23 to 25 - Observers
Finally, the observer is defined from **line 24 to 26**. The observers output the result at the end of search of each individual cycle or at the completion of a single search operation (defined by verbosity - line 25).

The protocol, initializers and observers are explained one by one.

2.2.2 Protocol

As mentioned protocols essentially implement the algorithms - random walk and proliferation, both their normal and restricted version. All the protocols have some common basic services. An abstract class `SearchProtocol` provides a first common implementation of those services; all `isearch` protocols are supposed to inherit from this class.

Usually, each protocol is made of two distinct parts: *active* and *passive*; the former represents the pro-active behavior (e.g.: nodes inject queries into the system according to a predefined distribution), and the latter reacts to the incoming messages according to the specific protocol behavior. *send* and *forward* are the common metaphors used to provide these behaviors. The *match* function returns the amount of match between the contents of the node and the content of the incoming token. Each node therefore performs the work of *send* and *forward* at each cycle through the function *nextcycle*.

Individual protocols on top of the framework

Each distinct protocol, random walk (RW), restricted random walk (RRW), proliferation ($Proli$) and restricted random walk ($RProli$), extends the basic `SearchProtocol` and customizes the program according to its underlying guiding algorithm. An example of restricted random walk will clarify this.

A restricted random walk differs from the simple random walk only in the strategy with which a neighbor is chosen. The node neighbors are probed to find a candidate which has never seen before the message that is going to be sent. If there isn't such a candidate, then a random node is chosen as in the random walk protocol. An outline of the RRWprotocol class which extends the standard version (RWProtocol) is given below.

```
public class RRWProtocol extends RWProtocol {
    public RRWProtocol(String prefix, Object obj)  {
        super(prefix, obj);
    }

    public void process(SMessage mes) {
        // checks for hits and notifies originator if any:
        boolean match = this.match(mes.payload);
        if (match) this.notifyOriginator(mes);

        // forwards the message to a random FREE neighbor:
        Node neighbor = this.selectFreeNeighbor(mes);
        this.forward(neighbor, mes);
    }
}
```

The only difference between random walk and restricted random walk is the way the `process()` method adopts to forward the messages. A specific function (`selectFreeNeighbor()` member function of the basic `SearchProtocol` class) supports the restricted protocol version and chooses a neighbor according to the restricted strategy.

2.2.3 Initializers

The model initializers - topology as well as data and query distribution - perform exactly the same way as in *disearch* package. The only major difference in design is that in *disearch*, the initializers are considered as a separate entity (independent of protocol), whereas, in peersim the topology and the distributions are attached to a specific protocol, here the search protocol.

2.2.4 Observers

The generation of the query statistic is made by a peersim observer interface object. At the end of each cycle the observer runs and collects the data about the packets stored at each node and generates statistics. At the end of each cycle or at the end of the whole simulation, the observer generates the following data tuples about query packets:

$$\begin{pmatrix} queryID \\ message\ TTL \\ number\ of\ nodes\ the\ query\ has\ visited \\ number\ of\ successful\ hits \\ total\ number\ of\ message\ packets\ used\ for\ this\ query \end{pmatrix}$$

The observer produces an output at the end of each cycle when verbosity = 1. Below we show a sample output of the 7th and 8th query items while running `Proliprotocol`. The number of time steps (ttl) to be executed is 50.

```
.
.
.
7 46 5415 723 7570
7 47 5718 778 8331
7 48 6001 808 9106
7 49 6286 840 9896
8 0 1 0 1
8 1 2 0 2
8 2 3 0 3
8 3 6 2 6
.
.
.
```

If the verbosity = 0, the output is shown at the end of each search operation, The sample output shows the output produced by the 30th to 34th query items while running `Proliprotocol`. The ttl here is 50.

```
.
.
.
30 49 38 0 63
31 49 780 56 13243
32 49 161 1 989
33 49 328 7 3475
34 49 240 2 385
.
.
.
```

We now give a brief outline of the data structure used by each node which has enabled us to produce the desired output.

2.2.5 Basic data structure

To run the above mentioned initializers, observers and protocols on the network, the data structures needed by each node in the infrastructure are the following:

- **messageTable**: hashtable that maps a packet to an integer; it represents the number of times the current node has seen this packet before.
- **hitTable**: hashset that stores the packets for which the current node reports a query hit.
- **incomingQueue**: list that buffers the incoming packets; it is accessed in a FIFO fashion.

- **view**: the current node neighbor list view. It is managed by the linkable interface methods. (Linkable interface method is a facility provided by peersim).
- **keyStorage**: hashmap mapping a key to an integer; the latter represents the frequency of the key.
- **queryDistro**: treeset mapping integers to key array; the former represents the cycle in which the query is scheduled and the latter represents the packet query payload.

2.2.6 Software package details

The peersim simulation platform is available for download from `http://peersim.sourceforge.org/`. The isearch software should be installed into the `example` directory of the peersim installation. Moreover, the file AnyTopology.java should be installed in the peersim/dynamics directory and the file Simulator.java to be installed in peersim/cdsim.

However, one can download the entire package (peersim + search) with all the files pre-arranged from the BISON website. Our software consists of *peersim* configuration files, topology data files, and Java source code files. The source files are located in subdirectories.

To run the config file, assuming the peersim classes presence in the CLASSPATH, just type:

```
java peersim.Simulator <path-to-configfile>/config-isearch.txt
```

where config-isearch.txt is a *peersim* configuration file.

3 Chemotaxis-inspired load balancing

This section describes the implementation of chemotaxis-inspired load balancing based upon the Peersim simulation platform [1]. We present an overview of the implementation details and describe how to run the software in the following subsections.

3.1 The software

Our simulation code is based on an old version of Peersim that is no longer available at the Peersim site (http://peersim.sourceforge.org/). Unfortunately, new and old versions of Peersim are not 100% compatible and we therefore bundle with our code the old version of Peersim that our code is based on.

The software tar-file un-packs into a directory called `Peersim`. All the chemotaxis code is located in the `example` sub-directory.

Our software consists of Peersim configuration files, various script files, topology data files, initial load distribution data files, and Java source code files. The source files are located in subdirectories.

Peersim experiments are configured using the Peersim configuration language [1]. A configuration specification is submitted to Peersim in a configuration file. A configuration lists all simulation modules and allows parameters to be supplied for each module.

The `run` script is used to run a configuration file, i.e., to run a Peersim simulation. The `make` script (located in the `Peersim` directory) compiles all the Java source code files.

3.2 Network topologies

The 10.000 node power-law topology used in all our simulations is stored in the file `inet`. To generate other power-law topologies, use the `toplogy-config.txt` Peersim configuration file whose content we list here for convenience:

```
random.seed 1234567890
simulation.cycles 1

overlay.size 10000
overlay.maxSize 10000

protocol.0 example.topology.InetNodeProtocol

init.0 example.topology.InetInitializer
init.0.alfa 4.0
init.0.protocol 0
init.0.d 2
```

The resulting topology is stored in a file with a filename reflecting the setting for node outdegree (d) and geometric distance influence (alfa), e.g., `inet` for the settings in the above configuration file.

To use a generated topology in a Peersim simulation, simply include the *TopologyInitializer* initialization module in the Peersim configuration file. The following configuration file code assumes the existence of a topology module with index 0 (notice that topology modules are just special protocol modules that implement the *Linkable* interface) and reads the topology data from the `inet` file:

```
init.0 example.topology.TopologyInitializer
init.0.protocol 0
init.0.filename inet
```

3.3 Initializing load distributions

Any protocol module implementing the *Loaded* interface can use our *LoadAndCapacityInitializer* initializer module to assign an initial load distribution to the nodes in an overlay network. The following example shows the configuration file code for placing the load of 10.000 on node 0 and giving each node a capacity of 1.0:

```
init.1 example.diffusion.LoadAndCapacityInitializer
init.1.protocol 1
init.1.algorithm 0
init.1.load 10000
init.1.loaded_node 0
init.1.capacity 1.0
```

Notice that in the above example the protocol module in question has index 1.

The *LoadAndCapacityInitializer* initializer module implements three load distribution algorithms. All three algorithms take total load and capacity per node as input parameters. Algorithm 0, chosen in the above example, places all the load on the node specified by the `loaded_node` parameter. Algorithm 1 distributes the load randomly, while algorithm 2 distributes the load evenly but slightly off.

The result of executing any load distribution algorithm can be saved to a file using the `save_to` parameter:

```
init.1.save_to distribution1size10000.init
```

The following configuration file excerpt shows how to initialize the nodes in an overlay network with load values read from the file `distribution1size10000.init`:

```
init.1 example.diffusion.LoadAndCapacityInitializer
init.1.load_from distribution1size10000.init
init.1.protocol 1
init.1.capacity 1
```

3.4 Running simulations

As mentioned above, a `run` script located in the `example` directory is used to run a simulation. The script takes a configuration file as input. A simulation run generates a trace of output from observer modules detailing the simulation state at each cycle. Typically the trace is piped to a file.

Following a simulation run, data may be extracted from the trace for analysis. To extract data about changes in load distribution use the Perl scripts `get-load-diffusion` (for basic diffusion simulations) and `get-load-chemotaxis` (for chemotaxis simulations). The `get-load-moved` script extracts data about load movement amounts.

Running basic diffusion simulation experiments. Configuration code for basic diffusion simulations is found in the file `diffusion-config.txt`. A configuration file parameter specifies which protocol version to use. Similarly, the diffusion constant to use is also read from the configuration file. The following example specifies a simulation using diffusion method version 6 (in addition to using a diffusion constant of $1/256$):

```
protocol.1 example.diffusion.Diffusion
```

```
protocol.1.linkable 0
protocol.1.inverse_c 256
protocol.1.version 6
```

Running chemotaxis simulation experiments. Configuration code for chemotaxis simulations is found in the file `chemotaxis-config.txt`. Both signal and load diffusion protocols can be parameterized through configuration file parameters. For signal diffusion, a `version` parameter offers a choice of protocol: either diffusion method 6 or diffusion method 10. If diffusion method 6 is chosen as signal diffusion protocol, then the diffusion constant may be set using the `inverse_c` parameter. Load diffusion speed is governed by an `inverse_c3` diffusion constant parameter. The following example specifies a simulation using diffusion method 6 with a diffusion constant of $1/1$ for signal diffusion and load diffusion with a diffusion constant of $1/2500$ (note how signal and load modules are linked through the `load_protocol` and `signal_protocol` protocol reference parameters):

```
# Signal
protocol.1 example.chemotaxis.SignalEmitter
protocol.1.linkable 0
protocol.1.load_protocol 2
protocol.1.version 6
protocol.1.inverse_c 1

# Load
protocol.2 example.chemotaxis.Chemotaxis
protocol.2.linkable 0
protocol.2.signal_protocol 1
protocol.2.inverse_c3 2500
```

Both basic diffusion and chemotaxis simulations use the *LoadObserver* observer module. It takes the load balancing criterion threshold as a parameter (named `threshold`). The *LoadObserver* observer module reports the minimum, average, and maximum load values for the overlay network at the end of each simulation cycle. Load movement statistics are also gathered and reported by the *LoadObserver* module. It can also output the load and signal values for a range of nodes. The node range is specified using parameters `start_index` and `stop_index`. The following configuration code example specifies a threshold of 0.1 and a single-node node range consisting of node 0:

```
observer.0 example.diffusion.LoadObserver
observer.0.protocol 2
observer.0.start_index 0
observer.0.stop_index 0
observer.0.threshold 0.1
```

The *LoadObserver* writes output to standard output which can be piped to a file. Simple scripts can then be used to extract and format data in a way suitable for plotting.

3.5 Simple batch processing

Running many simulations can be a tedious task. Prior to each run, the configuration file must be edited and after each run, scripts must be applied to the simulation trace to extract data for analysis. Also, for each run, plots of the data need to be generated.

In order to automate the process of running multiple simulations we have made a simple batch processing framework that automates all the said tasks. The user only needs to prepare a single file listing a set of configuration parameters and the range of values they may take. These parameters are called *variables* and the file containing the list of variables is called the *variables file*.

Using our batch processing framework, the following list of variables and their values will produce nine simulation runs:

```
protocol.1.version    v  10  6
protocol.1.inverse_c  c  1   1   2   4   8  16  32  64 128
```

Each line in the variables file specifies one variable. The line starts with the name of the variable which corresponds to the name of a Peersim configuration file parameter whose value changes between simulation runs. The second item on each line is the short form of the variable name. The short form is used when generating filenames for simulation traces and extracted data. The remaining items specify values that the variables may take on.

The variables file can be thought of as a matrix where rows specify variables and columns (beyond column two) specify the values assigned to variables for a particular simulation run. In the above example, the first run will generate a Peersim configuration file with parameter `protocol.1.version` assigned the value 10 and parameter `protocol.1.inverse_c` assigned the value 1. On the second run, the two parameters corresponding to the variables will be assigned values 6 and 1, while on the third run the values 6 and 2 will be assigned. Note that if a variable is specified with fewer values than another variable, the former variable will simply reuse the last value for those simulation runs where no value is specified explicitly. The ninth run will use the values 6 and 128 to the two variables.

The batch processing framework is implemented as a script written in the Perl language. The name of the script file is `batch.pl` and is located in the `example` directory. The script takes four parameters:

1. The name of the simulation run. The simulation name becomes part of every filename generated by the batch processing framework.
2. The name of the base configuration file. The base configuration file is a regular Peersim configuration file. Each variable must appear in the base configuration file as a normal, constant-value parameter, but no values need to be specified for those parameters.
3. The name of the variables file.
4. The name of a pre-existing directory that will hold generated files after the completion of all simulation runs.

The following is an example of how the batch processing script may be invoked from the operating system's command shell with the Perl interpreter available as an executable:

```
C:\Peersim\example> perl batch.pl chemotaxis-random-init \
 chemotaxis-config.txt run1.variables results-run1
```

Each simulation run performed by the batch processing framework results in a trace file containing all data output by the observer modules. Also, two load balance-specific files are generated: a trace contains load value statistics (minimum, average, and maximum load values over the entire network) for each cycle and a trace containing load movement statistics (total load moved so far, largest load amount moved in one cycle so far, and largest load amount moved during current cycle) for each cycle. Following the completion of the last simulation run, the batch processing framework compiles a report listing for each run the total number of cycles until completion in addition to summaries of various load balancing statistics. Finally, a script file with Gnuplot commands for plotting all data traces is generated. All generated files are given descriptive file names that clearly identify their content.

4 Overlay topology construction

The algorithm of the overlay management protocol T-MAN can be found in deliverable D10. The algorithms were implemented in PeerSim [4], the lightweight simulator developed by the project. Deliverable D12-13 describes the simulator in detail. The URL `http://peersim.sf.net/code/UBLCS-2004-7.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.

4.1 Configuring the protocol-instances

In deliverable D10 we have described a protocol space which contained several possibilities for instantiation.

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

```
protocol.0 topman.Topology
protocol.0.dist topman.LinearDistance
protocol.0.dist.loop
protocol.0.c 20
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 `dist` defines the distance function that is applied over the ID space (see below). The value of parameter `dist` is a classname, and it can have parameters itself.

In the example, distance class `LinearDistance` has a parameter `loop` which means that it will define a ring rather than a line. To define the other distance functions mentioned in D10, one must define `Grid2dDistance` or `BinaryDistance`. The former also understands the parameter `loop` in which case it will define a torus. The latter needs one parameter of its own: `bits`. It defines the number of bits in the ID, that are used the calculate the distance function that defines the binary tree.

Parameters `push` and `pull` define the symmetric gossip communication model, and are fixed throughout all the simulations.

4.2 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 100
```

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 scenario. We ran the experiments over a topology generated by `newscast`, one of the protocols described in D07. We did this in two steps. First, the topologies were generated in a separate experiment, saved to a file. To run newscast, please refer to D06. To save a topology to a file, you should add the following observer component:

```
observer.0 peersim.reports.GraphPrinter
observer.0.outf ncast
observer.0.until 0
observer.0.FINAL
observer.0.protocol 0
```

which will save the topology at the end of the simulation into file `ncast<cycle>.graph` where `<cycle>` is the id of the last cycle.

Second, this topology was loaded from the file for the T-MAN experiments. Loading a topology from a file is performed by the following initializer component:

```
init.0 peersim.dynamics.WireFromFile
init.0.file ncast49.graph
init.0.protocol 0
```

which adds an initializer with ID "0" to the simulation that loads the topology. Note that it would have been possible to do this without involving saving and loading topologies from files, but this solution allows from greater flexibility, since many experiments can be done without always re-running newscast.

4.3 Configuring the observers

We have defined several figures of merit in deliverable D04 and presented their evaluation in D10. 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 topman.TopologyObserver
observer.0.protocol 0

observer.1 topman.DistDistObserver
observer.1.protocol 0
observer.1.maxdist 80
```

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

```
observer.0: 61 1.0 362.0 2620355 181.39148588645432 5450.4144383736375 1.0 168.0
 131044 50.22477946338634 647.3755095360038
observer.1: 0 61 150 228 277 437 456 544 633 661 832 925 920 1034 1113 1183 1248
 1362 1450 1506 1532 1638 1699 1840 1871 1975 2017 2123 2195 2313 2295 2434 2490
 2690 2732 2655 2866 2878 2983 3124 3173 3295 3312 3348 3548 3615 3584 3637 3764
 3888 4037 4020 4144 4154 4302 4444 4358 4473 4651 4839 4757 4752 4953 4915 5212
 5190 5466 5178 5464 5300 5464 5581 5659 5612 5827 6047 5990 6074 6101 6242 6232
...
```

where lines are broken for display here.

Observer 0 prints statistics about the distance of a node and its neighbors. The first number denotes the number of links in the entire network that are "optimal", that is, part of the target topology. The next $2 \times 5 = 10$ numbers are the output of two instances of class `IncrementalStats`. The first one collects statistics over the distances associated to all the links in the entire network, resulting in a sample size of the number of links in the network, which is exactly the number of nodes times the view size (20 in the example above). The second one collects statistics only over the distances that are minimal among the neighbor set of a node, resulting in a sample size of the number of nodes. The numbers that belong to one 5-tuple are the minimum, maximum, number of samples, average and variance, respectively.

Observer 1 prints frequency information. The format is as follows: the ith number represents the number of links that have i as distance associated to them, indexing starts from zero. There are no loop edges so the first number (the 0th) is always 0. Parameter `maxdist` defines what is the maximal distance to collect statistics on.

5 Conclusion

In this deliverable we have outlined the implementation details of the three works presented in D10. The software packages are available at the BISON website in tar format. To use the packages, download the corresponding tar files and unpack it. The peersim-related packages can be run on any operating systems as long as java is installed. The *disearch* package runs

on linux, although with possible trivial changes, it can be run in any other environment. Each software package has a readme file in the home directory, which provides details of running the programmes. The readme also contains the name and e-mail address of the person responsible for maintaining the software. Please contact the respective persons, if you face any problem while handling the software.

References

[1] Gian Paolo Jesi. PEERSIM HOWTO: build a new protocol for the peersim simulation framework. http://peersim.sourceforge.net/tutorial1/tutorial1.html, November 2004.

[2] C Jin, Q Chen, and S Jamin. Inet: Internet Topology Generator. University of Michigan Technical Report CSE-TR-433-00, 2002.

[3] A Medina, A Lakhina, I Matta, and J Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS*, August 2001.

[4] PeerSim. http://peersim.sourceforge.net/.