Computer Science • 14 (1) 2013 http://dx.doi.org/10.7494/csci.2013.14.1.153

Aleksander Byrski
Michał Feluś
Jakub Gawlik
Rafał Jasica
Paweł Kobak
Grzegorz Jankowski
Edward Nawarecki
Michał Wroczyński
Przemysław Majewski
Tomasz Krupa
Jacek Strychalski

VOLUNTEER COMPUTING SIMULATION USING REPAST AND MASON

Abstract *Volunteer environments usually consist of a large number of computing nodes, with highly dynamic characteristics, therefore reliable models for a planning of the whole computing are highly desired. An easy to implement approach to modelling and simulation of such environments may employ agent-based universal simulation frameworks, such as RePast or MASON. In the course of the paper the above-mentioned simulation frameworks are adapted to support simulation of volunteer computing. After giving implementation details, selected results concerning computing time and speedup are given and are compared with the ones obtained from an actual volunteer environment.*

Keywords volunteer computing, agent-based simulation, RePast, MASON

1. Introduction

Volunteer computing prove to be an interesting and reliable approach to many difficult problems (cf. SETI@HOME [10] or Great Internet Mersenne Prime Search). As volunteer environments usually consist of a large number of computing nodes, with highly dynamic characteristics (the node may become up or down in random moments of time, the computing power highly depends on the volunteer configurations etc.), reliable models for planning of the whole computing (including e.g., load balancing algorithms or fault recovery strategies) are highly desired.

Utilizing the notion of agent [18] brings many improvements into the world of simulation, following the idea of decentralization of control. Each agent may be autonomous, differently configured, utilization different means of discovering the features of the environment and its neighbors, utilizing different algorithms and performing different actions in the system.

An easy to implement approach to modelling and simulation of dynamic, volunteer environments, has been proposed using popular, agent-based universal simulation frameworks, such as RePast or MASON [2]. In this contribution, the authors focused on network-related simulation aspects, taking insight into features such as throughput or node load.

In the course of paper the above-mentioned simulation frameworks are adapted to support simulation of volunteer computing. After giving implementation details, selected results concerning computing time and speedup are given and are compared with the ones obtained from an actual volunteer environment, consisting of web browsers serving as Java Script based computing nodes.

2. Agent-based simulation

There exists a plethora of multi-agent frameworks which may be used to support the construction of agent-based simulation systems. Some of them are oriented to specific kinds of simulation (see [12, 14]): e.g., simulating of movement of entities with 3D visualisation (see e.g., breve, [9]), networking (see e.g, ns2/ns3, [4]), possibility of visual programming (see e.g. SeSam, [17]).

When looking for mature, open-source, agent-based simulation project with universal applicability, supported by the wide society of programmers, two environments seem to especially attract attention, these are MASON and Repast.

MASON is an agent-oriented simulation framework developed at George Mason University. It is advertised as fast, portable, 100% Java based. Multi-layer architecture brings complete independence of the simulation logic from visualisation tools which may be altered anytime. The models are self-contained and may be included in other Java-based programs. Various means for 2D and 3D visualisation, and different means of output are avaiable (PNG snapshots, Quicktime movies, charts and graphs, data streams).

Programming model of MASON follows basic principles of object-oriented design. An agent is instantiated as an object of a class, added to a scheduler and its `step` method is called during the simulation. There are no predefined communication nor organisation mechanism, these may be realized using simple method calls. There are neither ready-to-use distributed computing facilities nor component-oriented solutions.

First released in 2003, the environment is still maintained as an open-source project, distributed under Academic Free license (ver. 3.0). The current version (16.0) was released at the end of 2011.

Repast—Recursive Porous Agent Simulation Toolkit—is a widely used agent-based modeling and simulation tool. Repast has multiple implementations in several languages and built-in adaptive features such as genetic algorithms and regression [13]. The framework utilizes fully concurrent discrete event scheduling, HPC version also exists [5]. In Repast 3, there are many programming languages interfaces (e.g., Java, Logo dialect, .NET languages, Lisp dialect, Prolog, Python). Logging and graphing tools are built-in. Dynamic access to the models in the runtime (introspection) is possible using graphical user interface. There are predefined libraries for different methods of modelling and analysis available, e.g., neural networks, genetic algorithms, social-network modelling, GIS support.

Repast 3 consists of different implementations of the platform (Repast J—Java-based, Repast.NET—MS .NET and Repast Py—Python). It has been renowned for a long time, however, recently Repast 3 has been superseded by its next stage development called Repast Simphony (Repast S) bringing newly developed GUI, with some significant changes into the programming paradigm.

The latest (Simphony 2.0 beta) version of this open-source project, licensed according to 'new BSD' license, has been released in the late 2010.

The implementation of a simulation system in Repast 3 and MASON is quite similar, and both projects are alive, thus attention was paid to both of the systems. In order to implement simulation, a class containing simulation model should be constructed, extending an appropriate superclass and implementing a method called during each step of the simulation. In this way constructed model is bound to a scheduler, that calls the above-mentioned stepping function.

3. Volunteer computing

Volunteer Computing is a type of distributed computing in which all (or at least some of) the computational resources come from the number of nodes dynamically connecting to a network, with the intent to share their computation power, either altruistically or using some service instead (e.g. gaining access to some resources) [15]. A very similar approach to *Volunteer Computing* is called *Sideband Computing*, though it requires a predefined client application installed in a desktop PC and the appropriately prepared server (usually a local gateway) to distribute the tasks [19].

The first Volunteer Computing project was Great Internet Mersenne Prime Search (`http://mersenne.org/`) and was started in 1996. The most famous project, called SETI@Home (`http://setiathome.berkeley.edu/`) (launched in 1999) is dedicated to analysing radio signals, gathered by the radio-telescope located in Arecibo (Puerto Rico), searching for signs of extra-terrestrial intelligence.

Volunteer Computing projects may be implemented using several middlewares, such as Berkeley Open Infrastructure for Network Computing (`http://boinc.berkeley.edu/`) (BOINC) (open source, base of SETI@Home), Xgrid (`http://www.apple.com/pl/server/macosx/technology/xgrid.html`) (a proprietary software prepared by and for Apple) or Grid MP (`http://www.univa.com/`) (a commercial product).

Besides computation (see e.g., [3]), other tasks may be performed in Volunteer environment. To name a few: web crawling (already attempted by [11]), MapReduce (volunteer implementation of MapReduce [7] is feasible in the opinion of authors), all such approaches may be implemented in volunteer environment, based on flexible delegation of the part of tasks to volunteers.

4. Agent-based network simulation

There exist many network simulation environments, such as e.g., NS3 [4] that may be used to extensive simulation of many complex networks (ethernet, WiFi, MANET and others). However, implementation of more sophisticated scenarios, such as emergent features of volunteer environment, may require more high-level approach, omitting features present in lower layers of ISO/OSI model [1] for the sake of abstraction.

The most important notion utilized in RePast and MASON that is taken advantage here of is the one of an agent. Computer networks consist of active (e.g., nodes, routers) and passive (switches, cables) elements in MASON. Therefore, utilizing of the notion of agent in the computer network simulation seems quite straightforward: agents are given capabilities of interacting inside the network environment (by sending and routing packets) while additional technical features supported by the simulation environments will be utilized to link the agents. Two proposed simulator prototypes presented here are implemented using MASON and Repast frameworks.

4.1. Network architecture and behaviour

The computer network is usually modelled as a connected undirected graph. The nodes of the graph represent devices and the edges represent network connections. All devices should have at least one connection to another node. The active devices utilize the passive ones to communicate by passing the information encapsulated into chunks (packets) [6]. Each endpoint has a unique network address. Addresses on the two ends of a link must belong the same subnet. The nodes send the packets to each other. The packets are marshaled by routers according to popular classful routing protocol [8].

4.2. Network simulation in RePast and MASON

The simulation frameworks implemented in RePast and MASON leverage appropriate components supported by these platforms by implementing discrete event-driven entities (agents) and providing means for communication among them. Active entities (routers and nodes) are implemented as agents in both platforms (in MASON as objects implementing the `Steppable` interface and derived from class `Node`, in RePast `@ScheduledMethod` annotation is used), while the communication among them is supported in the following way:

- *connections* are used in RePast, single instance of class `Connection` has references to many adjacent nodes, which allows direct communication between many participants,
- *links* are used in MASON, each link is an object of class `Link` and has exactly two ends, so only two nodes may communicate with each other.

Network nodes can communicate with each other by sending messages (packets). In MASON these are objects of a type derived from a base class `Packet`, delivered by calling the target node's `DeliverPacket` method. In RePast, `Connection` class provides methods for sending objects of type `Packet`. Each node is of type that extends `DTEDevice` class, which offers functionality for receiving packets and routing. Messages meant for a distant node are forwarded by routers using the implemented routing protocol. Each packet has a source and a destination address and may also contain additional data.

Packet processing is implemented using leaky bucket model [16]. The receiving device stores incoming packets in a buffer of limited size. This size is predefined in the network definition and is measured in packets—each packet is assumed to be of the same size. If there is no free space in the buffer overflowing packets are dropped. Sending a packet from a node to its neighbor takes 1 simulation step.

In each step the nodes process a set number of packets from their buffer. This number is called the devices processing power. The action taken depends on the type of node and type of received packet.

Routing is based on static routing tables created before the start of the simulation. Each table entry contains a network subnet address, the interface which should be used for sending packets to that subnet and the value of the metric (number of nodes to destination).

When processing a packet the router checks a routing table for a subnet address matching the packet's target address and passes the packet to an appropriate next node. If the target address cannot be matched to any entry in the routing table the packet is dropped.

The routing tables are created in the following manner: At the beginning a distance of 1 is assumed to the routers adjacent nodes. The router sends its routing table to all neighboring routers. Upon receiving a table a router adds all unknown addresses to its own table incrementing metric values by 1. If a path shorter then the existing entry is received the old entry is replaced by the new one.

Multiple routes with the same distance are stored and used for load distribution. The routers continue to transmit their tables until changes no longer occur.

The exact structure of the network is defined using a text file.

4.3. Repast implementation description

Devices. Figure 1 presents hierarchy of classes responsible for modeling network devices. DTEDevice is a base class. It provides basic communication mechanism (methods to send and receive messages: `receivePacket(Packet)`, `send(Packet)`), configuration (maximum buffer size, device processing power etc.) and defines method called by Repast environment for every step. Router, Master and Slave extend it and add additional behaviours. Slave represents a computing node. Whether it is active or not, it is defined by the implementation of `ActivityProvider` interface. When active, Slave requests Master for a new job.

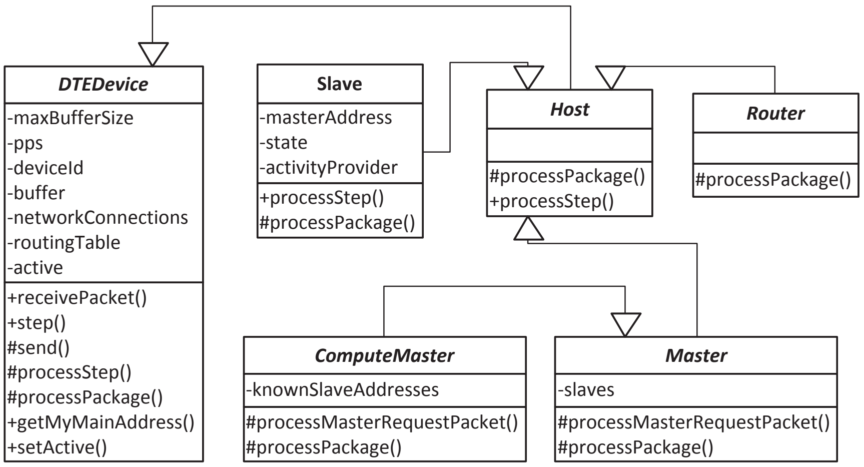


Figure 1. Repast class diagram.

Communication Devices communicate with each other by sending messages, referred to as Packets (Figure 2). A packet has two addresses: source and destination. Base class `Packet` is extended to add custom types of messages:

- *Packet* – an empty Packet.
- *MasterRequestPacket* send by Slave to the Master.
  - *JobDonePacket* – contains job results.
  - *JobRequestPacket* – request for a job.
- *MasterResponsePacket* send to Slave by the Master.
  - *JobPacket* – contains description of an assigned job.

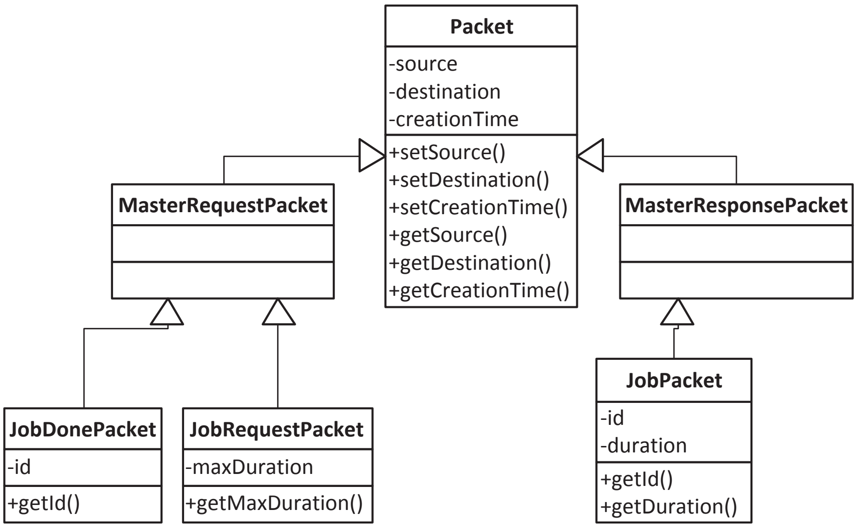


Figure 2. Repast packets diagram.

Communication is initiated by a slave. It sends `JobRequestPacket` to the master, which contains request for a job that can take maximum X computation ticks. The master responds with `JobPacket`. Slave computes results and returns `JobDonePacket`. The master periodically checks whether it has unfinished jobs that should have already been finished. If a job times out, it returns this job to unfinished jobs pool. It can then be assigned to different slave.

Configuration The Simulation can be configured by an XML configuration file. In this chapter we will show how to create basic configuration.

An example XML configuration file:

```
<?xml version="1.0"?>
<config>
  <defaultSlave buffer="40" pps="20">
    <activityProvider>
      ddos.activityproviders.SegmentActivityProvider
    </activityProvider>
    <master>666.666</master>
  </defaultSlave>
  <defaultRouter buffer="2000" pps="200"/>
  <device deviceType="master" deviceId="666" x="70.0" y="50.0" buffer="20
    <impl>ddos.model.compute.ComputeMaster</impl>
  </device>
  <!-- SUBJOBS -->
  <subjob times="48" duration="90000"/>
  <!-- 15 minutes -->
  <!-- SLAVES-->
  <device deviceType="slave" deviceId="2001" x="60.0" y="67.0"/>
  <device deviceType="slave" deviceId="2020" x="40.0" y="67.0"/>
  <!-- ... -->
  <!-- ROUTERS-->
  <device deviceType="router" deviceId="1" x="40.0" y="60.0"/>
  <device deviceType="router" deviceId="10" x="63.0" y="40.0"/>
  <!-- ... -->
  <!-- NETWORKS -->
  <network networkId="1">
    <int>1</int>
    <int>4001</int>
    <!-- ... -->
    <int>2020</int>
  </network>
  <network networkId="4">
    <int>9</int>
    <int>2015</int>
  </network>
  <!-- ... -->
</config>
```

The XML is self-descripting. Every single device should have its own address and default values defined. One major job is defined as set of sub-jobs.

4.4. MASON implementation description

Devices Network devices are implemented as classes extending the Node class, which interfaces with the simulator using MASON's `Steppable` interface. This class manages the packet buffers, handles incoming traffic and provides functions for sending new packets. Each specific device type class implements the `processPacket` function which is used to process incoming packets, and the `sendActivity` function which is called each simulation step and contains behaviour unrelated to incoming traffic. The node class hierarchy is presented in Figure 3

Communication The devices can communicate by sending packets to other devices. Packets are objects of classes derived from the `Packet` class, which contains source and destination IP addresses. If required the packet classes may define additional data fields. The packet class hierarchy is presented in Figure 4.

The following types of packets are used:

- *Packet* – an empty Packet.
- *AskPacket* – request for a job sent by a volunteer to the master.
- *ComputingInfoPacket* – a job description sent by the master in reply to AskPacket. Contains the job length.
- *EndPacket* – Noficication of a job completion sent by a volunteer to the master.

On becoming active volunteers initiate communication with the master by sending an `AskPacket` to a predefined IP address. The master replies with a `ComputingInfoPacket` which contains the time required to compute the assigned task. If the volunteer is able to complete the task before the end of it's activity period

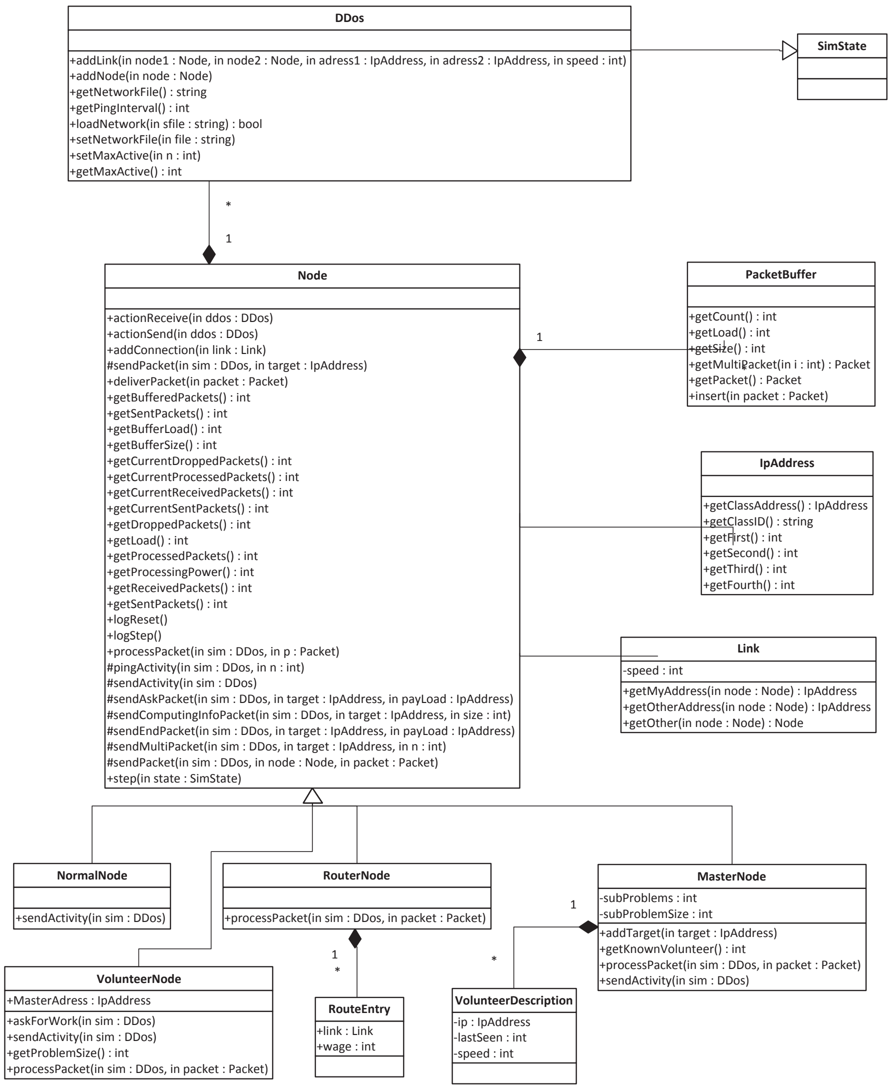

DDos
+addLink(in node1 : Node, in node2 : Node, in adress1 : IpAddress, in adress2 : IpAddress, in speed : int)
+addNode(in node : Node)
+getNetworkFile() : string
+getPingInterval() : int
+loadNetwork(in sfile : string) : bool
+setNetworkFile(in file : string)
+setMaxActive(in n : int)
+getMaxActive() : int
SimState
*
1
Node
+actionReceive(in ddos : DDos)
+actionSend(in ddos : DDos)
+addConnection(in link : Link)
#sendPacket(in sim : DDos, in target : IpAddress)
+deliverPacket(in packet : Packet)
+getBufferedPackets() : int
+getSentPackets() : int
+getBufferLoad() : int
+getBufferSize() : int
+getCurrentDroppedPackets() : int
+getCurrentProcessedPackets() : int
+getCurrentReceivedPackets() : int
+getCurrentSentPackets() : int
+getDroppedPackets() : int
+getLoad() : int
+getProcessedPackets() : int
+getProcessingPower() : int
+getReceivedPackets() : int
+getSentPackets() : int
+logReset()
+logStep()
+processPacket(in sim : DDos, in p : Packet)
#pingActivity(in sim : DDos, in n : int)
#sendActivity(in sim : DDos)
#sendAskPacket(in sim : DDos, in target : IpAddress, in payLoad : IpAddress)
#sendComputingInfoPacket(in sim : DDos, in target : IpAddress, in size : int)
#sendEndPacket(in sim : DDos, in target : IpAddress, in payLoad : IpAddress)
#sendMultiPacket(in sim : DDos, in target : IpAddress, in n : int)
#sendPacket(in sim : DDos, in node : Node, in packet : Packet)
+step(in state : SimState)
1
PacketBuffer
+getCount() : int
+getLoad() : int
+getSize() : int
+getMultiPacket(in i : int) : Packet
+getPacket() : Packet
+insert(in packet : Packet)
IpAddress
+getClassAddress() : IpAddress
+getClassID() : string
+getFirst() : int
+getSecond() : int
+getThird() : int
+getFourth() : int
Link
-speed : int
+getMyAddress(in node : Node) : IpAddress
+getOtherAddress(in node : Node) : IpAddress
+getOther(in node : Node) : Node
NormalNode
+sendActivity(in sim : DDos)
RouterNode
+processPacket(in sim : DDos, in packet : Packet)
MasterNode
-subProblems : int
-subProblemSize : int
+addTarget(in target : IpAddress)
+getKnownVolunteer() : int
+processPacket(in sim : DDos, in packet : Packet)
+sendActivity(in sim : DDos)
1
1
*
*
VolunteerNode
+MasterAdress : IpAddress
+askForWork(in sim : DDos)
+sendActivity(in sim : DDos)
+getProblemSize() : int
+processPacket(in sim : DDos, in packet : Packet)
RouteEntry
+link : Link
+wage : int
VolunteerDescription
-ip : IpAddress
-lastSeen : int
-speed : int

Figure 3. Mason node class diagram.

it informs the master that the job has been completed by sending an `EndPacket` and requests another job with an `AskPacket`.

Configuration The simulation can be configured using a text file. The file contains definitions of parameters, devices and network links. Each definition consists of 1 line beginning with a keyword specifying the type of element being defined, and is followed by a list of parameters.

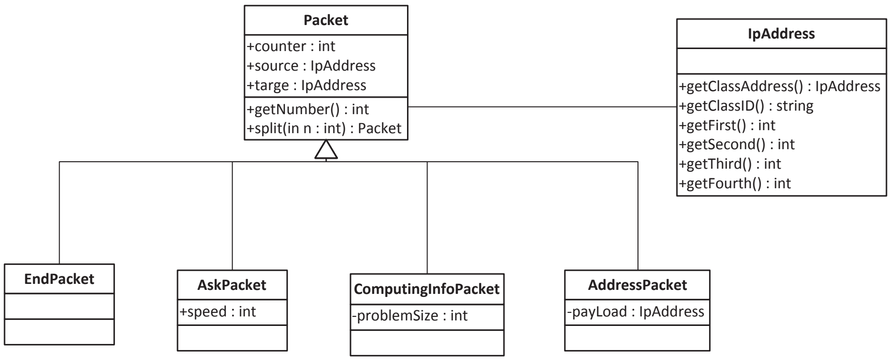


Figure 4. Mason packet class diagram.

An example configuration text file:

```
#PARAMETERS

const NumTasks 720
const TaskSize 6000
const RouterBuffer 20000
const RouterPower 200
const VolunteerBuffer 20000
const VolunteerPower 20
const NormalBuffer 20000
const NormalPower 20

#DEVICES
# type      name x    y    buffer size      computing power
master      m1   450  350  VolunteerBuffer  VolunteerPower   NumTasks TaskSize
router      r1   400  350  RouterBuffer     RouterPower
router      r2   400  350  RouterBuffer     RouterPower
volunteer   a1   200  100  VolunteerBuffer  VolunteerPower
volunteer   a2   200  200  VolunteerBuffer  VolunteerPower
volunteer   a3   200  300  VolunteerBuffer  VolunteerPower

#NETWORK
#     name1 address1     name2 address2
link    m1  192.168.1.1     r1 192.168.1.2
link    r1  192.168.2.1     r2 192.168.2.2
link    r2  192.168.3.1     a1 192.168.3.101
link    r2  192.168.3.2     a2 192.168.3.102
link    r2  192.168.3.3     a3 192.168.3.103
```

5. Agent-based simulation of volunteer computing

As a continuation of the research presented in [2], devoted to the simulation of the distributed security testing, this contribution focuses on the simulation of volunteer computing using the already constructed frameworks, based on RePast and MASON.

As in the above-mentioned publication, in this case also the computing nodes are simulated (and tested) as web browsers running Java Script. Although it might not be the best choice, from the computing speed point of view, the portability of such approach allows to recruit easily the volunteers from different communities of users in the Internet (Mozilla Firefox, Opera, Safari, Internet Explorer, Google Chrome and others).

The considered volunteer environment is presented in Figure 5.

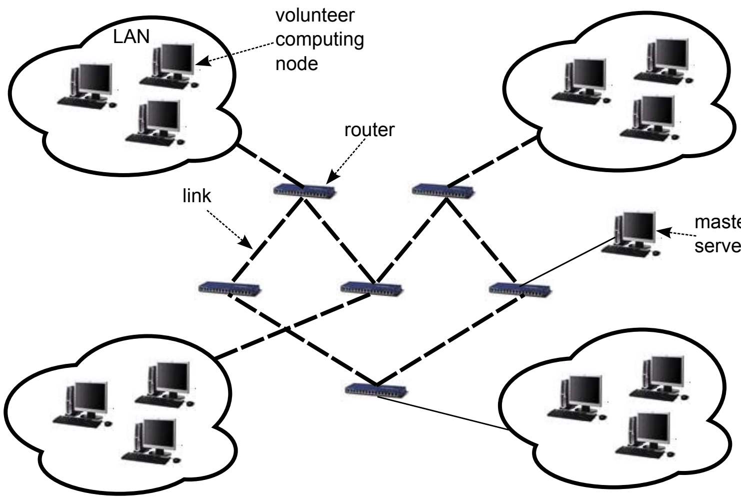


Figure 5. Volunteer network structure.

The network presented there, consists of the following types of nodes, implemented as agents in RePast and MASON:

- Router: joins two or more subnets and provides packet routing according to RIP protocol.
- Master: The computation coordinator. Distributes tasks to nodes, may implement sophisticated planning of distribution including load balancing (e.g., assigning targets to the testers), in order to utilize available computational power of the volunteer environment, at the same time trying to decrease the congestion in the network.
- Computing node (slave): A web-browser volunteer participating in the test. The computing node may be in one of two possible states: active or inactive. Depending on the simulation parameters it may change its state during the simulation. During its active status it retrieves the computing tasks from the master and after finishing the computation, it uploads the outcome to the master.

6. Experimental results

In the presented case of volunteer computing simulation, the following setting has been configured for MASON- and Repast-based experiments:

- Computation nodes activity: slaves change their state (active/passive) every random number of steps by normal distribution: next_state_change_after = nextGaussian()*18 min + 6 min.
- Slaves, Masters: buffer size = 20000, packets processed per step = 20.
- Routers: buffer size = 20000, packets processed per step = 200.
- Computation – 12 hours, divided into sub-jobs according to seven different scenarios:
    - 43200 * 1 second,
    - 720 * 1 minute,
    - 144 * 5 minutes,
    - 48 * 15 minutes,
    - 36 * 20 minutes,
    - 30 * 24 minutes.
- Number of participating volunteers: 1..20

Figures 6a and 6b present the relation between the computation time and the maximum number of volunteers. The computation time of course gets shorter, when more volunteers are present, however it is difficult to find an active volunteer for longer sub-jobs, especially when their maximum number is low. It explains generally higher computation time for the cases of longer sub-jobs.

Figures 7a and 7b present the complementary information to the one given previously, regarding the speedup of the computation. It is to note, that this dependencies are described with almost linear functions visualised in the graphs.

Final information is given in Figures 8a and 8b, presenting the number of volunteers sleeping (inactive), active but not working (idle), and working. The longer the sub-job duration is, the more volunteers are in the idle state that confirms the above-mentioned observations.

It is to note, that all the above mentioned experiments yielded quite similar results, both for MASON and RePast based simulations. Some difference is inevitable, but none of the tested systems was too far from another.

The presented simulation results seem to show reasonable behaviour of the system, however an ultimate test is to compare them to the results obtained from the real world experiment. Therefore, the following testbed configured according to the structure depicted in Figure 5 was constructed.

- The actual nodes and routers were configured using Ubuntu Linux ver. 10.10 and run as virtual machines using VMWare®Player 4.0.4. build-744019
- The actual computing was done by web browsers (Google Chrome ver. 20.0.1132.57).

- The hardware used to run the above-mentioned virtual machines were:
  - 2×Intel®Core^{TM}i7 CPU 920, 2.67 GHz, 3.5 GB RAM, running Microsoft Windows XP Home Edition SP 3.
  - 2×Intel®Core^{TM}2 Duo CPU E7400, 2.80 GHz, 3.25 GB RAM, running Microsoft Windows XP Home Edition SP 3.
  - 3×Intel®Core^{TM}i7 CPU 950, 3.07 GHz, 8 GB RAM, running Ubuntu Linux 10.04.4 LTS (64bit).
  - 3×Intel®Core^{TM}i3 CPU 540, 3.07 GHz, 4 GB RAM, running Microsoft Windows 7 Home Premium SP1 (64bit).

The experiments were repeated in real world environment for two configurations: 1 min and 20 min sub-jobs, for the whole spectrum of maximum volunteers.

The results of comparing simulated and real world environments, regarding the computing time, are shown in Figure 9. It is easy to see, that the computing times are nearly identical in the case of 1 min. subjob. More significant difference is observed in the case of 20 min. subjobs. As it was observed before, fluent processing larger tasks is more difficult, as finding active volunteer causes more problems (as they are either asleep, or processing large tasks). At the same time, reliable throughput attained in the system processing 1 min. subjobs is easier to maintain, as many volunteer do their jobs quick and are ready to process next tasks. These observations are confirmed when looking at the results presented in Figure 9. In order to additionally confirm the reliability of the constructed simulators, comparison of the involved volunteers percentage yields very similar results for three tested platforms (see Figure 10).

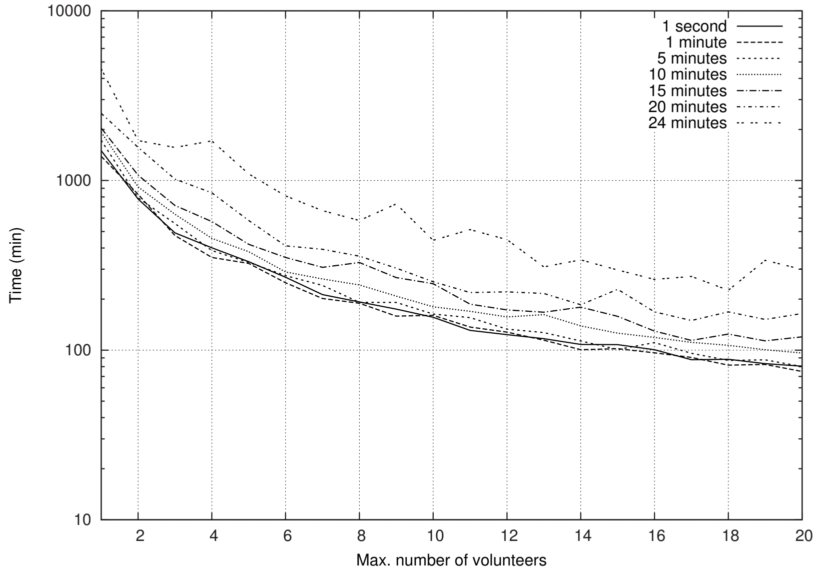
7. Conclusion

In this paper the already presented network simulation environments, built with use of RePast and MASON, previously applied to the problem of simulation of volunteer security testing system [2], are adapted to simulate volunteer computing environment consisting of web browsers running Java Script.

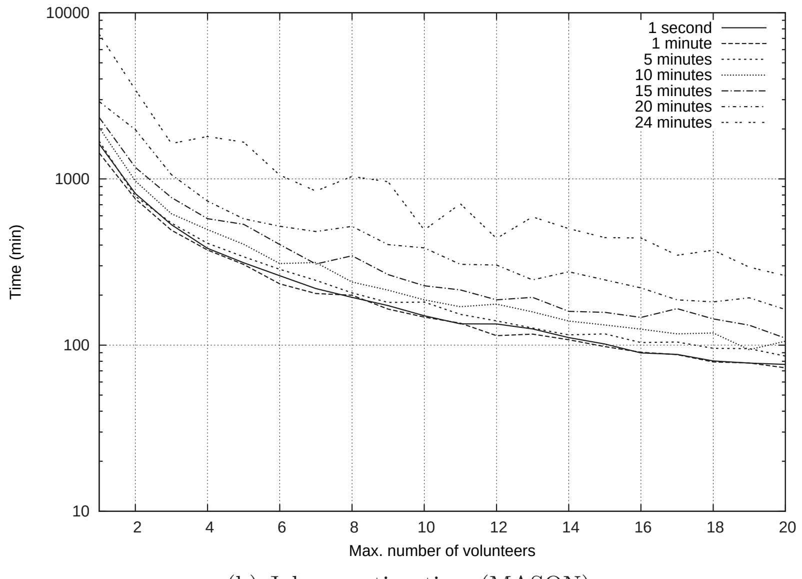
The construction of simulation environments for such cases seems to be indispensable, as such highly dynamic environments would require preparing and testing different complex strategies (such as load balancing or fault recovery).

It was shown that the constructed environments yielded reliable results, by comparing them one with another. However, the accuracy confirmation of the constructed simulators was obtained after preparing the real world testing scenario which implemented the structure well-known from the simulated test case in the virtual environment. The obtained results for 1 min. and 20 min. subjobs show, that both simulators constructed with RePast and MASON are reliable and may be further developed.

In the future it is planned to extend the structure of simulators, by introducing hierarchical structure of components (e.g., middle servers, for better distribution of the tasks in complex networks) and preparing the experiments involving different strategies of task distribution, load balancing etc.

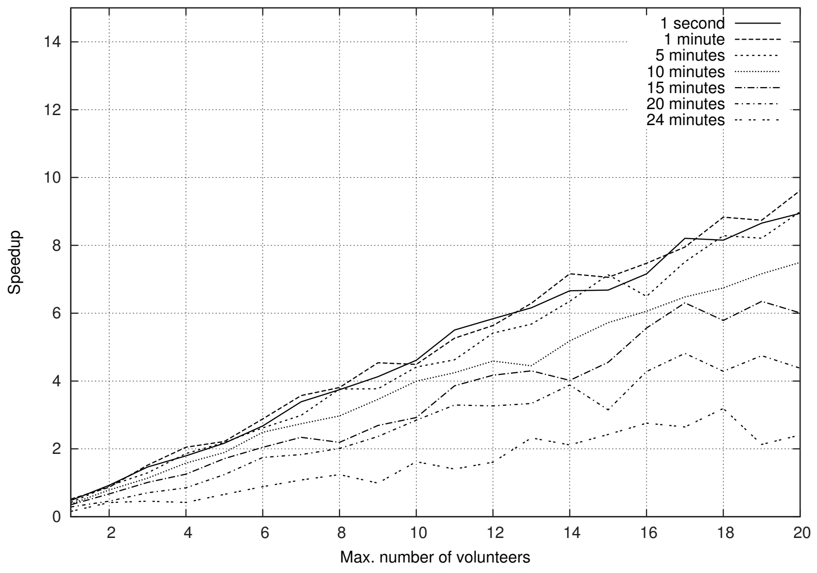


(a) Job executing time (RePast)

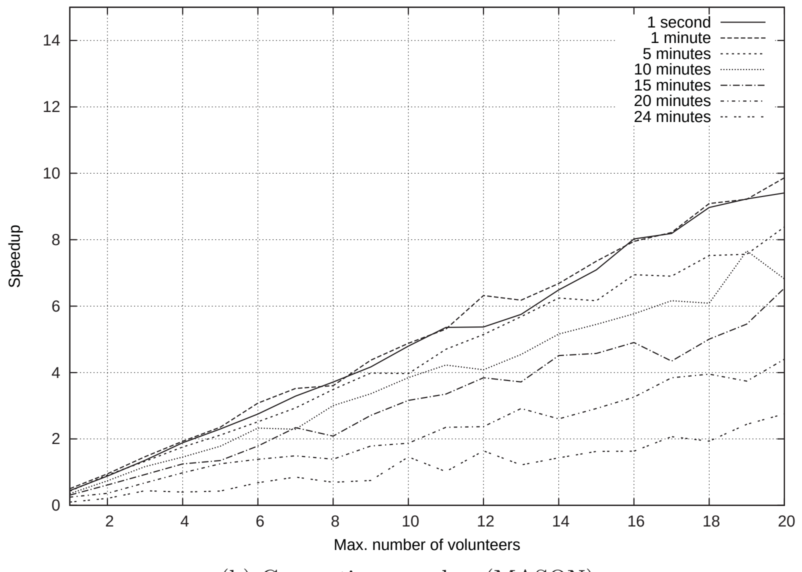


(b) Job executing time (MASON)

Figure 6. Job executing time in RePast and MASON.

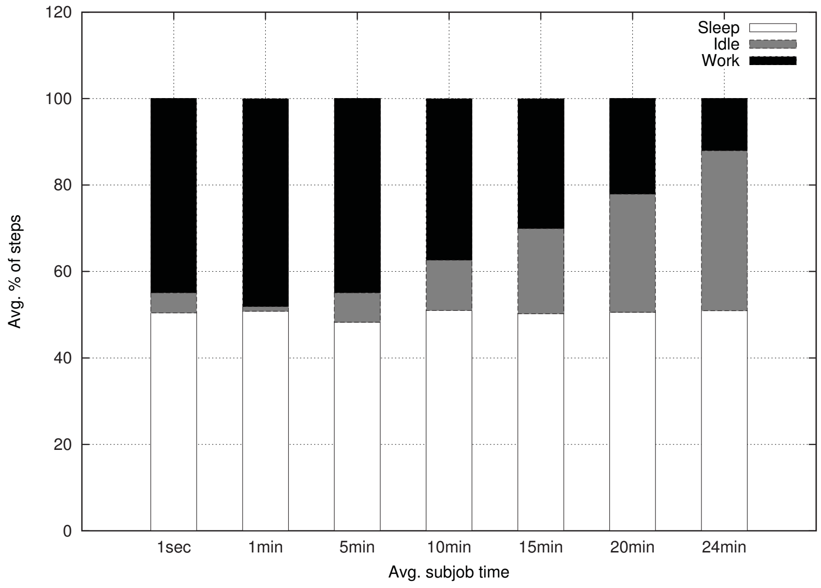


(a) Computing speedup (RePast)

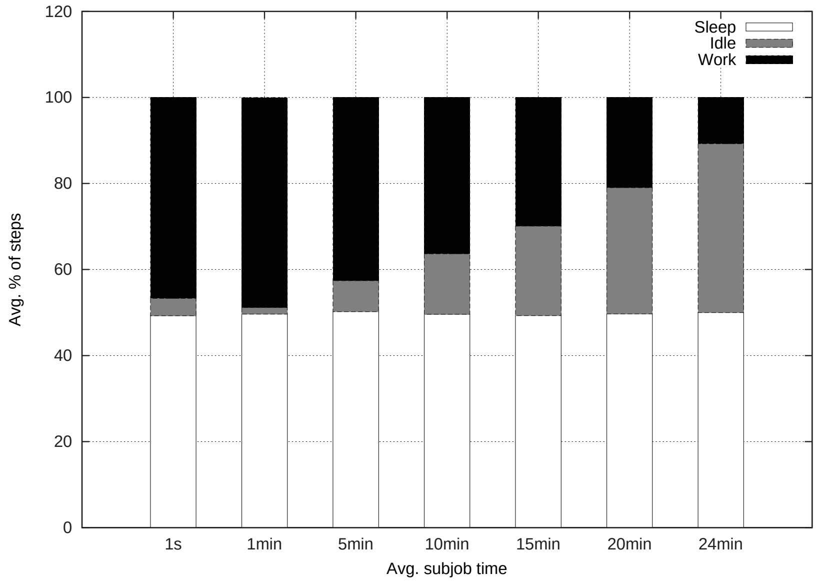


(b) Computing speedup (MASON)

Figure 7. Computing speedup in RePast and MASON.

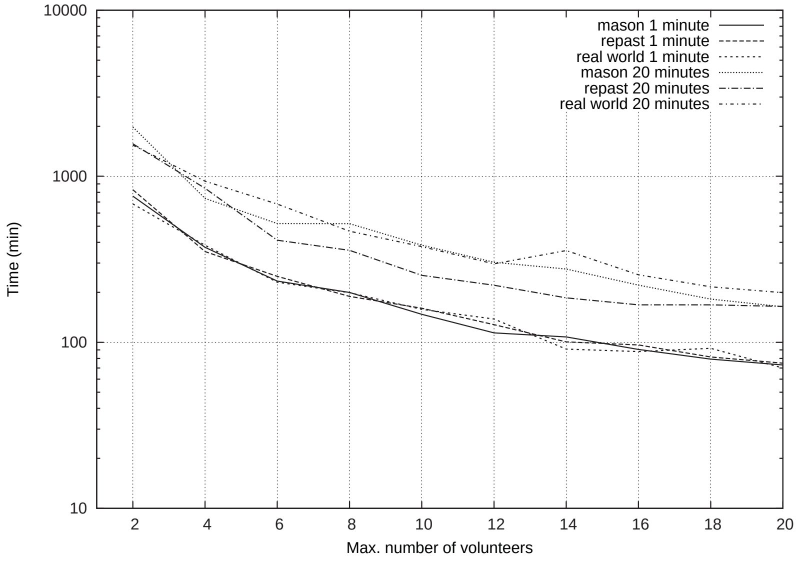


(a) Percentage of involved volunteers (Repast)

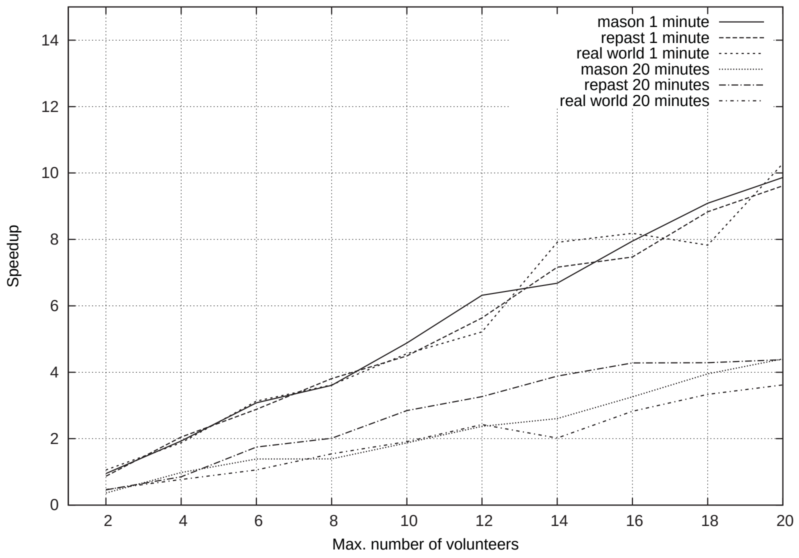


(b) Percentage of involved volunteers (MASON)

Figure 8. Percentage of involved volunteers in RePast and Mason.



(a) Job executing time



(b) Computing speedup

Figure 9. Job executing time and computing speedup comparison among RePast, MASON and real world experiment.

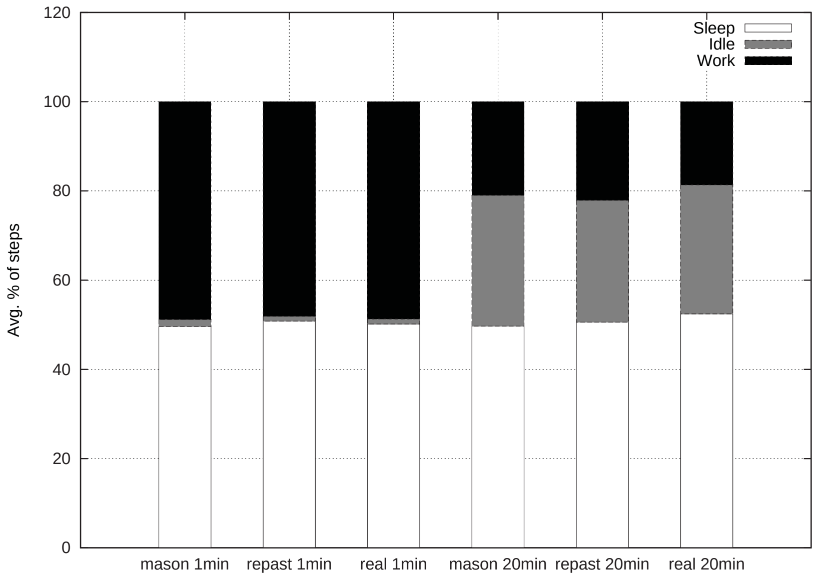


Figure 10. Percentage of involved volunteers comparison among RePast, MASON and real world experiment.

Acknowledgements

The research leading to these results has received funding from Polish National Centre for Research and Development with grant agreement number 0108/R/T00/2010/11.

References

[1] Bush R., Meyer D.: Some internet architectural guidelines and philosophy. RFC 1058, 2002.

[2] Byrski A., Feluś M., Gawlik J., Jasica R., Kobak P., Nawarecki E., Wroczyński M., Majewski P., Krupa T., Skorupka P.: Agent-based simulation of volunteer environment. In Troitzsch K., Möhring M., Lotzmann U., eds, *Proc. of 26th European Conference on Modelling and Simulation ECMS*, 2012.

[3] Byrski A., Debski R., Kisiel-Dorohinickki M.: Agent-based computing in augmented cloud environment. *Computer Systems Science & Engineering (In Press)*, 2012.

[4] Carneiro G., Fontes H., Ricardo M.: Fast prototyping of network protocols through ns-3 simulation model reuse. *Simulation Modelling Practice and Theory*, 19(9):2063 – 2075, 2011.

[5] Collier N., North M.: *Repast SC++: A Platform for Large-scale Agent-based Modeling.* Wiley, 2011.

[6] Comer D. E.: *Internetworking with TCP/IP – Principles, Protocols and Architecture.* Prentice Hall, 2006.

[7] Dean J., Ghemawat S.: Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation*, 2004.

[8] Hendrik C.: *Routing information protocol.* RFC 1058, 1988.

[9] Klein J.: Breve: A 3d environment for the simulation of decentralized systems and articial life. In *Proc. of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, 2002.

[10] Korpela E. J.: Seti@home, BOINC and volunteer distributed computing. *Annual Review of Earth and Planetary Science*, 40(1), April 2012.

[11] Krupa T., Majewski P., Kowalczyk B., Turek W.: On-demand web search using browser-based volunteer computing. In *Proc. of 6th Int. Conf. on Complex, Intelligent and Software Intensive Systems*, 2012.

[12] Nikolai C., Madey G.: Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2), 2008.

[13] North M., Howe T., Collier N., Vos J.: A declarative model assembly infrastructure for verification and validation. In Takahashi S., Sallach D., Rouchier J., eds, *Advancing Social Simulation: The First World Congress, Springer, Heidelberg, FRG (2007)*, 2007.

[14] Railsback S., Lytinen L.: Agent-based simulation platforms: review and development recommendations. *Simulations*, 82:609–623, 2006.

[15] Sarmenta L.: Bayanihan: Web-based volunteer computing using java. In *Proc. of the 2nd International Conference on World-Wide Computing and its Applications (WWCA'98), Tsukuba, Japan, March 3-4, LNCS 1368*, 1998.

[16] Tanenbaum A. S.: *Computer Networks*, 4th ed. Prentice Hall, 2003.

[17] Ventroux N., Guerre A., Sassolas T., Moutaoukil L., Blanc G., Bechara C., David R.: Sesam: An mpsoc simulation environment for dynamic application processing. In *CIT*, pp. 1880–1886. IEEE Computer Society, 2010.

[18] Wooldridge M., Jennings N.: Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[19] Xu Y.: Global sideband service distributed computing method. In *Proc. of the International Conference on Communication Networks and Distributed System Modeling and Simulation (CNDS'98)*, 1998.

Affiliations

Aleksander Byrski
AGH University of Science and Technology, Krakow, Poland, `olekb@agh.edu.pl`

Michał Feluś
AGH University of Science and Technology, Krakow, Poland, `felus@student.agh.edu.pl`

Jakub Gawlik
AGH University of Science and Technology, Krakow, Poland, `jgawlik@student.agh.edu.pl`

Rafał Jasica
AGH University of Science and Technology, Krakow, Poland, `jasica@student.agh.edu.pl`

Paweł Kobak
AGH University of Science and Technology, Krakow, Poland, `kobak@student.agh.edu.pl`

Grzegorz Jankowski
AGH University of Science and Technology, Krakow, Poland, `jankow@student.agh.edu.pl`

Edward Nawarecki
AGH University of Science and Technology, Krakow, Poland, `nawar@agh.edu.pl`

Michał Wroczyński
Fido Intelligence, Gdansk, Poland, `mwroczynski@fidointelligence.pl`

Przemysław Majewski
Fido Intelligence, Gdansk, Poland, `pmajewski@fidointelligence.pl`

Tomasz Krupa
Fido Intelligence, Gdansk, Poland, `tkrupa@fidointelligence.pl`

Jacek Strychalski
Fido Intelligence, Gdansk, Poland, `jstrychalski@fidointelligence.pl`

Received: 24.07.2012
Revised: 14.10.2012
Accepted: 3.12.2012