Jacek Szarski <jacek.szarski@gmail.com>
Mateusz Zawisza <mateusz.zawisza@gmail.com>
EiT, EAIiE, AGH

BroTalk - Implementation of fully distributed instant messaging application and protocol

1. Abstract

Objectives

The goal of our project was to create a fully-distributed instant messaging software, design and implement its protocol.

Peers are divided into regular nodes and supernodes. Each supernode knows the architecture of the whole system. Regular nodes only need to know the supernodes.
To achieve data redundancy, new supernodes are elected when needed.

Work done

- designed the protocol
- implemented the instant messaging application with rpsec automated tests
- implemented a virtual interface providing a simulation environment + web interface
- implemented vagrant-based UDP simulation environment

Achieved

- learned about different redundancy implementations
- learned about different P2P protocols problems & solutions
- learned about the ICMP protocol
- integrated with the Twitter API
- created a report on the experiment results

Encountered problems

- root provileges required for ICMP communication
- Twitter API spam filter blocking search results
- testing setup difficulties
- external IP required for supernodes

2. Protocol Specification

All the messages are encoded using the JSON standard.
Each message has a type and content.

a. greeting

message type: “ay bro”
content: {bros: bros_table}

where the bros_table is the array of all the bros known by sender

rules:

- when greeting a peer needs to send all his bros over
- when a peer is greeted by someone, he needs to update his bro table with all the new entries, update last_activity timestamps and greet all the new entries

b. election

After updating the bros table, peer needs to check whether he knows at least one supernode.
If not, he elects himself as a supernode.

c. clear_bro_table

Each peer periodically clears the bro table. If he knows more than 2 regular nodes, he clears them.

Each supernode remembers all the nodes he has ever knew about.
Each node must remember all supernodes.

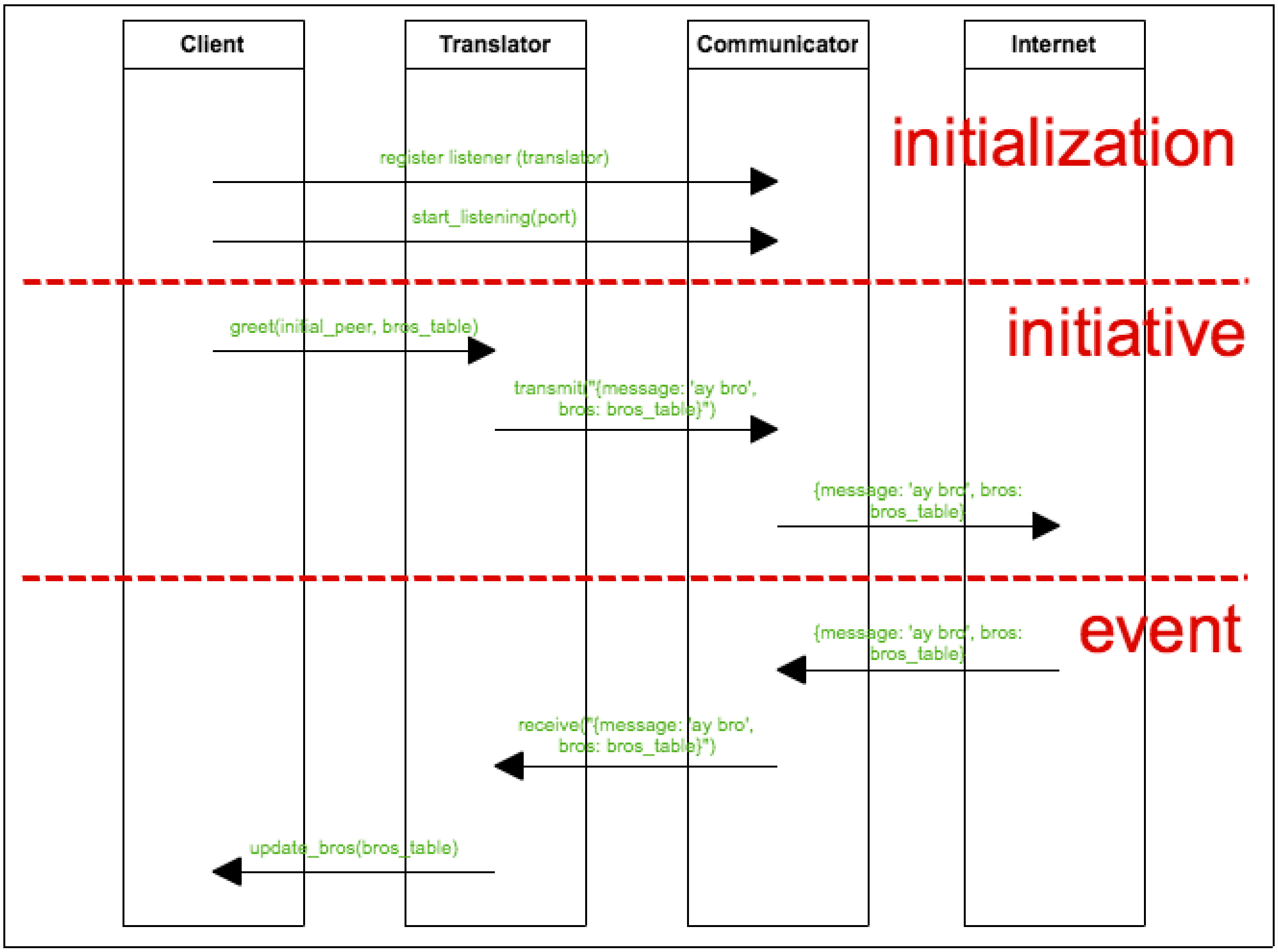
d. ping

Each peer is obligated to keep a timestamp of the last activity of every bro.
When this timestamp gets older than ZOMBIE_CHECK_LIMIT, it sends a ping request to the bro.
Upon receiving a ping, it is obligated to reply with a pong.

e. pong

When receiving a pong from a bro, the peer needs to update his last_activity timestamp

Example greeting scenario



3. Simulation

Environments

The application's structure is modular. One of the elements - communicator - is the only part responsible for communication with the outside world.
Since it does not know the protocol nor is it involved in any high-level logic, we took the opportunity to implement two different communicators and switch them to achieve different testing goals.

a. Virtual environment

In order to be able to simulate complex scenarios we implemented a virtual communicator.
It does not send actual packets, but allows to achieve and monitor communication between different Client instances within the same process.
Next, the webtest script creates an instance of Simulator that initializes Clients and opens up an interface for manipulation using DRB.
The watcher script opens up a http interface that serves as a proxy for the DRB api and hosts a javascript-enabled graphic interface that allows the user to play with the Clients network using just mouse clicks.

Instructions:
1. Download code from https://github.com/szarski/brotalk either with git or by downloading zip or tarball file and unpack it to directory "brotalk"
2. Install:
    - ruby 1.9 (http://www.ruby-lang.org/pl/downloads/)
    - rubygems (http://rubygems.org/pages/download)
    - bundler gem (http://gembundler.com/)

3. Run inside project directory:

```
bundle install
```

4. Go to the project directory and start watcher and webtest:

```
bundle exec ruby watcher.rb
bundle exec ruby webtest.rb
```

5. Open up a web browser and go to localhost:4567

b. Real environment

Real world simulations were done with use of virtual machines communicating via UDP protocol. Virtual machines are handled by VirtualBox environment and are managed by program called vagrant. Vagrant is nice wrapper around VirtualBox, which makes creating and launching new machines easy.

Requirements:

- VirtualBox - https://www.virtualbox.org/
- Vagrant - http://vagrantup.com/
- Brotalk code - https://github.com/szarski/brotalk

Instructions:

1. Install project

Download code from https://github.com/szarski/brotalk either with git or by downloading zip or tarball file and unpack it to directory “brotalk”

2. Launch Virtual Machines

Enter the directory and run command:

```
vagrant up
```

When command is executed vagrant will bring up and configure two virtual machines that are ready to launch brotalk and communicate. We can enter them via ssh by typing:

```
vagrant ssh host1
```

or

```
vagrant ssh host2
```

3. Start brotalk

When inside the machine enter the /vagrant directory (“cd /vagrant”) and run the script with command:

```
bundle exec ruby ./bin/brotalk.rb [IP_ADDRESS]
```

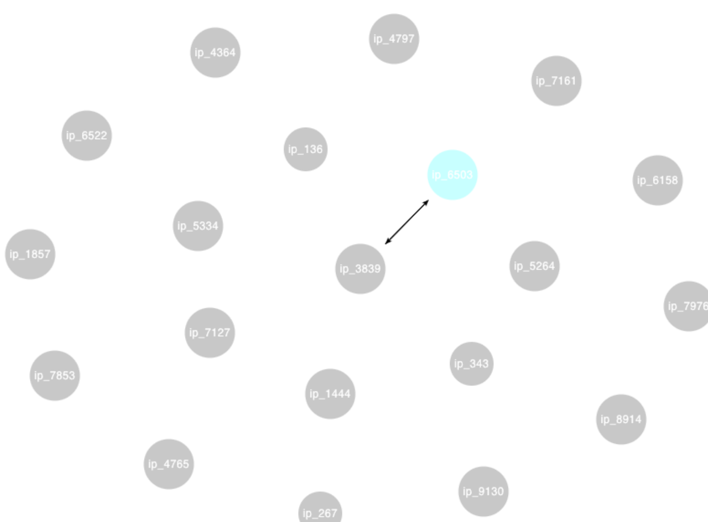
If IP_ADDRESS is specified, brotalk will send “greet” message to host at this address. If no address is given, brotalk will only listen for incoming messages.

Results

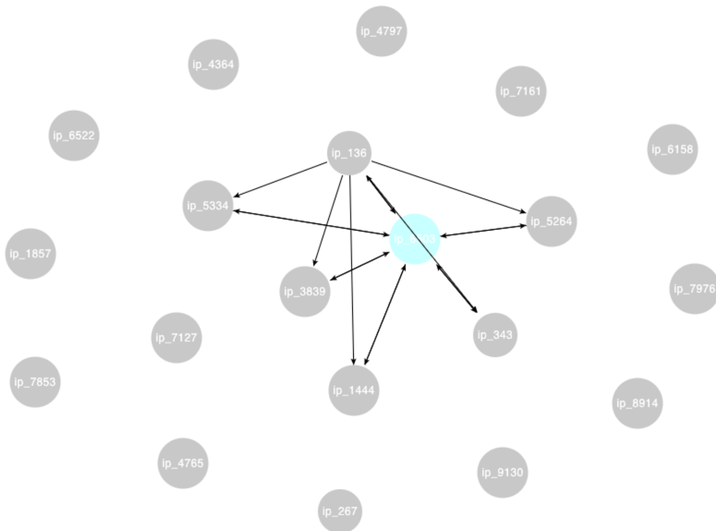
Firstly we simulated a more sophisticated scenario using the virtual communicator.
We managed to test how the protocol behaves with large amounts of nodes and operations.

1. network of nodes without any connections

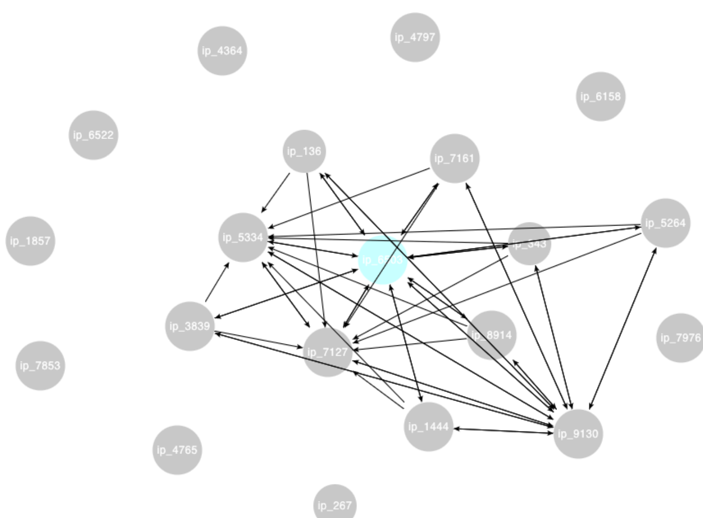
2. node is tolled to greet someone, the receiver becomes supernode immediately



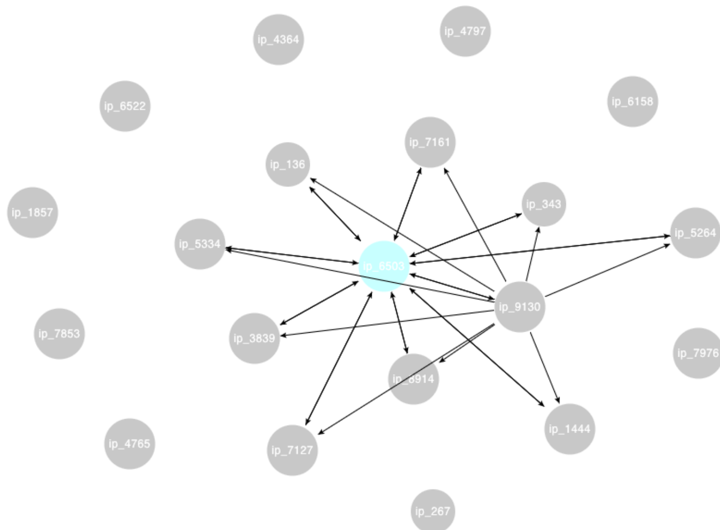
3. further connections made by greeting more nodes



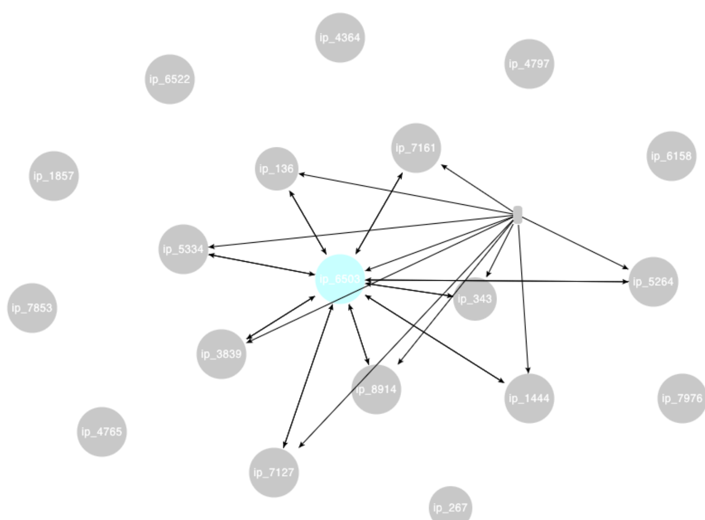
4. packet flood and connection flood after greeting one of the network members



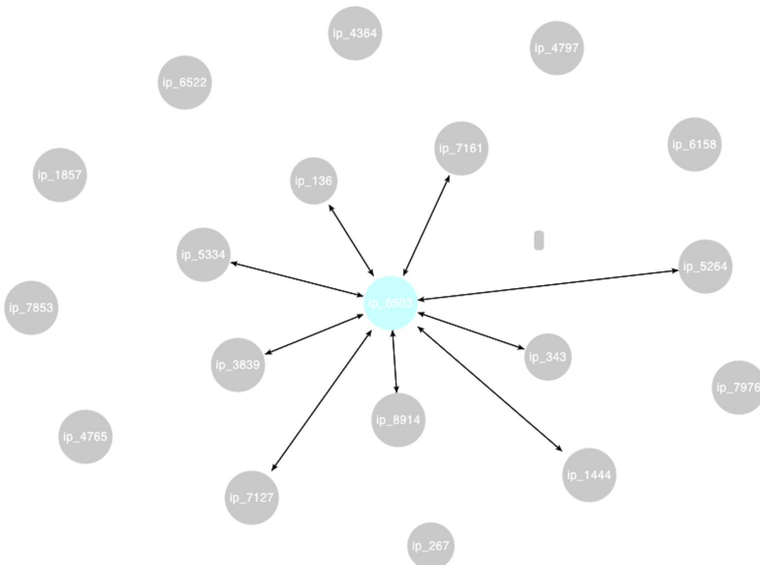
5. nodes forgotten, network relaxed after a few seconds



6. node removed, new nodes still keeping information about the missing node



7. after ZOMBIE_KILL_LIMIT seconds, the old entries are erased



Secondly, we conducted a test using real communicators hooked up to network interfaces of virtual machines, launched with Vagrant.
That way we could have a better insight on how the network packets of each of the messages look and how they are distributed across the nodes.
In order to illustrate the data, we provide tcpdump output analyzed with Wireshark.

1. We start with two nodes that does not know about each other.
Host 192.168.33.11 sends greet to host 192.168.33.10 with empty bros table.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.33.1	192.168.33.255	UDP	86	Source port: 57621
2	8.773004	192.168.33.11	192.168.33.10	UDP	92	Source port: 47216

```
0000  08 00 27 6c 32 09 08 00  27 7d 6e 2b 08 00 45 00   ..'l2... '}n+..E.
0010  00 4e 3d 33 40 00 40 11  3a 06 c0 a8 21 0b c0 a8   .N=3@.@. :...!...
0020  21 0a b8 70 22 c3 00 3a  ef 36 7b 22 6d 65 73 73   !..p"..: .6{"mess
0030  61 67 65 22 3a 22 61 79  20 62 72 6f 22 2c 22 62   age":"ay  bro","b
0040  72 6f 73 5f 74 61 62 6c  65 22 3a 5b 5d 2c 22 73   ros_tabl e":[],"s
0050  75 70 65 72 6e 6f 64 65  22 3a 30 7d               upernode ":0}
```

Host 192.168.33.10 sends greet to 192.168.33.11 with himself elected as a supernode

No.	Time	Source	Destination	Protocol	Length	Info
2	8.773004	192.168.33.11	192.168.33.10	UDP	92	Source port: 47216
3	8.773732	192.168.33.10	192.168.33.11	UDP	109	Source port: 47381

```
0000  08 00 27 7d 6e 2b 08 00  27 6c 32 09 08 00 45 00   ..'}n+.. 'l2...E.
0010  00 5f 50 36 40 00 40 11  26 f2 c0 a8 21 0a c0 a8   ._P6@.@. &...!...
0020  21 0b b9 15 22 c3 00 4b  c3 c2 7b 22 6d 65 73 73   !..."..K ..{"mess
0030  61 67 65 22 3a 22 61 79  20 62 72 6f 22 2c 22 62   age":"ay  bro","b
0040  72 6f 73 5f 74 61 62 6c  65 22 3a 5b 22 31 39 32   ros_tabl e":["192
0050  2e 31 36 38 2e 33 33 2e  31 31 25 30 22 5d 2c 22   .168.33. 11%0"],"
0060  73 75 70 65 72 6e 6f 64  65 22 3a 31 7d            supernod e":1}
```

Host 192.168.33.11 sends greet to 192.168.33.10 with full bros table

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.33.1..	192.168.33.255	UDP	86	Source port: 57621
3	8.773732	192.168.33.10	192.168.33.11	UDP	109	Source port: 47381
4	8.774531	192.168.33.11	192.168.33.10	UDP	127	Source port: 47147

```
0000  08 00 27 6c 32 09 08 00  27 7d 6e 2b 08 00 45 00   ..'l2... '}n+..E.
0010  00 71 3d 33 40 00 40 11  39 e3 c0 a8 21 0b c0 a8   .q=3@.@. 9...!...
0020  21 0a b8 2b 22 c3 00 5d  2d 79 7b 22 6d 65 73 73   !..+"..] -y{"mess
0030  61 67 65 22 3a 22 61 79  20 62 72 6f 22 2c 22 62   age":"ay  bro","b
0040  72 6f 73 5f 74 61 62 6c  65 22 3a 5b 22 31 39 32   ros_tabl e":["192
0050  2e 31 36 38 2e 33 33 2e  31 31 25 30 22 2c 22 31   .168.33. 11%0","1
0060  39 32 2e 31 36 38 2e 33  33 2e 31 30 25 31 22 5d   92.168.3 3.10%1"]
0070  2c 22 73 75 70 65 72 6e  6f 64 65 22 3a 30 7d      ,"supern ode":0}
```

2. To verify that hosts are still available in the network they exchange ping/pong messages.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.33.1..	192.168.33.255	UDP	86	Source port: 57621
6	13.765522	CadmusCo_7d:6e:	CadmusCo_6c:32:0	ARP	60	192.168.33.11 is at
7	14.778354	192.168.33.11	192.168.33.10	UDP	60	Source port: 43626

```
0000  08 00 27 6c 32 09 08 00  27 7d 6e 2b 08 00 45 00   ..'l2... '}n+..E.
0010  00 2e 3f 8b 40 00 40 11  37 ce c0 a8 21 0b c0 a8   ..?.@.@. 7...!...
0020  21 0a aa 6a 22 c3 00 1a  11 31 7b 22 6d 65 73 73   !..j"... .1{"mess
0030  61 67 65 22 3a 22 70 69  6e 67 22 7d               age":"pi ng"}
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.33.1..	192.168.33.255	UDP	86	Source port: 57621
7	14.778354	192.168.33.11	192.168.33.10	UDP	60	Source port: 43626
8	14.778874	192.168.33.10	192.168.33.11	UDP	60	Source port: 35151

```
0000  08 00 27 7d 6e 2b 08 00  27 6c 32 09 08 00 45 00   ..'}n+.. 'l2...E.
0010  00 2e 52 8f 40 00 40 11  24 ca c0 a8 21 0a c0 a8   ..R.@.@. $...!...
0020  21 0b 89 4f 22 c3 00 1a  c3 91 7b 22 6d 65 73 73   !..O"... ..{"mess
0030  61 67 65 22 3a 22 70 6f  6e 67 22 7d               age":"po ng"}
```

4. Conclusions

During the course of our experiment we used a lot of different languages, technologies, and tools available. We experimented with a lot of different protocols to conduct the desired experiments.

We learned about how real p2p protocols work in order to take example and implement in our project. We quickly discovered that it is very difficult to design a protocol serving even a simple purpose and that it’s almost impossible without running many simulations for different scenarios.

Our experiment showed that it is much easier to conduct tests in a virtual environment where you can quickly modify the implementation and measure the results.

We also learned that you should always think of a p2p-enabled application as of a modular infrastructure in order to be able to perform certain analysis and experiments.