



UNIVERSITÀ DEGLI STUDI DI ROMA  
TOR VERGATA

Sicurezza Informatica e Internet

A.A. 2015/2016

**Build Your Own Botnet v.1**



0211577 - Cappello Domenico - [domenico.cappello@gmail.com](mailto:domenico.cappello@gmail.com)

0213347 - Nazio Alessio - [alessio.nazio@gmail.com](mailto:alessio.nazio@gmail.com)

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Premessa . . . . .	1
1.2	HTTP Botnet . . . . .	2
1.3	Obiettivo . . . . .	3
<b>2</b>	<b>Panoramica del progetto</b>	<b>4</b>
2.1	Componenti principali . . . . .	4
2.1.1	HTTP . . . . .	5
2.1.2	Singleton Pattern . . . . .	5
2.1.3	Scheduler . . . . .	6
2.1.4	ID dei bot e MD5 . . . . .	6
2.1.5	Parsing . . . . .	6
2.1.6	Multi-piattaforma . . . . .	7
2.1.7	Multi-threading . . . . .	7
2.2	Piattaforma di sviluppo, Swing, Awt . . . . .	7
2.3	Build your own botnet . . . . .	8
2.3.1	Struttura del progetto . . . . .	8
2.3.2	ByobComm . . . . .	9
2.3.3	ByobSingleton . . . . .	9

2.3.4	ByobTask . . . . .	9
2.3.5	GUI . . . . .	10
2.3.6	Parser . . . . .	15
2.3.7	Tools . . . . .	15
2.3.8	URLDetails . . . . .	15
<b>3</b>	<b>Implementazione</b>	<b>16</b>
3.1	Avvio iniziale . . . . .	16
3.2	Informazioni di sistema . . . . .	17
3.2.1	Linux . . . . .	17
3.2.2	Windows . . . . .	18
3.2.3	Mac OSX . . . . .	19
3.3	File di configurazione . . . . .	20
3.3.1	Immissione e scrittura . . . . .	20
3.3.2	Lettura e caricamento . . . . .	24
3.4	Lancio . . . . .	26
3.4.1	Schedulazione dei task . . . . .	26
3.4.2	Comunicazione HTTP . . . . .	28
<b>4</b>	<b>Testing</b>	<b>29</b>
<b>5</b>	<b>Installazione</b>	<b>32</b>

# Capitolo 1

## Introduzione

### 1.1 Premessa

Oggigiorno, la maggior parte dei PC utilizza sistemi operativi senza patch e/o senza alcuna sicurezza dietro un *firewall*, rendendoli facili prede di attacchi diretti, orchestrati da malintenzionati, e di attacchi di tipo indiretto, mascherati dietro programmi che l'utente utilizza costantemente (vedi reti *P2P*).

Con l'incremento delle connessioni a banda larga si ha avuto anche un incremento del numero di potenziali vittime di attacchi, con cui i malintenzionati traggono beneficio dalla situazione, utilizzandola a loro vantaggio, sfruttando anche l'automatizzazione di tecniche per la scansione di porzioni della rete che semplifica la ricerca di sistemi vulnerabili. Una volta che un vasto numero di macchine sono state infettate, esse entrano a far parte di una “rete di macchine compromesse che posso essere controllate da remoto<sup>1</sup>”, chiamata **botnet**.

Una *botnet* consiste di tre elementi principali che sono i bot (cioè le macchine infettate che ne fanno parte), il *command and control server* (C&C - da cui ogni bot riceve istruzioni e con cui il malintenzionato ha privilegi amministrativi remoti su tutte le macchine infette) ed il *botmaster*; si basa, inoltre, su quattro concetti chiave:

---

<sup>1</sup>Provos Niels, Holz Thorsten (2007).

1. le *botnet* sono reti, quindi sistemi in cui la comunicazione è importante;
2. le macchine che fanno parte di una *botnet* sono, tipicamente, partecipanti ignari;
3. i *bot* sono controllabili da remoto, permettendo di fare rapporto o ricevere ordini da una struttura C&C (centralizzata o decentralizzata);
4. i *bot* sono controllati da persone con intenti malevoli che fanno capo a qualche forma di attività illegale.

## 1.2 HTTP Botnet

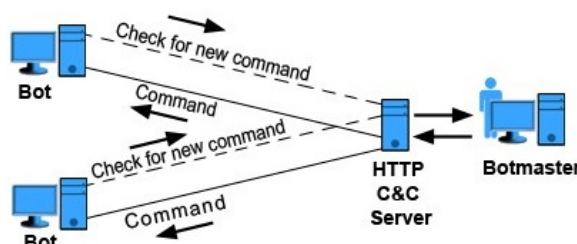


Figura 1.1: Struttura di una botnet

Quando il protocollo HTTP nacque nel 1999, nessuno avrebbe mai pensato che sarebbe stato utilizzato per le botnet. La prima generazione di botnet utilizzava l'*Internet Relay Chat* (IRC<sup>2</sup>) e relativi canali per instaurare un meccanismo di “controllo e comando”. I bot IRC seguono lo stesso approccio PUSH di quando ci si unisce ai canali, rimanendo connessi. Essi si connettono ai server IRC e ai canali che sono stati selezionati dal *botmaster* e attendono comandi. Invece di rimanere connessi, i bot

---

<sup>2</sup>Protocollo di messaggistica istantanea su Internet, che consente sia la comunicazione diretta fra due utenti, che il dialogo contemporaneo di gruppi di persone raggruppati in stanze di discussione dette canali.

HTTP controllano periodicamente per aggiornamenti oppure nuovi comandi: questo modello è detto di PULL e continua ad intervalli regolari definiti dal botmaster, che usa il protocollo HTTP per nascondere le proprie attività tra il normale flusso web, riuscendo ad evitare facilmente i metodi di rivelazione come i *firewall*.

## 1.3 Obiettivo

L'obiettivo è quello di sviluppare un software per workstation che definisca bot in grado di contattare delle URL a cui sono associati una serie di parametri fondamentali:

1. la periodicità di contatto (fissa o variabile in un intervallo temporale);
2. il numero massimo di contatti da effettuare;
3. una eventuale modalità di “*sleep*”, intesa come insieme di condizioni in cui non viene effettuata nessuna azione ;
4. un eventuale user-agent “*custom*”;
5. l'indirizzo ip e la porta di un eventuale “*proxy*” pubblico.

Tali parametri saranno impostati tramite un file di testo (pre-compilato oppure configurabile mediante una *Graphic User Interface*).

I contatti effettuati e i parametri di configurazione in uso verranno salvati su un file di log; le informazioni principali relative alla macchina su cui il codice è in esecuzione saranno salvate nel file *sys\_info.txt*.

# Capitolo 2

## Panoramica del progetto

*Nel capitolo verrà effettuata una breve introduzione alle componenti principali del progetto, verrà esposto il percorso di progettazione seguito per un suo corretto sviluppo, e illustrato l'utilizzo del software da parte dell'utente.*

### 2.1 Componenti principali

Le entità principali presenti nel software realizzato sono un *Parser*, con il ruolo di interpretare le informazioni contenute nel file di configurazione, uno *Scheduler*, che ha il compito di garantire la corretta esecuzione dei task al termine delle rispettive scadenze, un protocollo di comunicazione (*HTTP*) con cui interrogare le macchine specificate. Nello sviluppo dell'applicativo *multi-piattaforma* è stato utilizzato il *Singleton pattern*, per evitare la creazione di istanze multiple dello scheduler o del logger. In ottica di una futura integrazione del software in una rete di bot, sono state inserite funzioni per il calcolo di un ID univoco per ogni macchina appartenente alla rete controllata.

### 2.1.1 HTTP

**HTTP**, acronimo di *hypertext transfer protocol*. È un protocollo a livello applicativo utilizzato per la trasmissione di informazioni tramite un meccanismo di richiesta/risposta tra *client* e *server*. All'interno del messaggio di richiesta sono definiti diversi campi tra cui il tipo di richiesta (*GET*, *POST*) e lo *UserAgent*, che identifica la tipologia di *client* utilizzata. Il metodo *GET* può essere utilizzato per effettuare una richiesta:

- **assoluta**, senza ulteriori specificazioni sulla risorsa cercata;
- **condizionale**, se nell'*header* del messaggio di richiesta sono presenti i campi *If-Modified-Since*, *If-Match*, etc.;
- **parziale**, quando la risorsa richiesta è una sottoparte di una risorsa memorizzata.

Nel software sono effettuate solo *GET* assolute, e la scelta dello *user-agent* è lasciata alla discrezione dell'utilizzatore.

### 2.1.2 Singleton Pattern

In determinati contesti è necessario che venga istanziato un solo esemplare di una classe. Ad esempio: per evitare due riproduzioni audio contemporanee, sarà opportuno istanziare un unico riproduttore musicale, uno *spooler* di stampa dovrebbe tenere una sola coda anche se sono presenti più stampanti attive, etc.

Per garantire questa singola istanziazione è sufficiente rendere impossibile l'utilizzo del costruttore della classe da parte del resto del codice, e fornire un metodo indiretto per ottenere l'unica istanza di tale classe.



### 2.1.3 Scheduler

Lo *scheduler* ha il compito di garantire la corretta esecuzione dei task al termine delle rispettive scadenze.

Nell'applicativo è stato utilizzato uno *Scheduler Executor Service*, istanziato una sola volta mediante l'uso del *Singleton pattern*; data la presenza di istruzioni bloccanti nei task da eseguire, è stato messo a disposizione dello scheduler un pool di 30 *thread*.

La classe utilizzata permette la schedulazione di job periodici tramite la funzione *scheduleAtFixedRate* ma, data la necessità di schedulare task con rate non fissato, è stato deciso di effettuare le singole ri-schedulazioni utilizzando la più semplice funzione *schedule*.

### 2.1.4 ID dei bot e MD5

In una rete reale, i *bot* sono univocamente identificabili dal centro di comando e controllo: l'attaccante ha interesse nel conoscere il numero di *bot* attivi per effettuare un attacco in un determinato istante; questa informazione inoltre risulta fondamentale, a livello economico, se egli decide di vendere o affittare la *botnet* ad un acquirente esterno.

Per garantire l'unicità dell'identificativo di ogni bot, viene effettuato l'*hash md5* delle informazioni riguardanti l'hardware della macchina e il sistema operativo in esecuzione.

### 2.1.5 Parsing

La corretta interpretazione del file di configurazione è garantita dal *Parser* che, mediante la conoscenza del protocollo utilizzato per l'organizzazione delle informazioni e

la manipolazione di oggetti di tipo stringa, genera dal file l'*ArrayList* dei task in attesa di esecuzione.

La classe è utilizzata anche per l'operazione di scrittura del file di configurazione a partire dai *form* riempiti dall'utente nell'interfaccia principale dell'applicazione.

### 2.1.6 Multi-piattaforma

Quando si parla di un programma multi-piattaforma si intende un programma in grado di funzionare correttamente su diversi sistemi operativi.

Sono stati gestite quindi le funzionalità dipendenti dalla piattaforma su cui l'applicativo è in esecuzione (come la generazione dell' ID univoco del bot o l'individuazione dei browser installati), differenziandone il comportamento sulla base del sistema operativo presente sulla macchina.

### 2.1.7 Multi-threading

La presenza di istruzioni bloccanti nei *task* ha reso necessario l'utilizzo, da parte dello *scheduler*, di un *thread pool* che garantisse un miglior tempo di risposta medio ai *job* eseguiti.

## 2.2 Piattaforma di sviluppo, Swing, Awt

NetBeans è un ambiente di sviluppo integrato (*IDE - Integrated Development Environment*) multi-linguaggio, scelto dalla Oracle Corporation come *IDE* ufficiale da contrapporre al più diffuso Eclipse.

NetBeans utilizza due componenti principali: la piattaforma, che comprende una serie di librerie per fornire gli elementi base dell'*IDE* come presentazione dei dati e interfaccia

cia utente, e l'*IDE* vero e proprio, che permette di gestire il controllo e le funzionalità offerte dalla piattaforma. NetBeans utilizza *Abstract Window Toolkit (AWT)*, un insieme di API realizzate da Sun che permettono agli sviluppatori di modellare le interfacce grafiche delle finestre, pulsanti e altri elementi visuali. *AWT* fornisce gli elementi grafici base che dipendono dalla piattaforma utilizzata, mentre per gli aspetti di alto livello come gestione di colori e interazione con l'utente è usata la libreria Swing.

## 2.3 Build your own botnet

In questa sezione sono presentati la struttura del progetto ed i principali casi d'uso dell'applicativo realizzato.

### 2.3.1 Struttura del progetto

Il progetto è composto da diverse classi:

- **ByobComm**, responsabile del protocollo di comunicazione;
- **ByobSingleton**, tramite il quale è stato implementato il *Singleton pattern*;
- **ByobTask**, rappresenta il singolo *task* che deve essere schedato;
- **Byob\_v1**, la classe principale, da cui viene avviata la GUI;
- **GUI**, responsabile di ciò che riguarda la creazione e gestione della *Graphic User Interface* ;
- **Parser**, responsabile della creazione di file, lettura e scrittura dei parametri di configurazione inseriti dall'utente;

- **Tools**, contenente varie funzioni e metodi utili in diversi contesti;
- **URLDetails**, contenente i dati (parametri di configurazione) del singolo *contatto* da effettuare.

### 2.3.2 ByobComm

La classe *ByobComm* gestisce la comunicazione HTTP. Se presente, viene configurato il *proxy* specificato nel file di configurazione; si specifica il tipo di metodo HTTP utilizzato (GET), la codifica dell'*header* e lo *user agent*. Per la connessione si è utilizzata la classe *HttpURLConnection*.

### 2.3.3 ByobSingleton

Responsabile del *Singleton pattern* utilizzato, contiene al suo interno il *Logger* dell'applicazione, che trascrive:

- i contatti e i parametri di configurazione;
- il *timestamp* di contatto e il dettaglio delle URL contattate;
- le informazioni relative al Sistema Operativo e ai browser presenti sulla postazione su cui il software è installato.

Oltre al *Logger*, la classe comprende al suo interno lo *scheduler* dei *task* da eseguire.

### 2.3.4 ByobTask

*ByobTask* rappresenta il singolo task che lo *Scheduler Executor Service* deve eseguire. Essa implementa l'interfaccia *Runnable*, ed ogni istanza della classe ri-schedula la pro-

pria esecuzione in accordo con le informazioni immagazzinate nel contatto *URLDetails* a cui è associata.

### 2.3.5 GUI

La classe *GUI* rappresenta l'interfaccia grafica tramite la quale l'utilizzatore può fornire i parametri di configurazione dell'applicazione. Le azioni principali che possono essere intraprese sono due:

1. l'apertura di un file di configurazione già compilato (**Figura 2.1**);
2. la creazione di un nuovo file di configurazione.

Alla pressione del tasto "...", un *JFileChooser* permetterà la ricerca di un file di testo all'interno delle directory presenti nel dispositivo di memorizzazione in uso (**Figura 2.2**).

Una volta effettuata la selezione, il percorso assoluto del file scelto verrà scritto nella casella di testo relativa, e sarà attivato il tasto "Launch" per l'avvio dell'applicazione (**Figura 4.1**).

Tramite l'inserimento manuale dei parametri di configurazione, è possibile specificare (**Figura 2.4**):

- la **URL** da contattare;
- il **Contact Time**, ossia la periodicità di contatto che può essere:
  - Fissa ("*Fixed*");
  - Intervallo ("*Interval*"), cioè all'interno di un intervallo temporale;

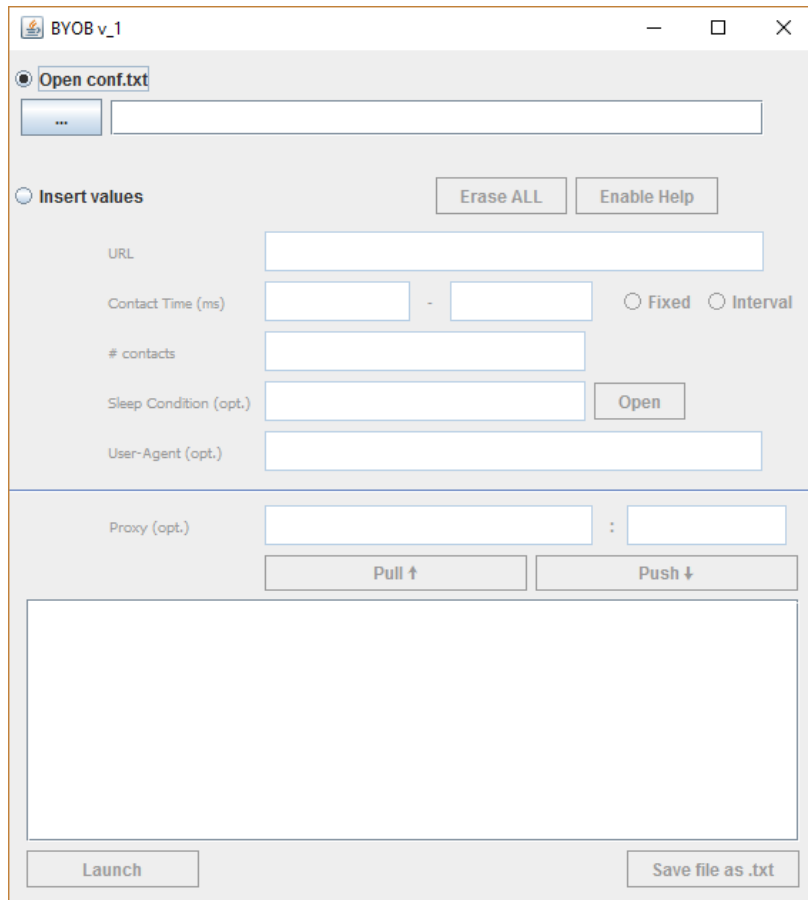


Figura 2.1: Apertura di un file di configurazione

- **#contacts**, ossia il numero massimo di contatti da effettuare verso la URL specificata;
- le **Sleep conditions** (campo opzionale), ovvero l'insieme delle condizioni temporali sotto le quali non deve essere svolta alcuna azione;
  - Condizione sui giorni pari (E), sui giorni dispari (O), nessuna restrizione;
  - Condizione sulle prime dodici ore (A), condizione sulle ultime 12 ore (P), nessuna restrizione;

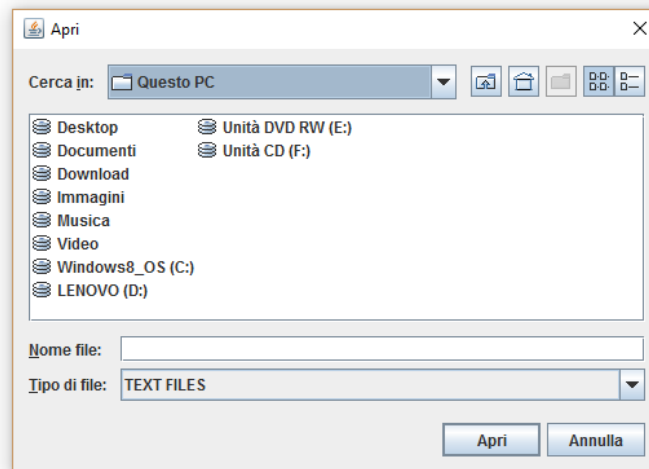


Figura 2.2: JFileChooser

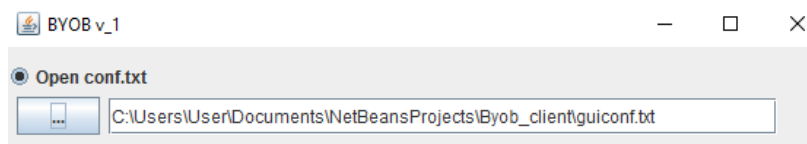


Figura 2.3: Scelta del file di configurazione

- lo **User-Agent** (opzionale), ossia la modifica da apportare al campo User-Agent dell'header HTTP;
- il **Proxy** (opzionale), ossia l'indirizzo IP e la porta di un proxy pubblico da utilizzare.

Una volta inseriti nelle relative caselle di testo, il tasto "Push" permette di visualizzare tali parametri nell'area sottostante (**Figura 2.6**).

Il tasto "Pull" permette di estrarre dall'area di testo l'ultimo inserimento effettuato per una eventuale modifica dei parametri non correttamente inseriti.

Al termine dell'inserimento manuale, la pressione del tasto "Save file as .txt" permette il salvataggio, tramite procedura guidata, del file di configurazione appena creato.

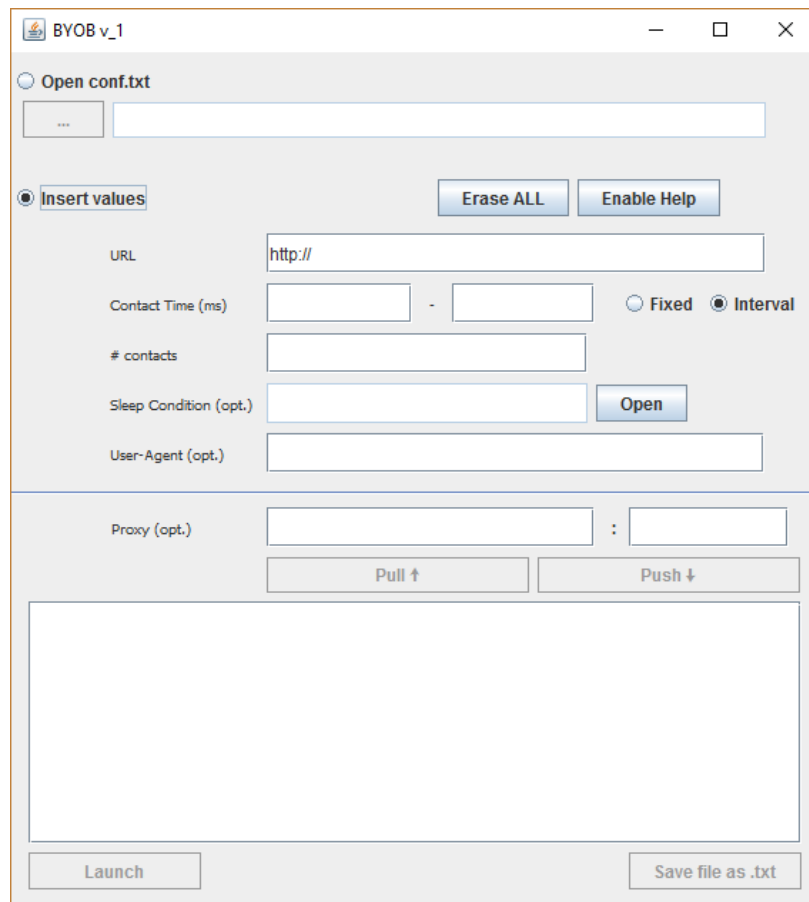


Figura 2.4: Inserimento manuale parametri di configurazione

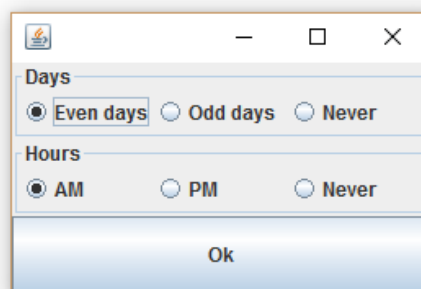


Figura 2.5: Sleep conditions

**GUI user-friendly** La pressione del tasto "Enable Help" permette la comparsa (al passaggio del mouse) di *tooltip* sulle caselle del form, rendendone più immediata la corretta compilazione (**Figura 2.7**).



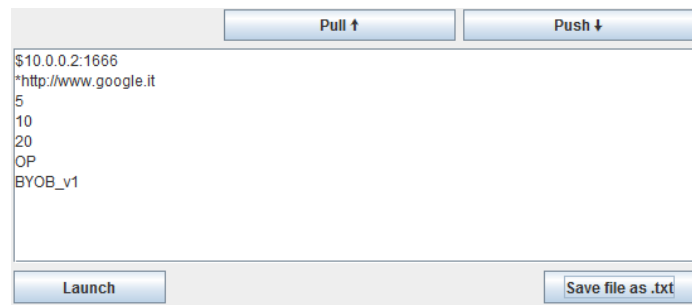


Figura 2.6: Push - Inserimento dell'input nell'area di testo

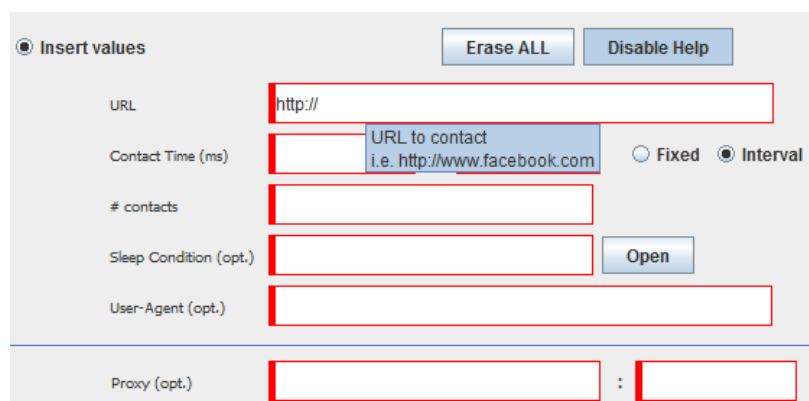


Figura 2.7: Utilizzo dell'help

La pressione del tasto "Erase ALL", da utilizzare in caso di molteplici errori, causa la cancellazione di tutto il file di configurazione compilato.

Nel caso vengano riscontrati degli errori al momento della pressione del tasto "Push", un popup guiderà l'utilizzatore nella risoluzione dei problemi individuati (**Figura 2.8**).

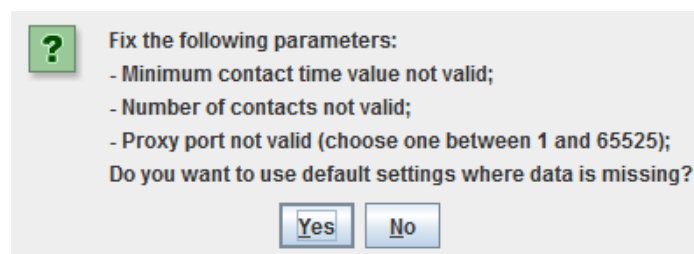


Figura 2.8: Esempio di messaggio d'errore per dati mancanti e/o errati

### 2.3.6 Parser

La classe *Parser* è responsabile di:

- lettura e scrittura del file contenente i parametri di configurazione;
- creazione di una lista di istanze della classe *URLDetails*, una per ciascun contatto da effettuare;
- conversione dei parametri di configurazione;
- controllo di validità per ciascun parametro di configurazione.

### 2.3.7 Tools

La classe *Tools* contiene al suo interno diversi metodi statici, utili in più sezioni del programma. Essa è responsabile della generazione dell'ID del bot, della prima schedulazione dei task programmati, della raccolta delle informazioni relative al sistema operativo e ai browser installati sulla macchina, dei controlli per la generazione dei *warning messages* della classe GUI.

### 2.3.8 URLDetails

*URLDetails* è la classe che contiene le informazioni inserite dall'utente per ciascun contatto che deve essere effettuato.

# Capitolo 3

## Implementazione

*Sono di seguito presentati dettagli riguardanti l'implementazione dei passi principali eseguiti dal programma in esame.*

### 3.1 Avvio iniziale

All'avvio dell'applicazione, viene visualizzata la schermata principale di configurazione; vengono inoltre raccolte e scritte sul file *sys\_info.txt* informazioni riguardanti il sistema operativo della macchina e le versioni dei browser installati.

```
public class Byob_v1 {  
  
    public static void main(String[] args) {  
        /**Start the GUI*/  
        GUI frame = new GUI();  
        final Toolkit toolkit = Toolkit.getDefaultToolkit();  
        final Dimension screenSize = toolkit.getScreenSize();  
        int x = (screenSize.width - frame.getWidth()) / 2;  
        int y = (screenSize.height - frame.getHeight()) / 2;  
        frame.setTitle("BYOB v_1");  
        frame.setLocation(x, y);  
        frame.setVisible(true);  
        /**Gather system informations and write them on sys_info.txt*/  
        Tools.writeInfoFile("sys_info.txt");  
    }  
}
```

## 3.2 Informazioni di sistema

Il sistema operativo in esecuzione sulla macchina è restituito dalla funzione *Tools.getOs()*.

```
public static String getOs(){  
    return System.getProperty("os.name");  
}
```

Per riuscire ad identificare i browser installati, sono state adottate strategie differenti per ogni sistema operativo individuato; la funzione *Tools.getBrowsers()* invoca *Tools.getOs()* e distingue le azioni da intraprendere:

```
public static String getBrowsers(){  
  
    String browsers = "";  
    String os = getOs().toLowerCase();  
    if(os.contains("linux")){  
        [...]  
    } else if(os.contains("windows")){  
        [...]  
    } else if(os.contains("mac")){  
        [...]  
    } else {  
        /**Couldn't recognize OS*/  
    }  
    return browsers;  
}
```

### 3.2.1 Linux

I browser ritenuti più comuni in ambiente *Linux* sono stati:

- Google Chrome
- Mozilla Firefox
- Opera
- Chromium

Per identificare l'eventuale versione installata di ogni browser, viene avviato un nuovo processo che esegue la *bash* invocando il programma relativo ad ogni browser con il parametro *--version*.

```
[...]
String tmp = unixTermOut("firefox --version");
[...]

private static String unixTermOut(String cmd){
    String[] args = new String[] {"/bin/bash", "-c", cmd};
    String out = "";
    try {
        Process proc = new ProcessBuilder(args).start();
        BufferedReader br = new BufferedReader(
            new InputStreamReader(proc.getInputStream()));
        out = br.readLine();
    } catch (IOException ex) {
        [...]
    }
    return out;
}
```

### 3.2.2 Windows

I browser ritenuti più comuni in ambiente *Windows* sono stati:

- Internet Explorer
- Google Chrome
- Mozilla Firefox

Per individuare in modo sistematico le versioni installate, si è scelto di interrogare il registro di sistema di Windows. Ciò è stato possibile grazie all'utilizzo di una libreria esterna, la *Java Native Access*<sup>1</sup>, in cui il package *com.sun.jna.platform.win32.Advapi32Util*

<sup>1</sup><https://github.com/java-native-access/jna#readme>

offre un'interfaccia semplice e immediata per l'accesso e la manipolazione dei registri di sistema.

```
[...]
String path = "SOFTWARE\\Microsoft\\Internet Explorer";
String vField = getOs().toLowerCase().equals("windows 8")? "svcVersion" :
"Version";
String version = Advapi32Util.registryGetStringValue(
WinReg.HKEY_LOCAL_MACHINE, path, vField);
[...]
```

### 3.2.3 Mac OSx

I browser ritenuti più comuni in ambiente *Mac OSx* sono stati:

- Google Chrome
- Mozilla Firefox
- Opera
- Safari

Come per *Linux*, per identificare l'eventuale versione installata di ogni browser, viene avviato un nuovo processo che esegue la *shell* di sistema avviando il programma *system\_profiler* con parametro *SPApplicationDataType*. L'output offerto dal profiler di sistema contiene al suo interno tutto il software installato sulla macchina, compreso numero di versione e autori. Tutte le informazioni vengono salvate all'interno del file *mac\_profile.txt*, da cui successivamente vengono estratte le versioni relative al software cercato.

```
[...]
linuxTermOut("system_profiler SPApplicationDataType > mac_profile.txt");
[...]
String[] args = new String[] {"/bin/bash", "-c", "grep
```

```

-e \"Google Chrome:\" -e \"Firefox:\" -e \"Opera:\" -e \"Safari:\"
-A 2 mac_profiler.txt});
String str = linuxTermOut(args);

```

### 3.3 File di configurazione

Il file di configurazione permette di impostare i parametri principali delle comunicazioni da effettuare verso l'esterno. Per ogni contatto è necessario definire una URL, la periodicità di contatto (che può essere fissa o scelta randomicamente in un intervallo pre-impostato) ed il numero massimo dei contatti da effettuare. È inoltre possibile impostare un proxy tramite il quale effettuare le connessioni, uno *user agent* differente da quello di default ed un set di condizioni sotto le quali non viene effettuata la connessione alla URL specificata.

Il file di configurazione consiste in un file di testo formattato nel seguente modo:

```

$proxy_ip:proxy_port /**Opzionale*/
*URL_1
minimo_intervallo_di_contatto
massimo_intervallo_di_contatto
numero_di_contatti_effettuabili
condizioni_di_sleep
user_agent
*URL_2
[...]
```

#### 3.3.1 Immissione e scrittura

Nel caso si scelga di inserire manualmente i parametri di configurazione, l'interfaccia mette a disposizione delle *JFormattedTextField*. Esse rappresentano un modo semplice per specificare l'insieme dei caratteri accettati tramite un "formattatore": nel caso di campi numerici come *Contact Time*, *#contacts* e la porta del *proxy*, è stato utilizzato *NumberFormatter*; nel caso del campo IP del *proxy*, è stata utilizzata la

classe *RegexFormatter*, essendo l'indirizzo IP caratterizzato da un *pattern* particolare, ovvero xxx.xxx.xxx.xxx, dove "xxx" è compreso tra 0 e 255.

In questa classe, viene costruita una espressione regolare che specifica il tipo di input:

```
String _255 = "(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
```

- "?" specifica che la restante espressione non è parte di un gruppo "cattura"<sup>2</sup>;
- "25[0-5]" specifica che è necessario un *match* del tipo "prima cifra pari a 2, seconda cifra pari a 5 e terza cifra compresa tra 0 e 5";
- "|" specifica *OR*;
- "[01]?[0-9][0-9]?" specifica che è necessario un *match* del tipo "0 o 1 o nulla; qualsiasi cifra; qualsiasi cifra o nulla".

Successivamente viene creato il pattern completo:

```
Pattern p = Pattern.compile( "^(?:" + _255 + "\\.){3}" + _255 + "$" );
```

- "^" specifica l'inizio dell'input;
- "?" specifica l'inizio di un gruppo di non "cattura";
- "String \_255" specifica il *pattern* che deve trovare un *match*;
- "\\." è necessario come parte del *pattern* per "escape" del periodo, il quale altrimenti farebbe il *matching* su ogni carattere.
- "3" fa il *matching* del gruppo, appena terminato, 3 volte;

---

<sup>2</sup>Nelle espressioni regolari, l'utilizzo di parentesi tonde o quadre permette di specificare "gruppi", il cui vantaggio è poter applicare, per esempio, quantificatori differenti a ciascuno di essi.



- `" + _255 + "$"` esegue nuovamente il *matching*, con il pattern specificato.

infine viene creato l'oggetto *RegexFormatter* con il *Pattern* `p`:

```
RegexFormatter ipFormatter = new RegexFormatter(p);
```

Tra tutti i parametri di configurazione, solamente *sleep condition* non può essere inserito manualmente, in quanto ogni sua condizione è rappresentata tramite una lettera dell'alfabeto. A tal proposito si è pensato di fornire all'utente un'esperienza semplificata, attraverso l'utilizzo di un *popup* e di due gruppi di *JRadioButton*. Una volta confermata la selezione, vengono restituite le lettere rappresentanti le condizioni nella relativa *JFormattedTextField*.

Lo scopo della *JTextArea* è quello di permettere la visualizzazione generale di tutte le informazioni inserite.

Una volta inseriti i dati, si hanno a disposizione due *JButton* chiamati "Push" e "Pull". Alla pressione del tasto "Push", vengono effettuati i controlli sui valori inseriti, che andranno a popolare la *JTextArea*. Se è stato commesso un errore nell'inserimento dei parametri di configurazione oppure è stato dimenticato un valore non opzionale, viene visualizzato un messaggio di *warning*.

```
public static List<String> warningMessage(String[] params){
    List<String> warning = new ArrayList<>();
    warning.add("Fix the following parameters:\n");
    if(!Parser.checkNumber(params[1]))
        if(params[2].equals("-"))
            warning.add("- Contact time value not valid;\n");
        else
            warning.add("- Minimum contact time value not valid;\n");
    if(!Parser.checkNumber(params[2]))
        if(!params[2].equals("-"))
            warning.add("- Maximum contact time value not valid;\n");
    if(!Parser.checkNumber(params[3]))
        warning.add("- Number of contacts not valid;\n");
    if(!params[6].equals(" ") && !Parser.checkIPv4String(params[6]))
```

```

        warning.add("- Proxy IP not valid;\n");
    if(!params[7].equals(" ") && !Parser.checkPort(params[7]))
        warning.add("- Proxy port not valid (choose one between 1
            and 65525);\n");
    if(warning.size() == 1)
        return null;
    else {
        warning.add("Do you want to use default settings where data
            is missing?");
        return warning;
    }
}

```

il messaggio indirizza l'utente verso una facile soluzione del problema, eventualmente tramite la selezione dei valori di *default*. Esso viene generato controllando che:

- *URL*, *contactTime* e *#contacts* siano stati correttamente inseriti;
- *contactTime*, *#contacts*, *proxy port* siano numeri interi;
- *proxy IP* segua una formattazione adeguata e valida.

Alla pressione del tasto "Pull" viene prelevato il *set* di parametri inserito nella *jTextArea* e riportato, campo per campo, nelle rispettive *jFormattedTextField*. È quindi possibile effettuare le modifiche opportune e utilizzare nuovamente il tasto "Push" per re-importare i parametri nella *jTextArea*.

Al termine della scrittura del file di configurazione, la pressione del tasto "Save" aprirà una finestra nella quale sarà possibile scegliere la directory di salvataggio del file.

Indirizzo IP e porta del proxy, qualora inseriti, verranno scritti direttamente nel file salvato.

```
try { FileWriter fw = new FileWriter(fileToWrite.getAbsolutePath());
```

```

BufferedWriter bw = new BufferedWriter(fw);
for (int i = 0; i < params.length; i++) {
    String param = params[i];
    bw.write(param);
    if (i < params.length - 1)
        bw.newLine();
}
bw.close();

```

Al termine del salvataggio viene abilitato il tasto "Launch": alla sua pressione saranno eseguiti tutti i comandi specificati precedentemente nel file di configurazione.

### 3.3.2 Lettura e caricamento

Nel caso si scelga di caricare un file di configurazione creato precedentemente, la pressione del tasto "..." porterà all'apertura del *jFileChooser* in una nuova finestra. Al termine della selezione, il percorso assoluto del file sarà visualizzato nella relativa *jFormattedTextField*. Cliccando sul  *JButton*  "Launch", verranno eseguiti tutti i comandi specificati nel file selezionato.

Tale lettura avviene tramite *BufferedReader*.

Se la prima riga del file inizia con il carattere "\$", ne verrà effettuato il parsing alla ricerca dell'indirizzo IP e della porta del *proxy* che si intende utilizzare.

```

while ((url = br.readLine()) != null) {
    //Search at the beginning of the configuration file for
    //proxy setup
    [...]
    if (url.charAt(0) == '$'){
        String[] proxyDet = splitString(url.substring(1), ":");
        URLDetails.setProxy(proxyDet[0],
            Integer.parseInt(proxyDet[1]));
        System.out.println(proxyDet[0] + ":" + proxyDet[1]);
        continue;
    }
}

```

Secondo il protocollo utilizzato, la riga in cui è specificato l'indirizzo della *URL* da contattare avrà un asterisco ("\*") come carattere iniziale.

```
// Check URL identification char
if (!url.contains("*")){
    System.err.println("Error in configuration file , aborting");
    System.exit(-1);
}
```

Vengono scandite, quindi, le righe successive del file (fino alla fine o alla successiva stringa che inizia con "\*"), allo scopo di creare una stringa del tipo:

URL;minT;maxT;numC;sleepC;userAgent;

Essa viene divisa ed i valori che la compongono sono utilizzati per la creazione di oggetti di tipo *URLDetails*, raggruppati nell'*ArrayList* "TaskList".

```
// Build the contact string ("URL;minT;maxT;numC;sleepC;userAgent;")
String contact = url.substring(1);
String line;
for (int i = 0; i < URLDetails.NUM_FIELDS - 1; i++){
    if ((line = br.readLine()) != null)
        contact = contact + delim + line;
}
// Build detail string array
String[] detail = splitString(contact, delim);
// Build URLDetails obj and add to configuration arrayList
URLDetails det = convertParam(detail);
[...]
configuration.add(det);
```

I task vengono schedulati tramite la funzione *Tools.schedule*.

```
Parser parser = new Parser(fileConfPath);
try {
    ArrayList <URLDetails> taskList = parser.readConfigurationFile();
    Tools.schedule(taskList);
} catch (IOException ex) {
    ByobSingleton.myLogger.severe("Parser I/O exception");
}
```

## 3.4 Lancio

Dopo aver selezionato un file di configurazione, cliccando sul tasto "Launch" viene eseguita la parte principale del progetto.

L'ArrayList di *URLDetails* viene passato alla funzione *Tools.schedule* che, per ogni elemento estratto dall'ArrayList, crea un'istanza della classe *ByobTask* e ne richiede la schedulazione allo *SchedulerExecutorService*.

```
public static void schedule(ArrayList<URLDetails> task) {  
    for(int i = 0; i < task.size(); i++) {  
        ByobSingleton.ses.schedule(new ByobTask(task.get(i)), 0,  
                                   TimeUnit.MILLISECONDS);  
    }  
}
```

### 3.4.1 Schedulazione dei task

*ByobTask* rappresenta il singolo task che deve essere eseguito dallo scheduler; ogni task è legato ad una connessione, ovvero ad un'istanza di *URLDetails*, che contiene tutti i dettagli delle comunicazioni da effettuare.

Nel metodo *run*, che viene sovrascritto dalla classe, sono dapprima controllate le condizioni di *sleep*: se una di queste risulta verificata, il task viene ri-schedulato dopo un intervallo che va da 30 a 45 minuti, al termine dei quali si effettuerà nuovamente il check delle condizioni; se nessuna condizione è verificata, allora viene decrementato il numero di contatti ancora da effettuare, ri-schedulato il task in esame ed eventualmente viene inviata una *GET http* alla *URL target*.

I dettagli del contatto avvenuto sono scritti sul file di log, così come l'eventuale risposta (qualora specificato) da parte del server.

```

public class ByobTask implements Runnable {

    final static ScheduledExecutorService ses =
        ByobSingleton.getInstance().ses;
    URLDetails contact;
    [...]

    @Override
    public void run() {

        if(contact.sleepMode()) {
            /**Sleep mode: try again in 30/45 minutes */
            int minTimeRestInterval = 30; //Minutes
            int maxTimeRestInterval = 45; //Minutes
            long randomInterval = minTimeRestInterval +
                random.nextInt(maxTimeRestInterval - minTimeRestInterval + 1);
            [...]
            synchronized(ses){
                ses.schedule(this, randomInterval, TimeUnit.MINUTES);
            }
        } else {
            /**Synchronized function*/
            if (contact.decreaseContactNum() < 0)
                return;
            else if(contact.getContactsNum() > 0){
                [...]
                synchronized(ses){
                    ses.schedule(this, (long)randomInterval,
                        TimeUnit.MILLISECONDS);
                }
            }
            /**Write to log file*/
            [...]
            int code = ByobComm.httpGet(contact.getUrl(),
                contact.getUserAgent(), URLDetails.proxyIp,
                URLDetails.proxyPort, contact.waitForResponse);
            [...]
        }
    }
}

```

### 3.4.2 Comunicazione HTTP

I contatti (http GET) sono effettuati tramite chiamata a una funzione statica della classe *ByobComm*. Il metodo *httpGet* permette di specificare, oltre alla URL da contattare, anche uno *user-agent* personalizzato, l'indirizzo IP e la porta di un server *proxy* che si desidera utilizzare.

```
static int httpGet(String url, String userAgent, String proxyIp,
                  int proxyPort, Boolean waitForResponse) {
    String charset = "UTF-8";
    HttpURLConnection connection;
    try {
        if(proxyPort > 0){
            Proxy proxy = new Proxy(Proxy.Type.HTTP,
                                   new InetSocketAddress(proxyIp, proxyPort));
            connection = (HttpURLConnection)
                new URL(url).openConnection(proxy);
        } else {
            connection = (HttpURLConnection) new URL(url).openConnection();
        }
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Accept-Charset", charset);

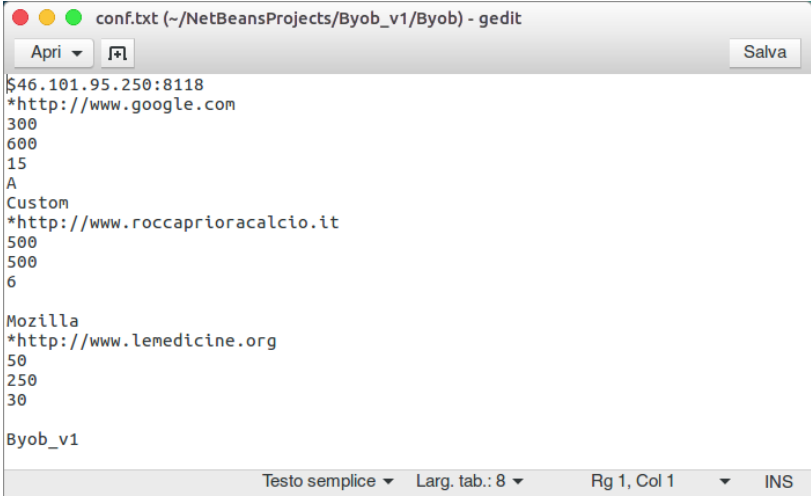
        if(!userAgent.isEmpty())
            connection.setRequestProperty("User-Agent", userAgent);
        else
            connection.setRequestProperty("User-Agent", "");
        connection.connect();
        int ret = waitForResponse ? connection.getResponseCode() : 0;
        connection.disconnect();
        return ret;
    } catch (MalformedURLException ex) {
        ByobSingleton.getInstance().myLogger.severe("MalformedURLException");
        return -1;
    } catch (IOException ex) {
        ByobSingleton.getInstance().myLogger.severe("IOException");
        return -2;
    }
}
```

# Capitolo 4

## Testing

Il software è stato testato con successo sui sistemi operativi *Windows 10*, *Ubuntu 16.04* e *Mac OSx El Capitan*.

Sono di seguito presentati esempi del file di configurazione (**Figura 4.1**), del file contenente le informazioni di sistema (**Figura 4.2**) e del file di log (**Figura 4.3**).



```
conf.txt (-/NetBeansProjects/Byob_v1/Byob) - gedit
Apri  Salva
$46.101.95.250:8118
*http://www.google.com
300
600
15
A
Custom
*http://www.roccaprioracalcio.it
500
500
6
Mozilla
*http://www.lemedicine.org
50
250
30
Byob_v1
Testo semplice  Larg. tab.: 8  Rg 1, Col 1  INS
```

Figura 4.1: File di configurazione



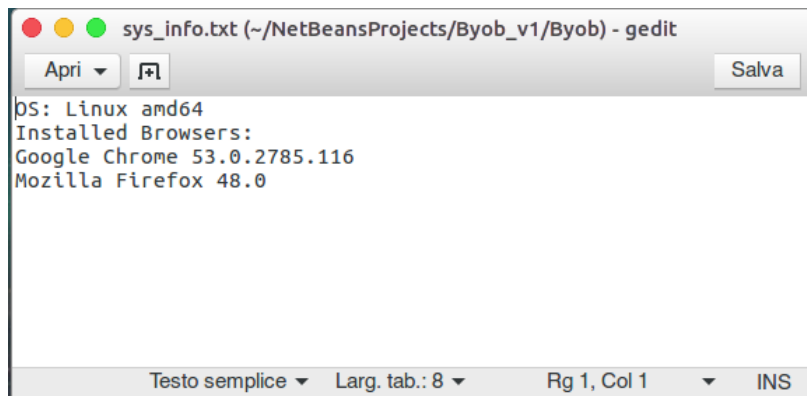


Figura 4.2: Informazioni del Sistema

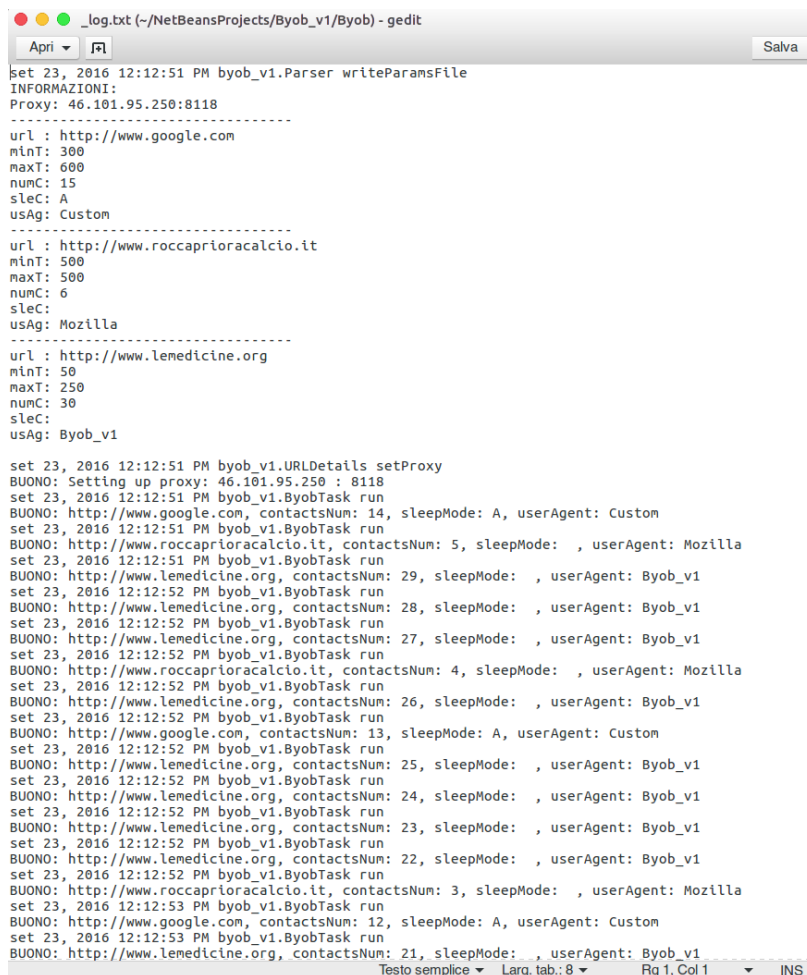


Figura 4.3: File di log

La verifica dei contatti avvenuti con le *URL* specificate è stato facilitato dall'utilizzo di un software di monitoraggio grafico della rete chiamato *EtherApe*<sup>1</sup>

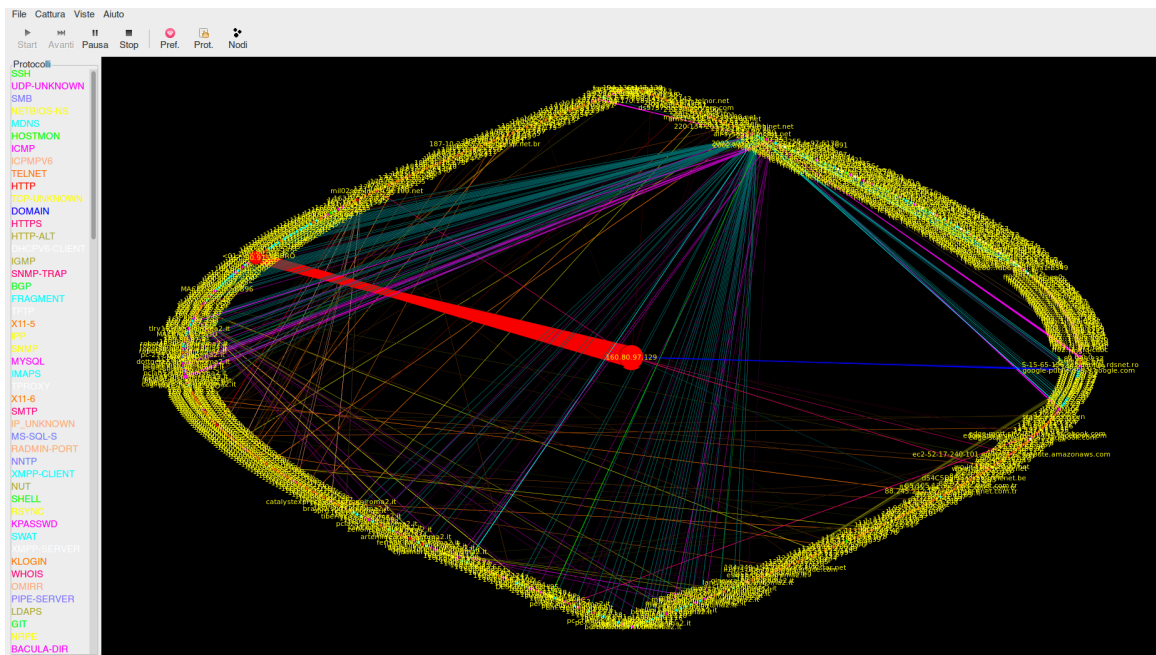


Figura 4.4: Graphical network monitor: etherApe

---

<sup>1</sup><http://etherape.sourceforge.net/>

# Capitolo 5

## Installazione

Il software non viene rilasciato con un *Installer*, ma è possibile effettuare l'esecuzione in maniera diretta (previa installazione di un *Java Runtime Environment*):

- in ambiente Windows è sufficiente effettuare un doppio click sul file *Byob\_v1.jar*;
- in ambiente Unix, basta eseguire da terminale il comando

```
java -jar Byob_v1.jar
```

All'inizio dell'esecuzione, verrà creata la cartella *Byob* nella directory corrente, che conterrà i file *\_log.txt* e *sys\_info.txt*.