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
0213347 - Nazio Alessio - alessio.nazio@gmail.com

Indice

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 per attacchi diretti, orchestrati da malintenzionati, e/o per attacchi di tipo indiretto, mascherati dietro programmi che l'utente usa 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[1]", chiamata una **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) e un *botmaster*; si basa, inoltre, su quattro concetti chiave:

[1] 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, come la diffusione di un virus, effettuare attacchi DDos, spam, etc.

1.2 HTTP Botnet

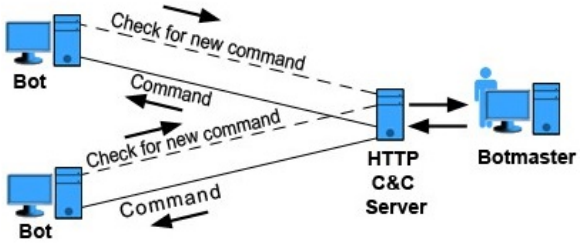

Figura 1.1: Struttura di una botnet

Quando il protocollo HTTP nacque nel 1999, nessuno avrebbe mai pensato che sarebbe stato utilizzato per le botnet. La prima generazione di botnet usava l'*Internet Relay Chat* (IRC[2]) 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

[2]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.

selezionati dal *botmaster* e attendono comandi. Invece di rimanere connessi, i bot HTTP controllano periodicamente per aggiornamenti oppure nuovi comandi: questo modello è detto di PULL e continua ad intervalli regolari definiti dal botmaster, il quale, 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 ai punti cardine 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 Punti cardine

In base ai requisiti richiesti, i punti cardine del progetto sono:

2.1.1 HTTP

HTTP, acronimo di *hypertext transfer protocol*, costituisce il cuore delle azione intraprese dal singolo *bot*.

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, quelli a noi più tili, il tipo di richiesta (*GET*, *POST*) e lo *UserAgent*, identificativo del tipo di *client* utilizzato. Per il nostro scopo, il metodo "*GET*" è usato per ottenere il contenuto della risorsa indicata come *URI* (come può essere il contenuto di una pagina *HTML*). Può essere:

- **assoluto**, ossia quando la risorsa viene richiesta senza altre specificazioni;

- **condizionale**, ossia quando la risorsa corrisponde ad un criterio indicato nell'*header*;
- **parziale**, ossia quando la risorsa richiesta è una sottoparte di una risorsa memorizzata.

Per quanto riguarda lo *user-agent*, all'interno dell'*header HTTP*, si hanno informazioni riguardanti il tipo di agente utente, cioè il tipo di *browser* che sta effettuando la richiesta. Se modificato, permette ad un *hacker* di diventare lo *user-agent* che vuole, potendo così selezionare i *payload* di software maligni sulla base del tipo di *user-agent*. Per il nostro scopo il metodo *GET* è stato utilizzato come "assoluto" e si è messo a disposizione dell'utente la possibilità di poter modificare lo *user-agent* a proprio piacimento.

2.1.2 Singleton Pattern

In molte situazioni è necessario che venga istanziato un solo esemplare di una data classe. Per esempio: se si ha un riproduttore di file musicali sarà bene che esso venga istanziato una sola volta per non ritrovarsi due riproduzioni contemporanee; uno *spooler* di stampa dovrebbe tenere una coda unica anche se ci sono piú stampanti attive; il manager del file system dovrebbe essere unico, come pure una cache; ecc. Per garantire questa singola istanziazione basta rendere impossible l'uso del costrutto *new* da parte del programma utente e di fornire un metodo indiretto per ottenere una istanza (l'unica) della classe. A tal fine occorre:

- dichiarare privato il costruttore, in modo che esso possa essere visto solo dall'interno della classe Singleton e non dal programma utente (ciò rende impossibile l'istanziazione di un oggetto dall'esterno della classe Singleton);

- prevedere il metodo (pubblico) statico e cioè di classe, in modo che esso sia comunque visibile. Questo metodo deve istanziare un esemplare se ciò non è ancora accaduto, oppure restituire l'oggetto giá istanziato in precedenza senza istanziare ulteriori esemplari.

Questa seconda scelta è quella applicata nel progetto per la creazione e gestione del *logger* e la creazione di uno schedulatore di task.

2.1.3 Schedulatore

Lo *scheduler* è un componente chiave per l'applicazione.
Grazie a questo, è stato possibile stabilire un ordinamento temporale per l'esecuzione delle azioni del singolo *bot*, così come specificato dall'utente attraverso i parametri di configurazione. La sua implementazione è stata possibile attraverso la creazione di un *executor* a singolo *thread* che schedula comandi da eseguire dopo un certo ritardo o da eseguire periodicamente[1]. Valori di ritardi pari a 0 o negativi (ma non i periodi) sono trattati come richieste di esecuzione immediata.
In sintedi, la schedulazione può essere così pensata: il *thread* dello schedulatore assegna i *task* a un *thread* estratto da un *pool*.

2.1.4 ID dei bot e MD5

I *bot* devono essere univocamente identificabili.
Questo perchè il malintenzionata che vuole effettuare un attacco (per esempio, *DDoS*) deve sapere quanti *bot* ha a disposizione in un certo lasso di tempo. Inoltre, se il malintenzionato decide di affidare un gruppo di *bot* ad un acquirente esterno (per esempio,

[1]Si noti che se il singolo *thread* termina in maniera anomale durante l'esecuzione, uno nuovo prenderá il suo posto, se necessario, per eseguire *task* successivi.

per condurre una piccola e veloce campagna di *spam*), mantenere un conteggio della potenza di attacco che possono essere offerti è necessario per accordarsi sul prezzo. Per garantire l'unicitá dell'identificativo di ciascuno, si è deciso di legare l'hardware della macchina al suo sistema operativo tramite*checksum MD5* per ottenere una stringa a 32 caratteri che è utilizzata come identificativo.
Si noti come questo è stato fatto per una integrazione con un futuro sviluppo del progetto.

2.1.5 Parsing

Per *parsing* di una stringa di caratteri si intende l'analisi della stessa per trovare *token*, oggetti o *pattern* per crearne una struttura.
Nel progetto era importante definire un *pattern* per i parametri di configurazione, per rivelare la loro correttezza e capire se si trattasse di un campo facoltativo o meno, per poter così capire se la computazione dovesse fermarsi lì. Quanto detto è stato applicato sia in lettura, sia in scrittura: questo perchè i parametri di configurazione possono essere anche salvati su un file di testo.

2.1.6 Multi-piattaforma

Quando si parla di un programma multi-piattaforma si intende un programma che sia in grado di funzionare correttamente su diversi sistemi operativi.
A tale fine, l'esecuzione di istruzioni "macchina-dipendenti" (come, per esempio, la generazione dell' ID del bot che dipende sia dall'hardware che dal sistema operativo) appariranno all'utente in maniera totalmente trasparente per sistemi operativi come OSX, Windows o Linux.

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 piú 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 utente, e l'*IDE* vero e proprio, che permette di gestire il controllo e le funzionalitá 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

Questa sezione tratta del programma e della sua architettura, nonchè delle scelte implementative rilevanti e dei casi d'uso principali.

2.3.1 Struttura del progetto

Il progetto è composto da diverse classi:

- **ByobComm**, responsabile del metodo GET del protocollo HTTP;

- **ByobSingleton**, responsabile del *log* delle azioni intraprese, delle informazioni della macchina dell'utente e dello *scheduling* dei *task* da effettuare;
- **ByobTask**, implementa l'interfaccia *Runnable* per il *multithreading*;
- **Byob_v1**, avvia il programma e l'interfaccia grafica;
- **GUI**, responsabile di tutto quello che riguarda la creazione e gestione della *Graphic User Interface* (quindi, componenti, loro visibilitá, loro aspetto e la loro abilitazione per l'uso);
- **Parser**, responsabile per la creazione di file, lettura e scrittura dei parametri di configurazione inseriti dall'utente, nonchè del loro *splitting* e della loro conversione nel giusto tipo di dato;
- **Tools**, contenente varie funzioni e vari metodi che non appartenevano a nessun'altra classe;
- **URLDetails**, contenente i dati (parametri di configurazione) del singolo *task* da effettuare.

2.3.2 ByobComm

La classe *ByobComm* gestisce tutto quello che riguarda il protocollo HTTP.
Prima di creare una connessione si controlla che tipo di parametri di configurazione ha inserito l'utente e se impattano sull'apertura della connessione. Uan volta istanziato l'oggetto (con o senza *proxy*, in base all'input dell'utente), la connessione è aperta, si setta il metodo di tipo *GET*, la codifica dell'*header* e lo *user-agent* (se è stato fornito). A questo punto si estrae il codice di risposta, prima di chiudere la connessione, che

viene analizzato all'interno del programma per i fini progettuali.

Con lo stesso approccio, si era originariamente pensato di creare una funzione booleana per il controllo dell'*URL* inserito dall'utente subito dopo aver premuto il pulsante "*Push*": purtroppo impattava sulle prestazioni dell'interfaccia grafica (il pulsante rimaneva cliccato finchè non era chiusa la connessione).

2.3.3 ByobSingleton

Come suggerisce il nome, la classe *ByobSingleton* è la responsabile per il *singleton* che si occupa di creare il *log* per la trascrizione di:

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

oltre il *log*, la classe gestisce lo schedulatore dei *task*, di cui si è parlato precedentemente.

2.3.4 ByobTask

La classe *ByobTask* si occupa della definizione del singolo *thread* (o *task*).

È stata focalizzata l'attenzione sul *polling* con intervallo esteso e casuale per verificare se non sono piú rispettate le *sleep condition*: la casualitá è stata dettata dal fatto che se il contatto avveniva sempre alla stessa ora in cui veniva schedulato poteva destare sospetti e, grazie alla sua semplicitá, permette di evitare di dover ogni volta controllare quale/i è/sono la/le *sleep condition*.

2.3.5 Byob_v1

La classe in esame non è nient'altro che la responsabile dell'invocazione della *GUI* (e del settaggio del titolo, assieme alla sua posizione per far in modo che appaia al centro dello schermo, indipendentemente dalla sua grandezza) ed è stata utilizzata a scopo di testing per trovare i giusti comandi da terminale per l'individuazione dei browser, così come richiesti nell'estensione del documento di progetto.

2.3.6 GUI

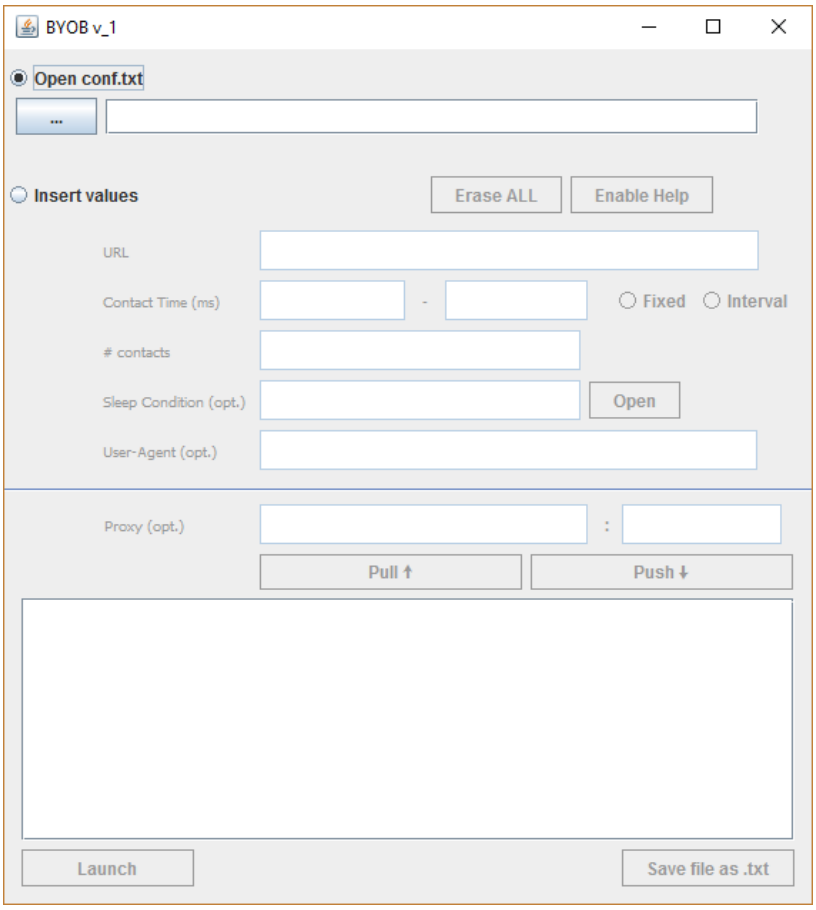

Figura 2.1: Apertura di un file di configurazione

La classe *GUI* è il cuore dell'applicazione e rappresenta l'interfaccia grafica con cui l'utente fornisce i parametri di configurazione.
A prima vista, l'interfaccia può risultare complicata, ma si è cercato di rendere la modalitá di fruizione più intuitiva possibile, attraverso piccole azioni per il suo utilizzo e con una grafica molto semplice, dato che ci si è soffermati piú sulla sua logica di funzionamento. Le azioni iniziali che l'utente può intraprendere, tramite le *checkbox*, sono due:

1. Aprire un file di configurazione giá in possesso dell'utente;

2. Inserire manualmente i parametri di configurazione.

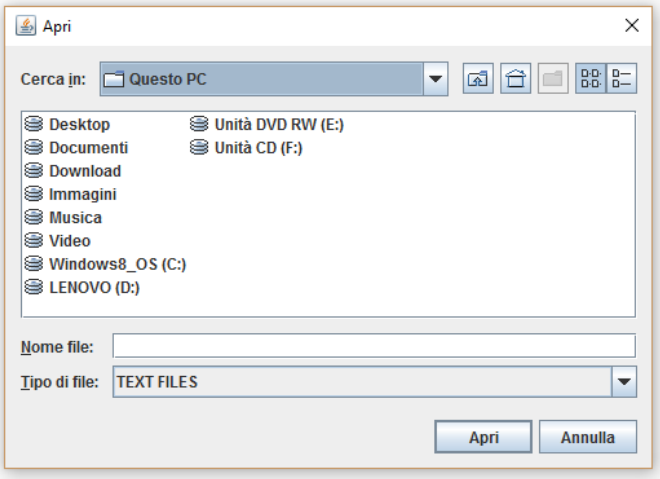

Figura 2.2: Apertura di un file di configurazione

L'apertura di un file di configurazione è riconducibile all'apertura di un qualsiasi file, gesto oramai assimilato nella vita di tutti i giorni. Cliccando sull'apposito pulsante "...", viene aperto il textitJFileChooser, un semplice meccanismo di scelta a disposizione dell'utente (una sorta di *file explorer* per muoversi tra file e cartelle del *file system*). Ad esso è stato applicato un filtro con cui sono visibili solo cartelle e file

di testo: questo perchè i parametri di configurazione vengono letti esclusivamente da file con estensione ".txt".

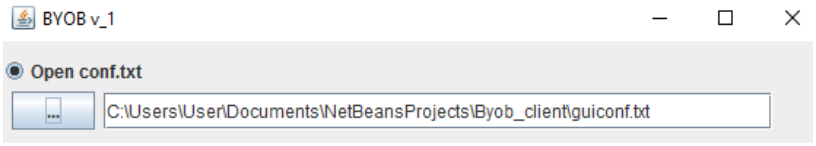

Figura 2.3: Scelta del file di configurazione

Una volta selezionato, il path del file scelto apparirá nella casella di testo relativa, sbloccando il pulsante "Launch" per il *run* dell'applicazione.

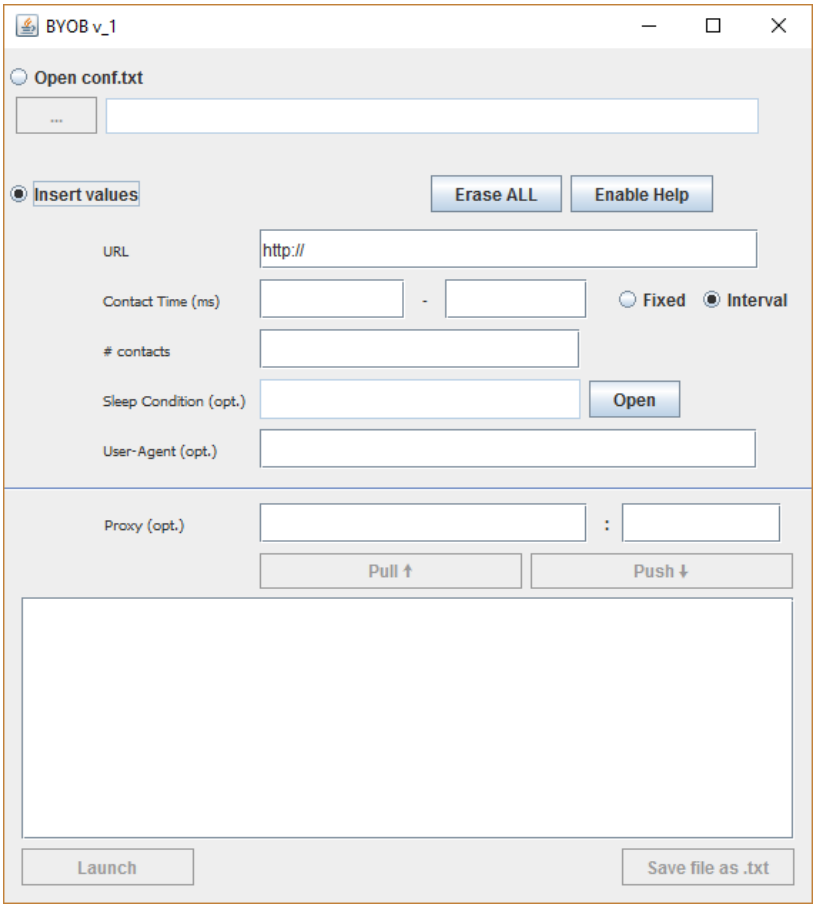

Figura 2.4: Inserimento manuale parametri di configurazione

L'altra opzione è quella dell'inserimento manuale dei parametri di configurazione. Tali parametri sono:

- **URL** da contattare;
- **Contact time**, ossia la periodicitá di contatto che puó essere:
  - Fissa ("*Fixed*");
  - Invervallo ("*Interval*"), cioè all'interno di un intervallo temporale;
- # **contacts**, ossia il numero massimo di contatti per ciascuna URL;
- **Sleep conditions** (opzionale), ossia l'insieme di condizioni temporali in cui non viene svolta alcuna azione; sono state divise in condizioni sui giorni e condizioni sulle ore della singola giornata, ognuna caratterizzata da una lettera dell'alfabeto o ad una stringa vuota nel caso di nessuna restrizione:

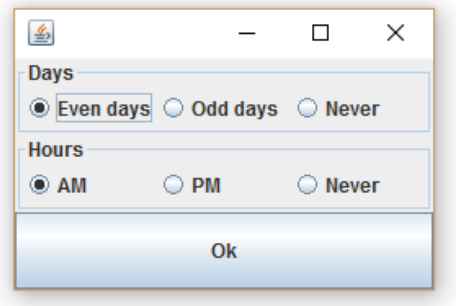

Figura 2.5: Sleep conditions

  - 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;
- **User-Agent** (opzionale), ossia la modifica da apportare allo User-Agent del protocollo HTTP;

- **Proxy** (opzionale), ossia l'indirizzo e porta di un proxy pubblico da utilizzare.

Una volta inseriti nelle relative caselle di testo, il pulsante "*Push*" permette di visualizzare tali parametri (come apparirebbero se fossero stati scritti sul file di configurazione) nell' area di testo sottostante: si noti come, inserendo altri parametri, essi vengano appesi a quelli giá presenti. Il pulsante "*Pull*" permette di estrarre dall'area di testo l'ultimo inserimento effettuato per una eventuale modifica dei parametri non correttamente inseriti.

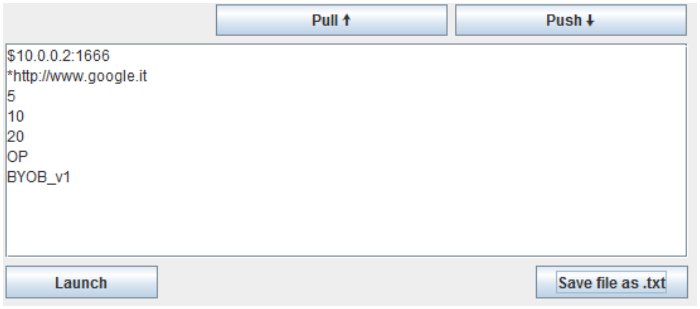

Figura 2.6: Push - Inserimento dell'input nell'area di testo

L'inserimento manuale ha il fine di salvare tutti i parametri di configurazione all'interno di un file di testo (azione svolta dal pulsante "Save file as .txt"), con un approccio simile a quello utilizzato per l'apertura del file di configurazione, cioè attraverso il textitJFileChooser.

GUI user-friendly Per rendere il tutto semplice e a portata di qualsiasi utente, il *toogle* "*Enable Help*" permette di invocare dei *tooltips* sulle caselle di testo per meglio capire che dato inserire e in che formato. Nell'esempio fornito, per sapere come correttamente inserire il parametro *URL*, basta posizionarsi col mouse sulla casella di testo corrispondente e attendere qualche secondo.
Per quanto riguarda il pulsante "*Erase ALL*", la sua implementazione è nata dalla

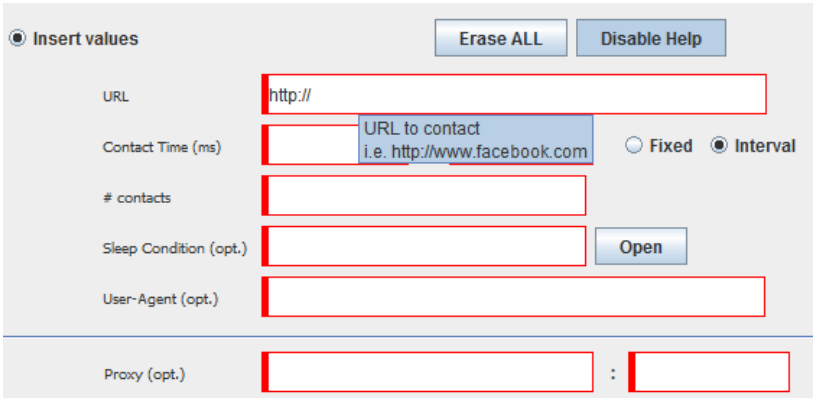

Figura 2.7: Utilizzo dell'help

volontá di fornire all'utente un metodo veloce per cancellare ogni inserimento nelle caselle di testo nel caso di molteplici errori.

Infine si è scelto di fornire un *warning message*, con l'approccio di un *pop-up*, in caso di errato e/o mancato inserimento di uno o piú parametri tramite il pulsante "*Push*": l'utente, a questo punto, puó decidere se tornare indietro e correggere da solo l'errore oppure utilizzare dei valori di *default* per i parametri di configurazione.

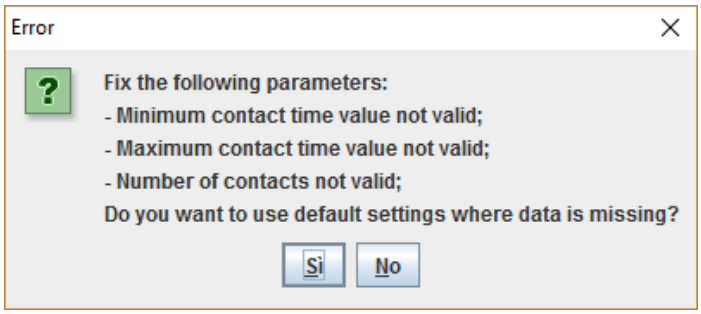

Figura 2.8: Esempio di messaggio d'errore per dati mancanti e/o errati

2.3.7 Parser

La classe *Parser* si occupa di tutti gli aspetti legati al controllo dell'*input*, sia dal punto di vista sintattico che dal punto di vista alfanumerico, ma anche la correttezza di

pattern, come nel caso venga inserito un indirizzo IP per il *proxy*. Altre responsabilitá della classe sono:

- 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 nel tipo corretto (per fornire un esempio, il numero di contatti da stringa a intero);
- controllo della validitá di ciascun parametro di configurazione (se il parametro è un numero, se gli estremi degli intervalli sono uno maggiore dell'altro, ...).

2.3.8 Tools

La classe *Tools* contiene al suo interno, tra varie funzioni e vari metodi, la generazione dell' identificativo del *bot*.
Tale generazione è effettuata tramite la ricerca del sistema operativo della macchina ospitante per poi lanciare, tramite riga di comando, delle istruzioni con cui ottenere dati riguardanti l'*hardware* della stessa. Queste informazioni sono poi "date in pasto" ad un funzione *hash* con cui è generato l'ID. La decisione di utilizzare entrambe le informazioni, sia *hardware* che *software* è stata dettata dalla volontá di ridurre le possibilitá di ID uguali, grazie all' indirizzo *MAC* univoco della macchina e al sistema operativo che può trovarsi su piú macchine differenti. La funzione *hash* usata è MD5, fornita dalla classe *MessageDigest*, perchè è veloce da generare e non si avevano grosse pretese di sicurezza (non si tratta di una password ma bensì di un identificativo che servirá al *botmaster* per capire quante unitá ha a disposizione per un eventuale

attacco).

Questa classe si occupa anche della schedulazione dei *task*, tramite l'unica istanza della classe *ByobSingleton*. Altri metodi e funzioni della classe hanno lo scopo trovare informazioni sulla macchina ospitante (sistema operativo, architettura, browser installati, lanciare istruzioni da riga di comando) ed effettuare i controlli per la generazione del *warning message* della classe GUI.

2.3.9 URLDetails

URLDetails è la classe che contiene tutte le informazioni inserite dall'utente per ciascun contatto che il *bot* deve effettuare. Oltre agli attributi, che coincidono con i parametri di configurazione forniti dall'utente, mette a disposizione i metodi getXX e setXX (dove XX è il parametro di configurazione) per recuperare e settare il loro valore.

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 linux 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*[1], in cui il package *com.sun.jna.platform.win32.Advapi32Util*

[1] 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), 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 condizione 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

Torniamo all'analisi della GUI.

Nel caso di scelta di inserimento manuale dei parametri di configurazione, l'interfaccia mette a disposizioni delle *jFormattedTextField*. Esse rappresentano un modo facile per specificare l'insieme di caratteri che possono essere accettati tramite un "for-

mattatore": nel caso di campi numerici come *Contact Time*, *# contacts* e la porta del *proxy* è stato utilizzato *NumberFormatter*, il quale accetta come input alla creazione una classe *abstract* chiamata *NumberFormat*, la quale fornisce un'interfaccia per formattare e parsare correttamente i numeri; nel caso del campo IP del *proxy*, è stato utilizzata una classe finale *RegexFormatter* che estende la classe *Default Formatter*. Tale estensione si è resa necessaria in quanto il campo IP richiedeva un determinato *pattern*, ossia quello xxx.xxx.xxx.xxx dove "xxx" Ã¨ compreso tra 0 e 255. Per tutte le altre non si è reso necessario nessun *format* preciso, in quanto stringhe.

Dapprima, viene costruita una espressione regolare che specifichi 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"[2];
- "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 è creato il pattern completo:

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

- "^ specifica l'inizio dell'*input*;

[2]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.

- "?:" 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;
- "+ _255 + "$"" fa il *matching*, con il pattern specificato, un'altra volta.

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

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

Solo un parametro di configurazione non puó essere inserito manualmente: *sleep condition*. Dato che per la rappresentazione di ciascuna condizione è utilizzata una lettera, è stato pensato di fornire all'utente un'esperienza semplificata tramite l'apertura di un *jFrame* (come se fosse un pop-up) in cui dividere le condizioni secondo la categoria di appartenenza (condizioni sulle ore e condizioni sui giorni) e permettendo la scelta di una sola di esse per ciascuna categoria. Una volta fatta e confermata la selezione, vengono restituiti le lettere rappresentanti le condizioni scelte nella relativa jFormattedTextField.

Una volta inseriti i dati, l'utente ha a disposizione due *jButton* chiamati "Push" e "Pull".

"Push" controlla e scrive i dati inseriti dall'utente all'interno della *jTextArea*, il cui scopo è quello di permettere una visualizzazione generale di tutte le informazioni inserite e, nel caso, correggerle. La *jTextArea* non è editabile: questo perchè si voleva

evitare che l'utente potesse cambiare il *format* dei parametri di configurazione e dovendo così rieffettuare un altro controllo sulla loro validitá.

Una volta cliccato sul suddetto pulsante, se l'utente ha, per esempio, erroneamente inserito un valore sbagliato o dimenticato un campo non opzionale, viene visualizzato un messaggio a video di *warning* con la lista degli errori e/o dimenticanze nelle *jFormattedtextField*:

```
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 1025 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 dell'errore e, inoltre, nel caso di mancato inserimento di parte o di tutti i dati, permette anche di utilizzare dei valori di *default* dove cè questa lacuna. Nel caso l'utente scelga di non voler utilizare i parametri di configurazione di default, la *popup* col messaggio di errore si chiude, ricordando all'utente di cambiare/inserire manualmente i valori errati e/o mancati.

La generazione del messaggio avviene controllando, di nuovo, che:

- *contactTime*, *#contacts*, *proxy port* siano numeri interi;
- *proxy IP* segua una formattazione adeguata e valida;
- nel caso di scelta di *contactTime* ad intervallo, che un valore sia minore dell'altro.

Tutti i controlli sono effettuati tramite funzioni create nella classe *Parser*.

Se l'utente si accorge di aver commesso un errore, "Pull" permette di prendere il *set* di parametri inserito e riportarlo, campo per campo, nelle rispettive jFormatted-TextField. A questo punto, è possibile effetuare le modifiche opportune e utilizzare di nuovo "Push" per re-importare i parametri nella *jTextArea*.

Una volta inseriti tutti i parametri di configurazione nella *jTextArea*, l'utente deve salvare il file di configurazione tramite il *jButton* "Save file as .txt", il quale permette di scegliere la cartella dove salvare il file, dargli un nome e confermarne o meno la correttezza. Una volta confermato, avviene l'estrazione dell'IP e della porta del *proxy* (se inseriti), successivamente l'estrazione dei dati dalla *jTextArea* che vengono inviati alla classe *Parser* insieme al numero massimo di attributi per ciascun contatto, per essere scritti su file specificato dall'utente.

```
public static void writeConfigurationFile(File fileToWrite, String[] params, i
if (!fileToWrite.exists())
fileToWrite.createNewFile();
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();
}
```

fileToWrite è l'identificativo del file che è stato creato dall'utente per salvare i parametri di configurazione secondo il formato accettato dall'applicativo, *params* coincide con la lista di stringhe estratte dalla *jTextArea* (il primo sará sempre il *proxy*, se

inserito dall'utente) e *columns* rappresenta il numero attributi di ciascun contatto (quest'ultimo è utile per sviluppi futuri nel caso vengano aggiunti nuovi parametri di configurazione).

Il metodo è molto semplice, in quanto controlla l'esistenza o meno del file (in caso negativo, lo crea) e utilizza un oggetto di *BufferedWriter* per scrivere riga per riga tutti i parametri inseriti dall'utente.

A questo punto, cliccando sul *jButton* "Launch", si avvia il programma tramite i parametri inseriti nel file con formato ".txt".

3.3.2 Lettura e caricamento

Nel caso venga scelta la lettura di un file di configurazione, tramite il *jButton* "...", avvia il *jFileChooser*, un componente grafico simile a quello del *file explorer*, con cui può selezionare un qualsiasi file di testo (e quindi con formato .txt) presente sulla propria macchina. Una volta selezionato, il percorso assoluto di tale file verrà visualizzato nella relativa *jFormattedTextField*. A questo punto, cliccando su *jButton* "Launch" si darà avvio alla lettura del file selezionato.

La lettura avviene tramite *BufferedReader*. Se la prima riga è il proxy, riconoscibile se il primo carattere è un carattere speciale ($), parso la riga in IP e porta e passo direttamente alla seconda riga.

```
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; }
```

Successivamente, a prescindere dalla presenza o meno del *proxy*, si cerca una stringa che inizi con un carattere speciale (*) che indica l'inizio del singolo insieme di parametri di configurazione. La stringa immediatamente successiva è l'url da contattare e quindi se manca, il sistema esce, segnalando l'errore.

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

Nel caso in cui sia presente, si scandiscono le righe successive del file (fino alla fine o alla successiva stringa che inizia per "*"), si crea una stringa del tipo "URL;minT;maxT;numC;sleepC;userAgent;", la si splitta per ";" e si convertono i parametri nel tipo di dato corretto. Una volta finito, tali valori sono utilizzati per creare un oggetto di *URLDetails*. Il processo è ripetuto fino alla fine del file, portando così alla creazione di una lista di questi oggetti.

```
// 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);
```

Checkbox, popup e lettura file. Popolamento arraylist di URLDetails

```
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(ArrayList<URLDetails> task)* che, per ogni elemento estratto dall'ArrayList, crea un'istanza della classe *ByobTask* (che implementa l'interfaccia *Runnable*) e ne richiede la schedulazione allo *Scheduler Executor Service*.

```
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, vengono 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 viene effettuato nuovamente il check delle condizioni; se nessuna condizione è verificata, allora viene decrementato il numero di contatti ancora da effettuare, viene 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;
        }
}
```