UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

Corso di Laurea in Sicurezza dei Sistemi e delle Reti Informatiche (Crema)

JDrone 2.0

The evolution of the Dorothy's Botnet Infiltration Module

RELATORE

Prof. Marco Cremonini

CORRELATORE

Dott. Marco Riccardi

TESI DI LAUREA DI

Domenico Chiarito

Matr. 751960

Anno Accademico 2010/2011

Acknowledgments

With no order of importance....

I would like to thank my family and expecially my sister Anna and my brother-in-law Dom for all the hours that he spent waiting for me at the airport.

I would also to thank my friends Emiliano, Salvatore and all the other classmates for sharing with them this unforgettable experience.

I would like to thank also Prof. Marco Cremonini, Marco Riccardi, Patrizia Martemucci and all the other Dorothy team's colleages for their supporting.

Special Thanks to Sabrina Papini for her infinite patience and excellent guidance through the maze of this course.

I would like to thank Prof. Teresa Loitile for her mathematical support.

Last but not lest I would like to thank Walter and Leonidas for reviewing this document.

There are not ideal conditions to write, study, work or think,
but it is the desire, passion and stubbornness
to push a man to carry out his plan.
"Konrad Lorenz (1903-1989), Nobel Prize in Physiology or Medicine

Contents

List of Figures

List of Tables

Introduction

JDrone 2.0 is the evolution of a software started in 2007 with the thesis work of Marco Riccardi [Ric08] [CR09]. He developed an automated platform for botnet detection and analyses.

Bot (from roBot or zombie) is the term that identifies a computer compromised by a virus or a malware, while botnet is the term that identifies a network of compromised computers connected to the Internet, commanded and controlled by someone, called botnet-master, operating one or more Command & Control Server (C&C).
Typically, a botnet is controlled via standard network protocols such as IRC or HTTP; the current Dorothy version mainly analyses IRC-botnets.
Dorothy is a modular system; each module is independent and it could be modified to accommodate different needs, or could be executed as a stand-alone program.
This thesis work refers to the improvement of the infiltration module: the Dorothy Drone.
The drone is the entity that is responsible to infiltrate a botnet, replicating all commands used by real bots (zombies). In this way the C&C Server can consider the drone as a real bot. The goal of this infiltration activity

is to discover and collect new commands, new nicknames and other information to analyse C&C Server's activity.
Java Drone (JDrone) is the first evolution of Dorothy Drone written by Patrizia Martemucci. The main reason for this evolution is that the original version was made with a non portable language, this means that was not supported by all operating systems.
This thesis work is mainly focused on the Database Management System introduction.
The older data source, based on text file, was replaced with a DBMS that allows to improve existing features and implementing new ones.

JDrone 2.0 adopts a new multi-tier architecture which provides various levels.
From bottom to top was placed the Database layer, the Data Access Layer (DAL), the business logic layer including ServerAS and the ServerLOG, finally the client on the top layer.
This architecture allows sharing the data, installing multiple servers on different machines, and improving scalability and fault tolerance.

The Data Access Layer is the only component that can access to the Database, the other components can access to data through it.
Its main function is to abstract the data source and to provide interfaces to interact with the Database, for example querying a Table, insert new data and/or update existing data.
DAL and the Database were completely designed and developed from the

scratch in this thesis work.

Among the issues considered, the communication between DAL and Database was protected by encryption all exchanged data. These two actors, where Database acts as server, establish a secure connection using the SSL protocol; this means that every message is encrypted using the public server key. SSL protocol is based on the exchange of two digital certificates. More specifically the first certificate is used to prove server's identity and the second to prove client's identify.

Another considered issue is the JDrone client authentication; the user identity was already been proved using the certificate; now it is verified the permission to access the application, as well.
In other words, in addition to prove his identity, he must also be authorized to access the system.

Acquiring C&C Profile data is the most sensitive issue handled in this thesis work. For reasons related to the IRC protocol, used by the botnets considered/studied, different JDrone instances cannot use simultaneously the same C&C Profile data. This means handling the concurrency.

Finally, as new feature, the JDrone keeps track of all the sessions created by its clients by storing information into the Database.

Problems that emerged, such as the exclusive use of resources, have been solved by using mechanisms provided by the database.

A short overview of the Chapters is given in the following list:

- The 1st chapter introduces the Dorothy framework: its architecture, the limitations of the JDrone current version and the goals to be achieved in this thesis work.
- The 2nd chapter illustrates how the solutions were achieved in the new version.
- The 3nd chapter describes in detail the JDrone's improvements, his new architecture and features.
- The 4th chapter describes the results obtained and future work.

Chapter 1

The Dorothy Project

The Dorothy Project is aimed at realizing a fully automated framework for botnet analysis and monitoring; it generates graphic and textual results. It is an open-source software developed to increase and share information knowledge about botnet diffusion and feature. Dorothy is composed by different modules, each one with a specific task to accomplish and completely independent from others; Java Dorothy Drone is one of main modules of Dorothy Project.

Malware, short for malicious software, consists of programming (code, scripts, active content, and other software) designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and other abusive behavior.

Malware includes computer viruses, worms, trojan horses, spyware, dishonest adware, scareware, crimeware, most rootkits, and other malicious software.

Dorothy analyses malware activities; this analysis can be done automatically and it is used to find zombies, i.e. zombies meaning malwares that try to connect to a remote system after compromising the host.

Botnet is a network of compromised host (i.e. zombies), which is infected by a particular kind of autonomous spreading malware, connected to Internet and controlled by a single entity through a one-to-many communication channel.
Typically, an IRC botnet is a commanded and controlled remotely by master computer that is an IRC server. This following example illustrates how a generic botnet is created and used:

- A botnet operator sends out viruses or worms, infecting ordinary users' computers, whose payload is a malicious application, the bot.
- The bot on the infected PC logs into a particular server (often an IRC server, but, in some cases an HTTP server).
- The operator instructs the compromised machines via the IRC server.
- The compromised machines, that were instructed via the IRC server, start their activity.

Dorothy's main goal is to find Who the malware is trying to connect by identifying the main entity called Command&Center (C&C) and if possible the zombies. The C&C is uniquely defined by an IP address, a port number and a hostname.

Dorothy is able to recognize messages that are exchanged by the C&C and the zombies according to IRC protocol.

The first original and authoritative RFC for IRC has been published in 1993, under the name "RFC1459". The Internet Relay Chat protocol (IRC) is one way of communication used by Botnets. This protocol provides messages exchanged between more entities, client and server. Typically an IRC client can connect to the server, choose one of the available channel and start sending messages/commands.
Recalling the RFC specifications, IRC commands are all declared with upper-case characters and the key-words used are: '' USERHOST | USER | NICK | JOIN | PASS | PRVMSG | MODE ''.
This information is sufficient for composing a regex filter that will used for the commands recognition, an essential process for the drone infiltration.
Knowing the right commands accepted by the C&C couldn't be enough. The right chronological sequence, and the exact string passed are other important information to know before we attempt to prepare a botnet infiltration. Otherwise Dorothy must consider the full string containing the known keyword as it is, then replicate it in the infiltration step.

It is important to know that it is not possible to infiltrate in a botnet with an ordinary irc-client, because all the IRC Clients provide a silent automatization like auto join, auto LIST, auto WHO, auto response to the VERSION request, etc; these automatisms are not expected by C&C server, that is then able to recognize a fake from a real bot.

Hiding the drone identity is needed during the infiltration process, because the C&C server is able to recognize its genuine zombies from a drone. Thus, it can ban a non-zombie client, or in the worst case use a DDos attack against the drone's IP address.
For this reason the real DDrone identity must be protected using, for example, a proxy for obfuscating IP address during the infiltration process. A main proxy usable for this scope is the TOR Network, which is an onion network built for protecting the host's identity within network communications.

1.1 Dorothy Architecture

Dorothy has been developed according to rigid specifications, among them: all process are fully automated; the entire platform has been developed according to modular principles; each module is independent and run-able individually as stand-alone tool; platform is scalable and open source.
Modular design allows to subdivide the platform into smaller parts independent from each-other, to avoid entire platform crash if one module fails.
Dorothy's modules are described below:

- Malware Collection Module (MCM) - the malware collection is the first step. To collect malwares requires a honeypot connected to the public network. The honeypot that must be easily exploitable. After a successful exploitation it has to be able to save the binary artifact sent by the attacker to a secure repository.

- Virtual Honeypot (VME) - after the malware collection process, Dorothy has to analyze all the binaries fetching that are needed for starting an investigation process against a botnet: a bot-malware. Analyzing malwares means discovering what the malware effectively does after its execution. The most reliable way to accomplish to this goal is to provide its execution on a demilitarized system and to observe its activity. VME is a virtual environment used for executing malware without risking to compromise the host.

- The Network Analysis Module (NAM) - constantly monitoring the honeypot network activity during the malware execution, allows to knowing information about who contacts infected host, protocol used, data exchanged.

- The Data Extraction Module (DEM) - it takes as input the raw file processed by the NAM and extracts from it critical information for all subsequent modules like the infiltration process, which requires all commands exchanged between zombies and C&C.

- The Geo-location Module (GLM) - knowing the C&C IP address allow Dorothy to be able to understand where is placed such C&C in the earth. Information regarding the geographical position of the ISP that detains the network mask where the C&C IP belongs, allows the GLM to define exactly the geographical coordination of the ISP. Thus, it is impossible to define exactly the C&C IP address' coordinates, but it is possible to refer to the relative ISP location for analysis purposes as required by the Dorothy Project.

- The Infiltration Module (IM) - after the DEM recognized the exact C&C language, the (IM) will be able to replay all the registration process against the C&C IRC server using its Dorothy-drone (DDrone).

- The Live Data Extraction Module (LDEM) - all the data acquired by the drone (Infiltration Module) during its snooping phase, are redirected to the Live Data Extraction Module (LDEM) for extracting all the relevant information about the live botnet activity.

- The Data Visualization Module (DVM) - the module should firstly filter the data gained from the DEM, then normalize them and finally generate the selected charts. For each type of information extracted from the previous module, it is necessary to choose the most suitable chart for visualizing the information acquired.

- The Web Graphical User Interface Module (WGUI) - this module allows the final user to show all the acquired results using graphical charts; a web interface represent C&C found on a geographical map.

Figure 1.1 shows module's dependencies.

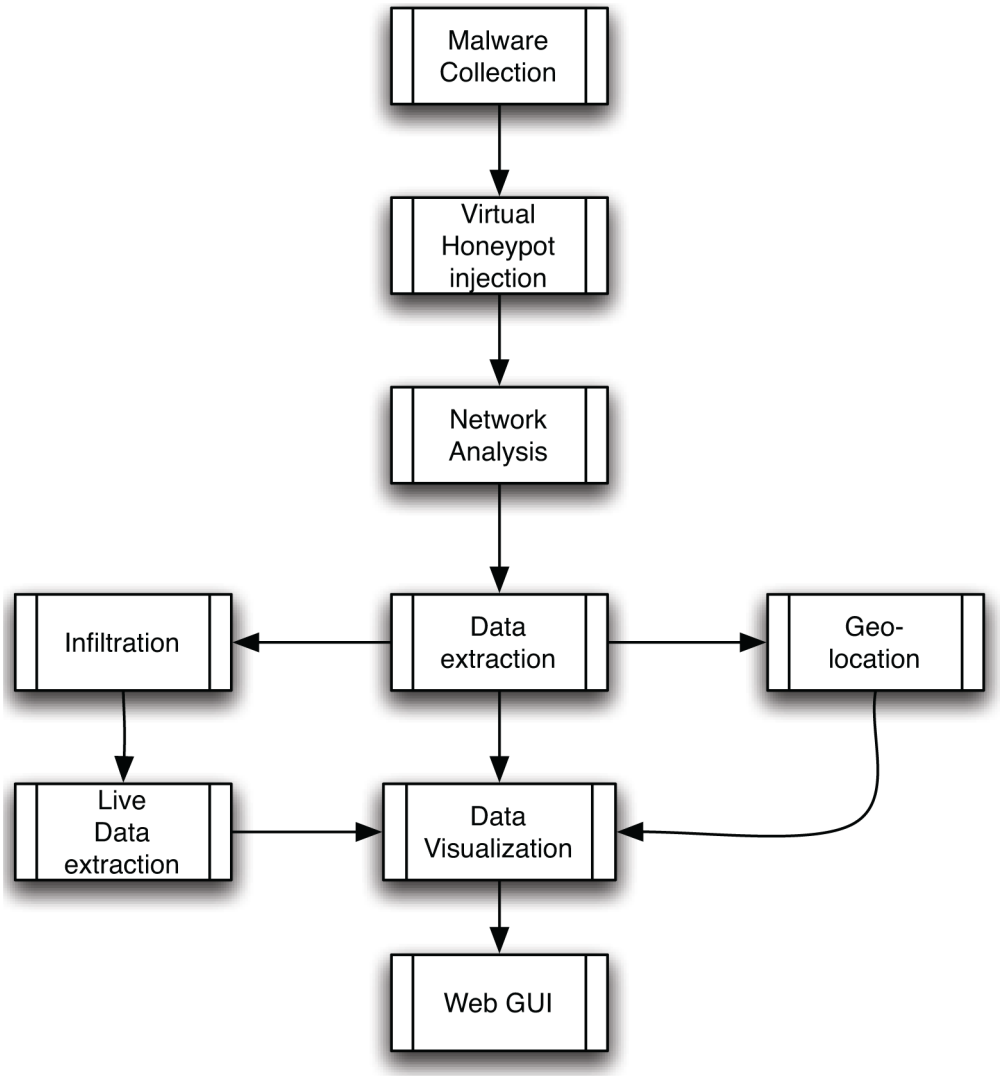


Figure 1.1: Dorothy Architecture

1.2 The Java Dorothy Drone

As just mentioned above, Dorothy's Infiltration Module is called DDrone. A DDrone needs, first of all, a C&C's IP Address and Port number in order to establish connection with it through an Anonymizing Proxy, as

TOR network. DDrone works on twofold system level: Network and Application level.

The Network level manages all the drone network connections. After the network's establishment DDrone can receive all the data sent by C&C and then send everything that is required to the C&C. The application level grants data communication on two different sockets for input and output (send and receive data).

DDrone parses all the data received to understand when it is possible to respond automatically by recognizing the related IRC commands that require a drone response.

After parsing, data is stored in the input-data socket; writing into the output-data socket means to send data to the C&C. DDrone logs all the data and error exchanged, if any, in textual files.

The first DDrone version was developed in Unix bash, but it was not supported by other computer operating systems. To cope with this limit, Patrizia Martemucci helped the Italian Honeynet Project to develop a cross-platform version using Java language; Java besides being a portable language, it is an open source platform according to Dorothy's specification.

Figure 1.2 illustrates every step of the JDrone workflow.

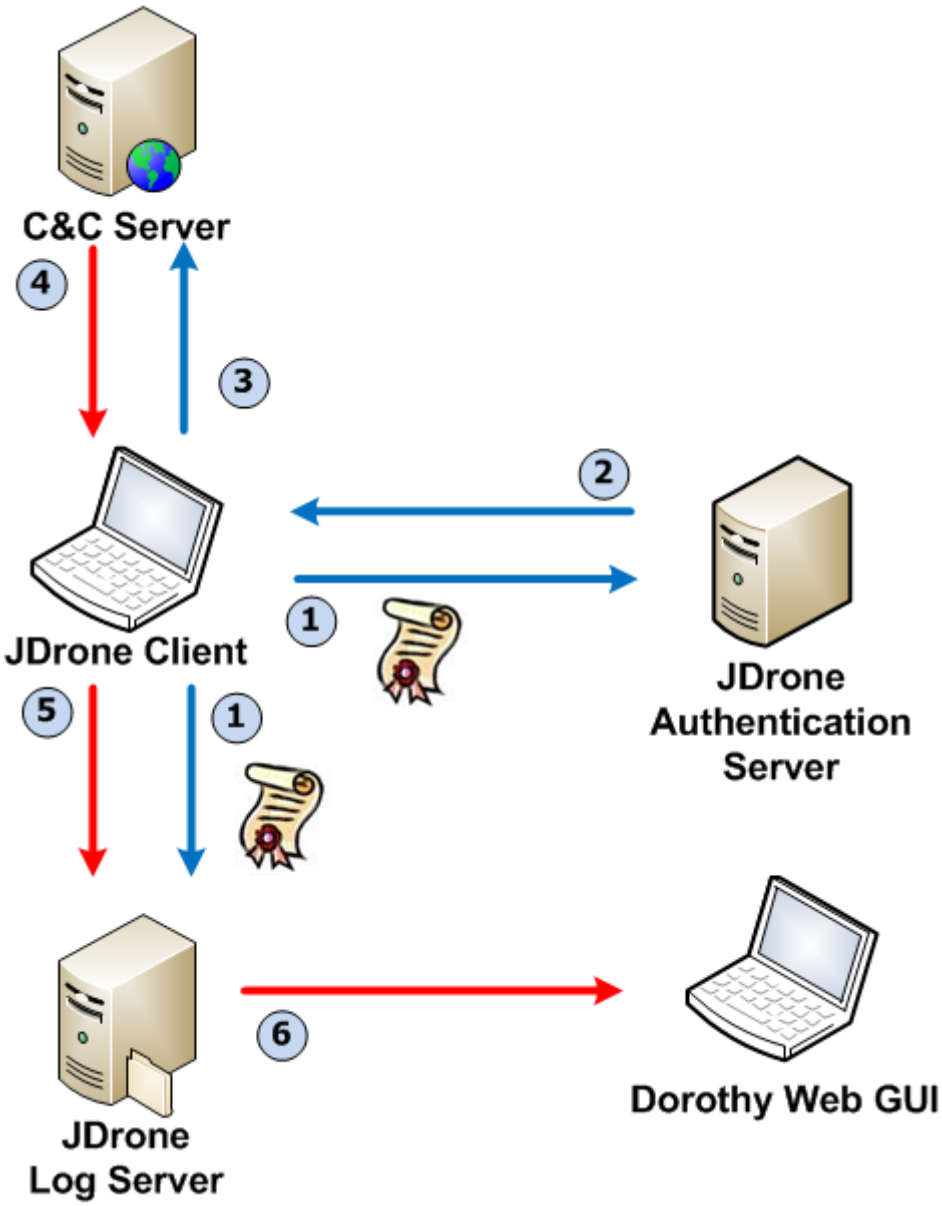


Figure 1.2: JDrone WorkFlow

1. The JDrone tries to authenticate simultaneously to the ServerAS and ServerLog components.

2. The ServerAS sends a CCProfile data to the JDrone only if the

authentication process went successful.

3. The JDrone uses the CCProfile to authenticate to the C&C server, then starts its infiltration activity.

4. C&C server sends messages and commands to the JDrone.

5. JDrone replies the received messages to the ServerLOG.

6. The Dorothy WebGUI module interacts with the JDrone ServerLog to extract and represent the collected data through the meaning of interactive charts.

1.2.1 Current version

JDrone is a complex infrastructure software developed according to the client-server architecture. Figure 1.3 shows its three components on two tier.

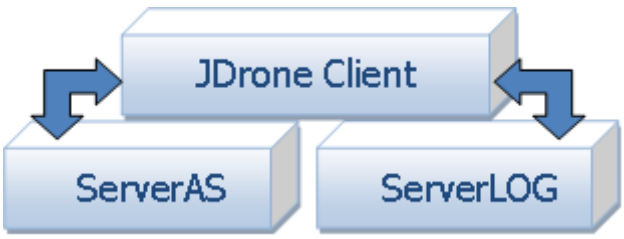


Figure 1.3: JDrone 1.0 Architecture

The component on the upper tier is the core. The JDrone Client is the component, which takes care of exchanging data with C&C. In this case it represents the IRC Server.

The first component on the bottom tier is the Authentication Server (ServerAS), it provides the two mainly functionalities: authenticate the clients and provide C&C Profiles data to the clients. All communications between JDroneClient and the others components happen through a secure socket connection (SSL). In fact JDroneClient sends a user certificate to the ServerAS, which contains CommonName, when at the same time it requires the certificate to verify his identity.

The user credentials are stored in the Authentication File, ServerAS, which checks the existence of CommonName received within this file. If the verification is successful, then it generates a token used for the connection to the ServerLOG, which is the next component. A C&C Profile contains all the information useful for DDrone's reply activity, which contains IP address, port number and IRC commands. This information is stored in textual file.

The second component on the bottom tier is the ServerLOG. After the connection is successfully established, it receives log messages from all connected clients, and takes care of storing the log messages into textual files. Moreover, ServerLOG sends log messages to his owner client.

This structure allows distribution of JDrone useful for several simultaneous running. This architecture allows running multiple clients instances.

1.2.2 Open issue and goals

Storing data on simple file does not allow applying a strong security measure in order to protect and prevent data tampering. For example, the current authentication file is a plaintext file without any protection. The ServerAS authenticates a client to find its CommonName within this file. This authentication process is considered weak because it has a unique protection provided by the operating system. In addition the ServerAS does not ensure any tampering prevention on authentication file.
Furthermore, the ServerAS takes care of the provision of the C&C Profile to the JDrone Clients. Also in this case, it cannot guarantee authenticity of textual files which contain C&CProfile's data.

For the infiltration process, it is very important to send messages with the same sequence that IRC server expects, and it is also very important that only one client at time uses a C&CProfile. In other words C&CProfile cannot be used by more than one JDrone Client simultaneously. This limit depends by IRC Protocol, in which it is not possible to have more than one identical nickname on the same server. So, the Server should manage concurrency on C&CProfile's requests.

Finally, the ServerLog saves messages of all JDrone Clients into a log file, which could contain several rows. Moreover, messages are stored in the same order in which they have been received. In conclusion, it is not easy to run queries on this log file.
With a database as data source, it has been possible to implement new features and enhance existing features, e.g. the client authentication.

Enhancing this feature was the main goal of this thesis.

A new feature to implement is the user activity tracking; an activity which is defined as a collection of operations done by a user. So, to allow queries and post-processing analysis, it is necessary to organize and to log these operations.

Make JDrone a scalable application is another goal of this thesis. An application is defined as scalable if it can handle growing amounts of work in order to meet users needs or its ability to be enlarged to accommodate that growth.

Implementing a new architecture and using a database help to reach this goal. A very good practice is to build a component dedicated to exchange data between the data source and the business logic, that could be remotable or not.

Having a remotable component provides the possibility to distribute it on different computers, so to balance the amount of work.

This implemented component is the Data Access Layer, described in the section 2.1.1.

The aim of this thesis is to resolve these issues.

Chapter 2

Software Design

One common way to describe a software architecture is to use UML diagrams.

Researchers have separated the architecture into four views: logical, module, execution and code. Different views support different goals and uses. [CH99]

The conceptual view (logical view) describes the architecture in terms of domain elements. Class diagrams are used to show the static configuration.

The module view describes the decomposition of the software and its organization into layers. The decomposition dependencies, the use-dependencies among layers and the assignment of modules to layers are shown using of package diagrams while class diagrams are used to depict class dependencies.

The execution view is the run-time view of the system. It is the mapping of modules to run-time images, defining the communication among them, and assigning them to physical resources.

The code view captures how modules and interfaces in the module

view are mapped to source files, and how run-time images in the execution view are mapped to executable files. Component diagrams are used to show the dependencies between source, intermediate, and executable files.

The following sections focus on the logical and module view showing layers' separation and dependencies between packages.

2.1 Architecture

JDrone is a layered application. The layers are concerned with the logical division of components and functionality, and they do not take into account the physical location of components. In contrast tiers describe the physical distribution of the functionality and components on separate servers, computers, networks, or remote locations.
Model diagram in Figure 2.1 shows three different layers Presentation, Business and Data layer; Users and Data Source are also represented as models.

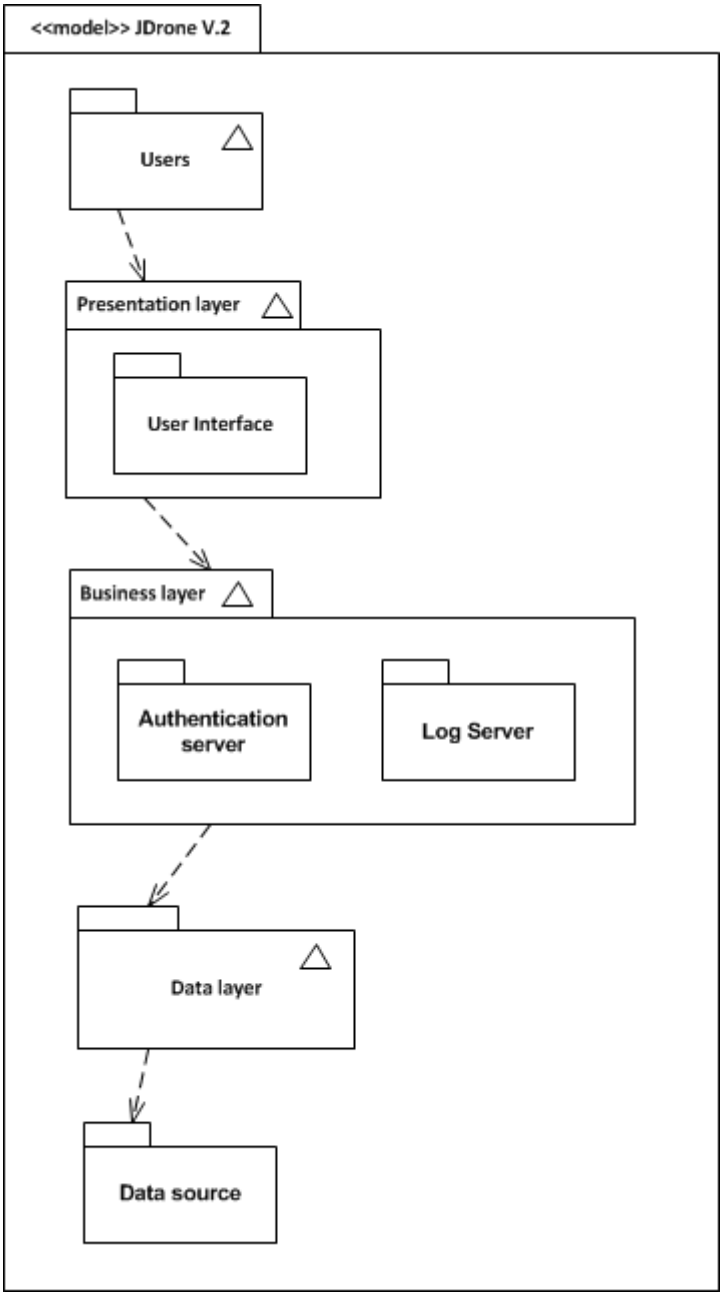


Figure 2.1: JDrone 2.0 Model Diagram

Figure 2.2 shows packages and dependencies between the packages.

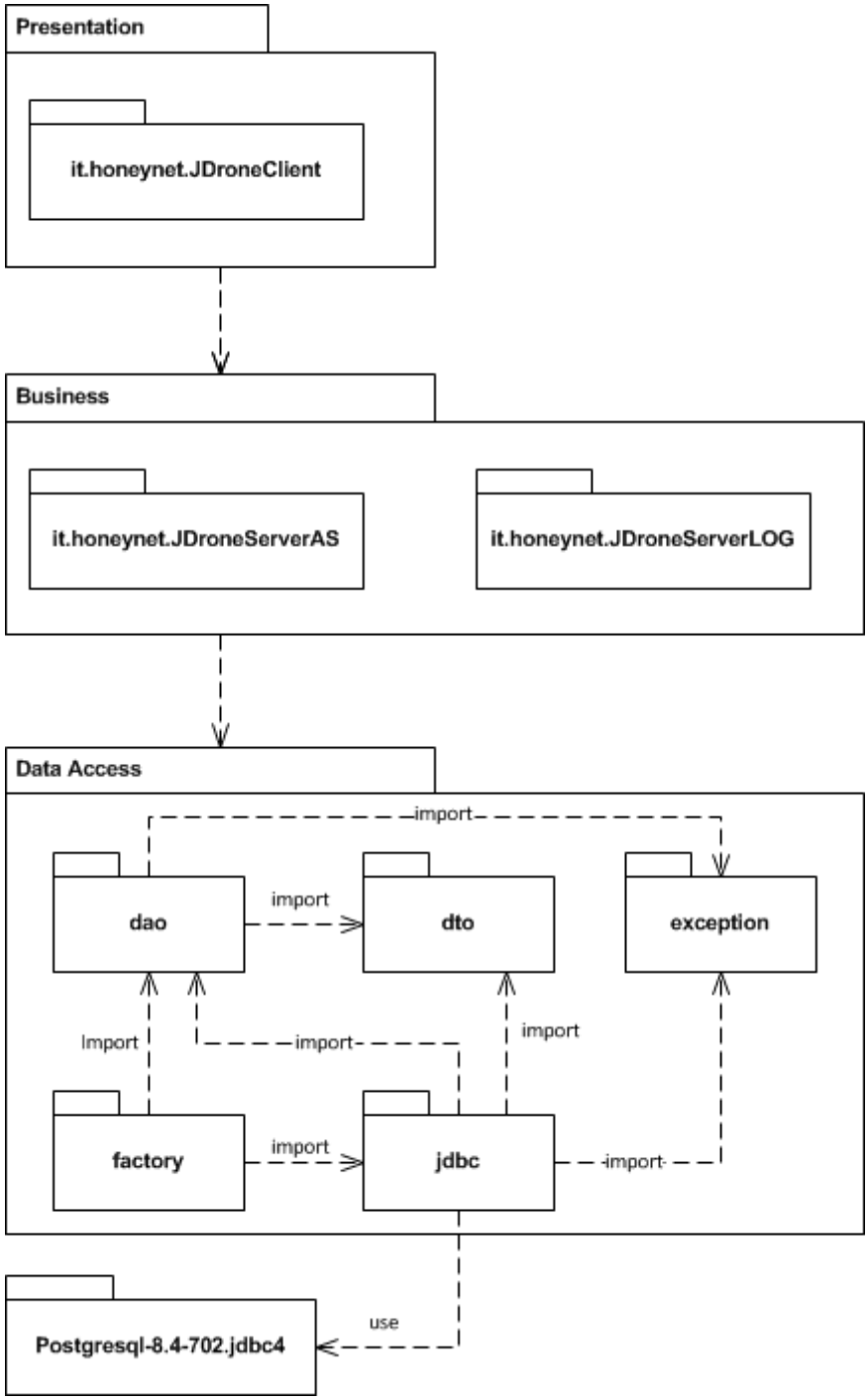


Figure 2.2: JDrone 2.0 Package Diagram

It is possible to think of package diagrams as a higher-level view into the system. This becomes important for understanding the system's architecture. Dependencies between packages are created in such a way to avoid circular dependencies among them.
JDroneClient package in the Presentation layer represents the user interface, while the packages JDroneServerAS and JDroneServerLOG represent the business logic and they are completely independent from each other.

2.1.1 DAL Pattern

Data access layer (DAL) is the layer of a software system, which provides access to data stored, such as a rational database (RDBMS). DAL presents various benefits. Most importantly, the Data Access Layer allows a clean separation of user interface code and data business logic, in other words it helps to abstract the database. Having a fully independent Data Access Layer of code that performs data operations allows maximum pluggability of the user interface code. This means that the same Data Access Layer can be used with different kind of user interface, e.g. form application, web application, etc. DAL provides a set of Java object that:

- represent database object and a set of java object
- have data access methods, such as create(), update(), and a collection of query methods (find*()) [CS06].

2.1.2 DAO Pattern

The DAL pattern adopts the Data Access Object (DAO) that is a common pattern which helps to isolate application's code from the code that accesses and manipulates database records; this means that its primary responsibility is to abstract data sources and to provide transparent access to the data source layer.

The DAO pattern adopts the Abstract Factory and the Factory Method patterns that produce a number of DAOs needed by the application.
Moreover, DAO uses Data transfer object (DTO) pattern to transfer data between software application subsystems.

2.2 Database

The first update that was made to resolve the issues described in the first chapter was to replace the old data sources with a Relational Database Management System (RDBMS).
By definition, a database is a collection of data and a set of rules that organize the data by specifying certain relationships among the data. Through these rules, the user describes a logical format for the data. [CPP06] The data items are stored in a file, but precise physical format of the file is of no concern to the developer.
Using a database presents various advantages over a simple file system:

- Shared access; many users can use one common and centralized set of data.

- Controlled access; only authorized users are allowed to view or to modify data values.
- Data integrity; data values are protected against accidental or malicious undesirable changes.
- Centralization data; there is no redundancy in the set of data included in the database.

Another advantage is the use of the stored procedures, that enhances the use of sql statements.

Usually, the client application sends each query to the database server, waits for it to be processed, receives and processes the results, does some computation, then sends further queries to the server.
All this causes interprocess communication and also network overhead if the client is on a different machine than the database server.
An advanced database system allows implementing stored procedures with which it is possible to group a series of queries and/or sql statements inside the database server, with considerable savings of client/server communication overhead.
Other advantages are:

- Extra round trips between client and server are eliminated.
- Intermediate results that the client does not need, don't have to be marshaled or transferred between server and client.

This implies considerable performance increases, compared to an application that does not use stored procedure.

2.2.1 ER-Diagram

Figure 3.1 shows the ER-Diagram containing the entities and the relations among them; the Chen's notation was used.

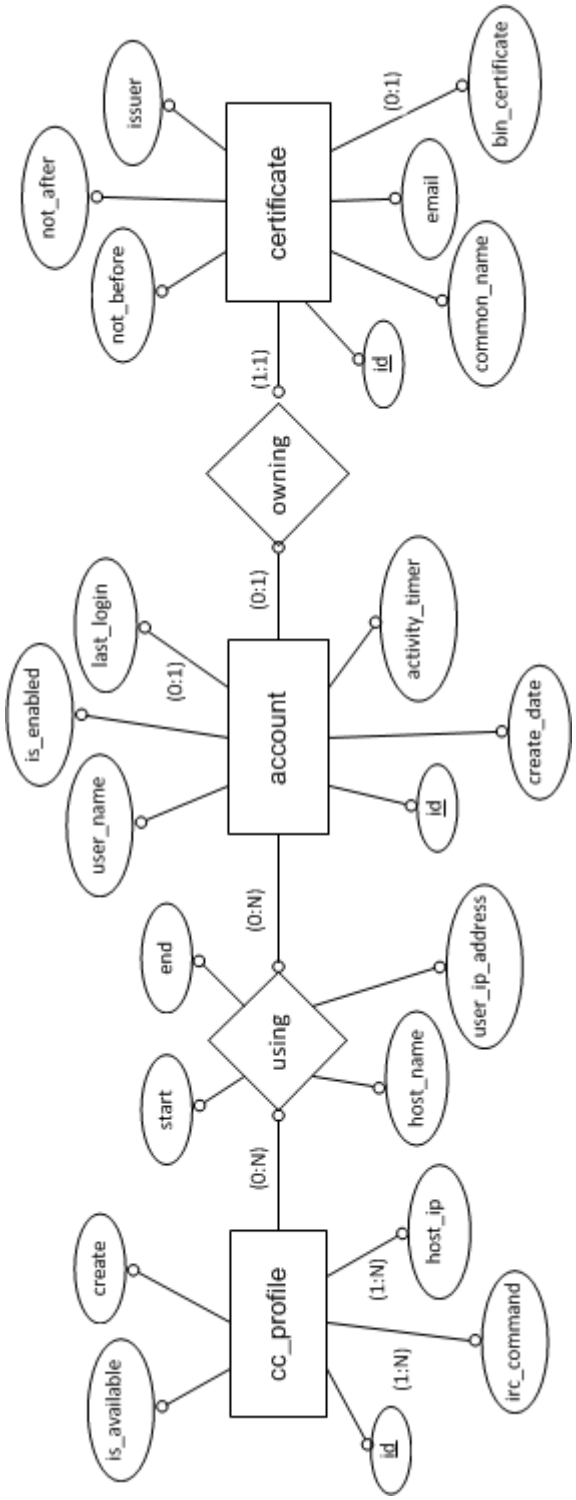


Figure 2.3: ER-Diagram

The meaning of the entities and their scope is described below:

- Account entity will be used for storing information about the application user (JDrone).
- Certificate entity will be used for storing information about user's certificate.
- Cc_profile entity will be used for storing information about the C&C's server and the related IRC commands.
- Session entity will be used for storing information about the JDrone activity infiltration.

For every attribute and relation the cardinality was specified. In general the cardinality is expressed by Minimum:Maximum Notation, where the Minimum value can assume 0 or 1, and the Maximum value can assume 0, 1, N; the last one indicates an undetermined number of values.
All attributes have the same cardinality (1:1) except irc_command and host_ip, in fact a cc_profile entity could have more than one irc_command or host_ip; for these information the cardinality is (1:N).
ER diagram contains the following relations:

- Owning indicates the relation between account and certificate entities; this means that an account owns a minimum of zero (0) certificates and a maximum of one (1) certificate; a certificate belongs to a minimum and maximum of one (1) account.
- Using indicates the relation between account and session entities; an account uses a minimum of zero (0) cc_profile and a maximum of

many (N) cc_profiles; a cc_profile can be used by a minimum of zero (0) account and a maximum of many (N) accounts.

2.3 Solutions

As communication between client and server is encrypted; also exchanging data between DAL and the database is protected by an SSL connection.
Encryption is one of the most important security principles, since it prevents network sniffers from intercepting sensitive data.
SSL protocol provides the encryption of all messages belonging to a communication; to do this, it requires two valid certificates, one client and one server side, used to prove the identity of the communication actors.
When connection was established JDrone can start his activities such as, authenticate his clients, acquire CCProfile data and finally store the data about the client infiltration activity.

Having a database, it is simpler to query it than to extract data from a text file. In fact a database provides a query system with which it is possible, for example, to extract data directly from an entity and choose specific information.
In this thesis various queries are defined in order to extract information such as user account for the authentication process, or CCProfile data for JDrone's activity.
The client authentication is password-less. This means that it is useless to store any password inside the database, because the first level of

authentication guarantees the user's identity by checking User's certificate. The database provides a further level of authentication: it is possible to verify if a user can access the system, just by reading a flag related to it.

The acquiring CCProfile data process was carried out almost entirely within the database. This is the most important process of the client application, because it allows collecting useful data for the infiltration activity.
A stored procedure has been implemented to carry out this process, taking care of the following two issues:

- A specific CCProfile data cannot be used simultaneously by multiple users.
- All CCProfile data must be used, not only the first available.

The first issue was solved by locking the entity, which means that when the database receives the first request, the subsequent requests have to wait for its completion
The second issue was solved by implementing the Round-Robin algorithm, that ensures the usage of all the CCProfile entities available.
The stored procedure returns an available CCProfile ID, if any, and then the Server component can use this ID for two purposes:

- Retrieve all CCProfile data sending a query to the database passing the ID as parameter.
- Store the ID within a new tuple belonging to the session entity.

A session entity was used to store the data about the client infiltration activity, such as his ID, the CCProfile used, the beginning and the end of the activity. The server component collects this information and stores it into the database using the DAL component.

This information can be further retrieved from the database simply by querying it from any client application.

Figure 2.4 shows the state machine UML related to this process.

The Diagram, as the related software, is based on the event-driven paradigm, which means that the machines continuously wait for the occurrence of some external or internal event.

In fact the two machines, Client and ServerAS, communicate by events; after recognizing an event, the related action is performed. In this specific context, for example, when the client receives the "connect_cc" event, the start action is executed and then the server state is changed.

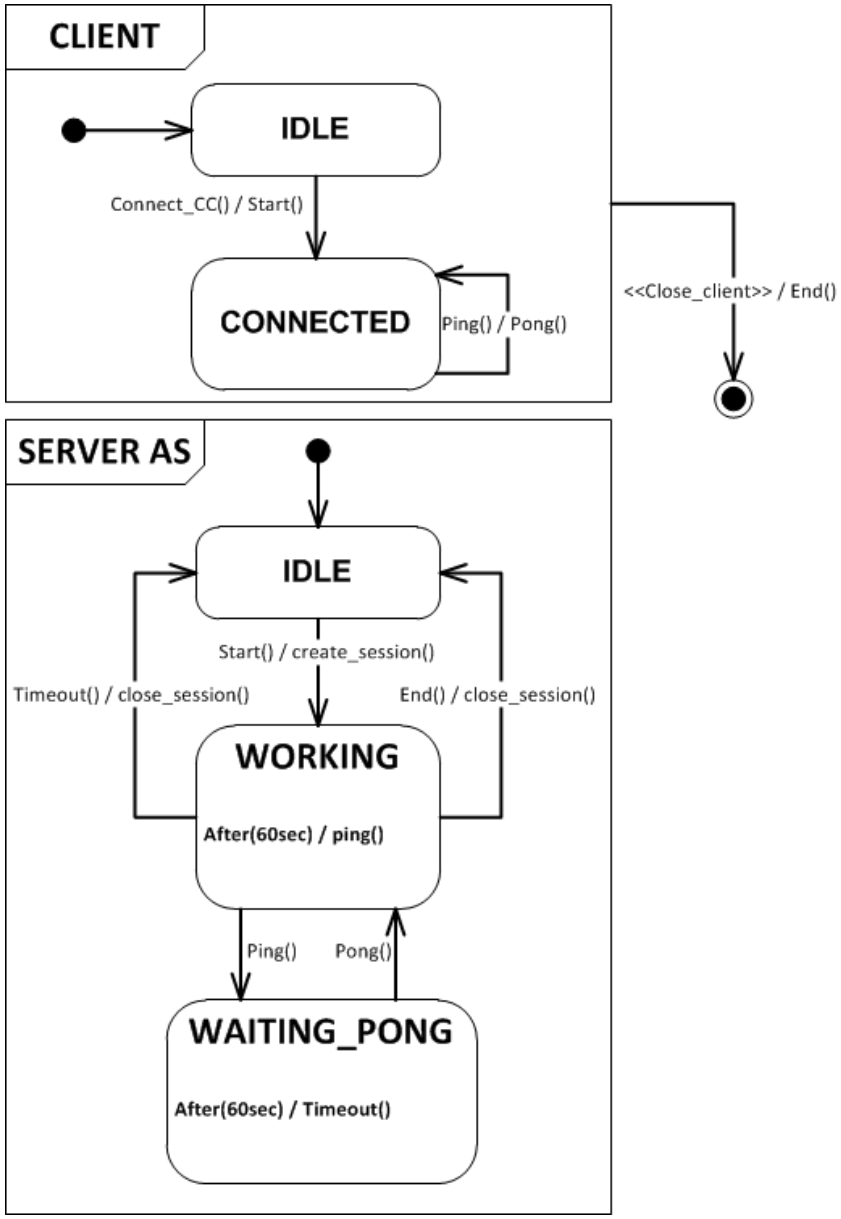


Figure 2.4: State Diagram of collecting session process

Further details about these solutions are described in the next chapter.

Chapter 3

Software Implementation

3.1 Database-side

This section explains database-side solutions for the JDrone previous version's issues.
This thesis uses PostgreSQL Server 8.4 as the data source. For more details please refert to section 3.1.1.

3.1.1 PostgreSQL

PostgreSQL is an object-relational database management system (ORDBMS). Its initial implementation started in 1986, and now it is widely considered as the most advanced ''Open Source database''' system available. PostgreSQL's predecessor was Ingres[1], which is a project developed at the University of California at Berkeley. [EG02]
''Open Source''' means that the source code of PostgreSQL is open and distributed with the package. The PostgreSQL is released under the

[1]post is the Latin word for after, PostgreSQL comes after Ingres

PostgreSQL License, which is a liberal ''Open Source license'', similar to the BSD or MIT licenses. [Pos]

Since it has been written in portable C language, PostgreSQL is a cross-platform system and it has been tested on several Operating Systems, such as Linux, Mac OS X, NetBSD, OpenBSD, Solaris, SunOS, Windows. [EG02]

PostgreSQL is fully compliant with Dorothy's specifics, such as Open Source, cross-platform, etc.

In the following sections the ER-Diagram, the database configuration and security settings are described. However, the database installation is out of scope for the specific Thesis.

This section explains common terms [KD06], some of them are similar in other database products:

- Schema

  A Schema is a named collection of tables. It can also contain views, indexes, sequences, data types, and functions;

- Database

  A database is a named collection of schemas. When a client application connects to a PostgreSQL server, it must specify the name of the database that it wants to access. A client can interact with only one database per connection.

- Command

  A command is a string that is used by the server to do something.

Statement word is a synonym.

- Query

  A query is a type of command that retrieves data from the server.

- Table

  A table is a collection of rows. A table usually has a unique name within schema, and all rows have the same shape, in other words every row contains the same set of columns.

- Column

  A column is the smallest unit of storage and represents one piece of information about an object. Every column has a name and a data type. Columns are grouped into rows, and rows are grouped into tables.

- Row

  A row is a collection of column values. Every row in a table is composed of the same set of columns. A row represents a real-word object, an instance of a generic object. Terms record or tuple are equivalent to a row.

- View

  A view is an alternative way to present one or more tables. It is possible to think a view as a virtual table, used frequently for join more tables through a query command.

- Trigger

  A trigger is a function that executes in response to a specific event in a given table. The PostgreSQL server automatically invokes a trigger (if defined) when executes a command, such as INSERT, UPDATE or DELETE.

- Function

  A function is a named sequence of statement that you can use within an SQL expression. Writing function in a server-side language means extending the server. These server extension are also known as stored procedure.

- Sequence

  A sequence is an object that automatically generates sequence numbers, every sequence must have a unique name.

- Client

  A client is an application that makes requests of the PostgreSQL server. Before starting to dialog with a server, a client must be connected to a postmaster and establish its identity.

- Postmaster

  The PostgreSQL is a client/server database, something has to listen for connection requests coming from a client application. When a connection request arrives, the postmaster creates a new server process in the host operating system.

- Server

  The PostgreSQL server is a system service that consumes commands coming from client applications and sends the appropriate responses to them. The server has no user interface, and so it is possible "to dialog" with it only through a client application.

- Transaction

  A transaction is a collection of database operations that are considered as a unit. PostgreSQL guarantee the all the operations within a transaction complete or that none of them complete. This is most important feature, common in many other database products. It ensures that if something goes wrong in the transaction, changes made before the point of failure will not be reflected in the database. Transaction usually starts with a BEGIN command and ends with a COMMIT or ROLLBACK commands. Functions and trigger procedures are always executed within a transaction established by an outer query.

- Commit

  Commit is the command that marks the successful end of a transaction. This command tell to PostgreSQL that all changes made to the database should become permanent.

- Rollback

  Rollback is the command that marks the unsuccessful end of a transaction. To roll back a transaction means discard any changes

made to the database since the beginning of the transaction.

- Index

  An index is a data structure that a database uses to reduce the amount of time spent to perform certain operations. An index can also be used to avoid inserting duplicate values.

- Result Set

  When PostgreSQL performs a query to a database, it returns a result set. The result set contains all the rows that satisfy query command.

All the implemented functions in JDrone database use a specific language of PostgreSQL, PL/pgSQL, a Procedure Language.

Figure 3.1 shows the Database Physical model The database is subdivided in two schemas. The blue area depicts the drone schema tables, while the green depicts a part of the dorothive schema tables. From the later, only one table is used by the JDrone.

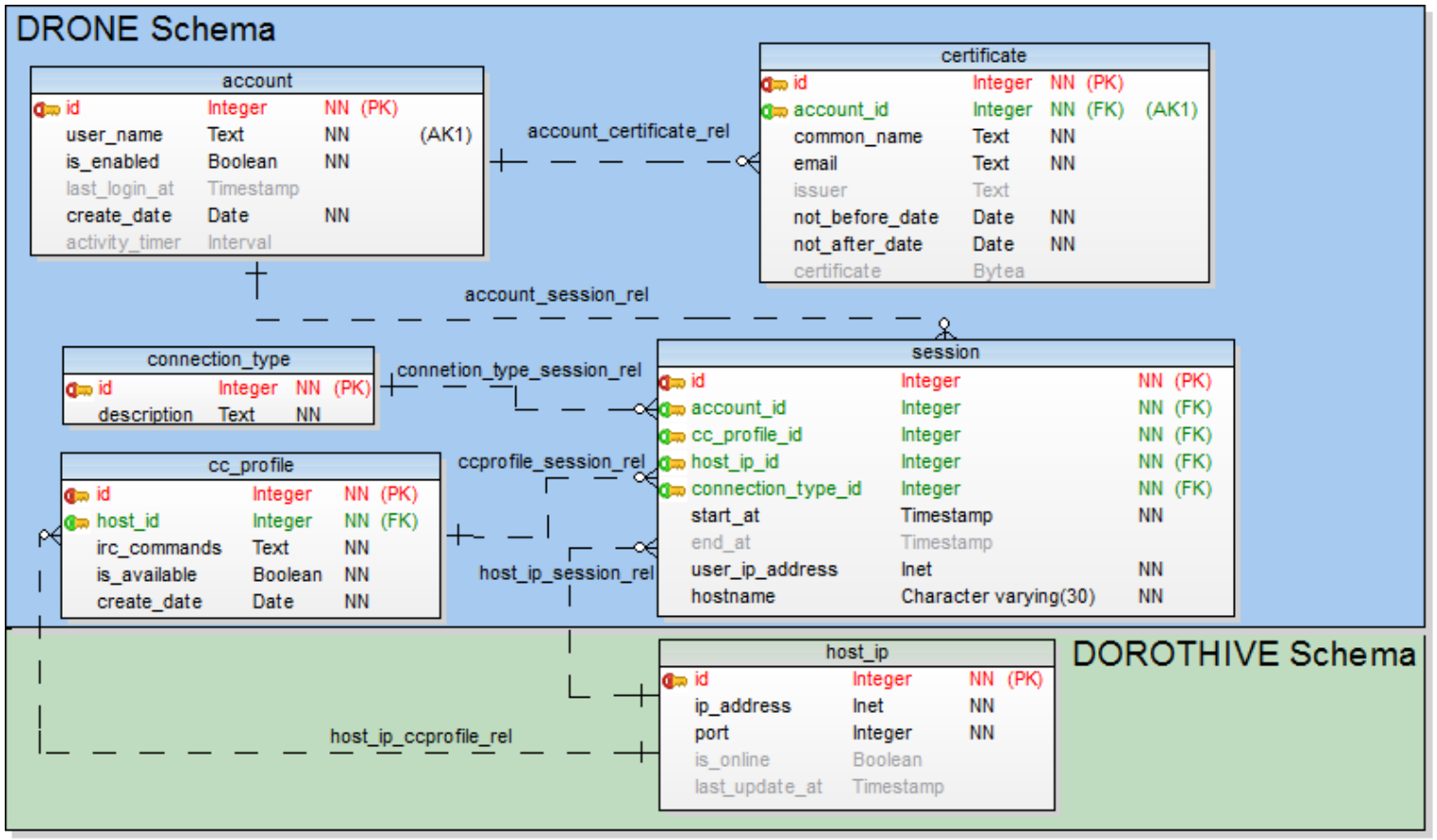


Figure 3.1: Database Physical Model

Additional information:

- Fields in red are primary keys (PK)
- Fields in green are foreign keys (FK)
- Fields in black are mandatory attributes
- Fields in gray are optional attributes
- AK means Alternate Keys
- NN means Not Null attribute

As shown in Figure 3.1, the Database Phisical Model involves the following Tables:

- Table "account": This table contains information about the user accounts used by the JDrone application, such as the user name, the boolean flag (*is_enabled*), which is useful to verify if the user is enabled to use JDrone application, and a field termed as *activity_timer* indicating the time duration of the user's activity. The data type for this field is ths so-called ''Interval'', which is a special type that expresses the interval of two times.

- Table ''certificate'': This table contains information about user's certificate. By design, it is possible to associate only one certificate per user. This table contains information subsets of X.509 standard, such as common name, user email, issuer identifier, validity time, and in addition a binary field to store certificate file.

- Table ''cc_profile'': This table is the most critical of the entire schema and contains important information for JDrone infiltration activity, e.g. IRC commands, boolean flag (*is_available*) that indicates when the

profile is available for a client, etc. *IRC_commands* is a particular field that stores values in CSV format representing the CCProfile related IRC commands.

- Table ''connection_type'': This table contains information about the client's connection mode (e.g. direct connection, proxy, tor).

- Table ''session'': This table plays the role of a log container, in which every record refers to a specific JDrone client activity. It is important to store information about the activities, e.g. CCProfile, start and end time of the activity, user IP address and hostname.

- Table ''host_ip'': This table belongs to the Dorothive schema, and contains the IP address and the port number of the IRC server, which are essential for allowing the JDrone client to connect directly to the IRC server (C&C). Moreover, the table contains a boolean flag, which is used to store the server's state, e.g. if it is reachable or not.

The relationship between the Tables are one-to-many except from the relationship between the Account and Certificate Tables, which is one-to-one. The later is defined by a unique index on the *Accound_id* field of the Certificate Table.

3.1.2 User Accounts

The JDrone client authentication process is translated as a request to the database server in order to verify user identity. All enabled users are stored into the account table and every user must have only one certificate related

for authentication. It is possible to disable a user account by setting the *is_enabled* field value to FALSE in the account table. The client request is directed on a virtual table created for filtering enabled users. This virtual table is *account_view*. In fact the view returns a result-set including only the enabled users with their own certificate data. The authentication is successful only if the user name requested is included in this result-set. A result-set sample is the following:

id int	username text	last_login_at timestamp	activity_timer interval
7	"domenico@xmail.com"	"2011-08-11 15:27:17.932"	"172 days"

Table 3.1: Example of account_view result-set

3.1.3 User Sessions

It is quite important to know the total time spent by a specific user for the infiltration activity. An Authenticated client user receives a set of information used for connecting to the IRCServer. This information is extracted by the view *ccprofile_view*. This view joins the tables *cc_profile* and *host_ip* and eventually, the merged information is sent to the client. The manner in which this information is sent to the client is described in the following section.

Information such as the Account Id and CCProfile Id is used to create a new record into the *session* table. This table contains also two timestamp fields, one for starting time and the other for the end time.

At the beginning of JDrone infiltration activity, a record is created in the

session table and the field *start_at* is set to the value of the current timestamp. On the other side, the field *end_at* is set only when JDrone infiltration activity comes to an end. Briefly, this activity's tracking process produces one row per activity into the *session* table.
When the field *end_at* is set an event, which is called ''update'' occurs on the *session* table. This event triggers the associated handler function termed as *update_account_activity_timer()*. This function computes the difference between *end_at* and *start_at* fields only if the first one is not null.
Finally, the function executes an update command on the *account* table in order to add the computed activity's time duration to the *activity_timer* field.

3.1.4 Concurrency

As mentioned in the previous section, information about CCProfile is stored into two different and linked tables (*cc_profile* and *host_ip* tables) using a one-to-many relationship.
According to the specifications, it is not possible to send the same CCProfile to more than one JDrone client application. Furthermore, it is a good practice to guarantee the use of all CCProfiles.

- The first issue has been solved by locking *cc_profile* table. Lock command ensures no concurrent access to that table. There are several ways to set a table in "locking mode". In this case the best choice is the so-called "ACCESS EXCLUSIVE MODE".
- The second issue has been solved by adopting the Round-Robin

algorithm. This one allows cyclic use of all CCProfiles.

A function called *get_id_profile()* has been implemented for carrying out these two solutions.
Client application sends the request by calling directly the function through a query command (SELECT). The function executes a few steps before returning a valid CCProfile's ID, if there is one available.
Steps executed by *get_id_profile()* function are the following:

- Lock cc_profile table in access exclusive mode.
- Retrieve last ID value of the previous request stored in a sequence.
- Query the view *ccprofile_view* by filtering the ID obtained in the previous step.
- Set the CCProfile found as unavailable and update sequence to that ID value, if the previous query returns a valid ID.
- Return CCProfile's ID.

An example of *ccprofile_view* result-set is the following:

id integer	is_available boolean	ip_address INET	port integer	irc_commands text[]	is_online boolean
1	t	"182.72.4.108"	3921	"{"NICK...", "USER ..."}"	t

Table 3.2: Example of ccprofile_view result-set

For completeness, there is another database feature that must be described and that is the ''sequence''. ''Sequences'' are used mainly for the auto-increment field value. In fact a sequence has important properties, e.g. minimum and maximum values, current value, increment step, cycled flag,

etc.. If the last property is true, the server automatically sets current value to the minimal value when it reaches the maximum value.
The sequence *finder_profile_seq* is used within the function *get_id_profile()* to store the last CCProfile ID used. This means that the maximum value of the sequence must be equal to the higher value of the CCProfile ID.
To avoid a manual update of this value the so-called trigger function has been implemented, e.g. *update_max_finder_seq()*, to update the maximum value only when a new record is inserted into CCProfile table. This function executes ALTER SEQUENCE command to update MAXVALUE.

3.2 Database Security Settings

PostgreSQL server provides security policies on three aspects; authentication; data encryption; access.
Authentication ensures that a user is in fact who claims to be. After having the identity proven, the server must ensure that the User can access the required data. Here is described how to grant or revoke user privileges and how to prevent tampering by intruders.

3.2.1 Database Authentication

Generally, the authentication process is commonly done through the use of logon passwords. This implies that authorized users collection is stored on database server. The Client application must send a valid couple of username and password to the database server, which identifies the

credentials on its *pg_shadow* system table.
PostgreSQL provides various ways to authenticate a client, which are configurable in the *pg_hba.conf* configuration file (HBA stands for host-based authentication) stored in the database-cluster's data directory. This file contains a set of records and each record specifies:

1. a connection type
2. a database name
3. a user name
4. a client IP address range (if relevant for the connection type)
5. the authentication method to be used for connections matching these parameters.

It is worth mentioning that in order to use an SSL connection ''hostssl'' must be specified as the connection type.

Starting from 8.4 version, PostgreSQL provides a new authentication method, called "Certificate Authentication", which is based on the exchange of digital certificates.
Authentication record with this kind of connection type could be the "hostssl DBDrone mrdrone 127.0.0.1/32 cert" whit the last value referring to the authentication method.

This authentication process, which is considered stronger than the Password Authentication, has been adopted in this thesis to grant access for JDrone Application.

In order to set up a server-side authentication protocol the following files must be present in the server's data directory:

- server.key
- server.crt (which must be signed by a CA)
- root.crt (verifies client authentication)
- root.crl (optional file which contains certificate revocation list).

Furthermore, for enabling the ''ssl'' connection, it is fundamental to change the server configuration file (postgresql.conf) and ensure that ''ssl'' property is set to ''ON'' (ssl=on).
Concerning the client-side authentication the following files must be in the client's home directory (depending on the operating system, e.g. for linux/unix the path is ~/.postgresql):

- postgresql.key
- postgresql.crt
- root.crt (verify server authentication)
- root.crl (certificate revocation list, optional).

As with the server's root.crt, the client's file, root.crt contains a list of server certificates that have been signed by a reputable third-party CA. The last file, root.crl, is optional and contains the revoked server certificates.
When a certificate signed by a certificate authority (CA) is not available, a self-signed certificate can be used. To generate a self-signed certificate using the OpenSSL tool, please follow the steps below:

- Run command "openssl req -new -text -out server.req". This requires information about certificate, such as the Common Name that must be the same with the user enabled to authenticate on the DB.
- Run command "openssl rsa -in privkey.pem -out server.key" to generate server.key file.
- Run command "openssl req -x509 -in server.req -text -key server.key -out server.crt" to generate server.crt file.

The client-side configuration is described in the following section.

3.2.2 Data Encryption

Enabling SSL connection for the Certificate Authentication process implies that the communication between a client and a PostgresSQL Server is encrypted. This means that when the database server accepts a client authentication request it creates a session where every exchanged message is crypted with the same password set in the *server.key* file.
In fact the database server looks in the data directory for the server private key and certificate in the *server.key* and *server.crt* files respectively. Those files must be set up correctly before an SSL-enabled server can start.

Encryption is one of the most important security principles. For example Encrypted sessions prevent network sniffers from intercepting sensitive data.

3.2.3 Securing database objects

Another security principle is the Least Privilege, that in general is applied to users of an operating system. In this thesis, this principle has been applied to the PostgreSQL Database, consiedering a system in which case several type of users can access and manipulate different database objects with different permissions.
Each user who is authorized to access a PostgreSQL Database has a unique username, which could belong to a named collection of users, called ''group''.

Least Privilege has been implemented as a Privileges Matrix where the objects represent rows, the privileges represent columns and the cell values indicate the state of privilege (if it is granted or not). Another way to represent the privileges matrix, similarly to PostgreSQL, is the access control list, or ACL. The privileges assigned to a table or other database object, are stored in an array in the *pg_class* system table (relacl column). citeKorryDouglas2006

In the JDrone database, a group ((*drone_users*) has been created, where its members have restricted access on database objects. For example a member of this group cannot create either a DB or a role. Moreover, members cannot insert records in the *account* table but only in the *session* table and they cannot delete any records. Finally this group does not inherit any privilege from its parent (if it is defined).
Table 3.3 describes all the granted privileges of the *drone_users* group.

Privileges	Objects
select	account, cc_profile, certificate, session, connection_type, account_view, ccprofile_view, host_ip, finder_profile_seq
update	account, cc_profile, session, finder_profile_seq
insert	session
execute	get_id_profile(), update_max_finder_seq(), update_account_activity_timer()
delete	

Table 3.3: Object privileges

The table 3.4 shows the ACL related to the *account* table, and the difference between a postgres user, that has all the privileges, and the drone_users group that can only read and write.

relname name	relacl aclitem[]
account	postgres=arwdDxt/postgres,drone_users=rw/postgres

Table 3.4: Account privileges

3.3 Business Logic

The communication between JDrone's components is encrypted by a Transport Layer Security (TSL) protocol. The JDroneClient establishes a secure connection with both server components, JDroneServerAS and JDroneServerLOG. These components establish a Secure Sockets Layer (SSL) connection with PostgreSQL through DAL. Each component, included the database is identified by an IP address and is ''listening'' to a configurable port. Both the IP and the port are specified into the configuration file.

TSL is a protocol based on SSL that requires the exchange of valid certificates. The Java platform uses a system called Java Keystore, that is a convenient mechanism for storing and deploying X.509 certificates and private keys. There are two basic types of Java Keystore divided as follows:

- Keystore contains a pair of public/private key certificates used by the server.
- Truststore contains the certificates used only by applications that act as client.

To establish a secure connection, the components of the business logic layer require the following files:

- server.jks, which is the keystore repository that contains the certificate and the key of the ServerAS.
- serverTrustStore.jks, which is the truststore repository that contains the certificates of ServerAS and database server.

- postgresql.p12, which is the keystore repository that contains the certificate and the key of the server that acts as the client for the database (postgres.crt and postgres.key).

The last file, mentioned above, represents the client identity. In fact a user must have this keystore authenticated on the JDrone server components. The JDroneClient component requires the following files:

- JDroneTrustStore.jks, which is the truststore repository that contains the certificate of the ServerAS.

- username.p12, which is the keystore repository that contains the client certificate and his private key.

In this thesis, the JDroneDAL component has been implemented and integrated with the existing components. The classes within this component were initially built by FireStorm tool according to the DAO pattern. Sequentially, all the DAO interfaces were customized according to the least privilege principle. For example, AccountDAO Interface provides methods to execute operations allowed by the application. The figure 3.2 shows the Class Diagram of the entire DAO package.

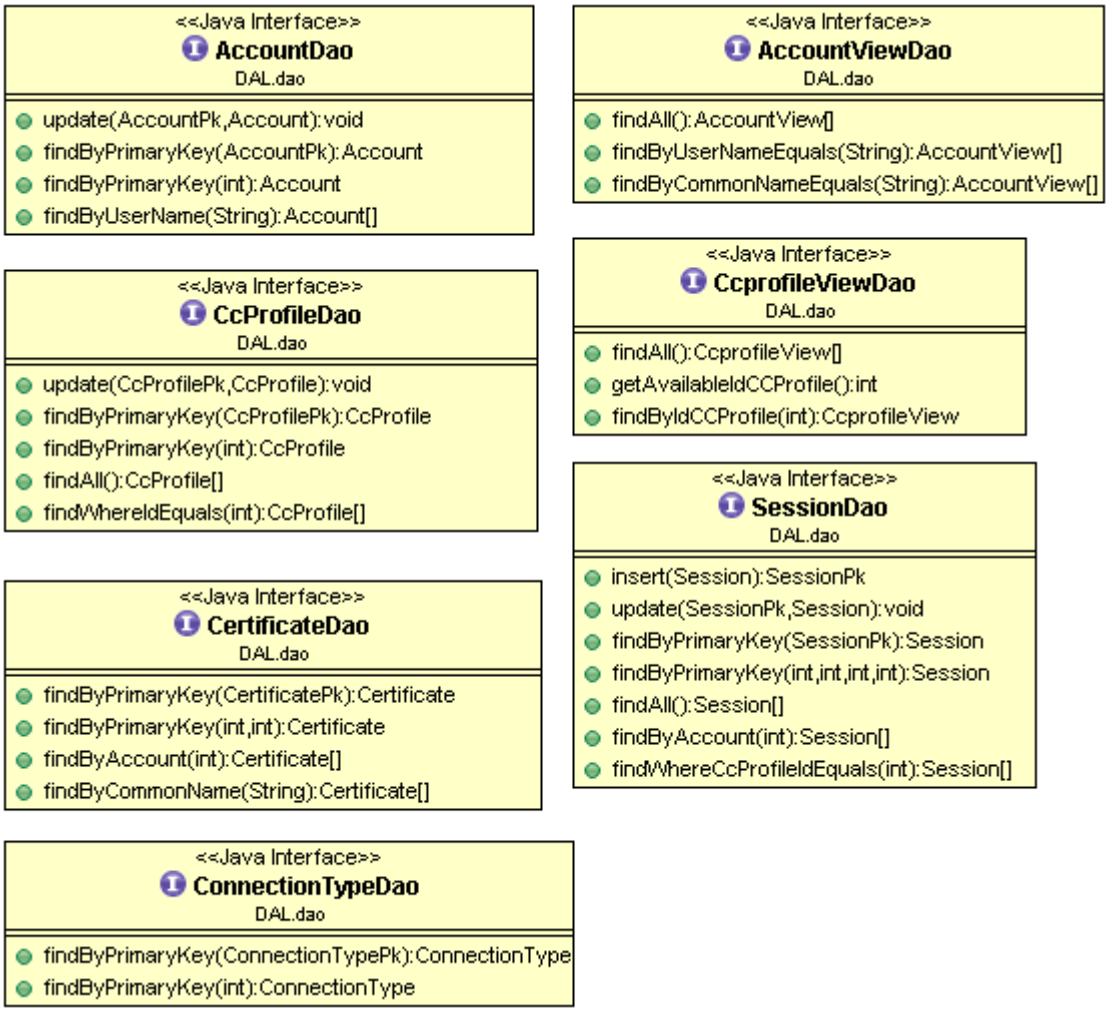


Figure 3.2: DAO Class Diagram

JDroneDAL contains various packages. Namely:

- DAO package, which contains one public interface for each database object managed. Each interface provides methods to interact with the database. The interfaces are used by the upper layer, in which there are JDroneServerAS and JDroneServerLog.

- DTO package, which contains the Data Transfer Objects; a DTO object is used to retrieve data from database and defines setter and getter methods related to the table columns. DTO objects have a public scope since they are used to transfer data between components (i.e. JDroneServerAS and JDroneDAL).

- Factory package, which contains the factory classes used to retrieve DAO objects.

- Jdbc package, which contains DAO implementation code based on JDBC.

- Exceptions package, which contains custom exception object for each DAO object.

The primary class of JDroneDAL project is ResourceManager (within jdbc package). ResoucerManager takes care of establishing the connection with the database using JDBC PostgreSQL library. ResourceManager class contains a static function (*getConnection()*) in order to retrieve a JDBC connection object. Every implementation of any DAO interface retrieves a connection object before executing any command (i.e, insert, update, etc). The connection object is static and this means that it is instanced only once.

3.3.1 Database Connection

JDBC PostgreSQL library provides the common interfaces to interact with the database server. In addition it allows enabling and customizing the SSL

connection.

JDBC represents the database as an URL in this form:

"jdbc:postgresql://host:port/database"; host, port and database are connection parameters related to the database server instance.

ResourceManager sets all the previous standard connection parameters as well as some additional properties specific to PostgreSQL, which are supported by the jdbc driver.

The following example shows the method used in order to establish the connection with JDrone database:

```
String url = "jdbc:postgresql://localhost/test";
Properties props = new Properties();
props.setProperty("user","mrdrone");
props.setProperty("ssl","true");
props.setProperty("sslfactory", "CertAuthFactory");
Connection conn = DriverManager.getConnection(url, props);
```

Listing 3.1: JDBC SSL Connection

The class name specified by sslfactory parameter must extend to javax.net.ssl.SSLSocketFactory and must be available to the driver's classloader. This means that CertAuthFactory class is automatically loaded by the java classloader. This class represents the client-side configuration of Certificate Authentication provided by PostgreSQL (section 3.2).

Within the CertAuthFactory class, the truststore and the keystore that contain the private key of the database user are specified. The password used to verify the database user's identity is prompted on ServerAS startup, in order to avoid hard-coded password.

These Java Keystores include the certificates used for both authentications toward the server components and the PostgreSQL database. Therefore, it is very important to specify as username the same value present into the certificate.

This authentication process is completely transparent for the JDrone client user and only the administrator is responsible for configuring all the certificates and starting the ServerAS.

In summary, CertAuthFactory class verifies the certificate and creates the SSLContext object used for encrypted communication.

3.3.2 Client Authentication

When JDroneServerAS receives a client authentication request, it instances a new thread that corresponds to the ASServerThread class. On startup, the thread verifies the user's identity querying the database through JDroneDAL.
The thread receives an SSLSock object which contains the Common Name of the client certificate. The Common Name value is used as a parameter for ccprofile_view query. In summary, the client authentication process is completed within the following steps:

- Create a new thread dedicated for client-server communication.
- Retrieve DAO object related to the Account entity using the factory class.
- Retrieve result-set by calling *findByUserName()* method.

- Verify the number of rows of the result-set. In case it is one, it means that authentication was successful. The *last_login* field is updated with the current timestamp. Finally, if the result-set is empty then an error is logged and the socket is closed.

The previous steps are carried out in the *isEnabledAccount()* method, as described in the appendix.

3.3.3 Acquiring CCProfile

JDroneServerAS provides CCProfile's data to the client. Primarily, the component tries to acquire an available Profile ID and then it gets the complete CCProfile data related to that ID. The acquiring process is the following:

- Retrieve DAO object related to *CCProfileView* entity using the factory class.
- Execute query on *get_id_profile()* function (as mentioned in the section 3.1.4). This database function returns a CCProfile's ID, if it is available.
- Retrieve DTO object related to *CCProfileView* entity by using the *findByIdCCProfile()* function passing the previously found ID as parameter.
- Return all CCProfile data as a string; toString() method, within *CCProfileView* DTO object, builds that string.

The string containing CCProfile data is built in such a way that is recognizable by the JDroneClient.

For example:

```
C&C:202.111.158.169
port:13001
NICK 'apdoypu
USER 'apdoypu * wiss1.lulzimehodza.com :'apdoypu
JOIN #.serve1 kr
```

The first two rows show respectively the C&C Ip address and port number, while the next rows show IRC commands. In addition, the found CCProfile ID is stored in a hash table. The utility of this hash table is twofold:

- To detect who is currently connected to the server
- To detect which CC profile is associated to the server.

Java Hashtable object is used in this context because is thread safe when accessing data in an multi-threading application like the JDrone server components. This means that whenever it executes get() or put() methods it locks the data.

The entire acquiring procedure is carried out by the functions *findByIdCCProfile()* and *getAvailableIdCCProfile()*, as described in the appendix.

3.3.4 Client Activity Tracking

This is a new feature implemented in the current version and provides the computation of the total amount spent for the infiltration activity of Drone

for each user account.
When the user is authenticated and receives an available CCProfile, the User can start the infiltration activity. Henceforth, the JDroneClient component is independent, sending IRC commands to the C&C specified in the CCProfile data. Anyway, for every action, JDroneClient sends log messages to other components. More specifically it sends messages towards JDroneServerAS about Users's activity status and it sends messages about exchanging commands with C&C towards JDroneServerLOG. These messages are also stored on file system and into the log files.

This activity starts by trying the connection with the IRC. Once the connection has been established, the JDroneClient sends a command (START_AT) on socket toward the JDroneServerAS.
In the same way, in the end of this activity it sends another command (STOP_AT) on socket towards the JDroneServerAS.
When JDroneServerAS component receives the START_AT command, a new session object is created.
The process continues as follows:

- Create new DTO object related to the *session* table.
- Set session attributes such as account ID, CCProfile ID and starting time on current timestamp.
- Retrieve DAO object related to the *session* table using the factory class.

- Persist the new session object into the database using the related DAO object.
- Update end_at session field when the infiltration activity is completed or the timeout is expired.

It is important to remember that the last operation initiates a database trigger to update the *activity_timer* on the *account* table.

3.3.5 Error handling

Java platform provides a mechanism to handle errors and exceptions called try/catch statement. All critical statements are surrounded by a try-catch-finally-block.
All operations, within JDroneDAL and the other components, that interact with database use the try/catch statement. If an error occurs on atabase-side, then the statement within catch block provides a way to log error messages and then to show them to the user.
For example, if a user tries to execute a ''select'' command towards an object without having the required permission a Java exception is generated:

> Exception: ERROR: permission denied for sequence finder_profile_seq
> Where: PL/pgSQL function "getidprofile" line 9 at assignment

Chapter 4

Results

Introducing a relational database as data source and improving the features has provided many advantages. This chapter describes the main differences with previous JDrone versions and the advantages obtained.

4.1 Advantages

One of the key advantages of the new architecture is the layers' separation. This separation implies that the upper layers do not care about data access, their format and structure.
With the multi-tier architecture, it is also possible to manage multiple database instances, which means distributing the database on multiple servers to manage fault tolerance by replicating the instance, or, in addition, to improve scalability.
Moreover, the Data Access Layer is reusable and the component provides reusable data access from another client application.

In terms of security, the data-protection was the first issue solved.

All set of data used by the the JDrone 2.0 within the database were protected from unauthorized access and tampering. Certificate Authentication and user access profiling were used for this purpose.
An authenticated User can access the data and based on the privileges to proceed respectively to the required actions. e.g. a standard user can access the CCProfile data in read-only mode. Apart from the standard user, it is possible to create a super user with more privileges on data, which means creating a user access profiling. It is possible to create a profile for each domain data or function. For example:

- a user profile with the permissions to read and update a user account
- another profile with only the permission to write the CCProfile data
- another profile with the permission to query the session table
- etc.

It is useful to know how the database reacts when a malicious user tries to modify or insert a user account without having permissions on it.
This is an example of what happens if mrdrone, that is a standard user, tries to call an insert/update:

```
INSERT INTO drone.account VALUES(20,'maluser',true,null,'
    2000-01-01') account VALUES(20,'maluser',true,null,'
    2000-01-01');
********** Errore **********
ERROR: permission denied for relation account
Stato SQL: 42501
```

Listing 4.1: Unauthorized access error

In addition the database user's identity is verified by checking the user's certificate preventing identity spoofing.

The table 4.1 summarizes the existing database users

Property	Values		
Name	postgres	mrdone	reporter
OID	10	25596	25726
Account expires			
Can login?	Yes	Yes	Yes
Superuser?	Yes	No	No
Create databases?	Yes	No	No
Create roles?	Yes	No	No
Update catalogs?	Yes	No	
Inherits?	Yes	Yes	No
Connection Limit	-1	-1	-1
Member of	-	drone_users	-

Table 4.1: Login role properties

The JDrone 2.0 allows collecting the history of its own monitoring process. After collecting the data, it is possible to query and analyze it. For example, the administrator can extract information about how many times a C&C server has been used. Moreover, the user can also extract which CCProfile has generated the longest activity session. Last but not the least an additional information is whether a user has used different IP addresses.

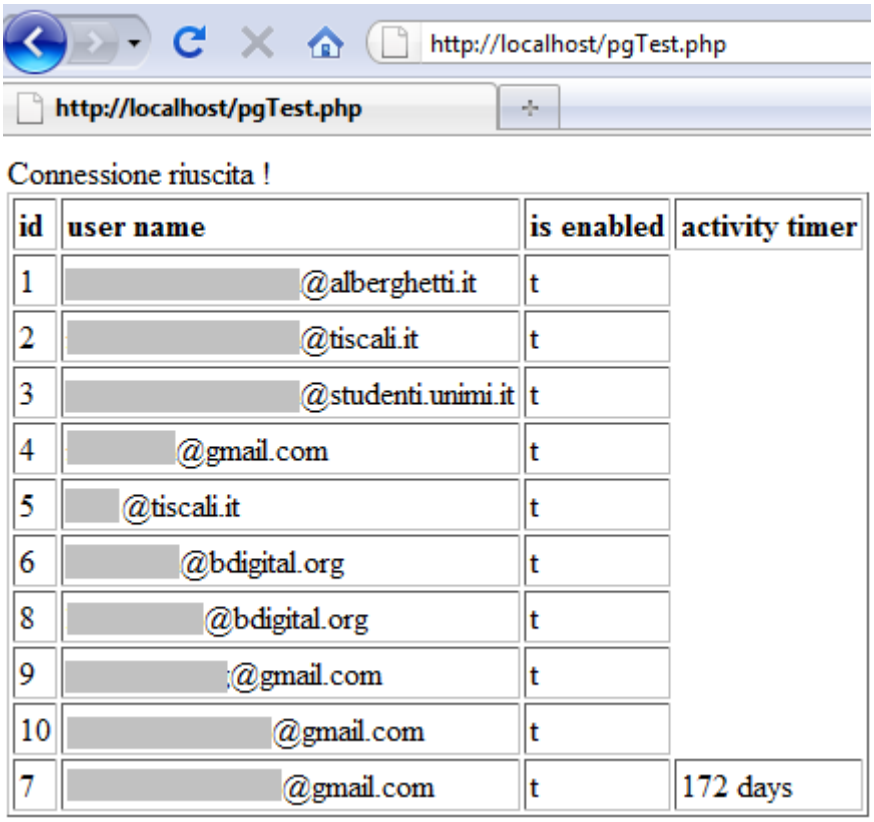
An additional user, called reporter, was created with only the read permission. This means that the specific User can be used only to query the data. It is very simple to access to the database from everywhere, e.g. console, client application or web application, etc.

The Listing 4.2 shows how to access a database from a PHP page with a

few code lines and extract data from the account table (Figure 4.1).

```
<?php
$conn  = @pg_connect('dbname=DBDrone user=reporter password=
    reporter');
$query = @pg_query("SELECT * FROM drone.account")
...
while($row = pg_fetch_assoc($query)) {
    ...
}
...
?>
```

Listing 4.2: PHP example



id	user name	is enabled	activity timer
1	@alberghetti.it	t	
2	@tiscali.it	t	
3	@studenti.unimi.it	t	
4	@gmail.com	t	
5	@tiscali.it	t	
6	@bdigital.org	t	
8	@bdigital.org	t	
9	@gmail.com	t	
10	@gmail.com	t	
7	@gmail.com	t	172 days

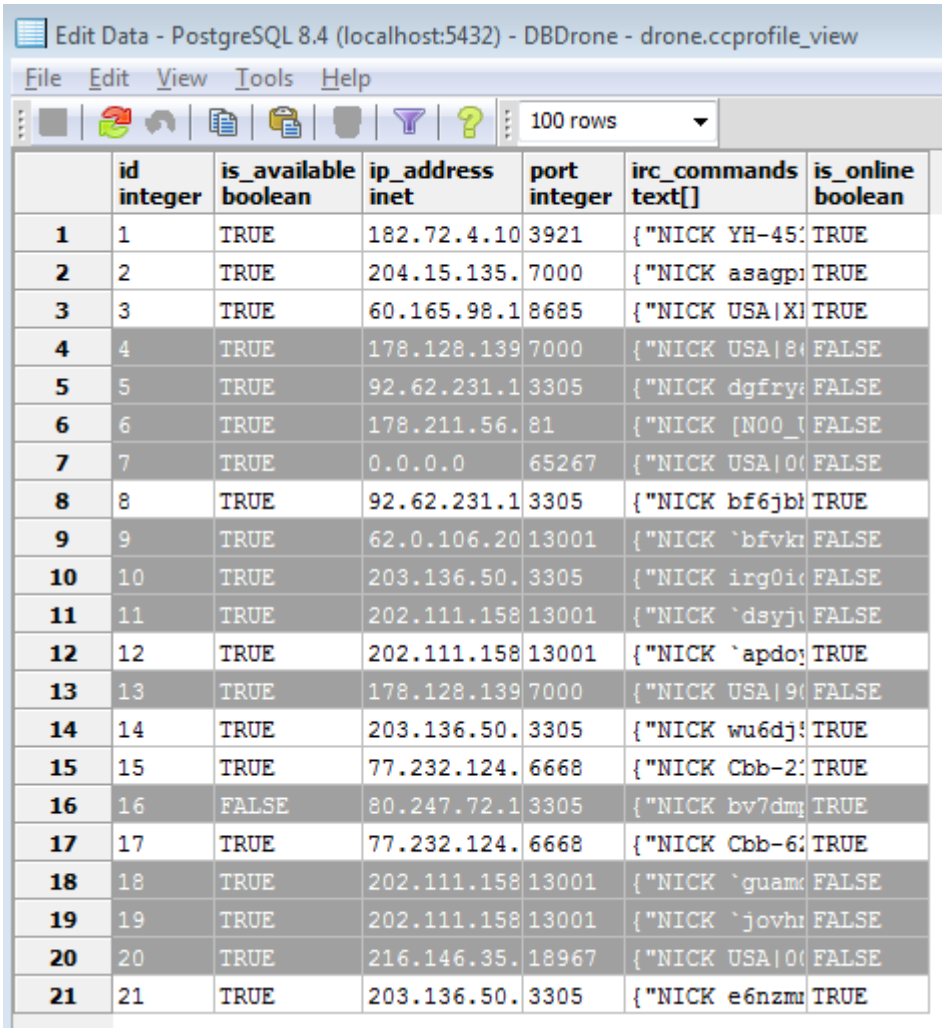
Figure 4.1: PHP Output

Moreover, in this new version of JDrone, the concurrency on CCProfile data was implemented completely on database-side, where the

database is the only responsible of the data distribution.

This choice led to some benefits. More specifically: sharing the data, saving the client resources, ensuring an exclusive use of CCProfile data and consumption of all "CCProfile data" available.

A simulation was carried out using five threads and some resources as shown in figure 4.2.



	id integer	is_available boolean	ip_address inet	port integer	irc_commands text[]	is_online boolean
1	1	TRUE	182.72.4.10	3921	{"NICK YH-45:	TRUE
2	2	TRUE	204.15.135.	7000	{"NICK asagpı	TRUE
3	3	TRUE	60.165.98.1	8685	{"NICK USA|XI	TRUE
4	4	TRUE	178.128.139	7000	{"NICK USA|8(	FALSE
5	5	TRUE	92.62.231.1	3305	{"NICK dgfrya	FALSE
6	6	TRUE	178.211.56.	81	{"NICK [N00_[	FALSE
7	7	TRUE	0.0.0.0	65267	{"NICK USA|0(	FALSE
8	8	TRUE	92.62.231.1	3305	{"NICK bf6jbl	TRUE
9	9	TRUE	62.0.106.20	13001	{"NICK `bfvkr	FALSE
10	10	TRUE	203.136.50.	3305	{"NICK irg0i(	FALSE
11	11	TRUE	202.111.158	13001	{"NICK `dsyju	FALSE
12	12	TRUE	202.111.158	13001	{"NICK `apdoy	TRUE
13	13	TRUE	178.128.139	7000	{"NICK USA|9(	FALSE
14	14	TRUE	203.136.50.	3305	{"NICK wu6dj5	TRUE
15	15	TRUE	77.232.124.	6668	{"NICK Cbb-2:	TRUE
16	16	FALSE	80.247.72.1	3305	{"NICK bv7dmp	TRUE
17	17	TRUE	77.232.124.	6668	{"NICK Cbb-6:	TRUE
18	18	TRUE	202.111.158	13001	{"NICK `guamc	FALSE
19	19	TRUE	202.111.158	13001	{"NICK `jovhr	FALSE
20	20	TRUE	216.146.35.	18967	{"NICK USA|0(	FALSE
21	21	TRUE	203.136.50.	3305	{"NICK e6nzmr	TRUE

Figure 4.2: Ccprofile view result

The figure 4.3 shows the output console messages that prove the following:

1. Round-Robin algorithm works correctly; in fact if a resource is released, a new one is gained.

2. Only the available/online resources are acquired.

3. No resources are used in more than one thread simultaneously.

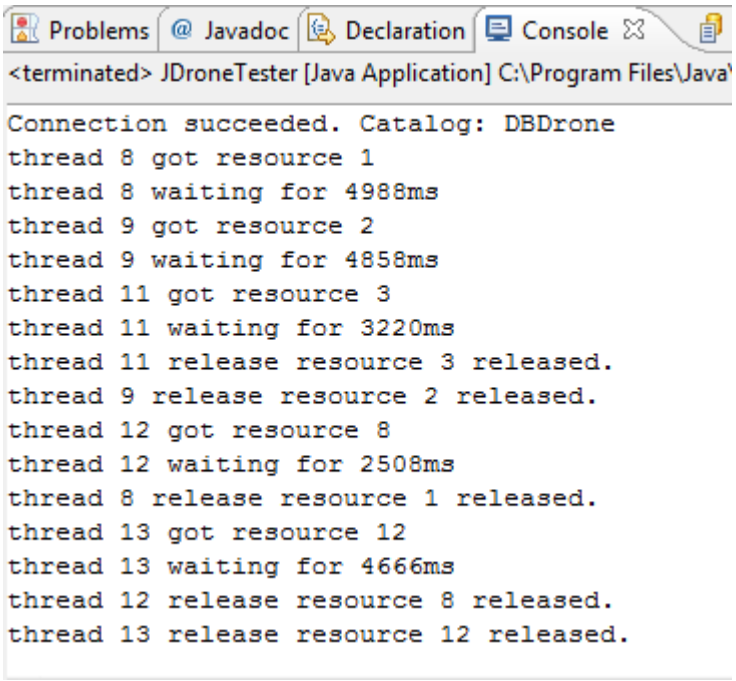


Figure 4.3: Acquiring resources output simulator

The Pie chart in figure 4.4 shows the available/not available resources.
The Line chart in figure 4.5 shows the average connection time for each drone.
The Bar chart in figure 4.6 shows the released daily resources (CCProfile) and the amount of daily successful authentications of the ServerAS.

The Line chart in figure 4.7 shows the waiting time for releasing a resource. These charts are real time generated through an AJAX web GUI, which belongs to the Dorothy framework.

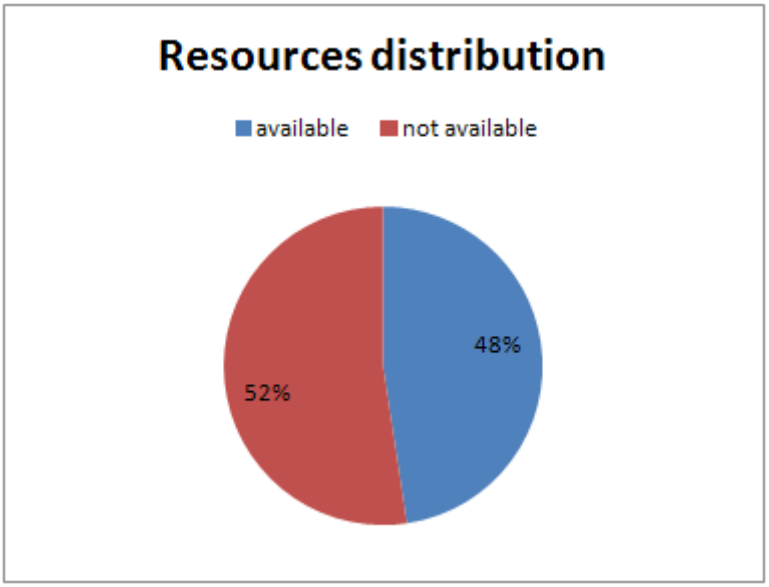


Figure 4.4: Resources distribution

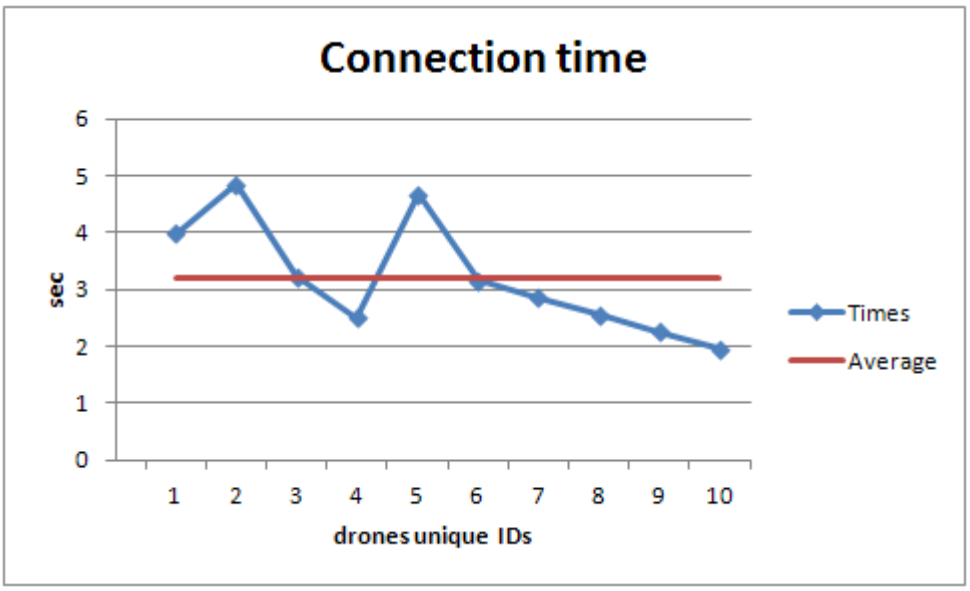


Figure 4.5: Drone connection time

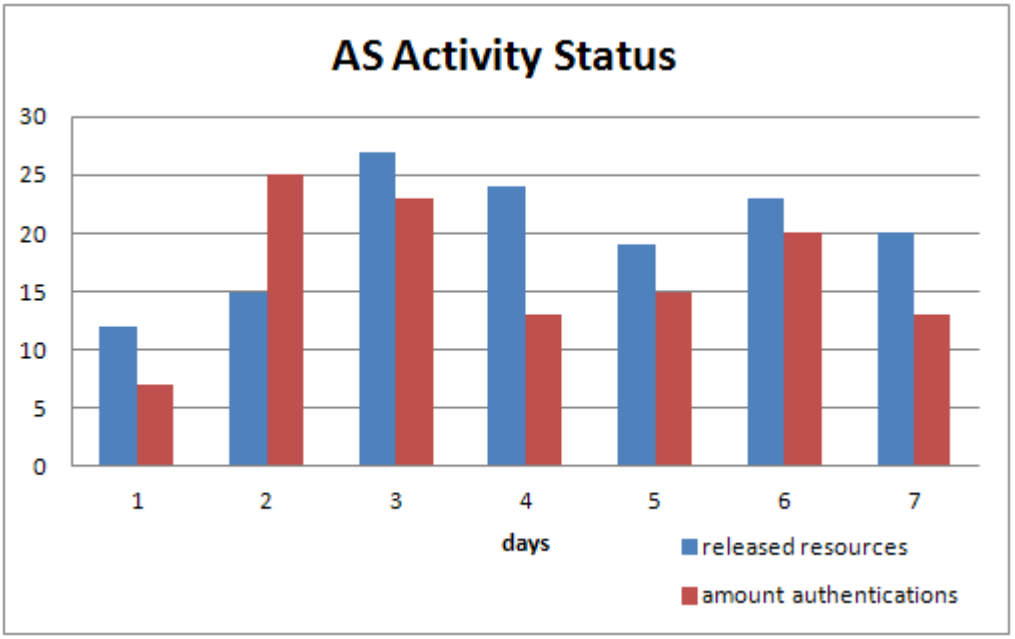


Figure 4.6: AS activity status

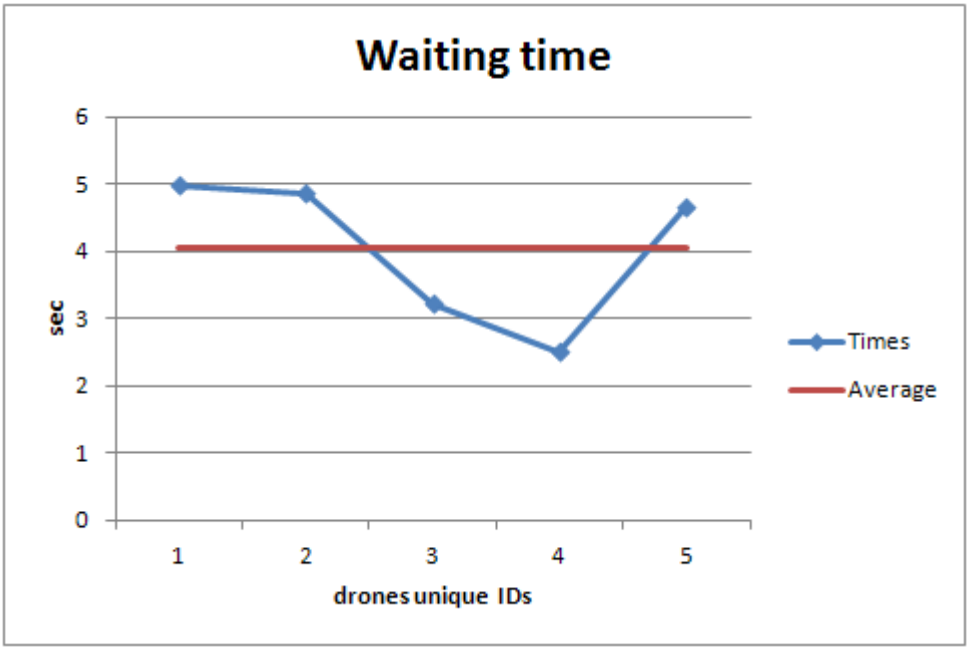


Figure 4.7: Drone waiting time

4.2 Future Work

One of the pending issue that can be managed in the future is the saving of all the log messages within the database.

This will allow further analysing and running simple queries on them extracting commands and other sensible data, such as time of receipt, nicknames, private messages, etc.

Moreover, storing log messages must be implemented in such a way to manage hundred or thousand simultaneously connected JDrone clients.

A common way to achieve this goal could be the ''database partitioning''. Partitioning refers to splitting what is logically one large table into smaller physical pieces. The partitioning could be based on various criteria and

each of them could have advantages and disadvantages.
For example, it is possible to create the partitions related to each CCProfile. In this case the advantage is that this partition allows the physical grouping of Profile's messages, while the disadvantage is that several messages could belong to a specific profile.

Currently, PostgreSQL supports partitioning via table inheritance. Each partition must be created as a child table of a single parent table. The parent table itself is normally empty and only represents the entire data set. In PostgreSQL it is possible to implement a Range Partitioning, where the table is partitioned into "ranges" defined by a key column or set of columns with no overlap between the ranges of values assigned to different partitions.

Another architectural improvement refers to communication among JDrone components. Currently the exchanged messages are represented as strings, and every component needs to parse it. Replacing the strings with a Data Transfer Object leads mainly to these advantages:

- Reduce the overhead due to parsing
- Transparent messages structure for the developer, in fact a DTO contains the necessary properties to read data.

Appendix

Stored Procedure get_id_profile

```
CREATE OR REPLACE FUNCTION drone.getidprofile()
  RETURNS integer AS
$BODY$
DECLARE
    id_profile drone.cc_profile.id%TYPE;
    next_id drone.cc_profile.id%TYPE;
BEGIN
    id_profile = 0;
    next_id = 0;

    LOCK TABLE ONLY drone.cc_profile IN ACCESS EXCLUSIVE MODE;
    next_id = nextval('drone.finder_profile_seq');
    SELECT INTO id_profile id
    FROM        drone.ccprofile_view
    WHERE       is_available = true
    AND         is_online = true
    AND         ccprofile_view.id >= next_id
    ORDER BY    ccprofile_view.id
    LIMIT 1;
    IF (id_profile > 0) THEN
       update drone.cc_profile set is_available = false where id
          = id_profile;
       --update sequence,imposta il valore attuale, serve a
          gestire i buchi degli id
       PERFORM setval('drone.finder_profile_seq', id_profile,
          true);
    END IF;

    RETURN id_profile;
END
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
```

Stored Procedure update_max_finder_seq

```
CREATE OR REPLACE FUNCTION drone.update_max_finder_seq()
  RETURNS trigger AS
$BODY$
BEGIN
    EXECUTE 'ALTER SEQUENCE drone.finder_profile_seq MAXVALUE '
        || NEW.id;
    RETURN NULL;
END
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
```

Stored Procedure update_account_activity_timer

```
CREATE OR REPLACE FUNCTION drone.update_account_activity_timer()
  RETURNS trigger AS
$BODY$
DECLARE delta INTERVAL;
DECLARE current_timer INTERVAL;
BEGIN
IF OLD.end_at IS NULL AND NEW.end_at IS NOT NULL THEN
  SELECT INTO current_timer activity_timer FROM drone.account
      WHERE id = NEW.account_id;
   delta = NEW.end_at - NEW.start_at;
  EXECUTE 'UPDATE drone.account SET activity_timer = ''' ||
      delta+current_timer || ''' WHERE id = ' || NEW.account_id;
END IF;
RETURN NULL;
END
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
```

View account_view

```
CREATE OR REPLACE VIEW drone.account_view AS
 SELECT account.user_name, certificate.common_name, certificate.
     email
   FROM drone.account
   JOIN drone.certificate ON account.id = certificate.account_id
  WHERE account.is_enabled = true;
```

View ccprofile_view

```
CREATE OR REPLACE VIEW drone.ccprofile_view AS
```

```
SELECT cc_profile.id, cc_profile.is_available, host_ip.
    ip_address, host_ip.port, cc_profile.irc_commands, host_ip.
    is_online
  FROM dorothive.host_ip
  JOIN drone.cc_profile ON host_ip.id = cc_profile.host_id;
```

Fuction getAvailableIdCCProfile

```
@Override
public int getAvailableIdCCProfile() throws
    CcprofileViewDaoException {

    // declare variables
    final boolean isConnSupplied = (userConn != null);
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    int ret = 0;

    try {
        // get the user-specified connection or get a
            connection from the
        // ResourceManager
        conn = isConnSupplied ? userConn : ResourceManager.
            getConnection();

        // construct the SQL statement
        final String SQL = "SELECT getidprofile AS id FROM
            drone.getidprofile()";

        if (logger.isDebugEnabled()) {
            logger.debug("Executing " + SQL);
        }

        // prepare statement
        stmt = conn.prepareStatement(SQL);
        stmt.setMaxRows(maxRows);

        rs = stmt.executeQuery();

        // fetch the results
        if (rs.next()) {
            ret = rs.getInt("id");
        }
        return ret;
    } catch (Exception _e) {
        logger.error("Exception: " + _e.getMessage(), _e);
        throw new CcprofileViewDaoException(
                "Exception: " + _e.getMessage(), _e);
```

```
        } finally {
            ResourceManager.close(rs);
            ResourceManager.close(stmt);
            if (!isConnSupplied) {
                ResourceManager.close(conn);
            }

        }

    }
```

Function findByIdCCProfile

```
    @Override
    public CcprofileView findByIdCCProfile(int idProfile)
            throws CcprofileViewDaoException {
        // declare variables
        final boolean isConnSupplied = (userConn != null);
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;

        try {
            // get the user-specified connection or get a
                connection from the
            // ResourceManager
            conn = isConnSupplied ? userConn : ResourceManager.
                getConnection();

            // construct the SQL statement
            final String SQL = SQL_SELECT + " WHERE id=" +
                idProfile;

            if (logger.isDebugEnabled()) {
                logger.debug("Executing " + SQL);
            }

            // prepare statement
            stmt = conn.prepareStatement(SQL);
            stmt.setMaxRows(maxRows);

            rs = stmt.executeQuery();

            // fetch the results
            return fetchSingleResult(rs);
        } catch (Exception _e) {
            logger.error("Exception: " + _e.getMessage(), _e);
            throw new CcprofileViewDaoException(
                    "Exception: " + _e.getMessage(), _e);
```

```
    } finally {
        ResourceManager.close(rs);
        ResourceManager.close(stmt);
        if (!isConnSupplied) {
            ResourceManager.close(conn);
        }

    }
}
```

Function CCProfileView.toString

```
/**
 * Method 'toString'
 *
 * @return String
 */
public String toString()
{
    StringBuffer ret = new StringBuffer();
    ret.append( "DAL.dto.Account: " );
    ret.append( "id=" + id );
    ret.append( ", userName=" + userName );
    ret.append( ", isEnabled=" + isEnabled );
    ret.append( ", lastLoginAt=" + lastLoginAt );
    ret.append( ", createDate=" + createDate );
    ret.append( ", activityTimer=" + activityTimer );
    return ret.toString();
}
```

Function isEnabledAccount

```
private boolean isAccountEnabled(SSLSession session)
        throws SSLPeerUnverifiedException {
    boolean accountFound = false;
    user_name = "";

    X509Certificate cert[] = session.getPeerCertificateChain()
        ;
    Principal p = cert[0].getSubjectDN();
    if (p instanceof Principal) {
        System.out.println("Nome: " + p.getName());
        user_name = p.getName();
        if (user_name.contains("=")) {
            user_name = user_name.split("=")[1];
        }

        Account[] _result = null;
        try {
```

```
            AccountDao _dao = AccountDaoFactory.create();

            _result = _dao.findByUserName(user_name);
            if (_result != null && _result.length > 0) {
               accountFound = true;
               //0 because is not yet started a session
               this.buffer.addAccountId(_result[0].getId(),0);
               // Set last_login_date
               Account account = _result[0];
               account.setLastLoginAt(new Date());
               _dao.update(account.createPk(), account);
            }
            _dao = null;
            _result = null;
         } catch (Exception ex) {
            ex.printStackTrace();
         }
      }
      return accountFound;
   }
```

Function findByUserName

```
   /**
    * Returns all rows from the account table that match the
      criteria
    * 'user_name = :userName'.
    */
   public Account[] findByUserName(String userName) throws
      AccountDaoException {
      return findByDynamicSelect(SQL_SELECT
            + " WHERE user_name = ? ORDER BY user_name",
            new Object[] { userName });
   }
```

Function getConnection

```
   public static synchronized Connection getConnection() throws
      SQLException {
      if (driver == null) {
         try {
            Class jdbcDriverClass = Class.forName(JDBC_DRIVER);
            driver = (Driver) jdbcDriverClass.newInstance();
            DriverManager.registerDriver(driver);
         } catch (Exception e) {
            System.out.println("Failed to initialise JDBC driver
               ");
            e.printStackTrace();
         }
```

```
        }
        Properties props = new Properties();
        props.setProperty("user", JDBC_USER);
        props.setProperty("ssl", "true");
        props.setProperty("sslfactory", "DAL.jdbc.CertAuthFactory"
            );
        // this password must be requested on server startup
        props.setProperty("sslfactoryarg", JDBC_PASSWORD);

        return DriverManager.getConnection(JDBC_URL, props);
    }
```

Class CertAuthFactory

```
package DAL.jdbc;

import java.io.FileInputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.security.GeneralSecurityException;
import java.security.KeyStore;
import java.security.SecureRandom;

import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;


public class CertAuthFactory extends SSLSocketFactory {

    public static String CONFIG_KEYSTORE_PWD = "org.postgresql.
        jdbc.keystore.password";
    public static String CONFIG_TRUSTSTORE_PWD = "org.postgresql.
        jdbc.truststore.password";

    public final static String SSL_PROTOCOL_NAME = "SSL";
    public final static String SECURE_RANDOM_NAME = "SHA1PRNG";

    protected SSLSocketFactory _internalFactory;

    public CertAuthFactory(String password)
    {
        SSLContext context;
        _internalFactory = null;
        CONFIG_KEYSTORE_PWD = password;
```

```
        CONFIG_TRUSTSTORE_PWD = password;
        //this password is used for serverTrustStore.jks
            postgresql.p12
        try {

            context = buildSSLContext();

            _internalFactory = context.getSocketFactory();
        } catch (IOException e) {

            e.printStackTrace();
        } catch (GeneralSecurityException e) {

            e.printStackTrace();
        }

    }


    /**
     * Builds an SSLContext with the specified trust store and
         key store
     * */
    protected static SSLContext buildSSLContext() throws
        IOException,
            GeneralSecurityException {
        FileInputStream fInKeyStore = null;
        FileInputStream fInTrustStore = null;
        SSLContext context = null;
        try {


            String trustPath = "";
            String trustPwd = CONFIG_TRUSTSTORE_PWD;
            String keyPath = "";
            String keyPwd = CONFIG_KEYSTORE_PWD;

            trustPath="serverTrustStore.jks";
            keyPath = "postgresql.p12";

            KeyManagerFactory managerFactory = null;
            TrustManagerFactory trustFactory = null;
            if (keyPath != null && !"".equals(keyPath)) {
                // Load the Key Managers
                KeyStore ks = KeyStore.getInstance("PKCS12");
                fInKeyStore = new FileInputStream(keyPath);
                ks.load(fInKeyStore, keyPwd.toCharArray());
                managerFactory = KeyManagerFactory.getInstance("
                    SunX509");
```

```
            managerFactory.init(ks, keyPwd.toCharArray());
        }
        if (trustPath != null && !"".equals(trustPath)) {
            // Load the trust store
            KeyStore trustKs = KeyStore.getInstance(KeyStore
                    .getDefaultType());
            fInTrustStore = new FileInputStream(trustPath);
            trustKs.load(fInTrustStore, trustPwd.toCharArray());
            trustFactory = TrustManagerFactory
                    .getInstance(TrustManagerFactory.
                        getDefaultAlgorithm());
            trustFactory.init(trustKs);
        }
        // Create + Initialize TLS context
        context = SSLContext.getInstance(SSL_PROTOCOL_NAME); //
             can
                                                 // be
                                                 // TLS
                                                 // too
        context.init(managerFactory.getKeyManagers(),
            trustFactory
                .getTrustManagers(), SecureRandom
                .getInstance(SECURE_RANDOM_NAME));
    }
    finally {
        try {
            if (fInKeyStore != null)
                fInKeyStore.close();
            if (fInTrustStore != null)
                fInTrustStore.close();
        } catch (IOException e) {
            // ignore it
        }
    }
    return context;
}

/**
 * Enables the client mode and the client authentication
 * */
protected SSLSocket enableClientAuth(SSLSocket sock) {
    sock.setNeedClientAuth(true);
    sock.setUseClientMode(true);
    return sock;
}

@Override
public Socket createSocket() throws IOException {
```

```
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            ();
        return enableClientAuth(sock);
    }

    @Override
    public Socket createSocket(InetAddress address, int port,
            InetAddress localhost, int localPort) throws
                IOException {
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            (address,
                port, localhost, localPort);
        return enableClientAuth(sock);
    }

    @Override
    public Socket createSocket(InetAddress host, int port) throws
         IOException {
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            (host, port);
        return enableClientAuth(sock);
    }

    @Override
    public Socket createSocket(Socket s, String host, int port,
            boolean autoClose) throws IOException {
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            (s, host,
                port, autoClose);
        return enableClientAuth(sock);
    }

    @Override
    public Socket createSocket(String host, int port) throws
        IOException,
            UnknownHostException {
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            (host, port);
        return enableClientAuth(sock);
    }

    @Override
    public Socket createSocket(String host, int port, InetAddress
         localAddress,
            int localPort) throws IOException, UnknownHostException
                {
        SSLSocket sock = (SSLSocket) _internalFactory.createSocket
            (host, port,
                localAddress, localPort);
```

```
        return enableClientAuth(sock);
    }

    @Override
    public String[] getDefaultCipherSuites() {
        return _internalFactory.getDefaultCipherSuites();
    }

    @Override
    public String[] getSupportedCipherSuites() {
        return _internalFactory.getSupportedCipherSuites();
    }

}
```

PHP Query Example

```
<?php
$conn = @pg_connect('dbname=DBDrone user=reporter password=
    reporter');

if(!$conn) {
    die('Connessione fallita !<br />');
} else {
    echo 'Connessione riuscita !<br />';

    if(!$query = @pg_query("SELECT * FROM drone.account"))
    die("Errore nella query: " . pg_last_error($conn));

    echo <<<EOD

    <table border="1" cellspacing="2" cellpadding="2">
        <tr>
            <td><b>id</b></td>
            <td><b>user name</b></td>
            <td><b>is anebled</b></td>
            <td><b>activity timer</b></td>
        </tr>

    EOD;

    while($row = pg_fetch_assoc($query)) {
        echo "\n\t<tr>\n\t\t<td>{$row['id']}</td>\n\t\t<td>{$row
            ['user_name']}</td>\n\t\t";
        echo "<td>{$row['is_enabled']}</td>\n\t\t<td>{$row['
            activity_timer']}</td>\n\t</tr>";
    }

    echo <<<EOD
```

```
    </table>
    EOD;

    pg_close($conn);
}
?>
```

Listing 3: PHP example

Glossary

BOTNET Network of linked zombies connected to one or more Command&Controls.

C&C (Command&Controls) Host that exchange commands with the zombie.

Zombie Health status of a system after the victim system had tried to connects to a remote system after a successful system violation.

Compromised Health status of a system after a system vulnerability has been exploited by an attacker.

TOR network Tor (The Onion Router) is a a system enabling its users to communicate anonymously on the Internet.

Round-Robin In computing, ''round-robin'' describes a method of choosing a resource for a task from a list of available ones, usually for the purposes of load balancing.

Hash table A table that contains ''hash values'', generally used for speeding up searching algorithms on sorted data. For example, a hash table for a dictionary might contain all the letters of the alphabet and the page numbers where each letter starts. That way, instead of searching through from start to finish for each word, you just use the hash table and get a big head start.

Firestorm FireStorm/DAO is a database access tool based on the Data Access Object design pattern.

Bibliography

[CH99] D. Soni C. Hofmeister, R. L. Nord. *Describing Software Architecture with UML*. Kluwer Academic Publishers, 1999.

[CPP06] Shari Lawrence Pfleeger Charles P Pfleeger. *Security in computing (fourth edition)*. Prentice hall, fourth edition, 2006.

[CR09] Marco Cremonini and Marco Riccardi. The dorothy project: An open botnet analysis framework for automatic tracking and activity visualization. *Computer Network Defense, European Conference on*, 0:52--54, 2009.

[CS06] James Holmes Chris Schalk, Ed Burns. *JavaServer Faces: The Complete Reference*. McGraw-Hill, 2006.

[EG02] Hans-Jürgen Schönig Ewald Geschwinde. *PostgreSQL developer's handbook*. 2002.

[KD06] Susan Douglas Korry Douglas. *PostgreSQL (second edition) the comphrensive guide to building, programming and administering PostgreSQL databases*. 2006.

[Pos] `http://www.postgresql.org/about/licence`.

[Ric08] Marco Riccardi. The dorothy project: inside the storm an automated platform for botnet analyses, 2008.