# Data Collection Framework
## *A Flexible and Efficient Tool for Heterogeneous Data Acquisition*

Luigi Sgaglione[1], Gaetano Papale[1], Giovanni Mazzeo[1], Gianfranco Cerullo[1], Pasquale Starace[1] and Ferninando Campanile[2]
[1]*Department of Engineering, University of Naples "Parthenope", Naples, Italy*
[2]*Sync Lab S.r.l., Naples, Italy*

Keywords: Big Data, Data Collection.

Abstract: The data collection for eventual analysis is an old concept that today receives a revisited interest due to the emerging of new research trend such Big Data. Furthermore, considering that a current market trend is to provide integrated solution to achieve multiple purposes (such as ISOC, SIEM, CEP, etc.), the data became very heterogeneous. In this paper a flexible and efficient solution about the data collection of heterogeneous data is presented, describing the approach used to collect heterogeneous data and the additional features (pre-processing) provided with it.

## 1 INTRODUCTION

Current market shows a trend of the vendors to offer integrated solution to their customers in the domain of the Big Data. This is the case, for example, of the Information Security Operations Center (ISOC) where enterprise information systems (applications, databases, web sites, networks, desktops, data centers, servers, and other endpoints) are monitored, assessed, and protected using advanced processing techniques (Complex Event Processing CEP). Another example is the new generation of Security Information and Event Management (SIEM) systems that combine Security Information Management (SIM) and Security Event Management (SEM) technologies (Coppolino et al., 2015). These examples are characterized by the high heterogeneity of the data produced by the sensors used to monitor the infrastructure to be protected.

The Data Collection Framework that is presented in this paper aims at mastering the data heterogeneity by providing a flexible and efficient tool for gathering and normalising security relevant information. Furthermore, we believe that, in many cases the possibility to perform a coarse-grain analysis of the data at the edge of the domain to be monitored can provide relevant advantages, such as an early detection of particular conditions, a reduction of the data stream volume that feeds the data and event processing platform, and the anonymization of the data with respect to privacy requirements.

This tool has been developed in the context of the LeanBigData (LBD) EU project. LeanBigData targets at building an ultra-scalable and ultra-efficient integrated big data platform addressing important open issues in big data analytics.

This paper is organized as follows. Section 2 describes the developed Data Collection Framework, Section 2.1 provides global implementation details, Section 2.2 describes the DCF Input Adapters, Section 2.3 describes the DCF Output Adapters, Section 2.4 describes the Processing At The Edge Component, Section 2.5 describes the DCF deployment schemas, and Section 3 provides some concluding remarks.

## 2 DATA COLLECTION FRAMEWORK

The main features of the Data Collection Framework (DCF) are 1) effective (i.e. high throughput) collection of data via a declarative approach and 2) efficient inclusion of new data sources. The Data Collection Framework enables the user to configure data sensors via a declarative approach, and to transform data that can be described via EBNF (Extended Backus-Naur Form) notation into structured data, by leveraging a "compiler-compiler" approach (Compiler-compiler, 2015).

The data collection framework allows users:

- to specify via a grammar the structure of the data to be collected,
- to provide a bidirectional data collection layer, with the capability to perform simple operations on data, such as aggregation, filtering, and pattern recognition, at the edge of the domain,
- to provide sensors for a broad set of common data sources, such as: operating system and server logs, network and security device logs, CPU information, VM information, IPMI, and application level information.

The Data Collection Framework is able to translate the format of virtually any data source into a general format in a declarative manner, by providing the translation scheme.

Figure 1 shows the global architecture of the Data Collection Framework.

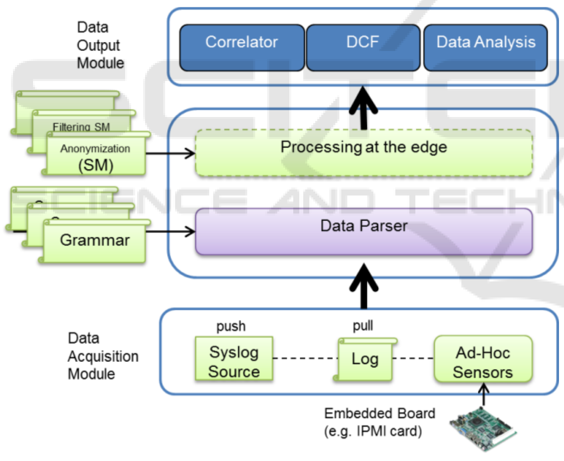

Figure 1: DCF architecture.

It is composed of three main building blocks, namely:

1. Data Acquisition module;
2. Data Parsing/Processing at the edge module;
3. Data Output module.

The Data Acquisition module is the part responsible for acquiring data from the sources, and of forwarding them to the Data Parsing/ Processing at the edge module. The Data Acquisition module provides many Agents in order to gather data from different types of sources and uses two possible collection modes:

1. Push - with this mechanism, the sources are responsible for forwarding data to the acquisition agent. The Syslog protocol (Syslog, 2015) is used to allow a simple integration of systems that already use this protocol to log their data.
2. Pull - with this mechanism, the agent reads data directly from the source (log file).

The Data Parsing/Processing at the edge module is in charge of transforming raw data to structured data by leveraging a "compiler-compiler" approach. Furthermore, each generated parser also converts raw data to a format compatible with the one used by the LeanBigData CEP and possibly pre-processes them using a State Machine Approach (SMC, 2015) depending on the requirements of the application domain. The result of this module is passed to the Data Output module.

The Data Output module is in charge of forwarding parsed data to one or more components responsible for attack detection.

## 2.1 DCF Implementation

The Data Collection Framework has been implemented in Java for the following reasons:

- The APIs of the LeanBigData CEP are provided for this language;
- The "write once, run anywhere (WORA)" advantages of Java code allow to run an application on different architectures without rebuild it;
- Only the presence of a Java Virtual Machine (JVM) is required.

The DCF is provided as a single runnable jar file that can be executed by a user:

- to acquire raw data;
- to parse data;
- to format data;
- to pre-process data;
- to provide input to a correlator.

All these functionalities are fully integrated in a single component to allow the execution on a single machine (the host to monitor).

## 2.2 DCF Input Adapters

The DCF comes with a series of input adapters already available to cover a high range of sources:

- Grammar Based
  - Syslog source
  - Log source

- Ganglia source
- DB source

New input adapters can be added in a simple way to allow future extensions. It is worth noting that at runtime only a single type of input adapter can be configured for each DCF instance (if more input adapters are configured only one will be executed).

### 2.2.1 Grammar based Input Adapters

This class of adapters covers a high range of possible sources. These adapters will be responsible for transforming raw data (that can be described using an EBNF notation) to structured data, leveraging a "compiler-compiler" approach, that consists in the generation of a data parser.

Since 1960, the tools that offer automatic generation of parsers are increased in number and sophistications. Today about one hundred different parser generator tools, including commercial products and open source software, are available. We analysed such tools and compared them taking into account the following requirements:

- Formal description of the target language in EBNF or EBNF-like notation;
- Generated parser in Java programming language;
- Optimized and high performance parsing;
- Low or no-runtime dependencies;
- BSD license;
- High quality error reporting;
- High availability of online/literature documentation.

The result of the comparison is that JavaCC (JAVACC, 2015) and ANTLR (ANTLR, 2015) are both valuable solutions for generating a parser. We selected them for the DCF implementation for the following motivations:

- Input Grammar notation

Both JavaCC and ANTLR accept a formal description of the language in EBNF notation. The EBNF notation is the ISO standard, well-known by developers and allows high flexibility.

- Type of parsers generated

JavaCC produces top-down parsers. ANTLR generates top-down parsers as well. A top-down parser is strongly customizable, simple to read and understand. These advantages allow high productivity and improve the debugging process.

- Output language

ANTLR, and in particular version 4, is able to generate parsers in Java, C# , Python2 and 3. JavaCC is strongly targeted to Java, but also supports C++ and JavaScript. The main advantage of a parser in Java is its high portability.

- Run-time dependencies

JavaCC generates a parser with no runtime dependencies, while ANTLR needs external libraries.

- Performance

To test the performance of JavaCC and ANTLR we have conducted many tests. One of the performance indicator that has been evaluated is the parsing time.

An example of the conducted tests (on the same machine – Windows 7 - 64 bit - i7 - 6GB) is the measurement of the time needed to parse the following mathematical expression:

$$\begin{gathered}11+12*(24/8)+(1204*3)+12*(24/8)+(1204*3)\\+12*(24/8)+(1204*3)+12*(24/8)+(1204*3)+1\\1+12*(24/8)+(1204*3)+12*(24/8)+(1204*3)+\\12*(24/8)+(1204*3)+12*(24/8)+(1204*3)+11\\+12*(24/8)+(1204*3)+12*(24/8)+(1204*3)+1\\2*(24/8)+(1204*3)+12*(24/8)+(1204*3)\end{gathered} \tag{1}$$

The grammar files written for the two parser generators are perfectly equivalent to have a common starting point. JavaCC is faster than ANTLR, in fact after repeated measures it is capable to parse the expression in an average time less than 3ms, while ANTLR(version 4) takes over 60ms.

- Generated code footprint

Starting from the same code used to evaluate the performance, JavaCC and ANTLR require, for the generated code, a comparable footprint (less than 20KB). ANTLR however, due to runtime dependencies, requires adding into the project an external library that takes about 1 MB.

- License

Both parser generators are under BSD license. BSD provides high flexibility for the developers and minimum restrictions for the redistribution of the software.

From this analysis, even if the features of JavaCC and ANTLR are comparable, the best performance in parsing and generation of the output source code, the smaller code footprint and the absence of runtime dependencies, have led us to select JavaCC as the parser generator to be used in the Data Collection Framework.

The adoption of the JavaCC parser generator, requires a declarative description of the data structure.

The declarative description of the data structure is provided via a grammar file (.jj) that describes the tokens and the relative semantics of the data to parse using an Extended-Backus-Naur Form (EBNF, 2015). The following classes have been used to optimize the integration of the parser with the DCF:

- The TupleElement class, to represent a parsed value in the paradigm (Name,Type,Value);
- The Tuple class, to represent a collection of parsed events (TupleElement)
- The ConvertAndSend class, to format the data in the DCF format and to forward them to the data analysis module.

The push and the pull operation mode have been implemented as shown in Figure 2.

In the push mode, the DCF instantiates a SysLog server to receive data using the SysLog protocol. In this way a generic method is provided to receive data from external sources and to integrate, with no overhead, sources that already use the SysLog format to log information.

In the pull mode data are retrieved from a log file. This method is useful when the sources save information in log files. The input adapters will be responsible for gathering data from the log files. This is implemented using a non-blocking (for the source) piped mechanism. In particular, the log file is read as a "RandomAccessFile" and, when it reaches the end of file, the reader goes into sleep mode for a pre-configured amount of time. On wakeup, it checks if there have been any changes in the file. The raw data is read and is pushed to an output pipe, which is connected to an input pipe that is used as input by the parsers during the parsing phase.

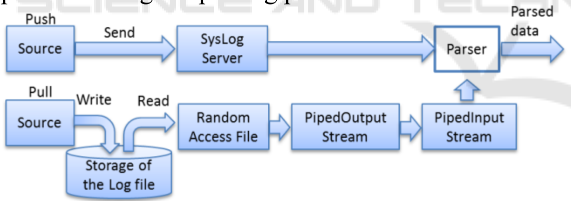

Figure 2: DCF push and pull mode.

### 2.2.2 Ganglia Source

This adapter has been implemented to retrieve data from sources that already use the Ganglia monitoring system (Ganglia, 2015). Ganglia is a scalable distributed monitoring system for high-performance computing systems, such as clusters and Grids. Ganglia allows to monitor a lot of physical machine parameters defined as metrics with very low per-node overhead and high concurrency. The physical machine metrics that are of interest to the application can be measured by appropriately configuring Ganglia. This input adapter uses a TCP server hosted by Ganglia to retrieve all monitored data.

### 2.2.3 DB Source Input Adapter

This adapter has been implemented to retrieve data from a specific table of a database. Typically, this adapter requires a trigger enabled on the table to be monitored in order to notify the adapter when a new value is ready. This adapter requires the table be ordered by means of an identifier of Integer type. This is needed to retrieve only new rows of the table.

## 2.3 DCF Output Adapters

The DCF provides the following output adapters to forward parsed data:

- Lean Big Data CEP Adapter
- Ganglia adapter
- TCP JSON adapter

New output adapter can be easily added to allow future extensions. It is worth noting that at runtime multiple types of output adapters can be configured for each DCF instance.

### 2.3.1 LDB CEP Output Adapter

This adapter is used to forward data to the Lean Big Data CEP using the LBD JCEPC driver. In particular, data are forwarded to a specific stream of a topology loaded into the LBD CEP.

### 2.3.2 Ganglia Output Adapter

This adapter is used to forward data to the Ganglia monitoring system. It is useful to create specific metrics using the DCF data. This adapter can be used in a machine running Ganglia.

### 2.3.3 TCP JSON Output Adapter

This adapter provides a general output adapter. It forwards data to a TCP server using the JSON format (JSON, 2015). An example of the output format is provided in the code below. This example is about the output produced by a DCF configured with a Ganglia input adapter and a TCP JSON output adapter. The code shows that all metrics configured in Ganglia are forwarded to the destination in a generic JSON format.

```
{"Type":"Long",
"Attribute":"ganglia_timestamp",
"Value":"1449740799000"},
{"Type":"String",
"Attribute":"ganglia_IP",
"Value":"127.0.0.1"},
{"Type":"String",
"Attribute":"ganglia_location",
"Value":"192.168.10.206"},
{"Type":"Double",
"Attribute":"cpu_user",
"Value":"6.4"},
{"Type":"Double",
"Attribute":"cpu_nice",
"Value":"0.0"},
{"Type":"Double",
"Attribute":"load_five",
"Value":"1.03"},
{"Type":"Double",
"Attribute":"cpu_system",
"Value":"5.6"},…
```

## 2.4 DCF Processing at the Edge

The “Processing At The Edge” (PATE) has been implemented using the State Machine Compiler technologies. This component is optional and can be enabled via a configuration file.

The idea is that the user can perform simple processing operation on the collected data before forwarding them to the output adapter. This pre-processing task allows for an early detection of specific conditions end events, a reduction of the data stream volume, and the anonymization of the data with respect to privacy requirements.

Many technological solutions have been evaluated, based on the following criteria:

- Simple processing operation definable by the user
- Low dependences
- Low overhead
- Good performance
- JAVA support

Based on the results of the evaluation process we adopted the Finite State Machine Approach. The user specifies the operations to perform on the data using a state machine description. Starting from the declarative description of the state machine the DCF generates and runs the corresponding finite state machine using the State Machine Compiler (SMC, 2015). The template for writing a state machine is provided in the following code excerpt.

```
%class Manager
%import com.parser.Tuple
%package com.stateMachine
%start StateMachineMap::State1
%fsmclass StateMachineName
%map StateMachineMap
%%
State1 {
  "transition(input:
Tuple)[condition]  NextState
{ actions…}"
…
}
…
StateN {…}
%%
```

In each state machine one or more states can be defined and each state is characterized by one or more transitions. Each transition has an input value of the Tuple type (i.e. a collection of parsed values), an activating condition (if the condition is true the transition is activated), a next state to be reached when the activating condition is satisfied, and a list of actions to be performed during the transition activation. The current implementation provides the following list of possible actions that can be performed while activating the transition:

- send(Tuple) to forward event to the DCF Output Adapter;
- print(Object) for debug purpose;
- getIntField(Tuple, String) to retrieve an integer field form the input Tuple;
- getStringField(Tuple, String);
- getDoubleField(Tuple, String);
- getTimestampField(Tuple, String);
- getLongField(Tuple, String);
- getFloatField(Tuple, String);
- addToAVG(int index, double value), resetAVG(int index), getAVG(int index) a set of functionalities to allow the computation of average values. Many averages can be managed, and each average is identified by a numeric index;
- incCounter(int index, int x), decCounter(int index, int x), getCounter(int index) a set of functionalities to allow the counting operation. Many counters can be managed, and each counter is identified by a numeric index;
- anonymize(Tuple tuple, String... name) to anonymize specific fields of the input tuple;
- addField(Tuple tuple, String name, types type, Object value) to add a new field to the input tuple;
- saveValue(String name, types type, Object value ), sendSavedData(String id), getIntSavedData(String name), getDoubleSavedData(String name), getLongSavedData(String name) to save and retrieve data between states;
- sendValue(String id, String name, types type, Object value) to forward specific value to the DCF Output Adapter.

## 2.5 DCF Deployment Schemas

Thanks to the high modularity of the DCF components three deployment schemas can be adopted: Basic Schema, Thin Client Schema, and Clustered Schema.

The Basic Schema represents the basic configuration of the DCF where a full instance of the DCF (including the PATE) is running on each data source as described in Figure 3.

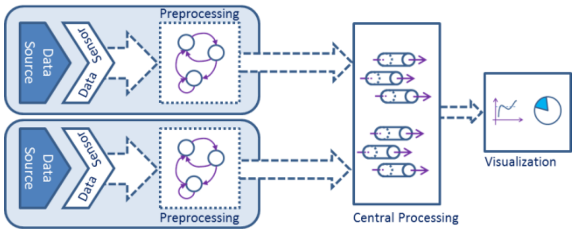

Figure 3: DCF Basic Schema.

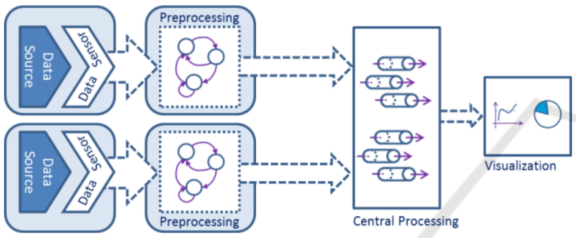

Figure 4: DCF Thin Client Schema.

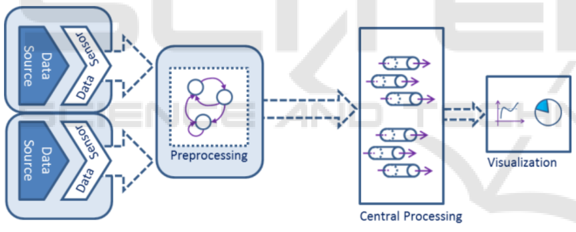

Figure 5: DCF Cluster Schema.

The Thin Client Schema (Figure 4) represents a configuration where only the data acquisition agent is deployed on the data source. The PATE component is possibly deployed in a separate machine. This is useful when the data source has low resources available (ex. Raspberry).

The Clustered Schema (

Figure 5) represents a configuration where all the gathered data are forwarded to a central PATE component, thus achieving a clustered data collection.

## 3 CONCLUSIONS

This paper presents the final implementation of a Data Collection Framework. The proposed framework provides an integrated and modular framework for instrumentation and performance analysis that has been explicitly designed to support the LeanBigData project, but can be used also in different context. The collection framework includes a number of features for achieving high-throughput, such as being able to handle thousands asynchronous data streams as well as having processing capabilities at the edge.

## ACKNOWLEDGEMENTS

The research leading to these results has received funding from European Commission within the context of Seventh Framework Programme (FP7/2007-2013) for research, technological development and demonstration under Grant Agreement No. 619606 (Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics, LeanBigData Project).

## REFERENCES

Coppolino, L.; D'Antonio, S.; Formicola, V.; Romano, L., 2014. *"Real-time Security & Dependability monitoring: Make it a bundle"*. In Security Technology (ICCST)

Compiler-compiler, 2015. Available from: princeton.edu: https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Compiler-compiler.html

SMC, 2015. Available from Sourceforge: http://smc.sourceforge.net/

Syslog, 2015. Wikipedia. Available from: https://en.wikipedia.org/wiki/Syslog

EBNF. 2015. Available from www.iso.org: http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153

JAVACC, 2015. Available from https://javacc.java.net/

ANTLR, 2015. Available from http://www.antlr.org/

Ganglia, 2015. Ganglia. Available from Sourceforge: http://ganglia.sourceforge.net

JSON, 2015. Available from JSON: http://www.json.org/