Horizon 2020 Program (2014-2020)

Big data PPP

Research addressing main technology challenges of the data economy

Industrial-Driven Big Data as a Self-Service Solution

D4.2: Distributed Event-Processing Engine[†]

Abstract: The main challenges in complex event processing are volume and velocity of data that needs to be processed in real-time. Data can come from different sources in different amounts and different frequencies, e.g. in the I-BiDaaS use cases from the three data providers CAIXA, CRF and TID where the data rates differ. However, not all data is connected, and one needs to divide the data into parts that need to be processed together. This document describes the design of an event-processing architecture where the event-processing engine is divided in separate, independent processing units that can share and exchange information and events. This reduces the load and enhances the performance without leaving out important relations between the data.

Contractual Date of Delivery	31/12/2019
Actual Date of Delivery	30/12/2019
Deliverable Security Class	Public
Editor	*Dr. Gerald Ristow (SAG)*
Contributors	FORTH, CRF
Quality Assurance	*Dr. Yolanda Becerra (BSC)* *Dr. Giorgos Vasiliadis (FORTH)* *Dr. Kostas Lampropoulos (FORTH)*

[†] *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780787.*

The *I-BiDaaS* Consortium

Foundation for Research and Technology – Hellas (FORTH)	Coordinator	Greece
Barcelona Supercomputing Center (BSC)	Principal Contractor	Spain
IBM Israel – Science and Technology LTD (IBM)	Principal Contractor	Israel
Centro Ricerche FIAT (FCA/CRF)	Principal Contractor	Italy
Software AG (SAG)	Principal Contractor	Germany
Caixabank S.A. (CAIXA)	Principal Contractor	Spain
University of Manchester (UNIMAN)	Principal Contractor	United Kingdom
Ecole Nationale des Ponts et Chaussees (ENPC)	Principal Contractor	France
ATOS Spain S.A. (ATOS)	Principal Contractor	Spain
Aegis IT Research LTD (AEGIS)	Principal Contractor	United Kingdom
Information Technology for Market Leadership (ITML)	Principal Contractor	Greece
University of Novi Sad Faculty of Sciences (UNSPMF)	Principal Contractor	Serbia
Telefonica Investigation y Desarrollo S.A. (TID)	Principal Contractor	Spain

Document Revisions & Quality Assurance

Internal Reviewers

1. *Dr. Yolanda Becerra, (BSC)*
2. *Dr. Giorgos Vasiliadis, (FORTH)*
3. *Dr. Kostas Lampropoulos, (FORTH)*

Revisions

Version	Date	By	Overview
1.1	27/12/2019	Dr. Gerald Ristow	Approval and submission of final version
1.0	18/12/2019	Dr. Gerald Ristow	Ready for final review
0.9	17/12/2019	Dr. Gerald Ristow	Considered suggestions from BSC and FORTH
0.8	03/12/2019	Ioannis Arapakis	Modified TID section
0.7	03/12/2019	Omer Boehm	Modified TDF section
0.6	29/11/2019	Dr. Gerald Ristow	Document ready for review
0.5	28/11/2019	Dr. Gerald Ristow	Added input from FORTH and CRF
0.4	25/11/2019	Dr. Gerald Ristow	Added section on distributed event processing
0.3	22/11/2019	Dr. Gerald Ristow	Added overview of Data Streams section and section on other input sources
0.2	07/11/2019	Dr. Gerald Ristow	Comments on the ToC incorporated.
0.1	21/10/2019	Dr. Gerald Ristow	ToC.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CEP	Complex Event Processing
EC	European Commission
EPL	Event Processing Language
GPU	Graphics Processing Unit
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
TDF	Test Data Fabrication
UM	Universal Messaging
WP	Work Package

Executive Summary

More and more data are generated to get new or more detailed insights e.g. into processes, machinery or the environment. Not all analysis can wait for the data to be stored in a database and then processed. Streaming analytics, i.e. analysing information in real-time or nearly real-time, becomes more and more important to react to complex scenarios in a suitable fashion. Such information is normally referred to as events and if the number of events becomes large, one needs a distributed event-processing engine. In the I-BiDaaS EU research project, such an engine is designed, tested and verified with many use cases from three data providers. The commercial event-processing engine Apama from SAG is used which allows separately running process instances to exchange information in various ways so that Big Data problems can be addressed easily. The most efficient way is to use an in-memory database for a fast inter-process communication. This mechanism can also be used to offload compute-intense tasks to GPUs which is described in detail in this deliverable as well giving some performance numbers to verify our approach.

1 Introduction

Nowadays, data is collected in many ways, e.g. via mobile devices, cars or sensors. The latter is mainly used in industrial production and one of I-BiDaaS main use cases. The collected data is mostly stored before it is analysed. However, to react faster and reduce production costs or machine downtime, it is desirable to get insight as fast as possible. This can be done by using a streaming approach where the data stream is analysed for specific patterns, also sometimes called *fingerprints*, before the data is stored. The information in the streams is normally referred to as *events* and if the amount and frequency of the events keep increasing, one soon needs a distributed event processing engine to be able to handle the workload. This approach was e.g. used in the research project BigPro[1].

Different approaches and products are available for this scenario[2]. A recent overview of the Apache open source frameworks is given here[3]. In I-BiDaaS, we have chosen the commercial product Apama provided by SAG for this task. A community version is also available, and the corresponding site[4] contains detailed description and documentation of the product. Different Apama program instances can exchange information via different means, especially interesting is the exchange of information via an in-memory solution which is described in section 2.1.

A clear innovation achieved by the I-BiDaaS platform is the connection of Apama with GPU-acceleration boards. Here the data is also exchanged via an in-memory solution to not lose too much of the GPU speed advantage to data access time. Details are given in section 2.2.

Even though Apama supports many input sources, like databases, files or HTTP requests, for the current use cases, we have mainly used input from the message broker Universal Messaging which is described in section 3.

Apama supports calling routines written in different programming languages like R, Java and Python. This is used in the CRF "Production process of aluminium Die-casting" use case by classifying events by calling a pre-trained Random Forest model written in Python, see section 3.2.1 for more details on the data used.

[1] See https://www.fir.rwth-aachen.de/en/research/research-work-at-fir/detail/bigpro-01is14011/

[2] See e.g. https://www.fusioncharts.com/blog/how-real-time-analytics-works-a-step-by-step-breakdown/

[3] See https://developer.ibm.com/code/2018/03/21/stream-processing-brief-overview/

[4] See http://www.apamacommunity.com/

2 Data Processing by the Event Processing

2.1 Distributed Event Processing

The engine that powers an Apama application is called Apama Correlator. Apama applications track inbound event streams and listen for events whose patterns match defined conditions by executing user-defined rules. Correlators execute the sophisticated event pattern-matching logic that you define in your Apama application. The correlator's patented architecture can monitor huge volumes of events per second where the exact number depends on the hardware specifications. Some scaling details are shown in Figure 1 which is taken from a report conducted by Intel[5].

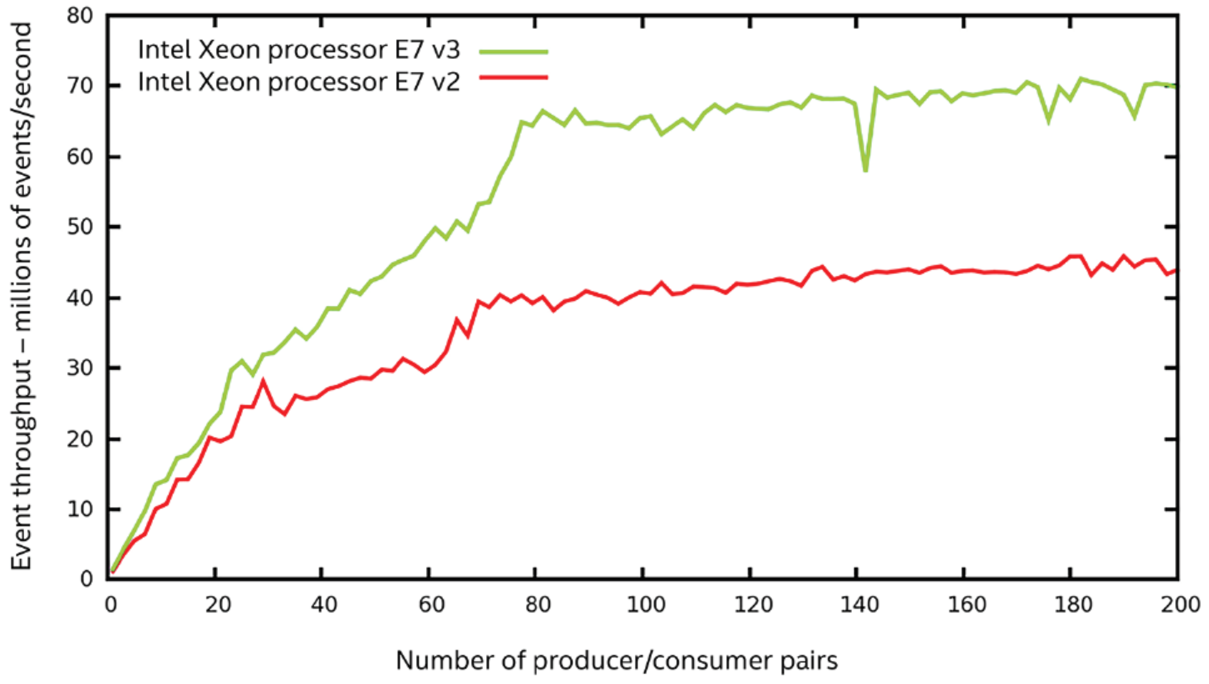

Figure 1. Internal event throughput of the Apama Correlator as function of producer/consumer context pairs

An example of a simple Apama script file that monitors incoming events and forwards them to another correlator when the float value in the event exceeds 10 is:

```
/**
 * A simple monitor - used to initially show functionality of
 * a new installation.
 *
 * The Event Correlator monitors for Tick events and if such an event is
 * received whose price is greater than 10.0 - it forwards (sends)
 * that event.
 */

/**
 * The event type (global) that we will use
 */
event Tick {
      string name;
      float price;
```

[5] See https://www.intel.de/content/www/de/de/big-data/apama-analytics-xeon-e7-v3-paper.html

```
}

/**
 * The monitor itself
 */
monitor simpleSend {

      action onload()
      {
            on all Tick(*, >10.0) as t
            {
                  send Tick(t.name, t.price) to "output";
            }
      }
}
```

To share and distribute the workload, one can use several correlators to form a cluster where each correlator in the cluster has a specific task, e.g. analyzing event patterns for a single use case. Correlators normally exchange information by sending events to other correlators which can be done in two ways:

1. Use the provided *engine_connect* utility or
2. Use of Universal Messaging to connect the correlators to the same set of Universal Messaging channels.

For building event-based application, we use EPL, which is Apama's native event processing language. When using EPL, there is a third mechanism to exchange data in an Apama cluster, using the so-called MemoryStore. The MemoryStore provides an in-memory, table-based, data storage abstraction within a correlator. All EPL code running in a correlator in any context can access the data stored by the MemoryStore. In other words, all EPL monitors running in a correlator have access to the same data. The Apama MemoryStore can also be used in a *distributed fashion* to provide access to data stored in a MemoryStore to applications running in a cluster of multiple correlators.

The MemoryStore can also store data on disk to make it persistent and copy persistent data back into memory. However, the MemoryStore is primarily intended to provide all monitors in the correlator with in-memory access to the same data.

In addition to those stores that are local to a single Apama process, Apama also supports a *distributed* store in which data can be accessed by applications running in multiple correlators. A distributed store makes use of Terracotta's TCStore or BigMemory Max, or a thirdparty distributed cache or datagrid technology that stores the data (table contents) in memory across a number of processes (nodes), typically across a number of machines. The collection of nodes is termed a *cluster*. One can use the same nodes as for the Apama cluster, however, normally different nodes are used to allow for a clearer and more flexible IT architecture.

A sketch of a distributed Apama cluster that exchanges data via a MemoryStore is shown in Figure 2. The Apama nodes are shown on top and the nodes of the MemoryStore which form a TCStore are shown below them. The Apama nodes access the TCStore via the TCStore API which provides an abstraction layer of the MemoryStore. The TCStore has a trigger mechanism so that the Apama nodes are informed when data is added, updated or deleted in the MemoryStore. Each node of the TCStore has a persistent storage device attached to it.

Such a setup provides a fast, through the in-memory component, and reliable, through the storage component, exchange of information between the Apama nodes leading to a scalable distributed event processing engine.

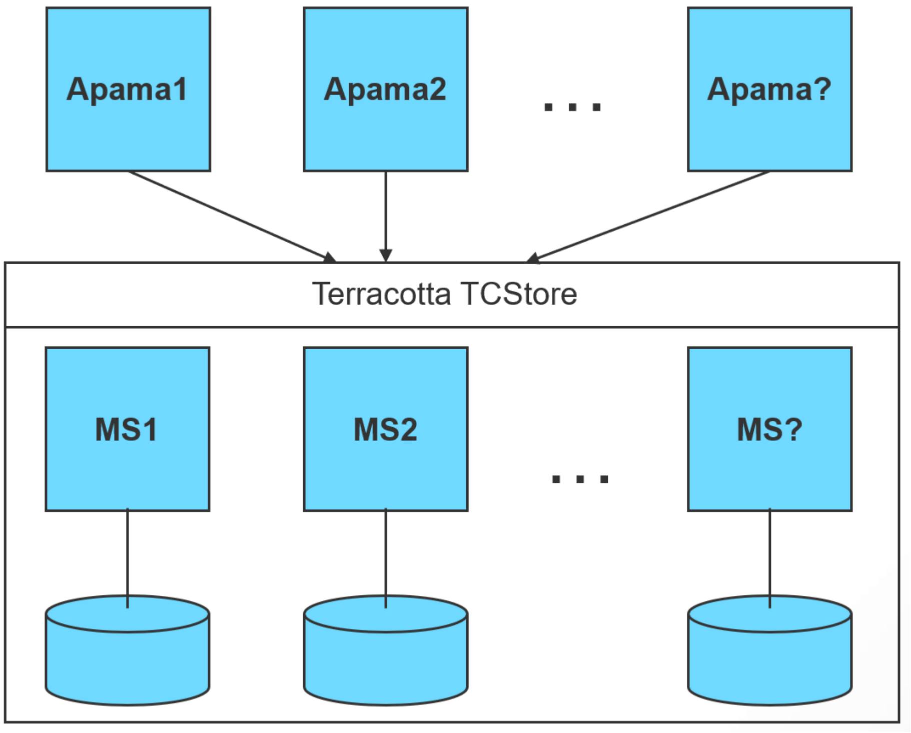

Figure 2. A distributed Apama cluster using a MemoryStore

2.2 Event Processing using GPU acceleration

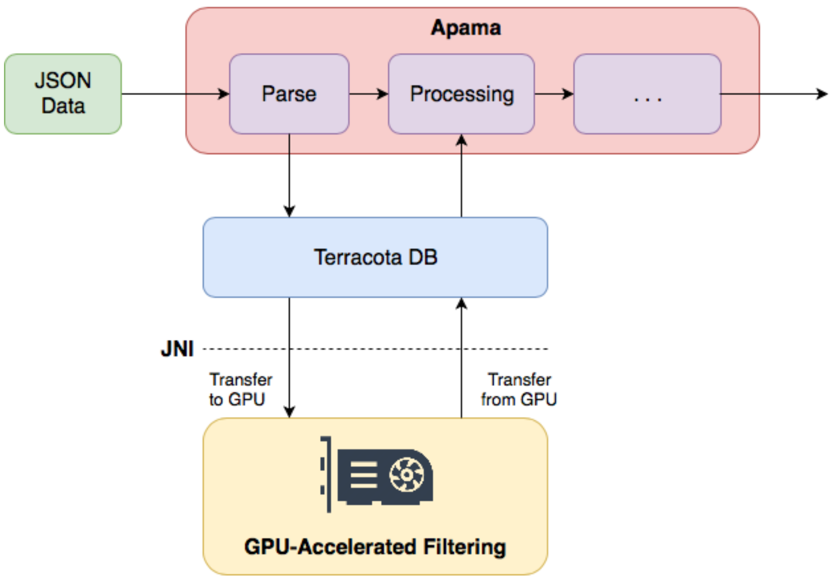

Figure 3. GPU-Accelerated filtering based on fixed strings and regular expressions.

In deliverable D4.1, we described the initial concept of how the Apama Stream Processing Engine can leverage the processing capabilities of GPUs, through the TerracottaDB. As depicted in Figure 3, the main concept is to utilize the TerracottaDB[6] as a buffer between the Apama Stream Processing engine and the GPU.

[6] See http://www.terracotta.org/documentation/ for documentation and further information.

In the following sections, we describe the actual implementation of the GPU-accelerated TerracottaDB, which can be leveraged transparently from the Apama Stream Processing Engine. The implementation consists of three phases: 1) Transfer to GPU, 2) GPU-Accelerated Filtering, and 3) Transfer from GPU.

2.2.1 Transfer to GPU

In order to read the entries from the TerracottaDB, we have to spawn a Reader instance on the Java side. The Reader instance gives us the ability to read the entries in a stream format without interrupting the execution of the Writer, who is responsible for the insertion of the new entries in the database that originate from Apama. Once the entries are gathered (from stream format) we convert them into a byte form: the reason for this conversion is due to the fact that the byte form in Java side can be easily read from C side, on the other hand leaving the data in String format (in Java) will cause a slight delay on the filter process due to the fact that more conversions are needed to take place.

The entries (in byte format from Java) are fed as input in a JNI function (implemented in C), which fetches the worker pointer from Java's stack frame; once this is done, the entries are temporarily stored in a host buffer called **`h_data`**. Once all the entries are copied to that specific buffer another memory copy takes place to transfer the **`h_data`** buffer to a GPU buffer called **`d_data`**. After initializing the last data copy, the filter process takes place.

2.2.2 GPU-Accelerated Filtering

As we have described in D4.1, the GPU-accelerated pattern matching operations are offered through an OpenCL library that supports both string searching and regular expression matching operations. The library provides a C/C++ API for processing incoming records, and returning any matches found back to the application.

Initially, all patterns are compiled to DFA state machines and state transition tables. The state table is compiled in the CPU and then copied and mapped to the memory space of the GPU. At the searching phase, each thread searches a different portion (i.e., a separate message) of the input data stream. In particular, each thread moves over the input data stream one byte at a time. For each consumed byte, the matching algorithm switches the current state according to the state transition table. The pattern matching is performed byte-wise, meaning that we have an input width of eight bits and an alphabet size of $2^8 = 256$. Thus, each state will contain 256 pointers to other states. The size of the DFA state transition table is $|\#\text{States}| * 1024$ bytes, where every pointer occupies four bytes of storage. When a final-state is reached, a match has been found, and the corresponding offset is marked. The format of the state table allows its easy mapping to the different memory types that modern GPUs offer. Mapping the state table to each memory yields different performance improvements.

2.2.3 Optimizations

One important optimization is related to the way the input data is loaded from the device memory. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput would be reduced by a factor of up to 32. To utilize the memory more efficiently, we redesigned the input reading process such that each thread is fetching multiple bytes at a time instead of one. We have explored fetching 4 or 16 bytes at a time using the **`char4`** and **`int4`** built-in data types, respectively, allowing the utilization of more than 50% of the memory bandwidth.

In addition to optimizing the device memory usage, we considered two other optimizations: the use of *page-locked* (or *pinned*) memory, and the reduction of the number of transactions between the host and the GPU device.

The page-locked memory offers better performance, as it does not get swapped. Furthermore, it can be accessed directly by the GPU through DMA. Hence, the use of page- locked memory improves the overall performance by reducing the data transferring costs to and from the GPU. The tuples are placed into a buffer allocated from page-locked memory, which can be transferred through DMA from the physical memory of the host to the device memory of the GPU.

Another thing to consider is the transfer of the incoming data to the memory space of the GPU. A major bottleneck for this operation is the extra overhead, caused by the PCIe bus that interconnects the graphics card with the base system. Unfortunately, the PCIe bus suffers many overheads, especially for small data transfers. To further improve performance, we use a large buffer to store the contents of multiple tuples, which is then transferred to the GPU in a single transaction, every time it gets full. This results in a reduction of I/O transactions over the PCI Express bus, which further results to increase throughput, as can been shown in Table 1.

Table 1. Sustained PCIe v2.0 Throughput (Gbit/s) for transferring data to a single GPU, when increasing the size of data that are transferred at once.

Buffer	**1KB**	**4KB**	**64KB**	**256KB**	**1MB**	**16MB**
Host to GPU	2.04	7.12	34.4	42.1	45.7	47.8
GPU to Host	2.03	6.70	21.1	23.8	24.6	24.9

2.2.4 Transfer from GPU

When the filtering stage is over, the GPU provides us with a report array that contains the patterns found. The size of the report array is defined from the number of entries multiplied by the number of cells the entry has. Using this report array, we are able to discard any entry that is unnecessary, allowing the remaining entries to be returned to the main Java program. Moreover, in order for the Java program to read the clean entries, we have to recast them again into String; we also measured those extra conversions and we will explain in detail in the evaluation stage.

Those filtered entries go through the final process stage. In each entry, we extract all the cell values where we eventually insert them into the final database and erase all the entries which the Reader has given to us previously.

2.2.5 Evaluation

In this section, we show the performance of our approach. For the evaluation, we use 500 unique name/surname pairs that they got inserted in the TerracottaDB 200 times each, in order to reach a more realistic scenario of 100k entries in the database. Then, we measure the performance of our GPU-based pattern matching and compare it with a CPU-based pattern matching implementation on CPU, using a standard Java implementation of the Aho-Corasick algorithm[7].

The setup of our benchmark is as follows. A separate entity (namely the Producer) is responsible to insert into the database the data described above in a streaming fashion. The data inserted into the database are then read by a Consumer instance, which is responsible to trigger our GPU-based pattern matching implementation and perform the filtering process. For the CPU version, the same instance uses the Java's original Aho-Corasick implementation. Furthermore, the filtered data are inserted in a separate database instance that also resides in TerracottaDB.

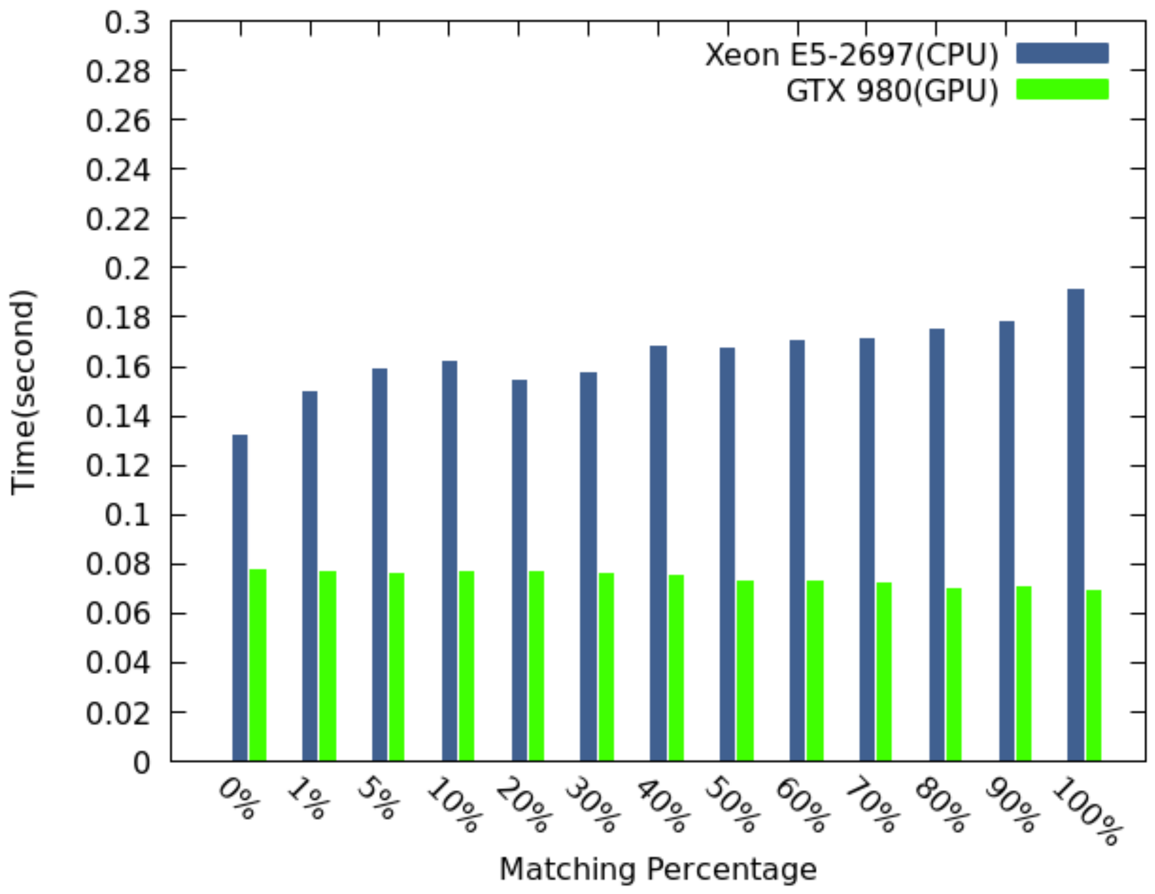

Figure 4. GPU performance comparison for Java's Aho-Corasick

Figure 4 shows that our implementation is outperforming the original Java's Aho-Corasick in all the evaluation experiments. We run each experiment 100 times, changing the matching pattern percentage each time, sorted the filtering time and took the median. The total patterns we use for filtering is 60k. The performance gained with our implementation starts from 52.02% up to 93.84% depending on the matching patterns. Nevertheless, a realistic scenario on the pattern matching is around 10%-30% each time.

[7] See https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/ or https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

3 Data Streams processed by Apama

In the overall architecture picture of I-BiDaaS, see Figure 5, it is shown which kind of information should be processed by Apama for the I-BiDaaS use cases. The data mainly comes from the Data Ingestion component, consisting of Universal Messaging as Message Broker, shown to the left in light blue. This path is used by IBM's TDF[8] which creates high-quality test data while minimizing the risks of using sensitive production data. More details are also given in D2.5. Also the data from the data providers is injected via this method, more details will be given in section 3.2 below.

Another source of information is from the Hecuba Tools[9], provided by the partner BSC, which,

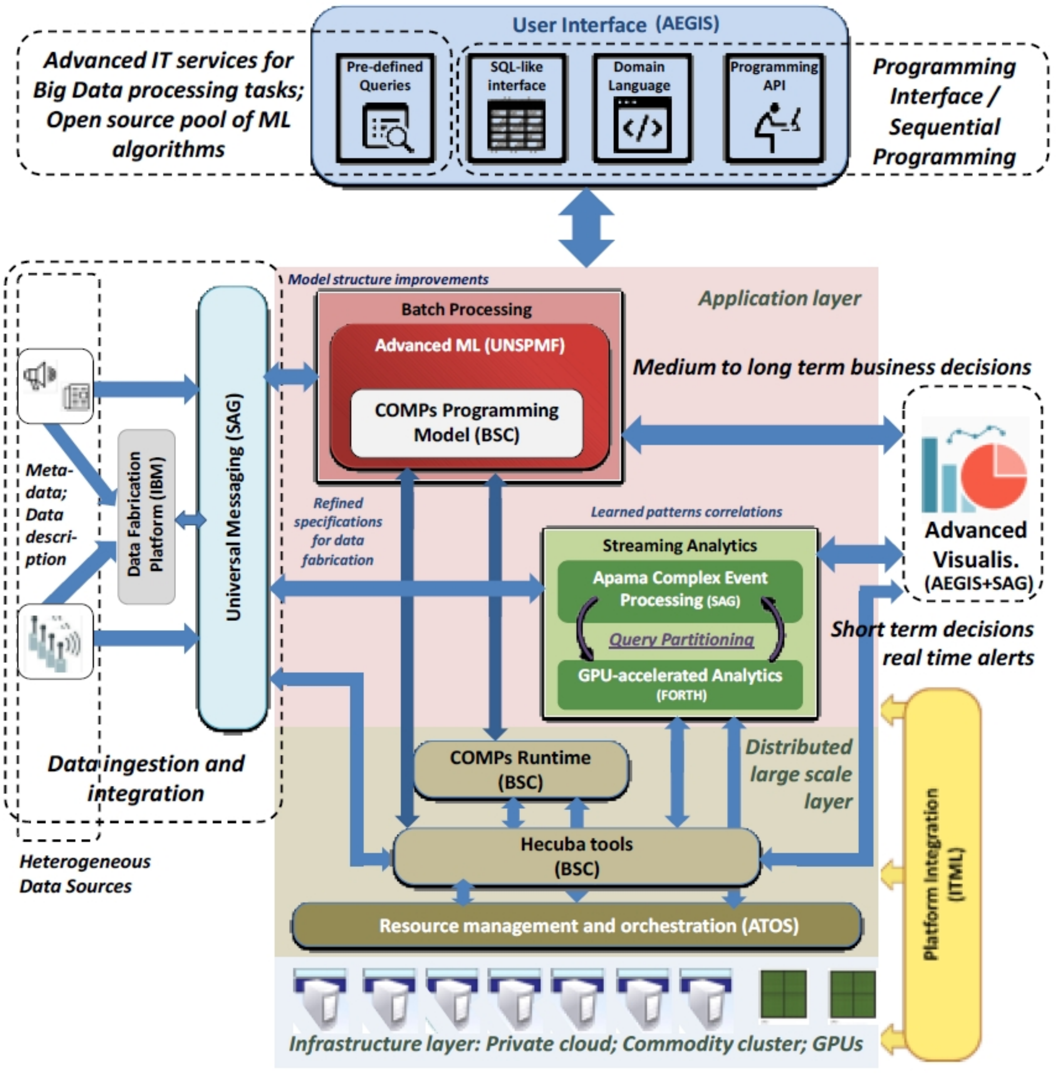
User Interface (AEGIS)
Advanced IT services for Big Data processing tasks; Open source pool of ML algorithms
Pre-defined Queries
SQL-like interface
Domain Language
Programming API
Programming Interface / Sequential Programming
Model structure improvements
Batch Processing
Advanced ML (UNSPMF)
COMPs Programming Model (BSC)
Application layer
Medium to long term business decisions
Meta-data; Data descri-ption
Data Fabrication Platform (IBM)
Universal Messaging (SAG)
Refined specifications for data fabrication
Learned patterns correlations
Streaming Analytics
Apama Complex Event Processing (SAG)
Query Partitioning
GPU-accelerated Analytics (FORTH)
Advanced Visualis. (AEGIS+SAG)
Short term decisions real time alerts
Data ingestion and integration
COMPs Runtime (BSC)
Distributed large scale layer
Hecuba tools (BSC)
Heterogeneous Data Sources
Platform Integration (ITML)
Resource management and orchestration (ATOS)
Infrastructure layer: Private cloud; Commodity cluster; GPUs

Figure 5. I-BiDaaS overall architecture.

in our case, is a Cassandra database[10]. Even though Apama provides many adapters out-of-the-box, for Cassandra, we had to write our own using the documented interfaces. In addition, one

[8] See https://www.ibm.com/il-en/marketplace/infosphere-optim-test-data-fabrication

[9] See https://github.com/bsc-dd/hecuba

[10] See http://cassandra.apache.org/

could use an open-source JDBC adapter for Cassandra[11] since Apama supports JDBC connections. So far, we have tested it only with a single Cassandra table, however, it seems that this structure is sufficient for the I-BiDaaS use cases.

In Figure 5 shown in green is the query partitioning which is done by exchanging information via an in-memory database with GPU-accelerated analytics provided by FORTH. This already was described in detail in section 2.2.

3.1 Input from IBM's TDF

For I-BiDaaS, IBM has extended its Test Data Fabrication software to write the fabricated data in addition to a file or database also to an MQTT message broker. The newly fabricated data are sent in JSON format following configurations options, i.e., broker, port, topic, quality of service, authentication method, secure communications, etc.

The JSON messaged is structured as tabular data, which is essentially a JSON array of arrays where the first array contains the columns names and the remaining arrays contain the table rows.

This allows to test streaming use cases with fabricated data. More details are given in D4.1 and D2.5.

3.2 Input from the data providers

Here we give a few examples how data enters the platform and is processed by Apama. More details on the uses cases are given in deliverables D5.2 and D5.4.

3.2.1 CRF

Data provided by CRF, for the two use-cases, are described as follows:

For the "Production process of aluminium Die-casting" use case, see Figure 6:

- Data about major process parameters and the quality level of the engines' crankcases that have been processed in the production line;
- Data about process temperatures, detected at different points of the crankcase and retrieved by mean of the usage of thermal cameras during the aluminium die casting process.

[11] See https://dbschema.com/cassandra-jdbc-driver.html, it is based on the DataStax Java Driver, see https://docs.datastax.com/en/developer/java-driver/. One could also use Apache Calcite, see https://calcite.apache.org/docs/cassandra_adapter.html.

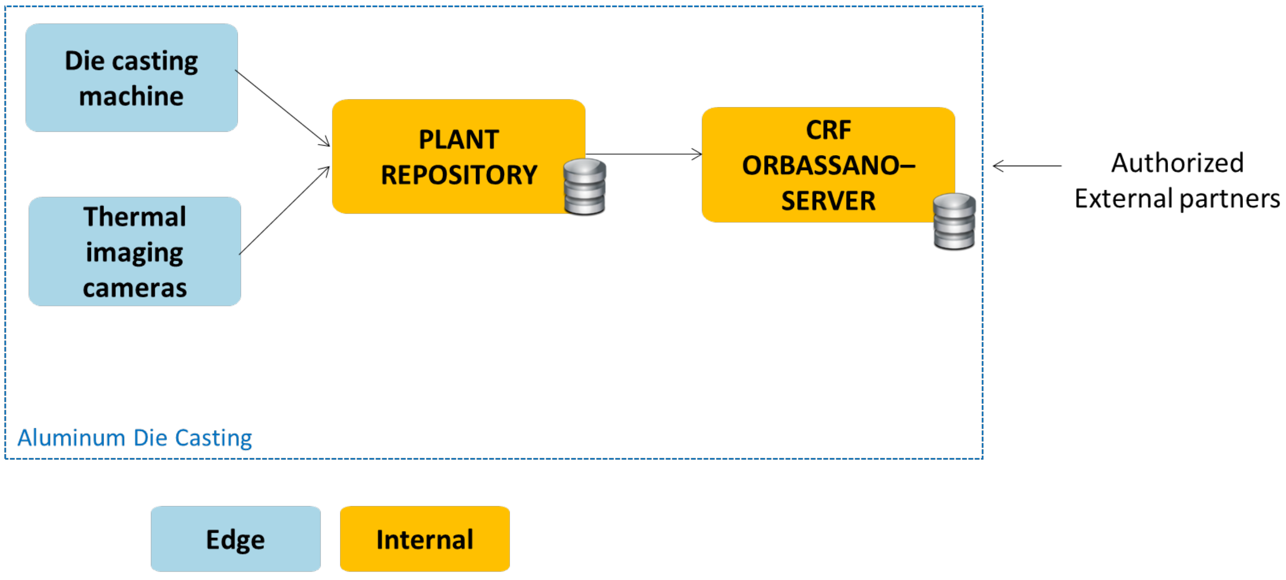

Figure 6. Picture of the architecture for the 1st CRF use-case

For the "Maintenance and monitoring of production assets" use case, see Figure 7:

- Data about major process parameters retrieved thanks to sensors installed on the welding cell;
- Data about the production sequence, available from the Manufacturing Enterprise System.

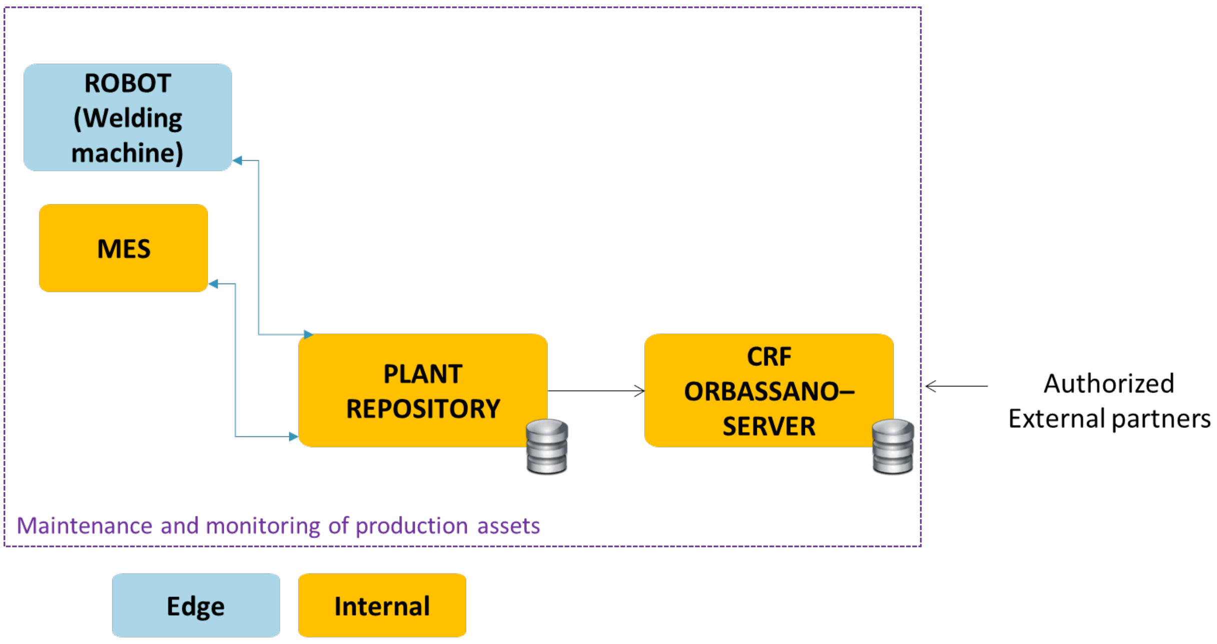
ROBOT (Welding machine)
MES
PLANT REPOSITORY
CRF ORBASSANO–SERVER
Authorized External partners
Maintenance and monitoring of production assets
Edge
Internal

Figure 7. Picture of the architecture for the 2nd CRF use-case

In both use-cases, data is collected in a plant repository, from which they are sent to CRF internal server, located in Orbassano (near Torino). This server is accessible by external partners by using web apps and logging-in with proper credentials.

In case the data is confidential (i.e. if they give information about process quality), an anonymization process is executed, by mean of algorithms which are already implemented inside the CRF internal server.

For the I-BiDaaS use case, the data was extracted from the CRF server and used to train a Random Forest model from a Python program. Once the model was trained, sample data from the CRF server was sent to UM, picked up by Apama to be passed on to the pre-trained model for classification. This simulates a real-time event processing where the machine data is analysed before it is stored in the CRF server.

3.2.2 CAIXA

For the first CAIXA use case, Analysis of relationships through IP address, individual historical connection information from users is analysed. If two or more people seem to be related to each other, e.g. working from the same IP address, the pairs are put in a Cassandra database via the I-BiDaaS batch layer. Once this is done, each transaction in a new connection is analysed if it is between users that are related or not. If the users are related, the transaction can be approved more easily. For the analysis, the connection information is sent to UM, grabbed from Apama and compared to the entries in the Cassandra database which was loaded into memory during start-up. This illustrates that Apama can combine information from different sources, here UM and Cassandra database.

3.2.3 TID

Currently, the TID use cases do not employ Apama for real-time analysis. However, the 1) “Accurate Location Prediction with High Traffic and Visibility” and 2) “Optimization of Placement of Telecommunication Equipment” use cases show promising potential for real-time analysis.

The corresponding dataset, TID_Mobility_data, can scale to over 4TB per and contains so-called cell network events which are picked up by the antennas that are closer to the mobile phone, thus providing an approximate location of the device. Every transaction of a mobile phone generates one of those events. A transaction can be, for instance, placing or receiving a call, sending or receiving an SMS, asking for a specific URL in your mobile phone browser, or sending a text message or a data transaction from/to any mobile phone app. There are also some synchronization events like, for instance, turning your mobile phone on or off, or when switching between location area networks (relatively big geographical areas comprising several cell towers).

This data comes in tabular form. There are three main tables: one related to the events, and two complementing those, related to the users that generate those events and the antennas that deliver the service and collect those events. Among these, we keep only the information, which is strictly related to mobile phone users' mobility, but log much more. Each event contains the start/end timestamp, the start/end sector ID, the country, and other event data (such as the device type, etc.), the antenna latitude/longitude, the azimuth and beam width, other antenna data (such as the frequency at which the antenna is emitting, or the height at which is located), the user ID, and other customer data (such as contract information from the user).

To this end, one would want to understand how users travel around the city creating traffic congestions in network, where and when they will appear next, and predict when new events will cause movements at scale. Hence, a high-value problem that arises is predicting timely places with high traffic and congestion events in order to optimise resource distribution. In addition, one would want to determine areas of low bandwidth that can be affected by many connections and propose counter-measures as fast as possible to improve the user experience and satisfaction. In other words, we are interested in avoiding the deployment of new antennas and use existing infrastructure instead. To do so, we have to implement methods for predicting with accuracy which antennas will become the next “hot spots”. By analysing streaming data, we can improve the routing and placement of the telecommunication equipment that is already in place or arrange accordingly the new equipment obtained, as well as optimize the network operations by providing caches and identifying optimal antenna locations.

The procedure will be similar to what is described for the CAIXA use case. First, historical data is analysed, and then real-time data is analysed to see if similar situations already occurred and propose appropriate solutions.

3.3 Input from other sources

Even though UM is mostly used as data ingestion component in the I-BiDaaS use cases, Apama supports a variety of other event sources. We will name a few and refer the reader to the Apama documentation[12] for a more comprehensive overview

1. The Apama Database Connector (ADBC) is an adapter that uses the Apama Integration Adapter Framework (IAF) and connects to standard ODBC and JDBC data sources as well as to Apama Sim data sources. With the ADBC adapter, Apama applications can store and retrieve data in standard database formats as well as read data from Apama Sim files. Data can be retrieved using the ADBCHelper API or the ADBC Event API to execute general database queries or retrieved for playback purposes using the Apama Data Player.
2. Apama provides a connectivity plug-in, the Kafka transport, which can be used to communicate with the Kafka distributed streaming platform[13]. Kafka messages can be transformed to and from Apama events by listening for and sending events to Kafka channels.
3. The File adapter uses the Apama Integration Adapter Framework (IAF) to read information from text files and write information to text files by means of Apama events. This lets you read files line-by-line from external applications or write formatted data as required by external applications.

[12] See http://www.apamacommunity.com/docs/

[13] See http://kafka.apache.org/

4 Conclusions

This deliverable described the Distributed Event-Processing Engine of the I-BiDaaS platform. It is based on SAG's Apama software. In the I-BiDaaS use cases, the input for the event-processing engine mostly comes from a message broker. We described the different input sources and formats in detail.

The most part of the document was devoted to describing how different Apama instances can work together to share information and distribute the workload. The most suitable solution uses an in-memory component for a fast data exchange.

To gain additional performance, the in-memory component can also be used to offload compute-intense tasks to GPU-accelerated boards. We showed how it is done and what the benefits are.

Edge processing is mostly used if a lot of data is needed to evaluate a specific situation quickly. Since data transfer rates in networks are much slower than in-memory processing rates the pattern detection needs to be done close to where the data is collected. So far, none of the I-BiDaaS use cases has such a requirement, however, the CRF "Production process of aluminium Die-casting" use case can easily be extended to become an Edge Processing case if the pre-trained Python model will run in the production site and not in the data centre. This needs to be evaluated by the data providers.