Enhanced fraud detection as a service supporting merchant-specific runtime customization

Davy Preuveneers, Bavo Goosens and Wouter Joosen
imec-DistriNet-KU Leuven, Leuven, Belgium
{davy.preuveneers, wouter.joosen}@cs.kuleuven.be,
bavo.goosens@student.kuleuven.be

ABSTRACT

We present a customizable, yet scalable data processing architecture for payment processors that aim to offer fraud detection as a service to e-commerce merchants. Due to the increasing complexity of payment solutions and the continuous evolution of fraud patterns, rule-based detection of fraudulent payments is no longer adequate. Our solution is implemented as a κ-architecture for streaming big data, augmented with semantic web techniques to enable constrained customization for merchants. Our evaluation shows that our data processing architecture meets the stringent real-time processing limits of payment transactions, while offering runtime customization for multiple merchants.

CCS Concepts

•**Applied computing** → **Electronic commerce;**

Keywords

adaptive systems; scalability; fraud detection.

1. INTRODUCTION

Fraud detection is an integral part of online payment services to protect merchants against financial loss caused by malicious users. The most widely used approach to detect suspicious spending behaviour relies on fraud rules because they can be (1) easily implemented, (2) efficiently evaluated, and (3) interpreted by the payment processor, the merchant and the customer to explain why a transaction was rejected. When a transaction enters the system, a set of fraud rules is evaluated that each contribute to a weighted global fraud score which results in a final verdict. A merchant can adapt these fraud rules to capture his specific needs by customizing thresholds or black-/whitelists (e.g. acceptable country codes of a credit card, billing or shipping address). However, merchants often feel uncomfortable to implement strict fraud rules as they must account for seasonality variations (e.g. mid-season sale discounts or Christmas shopping period), or mistakenly assume that what works for others will also work for them. Furthermore, fraudsters are moving to increasingly more complex fraud schemes. This poses a problem for payment processors as they see their classical rule-based fraud detection systems fail to capture the dynamic nature of payments and all the peculiarities associated with fraud attacks. This motivates the need for a new approach that maintains the efficiency and intuitivity of fraud rules, but with the ability to consider merchant-specific spending patterns. However, the customization by the merchant must be constrained by the payment processor to ensure valid configurations.

Section 2 revisits the application domain and discusses key requirements. Section 3 details the design of the data structures, whereas section 4 compares two high-level architectural patterns that we evaluated to merge the processing components into a scalable system. A proof-of-concept was built to validate the design. Section 5 will succinctly summarise the experiments and their results. Finally section 6 will concisely conclude this paper.

SAC'17, April 3-7, 2017, Marrakesh, Morocco

http://dx.doi.org/10.1145/3019612.3019886

2. CHALLENGES AND REQUIREMENTS

This section captures the key challenges the payment processor is faced with to enhance fraud detection:

1. **Classify the transactions more accurately**: This includes both lowering false positives and potentially missed fraud cases (i.e. false negatives).
2. **Improve the customizability for merchants**: Give them control over their data and eventually the entire fraud detection interaction.
3. **Improve the customizability for the payment processor**: Support for adding new payment options, detection methods or fraud rules.
4. **Improve the scalability of the system**: As the customer base of the payment processor grows (new merchants or more customers per merchant) the real-time processing capability needs to grow with it.

Processing payments involves two important steps executed in parallel: (1) authorising the payment and (2) executing

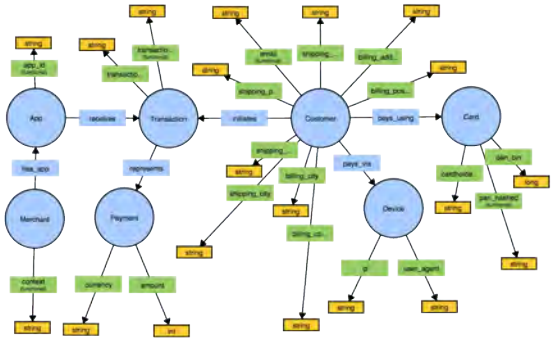


Figure 1: A payment data model described using OWL.

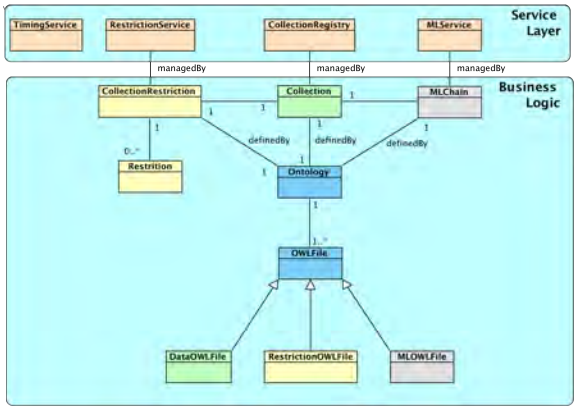


Figure 2: An overview of all the data structures.

the fraud detection. As such, it is important that the fraud detection returns before the authorisation is completed, i.e. in less than 20 msec. It is clear that when payment transactions arrive at a high velocity, the scalability of the system will be put to the test.

3. DATA MODEL DESIGN

Our design relies on ontologies expressed in the web ontology language (OWL) to semantically represent all data concepts, and enable consistency checking through logic reasoning (see Fig. 1). Such models are similar to EER diagrams in that they also represent entities, classes and subclasses. Previous research [4] has proven that transformations from ER to OWL are possible. Relations between these abstractions are modelled using object properties, and attributes can be added using data properties. Ontologies are interpreted by the system using reasoners to derive new knowledge, merge models, and much more. Ontologies are used extensively and essentially form the backbone of the constrained customization flexibility of our system.

An overview of all the different abstractions and their relations is given in Figure 2. The *service layer* will handle incoming requests and create and manage all the *business logic*. The system encapsulates all the different components within their own respective models. A merchant first creates an ontology which contains all the concepts he wants to use for fraud detection, and submits it as a valid OWL file. The definitions included in the *data model* will be reflected in the *restriction model* and *machine learning model*. The system creates a new in-memory `Collection`. This `Collection` contains an `Ontology` object built from the lineage of files submitted by a merchant. These files are encapsulated by the `OWLFile` objects. The ontologies enable the merchant to iteratively and dynamically adjust the data model as the `Ontology` is kept in-memory.

When defining new or altering existing data models, the merchant will supply the base model using an API. Each `Collection` will however have a `CollectionRestriction` that is initialised by the system and serves as an API. To model the restrictions, the system offers a general restriction model (see Fig. 3). It contains all the restrictions at a conceptual level the payment processor wants to offer to his merchants. The payment processor can implement new restrictions and enable all the merchants to use them. Thanks to the underlying `Ontology` such an update to existing restriction models can be made at runtime.

The system will create restrictable individuals when the data model is updated. As such, changes in the data model influence the expressiveness of the restriction model. When a merchant wants to enforce some restriction he uses the API to request a restriction model for a collection. The system will return an OWL file containing the restriction model containing the data model individuals. The merchant will then be able to add restriction individuals using OWL-editors. Once he is satisfied with the set of restrictions, the file is sent to the system which will initialise the restrictions. Upon completion of this process, the incoming data belonging to the specified `Collection` will have the restrictions described in the `CollectionRestriction` checked. Setting up new machine learning pipelines has a very similar usage pattern and as such will not be detailed here.

4. ARCHITECTURE

This section will compare two architectural patterns to deploy a scalable yet performant data processing system.

4.1 λ-architecture

The lambda (λ) architecture [2] uses an immutable master dataset which handles the intake of new data. The data is then sent to two subsystems which each handle it in a different fashion. The first subsystem is the *batch layer*. It uses batch systems such as Hadoop or Spark to execute the necessary operations over very large datasets. The results are sent to a *serving layer* upon completion. Periodically the operations are restarted to ingest new data into the results. Due to the relatively long start up and processing times this layer will introduce some delay. To fill this gap a *speed layer* is used. This layer quickly ingests the data and returns its results to the *serving layer*. This is accomplished by using high-end streaming systems such as Storm [5]. The *serving layer* will then merge the results from the *batch and speed layers* to get a real-time result over a large dataset. In the context of fraud detection the speed layer could handle the incoming screening while the batch layer would be responsible for re-evaluation of the fraud rules on historic data or the machine learning pipelines. The main disadvan-

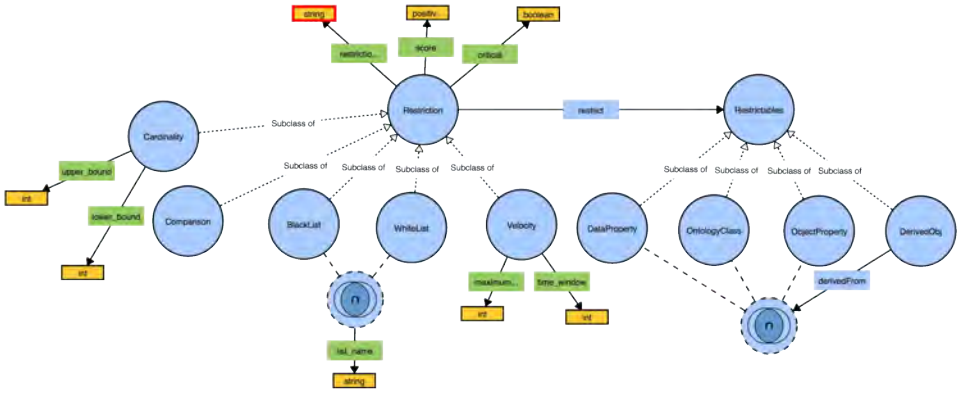


Figure 3: The structure of a general restriction model.

tage to this pattern is the separation between the layers, as each has its own processing characteristics resulting in different implementations of the same algorithms. This makes it more difficult to include new functionality and maintain the overview of the overall system. However, when using Spark and Spark Streaming, the API similarity of RDDs and D-streams [6] can make this pattern a very strong contender.

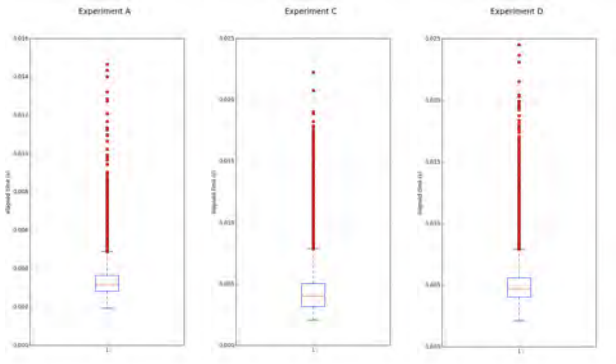
4.2 κ-architecture

The kappa (κ) architecture [3] was proposed as a response to the limitations of the λ-architecture. This pattern aims to simplify the interaction by using only one type of processing: streaming. It argues that streaming systems have matured to a point where they are able to offer most functionality, so they are a viable option to the general MapReduce [1] based batch processing. The overall idea is simple: use a streaming system all the time, and re-evaluate when changes are necessary. To achieve scalability, the architecture increases the parallelism of the streaming system. The results are stored in a way where each consecutive run is traceable. When applied to the case of payment processing, this approach would use a streaming system to set up the fraud rule checks and spawn new instances when more capacity is needed. Furthermore, re-evaluation of restrictions is built-in and machine learning based anomaly classification is handled in the same way. Due to the good fit, we adopted this pattern for a scalable architecture deployment.

5. EXPERIMENTAL EVALUATION

We built a prototype using MongoDB for storage and the Spring Boot framework to implement the business logic. Additionally, the Apache Jena and Weka library were used to incorporate the ontologies and machine learning.

We executed several experiments to evaluate the design and quantify the adaptation impact on the fraud rule checker performance. All experiments ran on commodity hardware (a Macbook Pro with a 2.2 GHz Intel Core i7 processor and 16GB of memory). The set-up was largely the same across all experiments. 4 threads were launched and started sending 10000 fraud requests. This was repeated 3 times. A single run creates approximately 153 MB of data.



(a) Experiment A (b) Experiment C (c) Experiment D

Figure 4: Response times for experiments A, C and D.

5.1 Experiments

Experiment A serves as a baseline for the storage and inferencing, as there were no restrictions on the incoming transaction data.

Experiment B will investigate the impact of data model changes on the ingestion as the incoming data is unchecked. The rate at which the changes are requested is increased by a ten fold across each run. At the highest rate changes are made every 10ms.

Experiment C covers the main functionality of the system: checking the restrictions over the incoming data. For every incoming transactions 48 restrictions had to be checked.

Experiment D combines both experiment B and C. Every 100ms a data model change is requested while the incoming data still has 48 restrictions to be checked to reach a verdict. This experiment effectively investigates the impact of the data model changes on the running restriction checks.

5.2 Results

The result for experiment A is visualised in Fig. 4a. As expected the inferencing and storage of incoming requests happens fast. The overall goal was to answer a fraud screening

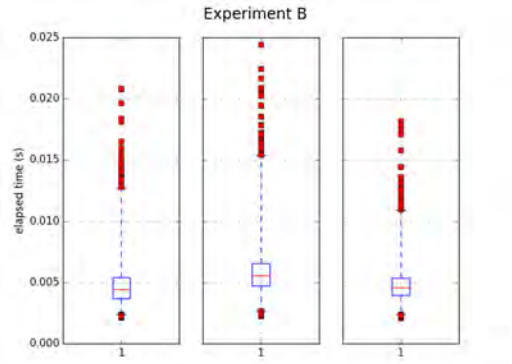


Figure 5: Three boxplots visualizing each of the runs (1s, 0.1s and 0.01s intervals) for experiment B.

request within a 20ms window. The processing times have a very skewed distribution with the bulk of the transactions completing within a 2 to 4 ms window. The results of experiment B were visualised separately per run since the rate was increased across each run. The overall processing time increases which causes the boxplot to shift upwards and some transactions take more than 20 ms to complete. Upon further inspection, we found them to be outliers due to other running processes or contention in the storage system. A peculiar shift developed when we increased the rate to 10ms between data model updates. One would expect the upward shift visible in the transition from the leftmost to the center boxplots in Fig. 4 to continue when comparing the center and rightmost plots. This is however not the case and the performance appears to improve. As it so happened, the storage system encountered issues due to the increasing size of the data models. This made it throw an exception which created less competition for incoming transactions causing the declining response times. The load generated in this experiment was however far heavier than one can expect in a real-world situation, and this issue was deemed manageable.

The results of experiment C show an upward shift of a skewed distribution when comparing it to experiment A. The distribution is almost identical to the one for experiment A but shifted 2ms to the right, i.e. an expected result when adding more restrictions. There were 2 of the 120000 transactions which returned a verdict outside of the 20 ms time window. These transactions are classified as outliers. They are possibly caused by either a large amount of small writes causing contention or triggered by interference from other running processes. The final experiment aimed to investigate the impact of data model updates on the restriction checks. Fig. 4c summarises the results in one boxplot. The most noticeable difference between experiment C and D is the larger variance introduced in the outliers. After analysing the individual runs, we discovered that the outliers which took more than 20 ms were all from the same run, subject to interference from other processes.

6. CONCLUSION

This paper presented an adaptable yet scalable data processing architecture that aims to improve fraud detection for e-commerce applications. The support for customization is driven by using semantic web techniques. The experiments showed that on a single node, the overhead to gain this customizability is well within the real-time performance requirements. Overall the goals and improvements detailed in section 2 were largely accomplished: The customizability for both the merchant and payment processor was met by using ontologies to represent all the models and their interaction. Furthermore, we adopted a pattern to achieve scalability in the form of the κ-architecture in order to achieve the goal to answer fraud screening requests within 20 ms seconds under heavy loads. A single node rule checking implementation proved that this was possible even while data model updates were being executed.

As future work, we plan a more in-depth analysis on the trade-off between the performance impact and classification accuracy of different machine learning techniques, as this was beyond the scope of this work. Using even more meta-information (currently the restriction and ML models can be seen as meta-information) in the system would offer the necessary foundations to have the system dynamically adjust itself to changing demands.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven, and CAPRADS. CAPRADS is a project realized in collaboration with imec. Project partners are JForce, Luciad and Televic Education, with project support from VLAIO (Flanders Innovation and Entrepreneurship).

7. REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[2] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *Proceedings of the 2015 IEEE International Conference on Big Data*, pages 2785–2792. IEEE Computer Society, 2015.

[3] J. Kreps. Questioning the Lambda Architecture, 2014.

[4] I. Myroshnichenko and M. C. Murphy. Mapping er schemas to owl ontologies. In *IEEE International Conference on Semantic Computing ICSC'09*, pages 324–329. IEEE, 2009.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.