U.P.B. Sci. Bull., Series C, Vol. 84, Iss. 3, 2022 ISSN 2286-3540

SOFTWARE ARCHITECTURE OF DATA AGGREGATION SYSTEM

Claudia Ifrim[1], Bogdan Costel Mocanu[2], Florin Pop[3,4], Valentin Cristea[5], Manolis Wallace[6]

The complexity of real world problems today are pushing the limits of software architectures. In this paper we present a novel software architecture that enables aggregation of data from multiple sources while giving granular access and privacy to data via access policies. The proposed software architecture is evaluated in two real life use cases. The first use case creates a large data set of scientific articles by aggregating three data sources. Citation lists of scientific papers are analyzed in order to quantify and understand the impact of the work, tracing the "footprints" of the author in science. The second use case analyze an Automatic Identification System (AIS) large data set, proposing a data reduction technique specific to AIS data. Granular access is provided to third party consumers. Through these we see that the proposed approach is able to easily enriching data knowledge and can provide granular data access to third party consumers.

Keywords: software architecture, access policy, large data sets

1. Introduction

Technology is strongly related to progress and is the prime ingredient for progress nowadays. Progress, in its turn, forces technology to adapt itself, to become better, more useful and respond in real time, to surpass its original design and provide new outcomes. Today we have an impressive explosion of available information that must be analyzed and efficiently used to take rational and concrete decisions for the near or long term future.

[1]PhD Student, Automatic Control and Computers Faculty, University "Politehnica" of Bucharest, Romania, e-mail: claudia.ifrim@upb.ro

[2]Lecturer, Automatic Control and Computers Faculty, University "Politehnica" of Bucharest, Romania, e-mail: bogdan_costel.mocanu@upb.ro

[3]Professor, Automatic Control and Computers Faculty, University "Politehnica" of Bucharest, Romania, e-mail: florin.pop@upb.ro, / [4]Scientific Researcher, 1st degree, National Institute for Research & Development in Informatics - ICI Bucharest, e-mail: florin.pop@ici.ro

[5]Professor, Automatic Control and Computers Faculty, University "Politehnica" of Bucharest, Romania, e-mail: valentin.cristea@upb.ro

[6]Professor, Knowledge and Uncertainty Research Laboratory, University of the Peloponnese, Greece, e-mail: wallace@uop.gr

The main issue we address is the modern application requirements, the traditional computing paradigms and monolithic application architecture are becoming inefficient. With this in mind, this paper propose an improved software architecture model that can be applied in software development of domain specific applications used for analytics on large data sets.

The rest of the paper is structured as follows. In Section 2 we present the main challenges faced in building software applications based on domain specific large data sets that are trying to solve real world complex problems. In Section 3 we present a novel software architecture that offers an easy way to enrich the data knowledge and granular access to it. Section 4 outlines our case studies used to evaluate the efficiency of the proposed architecture, cases presented in our published papers "Scientific Footprints in Digital Libraries" [7] and "Data reduction techniques applied on automatic identification system data" [5]. The final Section presents our conclusions and future work.

2. Challenges

We outline the challenges that we are now facing in complex real world problems and we focus on only three main directions: how frameworks, technologies and architectures affect software engineers design decisions, how large data sets are managed, pre-processed, analyzed, stored and shared, how complex real world problems request new software architecture models.

2.1. Challenges on Software Architecture and Software Design

A software architecture is an important property of a software system. Therefore, a solution for the challenges mentioned above is to separate as much as possible the requirements of an application from the implementation of the domain. This idea is used in industry products, especially those that use the Hexagonal Architecture (also known as Ports and Adapters architecture), interface discovery [3] or layered architecture and bounded context [2].

A challenge nowadays could be considered also the "modern monolith". The monolithic architectural style can still be seen even in the modern applications. This approach can work well even for a large application until you will need: different programming languages, independent deployability and independent scalability. Another important aspect that need to be taken into consideration is the complexity of the real world domains.

In "Domain-Driven Design" [2] we find a set of design patterns that solves the division of large models. The solution for easily maintain a large model is to split it into different "Bounded Contexts" and create a "Context Map" that will trace the relationships between prior defined bounded contexts.

Researchers are aware that modeling large complex systems to solve real world domain problems is, indeed, an open challenge [1]. Nowadays, microservices are promoted as a possible solution to reduce the complexity and the coupling between the systems and specific technologies.

2.2. Challenges on Large Data Sets

Exploring large data sets has become a necessity today. It is a difficult problem and information visualization techniques can help in solving it and can also bring improvements in data analysis [8]. The rapid fast evolution of Internet related technologies have provided the opportunity to collect huge data volumes from various domains [9]. Exploring this opportunity it's easy to reach a state of information overload. The collection and analysis of large data sets is a widely spread practice in the last decades in various domains like economics, marketing, health or sociology. In the last year this process was accelerated in security, molecular biology and health as we can see in [10], [4].

Even if it is not easy to define a large data set, we can outline the three most important characteristics: requires digital storage; humans do not have the cognitive capacity to process it; sufficiently large to encourage data mining. We have to keep in mind that large data sets are targeted to a specific domain or topic. They are analyzed by data mining techniques, not by hypothesis testing. The wide variety of the data stakeholders, available large data sets specific to a given domain and different way of data usage pushed us to think to a more efficient way to share it. Every system that provide access to it's data has authorization policies governing the data access and operations allowed.

2.3. Data Privacy and Data Ethics

First of all, we must define "Data Privacy" and "Data Ethics". Data privacy is meant to protect the security issues that can be generated by exposing personal identifiable information. Data ethics related to proper handling of data. Data must be preserved and promoted in a responsibly manner. Most common ethical problem example is related to the healthcare system.

Our work is meant to contribute to the privacy of the data stored in a system based on our proposed architecture. We propose a solution that will manage the authorization and attributes accessed by third parties based on policies. This component is described in Section 3.

3. Software Architecture for Specialized Large Data Sets

In this section, we propose a hexagonal architecture model that can successfully be used for domain specific applications that are using large data sets that are continuously enriched and are also exposing the data to third parties in a secure manner, using a privacy policy data access component. For sure in the following years our attention should be focused on software architectures. We have Big Data, Data Lakes, various devices from IoT that are generating huge amount of data. We developed various tools for data analysis and now a high demand for data scientists has arisen in industry and research field. A real problem is in fact the way software products are created and how the data is used and shared. The need of real time processing and analysis was clearly observed in 2020 due to COVID-19 pandemics.

The most common used software architecture nowadays is the Layered Architecture. It's known from the university, tutorials and even best practices or frameworks that we use in developing new applications.

By 2005, thanks to Entity-Boundary-Interactor (EBI) Architecture and Domain-Driven Design, we managed to realize the importance of the inner layers in a system. In 2005, Alistair Cockburn outlined the idea that the top and bottom layers, on the other hand, were simply entry and exit points.

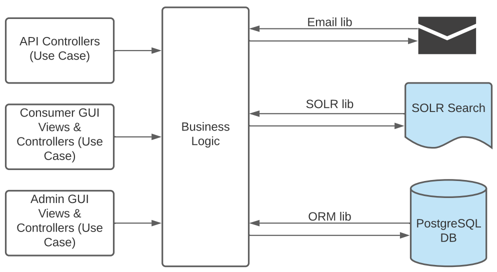
API Controllers (Use Case)
Consumer GUI Views & Controllers (Use Case)
Admin GUI Views & Controllers (Use Case)
Business Logic
Email lib
SOLR lib
SOLR Search
ORM lib
PostgreSQL DB

Fig. 1. A layered architecture perspective.

Before trying to find a solution for a problem, the case study/business context must be well framed and the large data set that will be analyzed must be well understood (if already exists) or should be properly created to fit our needs. If the problem is not well framed, all the efforts in analytics can lead to misleading or incorrect results. This step will help us in designing also the domain application. In order to consider that, we have a well defined case study problem we must consider previous findings, we need to have a clear definition of our scope, we should know the targeted audience and we must establish a set of measurement metrics for our results. This process flow must be followed prior implementation of the system as follows: framing which includes problem identification, review of previous findings and case study definition (scope); solving which includes data collection and understanding, data analysis and preparation, model building and components definition and external tools; presenting and reporting which includes actionable insights, communication, processing, access policies definition and share.

Therefore, having the case study framed and considering the idea promoted by Alistair Cockburn, we can change the classic layering diagram and represent in Figure 1 the two sides of the system as left and right, instead of the well known top and bottom. We can now easily observe that each side can have multiple entry and exit ports (API, ORM, Search engine, etc.).

Using this architecture we bring the application in the center of the system, allowing us to keep the application isolated from the implementation details like changing technologies or tools. This architecture empower us to mock the ports, making the development process of a prototype easy and faster. Using the ports and adapters architecture we can easily change the data repository or the search engine used by the system.

Another relevant example of the benefits of using this architecture is the need of having in our application a web user interface and an API that need to have the same new functionality in place. Using ports and adapters we implement the desired functionality in an application service method that can be designed as a use case. Our service implements an interface specifying the inputs, outputs and methods. Both the user web interface and the API have controllers (representing the Adapter). The controllers use the above defined interface that is injected with the service implementation.

3.1. Architecture Requirements

In order to address different usage scenarios, different types of queries, serve various data consumers and assure data privacy, we define an architecture for applications that relies on large data sets has a number of requirements: technology agnostic, access attributes based on privacy policy authorization, easily plug-in multiple data sources and data consumers, easily query the data, specify data access policy, support analysis of historical data.

3.2. Architecture Description

The main idea of this architecture is to think that our application is the central component of the entire system. All the input will reach the application and all the output will leave the application through ports. In this way our application is isolated from delivery mechanisms, technologies and external tools. Our application should not be aware of who is sending input or who is receiving its output. This type of isolation protect the application against the evolution of technology and business requirements and help the product to be easily maintained and tested. It applies the same principles that Robert C. Martin later described in more general terms in his book "Clean Architecture"[1].

The Hexagonal architecture clearly defines three components in the system: the user interface, the application core and the infrastructure. The application core is the main and most important part of our system. Here we define the model of our domain and the implementation of the specific use cases. The ecosystem in which the application core lives is represented by all tools needed by our application. Based on the case study you define, on the resources needed, the tools can be databases, search engines, repositories, etc.

3.2.1. *Connecting Application Core and Tools.* The components that are connecting the application core with the external tools are called adaptors and are implementing the code that allows our business logic to interact with specific external tools like databases or search engines. Each adapter is specific for a single external tool. The adapters are categorized as Primary or Secondary

[1]https://www.biblio.com/9780134494166

Adapters. An entry point in the application core is named port. For each port of the application core we need to define an adaptor. Each port is in fact an interfaces and we can consider it a specification that must be used by the adaptor and by the external tool to know how can communicate. The primary adapters are in fact controllers, they wrap around a specific port on the left side of the architecture (see Figure 2). Those adapters trigger events that are requesting use cases from the application core and in fact service method calls. The secondary adapters are implementations of interfaces (represented by ports) and are injected in the application core when the ports are called (see Figure 2). A specific characteristic is that adapters are linked to a specific port (interface) that communicates to a specific external tool.

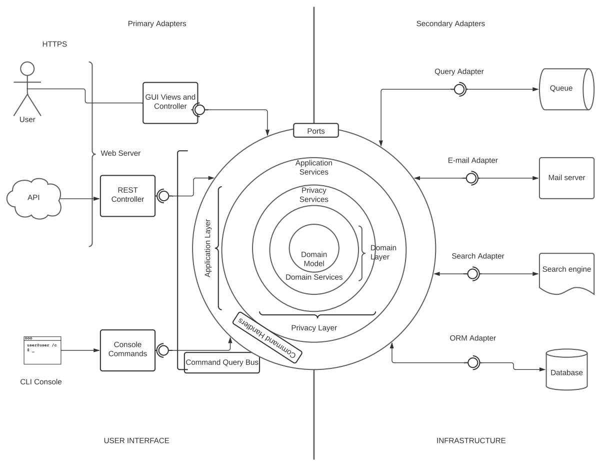

Fig. 2. Diagram describing the inner layers

3.**2**.2. *Application core components and layers.* The layers and components are meant to bring order inside of the hexagon, into the code base and business logic. We follow the inversion of control principle, so all the dependencies direction is to the interior of the hexagon. Considering the case study that we want to develop and the use cases defined to fit the needs of our case study, the architecture of our application can have several components and layers. In the following subsections we describe all the layers and provide guidance on component development. The minimal required layers that must be present in any application are: Application Layer, Privacy Layer and Domain Layer. For complex applications you can also add extra components.

3.**2**.3. *Application Layer.* Considering our application defined scope we must defined several use cases that cover all the application needs. Those use cases are owned by the Application Layer, described in the Figure 2, is the first layer of our architecture and is provided by Domain-Driven Design. Our use cases are triggered by events emitted by any User Interface that is defined in our application. The most common example nowadays is a web application that has multiple interfaces and APIs depending of the user type.

This layer contains all the application services and their interfaces characteristic to our case study. Here we also must define the Handlers for Command or Queries if the need to use a Command or a Query Bus is imposed by our case study. It also contains the ports and adaptors interfaces that are used to communicate with the external tools. The Services and Handlers owned by Application Layer contain the business logic of the use cases prior defined.

Command Handlers can be implemented using two approaches: can contain directly the business logic that address the use case or can call services that already contains the business logic of the requested use case.

Having the services and command handlers that are implementing the use cases, the Application Layer contains also the triggering mechanism of Application Events which are the output of a use case.

3.**2**.4. *Domain Layer.* Going deeper into the center of the hexagon we reach the Domain Layer (see Figure 2). Our Domain Layer contains the data entities and the rules to manipulate it and it's not related to the business processes from Application Layer that triggers Domains logic. Common use cases in the development process of the application have the need of involving different entities in the domain logic. The solution is to split the Domain Layer in two sub-layers: Domain Services and Domain Models.

Domain Services sub-layer has the role to execute some business logic on the entities that are received. A Domain Service must be part of the Domain Layer. A Domain Service, being in the Domain Layer can use other services available in the Domain Layer and the objects form the Domain Model.

Data Model sub-layer is represented in the center of our architecture. It contains the domain objects that are modeling the domain of our case study. Those objects are represented by entities, value object or any other object defined in the Domain Model. Domain Model contains also the Domain Events that are triggered when entities are changed.

3.**2**.5. *Privacy Layer.* Because data privacy is highly important topic and we also encourage to enforce the ownership over newly created data sets as a best practice, we introduce a Privacy Layer in our architecture. This is an optional layer that can be included in the architecture if the use case is based on sensitive data or we have multiple data consumers (APIs) that should not have access to all the information in entities.

Most of the applications are based on Access Control Lists (ACL) or Roles and Groups authorization. The authorization schema based on such concepts has major problems such as inefficient granularity management, data leakage, high cost maintenance or impossible consumers audit. The Domain Layer is the place where Data Model lives, so the best approach is to surround the Domain Layer by authorization policies as described in Figure 2. In this way we protect the access to the domain services and domain model. For instance, when an API sends a request to our system, the authorization policies defined for this specific API should be evaluated and respond with a Data Transfer Object only in case of successful access authorization of the API on the entity.

We promote the use of access policies based on attributes and actions per resource because they offer a fine grained control. Using policies we can even provide access to only some of the attributes of an entity. This pattern should persist for all the APIs that are querying data in our system as described in Figure 3. For a clear understanding of the "Policy" concept we need to define it. We consider a policy to be a set of rules that can be evaluated and is responding to the question "Can A execute an action B on the resource C?". A policy enforce an allow or deny response for an action over a resource.

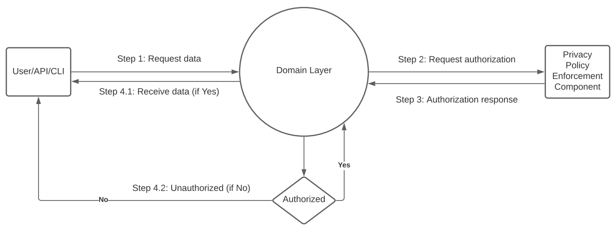
User/API/CLI
Step 1: Request data
Step 4.1: Receive data (if Yes)
Domain Layer
Step 2: Request authorization
Step 3: Authorization response
Privacy Policy Enforcement Component
Yes
Step 4.2: Unauthorized (if No)
No
Authorized

Fig. 3. Data access flow for consumers.

The Privacy Layer is responsible of data access for each API that our application will expose. It is also responsible to provide a quick answer of type allow (if the consumer is authorized to do the requested action on the resource), deny (if the consumer is not authorize do do the action) or error (if the system fail) to the consumer. Most of the systems are based on Access Control Lists (ACL) or roles and groups authorization and they have a direct connection with the central data store of the system. This is considered a major security issue that we want to avoid and this is the main reason why our Privacy Layer is communicating with it's own database. This database will contain information of attributes managed directly by the domain. Here we save information related to access policies and attributes.

For authorization, we need to save in our database information related to the user or the system that owns the attribute, the id of the object that contains the attribute, attribute type, value, creation and update timestamps.

To reduce the response time of the Privacy Layer we build also a persistent cache that contains only the needed information to evaluate the authorization, this means that we have only the attributes enforced by the policy defined in our system. In this way we quickly respond with a deny if the request contains a attribute that is not in the cache. Another important aspect is how we keep our database consistent. If the domain adds, remove attributes or update attributes values, it is also responsible to push the updates in the database used by our Privacy Layer. The cache is updated base on the time to live defined or can be rebuild by the system. After evaluating the authorization of the call, if it is evaluated to allow, meaning that the request is authorized to perform the action on the resource, the Privacy Layer requests the allowed attributes of the object from the domain and sends the response to the consumer. If the authorization of the call is evaluated as a deny, the response is directly sent to the consumer.

Using this mechanism we also avoid data leakage, because the consumer is receiving either a deny, either a response containing only the allowed attributes from the requested objects. If we don't need restricted access to our data we can bypass the access evaluation step in each implemented component. Either way, we have the Privacy Layer database updated and it can be used to enforce data ownership or for system audit. We always know what consumers we have.

Furthermore, we consider that this authorization system brings more granularity in the system and data leakage is avoided comparing with the existing solutions (Access Control Lists (ACL) or roles and groups authorization). Also, the system development, administration, maintenance and auditing process is improved. The initial step of gathering the attributes implies more time and effort. Some extra development time must be budgeted to complete the work on pushing the attributes from the domain central data store to Privacy Layer database and to cover the synchronization process.

3.**2**.6. *Components.* A component will cross-cut the layers of our application as it is described in the Figure 4. The components will be defined in our application base on the use cases that were prior defined. Following this model, the components will benefit from low coupling and high cohesion.

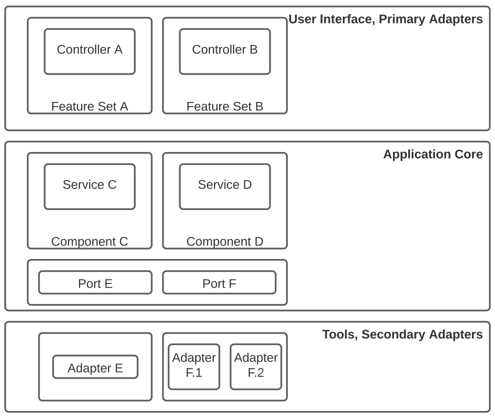

Fig. 4. Component of the application.

Examples of components are provided in Section 4, and each prototype mentioned has various components that cover the needs of our defined use cases. Using Dependency Injection and Dependency Inversion we manage to keep the depending class unaware of the concrete classes that will be used.

Also, having completely decoupled components, they have no knowledge of any other component. We need to let components to interact, but in the same time we want them decoupled. To achieve this we recommend the use of a discovery service. Another important aspect is that a component can query and use any data in read only mode, but can change only the data owned.

The proposed architecture offer us the option to have components with their own data storage or repository. In this case, we add more complexity to the storage level in order to maintain the integrity of the data. For each data storage of a component we need to have on one hand a data set that is owned by the component - the single source of truth and on the other hand a data set that is not owned by the component and can be used only in read-only mode. In Figure 5 we describe the application flow in more detail, considering also the interaction between layers, their services, repositories and data storage.

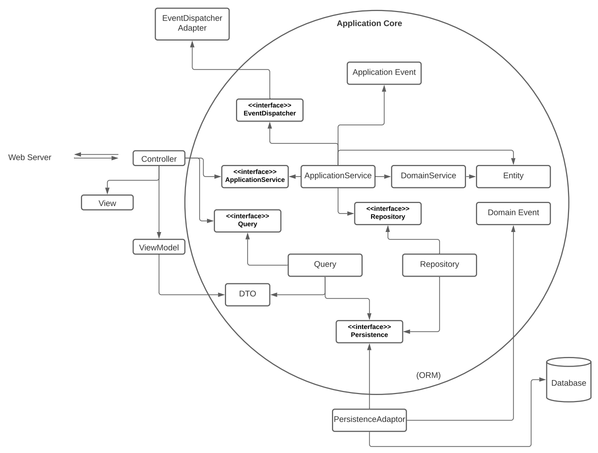

Fig. 5. Application flow - considering the interactions.

4. **Evaluation**

We proposed this architecture and guidelines in order to promote the idea of clean architecture and a code base that is high cohesive and loosely coupled. We need those for a maintainable project, that can be changed or extended easy and fast, without affecting the entire code base. We also consider that this architecture can help researchers and experts to easily build a large data set for a specific domain through collaborative work. The small development effort needed to add a new data source in the application using the propose architecture should enforce it's use in academia and research.

Moreover, in our case study presented in [7], we analyze citation lists of scientific papers in order to quantify and understand it's impact, by tracing the "footprints" that authors have left in science, i.e. the specific areas in which they have made an impact. In Table 1 we present as an example the

Table 1. **Anagnostopoulos - Scientific impact.**

Computer science	Applied sciences	Social sciences	Life sciences	Natural & physical sciences	Humanities
67%	27%	2%	2%	2%	0%

Table 2. **Initial records vs. reduced records.**

Initial records no.	Unique MMSI	Speed/Heading difference	Final records no.
752 552	458	less than 0.15 / less than 3.5	204 338
752 552	458	less than 0.2 / less than 1	202 248

scientific impact of Ioannis Anagnostopoulos, a member of the Department of Computer Science and Biomedical Informatics, at the University of Thessaly.

In order to create the large data set that was used for our analysis we developed a prototype based on the architecture and guidelines described above, adding also a Processing Component that normalized and corrected the errors found in the data stored in our central data store.

Our system, the Footprint Analyzer, was used to successfully identify the most prominent works and authors for each scientific field, regardless of whether their own research is limited to or even focused on the specific field.

In the second case study we analyze an Automatic Identification System (AIS) large data set and we proposed a data reduction technique that can be applied on AIS data without losing important information and reduce it to a manageable size that can be further used for analysis or visualization applications (see Table 2).

In order to evaluate our architecture, we developed a prototype that received data from an AIS data stream, decoded and stored the raw data. With dedicated components, such as Data Reduction component, Privacy Layer, near-Real-time Analytics Component, Time Series Analytics Component and Hybrid Analytics Component, we applied the data reduction and analytics algorithms. The reduced data is used to create visualizations for traffic monitoring. The reduction technique is described in [5] and [6].

5. Conclusions

A new direction in software architecture and software design, software engineering processes and guidelines must be carried out while engineering those applications. The main contribution of this paper is the proposed software architecture model that can help engineers and researchers to easily aggregate data from multiple data sources (streams, structured or unstructured data) with the aim to create a specialized data set on specific domain. Key component of the architecture is the Privacy Layer that protects access to data. The architecture is described in Section 3 and is evaluated in real world use cases.

As future work, we plan to develop an open-source software framework based on the proposed architecture. A strong documentation containing guidelines and development best practices will be compiled based on our experience gained from the case studies used to evaluate our architecture proposal.

Acknowledgements

The research presented in this paper was supported by project FARGO: Federated leARninG for human moBility (PN-III-P4-ID-PCE-2020-2204), the project CloudPrecis (SMIS code 2014+ 124812), project ODIN112 (PN-III-P2-2.1-SOL-2021-2-0223) and by the University Politehnica of Bucharest through the PubArt program. Also, this paper supported by REALS - Real-time Adaptive Scheduling, UPB Postdoctoral Research Project (2021-2023). We would also like to thank the reviewers for their valuable comments and insights.

References

[1] Betty HC Cheng and Joanne M Atlee. “Research directions in requirements engineering”. In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 285–303.

[2] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[3] Steve Freeman and Nat Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[4] Alex Graudenzi et al. “Mutational signatures and heterogeneous host response revealed via large-scale characterization of SARS-CoV-2 genomic diversity”. In: *Iscience* 24.2 (2021), p. 102116.

[5] Claudia Ifrim et al. “Data reduction techniques applied on automatic identification system data”. In: *Semanitic Keyword-based Search on Structured Data Sources*. Springer. 2017, pp. 14–19.

[6] Claudia Ifrim et al. “Methods and Techniques for Automatic Identification System Data Reduction”. In: *Big Data Platforms and Applications*. Springer, 2021, pp. 253–269.

[7] Claudia Ifrim et al. “Scientific Footprints in Digital Libraries”. In: *Transactions on Computational Collective Intelligence XXVI*. Springer, 2017, pp. 91–118.

[8] Mike Sips et al. “Selecting good views of high-dimensional data using class consistency”. In: *Computer Graphics Forum*. Vol. 28. 3. Wiley Online Library. 2009, pp. 831–838.

[9] Wil Van Der Aalst. “Data science in action”. In: *Process mining*. Springer, 2016, pp. 3–23.

[10] Penghui Zhang et al. “CrawlPhish: Large-scale Analysis of Client-side Cloaking Techniques in Phishing”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2021.