Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XXVI Ciclo – 2013

Timely Processing of Big Data in Collaborative Large-Scale Distributed Systems

Leonardo Aniello

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XXVI Ciclo - 2013

Leonardo Aniello

Timely Processing of Big Data in Collaborative Large-Scale Distributed Systems

Thesis Committee

Prof. Roberto Baldoni (Advisor)
Prof. Francesco Quaglia

Reviewers

Dr. Opher Etzion
Prof. Jean Bacon

AUTHOR'S ADDRESS:
Leonardo Aniello
Department of Computer, Control, and Management Engineering "Antonio Ruberti"
"La Sapienza" University of Rome
Via Ariosto 25, I-00185 Rome, Italy
E-MAIL: aniello@dis.uniroma1.it
www: http://www.dis.uniroma1.it/~dottoratoii/students/leonardo-aniello

To my family and my friends.
There’s no better way to support a person than making him feel loved and highly valued.

Abstract

Today's Big Data phenomenon, characterized by huge volumes of data produced at very high rates by heterogeneous and geographically dispersed sources, is fostering the employment of large-scale distributed systems in order to leverage parallelism, fault tolerance and locality awareness with the aim of delivering suitable performances. Among the several areas where Big Data is gaining increasing significance, the protection of Critical Infrastructure is one of the most strategic since it impacts on the stability and safety of entire countries. Intrusion detection mechanisms can benefit a lot from novel Big Data technologies because these allow to exploit much more information in order to sharpen the accuracy of threats discovery.

A key aspect for increasing even more the amount of data at disposal for detection purposes is the collaboration (meant as information sharing) among distinct actors that share the common goal of maximizing the chances to recognize malicious activities earlier. Indeed, if an agreement can be found to share their data, they all have the possibility to definitely improve their cyber defenses. The abstraction of Semantic Room (SR) allows interested parties to form trusted and contractually regulated federations, the Semantic Rooms, for the sake of secure information sharing and processing. Another crucial point for the effectiveness of cyber protection mechanisms is the timeliness of the detection, because the sooner a threat is identified, the faster proper countermeasures can be put in place so as to confine any damage.

Within this context, the contributions reported in this thesis are threefold

- As a case study to show how collaboration can enhance the efficacy of security tools, we developed a novel algorithm for the detection of stealthy port scans, named R-SYN (Ranked SYN port scan detection). We implemented it in three distinct technologies, all of them integrated within an SR-compliant architecture that allows for collaboration through information sharing: (i) in a centralized Complex Event Processing (CEP) engine (Esper), (ii) in a framework for distributed event processing (Storm) and (iii) in Agilis, a novel platform for batch-oriented processing which leverages the Hadoop framework and a RAM-based storage for fast data access. Regardless of the employed technology, all the evaluations have shown that increasing the number of participants (that is, increasing the amount of input data at disposal), allows to improve the detection accuracy. The experiments made clear that a distributed approach allows for lower detection latency and for keeping up with higher input throughput, compared with a centralized one.

- Distributing the computation over a set of physical nodes introduces the issue of improving the way available resources are assigned to the elaboration tasks to execute, with the aim of minimizing the time the computation takes to complete. We investigated this aspect in Storm by developing two distinct scheduling algorithms, both aimed at decreasing the average elaboration time of the single input event by decreasing the inter-node traffic. Experimental evaluations showed that these two algorithms can improve the performance up to 30%.

- Computations in online processing platforms (like Esper and Storm) are run continuously, and the need of refining running computations or adding new computations, together with the need to cope with the variability of the input, requires the possibility to adapt the resource allocation at runtime, which entails a set of additional problems. Among them, the most relevant concern how to cope with incoming data and processing state while the topology is being reconfigured, and the issue of temporary reduced performance. At this aim, we also explored the alternative approach of running the computation periodically on batches of input data: although it involves a performance penalty on the elaboration latency, it allows to eliminate the great complexity of dynamic reconfigurations. We chose Hadoop as batch-oriented processing framework and we developed some strategies specific for dealing with computations based on time windows, which are very likely to be used for pattern recognition purposes, like in the case of intrusion detection. Our evaluations provided a comparison of these strategies and made evident the kind of performance that this approach can provide.

Acknowledgements

The first person I want to thank is my advisor, Prof. Roberto Baldoni. He believed in my potential and supported me since my Master Thesis through all my PhD, including this thesis itself, and working with him allowed me to continuously grow.

I want to give special thanks to Dr. Silvia Bonomi. We attended the Bachelor of Science together, then took diverse paths, and finally met again at the Department. She has been, and still is always available for supporting me and providing valuable advises. Another special thank goes to Dr. Giorgia Lodi. She has been the first person I collaborated with when I started my Master Thesis, and she played a key role in motivating me to enroll. Another person that contributed relevantly to my growth and surely deserves special thanks is Dr. Leonardo Querzoni, who has been always providing interesting ideas, professional guidance and real supporting.

I also wish to thank Prof. Francesco Quaglia, Dr. Opher Etzion and Prof. Jean Bacon for their valuable and effective help in refining and validating my work. A unique thank goes to my friends Maria Luce and Federica, who helped me with the English for writing this thesis. Last but not least, I want to acknowledge any other Professor, Postdoc, Staff, PhD and Master student I collaborated with during these years, because everyone of them gave me the opportunity to learn something new.

Leonardo

Contents

Introduction **1**

I Collaborative Environments **7**

1 The Semantic Room Abstraction **9**
1.1 Information Sharing . . . 9
1.2 Related Work . . . 11
1.3 The SR Abstraction . . . 11
1.3.1 The SR Gateway . . . 13
1.3.2 Enforcing Confidentiality Requirements . . . 14
1.4 Fair Information Sharing . . . 15
1.5 An SR for Fraud Monitoring . . . 16
1.5.1 Analysis and Design . . . 17
1.5.2 Implementation . . . 19

2 Collaborative Port Scan Detection **23**
2.1 Related Work . . . 23
2.2 Inter-domain stealthy SYN port scan . . . 24
2.3 Rank-based SYN port scan detection algorithm . . . 24
2.3.1 Half open connections . . . 25
2.3.2 Horizontal and vertical port scans . . . 25
2.3.3 Entropy-based failed connections . . . 26
2.3.4 Ranking . . . 26
2.4 Esper Intrusion Detection Semantic Room . . . 26
2.4.1 Esper . . . 27
2.4.2 R-SYN Algorithm in Esper . . . 27
2.5 Storm Intrusion Detection Semantic Room . . . 28
2.5.1 Storm . . . 29
2.5.2 R-SYN Algorithm in Storm . . . 31
2.6 Agilis Intrusion Detection Semantic Room . . . 32
2.6.1 Hadoop and the MapReduce paradigm . . . 33
2.6.2 Agilis . . . 33
2.6.3 R-SYN Algorithm in Agilis . . . 36
2.7 Experimental Evaluation . . . 37
2.7.1 R-SYN in Esper . . . 38
2.7.2 R-SYN in Esper vs R-SYN in Storm . . . 40
2.7.3 R-SYN in Esper vs R-SYN in Agilis . . . 42

II Enhancing Technologies for Collaborative Environments **45**

3 Adaptive Scheduling in Storm **47**
3.1 Related Work . . . 48
3.2 Default and Custom Scheduler . . . 49
3.2.1 State Management . . . 49
3.3 Adaptive Scheduling . . . 50
3.3.1 Topology-based Scheduling . . . 51
3.3.2 Traffic-based Scheduling . . . 52
3.4 Experimental Evaluations . . . 55
3.4.1 Reference Topology . . . 55
3.4.2 Grand Challenge Topology . . . 61

4 Windowed Computations in Hadoop **63**
4.1 Related Work . . . 64
4.2 Batch processing for Time Window Based Computations . . . 65
4.2.1 Time Window Based Computations . . . 65
4.2.2 Batch processing . . . 66
4.2.3 Computation Model . . . 68
4.2.4 Input data organization strategies . . . 69
4.3 Implementation with Hadoop . . . 70
4.3.1 Hadoop Distributed File System . . . 70
4.3.2 Performance Metrics Optimization . . . 71
4.4 Experimental Evaluation . . . 74

Conclusion **81**

Introduction

The Big Data Phenomenon

In the last few years we are witnessing a huge growth in information production. IBM claims that "every day, we create 2.5 quintillion bytes of data - so much that 90% of the data in the world today has been created in the last two years alone" [71]. Domo, a business intelligence company, has recently reported some figures [21] that give a perspective on the sheer amount of data injected on the internet every minute, and on its heterogeneity as well: 3125 photos are added on Flickr, 34722 likes are expressed on Facebook, more than 100000 tweets are done on Twitter, etc. This apparently unrelenting growth is a consequence of several factors such as the pervasiveness of social networks, the smartphone market success, the shift toward an "Internet of things" and the consequent widespread deployment of sensor networks. This phenomenon, known by the popular name of **Big Data**, is expected to bring a strong growth in economy with a direct impact on available job positions; Gartner says that the business behind Big Data will globally create 4.4 million IT jobs by 2015 [128].

Big Data applications are typically characterized by the three *V*s [71] : large *volumes* (up to exabytes) produced at a high *velocity* (intense data streams that must be analyzed in quasi real time) with extreme *variety* (mix of structured and unstructured data coming from diverse sources). Classic data mining and analysis solutions showed quickly their limits when they have to face such loads. Indeed, storing data with these characteristics into relational databases or data warehouses has become prohibitive since it would require cleansing and transforming very heterogeneous pieces of raw data (because of its *variety*), that is produced at extremely high rates (because of its *velocity*), so that it can fit into some predefined schema. On the other hand, storage hardware is getting cheaper and this would ease the storage of larger *volumes* of data, indeed this is a key driver for developing new storage software that can cope with the *variety* and *velocity* of such data. Furthermore, querying such data at rest by employing traditional techniques, like SQL, becomes more and more challenging and expensive (because of its *volume*). Big Data applications, therefore, imposed a paradigm shift in the area of data management that brought us several novel approaches to the problem, represented mostly by NoSQL databases, for what concerns storage technologies, and batch data analysis tools together with advanced Complex Event Processing (CEP) engines for what regards querying mechanisms.

An additional consequence of the huge *volumes* of Big Data is the push to increase the amount of physical resources employed to store and query data, which in turn gives a boost to the development of highly **distributed systems** able to scale in and out as the need arises. The proliferation of data producers and consumers all over the world really contributes both to the *variety* of data and to the *velocity* it is generated and then retrieved. It becomes convenient to deploy these systems closer to their clients and users by distributing them geographically (multi data center deployment), so as to decrease physical distances and consequently improve the quality of provided services. The growing **large-scale** nature of stores and query engines entails additional challenges and requirements.

There are several contexts where Big Data trend is gaining particular relevance. In general, any area where large amounts of data can be gathered and analyzed in order to extract meaningful insights is suitable for employing Big Data techniques. In economics and finance, the availability of market-related mass media

and social media content and the emergence of novel mining techniques allow to advance research relating to smart market and money [60]. The start-up Inrix Inc. developed a system that leverages “millions of cellphone and GPS signals, makes sense of the jumble of information about car speeds, weather conditions and sports schedules, and translates the data into maps of traffic along 2600 miles of the state’s roads” [127].

Another valuable source for Big Data appliances is represented by the logs produced by software applications: companies like Facebook and Twitter generate petabytes of logs per day [71]. Log analysis can provide useful means for users profiling, activity monitoring and also security [153]. Indeed, by looking for anomalous or known malicious patterns, possible threats can be detected and even prevented. Subsequent analysis of attack detections can be leveraged to improve security defenses in order to avoid that further attacks of the same type would be successful. In addition, the earlier the detection is, the more effective the mitigation actions can be, which makes clear that the **processing** has to be as much **timely** as possible.

Big Data and Cyber Security

The importance of cyber security is growing very quickly, on one hand because the overall value managed by the means of cyber technologies is rocketing, on the other hand because the current rapid evolution of information systems has the inherent side effect of introducing new security breaches. Current trend toward Big Data is pushing for developing larger-scale storages where increasingly huger data volumes can be stored and queried at ever higher rates and, at the same time, providing zero-downtime and seemingly limitless scalability. State-of-the-art technologies for achieving such goals include highly scalable distributed NoSQL storages [56]. These are usually deployed in data centers, whose design and evolution are considerably driven by the ever changing requirements of the applications they have to host (e.g., Bigtable [58], Megastore [42], Dynamo [69]). The increasing spread and the continuous advancement of key technologies, such as virtualization and cloud computing, together with the overwhelming proliferation of mobile devices, are currently speeding up the ongoing transformation of data centers. This consequently leads to the growth of their complexity, which in turn gives rise to new security breaches as a side effect [66]. Such state of alert is also confirmed by the perceptions of involved people that are more and more worried about the actual effectiveness of current security mechanisms and about the thriving strength and complexity of cyber attacks [143]. Another relevant point concerns a trend that is typical of emerging technologies, which consists in favoring the development and enhancing of further features rather than focusing on security aspects of already existing functionalities. Such trend makes the situation even worse for the security guarantees of new cloud-based software because security issues are even more likely to be overlooked.

One of the most relevant issue of cyber security is the protection of Critical Infrastructures (CIs) [44]. It wasn’t by chance that the President of the USA Barack Obama has proclaimed December 2012 as Critical Infrastructure Protection and Resilience Month [122]. Even though the precise definition of CI varies from country to country, we can consider a CI as a structure of assets, systems and networks, either physical or virtual, such that its unavailability or destruction would have a debilitating effect on security, economy, health or safety, or any combination thereof [123]. For example, in the USA, the Presidential Policy Directive 21 (PPD-21) on Critical Infrastructure Security and Resilience identifies 16 CI sectors: chemical, commercial facilities, communications, critical manufacturing, dams, defense industrial base, emergency services, energy, financial services, food and agriculture, government facilities, healthcare and public health, information technology, nuclear plants, transportation system and water system [124]. Distinct CIs can be interconnected in intricate ways, so that a successful attack to one CI can produce a domino effect and damage further CIs. The interdependencies among CIs can be either technical (i.e. the communications service requires the energy service) or based on geographical proximity as well [131]. As CIs grow in complexity, figuring out their interdependencies in order to identify proper countermeasures becomes prohibitive. The situation is made even worse by the increasing spread of the Internet within CIs, indeed more and more parts of them are reachable through Internet and therefore become potential targets for cyber attacks.

Today's cyber security warnings, together with the real risk of cyber attacks to CIs, make the cyber protection of CIs a priority to be addressed promptly. The great complexity of the problem, together with the huge amount of data to be analyzed and of resources to provide and manage, suggest to undertake a roadmap which employs Big Data technologies as the main building blocks. In the specific, the timely elaboration of observed cyber activities can be a very effective mean to spot forthcoming, ongoing or already happened attacks in order to prevent, mitigate or trace them. Within this context, the possibility to elaborate larger volumes of data even becomes an opportunity to leverage, because it can enable the correlation of information provided by distinct actors having the common goal of improving their cyber defenses. After all, the more the input data it has at disposal, the higher the accuracy of the computation is. Assembling **Collaborative** Environments where distinct participants share their data (in this case, their logs about observed cyber activities) and resources (computational, storage and network) can be a convenient practice for reducing the provisioning cost (required resources are provided by all the participants) and at the same time sharpening the effectiveness of defensive mechanisms (thanks to greater quantity of information at disposal). Setting up Collaborative Environments in practice introduces several issues, mainly related to privacy, trustworthiness and sharing fairness enforcement [44].

Thesis Contents

This thesis is organized in two parts

1. the first part delves with collaborative environments by describing strengths and weaknesses of information sharing, by introducing the abstraction of Semantic Room and by presenting a case study concerning collaborative stealthy port scan detection

2. the second part focuses on two of the technologies employed in the first part

   - *Storm: a framework for distributed online event processing.*
     Two schedulers are presented that aim at decreasing the average computation latency by minimizing inter-node traffic.
   - *Hadoop: a framework for distributed batch-oriented processing.*
     A set of strategies is described for carrying out time window based computations on a batch-oriented fashion.

Collaborative Environments

The first part analyzes the potentialities of information sharing, identifies the obstacles that prevent or limit its adoption in real scenarios and propose possible guidelines to overcome them. At this aim, the abstraction of Semantic Room (SR) is introduced, which allows interested parties to form trusted and contractually regulated federations, the Semantic Rooms, for the sake of secure information sharing and processing. The SR abstraction is capable of supporting diverse types of input data ranging from security events detected in real time, to historical information about past attacks and logs. It can be deployed on top of an IP network and, depending on the needs of the individual participants, can be configured to operate in either peer-to-peer or cloud-centric fashion. Each SR has a specific strategic objective to meet (e.g., detection of botnets, of stealthy scans, of Man-In-The-Middle attacks) and has an associated contract specifying the set of rights and obligations for governing the SR membership. Individual SRs can communicate with each other in a producer-consumer fashion resulting in a modular service-oriented architecture. Parties willing to join an SR can be reluctant to share their data because of the lack of proper assurances about how these data are handled and because the risks of leaking sensitive information are not well-understood. State-of-the-art techniques and technologies can be integrated within an SR in order to precisely setup the information to disclose and to

mask the identity of the parties sharing the data. A further issue concerns the level of trust among participant parties, which makes them suspect each other of selfish behavior and take countermeasures that negatively impact on the overall effectiveness of the collaboration itself.

Among the possible employments of collaborative approaches for cyber defense purposes, this thesis delves into intrusion detection. In the specific, a novel algorithm is described for the detection of stealthy port scans, which is a common technique used by attackers as a reconnaissance activity aimed at discovering any vulnerability in the target system. Such algorithm is called R-SYN and is used in an SR to recognize SYN port scans, that are scans for detecting the status of TCP ports without ever completing the initial TCP connection phase. These scan activities are distributed among several target sites for scattering the evidence of the scan over a number of different parties. In this way, such parties in isolation cannot discover the intrusion, since the traffic information at disposal of each single party is not enough to give evidence of the malice nature of the activity. Three distinct implementations are provided: one based on a centralized CEP engine (Esper [11]), one employing a framework for distributed event processing (Storm [18]) able to sustain a wider range of input traffic and, finally, a further implementation based on Agilis [31], a novel platform for batch-oriented processing which leverages the Hadoop [150] framework and a RAM-based storage for fast data access.

Experimental evaluations show that the collaboration is really effective, indeed the accuracy of the detection is demonstrated to improve as more organizations contribute with their own traffic logs. Furthermore, the experiments make clear that a distributed approach allows for lower detection latency and for keeping up with higher input throughput. Both Esper and Storm adopt an approach based on continuous computation, meaning that the elaboration is ceaselessly executed on incoming events. Existing distributed processing platforms that operate with continuous queries don't cope well with dynamic reconfiguration because moving processing tasks at runtime requires the capability to manage stateful tasks and to buffer incoming events, and that's not properly implemented in available products yet. For this reason, the Agilis-based implementation provides an alternative approach that employs batch computations in order to avoid the problems deriving from dynamic reconfiguration: by running the computation periodically, there are enough opportunities to adjust the configuration to properly cope with required needs. As an additional result, the implementation based on Agilis gave evidence that, in Collaborative Environments where participants communicate through the Internet, the available physical bandwidth can easily become a bottleneck. Consequently, a batch approach in which each execution is scheduled so as to move the computation where required data is stored can provide better performance compared to a centralized approach where the computation is at rest while input data get moved accordingly.

Enhancing Technologies for Collaborative Environments

In the second part, two of the technologies employed for the implementation of the R-SYN algorithm are taken into account with the aim of enhancing them in order to obtain improved performance: Storm and Hadoop.

Distributing the computation over a set of physical nodes introduces the issue of improving the way available resources are assigned to the elaboration tasks to execute. What it means exactly improving resource allocation heavily depends on the specific scenario. In this thesis we deal with timely computations, so the main goal is the minimization of the time the computation takes to be completed. We investigated this aspect in Storm [18] by developing two distinct scheduling algorithms, both aimed at decreasing the average elaboration time of the single input event, which implies an overall reduction of the time required to get the outcome of an ongoing computation, like the detection of an intrusion. One algorithm works offline, before the computation is started, and takes into account the links among elaboration tasks in order to produce a schedule such that communicating tasks are likely to be allocated to the same physical machine. The other algorithm works online and monitors at runtime the amount of traffic among elaboration tasks in order to

generate a schedule such that the traffic among physical machines is minimized. Experimental evaluations show that these two algorithms can improve the performance up to 30%.

In Storm, the computation is specified by the means of a network of elaboration tasks that are setup and keep being continuously executed on incoming input streams. The need of refining already installed queries or adding new ones, together with the need to cope with the variability of the input, requires the possibility to adapt the resource allocation at runtime, which entails a set of additional problems. Among them, the most relevant are how to cope with incoming data and processing state while the topology is being reconfigured, and the issue of temporary reduced performance. In this thesis, the alternative approach is explored of running the computation periodically on batches of input data. Even though running the computation at fixed intervals involves a performance penalty for what concerns the elaboration latency, such an approach eliminates the great complexity of dynamic reconfigurations. We chose Hadoop [150] as batch-oriented processing framework and we developed some strategies specific for dealing with computations based on time windows, which are very likely to be used for pattern recognition purposes, like in the case of intrusion detection. Our evaluations provide a comparison of these strategies and make evident the kind of performance that this approach can provide.

Contributions

In the following the main contributions of this thesis are reported.

- the devising of a novel algorithm, R-SYN, for the detection of SYN port scans;
- the implementation of R-SYN in three different technologies (Esper, Storm and Hadoop-based Agilis) with a set of evaluations on detection accuracy and performance which demonstrate
  - the effectiveness of R-SYN algorithm in detecting SYN port scans,
  - the real added value of collaboration for sharpening the accuracy of a computation,
  - the quantitative benefits on sustainable input throughput and computation time derived by distributing the computation,
  - the impact of limited physical bandwidth on computation latency, and how a locality-aware scheduling can address such issue;
- the further improvement of elaboration time on distributed computations in Storm by developing two scheduling algorithms that take care of smartly allocating elaboration tasks on available physical resources;
- the investigation on the efficacy of a batch approach for carrying out computations on input streams coming from distributed sources, with focus on Hadoop elaborations based on time windows

Thesis Organization

Chapter 1 introduces the potentialities of information sharing and Collaborative Environments. It also describes the Semantic Room (SR) abstraction as a mean to enable the setup of such Collaborative Environments and proposes high level strategies for coping with the main obstacles to the widespread adoption of information sharing: confidentiality and fairness. The description of an SR used in a real scenario is reported to provide some indications on the practical issues to address when collaboration has to be implemented.

Chapter 2 describes an SR for intrusion detection, where a novel algorithm for detecting SYN port scans, R-SYN, is integrated so as to highlight the advantages of collaboration to detection accuracy. Three implementations of R-SYN are provided, one using a centralized CEP engine (Esper), one employing instead a

framework for distributed event processing (Storm), and another one based on Agilis, a batch-oriented processing platform where data is stored in a RAM-based store and computation is carried out using the Hadoop framework. Experimental evaluations allow for a comparison between Esper-based and Storm-based implementations, and between Esper-based and Agilis-based.

Chapter 3 delves more into the topic of distributed computation by detailing a way to improve even more the performance in Storm through the development of smart scheduling algorithms that aim at minimizing the traffic among physical nodes so as to get an overall reduction of elaboration time. The actual efficacy of this approach is quantified by the presentation of performance results deriving from experimental evaluations.

Chapter 4 underlines a possible problem in the way frameworks for distributed event processing like Storm carry out the computation. In the specific, they elaborate by the means of queries that are continuously executed on incoming data, and dynamic reconfiguration of the way the computation is allocated to available resources can cause temporary worsening of the performance. A viable way to address such issue is to adopt a batch approach where the computation is started periodically on a batch of input data. This chapter investigates such possibility in the specific case of elaborations based on time windows and presents an evaluation where batch-oriented strategies are implemented in Hadoop.

The contents of this thesis are mainly based on [107, 34, 31, 32, 35, 111, 73, 33].

Part I

Collaborative Environments

Chapter 1

The Semantic Room Abstraction

Organizations must protect their information systems from a variety of threats. Usually they employ isolated defenses such as firewalls, intrusion detection and fraud monitoring systems, without cooperating with the external world. Organizations belonging to the same markets (e.g., financial organizations, telco providers) typically suffer from the same cyber crimes. Sharing and correlating information could help them in early detecting those crimes and mitigating the damages.

This chapter discusses the Semantic Room (SR) abstraction (Section 1.3), which enables the development of collaborative event-based platforms, on top of the Internet, where data from different information systems are shared, in a controlled manner, and correlated to detect and react to security threats (e.g., port scans, distributed denial of service) and frauds. Confidentiality can be a key aspect in scenarios where organizations should disclose sensitive information, which could prevent the collaboration from taking place. The SR abstraction enables the employment of state-of-the-art technologies for addressing confidentiality issues. Before introducing the SR abstraction, the benefits of information sharing are described in details (Section 1.1), so as to provide a grounded foundation for the potential effectiveness of such methodology, and most relevant related work are presented (Section 1.2). A significant issue related to information sharing is the current lack of any mean to enforce some notion of fairness on the relationship between what one shares and what one gains by that sharing: this aspect is investigated in Section 1.4. Finally, in Section 1.5 we present an SR we prototyped for fraud monitoring with the aim to show how confidentiality requirements can be met in real scenarios.

The contents of this chapter are based on [107, 111].

1.1 Information Sharing

Threats such as frauds and cyber attacks can have serious and measurable consequences for any organization: lost revenue, downtime, damage to the reputation, damage to information systems, theft of proprietary data or customer sensitive information [139, 114]. Increasingly complex threats are extremely difficult to detect by single organizations in isolation, since their evidence is deliberately scattered across different organizations and administrative domains. This leads many political and technical contexts to strongly believe that *information sharing* among groups of organizations is the correct answer to timely detect and react to such threats with a consequent damage mitigation [62, 120]. The need for information sharing is particularly important for information systems of Critical Infrastructures (CIs) such as financial, air traffic control, power grid systems that are undergoing profound technological and usage changes. Globalization, new technological trends, increasingly powerful customers, and intensified competition have brought about a shift in the internal organization of the infrastructure of industries, from being confined within the organizational boundaries to a truly global ecosystem characterized by many cross domain interactions and heterogeneous information systems and data.

If we consider the context of cyber security for CIs, the complex proprietary infrastructures where "no hackers knew" are now replaced by an increasing usage of both the Internet as network infrastructure and off-the-shelf hardware and software with documented and well-known vulnerabilities. Although the sophistication of cyber attacks has increased over time, the technical knowledge required to exploit existing vulnerabilities is decreasing [36]. Attacking tools are often fully automated and the technology employed in many attacks is simple to use, inexpensive and widely available. Because of the increased sophistication of computer attack tools, a higher number of actors are capable of launching effective attacks against the IT of CIs. From a sophistication viewpoint, today's attacks are becoming widely distributed in space and time. Distributed in space, as attackers are able to launch any type of virus, trojan, worm in a coordinated fashion involving a large amount of hosts, possibly geographically dispersed and belonging to different organizations and administrative domains. They are also distributed in time, often consisting of a preparation phase spanning over several days or weeks, and involving multiple preparatory steps [95, 148]. In order to cope with such distribution of the attacks, information sharing is highly recommended, perhaps even mandatory.

In the context of fraud monitoring, information sharing is also very helpful to quickly detect and react to such threats. Nowadays, frauds can be so sophisticated and intersected with the cyber space that cooperation among stakeholders and law enforcement agencies is required in order to have an effective, wide and timely global picture of what is going on within the information systems of collaborating parties. From this constantly changing picture, frauds can be identified in the stakeholders information systems, new intelligence operations can be executed and repression measures can be timely identified and deployed by law enforcement agencies. Needless to say, the delay in the detection of fraud activities is an important parameter for mitigating their damages.

As remarked by some security studies [114], information sharing should be preferably carried out by (potentially competing) organizations belonging to the same markets (e.g., financial organizations, telco providers, power grid providers) as they typically suffer from the same vulnerabilities, cyber crimes and frauds. In the context of financial organizations, FS/ISAC in USA [3], Presidio Internet in Italy and OLAF in EU [2] are examples of information sharing bodies for cyber crimes and fraud detection. All of them facilitate the sharing of information pertaining to cyber threats, vulnerabilities, incidents, frauds, and also potential protective measures and best practices. This sharing is usually achieved by sending the information related to the potential threat to both analysts, for reviewing and scoring, and to experts of financial services sector. Once a vulnerability or incident is analyzed, it can be categorized as *Normal*, *Urgent*, or *Crisis*, depending on the risk to the financial services sector. After the analysis, alerts are delivered to participants. Alerts typically contain not only the details of the threat but also information about how to mitigate it. Needless to underline, all this human-based procedures (i) take a period of time to be executed that is usually much larger than the time required to deploy the attack and (ii) can work only on an amount of data manageable by humans.

There is then an urgent need to enhance these bodies with suitable collaborative software platforms that are able to timely correlate large volumes of data coming from multiple information systems, so as to collectively show the evidence of a threat (e.g., attack, fraud, incident, vulnerability) and then categorize it in an automatic way.

A number of research works have focused on the study of collaborative systems for detecting massive large scale security threats [154, 158]. However, organizations can be reluctant to fully adopt such collaborative approach as this may imply sharing potentially sensitive information (e.g., financial transactions, users identities). In these competitive contexts, a controllable platform should be developed, which is capable of meeting the necessary guarantees (e.g., reliability, availability, privacy) for data and resource management. Contracts established among the parties should be set up which specify the guarantees to be provided. Additionally, the platform has to be flexible enough to be easily customized for different contexts, ranging from detection of cyber attacks to fraud monitoring. With these guarantees, it is likely that even distrusting organizations can be motivated in making available their data for collaborative detection of massive threats.

1.2 Related Work

The effectiveness of correlating data coming from several sources (routers) for detecting network anomalies and failures has been suggested and evaluated in [81]. Also the usefulness of collaboration and sharing information for telco operators with respect to discovering specific network attacks has been pointed out in [154, 158]. In these works, it has been clearly highlighted that the main limitation of the collaborative approach concerns the confidentiality requirements. These requirements may be specified by the organizations that share data and can make the collaboration itself hardly possible as the organizations are typically not willing to disclose any private and sensitive information. In our platform, the SR abstraction can be effectively used in order to build a secure and trusted Collaborative Environment for information sharing, which can be enriched with the degree of privacy and anonymity needed by the participants.

The issue of providing a platform for enabling collaboration through resource sharing has been tackled at a lower level within the topic of Virtual Organization in grid computing [75]. The aim of this research front is to enable a set of organizations to share their resources in order to achieve a common goal. The focus is at operating system level [65] and the importance of interoperability is stressed in order to create the conditions to enable collaboration. On the contrary, the SR abstraction works at a higher level and addresses the problem of integrating distinct systems by proposing an architecture where organizations take part to an SR by the mean of a dedicated software component (Section 1.3). Furthermore, the SR abstraction has been tailored to cope with the problems that are likely to arise in the security field, while Virtual Organizations are more general.

In addition, collaborative approaches addressing the specific problem of Intrusion Detection Systems (IDSs) have been proposed in a variety of works [8, 106, 161, 146]. Differently from singleton IDSs, collaborative IDSs significantly improve latency and accuracy of misuse detections by sharing information on attacks among distinct IDSs hosted by participating organizations [160]. The main principle of these approaches is that there exist local IDSs that detect suspect activities by analyzing their own data and disseminate them by using possibly peer-to-peer communications.

This approach however exhibits two main limitations: (i) it requires suspect data to be freely exchanged among the peers; (ii) it does not fully exploit the information seen at every site; (iii) no privacy, security and performance requirements are considered when gathering and processing the data. The first limitation can be very strong due to the data confidentiality requirements that are to be met. The second limitation can affect the detection accuracy as emerges from an assessment of commercial solutions such as distributed Snort-based [6] or Bro-based [96] IDSs. These systems propose the correlation of alerts produced by peripheral sensors. Such alerts are generated using local data, only. The information that is cut away by the peripheral computations could bring them to miss crucial details necessary for gaining the global knowledge required to detect inter-domain malicious behaviors. Our solution addresses precisely this issue through the usage of general-purpose processing platforms that offer great flexibility to the management of the detection logic. For what concerns the third limitation, the SR abstraction is designed so as to be associated with a contract where specific requirements on security, performance and privacy can be defined. The SR aims at processing injected data with the aim of detecting suspect activities through the exploitation of all the data made available by every participating organization.

1.3 The SR Abstraction

The SR abstraction enables the construction of private and trusted Collaborative Environments through which organizations (e.g., financial institutions) can federate for the sake of data aggregation and near real time data correlation. The organizations participating in an SR can exploit it in order to effectively monitor the IT infrastructures and timely detect frauds and threats, and are referred to as the *members* of the SR. An SR is defined by the following three elements:

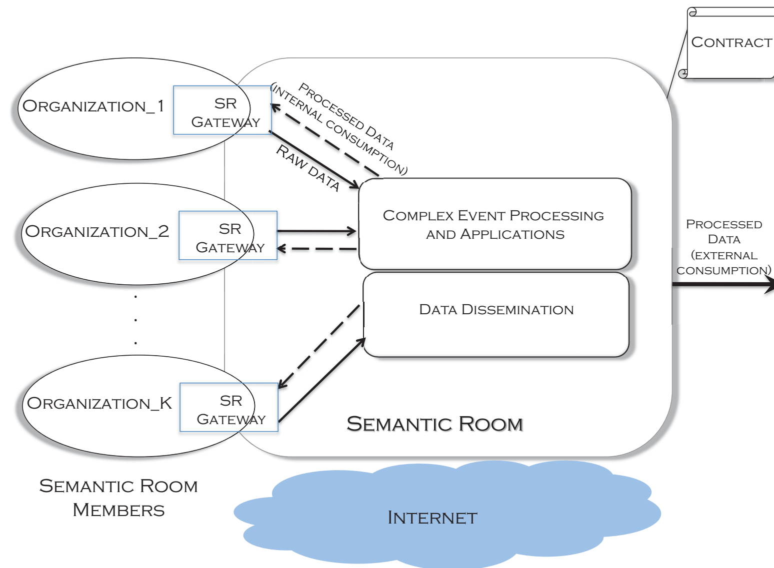


Figure 1.1: High level design of an SR. The core of an SR consists of a layer for carrying out the specific logic of the SR (*Complex Event Processing and Applications*) and a layer in charge of propagating provided/processed data to interested parties (*Data Dissemination*). The *SR Gateway* enables an organization to interface with an SR, and consequently with all the other organizations. Part of the processed data can be destined for external consumption. A *contract* regulates all the duties related to participating in an SR.

- *objective*: each SR has a specific strategic objective to meet. For instance, there can exist SRs created for large-scale stealthy port scans detection, or SRs created for detecting web-based attacks such as SQL injection or cross site scripting;
- *contract*: each SR is regulated by a contract that defines the set of processing and data sharing services provided by the SR along with the data protection, privacy, isolation, trust, security, dependability, and performance requirements. The contract also contains the hardware and software requirements a member has to provision in order to be admitted into the SR.
- *deployments*: The SR abstraction is highly flexible to accommodate the use of different technologies for the implementation of the SR logic. In particular, the SR abstraction can support different types of system approaches to the processing and sharing; namely, a centralized approach that employs a single processing engine, a decentralized approach where the processing load is spread over all the SR members, or a hierarchical approach where a preprocessing is carried out by the SR members and a selected processed information is then passed to the next layer of the hierarchy for further computations.

Figure 1.1 illustrates the high level design of the SR. As shown in this Figure, the SR abstraction includes two principal building blocks; namely, *Complex Event Processing and Applications*, and *Data Dissemination*. These blocks can vary from SR to SR depending on the software technologies used to implement the SR processing and sharing logic. In addition, SR management building blocks are to be employed in order to manage the SR lifecycle and monitor the adherence to the SR contract by its members. SR members can inject raw data into the SR by means of SR Gateways components deployed at the administrative boundaries of each SR member. Raw data may include real time data, inputs from human beings, stored data (e.g., historical data), queries, and other types of dynamic and/or static content that are processed in order to

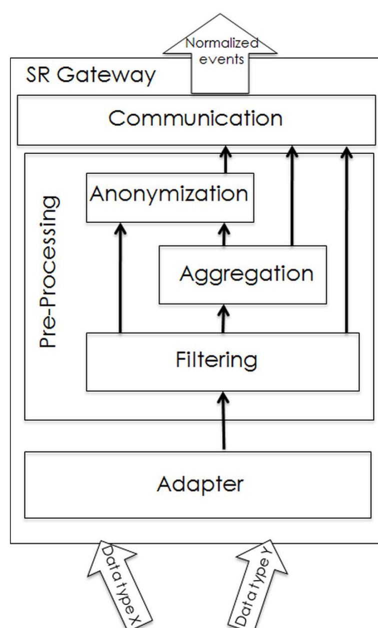


Figure 1.2: The SR Gateway. The *Adapter* module transforms raw input data of different types into a uniform format. *Pre-processing* modules filter (to discard useless data), aggregate (to decrease the amount of data to be injected into the SR) and anonymize (to meet privacy-related requirements) input data. The *Communication* module finally injects normalized events into the SR.

produce required output. Raw data are properly preprocessed by SR Gateways in order to normalize them and satisfy confidentiality requirements as prescribed by the SR contract. Processed data can be used for internal consumption within the SR: in this case, derived events, models, profiles, blacklists, alerts and query results can be fed back into the SR (the dotted arrows in Figure 1.1) so that the SR members can take advantage of the intelligence provided by the processing. The processed data are made available to the SR members via their SR Gateways; SR members can then use these data to properly instruct their local security protection mechanisms in order to trigger informed and timely reactions independently of the SR management. In addition, a (possibly post-processed) subset of data can be offered for external consumption. SRs in fact can be willing to make available for external use their output data. In this case, in addition to the SR members, there can exist clients of the SR that cannot contribute raw data directly but can simply consume a portion of the SR output data. SR members have full access to both the raw data provided by the other members, and the data output by the SR. Data processing and results dissemination are carried out by SR members based on obligations and restrictions specified in the contract.

1.3.1 The SR Gateway

An important component of the SR abstraction is represented by the SR Gateway. It is typically hosted within the IT boundaries of the SR members and acts as an access point for the local management systems. It is responsible for adapting the incoming raw data to the formats consumable by the SR processing engine, and sanitizing the data according to confidentiality constraints. Both the formats and the confidentiality requirements are specified in the SR contract, as previously described. Thus, the SR Gateway, along with the processing and data dissemination components, can vary from SR to SR. Figure 1.2 shows the principal modules that constitutes the SR Gateway. An adapter is used to interface the local systems management of the SR members. The adapter is designed so as to interface possibly heterogeneous data types (e.g., data from databases, data from networks, data included into files) and to convert the data in the format specific

of the SR. The adapter then dispatches the normalized data to a preprocessing module of the SR Gateway which performs the following three main functions

- filtering, which selects only the data of interest for the SR objective to meet;
- aggregation, which can reduce the amount of data being sent to the processing building block of the SR;
- anonymization, which arranges to remove any clue that could unveil the identity of the SR member providing the data.

Data format conversion and data aggregation typically constitute the so-called data unification process, as defined in [40].

Note that some of the preprocessing functions (and the preprocessing module itself) can be activated in the Gateway only if required; that is, according to specific SR contract clauses. For instance, if the contract states that some sensitive data must be anonymized before being injected into the SR, the anonymization sub-module is properly enabled in the Gateway. Section 1.5.2 describes an example of SR in which anonymization is performed by the SR Gateway.

The final output of the preprocessing (normalized events) is then sent to a communication module (or proxy) of the Gateway which is responsible for injecting it into the SR (Figure 1.2). The communication module can be properly configured to embody specific communication protocols when injecting the data to the SR.

1.3.2 Enforcing Confidentiality Requirements

As explained in Section 1.1, one of the main obstacles to the adoption of Collaborative Environments is the difficulty to meet confidentiality requirements. Such requirements consist in avoiding that sensitive information are inferred by inspecting computation input/output or by analyzing the traffic within the SR. Coping with these issues strictly depends on the level of trust among SR members.

In case a Trusted Third Party (TTP) is agreed upon by all the members and such TTP provides the computational infrastructure, input data can be directly sent to the TTP, which carries out the required elaboration and then disseminates the results to SR members. What is missing in this solution concerns how to prevent information leakage from the output diffused to SR members. Even though addressing this problem is very application-specific, several general techniques are described in the literature based on controlled perturbation or partial hiding of the output of a computation (association rule hiding [126, 134], downgrading classifier effectiveness [59, 118], query auditing and inference control [70, 89]. The side effect of these approaches is the worsening of the accuracy of the result, which is usually dealt with by focussing and leveraging the existing tradeoffs between the required level of privacy and the necessary accuracy of the output; a thorough analysis of such tradeoffs is reported in some of the earlier cited works. An additional solution is the so-called Conditional Release Privacy-preserving Data Aggregation (CR-PDA), presented in [37], which consists in disclosing only the items of the output that satisfy specific conditions (i.e., the item is the IP address of a malicious host). This class of scenarios characterized by the presence of a TTP in charge of executing the computation can be suitable for the SR abstraction, since the TTP can become an SR member and the SR Gateways can be configured so that all the other members send input data to the TTP and then receive back the results, which are properly anonymized by the TTP itself on the basis of the techniques just cited.

If no TTP can be employed, some additional mechanisms are required to anonymize both input data and the identity of the member which provided some specific datum. As for the anonymization of input data, while the fields of the single input datum that do not contribute to the final result can be filtered out before being fed into the SR (thanks to the Filtering module of the SR Gateway), the other fields that contain sensitive information have to be adequately modified. Several techniques exist for dealing with this issue:

adding noise to input data (randomization method [29, 28]) and reducing the granularity of data representation so as to make any item indistinguishable from a certain number of other items (k-anonymity [133] and l-diversity [108]). The cited papers also describe what kinds of computations can be performed on an input perturbed in this way and analyze the tradeoff between the provided privacy level and the correctness of the obtained results. These techniques can be integrated within the SR model by conveniently implementing the anonymization module of the SR Gateway so that the required perturbation is applied to input data before being injected into the SR. Techniques from the field of secure Multi-Party Computation (MPC) can be also employed, which allow for the anonymization of provided inputs by leveraging encryption and routing mechanisms to be enforced by the participants of the computation [110, 51, 88, 37]. When there is only the need to make anonymous the physical host that provides an event, lighter and faster techniques could be used [92, 88]. Specific routing algorithms can be implemented directly into the Communication module of the SR Gateway. For what concerns the anonymization of the output, the same observations and solutions reported for the scenario with a TTP hold.

In essence, the choice of the solution to adopt strongly depends on the specific application and on the privacy requirements. In the most general case, input data has to be filtered (Filtering module), encrypted and/or perturbed (Anonymization module) and a proper routing on some specific layout has to be enforced (Communication module).

1.4 Fair Information Sharing

So far, we have seen that mechanisms are available for enforcing privacy and anonymity within an SR, and in general within Collaborative Environments, although their effective employment in real scenarios has to be proved yet. This means that it is possible to have full control on what information get disclosed during the sharing. Nevertheless, the information that a member shares provide added value to all the other members, especially in today's data-centric scenarios where businesses mainly focus on the strategic value of data. Even if the benefits received by the collaboration exceeded the disadvantages due to the disclosure of sensitive information, potentially to market competitors, anyway the issue of selfish behaviors arises. Indeed, there is the possibility for opportunistic members to forge the data they share so as to avoid in practice any leakage of sensitive information. In this way, the overall value obtained by all the members is diminished since part of the input is not provided, but selfish members don't incur in any loss due to the disclosure of relevant information, so such an opportunistic strategy would result in a net gain without any cost.

What is required in order to address such issue consists of a set of mechanisms and policies to enforce some notion of fairness in the way members behave when sharing data in a Collaborative Environment. Layfield et al. [101] investigated a similar problem by framing it in the context of game theory. Their model comprises a set of participants (that match the notion of SR members) that exchange information to achieve a common goal and at the same time to maximize their own gain according to some cost function. Each participant doesn't know which strategies are adopted by the others, and it is willing to adapt by modifying its strategy to reflect the one it believes that performs best. As participants' strategies evolve in this way by mutually influencing each other, some strategies will become dominant while others will disappear. Available strategies include telling always the truth, always lying, telling the truth with a certain probability or when the estimated value of the data to be shared is over a certain threshold. An additional action in the model is the verification of the truth of the information provided by a participant, which is executed with a certain probability every time a new piece of data is received. Such verification has a cost (resources have to be employed for some period of time to execute the verification process) and a level of accuracy (the outcome of the verification could be wrong in reality), that have to be taken into account when deciding how to react on the basis of the outcome of this verification. The response to a false shared information can be the interruption of information exchange with the malicious participant for some period of time, which is likely to negatively impact on the gain of liars in the Collaborative Environment as more and more

participants discover they don't behave honestly. In order to facilitate and speed up the isolation of liars, trust scores are collected and disseminated among participants [87] so that the less a participant is trusted in the Collaborative Environment, the more likely it is that the truth of the information it shares is verified, and consequently a higher number of participants isolate it. Simulations have shown that most of the time all the participants end up adopting a honest strategy because it turns out to be the most convenient in the long run.

Even though based on a simplified model of Collaborative Environments and providing results out of simulations only, the work just presented suggests that reaching fairness in information sharing mainly depends on the behavior of participants, which should be as much honest as possible in order to enable the complete exploitation of the potential of collaboration. A reasonable way to make participants behave fairly is to setup policies of incentives and punishments such that honesty becomes the most advantageous strategy to adopt. A mean for verifying the truthfulness of what participants share should be a must-have feature of the Collaborative Environment.

1.5 An SR for Fraud Monitoring

In this section we describe an SR we implemented, and we highlight the aspects related to confidentiality issues. In the *Department of the Treasury* of the *Ministry of Economic and Finance* (MEF), the V Directorate entitled "Financial crime prevention", and in particular the *Central Office for Means of Payment Fraud* (UCAMP [17]) is responsible for monitoring counterfeit Euros and preventing fraud committed through the use of payment means other than cash [1]. Within the community system set up to protect Euro from being counterfeited, introduced by Regulation 1338/2001, UCAMP serves as the Italian central office for the collection and exchange of technical and statistical data regarding cases of falsifications detected throughout the national territory. On the basis of a strategic analysis of the information received by financial institutions, UCAMP is constantly able to evaluate the impact of the fraud phenomenon on the economic and financial system. In carrying out its functions, UCAMP performs complex data analysis and collaborative information sharing, acquiring data from banks and others financial institutions. The analysis aims at detecting specific types of frauds such as counterfeit euros and payment card frauds by correlating data that can be apparently fully uncorrelated with each other.

In order to show the high flexibility of the SR abstraction, we used it for enabling information sharing and processing between the earlier mentioned different financial institutions. In this section, we introduce the design and implementation of an SR we refer to as *FM-SR*, we built in collaboration with UCAMP (MEF) so as to deploy an online location intelligence event-based platform for fraud monitoring. The FM-SR exploits a Complex Event Processing (CEP) engine for collaboratively correlating fraud information coming from banks and others financial institutions. UCAMP is responsible for the processing of the SR; however, it needs to outsource this task to an application service provider that actually performs the correlation. The results produced by the SR are consumed by UCAMP in order to meet its institutional functions and by banks and other involved financial bodies for protecting themselves from fraudulent episodes (e.g., banks can plan to open a new agency in a geographical area with a low density of frauds). A simpler solution could have been used, consisting in a centralized database where participants share their own data and where correlation can be performed; the choice to use a more complex architecture that includes a CEP engine was a specific requirement set by UCAMP: in perspective, UCAMP plans to use such technology to accommodate for many other streams with very heterogeneous properties, and for other collaborative scenarios where an online processing can be necessary.

The FM-SR allowed us to assess: (i) the applicability of the SR abstraction to the real world in the financial context; (ii) the ability of the platform to be easily integrated into "legacy" IT infrastructures of financial stakeholders, without requiring any updates to existing IT solutions nor significant efforts; (iii) the feasibility of collaboratively correlate data coming from different institutions; (iv) the possibility to

[1]In this section, only the following text has been agreed to be disclosed.

instrument the SR gateway in order to anonymize sensitive data before being processed; (v) the capability to localize and study fraud phenomena in different areas of the Italian territory.

1.5.1 Analysis and Design

In order to exploit the SR abstraction capabilities for fraud monitoring purposes, we first carried out an assessment of the data UCAMP obtains from banks and other financial institutions, so as to evaluate which data streams could be used to enable a collaborative correlation.

Data streams

We identified and used two types of data streams: one related to counterfeit euro notes, and another related to frauds carried out with electronic payment means. As for the former, data come from reports of banks that register such information as date, ABI and CAB of the bank, serial number and denomination of euros presented as counterfeit banknote. In some cases, the identity of the person who presented the counterfeit euros at the bank is also sent to UCAMP. As for the latter, we distinguish between different types of data: *unauthorized POSs*, that are Points Of Sale managed by merchants who carry out malicious trade activities, *disregarded credit card transactions* that concern all those transactions that are denied by the owner of the credit card, and *tampered ATMs*.

Types of data correlation

From an assessment of the earlier discussed data, it emerged that the types of correlations we could perform in the SR principally regarded the geographical location of the occurred frauds. Note that the information related to disregarded credit card transactions turned out to be not relevant for our purposes, as the reported data were related only to the city where the denied credit card transactions were carried out (and mostly those cities were located outside the geographical boundaries of Italy).

Owing to this observation, we focused our attention on the remaining three types of data streams (i.e., counterfeit euros, unauthorized POSs and tampered ATMs) and we designed three different correlation algorithms. In all the algorithms, we identified the precise address in the Italian territory where the fraud events happened.

The address was obtained using both ABI and CAB identifiers of bank branches, and the location address of the POS. Each address has been further translated into a point in latitude and longitude necessary for carrying out a spatial correlation among fraud events. In doing so, we exploited geocoding services offered by Google and Yahoo in order to build a local database (see below) we periodically interrogate during the processing phase for executing the correlation. The correlation algorithms are summarized in the following.

GeoAggregation The underlying principle of the GeoAggregation algorithm is to divide the Italian territory in small areas of equal size, roughly of 1.1 km^2 each. The areas are obtained by truncating the latitude and longitude coordinates. For instance, given a point (41.876883, 12.474309) we truncate the digits up to the second and obtain the South-West point of an imaginary rectangle. Within each area, we keep track of the fraud events that are observed. To this end, we assign a weight to the events; the weight depends on the severity and certainty that the events are actually occurred in that area. Hence, in our current implementation of the GeoAggregation algorithm, we assign 8 as weight for tampered ATMs, 4 as weight for unauthorized points of sale and 3 to counterfeit euros. The low value we associated with the counterfeit euros is motivated by the fact that banknotes can be rapidly distributed and it is likely that the point in space in which they appear is not the actual location where they have been produced.

DEF: based on these weights, for a given geographical area x, we define $rank(x)$ as follows:

$$rank(x) = (T_ATM(x) + UAuth_POS(x) + C_Euro(x))$$

The rank is used to identify the so-called “hot areas" of the Italian territory, marking them with a low, medium or high concentration of fraud episodes.

Entropy We use the notion of entropy in order to compute how much the crimes are “different" among themselves within each geographical area (even in this case, the areas are computed as earlier described). With the term “different" we mean that the frauds are related to the three different types of events we focused on, and are carried out by different people. For instance, if the same ATM is tampered twice in a row with the same method (e.g., installing a micro-camera to register the pin code of users) it is likely that the same people maliciously act on it.

DEF: for a given geographical area x, we define $E(x)$ as follows:

$$E(x) = -\frac{\sum_j p_{z_j} log(p_{z_j})}{log(N_z)}$$

where p_{z_j} is the frequency of each criminal event z_j, and N_z is the cardinality of the set of events happened in the area. The entropy assumes a value in the range $[0, 1]$ so that if all the fraud events occurred within an area are different among each other, the entropy of that area is close to 1; otherwise the entropy is close to 0.

Counterfeit Euros Finally, we carried out a deep analysis of the distribution of counterfeit euros (Section 1.5.2 for a user-level evaluation of such a distribution). The algorithm we developed evaluates the spreading on the Italian territory over a certain interval of time of counterfeit banknotes; for each note, it computes its cardinality; that is, the number of different banks in which the same serial number of the banknote has been reported. A high cardinality likely means that the counterfeit banknote was made in a large quantity.

In addition, we observed that some banknotes were of the same type and their serial numbers were similar with one another. This observation turned out to be particularly interesting for UCAMP (MEF) people in order to study the phenomenon. Thus, we evaluated the distribution of such kinds of banknotes, that are likely produced by the same counterfeiter, by constructing a similarity graph $G(V, E)$. V is the set of vertices represented by the banknotes and $E : \{e :< v_1, v_2 >\in E| if\, and\, only\, if\, h(v_1, v_2) \leq t\}$. In other words, the similarity between banknotes is detected by computing the hamming distance h between two serial numbers. If that distance is <= 2 (i.e., t) then we mark those banknotes as similar.

Privacy Requirements

Some of the data to be exchanged, for example the tampered ATMs, contain sensitive information that, if not protected during the processing, can reveal the identity of banks subject to frauds. Since UCAMP entrusts the processing to an application service provider, UCAMP required us to design a processing system capable of preserving the anonymity of such sensitive information. In particular, the goal is to avoid simple traceability between an event injected into the SR and the bank that generated such event.

Since all the events are received by the application service provider, the latter should be prevented from inferring which bank produced an event by simply observing the fields of the event itself. To meet this requirement, in our SR we used two strategies: we removed sensitive data fields that do not contribute significantly to the correlation logic, and we modified the content of necessary data fields in such a way to make difficult identifying the bank that originated the data, performing at the same time the correlation.

A weakness of this approach is that the application service provider could discover the bank that originated an event just by tracing the host that sent the event itself, since that host is likely to be hosted exactly by the bank that generated the event. For this reason, another key point is preventing the application service provider, and any other SR member in general, from linking banks to events by analyzing the traffic. This

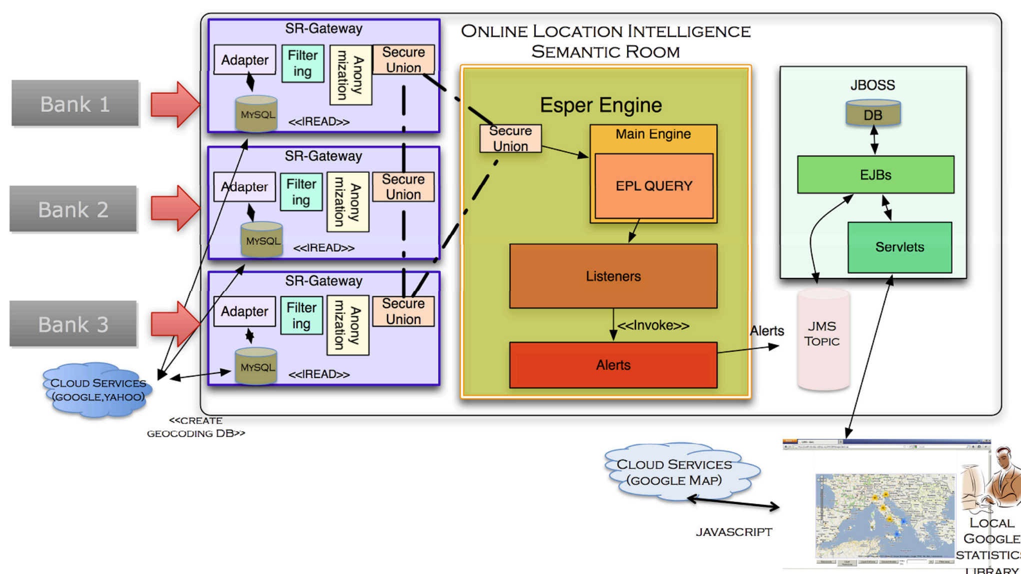


Figure 1.3: The online location intelligence SR for fraud monitoring

can be achieved by employing some combination of the techniques described in Section 1.3.2. How these privacy requirements are enforced in practice in the FM-SR is described in Section 1.5.2.

1.5.2 Implementation

The FM-SR consists of a number of preprocessing elements, a CEP engine and an advanced web-based Graphical User Interface (GUI). The preprocessing elements are the SR Gateways deployed at each SR member site. The SR members are banks and other financial institutions that send to UCAMP data regarding frauds.

The processing element is represented by the CEP engine managed by an application service provider to which UCAMP outsources the data correlation task. The results of the processing are rendered to SR Members by means of an advanced web-based interface that visualizes geolocation information. The individual components of the FM-SR are illustrated in Figure 1.3 and described in detail below.

SR Gateway

The SR Gateway takes as input the data provided by the SR members and performs a number of operations: it converts the data into events (the events are represented by Plain Old Java Objects (POJOs)) for the use by the Esper engine; it reads from a MySQL database (local to each SR Gateway) the information for the conversion of ABI and CAB of banks into addresses, and their related latitudes and longitudes. For this purpose, we use Google and Yahoo cloud services to construct such a database (see Figure 1.3). This information is then included into the POJOs sent by the SR Gateways.

Enforcing Privacy Requirements

Two different steps are executed in order to anonymize the data. In the first step, the data fields of the events that can uniquely identify the originating bank are obfuscated. For instance, ABI/CAB are filtered out and the latitude and longitude are modified by adding a small noise to both, like in the randomization method [29, 28]. This noise allows us to maintain the event in its geographical area, avoiding at the same time that the latitude and longitude are the same of the bank that produced the data. Note that without this

noise, it would be possible to discover, during the processing, the identity of the financial institution that produced the event. Nevertheless, perturbing coordinates by itself do not guarantee the anonymity of the bank producing the information when the number of banks in a rectangular area is very low. A similar issue has been debated in [27]. The authors proposed to distinguish and tune the anonymization requirements for distinct items. Thus we are working to a solution that adapts the dimension of the rectangular areas so that each area includes, at least, a predefined number of banks.

In the second step, the anonymized events are to be sent to the CEP engine. As shown in Figure 1.3, we used Esper [11] as CEP engine, which will be described in details in Section 2.4.1. Here we want to focus mainly on the way confidentiality requirements can be addressed in reality. Sending events directly to the engine would leak the identity of the banks, thus violating UCAMP requirements. Therefore, we use a communication proxy located in the SR Gateway capable of anonymizing the communication between the banks and the CEP engine. In particular, all the SR Gateways periodically execute a secure-union [92] of events; the secure-union is a well known primitive in multi-party computation that allows many participants $\{p_1, p_2, ..., p_n\}$ to compute the union $U : \{e_{p_1}, e_{p_2}, e_{p_3}, ..., e_{p_n}\}$ of their individual input (the input of p_j is e_{p_j}) in such a way that for a specific participant p_x it is not possible to pin-point an element $e_{p_x} \in U$. The secure union primitive has been used by many complex algorithms for Anonymous Intrusion Detection [88] and Oblivious Assignment of m-Slots [39].

In our case, the inputs are the events generated by each SR member, and the resulting set is the set of events that is sent to the CEP engine. We implemented this primitive using a cryptographic scheme that is re-encryptable (such as the scheme described in [77]).

In the scheme, the SR members act on the top of a logic ring; a circulating token is sent by a SR member leader and each SR member adds its encrypted element to the token and re-encrypts all the remaining elements. With this scheme, the CEP engine receives periodically a set of events : $\{e_{p_1}, e_{p_2}, e_{p_3}, ..., e_{p_n}\}$ from the banks where each even e_{p_j} is anonymous in the sense that is not possible to know its original sender. The only role of the leader is to start the union; it does not obtain any information respect to other parties. The election of the leader can be fixed, or in a synchronous faulty environment, can be demanded to standard techniques. In our implementation we used a rotating leader, with a new leader for each round; this approach has the advantage to balance the load in the system: a process has to re-encrypt all the elements added by the processes between it and the round leader.

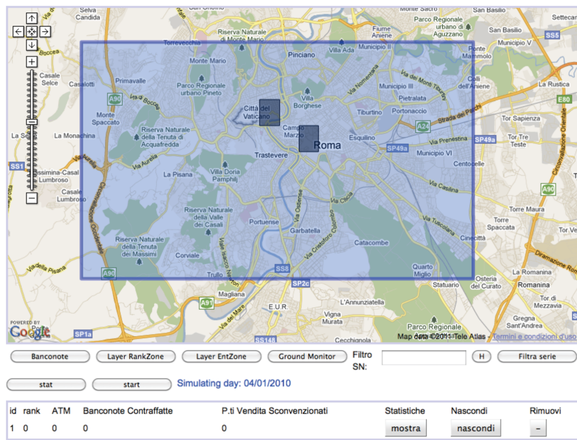
Presentation Layer: Web-based Graphical User Interface

The presentation layer of the FM-SR is represented by a web-based GUI [2] developed in Javascript with the AJAX technology. The web-based GUI calls a number of servlets that in turn interface the EJB layer in order to obtain the alerts generated by the CEP engine (see Figure 1.3). These alerts are then visualized on a map. For showing alerts on a map, we used such Google cloud services as the Javascript Maps APIs, without compromising possible confidentiality requirements required by UCAMP for sensitive data since the Google services receive only information related to the map size and locations. The web-based GUI (Figure 1.4) consists of four principal layers; namely the Banknote, GeoAggregation, Entropy, and Ground Monitor layers, each of them briefly described in the following.

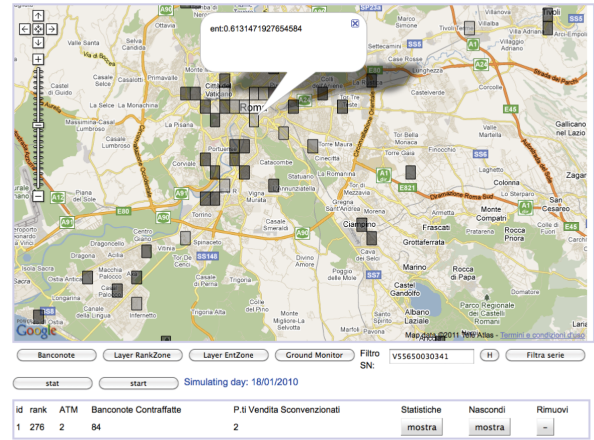
The Banknote Layer shows the spreading of the counterfeit euros in the territory at real time (Figure 1.4 (a), in the screenshot this layer is called “banconote"). The clusters include the number of banknotes and are colored accordingly (red for high concentration, yellow for medium and blue for low). The clusters are dynamically modified at run time as soon as new data are processed by the CEP engine.

The GeoAggregation layer is responsible for visualizing the so called “hot areas”; that is, areas that show a high (red color), medium (yellow color) or low (green color) concentration of fraud events (Figure 1.4 (b), in the screenshot, this layer is called “Layer RankZone").

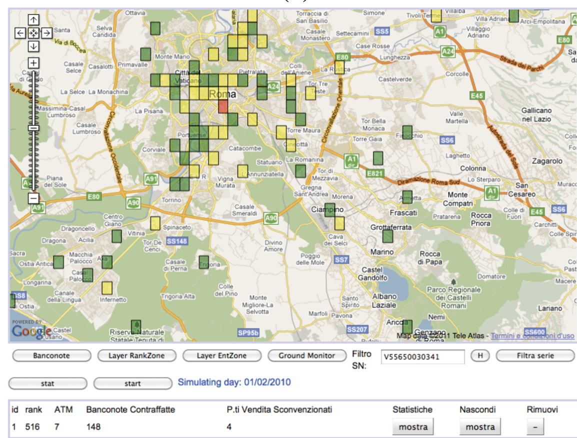
[2] In the screenshots of Figure 1.4, some parts are in Italian as the final prototype was meant to be shown to Italian law enforcement agencies.



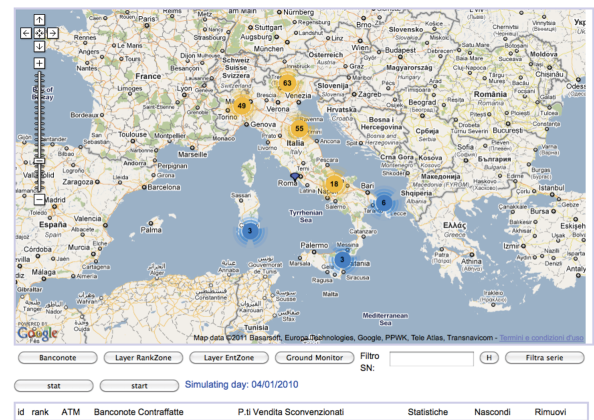
(a)



(b)



(c)



(d)

Figure 1.4: Web-based Graphical User Interface: (a) Banknote Layer; (b) GeoAggregation Layer; (c) Entropy Layer; (d) Ground Monitor Layer

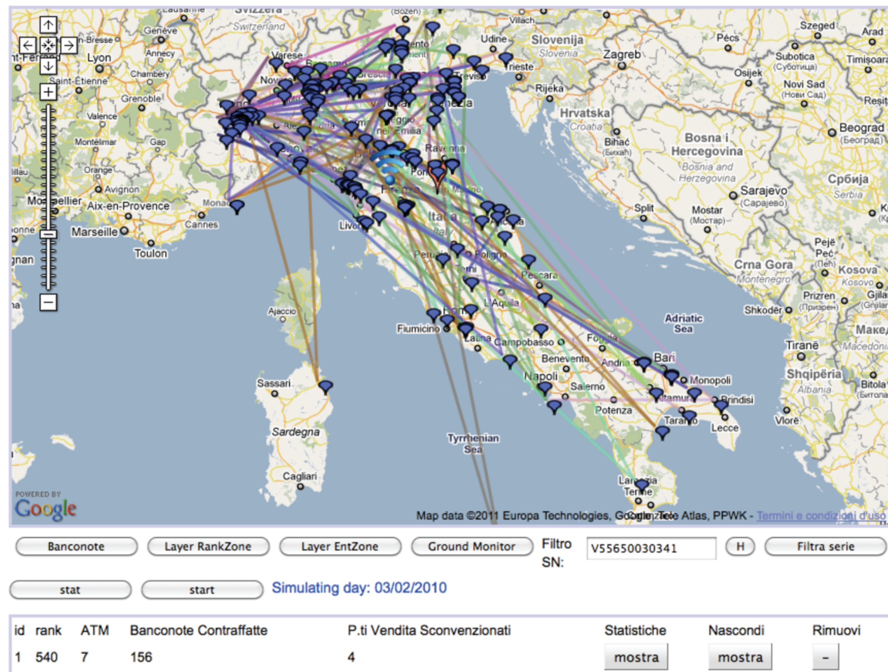


Figure 1.5: Connection among counterfeit banknotes events

The Entropy layer allows a SR member to visualize the fraud entropy in a zone (Figure 1.4 (c), in the screenshot, this layer is called “Layer EntZone"). Specifically, rectangles are blank if the entropy is low and hence there are similar types of frauds in that area, grey if the entropy is medium, and white if the entropy is close to 1; that is, if there exist many different types of frauds in that area.

The Ground Monitor Layer (see Figure 1.4 (d)) provides a sort of *on demand fraud monitoring*. In other words, a SR member can select a specific geographical area of interest by drawing a rectangle on the map, and monitor all the results of the previous layers in that rectangle, only.

Italy has unfortunately the sad primacy of being the European country with the largest production of counterfeit euros. These banknotes are distributed not only on the Italian territory but also in all European Union. A deep analysis of this phenomenon can then be crucial in the field of fraud monitoring and detection.

Using the FM-SR prototype we were able to evaluate the distribution in space and time of the counterfeit banknotes. Figure 1.5 illustrates how counterfeit banknotes events are connected with each other: events occurred on the same day are connected with dots of the same color.

In conclusion, the FM-SR allowed us to build a set of advanced tools to support intelligence operations carried out by UCAMP. These tools can be also conveniently exploited by banks and financial institutions so as to monitor geographical areas with a high density of frauds, and by police institutions and officers of the “Guardia di Finanza” in order to (i) identify the geographic distribution of frauds over the Italian territory and within specific areas of interest selected on demand by the users handling the tools, (ii) monitoring the spreading of counterfeit banknotes and of suspicious banknotes’ serial numbers that can be potentially produced by the same counterfeiters, and, finally, (iii) analyze the trend of the fraud phenomenon.

Chapter 2

Collaborative Port Scan Detection

In this chapter, we present a specific SR, which we refer to as *ID-SR* (Intrusion Detection SR), whose objective is to prevent potential intrusion attempts by detecting malicious inter-domain stealthy SYN port scan activities. Our goal here is to show the flexibility of the SR abstraction to accommodate a variety of different implementations and we do not concentrate on other aspects such as privacy or fairness. Thus, in this specific case, we assume that SR members trust each others and we propose three different implementations of the ID-SR: one based on Esper [11], one implemented in Storm [18] and another one based on Agilis [31], a platform based on the Hadoop framework [150].

Large enterprises are nowadays complex interconnected software systems spanning over several domains. This new dimension makes difficult for them the task of enabling efficient security defenses. Section 2.1 frames the content of this chapter within the works published in literature about single-source port scan detection. In Section 2.2, the inter-domain SYN port scan is presented as a reconnaissance activity performed by attackers to gather valuable information about possible vulnerabilities of information systems while remaining as much hidden as possible to common Intrusion Detection Systems (IDSs). This Chapter proposes an architecture of an IDS which uses a processing platform (online like Esper and Storm, or batch-oriented like Agilis) and includes software sensors deployed at different enterprise domains. Each sensor sends events to the processing framework for correlation. In Section 2.3, we devise an algorithm for the detection of SYN port scans named Rank-based SYN (R-SYN). It combines and adapts three detection techniques in order to obtain a global statement about the malice of hosts behavior. The implementation of R-SYN is presented in the three aforementioned processing platform: Esper (Section 2.4), Storm (Section 2.5) and Agilis (Section 2.6). Several evaluations have been carried out that show how the accuracy of the detection improves thanks to the collaboration. These evaluations also report interesting comparisons between the Esper-based and the Storm-based implementations, and between the Esper-based and the Agilis-based implementations (Section 2.7).

The contents of this chapter are based on [107, 34, 31, 73, 33].

2.1 Related Work

In this chapter, we focus on attacks carried out by a single host; attacks where multiple hosts are used in a coordinated way are addressed in [43]. A lot of works exist concerning techniques for detecting single-source port scans. According to the survey by Bhuyan et al. [50], these detection techniques can be classified into five distinct categories on the basis of the approaches employed for the detection

- *algorithmic approaches*: network traffic is analyzed using hypothesis testing and probabilistic models [141, 102, 90];
- *threshold-based approaches*: the detection is triggered when some monitored parameter goes beyond some pre-defined threshold [79, 132, 93];

- *soft computing approaches*: methods like approximate reasoning and partial truth are used to face the uncertainty and ambiguity usually met in real situations [48, 84, 72];

- *rule-based approaches*: a knowledge base of rules is employed to discern normal traffic from malicious activities [109, 125, 91];

- *visual approaches*: monitored data is displayed to human operators, that are in charge of recognizing suspicious patterns [64, 98, 119];

The R-SYN algorithm we present in this chapter combines three threshold-based techniques, and ranks their outcomes to get a numeric result which is in turn compared against another threshold. Therefore, the R-SYN algorithm belongs to the threshold-based category. A proper comparison of R-SYN with other threshold-based algorithms with respect to the detection effectiveness is infeasible because of the difficulty of obtaining the same used data sets. Anyway, the focus of this chapter is mainly on the comparison of distinct implementations of the same algorithm, rather than on showing how the R-SYN algorithm compares with other port scan detection algorithms.

2.2 Inter-domain stealthy SYN port scan

The goal of the attack is to identify open TCP ports of the attacked SR members. The ports that are detected as open can be used as intrusion vectors at a later time.

The attack is carried out by initiating a series of low volume TCP connections to ranges of ports at each of the targeted SR members. Specifically, the attack we consider is named TCP SYN (half-open) port scan and spans different administrative domains (the different domains of the SR members). We call this attack *inter-domain SYN port scan*.
It is characterized by probe connections that are never completed; that is, the three-way TCP handshake is never accomplished. Let us consider a single scanner S, a target T and a port P to scan. S sends a SYN packet to the endpoint $T : P$ and waits for a response. If a SYN-ACK packet is received, S can conclude that $T : P$ is open and can optionally reply with an RST packet to reset the connection. In contrast, if an RST-ACK packet is received, S can consider P as closed. If no packet is received at all and S has some knowledge that T is reachable, then S can conclude that P is filtered. Otherwise, if S does not have any clue on the reachability status of T, then it cannot assume anything about the state of P.

Note that not all the scans can be considered as malicious. For instance, there exist search engines that carry out port scans in order to discover web servers to index [85]. It becomes then crucial to distinguish accurately between actual malicious port scans and benign scans, thus minimizing so-called *false positives*; i.e., legitimate port scans that have been erroneously considered as malicious. In the next section, we describe the R-SYN algorithm, which aims at improving the detection accuracy of the ID-SR. Other algorithms for SYN port scan detection are possible; e.g., In [33], an algorithm based on line-fitting is presented, together with its comparison with R-SYN.

2.3 Rank-based SYN port scan detection algorithm

The R-SYN algorithm adapts and combines in a novel fashion three different port scan detection techniques; namely, *Half open connections*, *Horizontal and Vertical port scans*, and *Entropy-based failed connections* so as to augment the chances of detecting stealthy malicious scan activities. The three techniques are described below.

2.3.1 Half open connections

This technique analyzes the sequence of SYN, ACK, RST packets during the three-way TCP handshake. In the normal case, the sequence is the following: (i) SYN, (ii) SYN-ACK, (iii) ACK. In the presence of a SYN port scan, the sequence becomes (i) SYN, (ii) SYN-ACK, (iii) RST (or nothing) and we refer to it as an *incomplete connection*. For a given IP address, if the number of incomplete connections is higher than a certain threshold T_{HO} (see below), then we can conclude that such IP address is likely carrying out malicious port scanning activities.

DEF: for a given IP address x, let $count_{HO}(x)$ be the number of incomplete connections issued by x; we define $HO(x)$ as follows:

$$HO(x) = \begin{cases} 1 & \text{if } count_{HO}(x) > T_{HO} \\ 0 & \text{otherwise} \end{cases}$$

2.3.2 Horizontal and vertical port scans

In a horizontal port scan, the attackers are interested in a specific port across all the IP addresses within a certain range. In a vertical port scan, the attackers scan some or all the ports of a single destination host [138].

In our R-SYN algorithm, we adapt the Threshold Random Walk (TRW) technique described in [85]. TRW classifies a host as malicious by observing the sequence of its requests. Looking at the pattern of successful and failed requests of a certain source IP, it attempts to infer whether the host is behaving as a scanner. The basic idea consists in modeling accesses to IP addresses as a random walk on one of two distinct stochastic processes, which correspond to the access patterns of benign source hosts and malicious ones, respectively. The detection problem then boils down to observing a specific trajectory and then inferring the most likely classification for the source host. While the original technique considers as a failure a connection attempt to either an unreachable host or to a closed port on a reachable host, we adapt the TRW technique in order to distinguish between them, since the former concerns horizontal port scans whereas the latter concerns vertical port scans. We accordingly design two modified versions of the original TRW algorithm.

Specifically, in order to detect horizontal port scans, we identify connections to both unreachable and reachable hosts. Hosts are considered unreachable if a sender, after a time interval from the sending of a SYN packet, does not receive neither SYN-ACK nor RST-ACK packet, or if it receives an ICMP packet of type 3 that indicates that the host is unreachable. In contrast, hosts are reachable if a sender receives SYN-ACK or RST-ACK packet. For each source IP address, we then count the number of connections to unreachable and reachable hosts and apply TRW algorithm. Let $TRW_{HS}(\mathrm{x})$ be the boolean output computed by our TRW algorithm adapted for horizontal port scans for a certain IP address x. *True* output indicates that x is considered a scanner, otherwise it is considered a honest host.

DEF: for a given IP address x, we define $HS(x)$ as follows:

$$HS(x) = \begin{cases} 1 & \text{if } TRW_{HS}(\mathrm{x}) == \text{true} \\ 0 & \text{otherwise} \end{cases}$$

In order to detect vertical port scans, we first identify connections to open and closed ports. We then count such connections for each source IP address. Let $TRW_{VS}(\mathrm{x})$ be the boolean output computed by our TRW algorithm adapted for vertical port scans for a certain IP address x.

DEF: for a given IP address x, we define $VS(x)$ as follows:

$$VS(x) = \begin{cases} 1 & \text{if } TRW_{VS}(\mathrm{x}) == \text{true} \\ 0 & \text{otherwise} \end{cases}$$

2.3.3 Entropy-based failed connections

As previously noted, not all the suspicious activities are actually performed by attackers. There exist cases where the connections are simply failures and not deliberate malicious scans.

As in [159], we use an entropy-based approach to discriminate failures from malicious port scans. In contrast to [159], we evaluate the entropy by considering the destination endpoint of a TCP connection; that is, destination IP and destination port. The entropy assumes a value in the range [0, 1] so that if some source IP issues failed connections towards the same endpoint, its entropy is close to 0; otherwise, if the source IP attempts to connect without success to different endpoints, its entropy is close to 1. This choice originates by the observation that a scanner does not repeatedly probe the same endpoints: if the attempt fails, a scanner likely carries out a malicious port scan towards different targets.

Given a source IP address x, a destination IP address y and a destination port p, we define $failures(x, y, p)$ as the number of failed connection attempts of x towards $y : p$. For a given IP address x, we define $N(x)$ as follows:

$$N(x) = \sum_{y,p} failures(x, y, p)$$

In addition, we need to introduce a statistic about the ratio of failed connection attempts towards a specific endpoint. At this regard, we define $stat(x, y, p)$ as follows:

$$stat(x, y, p) = \frac{failures(x,y,p)}{N(x)}$$

The normalized entropy can then be evaluated by applying the following formula:

DEF: for a given IP address x,

$$EN(x) = -\frac{\sum_{y,p} stat(x,y,p) \cdot \log_2 stat(x,y,p)}{\log_2 N(x)}$$

2.3.4 Ranking

All the above statistics are collected and analyzed across the entire set of the ID-SR members, thus improving chances of identifying low volume activities, which would have gone undetected if the individual participants were exclusively relying on their local protection systems (e.g., Snort Intrusion Detection system [14]). In addition, in order to increase the probability of detection, suspicious connections are periodically calibrated through a ranking mechanism aimed at capturing distinct of the behavior of a scanner.

Our ranking mechanism sums up the three values related to half opens, horizontal port scans and vertical port scans, and weights the result by using the entropy-based failed connections.

DEF: for a given IP address x, we define $rank(x)$ as follows:

$$rank(x) = (HO(x) + HS(x) + VS(x)) \cdot EN(x)$$

Such ranking is compared ($\geq$) to a fixed threshold in order to mark an IP address as scanner. IP addresses marked as scanners are placed in a blacklist, which is then disseminated among ID-SR members.

2.4 Esper Intrusion Detection Semantic Room

The Esper-based ID-SR consists of a number of preprocessing and processing elements hosted on a cluster of machines provided by the SR members. Specifically, the preprocessing elements are the SR Gateways deployed at each SR Member site, while the processing element is represented by an Esper instance.

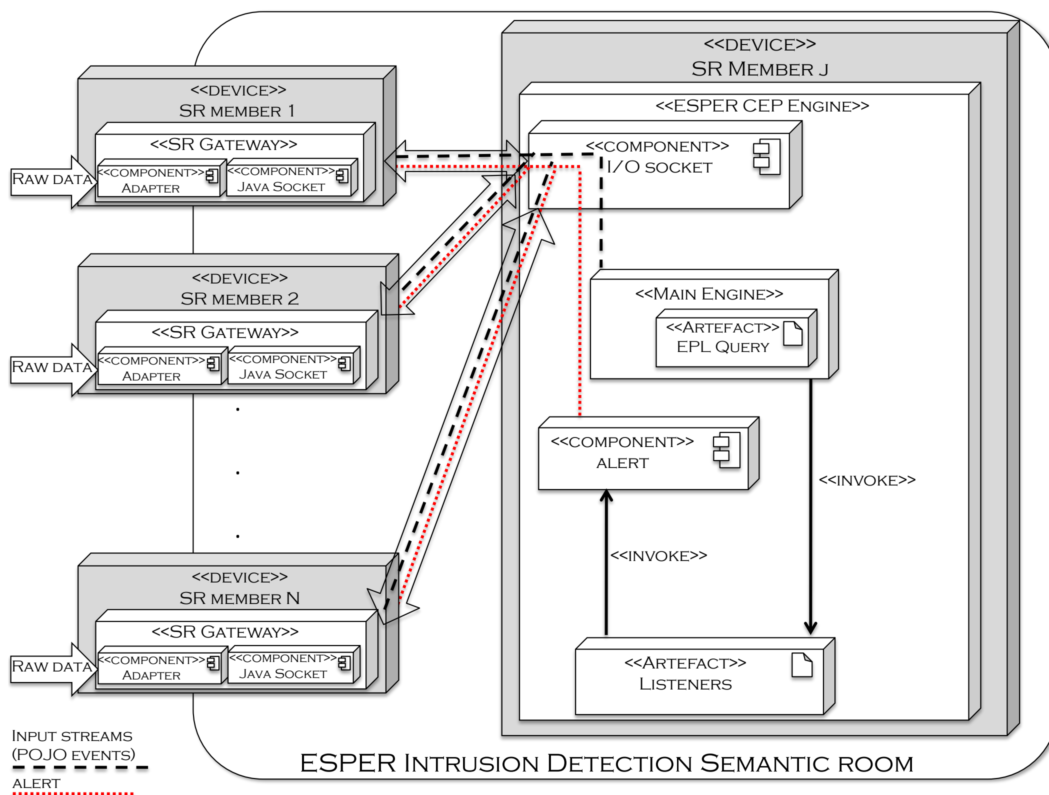


Figure 2.1: Architecture of the Esper-based Intrusion Detection Semantic Room

2.4.1 Esper

The processing in the ID-SR is carried out by Esper, a well known open source Complex Event Processing (CEP) engine [11]. This engine is fully implemented in Java; it receives Plain Old Java Objects (POJOs) that represent the events it has to analyze (input streams). The processing logic is specified in a high level language similar to SQL named Event Processing Language (EPL). EPL queries run continuously over input streams of POJOs; when an EPL query finds a match against its clauses in its input streams, its related listener is invoked. A *listener* is a Java object that can be subscribed to a particular stream so that whenever the query outputs a new tuple for that stream, the *update*() method of the listener is invoked. The listeners can execute specific functions or simply raise an alert to be further elaborated.

2.4.2 R-SYN Algorithm in Esper

The individual components of the ID-SR are illustrated in Figure 2.1 and described in detail below.

SR Gateway

Raw data owned by ID-SR Members can potentially originate from different sources such as databases, logs, or traffic captured from the internal networks of members. In order to be analyzed by Esper, the data are to be normalized and transformed in POJOs. To this end, the SR Gateway has been designed and implemented so as to incorporate an adapter component that (i) takes as input the flows of raw data (see Figure 2.1), (ii) preprocesses them through filtering operations (i.e., only packets related to TCP three-way handshake) in order to transform the data in the format prescribed by the SR contract, and (iii) wraps the preprocessed data in POJOs so that they can be analyzed by Esper. We implemented TCPPojo for TCP packets and ICMPPojo for ICMP packets. Each POJO maps every field of interest in the header of the related protocol. POJOs are

normalized and sent to Esper through sockets. When sending the POJOs, our implementation maintains the order of captured packets, which is crucial when evaluating sequence operators in the EPL queries.

ID-SR Processing Steps

The processing steps followed by the Esper-based ID-SR implementation can be summarized as follows. Firstly, raw data sniffed from the network of each SR Member are collected. We have implemented the SR Gateway component so as to collect data at real time by running the *tcpdump* utility and collect network traces. Each SR Gateway normalizes the incoming raw data by producing a stream of TCPPojos of the form ⟨SOURCEIP, DESTINATIONIP, SOURCEPORT, DESTINATIONPORT, FLAGTCP, SEQUENCENUMBER, ACKNUMBER⟩. TCPPojos are sent to Esper through socket. The incoming TCPPojo are received by Esper (see Figure 2.1) and passed to the *Main Engine* for correlation purposes.

Each detection technique is implemented by using a combination of EPL queries and listeners. The EPL queries are in charge of recognizing any packet pattern of interest according to the detection technique. Listeners are invoked whenever such matches occur. For instance, we use the EPL query introduced in Listing 2.1 in order to identify incomplete connections.

Listing 2.1: EPL query for half open connections

```
insert into halfopen_connection
  select ...
  from pattern [
    every a = syn_stream --> (
      ( b = syn_ack_stream(...) --> (
        (timer:interval(60 sec) or <c>) and not <d>
      ) where timer:within(61 sec) ) ) ]
```

In this query, we exploit the *pattern* construct of Esper to detect patterns of incomplete connections. In particular, a is the stream of SYN packets, b is the stream of SYN-ACK packets, $< c >$ is the stream of RST packets and $< d >$ is the stream of ACK packets, all obtained through filtering queries. Such pattern matches if the involved packets are within a time window of 61 seconds.

Further queries are then used and bound to listeners: they filter IP addresses that made more than T_{HO} (in our implementation, we set it to 2) incomplete connections and update the data structure representing the list of half open connections. Interested readers can refer to [34] for the comprehensive implementation in EPL of the R-SYN algorithm.

IP addresses marked as scanners are placed in a blacklist (the *Alert* component of Figure 2.1 performs this task); the blacklist is sent back to the SR Gateways so that the ID-SR members can take advantage of the intelligence produced by the SR.

2.5 Storm Intrusion Detection Semantic Room

The implementation described in the previous section uses a centralized CEP engine. Although Esper is claimed to provide impressive performance in terms of processing latency and event throughput [7], it also exhibits the typical weaknesses of a centralized system.

A key aspect of a Collaborative Environment like the SR is the ability to change the membership as the need arises. In case new members join an SR, the SR should reconfigure available resources to adapt to the new situation, in particular it should be able to sustain the event streams provided by new participants without any sensible degradation of the performance. This requires the employment of a processing platform that can scale in/out to arrange the computational power required to process the data volume generated by

SR members. Once the members altogether produce a traffic higher than a certain threshold, a centralized CEP engine becomes a bottleneck, worsening the overall performance.

Another well known issue of centralized architectures is their difficulty to supplying fault tolerance. In the specific case of Esper, there is an (non open source) extension called EsperHA [1] (Esper High Availability) which employs checkpointing and hot backup replicas to face possible failures. The required queries and the state of the computation are stored in some stable storage and recovered by replicas as required. Storing such information can have a detrimental effect on the performance, which makes Esper efficiency even worse.

A viable solution for these two problems is the usage of a distributed processing platform able to seamlessly scale to adapt to input load and efficiently provide robust mechanisms to meet fault tolerance requirements. In this section, we present a software called Storm [18] which provides the building blocks for carrying out a computation in a distributed fashion. We create an SR for intrusion detection that uses Storm to implement the R-SYN algorithm.

2.5.1 Storm

Storm is an open source distributed realtime computation system. It provides an abstraction for implementing event-based elaborations over a cluster of physical nodes. The elaborations consist in queries that are continuously evaluated on the events that are supplied as input. A computation in Storm is represented by a *Topology*, which is a graph where nodes are operators that encapsulate processing logic and edges model data flows among operators. In the Storm's jargon, such a node is called *Component*. The unit of information that is exchanged among Components is referred to as a *tuple*, which is a named list of values. There are two types of Components: (i) *spouts*, which model event sources and usually wrap the actual generators of input events so as to provide a common mechanism to feed data into a Topology, and (ii) *bolts*, which encapsulate the specific processing logic of operators such as filtering, transforming and correlating tuples.

The communication patterns among Components are represented by *streams*, which are unbounded sequences of tuples emitted by spouts or bolts and consumed by bolts. Each bolt can subscribe to many distinct streams in order to receive and consume their tuples. Both bolts and spouts can emit tuples on different streams as needed. Spouts cannot subscribe to any stream, since they are only meant to produce tuples. Users can implement the queries to be computed by leveraging the Topology abstraction. They put into the spouts the logic to wrap external event sources, then compile the computation in a network of interconnected bolts which take care of properly handling the output of the computation. Such a computation is then submitted to Storm, which is in charge of deploying and running it on a cluster of machines. Figure 2.2 shows an example of Topology with spouts generating streams of tuples towards bolts which elaborate them, exchange other tuples among them and use an external storage to save the output.

An important feature of Storm consists in its capability to scale out a Topology to meet the requirements on the load to sustain and on fault tolerance. There can be several instances of a Component, called *Tasks*. The number of Tasks for a certain Component is fixed by the user when it configures the Topology. If two Components communicate through one or more streams, also their Tasks do. The routing logic among the Tasks of these two communicating Components is driven by the *grouping* chosen by the user. Let A be a bolt that emits tuples on a stream consumed by another bolt B. When a Task of A emits a new tuple, the destination Task of B is determined on the basis of a specific grouping strategy. Storm provides several kinds of groupings

- *shuffle grouping*: the target Task is chosen randomly, ensuring that each Task of the destination bolt receives an equal number of tuples;

- *fields grouping*: the target Task is decided on the basis of the content of the tuple to emit; for example, if the target bolt is a stateful operator analyzing events about customers, the grouping can be based on

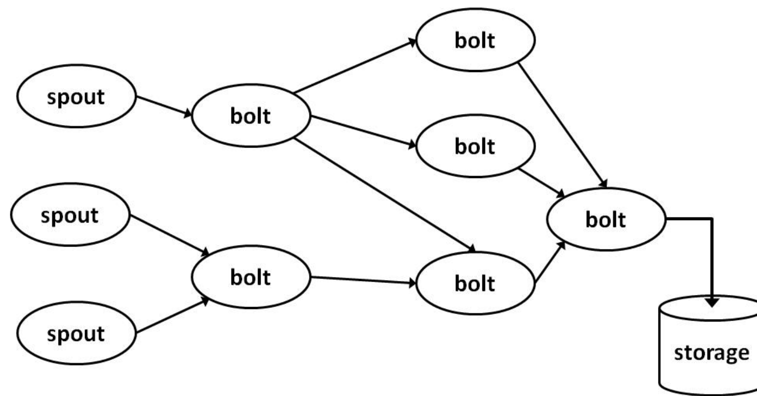


Figure 2.2: A Storm Topology where spouts produce input tuples and bolts carry out the computation, finally saving the output in a storage.

a customer-id field of the emitted tuple so that all the events about a specific customer are always sent to the same Task, which is consequently enabled to properly handle the state of such customer;

- *all grouping*: each tuple is sent to all the Tasks of the target bolt; this grouping can be useful for implementing fault tolerance mechanisms;
- *global grouping*: all the tuples are sent to a designated Task of the target bolt;
- *direct grouping*: the source Task is in charge of deciding the target Task; this grouping is different from fields grouping because in the latter such decision is transparently made by Storm on the basis of a specific set of fields of the tuple, while in the former such decision is completely up to the developer.

Worker Nodes and Workers

From an architectural point of view, a Storm cluster consists of a set of machines called *Worker Nodes*. Once deployed, a Topology consists of a set of threads running inside a set of Java processes that are distributed over the Worker Nodes.

A Java process running the threads of a Topology is called a *Worker* (not to be confused with a Worker Node: a Worker is a Java process, a Worker Node is a machine of the Storm cluster). Each Worker running on a Worker Node is launched and monitored by a *Supervisor* executing on such Worker Node. Monitoring is needed to handle failures of Workers.

Each Worker Node is configured with a limited number of *slots*, which is the maximum number of Workers in execution on that Worker Node. A thread of a Topology is called *Executor*. All the Executors executed by a Worker belong to the same Topology. All the Executors of a Topology are run by a specific number of Workers, fixed by the developer of the Topology itself. An Executor carries out the logic of a set of Tasks of the same Component, while Tasks of distinct Components live inside distinct Executors. Also the number of Executors for each Component is decided when the Topology is developed, with the constraint that the number of Executors has to be lower than or equal to the number of Tasks, otherwise there would be Executors without Tasks to execute.

Requiring two distinct levels, one for Tasks and one for Executors, is dictated by a requirement on dynamic rebalancing that consists in giving the possibility at runtime to scale out a Topology on a larger number of processes (Workers) and threads (Executors). Changing at runtime the number of Tasks for a given Component would complicate the reconfiguration of the communication patterns among Tasks, in particular in the case of fields grouping where each Task should repartition its input stream, and possibly its state, accordingly to the new configuration. Introducing the level of Executors allows to keep the number

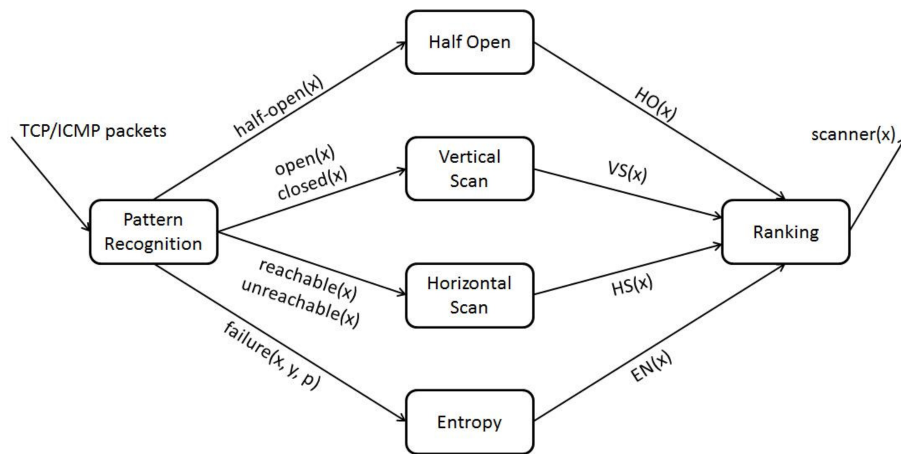


Figure 2.3: Computation graph (Storm Topology) of the R-SYN algorithm

of Tasks fixed. The limitation of this design choice consists in the Topology developer to overestimate the number of Tasks in order to account for possible future rebalances.

At the time of this writing, the unique way to overcome this limitation is to kill the Topology, apply the required changes and redeploy it. Obviously, this represents a big issue in scenarios where input events are generated continuously, indeed a number a number of additional problems arise, like how and where to buffer input events during the redeployment, and how to manage the state of stateful Components. The same problems hold when coping with faults. Indeed, in case of failure, affected Tasks have to be moved in a different physical machine, so their state has to be kept somewhat and their input events have to be buffered somewhere. Even though works exist in the literature that address these aspects [55], the mechanisms they propose have not been implemented yet in available processing platforms, and in particular in Storm. In this thesis we don't focus on this aspect, and to keep things simple we always consider topologies where the number of Tasks for any Component is equal to the number of Executors, that is each Executor includes exactly one Task.

Nimbus

Nimbus is a single Java process in charge of accepting a new Topology, deploying it over Worker Nodes and monitoring its execution over time in order to properly handle any failure. Thus, Nimbus plays the role of master with respect to Supervisors of Workers by receiving from them the notifications of Workers failures. Nimbus can run on any of the Worker Nodes, or on a distinct machine.

The coordination between Nimbus and the Supervisors is carried out through a ZooKeeper cluster [86]. The states of Nimbus and Supervisors are stored into ZooKeeper, thus they can be restarted without any data loss in case of failure.

The software component of Nimbus in charge of deciding how to deploy a Topology is called *scheduler*. On the basis of the Topology configuration, the scheduler has to perform the deployment in two consecutive phases: (i) assign Executors to Workers, (ii) assign Workers to slots.

2.5.2 R-SYN Algorithm in Storm

The first step towards the implementation of an algorithm in Storm is the definition of the related Topology; that is, the computation graph which characterizes the algorithm itself. Figure 2.3 shows such a computation graph for the R-SYN algorithm.

The *Pattern Recognition* Component is the unique spout of the Topology. It reads TCP/ICMP packets from the network to be monitored and performs pattern recognition tasks for producing its output streams.

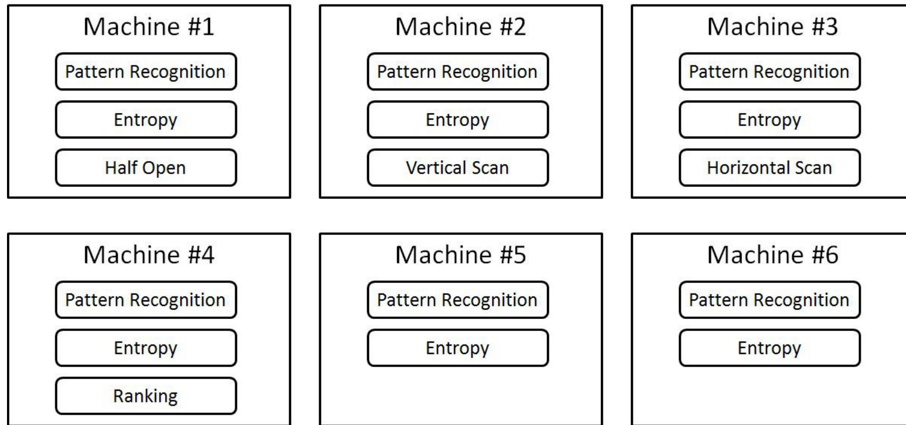


Figure 2.4: Possible deployment of the R-SYN Topology in a six-machine cluster

The Component is replicated: each SR member has a Pattern Recognition Task and can provide its own input data to the SR. Its implementation is done by embedding in each Task an Esper instance which is in charge of recognizing the patterns of interest. Since all the bolts are stateful, all the stream groupings are setup as *field grouping*.

The *Half Open* bolt receives events representing the occurrences of half open connections made by some host x (half-open(x)), and maintains a state for tracking how many half open attempts have been issued by any source host. In case a predefined threshold is exceeded for a certain host x, then an event $HO(x)$ is sent to the *Ranking* bolt.

The *Vertical Scan* (*Horizontal Scan*) bolt receives events modeling connection attempts to open/closed ports (reachable/unreachable hosts) and outputs a $VS(x)$ ($HS(x)$) event when for a certain host x a fixed threshold has been exceeded, meaning that a vertical (horizontal) scan has been detected. It has to maintain a state about how many events have been received for each source host.

The *Entropy* bolt gets $failure(x, y, p)$ events as input. Each of these events means that a host x has issued a failed connection attempt (closed port or unreachable host) towards the port p of the destination host y. Upon the reception of such an event, it updates the entropy value for the host x according to the formulas presented in Section 2.3. It then sends to the Ranking bolt an event $EN(x)$ containing a new value.

The *Ranking* bolt is in charge of collecting all the events sent by the other bolts, computing the ranking for each suspicious host and triggering a scanner(x) event if the ranking is over a predefined threshold.

As for the deployment, we configured Storm so as to consider one thread for each Task of each Component. A possible deployment of the R-SYN Topology in a cluster with six machines is shown in Figure 2.4. The Topology has been configured to replicate both the Pattern Recognition and Entropy Components in each available machine; the other bolts are not replicated at all. In this case, the replication is aimed at providing load balancing.

2.6 Agilis Intrusion Detection Semantic Room

So far, we have seen computations executed continuously on flowing input streams by employing either centralized (Esper) or distributed (Storm) approaches. An alternative approach consists in running the computation periodically rather than continually. Regularly over time, the elaboration can be executed on a batch of data, which at least includes all the input data gathered since the last execution. Whereas this batch-oriented approach leads to greater delays for the availability of the results, which negatively impacts on the timeliness of the computations, yet it brings some benefits to take into account when deciding what approach to embrace for a specific scenario.

Running the computation at regular intervals gives the opportunity to perform required reconfiguration operations that would be complicated to carry out in case the computation were continuously running. For example, operations like reallocating at runtime Storm Tasks so as to optimize performance (see Chapter 3), or scaling out Storm Components to adapt to traffic evolution over time in order to avoid bottlenecks, or simply moving a Storm Task to another physical machine because of a failure, they all require the computation (or part of it) to be put in stand-by, events to be buffered in upstream Components and possibly a proper management of any stateful Component. All these operations are a must for enabling the employment of Storm in practice, indeed dynamic optimizations, facing changes in traffic patterns and especially coping with hardware failures have become common daily routines within today's data centers. If a batch approach is adopted, each computation can be optimized before being started by taking into account any opportunity to improve performance, by replicating parts of the computation that receive heavier loads, and by avoiding to use resources that have already been detected as failed.

This section presents Agilis, a batch-oriented distributed processing platform based on Hadoop [150]. One of the main feature of Agilis is the employment of a RAM-based storage rather than the Hadoop default disk-based storage (HDFS [137]), with the aim to speed up data accesses and mitigate the intrinsic performance penalty due to the embracement of a batch approach. Another key aspect concerns the locality awareness inherited by Hadoop which allows to move the computation where input data is stored rather than the opposite. This has a beneficial impact on the performance since the transfer of huge volumes of data is very time demanding. Furthermore, in Collaborative Environments where data have to be exchanged among distinct organizations through the Internet, reducing data transfers allows to save network bandwidth, which is a limited, and consequently very valuable, resource. Before introducing Agilis, a brief description of Hadoop is provided in order to give all the required basis to fully understand the characteristics of Agilis. Then, the implementation of a slightly different version of the R-SYN algorithm in Agilis is presented.

2.6.1 Hadoop and the MapReduce paradigm

Apache Hadoop [150] is a Java-based open source framework which allows the distributed processing of large data sets across clusters of computers. It is based on the MapReduce programming model [68], which lets a user define the desired computation in terms of *map* and *reduce* functions. The underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks.

A Hadoop computation is called *job* and its work is decomposed in *map* tasks and *reduce* tasks. Input data is fed to a variable set of map tasks that perform part of the overall elaboration and produce an halfway output. Such halfway output is then computed by a fixed number (cluster-wise configuration parameter) of reduce tasks for the provision of the final output. A Hadoop deployment consists of (i) a single *JobTracker* in charge of creating and scheduling tasks and monitoring job progress and (ii) a set of *TaskTrackers* which execute the tasks allocated by the JobTracker and report to it several status information.

One of the key characteristics of Hadoop is its ability to allocate map tasks where input data are placed, so as to minimize data transfers and decrease the overall job latency. Such locality awareness fits very well with the need of timely and smartly reconfiguring the allocation of maps to resources since a proper task placement is enforced before starting each job.

2.6.2 Agilis

Agilis is a distributed platform for sharing and correlating event data generated at real time by multiple sources which may be geographically distributed. To simplify programming, the processing logic is specified in a high-level language, called *Jaql* [10], and expressed as a collection of *Jaql queries*. Each query consists of SQL-like constructs combined into flows using the "->" operator (a few examples of Jaql queries can be found in Section 2.6.3).

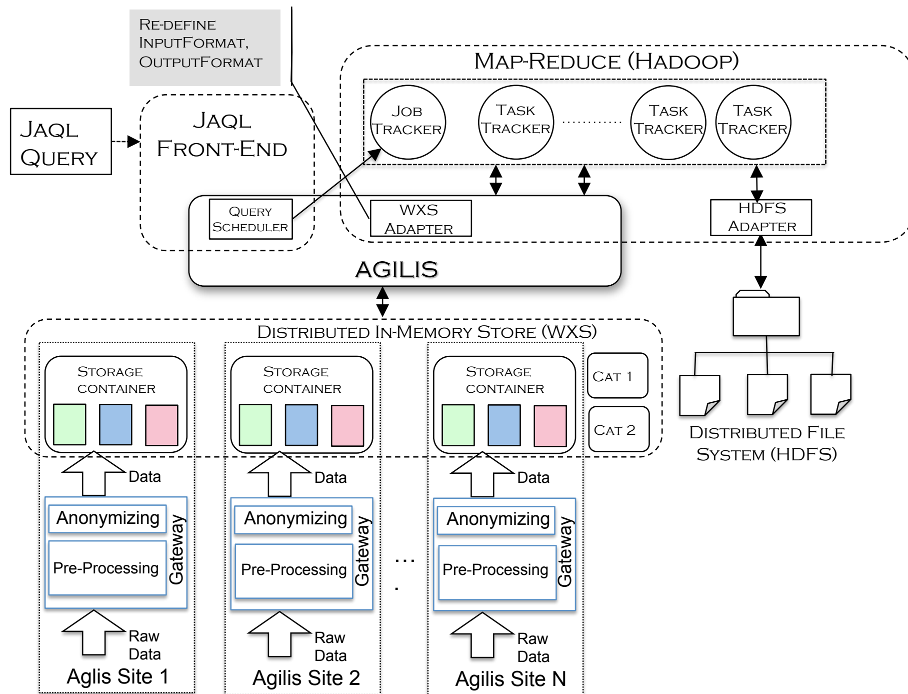


Figure 2.5: The architecture of Agilis

The queries are processed by the Jaql front-end, which breaks them into a series of MapReduce jobs to be executed on the Hadoop infrastructure (see Figure 2.5). The MapReduce framework naturally lends itself to supporting collaborative processing thanks to the mutual independence of map tasks, which as a result can be placed close to the respective sources input data (see Section 2.6.2).

Each MapReduce job submitted by the Jaql front-end implements a portion of the original query. The data is communicated through the platform-independent JSON [5] format, which streamlines the integration of data stored in a variety of formats. In the context of collaborative processing, this feature is useful for combining data of different nature into a single cohesive flow. For example, in addition to a real time data, which can be supplied through a RAM-based data store or a message queue, some of the participants may choose to also provide long-lived historical data that can be loaded from a stable storage, such as a distributed file system (e.g., Hadoop Distributed File System (HDFS) [137]) or a database.

Collaboration and Scalability

Agilis is deployed on a collection of physical (or virtual) machines, henceforth referred to as *Agilis Nodes*, spread over a number of (possibly widely) distributed *Agilis Sites*. Each Agilis Site corresponds to a single organization participating in the Collaborative Environment, and is responsible for hosting at least one Agilis Node. The specific components being within the boundaries of each Agilis Site are the *Agilis Gateway* (see Section 2.6.2), the *Data Storage*, and the Hadoop TaskTracker. The TaskTracker instance hosted at each Agilis Node is in charge of managing the lifecycle of locally executed map and reduce tasks as well as monitoring their progress. The TaskTracker instances are coordinated by a single instance of the JobTracker component, which is deployed at one of the Agilis Node. The JobTracker is also responsible for handling the incoming MapReduce jobs submitted by the Jaql front-end.

To reduce the latency and overheads associated with the disk I/O, Agilis implements support for a RAM-based data store, which can be used for feeding the real time input data as well as storing the intermediate and output data generated by MapReduce. We chose the Java-based *IBM WebSphere eXtreme Scale (WXS)*

system [9] as the implementation of the RAM-based store due to its scalability, robustness, and interoperability with the other components of our design. WXS exposes an abstraction of a *map* (meant as a data structure, not a MapReduce task) consisting of relational data records. For scalability, a WXS map can be split into a number of *partitions*, each of which is assigned to a distinct WXS *container*. The information about the available WXS containers, as well as the mapping of the hosted maps to the live containers, is maintained internally by the WXS *Catalog* service, which is replicated for high availability.

In Agilis, the entire input data is treated as a single logical WXS map, which is partitioned so that each Agilis Node hosted within a particular Agilis Site is responsible for storing a portion of the event data produced locally at that site. Subsequently, each partition is mapped to a single Hadoop input split (which is the minimum unit of data that can be assigned to a single map task), and processed by an instance of map task scheduled by the JobTracker through one of the local TaskTrackers. To ensure collocation of input splits with their assigned map tasks, Agilis provides a custom implementation of the Hadoop *InputFormat* API, which is packaged along with every MapReduce job submitted to the JobTracker by the Jaql front-end (see Figure 2.5). In particular, the information about the physical locations of the input splits is coded into the InputFormat *getSplits()* implementation, which queries the WXS Catalogue to extract the endpoints (IP address and port) of the WXS containers hosting the corresponding WXS partitions. In addition, the implementation of *createRecordReader()* is used to create an instance of *RecordReader*, which codes the logic to convert the input data to a form that can be consumed by an instance of map task.

To further improve the locality of processing, we utilize the embedded WXS SQL engine to execute simple queries directly on the data stored within the container. Specifically, our implementation of RecordReader recognizes the SQL *select*, *project*, and *aggregate* constructs (all of which are understood by the embedded SQL engine) by interacting with the Jaql interpreter wrapper, and delegates their execution to the SQL engine embedded into the WXS container.

As we demonstrate in Section 2.6.3, the above locality optimizations are effective to substantially reduce the amount of intermediate data reaching the reduce stage, thus contributing to system scalability.

Data Preprocessing

The Agilis Gateway is a component that feeds WXS partitions and is deployed within the boundaries of the organization participating in the Collaborative Environment. Raw data are given as input to the Agilis Gateway which preprocesses the data (e.g., data could be aggregated and filtered) before storing them in WXS containers. The Agilis Gateway can also include the logic to anonymize and sanitize the data before injecting them into Agilis. To reduce data transfer overheads, the WXS containers and the Agilis Gateway are typically hosted by the same Agilis Node.

Overview of Agilis Operation

The execution of Jaql queries is mediated through a scheduler collocated with the JobTracker. In the current implementation, the scheduler accepts the MapReduce jobs (along with some configuration parameters) associated with each submitted query, and resubmits them to JobTracker for periodic execution, according to a configured cycle.

The preprocessed data are fed into Agilis by the Gateway and stored in the WXS containers located on the local Agilis Node. The data stored in the WXS containers are processed by the locally spawned map tasks as explained above. The intermediate results are then stored in temporary buffers in RAM created through WXS, from where they are picked up by the reduce tasks for global correlation. The final results are then stored in either WXS or HDFS (as prescribed by the configuration data submitted with each Jaql query).

One limitation of the above approach is that each repeated submission of the same collection of MapReduce jobs incurs the overheads of processing initialization and re-spawning of map and reduce tasks, which negatively affects the overall completion time (see Section 2.7.3). Another deficiency stems from the lack

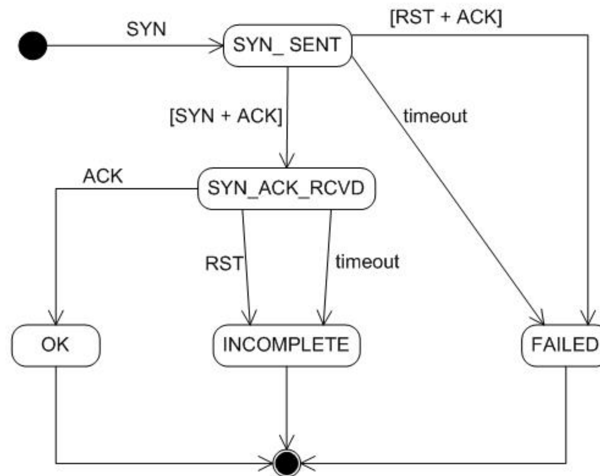


Figure 2.6: The Finite State Machine of Agilis Gateway

of dynamic adjustment to incoming data rates, with the risk that the processing can be initiated on an incomplete input window, or cause it to overflow.

2.6.3 R-SYN Algorithm in Agilis

We have implemented a slightly different version of the R-SYN algorithm in Agilis. The principal processing steps are described below, and are implemented by two principal components; namely an Agilis Gateway, responsible for performing preprocessing activities, and the Agilis middleware that carries out the correlation analysis on the data provided by the Gateways.

Preprocessing: Agilis Gateway

The Gateway component is deployed locally at each Agilis Site and is responsible for: (i) sniffing TCP packets flowing through the monitored network of an organization; (ii) recognizing incomplete and failed connections; (iii) maintaining the pairs (IP address, TCP port) probed by suspected source IP addresses; (iv) storing all the produced detection information into the collocated Data Storage.

Specifically, raw data captured from organization network are properly filtered in order to use only the TCP three-way handshake packets. Then, once a SYN packet is recognized, an instance of a Finite State Machine (FSM) is allocated by the Gateway and updated when other packets concerning the same TCP connection are captured. The Gateway uses the FSM in order to detect correct, incomplete and failed connections.

Incomplete and Failed Connections The FSM models the patterns of packets related to the three-way handshake phase of the TCP protocol as they are seen from the point of view of a source host S which begins a connection to a destination host D (see Figure 2.6). The transitions in the FSM represent the sending of a packet (e.g., SYN sent by S), the receipt of a packet (e.g., the transition marked as $[SYN + ACK]$ in Figure 2.6 received by S; we use square brackets to indicate this kind of packets) and the expiration of a timeout (e.g,: the edge marked as *timeout*).
Due to the presence of the two timeout transitions shown in Figure 2.6, it is certain that the FSM reaches its final state eventually, thus classifying the connection as either correct (i.e., OK in Figure 2.6), incomplete or failed; the FSM instance can then be deleted.

Visited (IP address, TCP Port) The Gateway maintains all the pairs (IP address, TCP port) probed by suspected source IP addresses. Once an incomplete or failed connection is identified, the Gateway considers it only if its destination (IP address, TCP port) has not been visited yet by its source IP. The final output of the Gateway is a set of suspected IP addresses, each one associated with the number of incomplete connections and the number of failed connections. Such a set is continuously updated and stored in the WXS partition local to the Agilis Site hosting the Gateway.

Agilis query execution

The processing specified by the Jaql query consists in grouping the outputs of all available Gateways by IP address. The sets of suspected IP addresses stored in different WXS partitions are then merged so that a single set of suspected IP addresses is produced, containing the information on the number of incomplete and failed connections issued by each of these sources over the networks of Agilis Sites.

Ranking The pair [(incomplete connections count), (failed connections count)] is the mark produced by the R-SYN implementation in Agilis for each suspected IP address. Such mark is compared to a fixed pair of thresholds, one for incomplete connections and the other for failed connections. If at least one of these thresholds is exceeded, the IP address is considered a scanner. The final output of Agilis (i.e., the set of scanner IP addresses detected by the system), is stored into the WXS partitions for the use by the Agilis Sites. The Jaql query which implements the ranking computation is the following.

```
read($ogPojoInput(
    "SuspectedIP",
    "agilis.portscan.data.SuspectedIP"))
-> group by $ip = {$.ip} into {
ip: $ip.ip,
incompleteConnections: sum($[*].incompleteConnections),
failedConnections: sum($[*].failedConnections)}
-> filter
    $.incompleteConnections > 4 or
    $.failedConnections > 12
-> transform {$.ip}
-> write($ogPojoOutput(
    "ScannerIP",
    "agilis.portscan.data.ScannerIP",
    "ip"));
```

The objects representing IP addresses suspected by any Agilis Site are first read from the $SuspectedIP$ WXS map (i.e., the *read* statement) and grouped by IP address, summing up their counters related to incomplete and failed connections (i.e., the *group by* and *sum* statements). The resulting merged set is then filtered using two fixed thresholds (i.e., the $filter$ statement), projected to the ip field (i.e., the $transform$ statement) and stored into the $ScannerIP$ WXS map (i.e., the *write* statement).

2.7 Experimental Evaluation

We carried out many experiments for assessing the effectiveness and the performance of the distinct implementations of the R-SYN algorithm. We first describe our testbed and the traces we used, then we present our results on accuracy and latency of scanner detection for an Esper-based implementation. Then, we provide our results on the comparison between the Esper-based and the Storm-based implementations, and between the Esper-based and the Agilis-based.

Testbed For our evaluation, we used a testbed consisting of a cluster of ten Linux-based Virtual Machines (VMs), each of which equipped with 2 GB of RAM and 40 GB of disk space. The ten VMs were hosted in a cluster of four quad core 2.8 Ghz dual processor physical machines equipped with 24 GB of RAM. The physical machines are connected through an Ethernet LAN of 10 Gbit.

Traces We used five intrusion traces. All traces include real traffic of networks that have been monitored. The traces are obtained from the ITOC research web site [12], the LBNL/ICSI Enterprise Tracing Project [13] and the MIT DARPA Intrusion detection project [4]. The content of the traces is described in Table 2.1. In each trace, the first TCP packet of a scanner always corresponded to the first TCP packet of a real port scan activity.

	trace1	trace2	trace3	trace4	trace5
size (MB)	3	5	85	156	287
source IPs	10	15	36	39	23
connections	1429	487	9749	413962	1126949
scanners	7	8	7	10	8
TCP packets	18108	849816	394496	1128729	3462827
3w-h packets	5060	13484	136086	883500	3393087
length (sec.)	5302	601	11760	81577	600
3w-h packet rate (p/s)	0.95	22.44	11.57	10.83	5655

Table 2.1: Content of the traces

2.7.1 R-SYN in Esper

We have carried out an experimental evaluation of the Esper-based R-SYN to assess two metrics; namely the *detection accuracy* in recognizing inter-domain stealthy SYN port scans and the *detection latency*. We used a VM to host the Esper CEP engine, and each of the remaining nine VMs represented the resources made available by nine simulated SR members participating in the SR. We emulated a large-scale deployment so that all the VMs were connected with each other through an open source WAN emulator we have used for such a purpose. The emulator is called WANem [19] and allowed us to set specific physical link bandwidths in the communications among the VMs.

Detection Accuracy In order to assess the accuracy of R-SYN, we partitioned the traces in order to simulate the presence of nine SR members participating in the SR. The partitioning was based on the destination IP addresses in order to simulate that distinct SR members can only sniff TCP packets routed to diverse addresses. The resulting sub-traces were injected to the available Gateways of each member in order to observe what the algorithm was able to detect. To this end, we ran a number of tests considering four accuracy metrics (following the assessment described in [162])

(i) *True Positive* (TP): the number of suspicious hosts that are detected as scanners and are true scanners;

(ii) *False Positive* (FP): the number of honest source IP addresses considered as scanners;

(iii) *True Negative* (TN): the number of honest hosts that are not indeed detected as scanners;

(iv) *False Negative* (FN): the number of hosts that are real scanners that the systems do not detect.

With these values we computed the *Detection Rate* (DR) and the *False Positive Rate* (FPR) as follows: $DR = \frac{TP}{TP+FN}$, and $FPR = \frac{FP}{FP+TN}$.

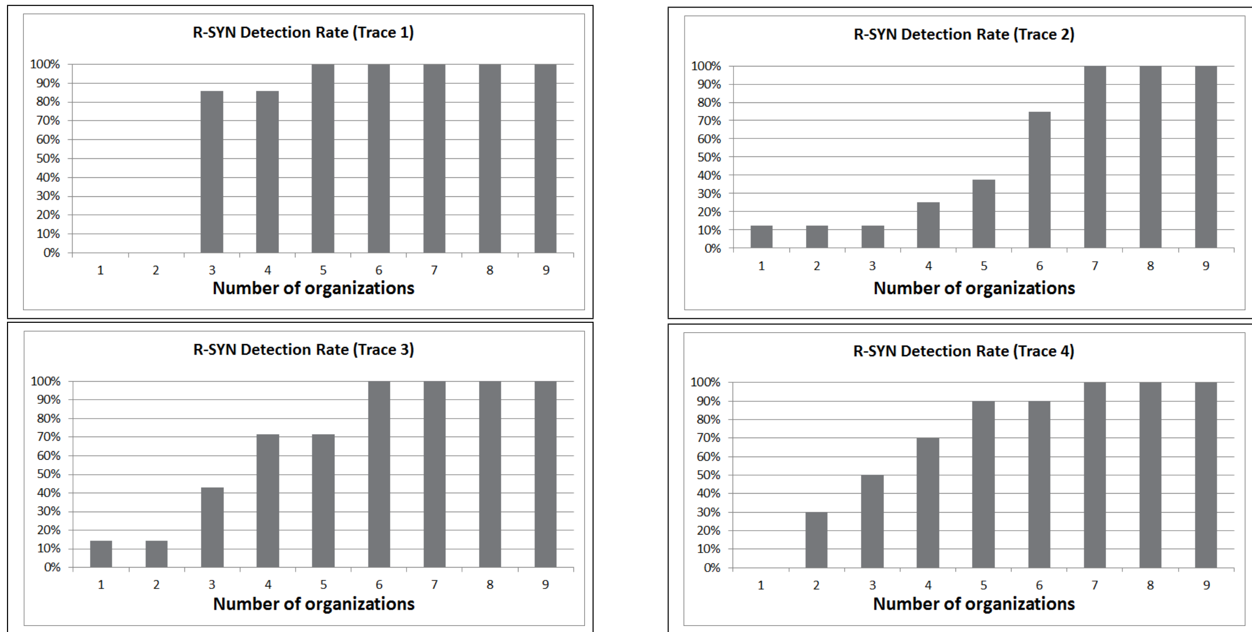


Figure 2.7: Port scan detection rate vs number of SR members for R-SYN algorithm. Each member contributes to the processing with a number of network packets that is on average $\frac{1}{9}$ of the size of the trace.

With all the traces, with the exception of trace 4, we observed that the algorithm didn't introduce errors in the detection; that is, the FPR was always 0% in our tests. With trace 4, R-SYN exhibited an FPR equal to 3.4%; that is, R-SYN introduced one False Positive scanner.

Figure 2.7 shows the obtained results for the *DR*. It emerges that the collaboration can be beneficial for sharpening the detection of port scanners. Augmenting the number of SR members (i.e., augmenting the volume of data to be correlated) leads to an increase of the detection rate as computed above.

Detection Latency In the port scan attack scenario, the detection latency should be computed as the time elapsed between when the first TCP packet of the port scan activity is sent by a certain IP address, and when the SR marks that IP address as scanner (i.e., when it includes such address in the blacklist). It is worth noting that we cannot know precisely which TCP packet should be considered the first of a port scan, since that depends on the true aims of who sends such packet. As already said, in our traces the first TCP packet of a scanner always corresponds to the first TCP packet of a real port scan activity, so we can compute the detection latency for a certain IP address x as the time elapsed between the sending of the first TCP packet by x and the detection of x as scanner.

In doing so, we need the timestamps of the packets. For such a purpose we developed a simple Java application named *TimerDumping* which (i) takes a trace as input; (ii) sends the packets contained in the trace (according to the original packet rate) to the Gateway by using a simple pipe; (iii) maintains the timestamp of the first packet sent by each source IP address in the trace.

We deployed an instance of TimerDumping on each VM hosting a Gateway component. Each TimerDumping produces a list of pairs $< ip_address, ts >$, where ts is the timestamp of the first TCP packet sent by $ip_address$. The timestamps are then used as beginning events for detection latency computation. Since there are more TimerDumping instances, pairs with the same IP address but different timestamps may exist. In those cases, we consider the oldest timestamp.
Timestamps are generated by using local clocks of the hosts in the cluster. In order to ensure an acceptable degree of synchronization, we configured all the clustered machines to use the same NTP server which has been installed in a host placed in the same LAN. The offset between local clocks is within 10 milliseconds,

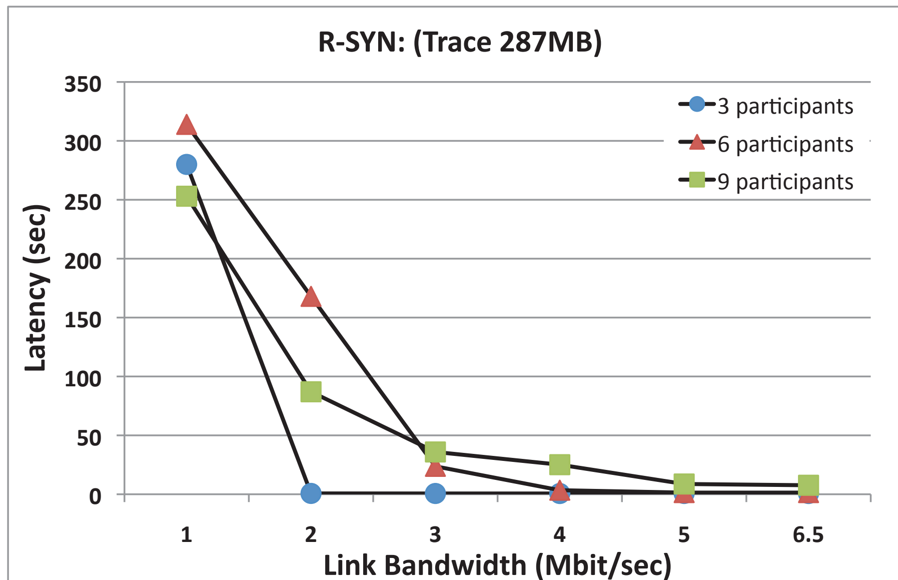


Figure 2.8: R-SYN detection latencies for different link bandwidths, in the presence of three, six, and nine SR members.

which is accurate for our tests as latency measures are in the order of seconds.

For detection latency tests we used trace 5 and changed the physical link bandwidths to Esper in order to show how detection latency is affected by available bandwidth. Links bandwidth is controlled by the WANem emulator. We varied the bandwidth with values ranging from 1 Mbit/s to 6.5 Mbit/s. Figure 2.8 shows the average detection latency (in seconds) we obtained in different runs of the algorithm.

For reasonable link bandwidths of large-scale deployments (between 3 Mbit/s and 6.5 Mbit/s), the R-SYN algorithm shows acceptable detection latencies for the inter-domain port scan application (latencies vary between 0.6 to 35 seconds). In addition, results show that when the collaborative system is formed by a higher number of SR members (e.g., nine), detection latencies are better than those obtained with smaller SRs. This is principally caused by the larger amount of data available when the number of SR members increases: more data allow to detect the scanners more quickly. In contrast, when three or six SR members are present, we need to wait more in order to achieve the final result of the computation. These results also indicate that available bandwidth can become a bottleneck as SR members increase in number or in the rate of provided events.

2.7.2 R-SYN in Esper vs R-SYN in Storm

We carried out some experiments to compare the performance of R-SYN implemented in Esper against those obtained from its implementation in Storm. We setup two SRs with six members each (and six machines, one for each member): one SR with Esper as processing engine and the other SR with Storm.

Detection Latency We first evaluated the detection latency using trace 2 (see Table 2.1); the results we collected show that Storm exhibits on average lower latencies compared to Esper (see Figure 2.9). Although the processing of a single event in Storm includes message passings between distinct physical machines due to its distributed setting, part of the computation can be performed in parallel, in particular the processing of the bolts *Half Open*, *Vertical Scan*, *Horizontal Scan* and *Entropy*. This explains why we can observe detection latencies lower than those provided by Esper.

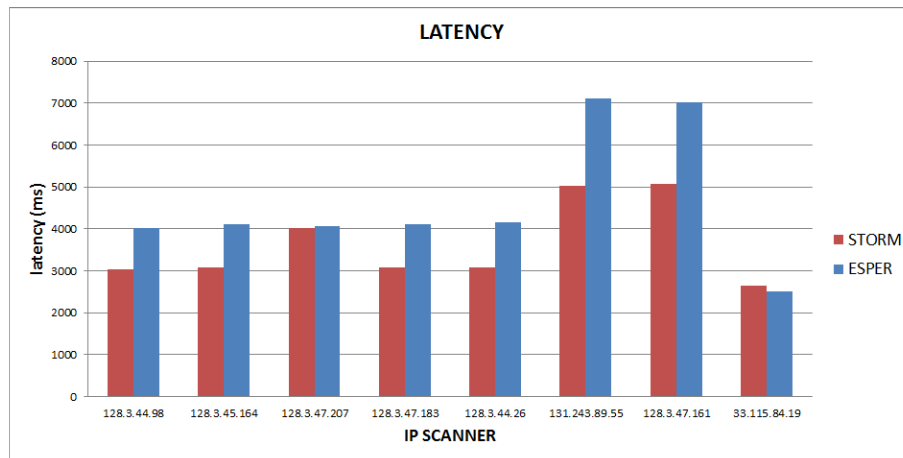


Figure 2.9: Detection latency comparison between Esper and Storm implementations of R-SYN algorithm

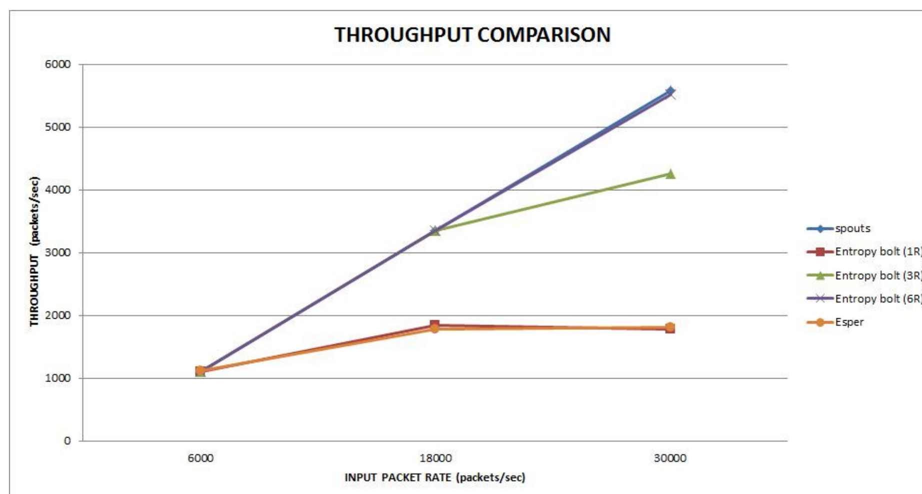


Figure 2.10: Throughput comparison between Esper and Storm implementations of R-SYN algorithm

Event Throughput We then evaluated the throughput of the two engines with respect to the input event rate generated by the sources. By varying the rate of the packets that are sniffed by the edge components (i.e., the SR gateways in the Esper-based SR and the spouts in the Storm-based SR), we measured the number of events per second processed by the engine. For the Storm-based SR, we chose the *Entropy* bolt for our measurements since it is the most computationally demanding Component, and the overall throughput cannot be higher than that provided by it. For the Esper-based SR, we counted the number of updates per second performed by the listener in charge of computing the entropy.

In order to analyze the scaling capabilities of Storm, we setup distinct configurations of the Topology, changing the number of replicas for the *Entropy* bolt. Figure 2.10 shows the results we obtained. We also included the throughput of the spouts to put an upper bound to the throughput achievable by the two engines. Comparing Esper with the configuration of Storm with a single replica of the Entropy bolt, it emerges that both engines are not able to sustain the input load as this increases, and the exhibited throughputs are almost equal. The configuration with three replicas can sustain higher input loads; however, it fails when the global input packet rate reaches 30 K packet/s. With six bolts, which means one Entropy bolt for each spout, Storm is able to sustain the load generated by the spouts.

These results highlight the ability of Storm to scale out on demand just by increasing the number of

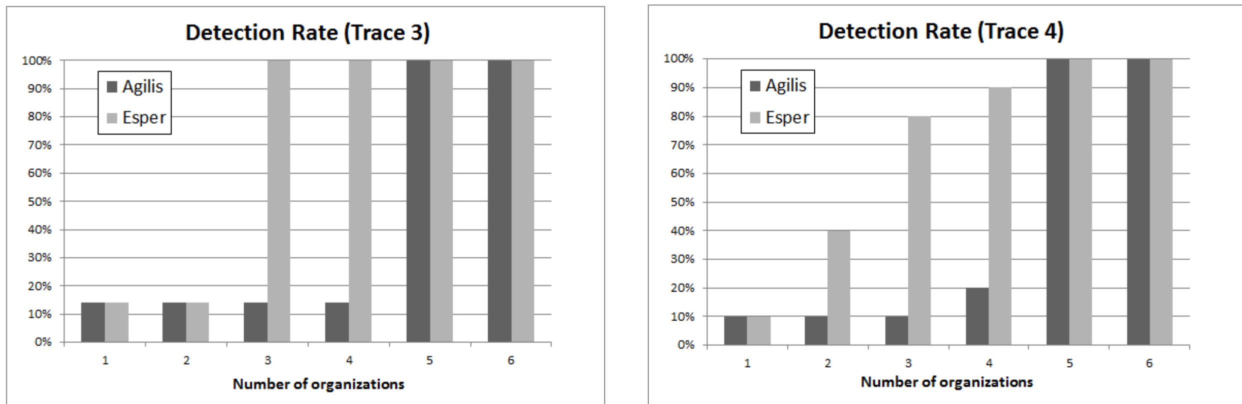


Figure 2.11: Port scan detection rate in a Collaborative Environment with 6 organizations. We use the trace 3 and 4, respectively. Bars show the number of packets required to achieve 100% detection rate.

replicas of bottleneck bolts, and by also providing the required physical resources where replicas can be deployed.

2.7.3 R-SYN in Esper vs R-SYN in Agilis

We have carried out an experimental evaluation of the Agilis-based R-SYN implementation and compared with the Esper-based one. This evaluation aims at assessing the detection accuracy in recognizing stealthy port scans, and the detection latency on varying available physical bandwidth among SR members.

The layout of the components on the cluster in case of Esper consisted of one VM dedicated to host the Esper CEP engine. The other six VMs represented the resources made available by six simulated organizations participating in the Collaborative Environment. Each resource hosted the Gateway component.

In contrast, the layout of the Agilis components on the cluster consisted of one VM dedicated to host all of the Agilis management components: JobTracker and WXS Catalog Server. The other six VMs represented a single Agilis Site and hosted one TaskTracker, one WXS container and one Agilis Gateway.

In order to emulate a large-scale deployment, all the VMs were connected through WANem.

Detection Accuracy

We used two intrusion traces in order to test the effectiveness of our prototypes in detecting malicious port scan activities, in the specific we used trace 3 and 4 (see Table 2.1). In order to assess the accuracy of the two implementations, we partitioned the traces simulating the presence of six organizations participating to the collaborative processing systems. We observed that none of the two systems introduced errors in the detection of port scanners. In other words, the False Positive Rate was always 0% in our tests. Results are shown in Figure 2.11.

The collaboration can be beneficial for sharpening the detection of port scanners. In both systems, augmenting the volume of data to be analyzed (i.e., augmenting the number of participants in the collaborative processing system) leads to an increase of the detection rate as computed above. However, the behavior of the two systems is different: in general, Esper achieves a higher detection rate with a smaller amount of data compared to Agilis. This is mainly caused by the two different approaches employed by the systems.

Detection Latency

In the current Agilis implementation, the Jaql queries cannot be made *persistent* yet; that is, when another computation has to be launched, an additional submission of the same collection of MapReduce jobs is required. This issue notably impacts on detection latency evaluation (see results below). To this end, we are

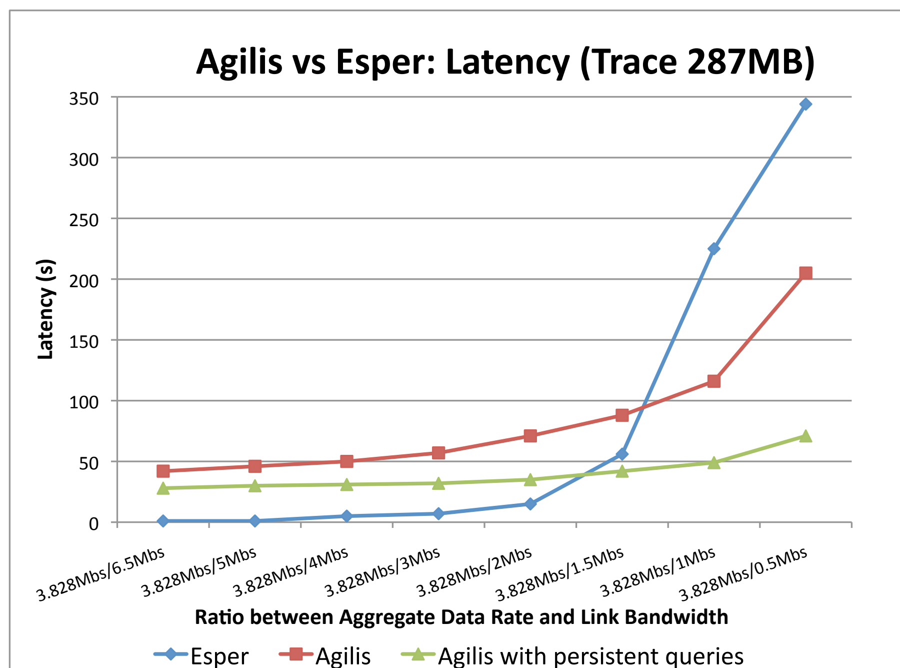


Figure 2.12: Scanner detection latency. It considers the ratio between the aggregate data rate coming from all organizations (3.828 Mbit/s) and the link bandwidth available at each organization site for the collaborative system. Collaboration is carried out by six organizations.

planning to extend Agilis in order to allow it to compile Jaql queries once and then keep running the queries continuously without the need to initialize them at every batch window. However, since such implementation at the time of this writing was not available, we estimated the detection latencies we could obtain in case of persistent Jaql queries. Such estimation is based on the following observations. Let us consider the detection of a certain scanner x: we can identify which iteration of the query execution has generated such a detection. Let i be this iteration. Since x has not been detected at iteration $i-1$, we know that the data required for the detection were not in the system at that time (otherwise the detection would have occurred), so there was no way for the system to detect x before iteration i started. This means that, even if we had the optimal Agilis implementation with no query initialization at each batch window, we couldn't detect x before iteration i started; however, we can expect to detect it before iteration i ends, thanks to the improvement in Agilis that could be obtained by using persistent queries. Owing to these observations, we estimated the latency as the average between the time the detection iteration begins and the time it ends.

Results For these tests we have used the trace 5 (see Table 2.1) and different physical link bandwidths in order to show in which setting one of the two systems can be preferable. Link bandwidth is controlled by the WANem emulator. We varied the physical link bandwidth with values ranging from 6.5 Mbit/s to 500 kbit/s, and we kept constant the aggregate data rate sniffed from the organizations networks. This value in our experiment was equal to 3.828 Mbit/s and is the sum of the rates of the data sniffed within each single organization. Figure 2.12 illustrates the results we have obtained.

From the assessment, it resulted that Esper itself is never a bottleneck. However, its centralized approach can become an issue when the ratio between the aggregate data rate and link bandwidth increases (i.e., the link bandwidth available at each site decreases). In the tests, we first observed the behavior of Agilis and Esper in a LAN environment characterized by a high available link bandwidth: Esper exhibits an average latency of 1 second whereas Agilis shows an average latency of 30 seconds using its current implementation, and 20 seconds of estimated latency in case of persistent queries. Hence, Esper in a LAN environment

notably outperforms Agilis.

The situation changes in case of large-scale deployments. Figure 2.12 shows the average detection latency in seconds we have obtained in different runs of the two systems. When the link bandwidth starts decreasing, the latency of Esper rapidly increases. This is mainly caused by the large amount of data sent to Esper. In contrast, in case of Agilis, the distributed approach is advantageous: the locality-awareness mechanisms employed by Agilis (i.e., processing is carried out at the edges locally to each Gateway), allows to exchange smaller amounts of data. Therefore, when the link bandwidth decreases, the detection latency exhibited by Agilis is significantly lower than the one provided by Esper (right side of Figure 2.12). This is particularly evident in case of Agilis estimated latency (i.e., the latency that could be obtained in case of persistent queries). Therefore, Agilis scales better in terms of large-scale deployments, and also in terms of number of participants. Indeed, an increase of the number of participants leads to an increase of the aggregate data rate, and the systems exhibit the detection latencies shown on the right side of Figure 2.12.

Part II

Enhancing Technologies for Collaborative Environments

Chapter 3

Adaptive Scheduling in Storm

In Chapter 2 we saw distinct technologies for carrying out processing. Among them, *Storm* proved to have relevant strengths with respect to Esper for what concerns the online processing. At this regard, we want to investigate ways to enhance Storm even more.

When a Topology is submitted to Storm, it schedules its execution in the cluster, i.e., it assigns the execution of each Executor of each spout and bolt to one of the nodes forming the cluster (see Section 2.5.1). Similarly to batch data analysis frameworks like Hadoop, Storm performances are not generally limited by computing power or available memory as new nodes can always be added to a cluster. In order to leverage the available resources, Storm is equipped with a *default scheduler* that evenly distributes the execution of Topology Executors on available nodes using a round-robin strategy. This simple approach aims to avoid the appearance of computing bottlenecks due to resource overloading caused by skews in load distribution. However, it does not take into account the cost of moving events through network links to let them traverse the expected sequence of bolts defined in the Topology. This latter aspect heavily impacts on the average event processing latency, i.e., how much time is needed for an event injected by a spout to traverse the Topology and be thus fully processed, which is a fundamental metric to evaluate the responsiveness to incoming stimuli.

In this chapter, we target the design and implementation of two general purpose Storm schedulers which could be leveraged by applications to improve their performances. Differently from the default scheduler, the ones introduced in this chapter aim at reducing the average event processing latency by adapting the schedule to specific application characteristics [1]. The rationale behind both schedulers is summarized by these points: (i) identifying potential *hot edges* of the Topology, i.e., edges traversed by a high rate of events, and (ii) mapping a hot edge to a fast inter-process channel rather than to a slow network link, for example by scheduling the execution of the Executors connected by that hot edge on the same cluster node. This rationale also has to take into account that processing resources have limited capabilities to not be exceeded in order to avoid an undesired explosion of the processing time experienced at each Topology Component. Such pragmatic strategy has the advantage of being practically workable and providing better performance.

The two general purpose schedulers introduced in this chapter differ each other on the way they identify hot edges in the Topology. The first scheduler, named *offline*, simply analyzes the Topology graph and identifies possible sets of Components to be scheduled on a same node by looking at how they are connected. This approach is simple and has no overhead on the application with respect to the default Storm scheduler (except for negligible increased delay when the schedule is calculated), but it is oblivious with respect to the application workload: it could decide to schedule two Executors on a same node even if the rate of events that will traverse the edge connecting them will be very small. The second scheduler, named *online*, takes this approach one step further by monitoring the effectiveness of the schedule at runtime and re-adapting it for a performance improvement when it sees fit. Monitoring is performed at runtime on the scheduled

[1] The source code of the two schedulers can be found at `http://www.dis.uniroma1.it/~midlab/software/storm-adaptive-schedulers.zip`

Topology by measuring the amount of traffic among its Executors. Whenever there is the possibility for a new schedule to reduce the inter-node network traffic, the new schedule is calculated and applied on the cluster.The online scheduler thus provides adaptation to the workload at the cost of a more complex architecture. We have tested the performance of our general purpose schedulers by implementing them on Storm and by comparing the schedules they produce with those produced by the default Storm scheduler. The tests have been conducted both on a synthetic workload and on a real workload publicly released for the *DEBS 2013 Grand Challenge*. The results show how the proposed schedulers consistently deliver better performances with respect to the default one, promoting them as a viable alternative to more expensive and complex ad-hoc schedulers. In particular, tests performed on the real workload show a 20% to 30% performance improvement on average event processing latency for the online scheduler with respect to the default one, then proving the effectiveness of the proposed scheduling approach.

After presenting the related work (see Section 3.1) and describing in details how scheduling works in Storm (see Section 3.2), an exhaustive explanation of the design of our two schedulers is illustrated in Section 3.3. Finally, Section 3.4 includes all the evaluations carried out to test their effectiveness.

The contents of this chapter are based on [32].

3.1 Related Work

There exist alternative distributed event and stream processing engines besides Storm. Similarly to Storm, these engines allow to model the computation as continuous queries that run uninterruptedly over input event streams. The queries in turn are represented as graphs of interconnected operators that encapsulate the logic of the queries.

System S [30] is a stream processing framework developed by IBM. A query in System S is modeled as an Event Processing Network (EPN) consisting of a set of Event Processing Agents (EPAs) that communicate each other to carry out the required computation. The similarity with Storm is very strong, indeed EPNs can be seen as the equivalent of Storm Topologies and EPAs as the analogous of bolts. S4 [121] is a different stream processing engine, developed by Yahoo, where queries are designed as graphs of Processing Elements (PEs) which exchange events according to queries' specification. Again, the affinity with Storm is evident as PEs definitely correspond to bolts.

Another primary paradigm of elaboration in the scope of Big Data is the batch oriented MapReduce [68] devised by Google, together with its main open source implementation Hadoop [150] developed by Apache. The employment of a batch approach hardly adapts to the responsiveness requirements of today's applications that have to deal with continuous streams of input events, but there are scenarios [35] where it still results convenient adopting such an approach where the limitations of the batch paradigm are largely offset by the strong characteristics of scalability and fault tolerance of a MapReduce based framework.

An attempt to address the restrictions of a batch approach is MapReduce online [63], that is introduced as an evolution of the original Hadoop towards a design that fits better to the requirements of stream based applications.

Other works [113, 112, 53] try to bridge the gap between continuous queries and MapReduce paradigm by proposing a stream based version of the MapReduce approach [23] where events uninterruptedly flow among the map and reduce stages of a certain computation without incurring in the delays typical of batch oriented solutions.

The problem of efficiently schedule operators in CEP engines has been tackled in several works. Cammert et al. [54] investigated how to partition a graph of operators into subgraphs and how to assign each subgraph to a proper number of threads in order to overcome common complications concerning threads overhead and operators stall. Moakar et al. [116] explored the question of scheduling for continuous queries that exhibit different classes of characteristics and requirements, and proposed a strategy to take into account such heterogeneity while optimizing response latency. Sharaf et al. [136] focus on the importance of

scheduling in environments where the streams to consume are quite heterogeneous and present high skews; they worked on a rate-based scheduling strategy which accounts for the specific features of the streams to produce effective operators schedules.

Hormati et al. [80] and Suleman et al [142] propose works conceived for multi core systems rather than clusters of machines. Differently from our solutions, the former aims at maximizing the throughput by combining a preliminary static compilation with adaptive dynamic changes of the configuration triggered by variations in resource availability. The latter focuses on chain topologies with the goal of minimizing execution time and number of used cores by tuning the parallelism of bottleneck stages in the pipeline. On the other hand, Pietzuch et al. [129] are concerned with operator placement within pools of nodes of wide-area overlay networks. They proposed a stream-based overlay network in charge of reducing the latency and leveraging possible reuse of operators. The solution proposed in this section, differently from [129], does not consider the efficient use of the network as a first class goal, nor does consider possible operator reuse. SODA [152] is an optimized scheduler specific for System S [30] which takes into account several distinct metrics in order to produce allocations that optimize an application-specific measure ("importance") and maximize nodes and links usage. One of the assumptions that drives their scheduling strategy is that the offered load would far exceed system capacity much of the time, an assumption that cannot be made for Storm applications. Xing et al. [157] presented a methodology to produce balanced operator mapping plans for Borealis [25]. They only consider node load and actually ignore the impact of network traffic. In a later work, Xing et at. [156] described an operator placement plan that is resilient to changes in load, but makes the relevant assumption that operators cannot be moved at runtime.

3.2 Default and Custom Scheduler

The Storm default scheduler is called *EvenScheduler*. It enforces a simple round-robin strategy with the aim of producing an even allocation. In the first phase, it iterates through the Topology Executors, grouped by Component, and allocates them to the configured number of Workers in a round-robin fashion. In the second phase, the Workers are evenly assigned to Worker Nodes, according to the slot availability of each Worker Node. This scheduling policy produces Workers that are almost assigned an equal number of Executors, and distributes such Workers over the Worker Nodes at disposal so that each one node almost runs an equal number of Workers.

Storm allows implementations of custom schedulers in order to accommodate for users' specific needs. In the general case, the custom scheduler takes as input the structure of the Topology (provided by Nimbus service), represented as a weighted graph $G(V,T), w$, and set of user-defined additional parameters (α, β, ...). The custom scheduler computes a *deployment plan* which defines both the assignment of Executors to Workers and the allocation of Workers to slots. Storm API provides the *IScheduler* interface to plug-in a custom scheduler, which has a single method *schedule* that requires two parameters. The first is an object containing the definitions of all the Topologies currently running, and including Topology-specific parameters provided by who submitted the Topology, which enables to provide the previously mentioned user-defined parameters. The second parameter is an object representing the physical cluster, with all the required information about Worker Nodes, slots and current allocations. A Storm installation can have a single scheduler, which is executed periodically or when a new Topology is submitted.

3.2.1 State Management

Currently, Storm doesn't provide any mean to manage the movement of stateful Components, meaning that it is up to the developer to implement application-specific mechanisms to save any states to storage and to properly reload them once a rescheduling is completed. The problem of state management for stateful operators impacts on the following key operations carried out on a cluster hosting a distributed processing platform:

(i) moving operators (i.e., Storm Executors) to optimize the performance;

(ii) scaling in/out operators to adapt to traffic characteristics (scale out to avoid bottlenecks, scale in to save resources);

(iii) reallocating operators that were executing on a failed machine.

While in the first case the state simply has to be moved together with the operator, in the second case the state has to be repartioned among the new set of replicas. In the third case, the state cannot be stored or replicated where the operator was executing, otherwise it is likely to be lost.

A comprehensive work that addresses the issue of state management is presented in [55], where the state of an operator is divided into three distinct pieces

- *processing state*: the state specific of the computation carried out by the operator;
- *buffer state*: the content of output buffers, which contain all the tuples sent or to be sent to downstream operators that have not been acknowledged yet;
- *routing state*: the routing rules for dispatching tuples to the right replica of a downstream operator.

Such state is explicitly exposed to the processing platform, which performs periodical backup of the processing and buffer state of an operator to one of its upstream operators. In case of rescheduling or scaling out or fault recovery of an operator, the new configuration (i.e., the new number of replicas decided for that operator) is used to properly update the routing state, then the most recent available processing state is retrieved and repartitioned accordingly among the new set of operator replicas. Finally, first of all the tuples in the backed buffer state, and then the tuples in upstream operators buffer state, are replayed in order to bring the processing state back to the situation preceding the reconfiguration.

In our work, we don't address the state management issue and thereby don't take into account the overheads due to keeping stateful operators consistent after a rescheduling.

3.3 Adaptive Scheduling

The key idea of the scheduling algorithms we propose is to take into account the communication patterns among Executors in order to place in the same slot pairs of Executors that communicate with high frequency. In Topologies where the computation latency is dominated by tuples transfer time, limiting the number of tuples that have to be sent and received through the network can contribute to improve the performance. Indeed, while sending a tuple to an Executor located in the same slot simply consists in passing a pointer, delivering a tuple to an Executor running inside another slot or deployed in a different Worker Node involves much larger overheads.

We developed two distinct algorithms based on such idea. One looks at how Components are interconnected within the Topology to determine what are the Executors that should be assigned to the same slot. The other relies on the monitoring at runtime of the traffic of exchanged tuples among Executors. The former is less demanding in terms of required infrastructure and in general produces lower quality schedules, while the latter needs to monitor at runtime the cluster in order to provide more precise and effective solutions, so it entails more overhead at runtime for gathering performance data and carrying out re-schedulings.

In this work we consider a Topology structured as a directed acyclic graph [45] where an upper bound can be set on the length of the path (measured in number of hops) that any input tuple follows from the emitting spout to the bolt that concludes its processing. This means that we don't take into account Topologies containing cycles, for example back propagation streams in online machine learning algorithms [47].

A Storm cluster includes a set $\mathcal{N} = \{n_i\}$ of Worker Nodes ($i = 1...N$), each one configured with S_i available slots ($i = 1...N$). In a Storm cluster, a set $\mathcal{T} = \{t_i\}$ of Topologies are deployed ($i = 1...T$), each one

configured to run on at most W_i Workers ($i = 1...T$). A Topology t_i consists of a set C_i of interconnected Components ($i = 1...T$). Each Component c_j ($j = 1...C_i$) is configured with a certain level of parallelism by specifying two parameters: (i) the number of Executors, and (ii) the number of Tasks. A Component is replicated on many Tasks that are executed by a certain number of Executors. For each Component, the number of Tasks is greater than or equal to the number of Executors. An Executor can only execute Tasks of a single Component. Similarly, a Worker can only execute Executors of a single Topology. A Topology t_i consists of E_i Executors $e_{i,j}$ ($i = 1...T$, $j = 1...E_i$).

The actual number of Workers required for a Topology t_i is $\min(W_i, E_i)$. The total number of Workers required to run all the Topologies is $\sum_{i=1}^{T} \min(W_i, E_i)$. A schedule is possible if enough slots are available, that is $\sum_{i=1}^{N} S_i \geq \sum_{i=1}^{T} \min(W_i, E_i)$.

Both the algorithms can be tuned using a parameter α that controls the balancing of the number of Executors assigned per slot. In particular, α affects the maximum number M of Executors that can be placed in a single slot. The minimum value of M for a Topology t_i is $\lceil E_i/W_i \rceil$, which means that each slot roughly contains the same number of Executors. The maximum number of M corresponds to the assignment where all the slots contain one Executor, except for one slot that contains all the other Executors, so its value is $E_i - W_i + 1$. Allowed values for α are in $[0, 1]$ range and set the value of M within its minimum and maximum: $M(\alpha) = \lceil E_i/W_i \rceil + \alpha(E_i - W_i + 1 - \lceil E_i/W_i \rceil)$.

3.3.1 Topology-based Scheduling

The *offline scheduler* examines the structure of the Topology in order to determine the most convenient slots where to place Executors. Such a scheduling is executed before the Topology is started, so neither the load nor the traffic are taken into account, and consequently no constraint about memory or CPU is considered. Not even the stream groupings configured for inter-Component communications are inspected because the way they impact on inter-node and inter-slot traffic can be only observed at runtime. Not taking into account all these points obviously limits the effectiveness of the offline scheduler, but on the other hand this enables a very simple implementation that still provides good performance, as will be shown in Section 3.4. A partial order among the Components of a Topology can be derived on the basis of streams configuration. If a Component c_i emits tuples on a stream that is consumed by another Component c_j, then we have $c_i < c_j$. If $c_i < c_j$ and $c_j < c_k$ hold, then $c_i < c_k$ holds by transitivity. Such order is partial because there can be pairs of Components c_i and c_j such that neither $c_i > c_j$ or $c_i < c_j$ hold. Since we deal with acyclic Topologies, we can always determine a linearization ϕ of the Components according to such partial order. If $c_i < c_j$ holds, then c_i appears in ϕ before c_j. If neither $c_i < c_j$ nor $c_i > c_j$ hold, then they can appear in ϕ in any order. The first element of ϕ is a spout of the Topology. The heuristic employed by the offline scheduler entails iterating ϕ and, for each Component c_i, placing its Executors in the slots that already contain Executors of the Components that directly emit tuples towards c_i. Finally, the slots are assigned to Worker Nodes in a round-robin fashion.

A possible problem of this approach concerns the possibility that not all the required Workers get used because, at each step of the algorithm, the slots that are empty get ignored since they don't contain any Executor. The solution employed by the offline scheduler consists in forcing to use empty slots at a certain point during the iteration of the Components in ϕ. When to start considering empty slots is controlled by a tuning parameter β, whose value lies in $[0, 1]$ range: during the assignment of Executors for the i-th Component, the scheduler is forced to use empty slots if $i > \lfloor \beta \cdot C_i \rfloor$. For example, if traffic is likely to be more intense among upstream Components, then β should be set large enough such that empty slots get used when upstream Components are already assigned.

3.3.2 Traffic-based Scheduling

The *online scheduler* produces assignments that reduce inter-node and inter-slot traffic on the basis of the communication patterns among Executors observed at runtime. The goal of the online scheduler is to allocate Executors to nodes so as to minimize the inter-node traffic and satisfy the constraints on (i) the number of Workers each Topology has to run on ($\min(W_i, E_i)$), (ii) the number of slots available on each Worker node (S_i) and (iii) the computational power available on each node (see Section 3.3.2). Such scheduling has to be performed at runtime so as *to adapt the allocation to the evolution of the load in the cluster*. Figure 3.1 shows the integration of our online scheduler within the Storm architecture. Notice that the performance log depicted in the picture is just a stable buffer space where data produced by monitoring Components running at each slot can be placed before it gets consumed by the custom scheduler on Nimbus. The custom scheduler can be periodically run to retrieve this data and check if a more efficient schedule can be deployed.

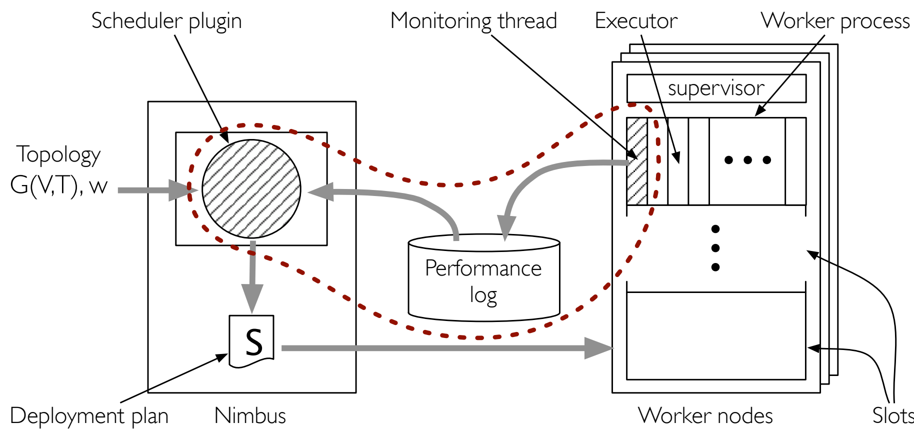


Figure 3.1: A Storm cluster with the Nimbus process controlling several Worker Nodes. Each Worker Node hosts a *Supervisor* and a number of *Workers* (one per slot), each running a set of *Executors*. The dashed red line shows Components of the proposed solution: the offline scheduler only adds the plugin to the Nimbus process, while the online scheduler also adds the performance log and monitoring processes.

Measurements

When scheduling the Executors, taking into account the computational power of the nodes is needed to avoid any overload, and this requires some measurements. We use the CPU utilization to measure both the load a node is subjected to (due to Worker processes) and the load generated by an Executor. Using the same metric allows us to make predictions on the load generated by a set of Executors on a particular node. We also want to deal with clusters comprising heterogeneous nodes (different computational power), so we need to take into account the speed of the CPU of a node in order to make proper predictions. For example, if an Executor is taking 10% CPU utilization on a 1GHz CPU, then migrating such Executor on a node with 2GHz CPU would generate about 5% CPU utilization. For this reason, we measure the load in Hz. In the previous example, the Executor generates a load of 100MHz (10% of 1GHz).

We use L_i to denote the load the node n_i is subjected to due to the Executors. We use $L_{i,j}$ to denote the load generated by Executor $e_{i,j}$. We use CPU_i to denote the speed of the CPU of node n_i (number of cores multiplied by single core speed).

CPU measurements have been implemented by leveraging standard Java API for retrieving at runtime the CPU time for a specific thread (*getThreadCpuTime(threadID)* method of *ThreadMXBean* class). With these measures we can monitor the status of the cluster and detect any imbalance due to node CPU overloads. We can state that if a node n_i exhibits a CPU utilization trend such that $L_i \geq B_i$ for more than X_i seconds,

then we trigger a rescheduling. We refer to B_i as the capacity (measured in Hz) and to X_i as the time window (measured in seconds) of node n_i. One of the goals of a scheduling is the satisfaction of some constraints on nodes load.

We don't consider the load due to I/O bound operations such as reads/writes to disk or network communications with external systems like DBMSs. Event-based systems usually work with data in memory in order to avoid any possible bottleneck so as to allow events to flow along the operators network as fast as possible. This doesn't mean that I/O operations are forbidden, but they get better dealt with by employing techniques like executing them on a batch of events instead of on a single event, and caching data in main memory to speed them up.

In order to minimize the inter-node traffic, the volumes of tuples exchanged among Executors have to be measured. We use $R_{i,j,k}$ to denote the rate of the tuples sent by Executor $e_{i,j}$ to Executor $e_{i,k}$, expressed in tuples per second ($i = 1...T; j,k = 1...E_i; j \neq k$). Summing up the traffic of events exchanged among Executors deployed on distinct nodes, we can measure the total inter-node traffic. Once every P seconds, we can compute a new scheduling, compare the inter-node traffic such scheduling would generate with the current one and, in case a reduction of more than $R\%$ in found, trigger a rescheduling.

Formulation

Given the set of nodes $\mathcal{N} = \{n_i\}$ ($i = 1...N$), the set of Workers $\mathcal{W} = \{w_{i,j}\}$ ($i = 1...T,\ j = 1...\min(E_i, W_i)$) and the set of Executors $\mathcal{E} = \{e_{i,j}\}$ ($i = 1...N, j = 1...E_i$), the goal of load balancing is to assign each Executor to a slot of a node. The scheduling is aimed at computing (i) an allocation $A_1 : \mathcal{E} \to \mathcal{W}$, which maps Executors to Workers, and (ii) an allocation $A_2 : \mathcal{W} \to \mathcal{N}$, which maps Workers to nodes.

The allocation has to satisfy the constraints on nodes capacity

$$\forall k = 1...N \sum_{\substack{A_2(A_1(e_{i,j}))=n_k \\ i=1...T; j=1...E_i}} L_{i,j} \leq B_k \tag{3.1}$$

as well as the constraints on the maximum number of Workers each Topology can run on

$$\begin{aligned} &\forall i = 1...T \\ &|\{w \in \mathcal{W} : A_1(e_{i,j}) = w, j = 1...E_i\}| = \min(E_i, W_i) \end{aligned} \tag{3.2}$$

The objective of the allocation is to minimize the inter-node traffic

$$\min \sum_{\substack{j,k: A_2(A_1(e_{i,j})) \neq A_2(A_1(e_{i,k})) \\ i=1...T; j,k=1...E_i}} R_{i,j,k} \tag{3.3}$$

Algorithm

The problem formulated in Section 3.3.2 is known to be NP-complete [46, 97]. The requirement of carrying the rebalance out at runtime implies the usage of a quick mechanism to find a new allocation, which in turn means that some heuristic has to be employed. The following algorithm is based on a simple greedy heuristic that place Executors to node so as to minimize inter-node traffic and avoid load imbalances among all the nodes. It consists of two consecutive phases.

In the first phase, the Executors of each Topology are partitioned among the number of Workers the Topology has been configured to run on. The placement is aimed to both minimize the traffic among Executors of distinct Workers and balance the total CPU demand of each Worker.

In the second phase, the Workers produced in the first phase have to be allocated to available slots in the cluster. Such allocation still has to take into account both inter-node traffic, in order to minimize it, and node load, so as to satisfy load capacity constraints.

Algorithm: 1 Online Scheduler

Data:
$\mathcal{T} = \{t_i\}$ $(i = 1...T)$: set of Topologies
$\mathcal{E} = \{e_{i,j}\}$ $(i = 1...T; j = 1...E_i)$: set of Executors
$\mathcal{N} = \{n_i\}$ $(i = 1...N)$: set of nodes
$\mathcal{W} = \{w_{i,j}\}$ $(i = 1...T; j = 1...\min(E_i, W_i))$: set of Workers
$L_{i,j}$ $(i = 1...T; j = 1...E_i)$: load generated by Executor $e_{i,j}$
$R_{i,j,k}$ $(i = 1...T; j, k = 1...E_i)$: tuple rate between Executors $e_{i,j}$ and $e_{i,k}$

begin
  // First Phase
  foreach *Topology* $t_i \in \mathcal{T}$ **do**
    // Inter-Executor Traffic for Topology t_i
    $IET_i \leftarrow \{\langle e_{i,j}; e_{i,k}; R_{i,j,k}\rangle\}$ sorted descending by $R_{i,j,k}$ **foreach** $\langle e_{i,j}; e_{i,k}; R_{i,j,k}\rangle \in IET_i$ **do**
      // get least loaded Worker
      $w* \leftarrow \text{argmin}_{w_{i,x} \in \mathcal{W}} \sum_{A_1(e_{i,y})=w_{i,x}} L_{i,y}$ **if** !assigned($e_{i,j}$) **and** !assigned($e_{i,k}$) **then**
        // assign both Executors to w^*
        $A_1(e_{i,j}) \leftarrow w^*$ $A_1(e_{i,k}) \leftarrow w^*$
      else
        // check the best assignment of $e_{i,j}$ and $e_{i,k}$ to the Workers that already
        // include either Executor and to w^* (at most 9 distinct assignments to consider)
        $\Pi \leftarrow \{w \in \mathcal{W} : A_1(e_{i,j}) = w \vee A_1(e_{i,k}) = w\} \cup \{w^*\}$ $best_w_j \leftarrow null$ $best_w_k \leftarrow null$ $best_ist \leftarrow MAX_INT$ **foreach** $\langle w_j, w_k\rangle \in \Pi^2$ **do**
          $A_1(e_{i,j}) \leftarrow w_j$ $A_1(e_{i,k}) \leftarrow w_k$ $ist \leftarrow \sum_{x,y:A_1(e_{i,x}) \neq A_1(e_{i,y})} R_{i,x,y}$ **if** $ist < best_ist$ **then**
            $best_ist \leftarrow ist$ $best_w_j \leftarrow w_j$ $best_w_k \leftarrow w_k$
          end
        end
        $A_1(e_{i,j}) \leftarrow best_w_j$ $A_1(e_{i,k}) \leftarrow best_w_k$
      end
    end
  end
  // Second Phase
  $IST \leftarrow \{\langle w_{i,x}; w_{i,y}; \gamma_{i,x,j}\rangle : w_{i,x}, w_{i,y} \in \mathcal{W}, \gamma_{i,x,j} = \sum_{A_1(e_{i,j})=w_{i,x} \wedge A_1(e_{i,k})=w_{i,y}} R_{i,j,k}\}$ sorted descending by $\gamma_{i,x,j}$ **foreach** $\langle w_{i,x}; w_{i,y}; \gamma_{i,x,j}\rangle$ **do**
    $n* \leftarrow \text{argmin}_{n \in \mathcal{N}} \sum_{A_2(A_1(e_{i,y}))=n} L_{i,y}$ **if** !assigned($w_{i,x}$) **and** !assigned($w_{i,y}$) **then**
      $A_2(w_{i,x}) \leftarrow n^*$ $A_2(w_{i,y}) \leftarrow n^*$
    else
      // check the best assignment of $w_{i,x}$ and $w_{i,y}$ to the nodes that already
      // include either Worker and to n^* (at most 9 distinct assignments to consider)
      $\Xi \leftarrow \{n \in \mathcal{N} : A_2(w_{i,x}) = n \vee A_2(w_{i,y}) = n\} \cup \{n^*\}$ $best_n_x \leftarrow null$ $best_n_y \leftarrow null$ $best_int \leftarrow MAX_INT$ **foreach** $\langle n_x, n_y\rangle \in \Xi^2$ **do**
        $A_2(w_{i,x}) \leftarrow n_x$ $A_2(w_{i,y}) \leftarrow n_y$ $int \leftarrow \sum_{j,k:A_2(A_1(e_{i,j})) \neq A_2(A_1(e_{i,k}))} R_{i,j,k}$ **if** $int < best_int$ **then**
          $best_int \leftarrow int$ $best_n_x \leftarrow n_x$ $best_n_y \leftarrow n_y$
        end
      end
      $A_2(w_{i,x}) \leftarrow best_n_x$ $A_2(w_{i,y}) \leftarrow best_n_y$
    end
  end
end

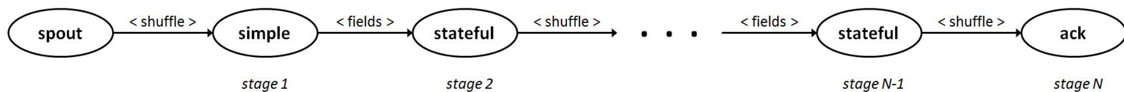


Figure 3.2: Reference Topology.

Algorithm 1 presents the pseudo-code for the online scheduler. This is an high level algorithm that doesn't include the implementation of many corner cases but shows instead the core of the heuristic.

In the first phase, for each Topology, the pairs of communicating Executors are iterated in descending order by rate of exchanged tuples. For each of these pairs, if both the Executors have not been assigned yet, then they get assigned to the Worker that is the least loaded at that moment. Otherwise, the set Π is built by putting the least loaded Worker together with the Workers where either Executor of the pair is assigned. Π can contain three elements at most: the least loaded and the two where the Executors in the pair are currently assigned. All the possible assignments of these Executors to these Workers are checked to find the best one, that is the assignment that produces the lowest inter-worker traffic. At most, there can be nine distinct possible assignments to check.

Similarly, in the second phase the pairs of communicating Workers are iterated in descending order by rate of exchanged tuples. For each pair, if both have not been allocated to any node yet, then the least loaded node is chosen to host them. If any or both have already been assigned to some other nodes, the set Ξ is built using these nodes and the least loaded one. All the possible allocations of the two Workers to the nodes in Ξ are examined to find the one that generates the minimum inter-node traffic. Again, there are at most nine distinct allocations to consider.

3.4 Experimental Evaluations

Our experimental evaluation aims at giving evidence that the scheduling algorithms we propose are successful at improving the performance on a wide range of Topologies. We first test their performance on a general Topology that captures the characteristics of a broad class of Topologies and show how the algorithms' tuning parameters impact on the efficiency of the computation, comparing the results with those obtained by using the default scheduler. Then, in order to evaluate our solution in a more realistic setting, we apply our scheduling algorithms to the DEBS 2013 Grand Challenge dataset [24] by implementing a subset of its queries. Performance were evaluated on two fundamental metrics: the average latency experienced by events to traverse the entire Topology and the average inter-node traffic incurred by the Topology at runtime.

All the evaluations were performed on a Storm cluster with eight Worker Nodes, each with five slots, and one further node hosting the Nimbus and Zookeeper services. Each node runs Ubuntu 12.04 and is equipped with 2x2.8 GHz CPUs, 3 GB of RAM and 15 GB of disk storage. The networking infrastructure is based on a 10 Gbit LAN. These nodes are kept synchronized with a precision of microseconds, which is sufficiently accurate for measuring latencies in the order of milliseconds. Such synchronization has been obtained by leveraging the standard NTP protocol to sync all the nodes with a specific node in the cluster.

3.4.1 Reference Topology

In this section, we analyze how the tuning parameters actually affect the behavior of scheduling algorithms and consequently the performance of a Topology. In order to avoid focusing on a specific Topology, we developed a reference Topology aimed at capturing the salient characteristics of many common Topologies.

Workload characteristics According to the kind of Topologies we deal with in this work (see Section 3.3), we consider acyclic Topologies where an upper bound can be set on the number of hops a tuple

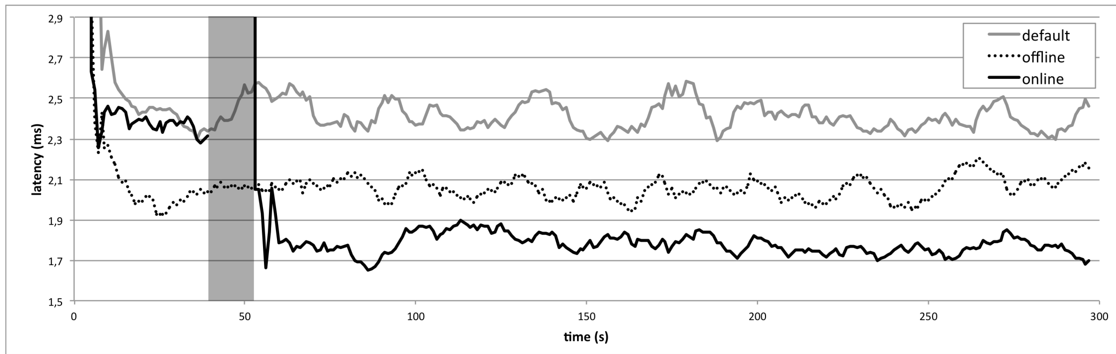


Figure 3.3: Tuple processing latency over time, for default, offline and online schedulers.

has to go through since it is emitted by a spout up to the point where its elaboration ends on some bolt. This property allows us to assign each Component c_i in the Topology a number $stage(c_i)$ that represents the length of the longest path a tuple must travel from any spout to c_i. By grouping Components in a same *stage*, we obtain a horizontal stratification of the Topology in stages, such that Components within the same stage don't communicate each other, and Components at stage i receive tuples from upstream Components at stages lower than i and send tuples to downstream Components at stages greater than i. This kind of stratification has been investigated in [99]. Recent works on streaming MapReduce [113, 112, 53] also focus on the possibility to model any computation as a sequence of alternated map and reduce stages, supporting the idea that a large class of computations can be structured as a sequence of consecutive stages where events always flow from previous to subsequent stages.

These considerations led us to propose the working hypothesis that a chain Topology can be employed as a meaningful sample of a wide class of possible Topologies. Chain Topologies are characterized by two parameters: (i) the number of stages, that is the horizontal dimension and (ii) the replication factor for each stage, that is the vertical dimension corresponding to the number of Executors for each Topology Component. We developed a *reference Topology* according to such working hypothesis. Taking inspiration from the MapReduce model [68], in the chain we alternate bolts that receive tuples using shuffle grouping (similar to mappers) to bolts that are fed through fields grouping (similar to reducers). In this way we can also take into account how the grouping strategy impacts on the generated traffic patterns.

Figure 3.2 shows the general structure of the reference Topology. It contains a single spout followed by an alternation of *simple* bolts, that receive tuples by shuffle grouping, and *stateful* bolts, that instead take tuples by fields grouping. Stateful bolts have been named so because their input stream is partitioned among the Executors by the value embedded in the tuples, and this would enable each Executor to keep some sort of state. In the last stage there is an *ack* bolt in charge of completing the execution of tuples.

Each spout Executor emits tuples containing an incremental numeric value at a fixed rate. Using incremental numeric values allows to evenly spread tuples among target Executors for bolts that receive input through fields grouping. Each spout Executor chooses its fixed rate using two parameters: the average input rate R in tuples per second and its variance V, expressed as the largest difference in percentage of the actual tuple rate from R.

The i-th spout Executor sets its tuple rate as $R_i = R(1-V(1-2\frac{i}{C_0-1}))$ where C_0 is the number of Executors for the first Component, that is the spout itself, and $i = 0, ..., C_0 - 1$. Therefore, each spout Executor emits tuples at a distinct fixed rate and the average of these rates is exactly R. In this way, the total input rate for the Topology can be controlled (its value is $C_0 \cdot R$) and a certain degree of irregularity can be introduced on traffic intensity (tuned by V parameter) in order to simulate realistic scenarios where event sources are likely to produce new data at distinct rates.

In order to include other factors for breaking the regularity of generated traffic patterns, bolts in the reference Topology have been implemented so as to forward the received value with probability $1/2$ and to

emit a different constant value (fixed for each Executor) the rest of the times. The traffic between Executors whose communication is setup using fields grouping is affected by this mechanism since it makes the tuple rates much higher for some Executor pairs. This choice models realistic situations where certain pairs of Executors in consecutive stages communicate more intensively than others.

Results The first experiments were focussed at evaluating the runtime behavior of the proposed schedulers with respect to the default one. Figure 3.3 reports how event latency evolves over time for an experiment. Each points reported in the figure represents the average of latencies for a ten events window. The reference Topology settings used in this test include seven stages, and variable replication factors: four for the spout, three for the bolts receiving tuples through shuffle grouping and two for the bolts receiving tuples through fields grouping.

At the beginning, all the schedules experience a short transient state where the system seems overloaded and this heavily impacts measured latencies. This transient period lasts approximately 15-20 seconds and is characterized by large latencies. In the subsequent 20 seconds time frame (up to second 40), it is possible to observe some characteristic behavior. The performance for all three schedules are reasonably stable, with both the default and online schedulers sharing similar figures, and the offline scheduler showing better results. This result proves how the Topology-based optimizations performed by the offline scheduler quickly pay-off with respect to the default scheduler approach. The online scheduler performance in this timeframe are instead coherent with the fact that this scheduler initializes the application using a schedule obtained by applying exactly the same approach used by the default scheduler (hence the similar performance). However, during this period the active scheduler collects performance measures that are used later (at second 40) to trigger a re-schedule. The shaded interval in the figure shows a “silence” period used by the active scheduler to instantiate the new schedule. The new schedule starts working at second 50 and quickly converges to performance that are consistently better with respect to both the default and the offline scheduler. This proves that the online scheduler is able to correctly identify cases where a different schedule can improve performance, and that this new schedule, built on the basis of performance indices collected at runtime, can indeed provide performance that surpasses a workload-oblivious schedule (like the one provided by the offline scheduler).

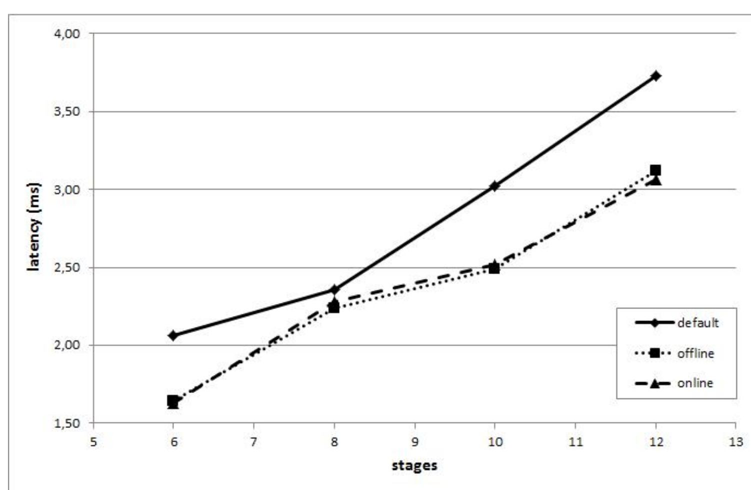


Figure 3.4: Average latency as the number of stages varies, with a replication factor of 2 for each stage.

We then evaluated how the proposed schedulers behave as the number of stages increases for different replication factors. As the number of stages increases, the latency obviously becomes larger as each tuple has to go through more processing stages. With a small replication factor traffic patterns among Executors are quite simple. In general, with a replication factor F, there are F^2 distinct streams among the Executors

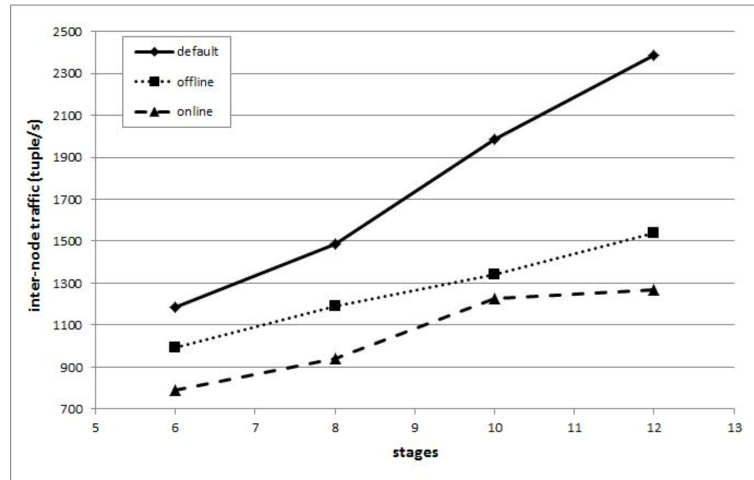


Figure 3.5: Average inter-node traffic as the number of stages varies, with a replication factor of 2 for each stage.

of communicating bolts because each Executor of a stage possibly communicate to all the Executors at the next stage. The offline scheduler does its best to place each of the F Executors of a bolt c_i where at least one of F Executors of the Component c_{i-1} has been already placed, which means that, in general, the latency of up to F streams out of F^2 is improved. Therefore, about $1/F$ of the tuples flowing among consecutive Components get sent within the same node with a consequent latency improvement. As the replication factor increases, the portion of tuples that can be sent locally gets lower and the effectiveness of the offline scheduler becomes less evident. The precise trend also depends on whether the streams that are optimized are intense or not; however, the offline scheduler is oblivious with respect to this aspect as it calculates the schedule before the Topology is executed. On the other hand, the online scheduler adapts to the actual evolution of the traffic and is able to identify the heaviest streams and consequently place Executors so as to make such streams local.

These evaluations have been carried out setting the parameters $\alpha = 0$ and $\beta = 0.5$, considering an average data rate $R = 100$ tuple/s with variance $V = 20\%$.

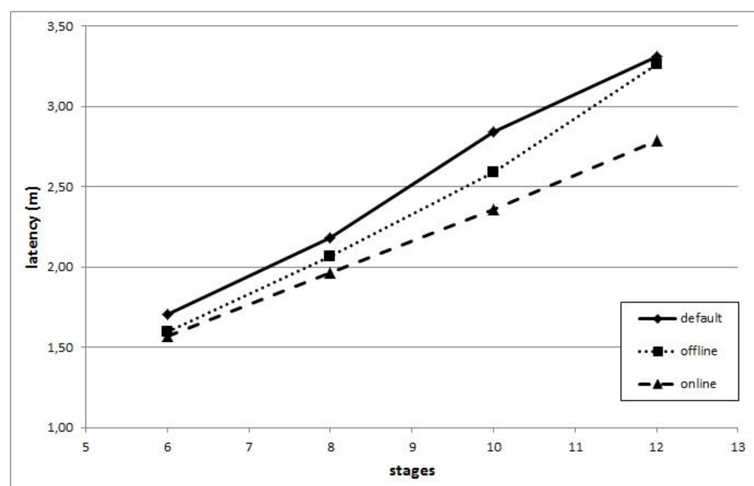


Figure 3.6: Average latency as the number of stages varies, with a replication factor of 4 for each stage, for default, offline and online schedulers.

Figures 3.4 and 3.5 report average latency and inter-node traffic for a replication factor two, i.e. each

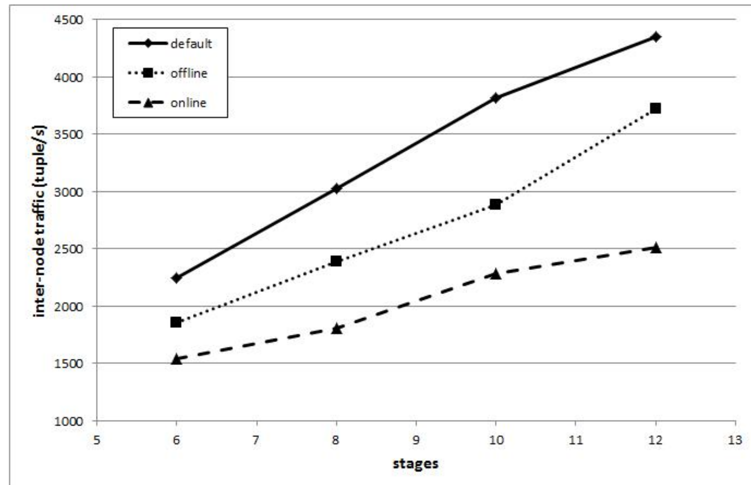


Figure 3.7: Average inter-node traffic as the number of stages varies, with a replication factor of 4 for each stage, for default, offline and online schedulers.

Component is configured to run on two Executors. Latencies for offline and online schedulers are close and always smaller with respect to the default scheduler. The low complexity of communication patterns allows for only a little number of improvement actions, which are leveraged by both the schedulers with the consequent effect that performance are very similar. The results about the inter-node traffic reflect this trend and also highlight that the online scheduler produces schedules with smaller inter-node traffic. Note that smaller inter-node traffic cannot always be directly related to a lower latency because it also depends on whether and to what extent the most intense paths in the Topology are affected.

Figures 3.6 and 3.7 show the same results for a replication factor set to four. While the online scheduler keeps providing sensibly lower latencies with respect to the default one, the effectiveness of offline scheduler begins to lessen due to the fact that it can improve only 4 out of 16 streams for each stage. Such a divergence between the performance of offline and online schedulers is also highlighted by the results on the inter-node traffic; indeed the online scheduler provides assignments that generate lower inter-node traffic.

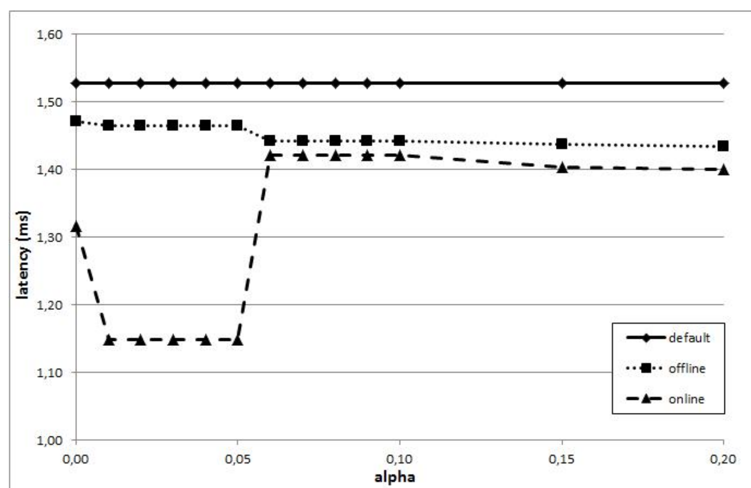


Figure 3.8: Average latency as α varies for default, offline and online schedulers.

We finally evaluated the impact of the α parameter on the schedules produced by our two algorithms. Figures 3.8 and 3.9 report results for a setting based on a five stages Topology with replication factor five,

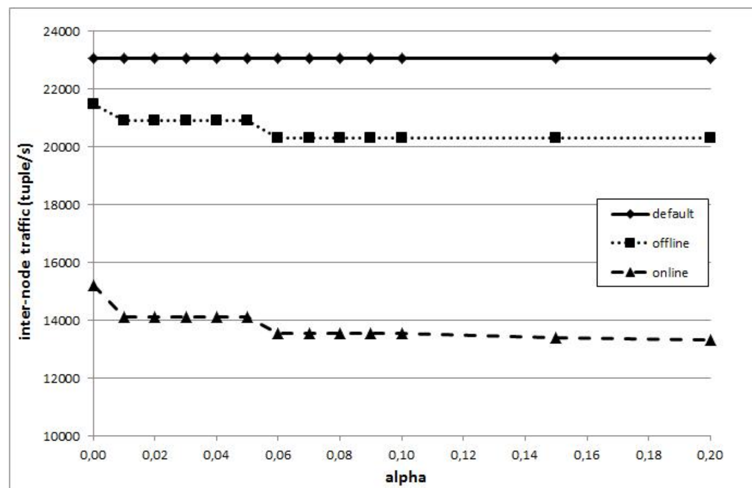


Figure 3.9: Average inter-node traffic as α varies for default, offline and online schedulers.

R = 1000 tuple/s and V = 20%.

In such setting, a Topology consists of 30 Executors, and we varied α from 0 to 0.2, which corresponds to varying the maximum number of Executors per slots from four to eight. The results show that the offline scheduler slightly keeps improving its performance as α grows, for what concerns both the latency and the inter-node traffic. The online scheduler provides its best performance when α is 0.05, that is when the upper bound on the number of Executors is five. Larger values for α provide larger latencies despite the inter-node traffic keeps decreasing. This happens because there is a dedicated thread for each Worker in charge of dequeuing tuples and sending them to the others Workers, and placing too many Executors in a single slot makes this thread a bottleneck for the whole Worker.

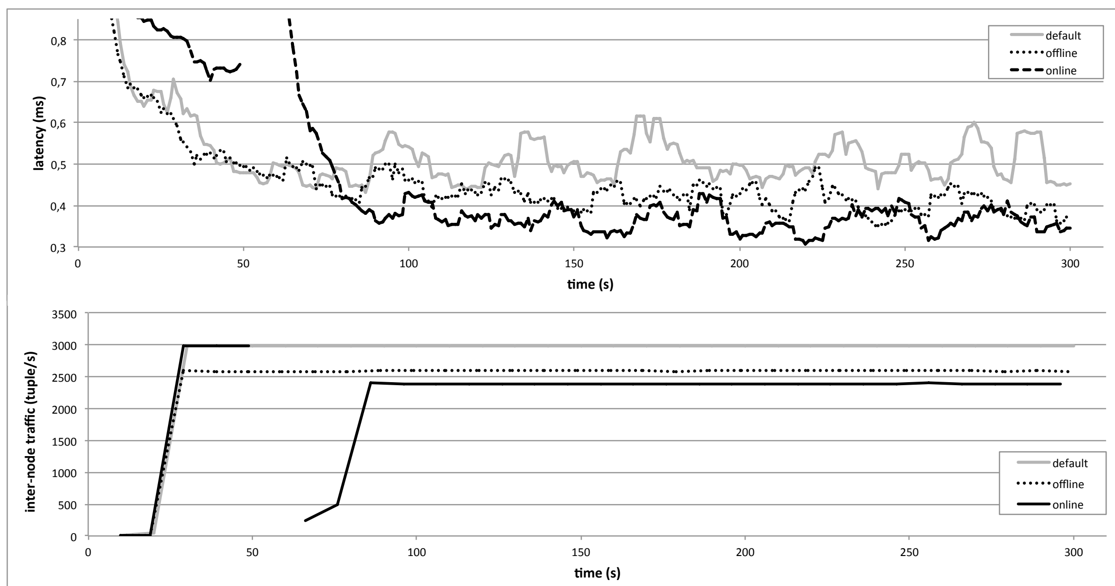


Figure 3.10: Latency (top) and traffic (bottom) over time for default, offline and online schedulers for the first query of DEBS 2013 Grand Challenge.

3.4.2 Grand Challenge Topology

We carried out some evaluations on the scenario described in the Grand Challenge of DEBS 2013 [24], by considering in particular a reduced version of the first query. In such scenario, sensors embedded in soccer players' shoes emit position and speed data at 200 Hz frequency. The goal of the first query is to perform a running analysis by continuously updating statistics about each player. The instantaneous speed is computed for each player every time a new event is produced by the sensors, a speed category is determined on the basis of computed value, then the global player statistics are updated accordingly. Such statistics include average speed, walked distance, average time for each speed category.

The Topology includes three Components: (i) a spout for the sensors (*sensor* Component in the figure, replication factor eight, with a total of 32 sensors to be simulated), (ii) a bolt that computes the instantaneous speed and receives tuples by shuffle grouping (*speed* Component in the figure, replication factor four), (iii) a bolt that maintains players' statistics and updates them as new tuples are received by fields grouping from the speed bolt (*analysis* Component in the figure, replication factor two).

Figure 3.10 shows how latency (top) and inter-node traffic (bottom) evolve over time. It can be noticed that most of the time the offline scheduler allows for lower latencies and lighter traffic than the default one, while the online scheduler in turn provides better performance than the offline one as soon as an initial transient period needed to collect performance indices is elapsed. The performance improvement provided by the online scheduler with respect to the default one can be quantified in this setting as oscillating between 20% and 30%. These results confirm that the optimizations performed by the online scheduler are effective also in real workloads where they provide noticeable performance improvements.

Chapter 4

Windowed Computations in Hadoop

Another technology employed in Chapter 2 for executing processing activities is *Hadoop*. Unlike Storm (see Chapter 3), which allows for continuous elaborations, Hadoop runs the processing in batches, periodically. We want to investigate possible ways to carry out windowed computations by using a batch-oriented processing platform.

Event processing is a constantly evolving research area which keeps growing to adapt to emerging technologies and paradigms [74]. Events produced by possibly different sources are usually collected in bunches delimited by time windows, then they are elaborated to produce other events as output. Several real applications require indeed to recognize particular patterns within specific time lapses or to produce periodic reports on what happened in precise time ranges. A relevant example for the first case is represented by Intrusion Detection Systems (IDSs), which keep monitoring network traffic searching for known malicious signatures in order to trace them and raise alerts whenever too many suspect activities occur within a defined time interval. Port scan detection techniques based on the activities observed in specific time windows are investigated in Chapter 2. In this scenario, where a large number of network probes can produce high rate event streams, it is advisable to adopt large time windows (hours, days) to catch *slow attacks*; at the same time, it is necessary to have frequent (every few seconds) analysis results to promptly react to alarms. An example of the second case is represented by algorithmic trading, an application scenario where result latency is critical to profitability. Such scenario is characterized by large data volumes (millions of trades per day), medium to large observation windows (hours, days) and frequent (down to a second) updates to quickly catch highly volatile financial opportunities. We refer to this kind of event processing as Time Window Based Computations (TWBCs).

Event processing engines managing TWBCs must cope with an ever increasing number of event sources and with continuously growing data rates. To keep up with this trend, processing engines must be able to manage *huge input data volumes*. Output of such computations are often used to take the best decision about some next action to be performed. Therefore, it is crucial to get these results as soon as possible, otherwise they are likely to become obsolete before they can be actually used. To cope with this requirement, processing engines must be *timely* in the production of their output.

Event processing engines can adopt two possible approaches for TWBCs with respect to the relationship between when events arrive and when they are elaborated. If events are processed as soon as they enter the engine, we talk about *online* event processing. Conversely, if events are first stored and then periodically processed in batches, we talk about *batch* event processing. This latter approach is usually preferred when timeliness requirements are not that strict, indeed an inner characteristic of batch processing resides in the delays to be paid in order to get updated results, because of its periodical nature. On the other hand, running computations every so often allows to cope with load spikes, failures and imbalances much more easily than the online approach does. Furthermore, a batch approach enables the decoupling of data loading and data elaboration, which provides higher flexibility to accommodate for possible distinct requirements.

Batch processing is heavily employed within business workflows of many medium to large companies

for periodical ETL (Extract, Transform, Load) operations where large data sets produced daily up to hourly have to be moved, analyzed and archived so as to provide the proper means for enforcing specific business intelligence strategies. These scenarios are representative examples of the challenges and opportunities that the emerging BigData trend is fostering. Some well Known companies that are employing this kind of approach are Oracle [22], Dell [15], MicroStrategy [16] and Cisco [20].

In this chapter, we focus on the batch approach and investigate what are its strengths and weaknesses in order to understand whether present batch-oriented computation frameworks can properly meet the previously introduced requirements for TWBCs. In particular, the possibility to increase the frequency of computations on sliding time windows is thoroughly checked into so as to get a better understanding about the class of use cases where a batch approach of this kind can be effectively employed. Section 4.1 provides an overview of the related work in literature in order to properly frame the contributions of this chapter. A simple model for batch processing in TWBCs is defined in Section 4.2, which includes a set of important performance metrics providing the basis for fundamental optimizations. The impact of input data organization on these metrics is analyzed in order to show how a smart subdivision of incoming events in data batches can help in maximizing performance. The instantiation of such model in Hadoop is presented in Section 4.3, together with *ad-hoc* input data organization strategies that aim at reducing computation latency. An experimental evaluation is also provided in Section 4.4 with the aim of highlighting strengths and weaknesses of the proposed strategies.

The contents of this chapter are based on [35].

4.1 Related Work

The developments in the area of distributed event processing happened during last decade have been based mostly on the concept of continuous queries, which run unceasingly over streams of events provided by external sources. These queries are compiled in a network of processing elements that can be distributed over available resources. Several projects have been on this line, although the structures used to model the compiled query are named differently. Among the most cited, we find InfoSphere [94, 130] (networks of InfoPipes), Aurora [26, 61] (networks of processing boxes), TelegraphCQ [57] (networks of dataflow modules), STREAM [38] (query plans composed by operators, queues and synopses), Borealis [25] (networks of query processors), and System S [30] (Event Processing Network (EPN) of Event Processing Agents (EPA)). In this section, we adopt the jargon introduced by the latter. The reconfiguration of an EPN at runtime introduces several issues. The main one is the rebalancing of the load among nodes.

Shah et al. [135] define a dataflow operator called flux, which is integrated in an EPN and takes care of repartitioning stateful operators while the processing is running. Its limitations concern the dependance on configuration parameters that need to be tuned manually and the lack of fault tolerance mechanisms.

Gu et al. [78] propose a mechanism to process Multiway Windows Stream Joins (MWSJs) which distributes tuples to distinct nodes to allow for parallel processing. Their algorithm is specific for MWSJs. Xing et al. [155] describe an algorithm for placing operators such that no replacement is required at runtime. They deem that operators cannot be moved at all. Xing et al. [157] introduce a load distribution algorithm for minimizing latency and avoiding overloading by minimizing load variance and maximizing load correlation. Liu et al. [103] propose a dynamic load balancing operators for stateful algorithm, which spills state to disk or moves the operators to other nodes to resolve imbalances. Lakshmanan et al. [99] present a stratified approach where the EPN is partitioned horizontally in strata and operators can be moved within a single stratum only.

Stateful operators in a continuous query pose the question of memory constraints. The most studied case is that of the joins over distinct event flows or data streams, which require the usage of some time or count based window in order to avoid maintaining the whole history of input data. Time windows cannot guarantee a consequent bound on required memory because of the variability of input event rate. Employing load shedding as a solution [76, 140, 67, 144] could not be feasible in several scenarios where the accuracy

of the processing is a main requirement, for example decision support, intelligence or disaster recovery. In this case, the employment of some disk-based storage is required, as described in several works [104, 145, 117, 147] which however deal with the processing of finite data sets.

There exist some projects addressing the topic of distributed processing of data stored to secondary storage without employing continuous queries.

DataCutter [49] is a middleware which breaks down on-demand clients' requests into processing filters in charge of carrying out required computations. It has been devised to carry out complex processing over large distributed data sets stored to disk.

The MapReduce paradigm [68] implemented in Google and its open source implementation Hadoop [150] have received a great interest by the community and a lot of related projects [105, 115, 149]) have been developed adopting a similar approach.

Dryad [82] is a project developed by Microsoft which organizes the processing as a dataflow graph with computational vertices and communication channels. The computation is batch and can elaborate files stored to a distributed file system.

The possibility of employing a batch approach is touched on in [41], where it is proposed as an appropriate solution when the computation is too slow compared to event arrival, and executing the elaboration for each event doesn't allow to keep up with event rate. In this case, events are buffered and computation runs periodically on the current batch.

4.2 Batch processing for Time Window Based Computations

This section explores how to carry out windowed computations by using a batch approach. Section 4.2.1 defines the key properties of time window based computations, while Section 4.2.2 presents the differences between batch and online approaches. Section 4.2.3 details the computational model we adopted for our purposes and finally Section 4.2.4 describes the strategies we devised to carry out batch computations based on time windows.

4.2.1 Time Window Based Computations

Etzion and Niblett in [74] define an *event* as "an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain". Event processing is performed by feeding events in the form of streams in a processing engine. A stream is "a set of associated events", often "a temporally totally ordered set". The ordering within a stream is defined by a timestamp associated to each event. By elaborating groups of events the engine can output new events that represent the the result of its computation.

With the name *time window based computation* (TWBC) we refer to the elaboration of a set of events that happened within a specific time window. The length of the window and the way such length changes over time depend on the scenario of interest. We can have either a single fixed-length time period, or a fixed-length time period that repeats in regular fashion, or windows that are opened or closed by particular events in the event stream, or, finally, windows that are opened at regular intervals where each window is opened at a specified time after its predecessor [74]. In this chapter, we focus on this latter type of time windows.

A new window is started every ΔT time units, and here we assume that $\Delta T \leq T$[1]. If $\Delta T = T$ then a new window is opened whenever the current one is closed: at any time there is a single window opened and each event belongs to one and only one window. We refer to this case as *juxtaposed windows*. If $\Delta T < T$ then a new window is opened before the previous one is closed, so at any time t (at steady state) there are $n_{ow}(t)$

[1]Cases where $\Delta T > T$, that imply a voluntary loss of events in the periods of length $\Delta T - T$ that occur between the end of a window and the start of the next one, are of little interest and thus ignored in this chapter.

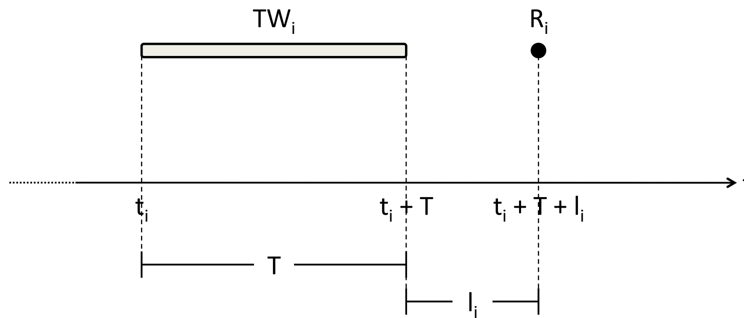


Figure 4.1: Placement in time of a time window and the related result.

open windows, where $\lfloor \frac{T}{\Delta T} \rfloor \leq n_{ow}(t) \leq \lceil \frac{T}{\Delta T} \rceil$. We refer to this case as *interleaved windows*. Each event e having t_e as timestamp belongs to $n_{ow}(t_e)$ different windows. We concentrate on the cases where $n_{ow}(t)$ is fixed to N, that is where T is a multiple of ΔT. Such assumption comes from the observation that often the length of the window is a multiple of the window period.

In general, given a time window TW_i which begins at time t_i and ends at $t_i + T$, we want to decrease the latency l_i between the end of TW_i and the availability of the related result R_i (see Figure 4.1). This latency depends on several distinct factors regarding both the time to produce the result itself and the synchronization between the end of the window and the beginning of the computation, as will be shown in next section.

4.2.2 Batch processing

TWBCs can be processed using either an *online* or a *batch* approach. With the former approach events are kept in memory and processed as soon as they enter in the system in order to minimize the output delay. Conversely, with batch processing incoming events are first stored in a secondary storage (e.g. in a disk-based database or, more commonly, in a file system) and then periodically processed in batches. The frequency of these computations depends on application specific requirements and on the feasibility of running many concurrent computations. While the online approach allows for continuous output, periodical batch computations can only produce periodical output.

Batch processing provides some advantages with respect to online solutions that make it suited to several application scenarios. The amount of data that must be analyzed within a time window, being a function of both the window size and the event rate, can easily grow to huge amount. Storing this data in secondary storage instead of main memory can allow to support applications with massive data rates and large time windows with simpler and less expensive computing infrastructures. Moreover, systems based on the online approach process events as they enter the system; this can easily limit system scalability in applications where processing is computationally intensive and data rates are large. Batch processing, on the other side, defers computation to the end of a time window; incoming events are directly stored in the secondary storage thus allowing very large input rates.

Existing batch processing solutions can be adopted to perform TWBCs. We found that the class of technologies that naturally fits what we need is the one, described in Section 4.1, that includes the projects focused on distributed batch processing of data kept on secondary storage [49, 68, 150, 105, 115, 149, 82]. All them allow to execute distributed complex computations on huge volumes of data. Such data is organized in large files partitioned over available storage nodes. File systems are generally preferred to DBMSs because the operations we need to execute on input data are very simple and don't require most of the high level functionalities provided by today's DBMSs. Events pertaining to each window are stored in files which become the input of the processing engines in charge of producing the output for such time windows.

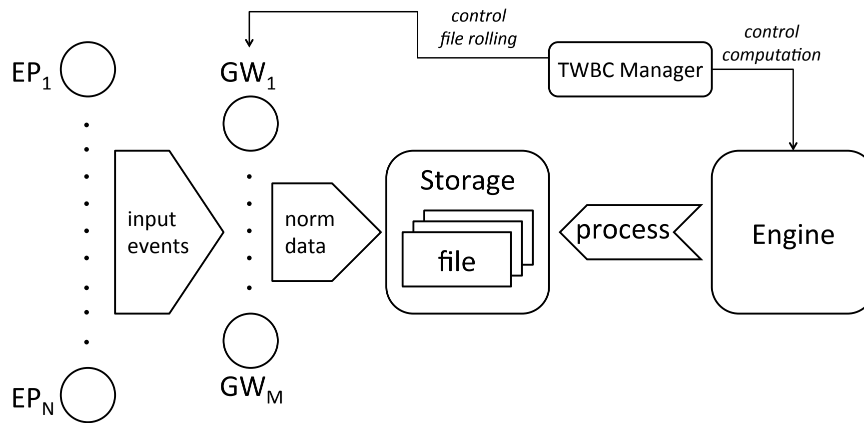


Figure 4.2: High-level architecture of a batch-oriented event processing system.

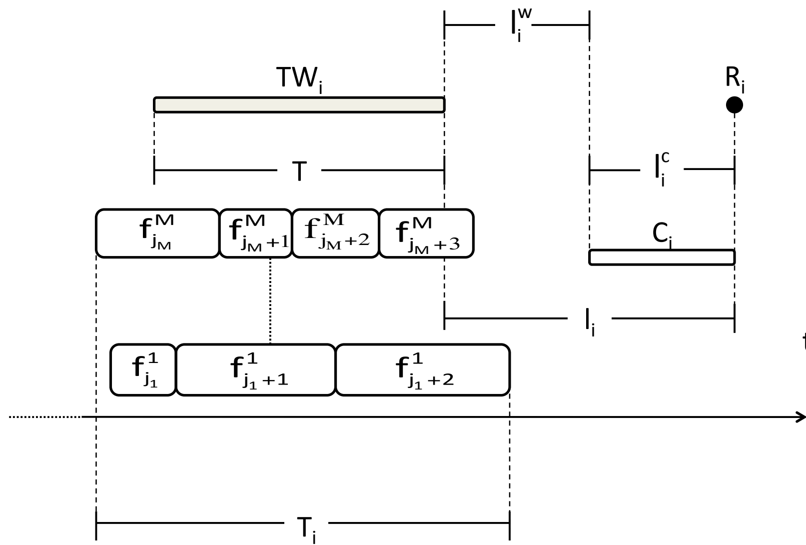


Figure 4.3: Performance metrics in batch-oriented TWBCs.

4.2.3 Computation Model

A high level architecture highlighting the interactions between *event producers* (EPs) and the processing engine is shown in Figure 4.2. Several EPs provide *input events* to a set of *gateways* (GWs) in charge of *normalizing* and storing them in the *storage*. Data in the storage is organized in *files*. A new event is appended to an open file f. Upon certain conditions, a file f is closed and a new file f' is created. We refer to this operation as *file rolling*. At any time, there can be several files that are being written to by distinct GWs and the temporal coverages of different files can overlap. In Figure 4.3 a time window TW_i is shown together with the temporal coverages of the files containing its events ($f_{j_1}, \cdots, f_{j_1+2}$ to $f_{j_M}, \cdots, f_{j_M+3}$) and the position in time of the related computation C_i. The trigger of file rollings is performed by a *TWBC Manager*, which is also responsible of starting the computations by properly controlling the processing *engine*. The engine simply runs the desired computation over all the files intersecting the time window of interest. The output of the computation is not shown here because it is not relevant for our purposes.

A crucial issue in this setting is represented by the synchronization between events and time windows. If event timestamps were set by the EPs, an accurate synchronization would be impossible as the clocks of the EPs and of the TWBC Manager present in general non negligible skews. Various synchronization techniques (e.g. [100]) can be used to mitigate this problem. Event processing systems usually rely on their internal clocks, so an event is considered included within a certain time window according to the time such event enters the engine. In our model, we assume that events are timestamped by GWs as they are written to file. Note that this solution does not completely solve the synchronization issue but effectively alleviates it as synchronizing multiple GWs is a reasonably easy to solve task because GWs are usually deployed in a controlled environment within a single administrative domain. In general, synchronization guarantees depend on the employed technologies and on the type and extent of the deployment. Whether such guarantees allow for a properly accurate computation depends on the specific application.

The latency l_i separating the end of the time window TW_i from the output of the result R_i, is constituted by two distinct temporal components (see Figure 4.3):

- the *wait latency* l_i^w representing the time between the end of the time window TW_i and the beginning of the computation C_i;
- the *computation latency* l_i^c representing the time it takes for the *computation* C_i to complete.

Note that that nothing prevents C_i from beginning before the file containing the last of event in TW_i is closed, assuming that the chosen technologies allow to elaborate files while they are being written to. In that case, also the wait latency doesn't depend on when such last file is closed. Figure 4.3 represents a simple case where the computation begins after the closure of such last file.

In order to reduce the wait latency l_i^w, the computation should be started as soon as a time window ends. By making the TWBC Manager in charge of determining when time windows begin and end, we can minimize it.

The reduction of the computation latency is more complex at this level of abstraction. As shown in Figure 4.3, the computation of TW_i requires to process all the files that contain events in TW_i. These files cover a time range of length T_i, where in general $T_i \geq T$. It is up to the computation itself to filter out the events outside the time window. Since distinct files can overlap in time and also be partitioned without taking into account the temporal order of contained events, the processing engine has to read the content of all these files in order to decide which events must be processed. If we assume that (i) the length of the computation increases as the size of data read from the storage grows (I/O bound computation) and that (ii) the size of stored data grows with the length of the time interval it covers, then we can conclude that a possible way of reducing the computation latency is to include in the processing all and only the events that are included in the time window of interest. A convenient metric able to capture this aspect is the *time efficiency*, defined as the ratio T/T_i, which represents an approximation of the fraction of processed events that actually are within the time window. The approximation is based on the assumption that event rate

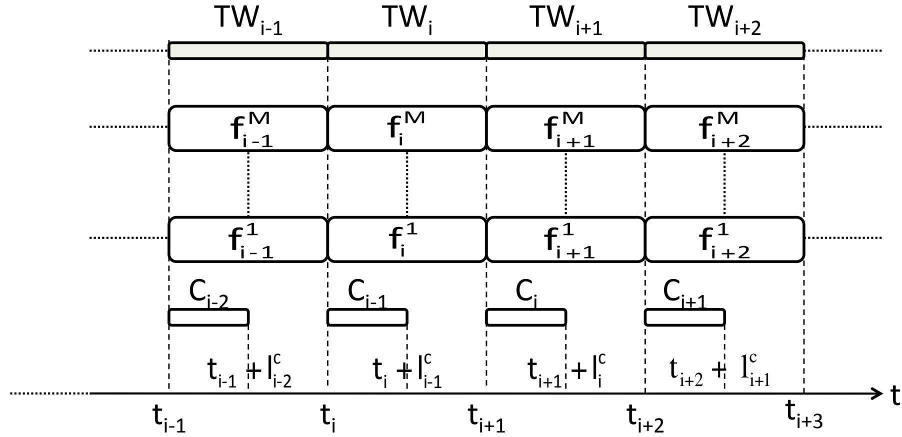


Figure 4.4: Strategy for juxtaposed time windows.

is stable during the period T. The time ratio is a real number in the range $[0, 1]$ where 1 represents the optimum, i.e. only events in TW_i are processed.

4.2.4 Input data organization strategies

We can define proper strategies for organizing input data in files with the aim of optimizing the metrics introduced in the previous section. A strategy defines when file rollings are triggered and thus determines the positioning of events in files and how these files are organized for batch processing. We first present the strategy for juxtaposed windows ($\Delta T = T$) and then that for interleaved windows ($T = N \cdot \Delta T$).

Juxtaposed Windows The case where $\Delta T = T$ is the simplest one and the strategy we propose for it is straightforward but allows for the optimization of all the metrics.

Figure 4.4 illustrates this strategy. We assume there are M GWs, each writing events to distinct files. During time window TW_i, the GW_g writes its incoming events to file f_i^g. When TW_i ends, the TWBC Manager issues a file rolling to all the GWs and tells the Engine to start the computation. This picture represents the collocation of time windows (TW_i), temporal coverage of files (f_i^g) and computations (C_i) on a timeline that spans a period of four consecutive windows. All the events related to time window TW_i are stored in files f_i^x, where $x = 1...M$. At time t_i, files $f_i^1, \cdots, f_i^M$ are opened. At time $t_{i+1} = t_i + T$, all these files are closed and the computation C_i for TW_i is started. The computation ends at time $t_{i+1} + l_i^c$. The figure is simplified so that it seems that the ideal relation $l_i = l_i^c$ holds. In practice, there is some operational delay between the end of TW_i and the beginning of C_i (that is l_i^w). If the Engine cannot process files while they are being written to, then coordination is necessary between the TWBC Manager and the GWs in order to start the computation only after files $f_i^1, \cdots, f_i^M$ are closed, but these are implementation details that depend on the technologies employed. Summing up, we can state that (i) the wait latency is minimized and (ii) the time efficiency is optimized ($T = T_i$).

In Figure 4.4, we are implicitly assuming that $l_i \leq \Delta T$, in order to avoid that computations execute concurrently. This is not a mandatory requirement, but running each computation in isolation allows to use the whole computational power provided by the underlying infrastructure, which possibly means that the latency can be further minimized.

Interleaved Windows In reference to Section 4.2.1, when $\Delta T < T$ we talk about *interleaved* time windows. In particular, we concentrate on the case $T = N \cdot \Delta T$, where $N \in \mathbb{N},\ N > 1$. Also in this case we can

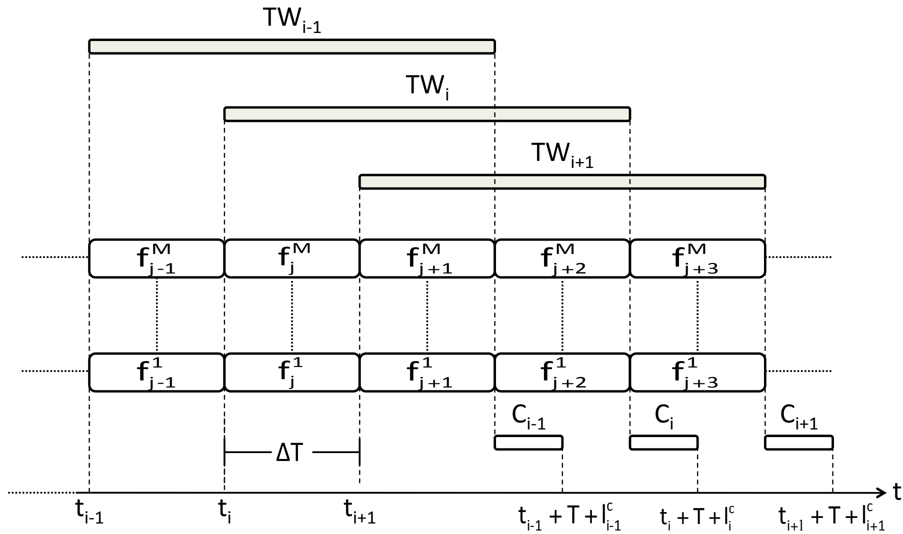


Figure 4.5: Strategy for interleaved time windows.

define a simple strategy, which optimizes all the metrics and can be considered as a generalization of the strategy described in the previous section.

Figure 4.5 shows that each time window is covered by $N \cdot M$ distinct files. In this figure we have reported the time windows, the temporal coverage of files and the computations related only to three consecutive time windows in a setting where $\Delta T = T/3$ ($N = 3$). We have not included occurrences related to other time windows that actually happen in the time span shown in the figure in order to avoid to complicate the figure itself. Consecutive time windows begin with a delay of ΔT, so $t_{i+1} = t_i + \Delta T$. Assuming that TW_0 and the temporal coverage of f_0^x, for $x = 1...M$, begins at the same instant t_0, and making each GW use one file for each period spanning ΔT time units, we have that TW_i is covered by files f_i^x to f_{i+N-1}^x, for $x = 1...M$. Since T is a multiple of ΔT, we obtain an optimum time coverage of the periods of length T, which means that (i) the wait latency is minimized and (ii) the time efficiency is optimized ($T = T_i$).

An interesting aspect of interleaved windows is that the constraint $l_i \leq \Delta T$ becomes much more difficult to comply with respect to juxtaposed windows, because the amount of data to be processed is the same, but the time available for the computation is $1/N$.

4.3 Implementation with Hadoop

Starting from the model introduced in the previous section, we implemented a batch processing framework for TWBCs using Hadoop (Section 2.6.1) as processing platform and HDFS (Section 4.3.1) for input data storage. In this section, we analyze which additional factors to take into account for the optimization of performance metrics with this specific setup.

4.3.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [137] is the default storage used by Hadoop and has been designed to properly support read/write access patterns typical of Hadoop jobs. Data in HDFS is organized in files and directories. Like in a standard file system, each file is broken into block-sized (64 MB by default) chunks, stored as independent pieces to simplify the storage subsystem and to properly fit with replication for providing fault tolerance and availability.

An HDFS installation includes (i) a single *NameNode* which manages the whole namespace of stored

data and controls how data is spread over available resources and (ii) a set of *DataNodes* responsible of storing the actual data.

The default data replication factor is three, which means that in the cluster there are in total three replicas for each block. In our implementation, we set the replication factor to one because we want to be as fast as possible in storing events, and replicating data would consume too much time. This choice penalizes the fault tolerance of our system, but in the scenarios of interest it is more important to keep up with input data rates than loosing some input data because of failures of cluster nodes.

In our implementation, the GWs are in charge of (i) receiving events from the EPs and (ii) converting them in a format suitable for being written to HDFS files. Data in an HDFS file cannot be read by map tasks until such file is closed, so some coordination is needed to ensure that required files have been closed before starting a job.

Although in the scenarios of interest we want to deal with several EPs providing events to a set of GWs, our current implementation supports a single GW. This has no impact on our evaluation of a strategy because the metrics we want to optimize are independent from the number of GWs.

4.3.2 Performance Metrics Optimization

In our implementation, we developed a Java application which encapsulates the functionalities of both a GW and the TWBC Manager. Such application is thus in charge of (references to items of Figure 4.2 are reported in parentheses)

- receiving input events (*input events*)
- converting them in ASCII format and writing them to a HDFS file (*norm data*)
- deciding when to execute a file rolling (*control file rolling*)
- triggering the execution of Hadoop jobs (*control computation*)

As explained in Section 4.2.3, a key aspect for the optimization of performance metrics is the synchronization between time windows and computations. In our implementation based on Hadoop, a coordination between file rollings and jobs executions is also required. By assigning to the Java application the duties of both the GW and the TWBC Manager, we have the possibility of enforcing such synchronization and minimizing the wait latency. For what concerns the computation latency, the time efficiency is important because Hadoop jobs are known to be I/O bound, as shown by Wlodarczyk et al. [151].

Besides the synchronization issues and the optimization of time efficiency, a strategy has further influence on the computation latency. The general problem of data acquisition in Hadoop has been studied in part by Jia et al. [83]. The important point emerging from this paper is that Hadoop works much better with a small number of large files than with a large number of small files. Hadoop divides the input of a job into fixed-size pieces called *input splits*, then creates one map task for each split, which runs the user-defined map function for each record in the split. In an Hadoop cluster configured with a block size B, for a file of size S Hadoop considers $\lceil S/B \rceil$ splits [2]. Since a limited number of map tasks can run concurrently, and since each map task involves a management overhead, the overall performance would improve by using files having size B. Indeed, in this way each map task would work on a chunk of size B and the number of allocated map tasks would be minimized.

More formally, we consider a data set of total size D organized in N files of size d_i ($i = 1...N$), such that

$$\sum_{i=1}^{N} d_i = D \tag{4.1}$$

[2]This relation has been obtained using default values for the configuration parameters that control the way input splits are computed: $minimumSize$ and $maximumSize$; they constraint the minimum and maximum size of a split, respectively. The formula used to compute the split size is $max(minimumSize, min(maximumSize, B))$, where B is the block size [150].

The number of splits required for such data is

$$S^{(N)} = \sum_{i=1}^{N} \left\lceil \frac{d_i}{B} \right\rceil \tag{4.2}$$

where $S^{(l)}$ indicates the number of splits using l files. To show how the number of splits decreases with the number of files, we consider what would happen if we used a single file of size D.

$$S^{(1)} = \left\lceil \frac{D}{B} \right\rceil \tag{4.3}$$

The comparison between the terms in equations (4.2) and (4.3) requires to express both D and d_i as a function of B. Let $b = D \ div \ B$ and $r = D \ mod \ B$, where *div* and *mod* are the quotient and the remainder of the division, respectively. Then we can write

$$D = b \cdot B + r \tag{4.4}$$

where $r < B$. Similarly, for $i = 1, \cdots, N$ we can write

$$d_i = b_i \cdot B + r_i \tag{4.5}$$

where $\forall i,\ b_i = d_i \ div \ B,\ r_i = d_i \ mod \ B,\ r_i < B$. Replacing equations (4.4) and (4.5) into (4.1) we have

$$b \cdot B + r = B \cdot \sum_{i=1}^{N} b_i + \sum_{i=1}^{N} r_i \tag{4.6}$$

We state that

$$b \geq \sum_{i=1}^{N} b_i \tag{4.7}$$

which in turn by (4.6) implies $r \leq \sum_{i=1}^{N} r_i$, and we prove it by contradiction as follows. Let us assume that $b < \sum_{i=1}^{N} b_i$, so we can write

$$b = \sum_{i=1}^{N} b_i - a \tag{4.8}$$

where $a \in \mathbb{N}$, $a > 0$. Replacing (4.8) in (4.6) we have

$$\begin{aligned}
(\sum_{i=1}^{N} b_i - a) \cdot B + r &= B \cdot \sum_{i=1}^{N} b_i + \sum_{i=1}^{N} r_i \Longrightarrow \\
r &= a \cdot B + \sum_{i=1}^{N} r_i \Longrightarrow \\
r &\geq B
\end{aligned}$$

which is in contradiction with the definition of r. By replacing (4.4) and (4.5) in (4.3) and (4.2), respectively, we can write

$$S^{(1)} = b + 1 \tag{4.9}$$

$$S^{(N)} \leq N + \sum_{i=1}^{N} b_i \tag{4.10}$$

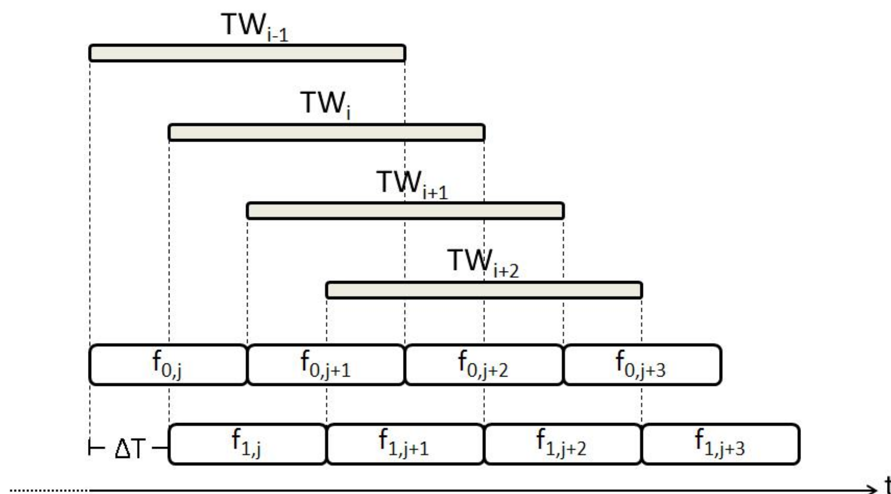


Figure 4.6: Rolling Strategy for interleaved time windows with Multi Files Flows.

The ≤ relation in (4.10) comes from the fact that $r_i = 0$ can hold for some i. Subtracting (4.9) from (4.10) we have

$$S^{(N)} - S^{(1)} \leq N - 1 + (\sum_{i=1}^{N} b_i - b) \leq N - 1 \tag{4.11}$$

where the last ≤ relation comes from (4.7).

Equation (4.11) expresses the fact that using N files instead of one can make the number of splits increase by up to $N - 1$. The worst case is when $b = \sum_{i+1}^{N} b_i$.

In an Hadoop cluster with L TaskTrackers configured with MS available map slots each, at most $L \cdot \text{MS}$ map tasks can run at the same time. Depending on the value of D, it could be impossible to run all the map tasks in a single round. Indeed, if $D > B \cdot L \cdot \text{MS}$, at least $L \cdot \text{MS} + 1$ map tasks are required, regardless of how data is organized in files. However, the goal remains to arrange input data so that resulting map tasks can be executed in the minimum number of rounds, and at the same time ensuring that the load is fairly distributed among TaskTrackers. The last point is important since the reducer tasks start their work whenever all map tasks are completed, so load imbalances can make a TaskTracker employ more time than the others to complete the map phase, which in turn causes the beginning of reduce phase to delay.

As discussed in [52], the best solution for loading this kind of input data to HDFS is employing some technology like Chukwa, which collects external streams and writes them to HDFS files. While Chukwa proves to be very useful when there are several geographically distributed GWs, simpler scenarios with a single local GW can be managed by a single application like ours, which keeps writing data to HDFS as new events arrive.

Multi Files Flows In an Hadoop based implementation, a possible problem of the interleaved window strategy described in Section 4.2.4 is that N can be very large and involve high computation latencies. In Section 4.4, we will see some scenarios where this actually happens. Trivially using larger files doesn't work. In reference to Figure 4.5, if for example we set each file to cover $2 \cdot \Delta T$, then for time windows TW_j, with j even, we would have a wait latency of ΔT (due to the fact that a Hadoop job cannot read open files), which is unacceptable. The synchrony between time windows, HDFS files rollings and Hadoop jobs has to be kept in order to optimize wait latency. This also implies that each time window still has to be perfectly fit by HDFS files, that is the time efficiency has to be optimal.

The solution we propose consists in replicating data so as to (i) use a lower number of bigger files and (ii) provide each time window with the HDFS files required to have a perfect coverage. As shown in Figure 4.6,

data is replicated K times ($K = 2$ in the figure) using K distinct *files flows*. K is required to be a divisor of N, that is $N \ mod \ K = 0$. Since we have a single GW, we don't need to specify which GW writes which file. In this case we use the notation $f_{w,j}$ to indicate the j-th HDFS file of the w-th files flow. With K files flows, each file can cover a time interval of $K \cdot \Delta T$ and the number of files required to cover a time window becomes N/K. Files in flow w are closed with a delay of ΔT with respect to files in flow $w - 1$. In this way, each time window TW_i has a perfect coverage with the files $f_{w,j}$ to $f_{w,j+\frac{N}{K}-1}$ where $w = i \ mod \ k$ and $j = i \ div \ k$. In reference to Figure 4.6, TW_{i-1} is covered by $f_{0,j}$ and $f_{0,j+1}$, while TW_i is covered by $f_{1,j}$ and $f_{1,j+1}$, and so on.

Compared to the solution for interleaved windows reported in Section 4.2.4, this strategy allows to decrease the number of required files by a factor K at the cost of replicating input data K times.

4.4 Experimental Evaluation

Our experimental evaluations are aimed at validating the model introduced so far, giving a hint about the exhibited computation latencies and comparing the two strategies defined for interleaved time windows. We didn't evaluate the strategy for juxtaposed windows because it can be considered as a special case of interleaved windows with $N = 1$, and the evaluations of these strategies become interesting when N is large.

We carried out several evaluations simulating a scenario where a fictional traffic monitoring application is requested to produce statistics every minute ($\Delta T = 1$ minute) about the packets observed in the last hour ($T = 1$ hour, $N = 60$). We vary input packet rate and observe the latency of the jobs. For each fixed packet rate, we measured the latency of the first 12 jobs. Latencies of subsequent jobs did not reveal any further insights on the performance of the system and are thus not shown. As a warm up phase, we let the system load data for 1 hour before starting the first job. In this way each job actually works on a time window of 1 hour.

The equations defined in Section 4.3.2 can be simplified by introducing some assumptions based on the properties of our evaluations. Since packet rate is kept fixed for each run, the size of the input data stored for each ΔT can be considered constant, so we can assume that

$$\forall i, \ d_i = d \tag{4.12}$$

With the largest packet rate we used, the size of the input data stored for each ΔT was at most 40 MB, which is less than the size of a block (64 MB), so we can also assume

$$d < B \tag{4.13}$$

We can use (4.12) and (4.13) to rewrite (4.1), (4.2) and (4.3) as follows, respectively

$$D = N \cdot d$$

$$S^{(N)} = \sum_{i=1}^{N} \lceil \frac{d}{B} \rceil = N$$

$$S^{(1)} = \lceil \frac{N \cdot d}{B} \rceil$$

We use a simple TCP traffic data analysis computation, where packets are filtered out on the basis of the value of TCP flags and are then written to an output HDFS file. The filter we considered is such that almost any packet is dropped. The number of reducers is set to 1. This kind of computation is executed by a very simple event processing network with a set of map tasks which execute I/O bound work and send filtered data to a single reduce task in charge of writing the resulting data to a file. The choice of using this kind of filter has been driven by the observation that different rolling strategies can only impact

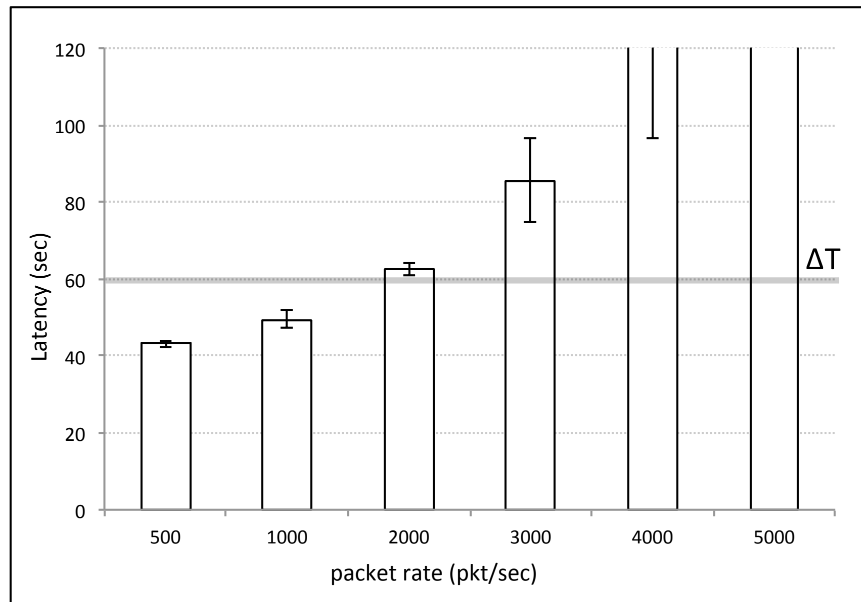


Figure 4.7: Average latencies and standard deviations for the basic interleaved windows strategy.

the performance of map tasks, so, in order to better observe their effectiveness, we consider a computation where the contribution of reduce tasks to the latency is negligible.

We setup an Hadoop deployment with seven nodes. In one node we placed the JobTracker and the NameNode. In each of the other nodes we placed a TaskTracker and a DataNode, so as to enable the JobTracker to allocate map tasks where data actually was and minimize data transfers. We used another node where we installed our Java-based application. Each node was a Virtual Machine (VM) running Ubuntu 10.04 and equipped with 2x2800 MHz CPUs, 3 GB of RAM and 15 GB of disk storage. The networking infrastructure was based on a 10 Gbit LAN.

Basic Interleaved Windows Strategy The evaluations for the basic interleaved windows strategy (as introduced in Section 4.2.4) are aimed at showing (i) how the engine is able to keep up with increasing input data rates and (ii) what happens when the constraint $l_i \leq \Delta T$ is violated, i.e. the time needed for computing over a time window is greater than the time-span between the start of two subsequent windows. The average latencies and related standard deviations registered in these experiments are reported in Figure 4.7. These results confirm that Hadoop jobs are I/O bound: as the packet rate grows, input data size grows as well and the latencies become larger. When packet rate is set at 3000 pkt/sec the standard deviation is quite large (10.86 sec.), which denotes a high variability in the observed latencies. The latency values for packet rates 4000 and 5000 pkt/sec are purposely left out of scale to highlight how such variability grows with input data size.

We say that there is *stability* when the computation latency doesn't keep growing job after job. It is to note that when packet rate is set at 2000 pkt/sec the average latency is 63 seconds, i.e. slightly greater than ΔT, stability is still preserved. This happens because the overlapping between consecutive jobs is quite small and unable to make the engine run out of available resources. With packet rates 3000 pkt/sec or greater, we observe that stability doesn't hold anymore, as shown in Figure 4.8, where it is made clear that the latencies keep increasing as new jobs are executed. In this case, the initial latencies are more than 10 seconds larger than ΔT and subsequent jobs are progressively delayed. The consequence is that the engine becomes unable to deliver acceptable performance.

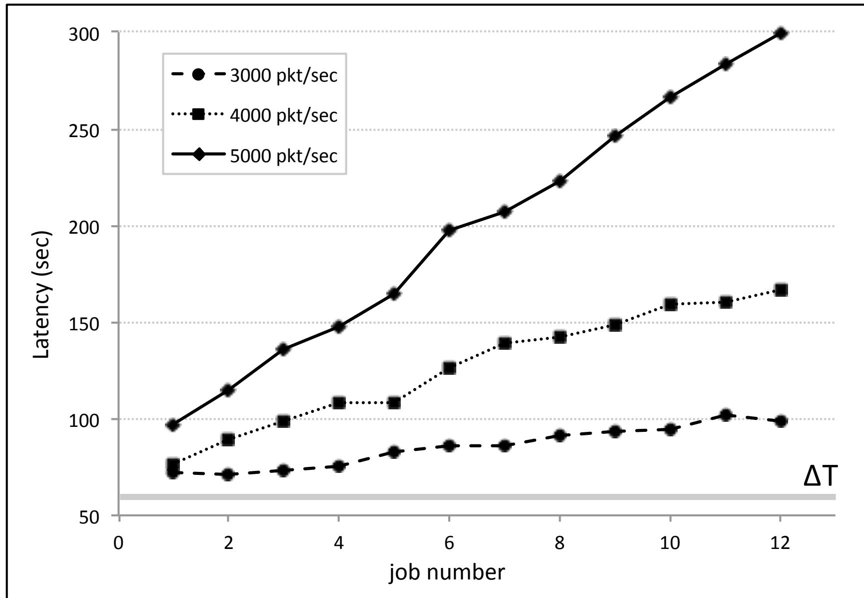


Figure 4.8: Observed latencies when packet rate is greater than 2000 pkt/sec.

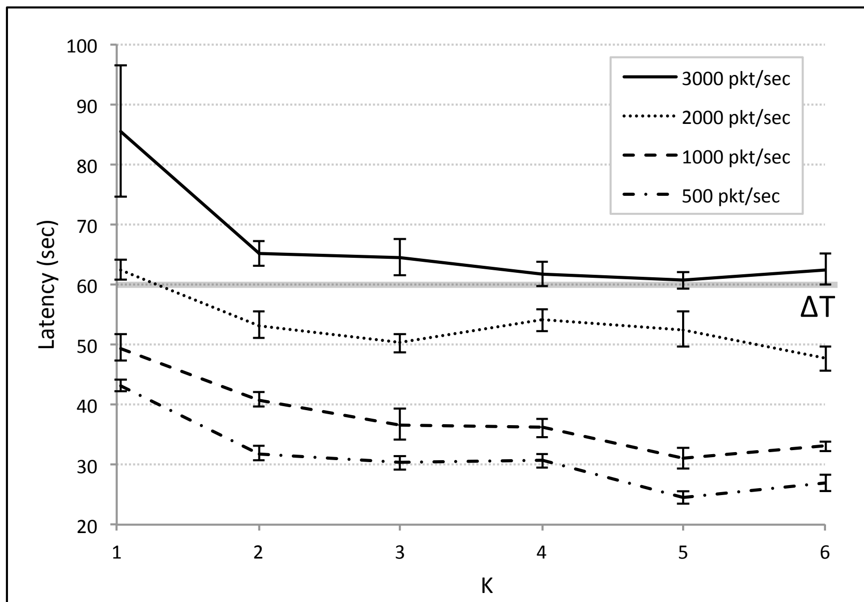


Figure 4.9: Average latencies varying both input packet rate and K.

K	500 p/s	1000 p/s	2000 p/s	3000 p/s
1 60 files	60 split 4 MB	60 split 8 MB	60 split 16 MB	60 split 24 MB
2 30 files	30 split 8 MB	30 split 16 MB	30 split 32 MB	30 split 48 MB
3 20 files	20 split 12 MB	20 split 24 MB	20 split 48 MB	40 split 72 MB
4 15 files	15 split 16 MB	15 split 32 MB	15 split 64 MB	30 split 96 MB
5 12 files	12 split 20 MB	12 split 40 MB	24 split 80 MB	24 split 120 MB
6 10 files	10 split 24 MB	10 split 48 MB	20 split 96 MB	30 split 144 MB

Table 4.1: Number of splits and file sizes as packet rate and K change.

Multi Files Flows Strategy The evaluation of the Multi Files Flows Rolling Strategy is aimed at showing how decreasing the number of files positively impacts on the computation latency. We used again the same packet rates reported in Section 4.4 and we tracked average latencies and related standard deviations with K (replication factor) varying from 1 to 6. We didn't tested larger values for K because it would have required too much space on disk and generated too much overhead network traffic due to the high level of data replication.

Results are reported in Figure 4.9. Each curve represents the average latency for a specific packet rate for different values of K, with error bars used to plot the standard deviations. Introducing data replication, that is moving from $K = 1$ to $K = 2$, makes stability hold even with a packet rate set at 3000 pkt/sec. This can be noted by looking at the difference between the standard deviations for $K = 1$ and $K = 2$. With reference to the equations derived before in this section, by using a specific K we can organize input data in N/K files and the resulting number of splits is

$$S^{(N/K)} = \frac{N}{K} \cdot \lceil \frac{K \cdot d}{B} \rceil \tag{4.14}$$

The relations between packet rates, K, number of splits and file sizes are reported in Table 4.1. Let us define the set div_N as the set of divisors of N. The value of K which minimizes the number of splits is

$$K_{opt} = \underset{K \in div_N}{\operatorname{argmin}} \, S^{(N/K)} \tag{4.15}$$

Figure 4.10 plots the distance in percentage from the optimum number of splits as K varies for some distinct ratios d/B, so as to provide results that are oblivious from the specific dimensions that can come into play in individual scenarios. As such picture makes evident, just using small values for K allows to obtain numbers that are not far from the minimums, which means that a convenient tradeoff between storage requirements and resulting mappers count can be firstly foreseen and then achieved.

Figure 4.11 shows how the number of splits varies in function of K (a logarithmic scale is used) for several packet rates. As already stated, the number of splits commonly decreases with the increase of K, except for some unlucky cases where the dimension of resulting files doesn't fit well with the block size B, but we will see soon that this is not necessarily a problem. With large packet rates, obtaining a near optimum number of splits doesn't require the use of large values of K. For example, when packet rate is 5000 pkt/sec and $K = 3$ there are 40 splits while the optimum is 38. When packet rate is 3000 pkt/sec, $K = 5$ allows to have 24 splits, that is only one more than the optimum.

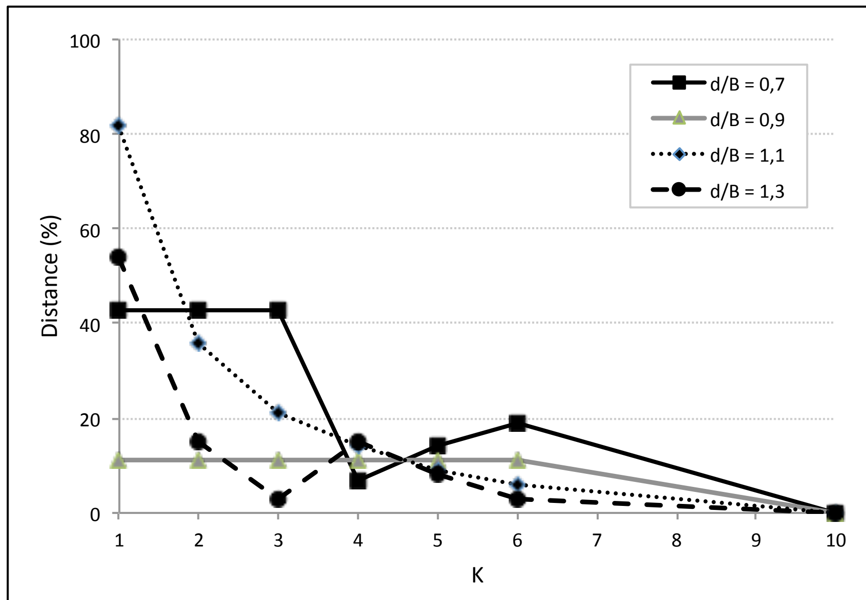


Figure 4.10: Distance (%) from the optimum for the number of splits in function of K, for distinct ratios d/B.

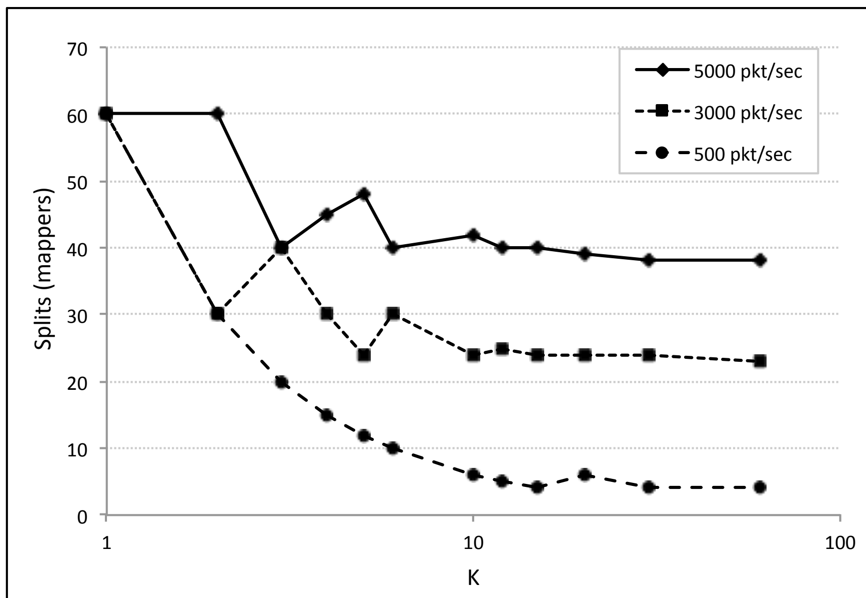


Figure 4.11: Number of splits in function of K for some packet rates.

Optimizing the number of splits with lower packet rates require larger values of K. Indeed with 500 pkt/sec a good result can be achieved by setting $K = 10$, which entails six splits while the optimum is four. This trend mainly depends on the fact that when the packet rate is small the size of files is small as well, and several files can be merged within a single split, which means that the decrease of the number of files coincides with the decrease of the number of splits. When the packet rate is large, merging files is not likely to decrease the number of splits, because the size of merged files is probably greater than the block size. For example, when packet rate is 5000 pkt/sec, $d = 40$ MB and 60 splits are required. By setting $K = 2$, the size of a single file becomes 80 MB, which requires two splits, so even if the number of files has been halved, the total number of splits remains the same.

For what concerns the latencies reported in Figure 4.9, for packet rates 500 and 1000 pkt/sec the situation is quite easy to analyze because $K \cdot d \leq B$ for $K \leq 6$, so the (4.14) becomes $S^{(N/K)} = N/K$. This implies that the number of splits decreases monotonically as K is increased, which in turn makes the average latency decrease as well, except for the case $K = 6$ where a small growth occurs. The reason of such worsening is the imbalance of the allocation of map tasks to TaskTrackers, as previously introduced in Section 4.3.2. Looking at Table 4.1, when $K = 5$ there are 12 splits having the same size, which allows for a fair allocation of two map tasks for each TaskTracker (with six TaskTrackers in the cluster). Each TaskTracker can execute such tasks in parallel because there are two available slots for map tasks for each TaskTracker. When $K = 6$ we have instead ten equally-sized splits, which makes a fair allocation impossible, indeed two slots remain not allocated. In this case, each TaskTracker runs in parallel one or two map tasks that work on splits larger than those of the case $K = 5$, which causes the small increase in the overall latency.

The case for packet rate 2000 pkt/sec is a little bit more complex. For K up to three, everything is fine and the average latency keeps decreasing. When $K = 4$, latency grows by four seconds (8%) despite the number of splits is minimized (15 splits is indeed the minimum for $K \leq 6$). The reason again is the imbalance in the allocation of map tasks to TaskTrackers. The first 12 map tasks can run in parallel, while the remaining three have to run after the first batch completes. Furthermore, these three tasks work on block-sized splits, which makes their completion time not marginal. Therefore, the critical path in the map phase is the sequence of two map tasks each working on a block-sized split (64 MB + 64 MB). A possible solution would consist in reorganizing the last three splits so as to fairly spread the load among all the TaskTrackers, but such feature is not provided by Hadoop. When $K = 5$, the number of splits raises to 24 but the average latency is one second lower. Each of the 12 files has size 80 MB, so it is divided in two splits, one with size 64 MB and the other 16 MB. In total there are 12 block-sized splits and 12 splits of 16 MB. The allocation in this case can be very fair because each map slot can be assigned in sequence to two map tasks, the first works on a 64 MB split and the other on 16 MB one, which would shorten the critical path (64 MB + 16 MB). In reality, the locality awareness of Hadoop (see Section 2.6.1) affects such a theoretically optimal allocation strategy because the placement of some map tasks is driven by the actual position of the input data they need to work on. Depending on how the blocks of input data are distributed over the DataNodes, this can entail an allocation where two tasks working on a 64 MB split are assigned to the same slot, making the critical path equal to the one for $K = 4$. Furthermore, the presence of two classes of splits having so different sizes (64 MB vs 16 MB) can make some slots become free much sooner than others, forcing the JobTracker to assign them available splits without concerning too much about whether the new allocation could cause future imbalances. Among the 12 jobs we ran for $K = 5$, for seven of them we observed a 64 MB + 64 MB critical path, while for the other five the critical path was 64 MB + 16 MB + 16 MB. Such variability is also highlighted by the higher value of the standard deviation. In this case, the potential benefits of an optimal allocation are offset by an unlucky distribution of blocks over the cluster, which on average makes the performance improve only marginally. When $K = 6$, the average latency is improved by five seconds (9%) with respect to $K = 5$. Such an enhancement is due to (i) the reduction of splits and (ii) the reduction of the difference between the sizes of the splits, which helps to prevent allocations resulting in imbalances. Indeed, each file is 96 MB and is divided into two splits of 64 MB and 32 MB, which is much less likely to cause the imbalances we observed for $K = 5$. All the 12 jobs we ran exhibited a critical path of 64 MB + 32 MB.

For packet rate 3000 pkt/sec, we get stability using $K = 2$ and we registered a critical path of 48 MB + 48 MB + 48 MB. Setting $K = 3$, the critical path reduces to 64 MB + 64 MB + 8 MB but the great difference between split sizes causes imbalances that make the latency improve negligibly and the standard deviation increase. When $K = 4$, in most of the jobs we observed a critical path of 64 MB + 64 MB while in a few we reported 64 MB + 64 MB + 32 MB, but the reduced difference between the splits and the lower number of mappers (ten less) entails a decrease of the latency of three seconds. Going up to $K = 5$, the situation improves further because we get no imbalances thanks to the very small variation between split sizes (64 MB vs 56 MB), and the number of map tasks allows for an optimal allocation to the TaskTrackers. Finally, using $K = 6$, a small worsening (two seconds) is noticed due the increment of both the difference between split sizes (64 MB vs 16 MB) and the number of the map tasks, which together entails for a worse allocation to TaskTrackers.

The table does not report the results for packet rates 4000 and 5000 pkt/sec because we didn't manage to make stability hold using $K \leq 6$.

Figure 4.9 gives evidence that major improvements are achieved with small values of K (two or three), and that larger settings of K don't provide relevant enhancements. What we can get from these evaluations is that we can apply the Multi Files Flows Rolling Strategy for successfully decreasing the computation latency at the price of replicating data with a reasonable replication factor.

Conclusion

The first part of this thesis has delved into the topic of Collaborative Environments focusing on the advantages of leveraging information sharing to improve the effectiveness of security defenses. Also the obstacles to the employment of information sharing have been described, and possible strategies for overcoming them have been proposed. The Semantic Room (SR) abstraction has been introduced as a mean to create Collaborative Environments in practice, and three distinct implementations of an SR for detecting inter-domain stealthy port scans have been presented. Many evaluations have given evidence that the collaboration can sharpen detection accuracy in practice, and several insights have emerged for what concerns the best technology to use (either Esper, Storm, Hadoop) depending on the specific real scenario.

In the second part of this thesis, two technologies have been examined more in depth, namely Storm and Hadoop, with the aim of enhancing their performances. For what concerns Storm, two scheduling algorithms have been designed and implemented which minimize the inter-node traffic in order to decrease computation latency. Evaluations showed a relevant improvement of the performance with respect to the default Storm scheduler. For what concerns Hadoop, a model has been developed for running computations based on time windows by adopting a batch approach. Within such model, some strategies have been illustrated to organize input data into the storage (HDFS) in order to optimize a set of metrics related to the completion time of Hadoop jobs.

Achieved Results

Rather than providing some definitive response regarding a specific topic, this thesis has put together heterogeneous pieces ranging from programming abstractions and high-level methodologies (SR, continuous queries, batch processing) to state-of-the-art technologies (Esper, Storm, Hadoop), from latest security frontiers (today's cyber attacks implications, confidentiality requirements) to practical software engineering issues (integration of legacy infrastructures in the FM-SR), and the result is a sort of "architectural toolkit" consisting of practical guidelines, examples and solutions that can be combined together to build effective architectures when Big Data problems have to be solved, and in particular when information sharing becomes a key aspect of the solution to undertake. The following list summarizes such a toolkit as a set of statements recapping learned lessons.

- **Collaboration is worth**
  When confidentiality and fairness issues can be overcome, making distinct organizations and actors collaborate by sharing their information is effective for bringing added value to all the participants. The intuition behind such result has been provided in Section 1.3, where it has been explained that having at disposal larger amount of input data coming from distinct sources allows to obtain a more accurate picture about some target phenomenon to analyze. The practical effectiveness of a collaborative approach has been shown both when available input is homogeneous (as in the ID-SR, see Section 2.7) and when heterogeneous streams have to be correlated (as the FM-SR, see Section 1.5).

- **Distribute the computation**
  In a Collaborative Environment where participants provide a number of resources, these latter should

be leveraged to distribute the computation because this allows for better fault-tolerance, scalability and performance. Having more physical machines at disposal reasonably allows to enforce effective strategies to reallocate computations that were running on failed resources. For the same reason, more resource-demanding computations can be replicated over a proper number of machines in order to avoid possible bottlenecks. Storm has been described as a powerful framework for distributed event processing that inherently allows to distribute any computation (see Section 2.5.1). The evidence of this result is provided in Section 2.7.2, where the Storm-based implementation of the R-SYN algorithm achieves lower latency and higher throughput than the Esper-based implementation.

- **Distribute the data**
  As for the previous point, distributing the data allows to enforce locality-aware scheduling that moves computation rather than data, which leads to better performance and allows to avoid possible network bottlenecks. Hadoop is a well-known batch processing framework that enforces locality-awareness to minimize data movements (see Section 2.6.1). Agilis (see Section 2.6.2) goes one step further by employing a RAM-based storage rather than a disk-based in order to get faster data accesses.

- **Limit data exchanges through the Internet**
  Available physical bandwidth can become a bottleneck when inter-participant communications are carried out over the Internet. Possible countermeasures include employing locality-aware scheduling and input pre-processing (i.e., filtering, aggregating) in order to decrease the amount of data to exchange. The evaluations presented in Section 2.7.1 gave evidence that, when all the data is sent to a CEP engine deployed on a single site, the performance degrades as the available bandwidth decreases. Furthermore, Section 2.7.3 showed that such limitation can be overcome by an implementation where the computation is moved rather than the data to analyze.

- **Storm computations benefit from minimizing inter-node traffic**
  A scheduling aimed at decreasing the amount of events to be sent among physical nodes positively impacts on the average computation latency. Employing a Storm scheduler that takes into account this lesson allows to get considerable lower average latencies for completing the computation of an input event, as clearly shown in Section 3.4. In addition, it is shown that an adaptive scheduler capable of adjusting the allocations at runtime achieves better results compared to a scheduler that fixes the allocation before the computation starts.

- **Hadoop jobs benefit from a clever organization of the input on HDFS**
  Especially in computations based on sliding time windows, organizing input data in files so that jobs have to read all and only the required data is crucial to minimize job completion time. As explained is Section 4.3.2 and shown in Section 4.4, Hadoop jobs are I/O bound and this implies a sharp impact of the amount of data to read from HDFS on jobs completion time. Furthermore, another crucial aspect to take into account when organizing data in HDFS is the number of files: the more the files are, the more the input is fragmented among a larger number of map tasks, which leads to higher jobs completion time.

Future Research Directions

Considering the broad scope of the topics touched upon in this thesis, very diverse future directions can be identified. It is convenient to list them according to the way the topics of the thesis are organized.

- **Collaborative Environments**
  - The area of secure multi-party computations is fundamental to provide the technologies required to enforce proper confidentiality requirements. In the specific, more efficient techniques are

necessary to enable their employment in real scenarios where huge amounts of data have to be exchanged, possibly over the Internet.

◇ So far, the issue of fairness in information sharing has been addressed only marginally; new strategies of incentives and penalties should be devised and, above all, detailed experiments should be carried out in practical contexts in order to assess the real effectiveness of such strategies.

◇ The crucial importance of applying the collaborative approach in real scenarios cannot be stressed enough: a greater number of case studies is required where collaboration through information sharing is shown to be beneficial for all the participants.

- **Scheduling in distributed processing platforms**
  A key issue related to dynamic reconfiguration of a query in distributed processing platforms is the management of stateful operators; even though works exist in this field [55], there is still room for further improvements and evaluations on real scenarios.

- **Batch computations**
  Also in this area, further evaluations are required to test in practice the effectiveness of multi-tenant Internet-wide deployments where the nodes of the (Hadoop) cluster are provided by distinct actors, and where input data is kept locally to the producer of the data itself. Besides sensibly reducing the volumes of data to be transferred over the Internet, this schema would also ease the enforcement of privacy-preserving and anonymity techniques.

- **Blending online-oriented and batch-oriented approaches**
  Both the approaches have strengths and weaknesses, and rarely either approach covers all the needs in a real scenario; it would be definitely interesting to investigate methodologies to define frameworks where online-oriented and batch-oriented technologies co-exist and interact together with the aim of delivering the best possible architectural solution.

Future Scopes of the SR Abstraction

The key goal of the SR abstraction is enabling information sharing among distinct actors. The architectures and technologies presented in this thesis allow to manage SRs comprising a few participants, each providing large inputs. The present shift toward the "Internet of Things", also fostered by the amazing spread of smartphones and tablets and by the forthcoming global adoption of IPv6, prompts for novel and very interesting scopes of the SR abstraction, characterized by Collaborative Environments comprising much more participants. The main obstacle to overcome in order to make the SR abstraction fit such a new class of scenarios is represented indeed by the scalability in the number of participants.

Let us resume the example reported in the Introduction [127]. The application developed by Inrix Inc. leverages the information related to traffic events provided by thousands to millions of mobile devices (i.e., smartphones equipped with GPS) in order to deliver the most updated and accurate view about current traffic conditions and best routes to go through. Even though the aggregate amount of generated data could be compared to the volume of data that can be managed by the kind of SRs detailed in this thesis, the clear difference lies in the diverse order of magnitude of the number of endpoints taking part in the collaboration.

From a technical point of view, the SR abstraction and its instantiations should evolve so as to cope with an overhead of higher level, due to the fact that the per-participant overhead is now multiplied for a much greater number of participants. Such per-participant overhead is mainly spread over two aspects: membership and data sources.

- **Membership overhead**
  When a generic actor wants to join an SR, it has to sign a contract (see Section 1.3), provision re-

quested resources and deploy required software to be integrated with its own information systems. This procedure is obviously excessively burdensome to carry out when millions of members have to be managed within the single SR. A direction to investigate regards the employment of lighter, more flexible join procedures that would allow the SR infrastructure to easily keep up with the required sizes of SR memberships. A resulting drawback comes from a diminished control on who can become an SR member, which could possibly lead to the adoption of malicious or selfish behaviors. In order to contrast this kind of situations, employing policies based on incentives and reputation (see Section 1.4) can be a viable direction to analyze in depth.

- **Data source overhead**
  Each SR member is a data source that injects events to the processing engine, so a dedicated stream has to be established for each participant, which entails an unfeasible overhead with the expected sizes of SR memberships. Another relevant point concerns the consequent high rate of joins and leaves to expect. Considering a near real time processing platform like Storm, such a high variability of the number of input sources over the time translates to frequent reconfigurations that negatively impact on the performance and make Storm scalability useless. Advanced architectural designs are required to cope with data source overhead. Possible research directions include the possibility to aggregate input streams on the edge as much as possible with the aim of masking partially the continual addition and removal of input sources.

Bibliography

[1] EsperHA: High-Availability for Event Processing. http://www.espertech.com/products/esperha.php. Online; accessed on 28-Aug-2013.

[2] European Anti-Fraud Office. http://ec.europa.eu/anti_fraud/index_en.htm. Online; accessed on 26-Aug-2013.

[3] Financial Services - Information Sharing and Analysis Center. https://www.fsisac.com/. Online; accessed on 26-Aug-2013.

[4] 2000 DARPA Intrusion Detection Scenario Specific Data Sets. http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/2000data.html, 2000. Online; accessed on 28-Aug-2013.

[5] JavaScript Object Notation (JSON). http://www.json.org/, 2001. Online; accessed on 05-Sep-2013.

[6] Complete Snort-based IDS Architecture. http://cybervlad.net/ids/index.html, 2002. Online; accessed on 27-Aug-2013.

[7] EsperTech first to unveil ESP/CEP performance benchmark. http://www.espertech.com/news/20070814_performance.php, 2007. Online; accessed on 17-Sep-2013.

[8] DShield: Cooperative Network Security Community - Internet Security. http://www.dshield.org/indexd.html, 2009. Online; accessed on 27-Aug-2013.

[9] IBM WebSphere eXtreme Scale. http://www-01.ibm.com/software/webservers/appserv/extremescale/, 2009. Online; accessed on 05-Sep-2013.

[10] Jaql - Query Language for JavaScript(r) Object Notation (JSON). https://code.google.com/p/jaql/, 2009. Online; accessed on 05-Sep-2013.

[11] Where Complex Event Processing meets Open Source: Esper and NEsper. http://esper.codehaus.org/, 2009. Online; accessed on 27-Aug-2013.

[12] ITOC Research: CDX Datasets. http://www.westpoint.edu/crc/SitePages/DataSets.aspx, 2010. Online; accessed on 28-Aug-2013.

[13] LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing/, 2010. Online; accessed on 28-Aug-2013.

[14] Snort: an open source network intrusion prevention and detection system (IDS/IPS). http://www.snort.org/, 2010. Online; accessed on 28-Aug-2013.

[15] Cloudera and dell partner to deliver complete apache hadoop solution. http://www.cloudera.com/content/dam/cloudera/documents/Dell-Solution-Brief.pdf, 2011. Online; accessed on 29-Aug-2013.

[16] Cloudera and microstrategy announce integration between business intelligence and apache hadoop. http://finance.yahoo.com/news/Cloudera-MicroStrategy-iw-812071667.html?x=0, 2011. Online; accessed on 29-Aug-2013.

[17] Dipartimento del Tesoro - Anti Fraud System. http://www.dt.tesoro.it/en/antifrode_mezzi_pagamento/, 2011. Online; accessed on 30-Aug-2013.

[18] Storm, distributed and fault-tolerant realtime computation. http://storm-project.net/, 2011. Online; accessed on 27-Aug-2013.

[19] WANem The Wide Area Network emulator. http://wanem.sourceforge.net/, 2011. Online; accessed on 28-Aug-2013.

[20] Cloudera enterprise with cisco unified computing system. http://files.cloudera.com/cisco/SolutionBrief_Cisco_Oct_2012.pdf, 2012. Online; accessed on 29-Aug-2013.

[21] Data Never Sleeps. http://www.domo.com/learn/7/236#videos-and-infographics, 2012. Online; accessed on 6-Aug-2013.

[22] Oracle big data appliance x3-2. http://www.oracle.com/technetwork/server-storage/engineered-systems/bigdata-appliance/overview/index.html, 2012. Online; accessed on 29-Aug-2013.

[23] StreamMine3G. https://streammine3g.inf.tu-dresden.de/trac, 2013. Online; accessed on 29-Aug-2013.

[24] The ACM DEBS 2013 Grand Challenge. http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails, 2013. Online; accessed on 29-Aug-2013.

[25] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, 2005.

[26] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[27] Charu C. Aggarwal and Philip S. Yu. On variable constraints in privacy preserving data mining. In *SDM*, 2005.

[28] Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 247–255, New York, NY, USA, 2001. ACM.

[29] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 439–450, New York, NY, USA, 2000. ACM.

[30] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, 2006.

[31] Leonardo Aniello, Roberto Baldoni, Gregory Chockler, Gennady Laventman, Giorgia Lodi, and Ymir Vigfusson. Distributed Attack Detection Using Agilis. In Roberto Baldoni and Gregory Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 157–174. Springer Berlin Heidelberg, 2012.

[32] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, DEBS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[33] Leonardo Aniello, Giuseppe Antonio Di Luna, Giorgia Lodi, and Roberto Baldoni. A Collaborative Event Processing System for Protection of Critical Infrastructures from Cyber Attacks. In *Proceedings of the 30th international conference on Computer safety, reliability, and security*, SAFECOMP'11, pages 310–323, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Leonardo Aniello, Giorgia Lodi, and Roberto Baldoni. Inter-Domain Stealthy Port Scan Detection Through Complex Event Processing. In *Proceedings of the 13th European Workshop on Dependable Computing*, EWDC '11, pages 67–72, New York, NY, USA, 2011. ACM.

[35] Leonardo Aniello, Leonardo Querzoni, and Roberto Baldoni. Input Data Organization for Batch Processing in Time Window Based Computations. In *Proceedings of the 28th Symposium On Applied Computing*, 2013.

[36] Albert K. Ansah, Janus Kyei-Nimakoh, and Millicent Kontoh. Analysis of Freeware Hacking Toolkit. In *Proceedings of the World Congress on Engineering and Computer Science (WCECS)*, Lecture Notes in Engineering and Computer Science, pages 140–149. Newswood Limited, 2012.

[37] Benny Applebaum, Haakon Ringberg, Michael J. Freedman, Matthew Caesar, and Jennifer Rexford. Collaborative, privacy-preserving data aggregation at scale. In *Proceedings of the 10th international conference on Privacy enhancing technologies*, PETS'10, pages 56–74, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Technical report, Stanford University, 2004.

[39] Giuseppe Ateniese, Roberto Baldoni, Silvia Bonomi, and Giuseppe Antonio Di Luna. Oblivious assignment with m slots. In *Proceedings of the 14th international conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'12, pages 187–201, Berlin, Heidelberg, 2012. Springer-Verlag.

[40] Nikolaos Avourdiadis and Andrew Blyth. Data unification and data fusion of intrusion detection logs in a network centric environment. In *ECIW*, pages 9–20. Academic Conferences Limited, Reading, UK, 2005.

[41] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[42] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the 5th Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[43] R. Baldoni, G. Di Luna, and L. Querzoni. Collaborative Detection of Coordinated Port Scans. In *In Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN)*, 2013.

[44] Roberto Baldoni and Gregory Chockler. *Collaborative Financial Infrastructure Protection: Tools, Abstractions, and Middleware*. Springer Publishing Company, Incorporated, 2012.

[45] Roberto Baldoni, Giuseppe Antonio Di Luna, Donatella Firmani, and Giorgia Lodi. A model for continuous query latencies in data streams. In *Proceedings of the 1st International Workshop on Algorithms and Models for Distributed Event Processing*, 2011.

[46] Savina Bansal, Padam Kumar, and Kuldip Singh. An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[47] Yoram Baram, Ran El-Yaniv, and Kobi Luz. Online choice of active learning algorithms. *The Journal of Machine Learning Research*, 2004.

[48] Raj Basu, Robert K. Cunningham, Seth E. Webster, and Richard P. Lippmann. Detecting Low-Profile probes and novel Denial-of-Service attacks. volume 154, pages 477+, January 2001.

[49] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, November 2001.

[50] Monowar H. Bhuyan, D.K. Bhattacharyya, and J.K. Kalita. Surveying port scans and their detection methodologies. *Comput. J.*, 54(10):1565–1581, October 2011.

[51] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Jerome Boulon et al. Chukwa, a large-scale monitoring system. In *Cloud Computing and its Applications*, pages 1–5, 2008.

[53] Andrey Brito, Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 2011.

[54] Michael Cammert, Christoph Heinz, Jurgen Kramer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. Flexible multi-threaded scheduling for continuous queries over data streams. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, 2007.

[55] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[56] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.

[57] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, 2003.

[58] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218. USENIX Association, 2006.

[59] LiWu Chang and Ira S. Moskowitz. Parsimonious downgrading and decision trees applied to the inference problem. In *Proceedings of the 1998 workshop on New security paradigms*, NSPW ’98, pages 82–89, New York, NY, USA, 1998. ACM.

[60] Hsinchun Chen. Trends & Controversies. *IEEE Intelligent Systems*, 26(6):82–96, November 2011.

[61] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[62] United States Intelligence Community. Strategic Intent for Information Sharing, 2011 - 2015. `http://www.mcafee.com/us/resources/reports/rp-in-crossfire-critical-infrastructure-cyber-war.pdf`, 2011. Online; accessed on 26-Aug-2013.

[63] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[64] Gregory Conti and Kulsoom Abdullah. Passive visual fingerprinting of network attack tools. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, VizSEC/DMSEC ’04, pages 45–54, New York, NY, USA, 2004. ACM.

[65] Massimo Coppola, Yvon J?gou, Brian Matthews, Christine Morin, Luis Pablo Prieto, Óscar David Sánchez, Erica Y. Yang, and Haiyan Yu. Virtual organization support within a grid-wide operating system. *IEEE Internet Computing*, 12(2):20–28, 2008.

[66] Symantec Corporation. State of the Data Center Survey, Global Results. `http://bit.ly/OHGNw0`, 2012. Online; accessed on 7-Aug-2013.

[67] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD ’03, pages 40–51, New York, NY, USA, 2003. ACM.

[68] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[69] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazonís Highly Available Key-value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP ’07, pages 205–220, New York, NY, USA, 2007. ACM.

[70] David Dobkin, Anita K. Jones, and Richard J. Lipton. Secure databases: protection against user influence. *ACM Trans. Database Syst.*, 4(1):97–106, March 1979.

[71] Chris Eaton, Dirk Deroos, Tom Deutsch, George Lapis, and Paul Zikopoulos. *Understanding Big Data*. Mc Graw Hill, 2012.

[72] W. El-Hajj, F. Aloul, Z. Trabelsi, and Nazar Zaki. On detecting port scanning using fuzzy based intrusion detection system. In *Wireless Communications and Mobile Computing Conference, 2008. IWCMC '08. International*, pages 105–110, 2008.

[73] Paulo Esteves Verssimo, Leonardo Aniello, GiuseppeAntonio Luna, Giorgia Lodi, and Roberto Baldoni. Collaborative Inter-domain Stealthy Port Scan Detection Using Esper Complex Event Processing. In Roberto Baldoni and Gregory Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 139–156. Springer Berlin Heidelberg, 2012.

[74] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[75] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001.

[76] Buğgra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 171–178, New York, NY, USA, 2005. ACM.

[77] Philippe Golle, Markus Jakobsson, Ari Juels, and Paul Syverson. Universal re-encryption for mixnets. In *Proceedings of the RSA Conference, Cryptographer's track*, pages 163–178. Springer-Verlag, 2002.

[78] Xiaohui Gu, Philip S. Yu, and Haixun Wang. Adaptive load diffusion for multiway windowed stream joins. In *Proceedings of the 23rd International Conference on Data Engineering,*, pages 146–155, 2007.

[79] L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 296–304, 1990.

[80] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[81] Yiyi Huang, Nick Feamster, Anukool Lakhina, and Jim (Jun) Xu. Diagnosing network disruptions with network-wide analysis. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 61–72, New York, NY, USA, 2007. ACM.

[82] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[83] Baodong Jia, Tomasz Wiktor Wlodarczyk, and Chunming Rong. Performance considerations of data acquisition in hadoop system. *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 545–549, 2010.

[84] Juan juan Chen and Xi jun Cheng. A novel fast port scan method using partheno-genetic algorithm. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 219–222, 2009.

[85] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[86] Flavio P. Junqueira and Benjamin C. Reed. The life and times of a zookeeper. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures*, 2009.

[87] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 640–651, New York, NY, USA, 2003. ACM.

[88] H. Kargupta. Multi-agent, distributed, privacy-preserving data management and data mining. 2012.

[89] Krishnaram Kenthapadi, Nina Mishra, and Kobbi Nissim. Simulatable auditing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 118–127, New York, NY, USA, 2005. ACM.

[90] H. Kim, S. Kim, M. A. Kouritzin, and W. Sun. Detecting network portscans through anomoly detection. In I. Kadar, editor, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 5429 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 254–263, August 2004.

[91] Jaekwang Kim and Jee-Hyong Lee. A slow port scan attack detection mechanism based on fuzzy logic and a stepwise p1olicy. In *Intelligent Environments, 2008 IET 4th International Conference on*, pages 1–5, 2008.

[92] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Proceedings of Advances in Cryptology - CRYPTO 2005, LNCS*, pages 241–257. Springer, 2005.

[93] Shijin Kong, Tao He, Xiaoxin Shao, Changqing An, and Xing Li. Scalable double filter structure for port scan detection. In *Communications, 2006. ICC '06. IEEE International Conference on*, volume 5, pages 2177–2182, 2006.

[94] Rainer Koster, Andrew P. Black, Jie Huang, Jonathan Walpole, and Calton Pu. Infopipes for composing distributed information flows. In *Proceedings of the 2001 international workshop on Multimedia middleware M3W*, 2001.

[95] Munir Kotadia. National Australia Bank hit by DDoS attack. `http://www.zdnet.com/national-australia-bank-hit-by-ddos-attack-1339271790/`, 2006. Online; accessed on 26-Aug-2013.

[96] Christian Kreibich and Robin Sommer. Policy-controlled event management for distributed intrusion detection. In *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05) - Volume 04*, ICDCSW '05, pages 385–391, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1996.

[98] Kiran Lakkaraju, William Yurcik, and Adam J. Lee. Nvisionip: netflow visualizations of system state for security situational awareness. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, VizSEC/DMSEC '04, pages 65–72, New York, NY, USA, 2004. ACM.

[99] Geetika T. Lakshmanan, Yuri G. Rabinovich, and Opher Etzion. A stratified approach for supporting high throughput event processing applications. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, 2009.

[100] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[101] Ryan Layfield, Murat Kantarcioglu, and Xiaohu Li. Incentive and Trust Issues in Assured Information Sharing. In *Proceedings of the International Conference on Collaborative Computing*, Washington D.C., USA, 2009.

[102] C. Leckie and R. Kotagiri. A probabilistic approach to detecting network scans. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 359–372, 2002.

[103] Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. Optimizing state-intensive non-blocking queries using run-time adaptation. In *Proceedings of the 23rd International Conference on Data Engineering - Workshop*, pages 614–623, Washington, DC, USA, 2007. IEEE Computer Society.

[104] Bin Liu, Yali Zhu, and Elke Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 347–358, New York, NY, USA, 2006. ACM.

[105] Huan Liu and Dan Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 295–305, 2008.

[106] M.E. Locasto, J.J. Parekh, A.D. Keromytis, and S.J. Stolfo. Towards collaborative security and P2P intrusion detection. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, pages 333–339, 2005.

[107] Giorgia Lodi, Leonardo Aniello, Giuseppe A. Di Luna, and Roberto Baldoni. An event-based platform for collaborative threats detection and monitoring. *Information Systems*, to appear, 2013.

[108] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[109] Matthew V. Mahoney and Philip K. Chan. PHAD: Packet header anomaly detection for identifying hostile network traffic. Technical Report CS-2001-4 2004, Florida Institute of Technology, 2001.

[110] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[111] Mirco Marchetti, Michele Colajanni, Michele Messori, Leonardo Aniello, and Ymir Vigfusson. Cyber attacks on financial critical infrastructures. In Roberto Baldoni and Gregory Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 53–82. Springer Berlin Heidelberg, 2012.

[112] Andre Martin, Christof Fetzer, and Andrey Brito. Active replication at (almost) no cost. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011.

[113] Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, Christof Fetzer, and Andrey Brito. Low-overhead fault tolerance for high-throughput data processing systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, 2011.

[114] McAfee. In the Crossfire - Critical Infrastructure in the Age of Cyber War. http://www.mcafee.com/us/resources/reports/rp-in-crossfire-critical-infrastructure-cyber-war.pdf, 2010. Online; accessed on 26-Aug-2013.

[115] Chris Miceli, Michael Miceli, Shantenu Jha, Hartmut Kaiser, and Andre Merzky. Programming abstractions for data intensive computing on clouds and grids. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 478–483, Washington, DC, USA, 2009. IEEE Computer Society.

[116] Lory Al Moakar, Alexandros Labrinidis, and Panos K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012.

[117] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering*, pages 251–262. IEEE Computer Society, 2004.

[118] Ira S. Moskowitz and Liwu Chang. A decision theoretical based system for information downgrading. In *In Proceedings of the 5th Joint Conference on Information Sciences*, 2000.

[119] Chris Muelder, Kwan-Liu Ma, and Tony Bartoletti. Interactive visualization for network and port scan detection. In *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*, RAID'05, pages 265–283, Berlin, Heidelberg, 2006. Springer-Verlag.

[120] North Atlantic Treaty Organization (NATO). Nato Network Enabled Capability (NNEC). http://www.act.nato.int/subpages/nnec, 2011. Online; accessed on 26-Aug-2013.

[121] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, 2010.

[122] Barack Obama. Presidential Proclamation – Critical Infrastructure Protection and Resilience Month, 2012. http://www.whitehouse.gov/the-press-office/2012/11/30/presidential-proclamation-critical-infrastructure-protection-and-resilie, 2012. Online; accessed on 21-Aug-2013.

[123] U.S. Department of Homeland Security. CIPR Month 2012. http://www.dhs.gov/cipr-month-2012, 2012. Online; accessed on 21-Aug-2013.

[124] U.S. Department of Homeland Security. Critical Infrastructure Sectors. http://www.dhs.gov/critical-infrastructure-sectors, 2012. Online; accessed on 21-Aug-2013.

[125] Gülay Öke and Georgios Loukas. A denial of service detector based on maximum likelihood detection and the random neural network. *Comput. J.*, 50(6):717–727, November 2007.

[126] Stanley R. M. Oliveira, Osmar R. ZaÔane, and Yucel Saygin. Secure association rule sharing. In *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004*, pages 74–85. Springer, 2004.

[127] Shira Ovide. Tapping 'Big Data' to Fill Potholes. http://online.wsj.com/article/SB10001424052702303444204577460552615646874.html, 2012. Online; accessed on 7-Aug-2013.

[128] Christy Pettey. Gartner Says Big Data Creates Big Jobs: 4.4 Million IT Jobs Globally to Support Big Data By 2015. http://www.gartner.com/newsroom/id/2207915, 2012. Online; accessed on 6-Aug-2013.

[129] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[130] Calton Pu and Karsten Schwan. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30:25–34, 2001.

[131] Benjamin Rey, Jerome Tixier, Aurelia Bony-Dandrieux, Gilles Dusserre, Laurent Munier, and Emmanuel Lapebie. Interdependencies Between Industrial Infrastructures: Territorial Vulnerability Assessment. *Chemical Engineering Transactions*, 31:12–27, 2013.

[132] Steve Romig. The osu flow-tools package and cisco netflow logs. In *Proceedings of the 14th USENIX conference on System administration*, LISA '00, pages 291–304, Berkeley, CA, USA, 2000. USENIX Association.

[133] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. on Knowl. and Data Eng.*, 13(6):1010–1027, November 2001.

[134] Yücel Saygin, Vassilios S. Verykios, and Chris Clifton. Using unknowns to prevent discovery of association rules. *SIGMOD Rec.*, 30(4):45–54, December 2001.

[135] M.A. Shah, J.M. Hellerstein, Sirish Chandrasekaran, and M.J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering*, pages 25 – 36, march 2003.

[136] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Preemptive rate-based operator scheduling in a data stream management system. In *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, 2005.

[137] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[138] Himanshu Singh and Robert Chun. Distributed port scan detection. In Peter P. Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, pages 221–234. Springer, 2010.

[139] Kroll Advisory Solutions. Global Fraud Report, Annual Edition 2012/2013. http://www.krolladvisory.com/library/KRL_FraudReport2012-13.pdf, 2013. Online; accessed on 26-Aug-2013.

[140] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 324–335. VLDB Endowment, 2004.

[141] S. Staniford-chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph based intrusion detection system for large networks. In *In Proceedings of the 19th National Information Systems Security Conference*, pages 361–370, 1996.

[142] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.

[143] Cisco Systems. Cisco Annual Security Report. `http://www.cisco.com/en/US/prod/vpndevc/annual_security_report.html`, 2013. Online; accessed on 7-Aug-2013.

[144] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 309–320. VLDB Endowment, 2003.

[145] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[146] Nischal Verma, FranÁois Trousset, Pascal Poncelet, and Florent Masseglia. Intrusion detections in collaborative organizations by preserving privacy. In Fabrice Guillet, Gilbert Ritschard, DjamelAbdelkader Zighed, and Henri Briand, editors, *Advances in Knowledge Discovery and Management*, volume 292 of *Studies in Computational Intelligence*, pages 235–247. Springer Berlin Heidelberg, 2010.

[147] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 285–296. VLDB Endowment, 2003.

[148] Jaikumar Vijayan. Update: Credit card firm hit by DDoS attack. `http://www.computerworld.com/s/article/96099/Update_Credit_card_firm_hit_by_DDoS_attack`, 2004. Online; accessed on 26-Aug-2013.

[149] Peng Wang, Dan Meng, Jizhong Han, Jianfeng Zhan, Bibo Tu, Xiaofeng Shi, and Le Wan. Transformer: A new paradigm for building data-parallel programming models. *IEEE Micro*, 30:55–64, 2010.

[150] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[151] Tomasz Wiktor Wlodarczyk, Yi Han, and Chunming Rong. Performance analysis of hadoop for query processing. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications - Workshops*, pages 507–513, 2011.

[152] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008.

[153] F. Xhafa, J.J. Ruiz, S. Caballe, E. Spaho, L. Barolli, and R. Miho. Massive Processing of Activity Logs of a Virtual Campus. In *Proceedings of Third International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*, pages 104–110, 2012.

[154] Yinglian Xie, Vyas Sekar, Michael Reiter, and Hui Zhang. Forensic analysis for epidemic attacks in federated networks. In *Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, ICNP '06, pages 43–53, Washington, DC, USA, 2006. IEEE Computer Society.

[155] Ying Xing, Jeong-Hyon Hwang, Ugur Çetintemel, and Stanley B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, pages 775–786, 2006.

[156] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, 2006.

[157] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, 2005.

[158] Guangsen Zhang and Manish Parashar. Cooperative detection and protection against network attacks using decentralized information sharing. *Cluster Computing*, 13(1):67–86, March 2010.

[159] Hai Zhang, Xuyang Zhu, and Wenming Guo. TCP Portscan Detection Dased on Single Packet Flows and Entropy. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 1056–1060, New York, NY, USA, 2009. ACM.

[160] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. A survey of coordinated attacks and collaborative intrusion detection. *Computers & Security*, 29(1):124–140, 2010.

[161] C.V. Zhou, S. Karunasekera, and C. Leckie. A peer-to-peer collaborative intrusion detection system. In *Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication., 2005 13th IEEE International Conference on*, volume 1, pages 6 pp.–, 2005.

[162] C.V. Zhou, S. Karunasekera, and C. Leckie. Evaluation of a Decentralized Architecture for Large Scale Collaborative Intrusion Detection. In *Proceedings of the 10th International Symposium on Integrated Network Management*, pages 80–89, 2007.