AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

Department of Computer Science

MASTER OF SCIENCE THESIS

GRZEGORZ DYK

GRID MONITORING BASED ON COMPLEX EVENT PROCESSING TECHNOLOGIES

Major: Computer Science
Specialization: Distributed Systems and Computer Networks
Album ID: 196810

SUPERVISOR:
Marian Bubak Ph.D

CONSULTANCY:
Bartosz Baliś Ph.D

Kraków 2010

OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

. . . .

PODPIS

Akademia Gŕniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Katedra Informatyki

PRACA MAGISTERSKA

GRZEGORZ DYK

MONITOROWANIE GRIDU W OPARCIU O TECHNOLOGIE ZŁOŻONEGO PRZETWARZANIA ZDARZEŃ

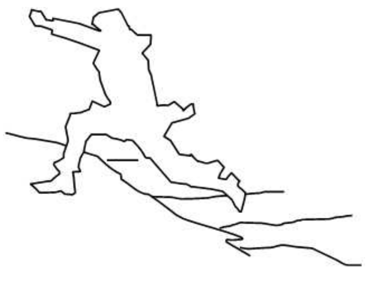
Kierunek: Informatyka
Specjalność: Systemy Rozproszone i Sieci Komputerowe
Numer albumu: 196810

PROMOTOR:
dr inż. Marian Bubak

KONSULTACJA:
dr inż. Bartosz Baliś

Kraków 2010

I express my gratitude to Marian Bubak – supervisor of this work, for his guidance, favor and advice when creating this Thesis. I also would like to kindly acknowledge my grateful thanks to Bartosz Baliś for valuable consultations, commitment, insightful look and very helpful collaboration.
This work was made possible owing to the UrbanFlood (`http://urbanflood.eu`*) project.*



Abstract

Diversity of resources and launched tasks in grid systems combined with various QoS policies defined by different Virtual Organizations produces a need for sophisticated monitoring tools. Monitoring data is required to keep QoS contracts preserved and enable grid networks provide services of high quality by assigning proper resources for tasks and optimizing their usage. Resource state information should be provided (1) on-line (2) with minimal network and computing nodes load. However, most existing monitoring systems for distributed environments do not provide on-line monitoring capabilities, but expose monitoring information in data repositories refreshed periodically. Since this is not sufficient for certain scenarios, a new approach to monitoring is required.

This Thesis presents the problem of using Complex Event Processing technologies to the following issues: (1) on-line provisioning of monitoring data and (2) minimizing monitoring overhead on resources when obtaining and transporting this data. Special emphasis is placed on distributed event processing within a monitoring system. The advantages of CEP approach to monitoring over existing solutions are discussed in this Thesis. The concept of distributed CEP is described along with problems regarding it and possible solutions. Potential benefits of applying such approach in monitoring infrastructures are given. In addition, some issues regarding resource handling in CEP-based monitoring infrastructures are identified, defined and resolved. The proposed concepts and designed solutions are practically verified by extending the capabilities of an existing monitoring framework GEMINI-2 and its subsequent evaluation for monitoring of storage resources. Results of measurements of its impact on the working environment are also presented.

The contents of this thesis are organized as follows. Firstly, some background information regarding monitoring distributed systems is given and motiviation and goals for this research are presented. Then, an overview of existing monitoring systems for distributed environment with analysis of their functionality in terms of on-line data acquisition is shown. Next, the general concept of Complex Event Processing and ways it can be used in monitoring frameworks for distributed environments are presented. After that, some issues concerning resource handling when it comes to on-line monitoring of distributed systems are highlighted. These are followed by a discusson on the concept of distributed approach to CEP with possible problems involved and solution drafts for some of them, architectural patterns and examples. Finally, description of introduced modifications in the GEMINI-2 framework concerning support of distributed CEP is given along with evaluation information and tests results.

Keywords: Complex Event Processing, Grid monitoring, on-line, network utilization, distributed CEP, resource, overhead

Contents

List of Figures 6
List of Tables 7
1. Introduction 8
1.1. Significance of monitoring in grid networks 8
1.1.1. Virtual Organizations and resource sharing 8
1.1.2. Reliability 9
1.1.3. Security 9
1.1.4. Quality of Service 9
1.1.5. Availability 10
1.2. Motivation and goals of this work 10
1.3. Requirements for the grid on-line monitoring infrastructure 11
1.3.1. Data persistence 11
1.3.2. Diverse measurement granularity 11
1.3.3. Environment awareness 11
1.3.4. Visualization and analysis support 12
1.3.5. Extensibility 12
1.3.6. On-line operating 12
1.3.7. Low overhead 12
1.3.8. Interoperability 13
1.3.9. Ease of installation and deploying 13
1.3.10. Integration with existing monitoring software 14
1.3.11. Security 14
1.4. Thesis organization 14
2. Overview of existing monitoring systems 15
2.1. GridICE 15
2.2. Ganglia 16
2.3. R-GMA 16
2.4. Inca2 16
2.5. Conclusion 17
3. CEP technology in monitoring 18
3.1. Idea behind Complex Event Processing 18
3.1.1. Event processing 18
3.1.2. Event sources 19
3.2. CEP applications 19

3.2.1. CEP and business processes 20
3.2.2. Industry 20
3.3. CEP attributes in monitoring 20
3.3.1. Simple and complex metrics 20
3.3.2. Event correlation 21
3.3.3. On-line processing 22
3.3.4. Data reduction 22
3.4. Conclusion 23
4. Resource handling 24
4.1. Resource examples 24
4.2. Resource description 25
4.2.1. Hierarchy of resources 25
4.2.2. Nature of resources 26
4.2.3. Resource type 28
4.2.4. Properties 28
4.3. Resource identification 30
4.3.1. Problems 31
4.3.2. Requirements for proper identification 31
4.3.3. Possible solutions 31
4.3.4. Connection with CEP 32
4.3.5. Identification proposition 33
4.4. Resource to event mapping 35
5. Distributed CEP 36
5.1. EPL statements 36
5.1.1. Event streams 36
5.1.2. Sliding windows 37
5.1.3. Aggregation functions 37
5.1.4. Output control 38
5.1.5. EPL examples 38
5.2. System architecture 38
5.2.1. Nomenclature 41
5.3. EPL request distribution 42
5.3.1. The problem 42
5.3.2. Distribution patterns 43
5.3.3. Distributing stream joins 48
5.3.4. Mixing patterns 50
5.4. Handling partial producers 50
5.5. Event synchronization 54
5.5.1. Data accuracy 55
5.5.2. Output time control 56
5.5.3. Error accumulation 58
5.5.4. Evaluation 58

5.6. Benefits 60
5.6.1. Beneficial EPL statements 60
5.6.2. Balanced memory usage 64
6. GEMINI-2 Development 65
6.1. Background information 65
6.1.1. GEMINI-2 overall architecture 65
6.1.2. Endpoints 65
6.1.3. The Esper CEP 66
6.2. Resource representation 67
6.3. Monitor modifications 68
6.4. Sensor environment 70
6.5. Sensor features 70
6.6. Sensor-monitor interaction 72
6.6.1. Monitor discovery 72
6.6.2. Communication channels 72
6.6.3. Control messages 73
6.6.4. Subscriptions 74
6.6.5. Communication sequence 75
6.7. Sensor architecture 76
6.8. Sampling modules 78
6.8.1. Network bandwidth sampling module 78
6.8.2. CPU sampling module 79
6.9. Distributed CEP support 80
6.9.1. CEP Engine in sensor 80
6.9.2. Request distributor stub 80
6.9.3. Event addressing 81
6.9.4. Complex event handling 81
6.10. Used technologies 82
6.11. Configuration 82
7. Evaluation 84
7.1. Performance 84
7.1.1. Memory usage 84
7.1.2. Network utilization 88
7.1.3. Event processing time 90
7.2. Storage device monitoring 90
7.2.1. Background information 90
7.2.2. Storage devices properties 91
7.2.3. Resource type definition 91
7.2.4. Event types definition 93
7.2.5. Sampling module implementation 93
7.2.6. Example 95
8. Summary 98

8.1. Functionality compliance 98
8.2. Future challenges 99
Bibliography 101

List of Figures

3.1 The CEP concept . . . 19

4.1 Example resource hierarchy . . . 26
4.2 Example resource registry content . . . 34

5.1 Centralized CEP concept . . . 39
5.2 Distributed CEP concept . . . 42
5.3 Place of request distribution in distributed CEP . . . 43
5.4 Example of distributed average calculation using unique window . . . 46
5.5 Event group to producer layout . . . 47
5.6 Event request and producer addressing . . . 53
5.7 Event synchronization and time window . . . 56
5.8 Event synchronization and length window . . . 57
5.9 Outdated output effect . . . 59
5.10 Timed output control error escalation example architecture . . . 59
5.11 Example distributed system architecture being subjected to monitoring . . . 63

6.1 GEMINI2 architecture . . . 66
6.2 Monitor architecture . . . 70
6.3 Sensor environment . . . 71
6.4 Sensor-monitor interaction diagram . . . 72
6.5 Subscription concept . . . 75
6.6 Sensor architecture . . . 77
6.7 Sampling module state diagram. . . . 79

7.1 Memory usage scenario 1 results . . . 85
7.2 Memory usage scenario 2 results . . . 86
7.3 Memory usage scenario 3 results . . . 86
7.4 Memory usage scenario 4 results . . . 87
7.5 Memory usage scenario 5 results . . . 88
7.6 Network utilization scenario 1 results . . . 88
7.7 Network utilization scenario 2 results . . . 89

List of Tables

4.1 Properties for example resources 30

4.2 Discriminators for example resources 33

5.1 EPL distribution patterns 45

5.2 Example of discrimating producer event property values 52

6.1 CPU Resource representation 67

6.3 Resource record content used in resource update message 74

6.2 Control messages 74

7.1 Common properties for storage devices 91

7.2 Unique properties of disk array resource 91

7.3 Unique properties of disk array resource 92

7.4 Resource metadata for storage devices 94

7.5 Event types for storage devices 95

7.6 Additional event types for example purposes 95

1. Introduction

1.1. Significance of monitoring in grid networks

One of the main purposes of grid infrastructure is to provide services of high quality to clients. This is achieved by "coordinating resources that are not subject to centralized control" [25]. These resources can be varied, including elements such as disk matrices for data storage or CPU-nets to carry out computing, and are usually connected through computer network. The lack of centralized control means that these resources can be shared among grid network participants in a direct way [26] by different providers using varied policies. Therefore, a successful functioning of the grid infrastructure depends heavily on maintaining a complex net of relationships between clients and resources that the grid infrastructure makes accessible to them, as well as between resources themselves (for example jobs and disk arrays).

This is a nontrivial task as, among others, security, quality of service, sharing between mutliple users and accessibility of shared resources have to be taken into account. Monitoring the state of these resources can greatly contribute to solving discussed problem in above mentioned aspects. Following sections present the most important of them in greater detail.

1.1.1. Virtual Organizations and resource sharing

Grid network resources may be controlled by organizations with diverse security and access policies. These organizations also have various goals (research, production, providing services), tasks and lines of business (pharmacy, information technologies, electronics etc.) [26]. This diversity creates a natural need to exchange resources. On the other hand, it may result in some problems: creating and maintaining contracts between organizations that comply with varied security and sharing policies can be complicated. Moreover, these organizations may be a part of one or more Virtual Organizations (*VO*) that group members with common goals and purposes, such as solving particular large scale scientific problem [44]. Just like normal organizations, each VO has it's own policies regarding security, resource access and membership.

In [26] it is stated that single resource can be used in different ways by different VOs. Conditions of resource sharing often contain constraints of how and when it may be used. This may include performance and capacity metrics, assuring quality (see 1.1.4) or security. All parameters have to be monitored by service provider to ensure that service level agreement declarations are met and that tasks invoked by one of participants do not interfere with others.

This is where monitoring software may show its usefulness. It can provide information about utilization level of each resource and enable proper reaction of entities responsible for keeping proper quality of service intact. However, in order for it to be possible, monitoring infrastructure has to provide most recent data. An out-of-date data may trigger improper reactions that try to respond to past events. Therefore, a need for efficient, *on-line* monitoring emerges here.

1.1.2. Reliability

Grid networks are increasingly being used to execute complex tasks, each composed of multiple process executions and resource access operations. Every of those actions may fail. In such case it is crucial that the entities responsible for handling task execution are informed about the causes of the problem to react properly. For example, consider a taks that is supposed to write a large amount of data to disk array installed on another node. If selected disk array happens to refuse to accept incoming data during the process, the main task should be informed about it in order to switch to other storage device [23].

In order for this to be possible, the real cause of the failure needs to be discovered. This a non-trivial task as the failure may be caused by an error of single component, such as resource incaccessibility, software exception etc. or some inconsistencies in the interaction between resources themselves. Referring to given example, the inability to send data to given disk array may be a result of broken network link, filled up disks, broken disks or not responding machine that operates the storage device. The problem becomes even more complicated when it comes to workflows (a set of ordered tasks that are invoked to achieve common goal, business, industrial or scientific in nature[19]), as the dependency graph between particular tasks can be very complex.

Detecting the cause of failure is impossible without the knowledge of current state of involved resources (disks and network links in aforesaid example). Therefore, an on-line monitoring infrastructure may contribute in this aspect.

1.1.3. Security

Authors of [12] claim that grid network security can be enhanced by proper resource allocation. That is, security issues should be taken into consideration when assigning resources to given tasks by the scheduler. Such approach cannot be achieved if proper information about resource state is available which can be provided by monitoring system. Therefore, a monitoring infrastructure may be contribute to Grid network security.

1.1.4. Quality of Service

Services provided by grid network in most cases have declared quality [27]. This quality may be expressed by various parameters, such as available throughput, CPU time, disk space etc. In fact, the nature of grid networks enables many users to use same resources at the same time. It is very common that multiple tasks are being run by single CPU. This significantly complicates the problem of defining and keeping quality of services.

Firstly, service provider must know how much it can offer at given time. For example, if network link is being used in 75% by other users' tasks, it cannot offer a new user a 45% part of maximum throughput on this link. In other words, it must be known what available capabilities of each resource are.

Secondly, the state of resources must be monitored to ensure that declared quality is kept through all the time is it being used by clients. This is important, because clients' tasks may be faulty or even malicious and try to use more "goods" than they have been assigned to. Moreover, some resources may break-down and appropriate reaction may be needed to keep declared quality (see 1.1.5 and [42]).

Both these problems cannot be solved without information about state of the resources. Monitoring middleware may be useful not only by providing raw data on state of certain resources. It also can help to estimate available resource "capacity" (how many other clients can use given resource at given quality) by applying some sophisticated queries to monitoring data. Again, resource state information timeliness is crucial here to make proper actions possible and maintain declared QoS [11].

1.1.5. Availability

One of features of services provided by grid networks is their accessibility [26]. In short, it means that given service can be used at virtually any time and anywhere. To ensure this, redundant resources are often used. In case of breakdown of one resource instance, services may chose to switch so secondary (spare) one.

Such reaction is possible if information about resource failure is available. Without it neither service itself nor humans can make decision to switch to spare one. This is where monitoring system comes in. It may provide information about state an accessibility of main resources and spare ones. Sometimes it is even possible to know that resource will fail before it actually happens (i.e. when available disk capacity is running down or number of bad sectors on given disk is becoming significant). Monitoring middleware may also provide data on designated backup resource to make sure it is available and avoid invalid, "blind" switching to resource of an unknown state.

It is quite obvious that proper resource inspection is a must to keep services accessible in distributed environment.

1.2. Motivation and goals of this work

Aspects discussed in 1.1 indicate that in many cases an on-line monitoring system would contribute to overall efficiency of a grid infrastructure. Unfortunately, such solutions usually come at price of high network and CPU utilization as a result of frequent updates of monitoring data. Therefore, in order to make them more usable some techniques of reduction of this cost have to be found and implemented. The work described in this paper aims at providing information whether and how Complex Event Processing technologies can be used to limit these handicaps.

The starting point of this work is research of CEP-based Grid monitoring [17] and implemented GEMINI-2 monitoring infrastructure. The mentioned research proved that CEP approach can be successfully used for on-line resource monitoring purposes in distributed environment. Still, some problems remain unresolved when it comes to concepts and design themselves as well as implementation of propert functionalities in GEMINI-2. This work continues the research towards CEP-based grid resource monitoring with emphasis on efficiency in terms of data rates sent over the network. **The main subject of this Thesis is an applicability and evaluation of distributed CEP in grid infrastructure monitoring**.

In detail, the issues discussed in this Thesis are as follows:

- develop a concept of distributed CEP with particular emphasis on its application in processing monitoring data in distributed environment

- introduce CEP mechanisms to sensor level of monitoring infrastructure in order to provide better data reduction

- develop resource handling mechanisms, especially identification, that are required for proper functioning of CEP-based monitoring infrastructure

- implement sensor component for GEMINI-2 framework
- introduce necessary changes into monitor-layer of GEMINI-2 in order to make cooperation with aforesaid sensors possible.

1.3. Requirements for the grid on-line monitoring infrastructure

Points mentioned in 1.1, analysis in [8] and requirements defined in [51] can be used to establish a set of funtionalities and characteristics that monitoring infrastructure should posses in order to be effective. Such list is presented in this section. While most of the listed points apply to any monitoring service for distributed system, the whole set focuses on on-line-operating monitoring infrastructures. Therefore, this list is not complete in general.

1.3.1. Data persistence

Grid monitoring should persist extracted data for future use. Retrospective analysis can be useful to detect patterns of task behaviors (i.e. how often such tasks use access to data storage), resource error proneness (i.e. frequency of spare disk usage in RAID matrices) etc. Moreover, monitoring data persistence increases monitoring infrastructure resilience. Without it, in case of failure of monitoring system entities that are interested in receiving data could not get measurement results that were taken during their interoperability. However, to make this kind of protection work measurement results should be stored as close to their source as possible to make them less vulnerable to communication layer failure.

Apart from raw monitoring data some additional information should be stored such as: time of measurement, history of measurement, request that was issued to invoke given measurement.

1.3.2. Diverse measurement granularity

Good monitoring systems should provide data on every resource that is part of distributed environment. These resources may be of any kind and granularity, from small hardware parts to whole group of machines. For example, one may wish to measure single CPU core, single CPU, single node or whole cluster. This implies that monitoring data concerning those resources is also very varied in terms of size and semantics. Therefore monitoring infrastructure should be able to accept request that concern any level of granularity. That is, consumer may request monitoring data on any abstraction level (from single, primitive resource such as CPU to whole node or cluster) and monitoring system should handle it and provide proper information that covers no less no more than selected scope.

1.3.3. Environment awareness

Distributed environment, especially Grid networks, can be seen on different levels of specificity: from single nodes, to sites to clusters to whole VOs. Monitoring infrastructure that is to work in such environment should be able to somehow grasp this complexity. Monitoring system should be aware nodes, clusters and possibly VOs to give client a better view of existing resources and provide context for gathered data. For example, a user should be informed that given monitoring results come from specific cluster or should be able to gather data from one single network node. A part of this problem has already been discussed in 1.3.2. Still, other mechanisms, such as data gathering and resource discovery [46; 30] should reflect the the complex nature of distributed environment in order to work efficiently in it and acquire enough information about it to present monitoring data to end user appropriately.

1.3.4. Visualization and analysis support

Data provided by monitoring infrastructure is often subject of analysis performed by humans. Raw text or binary format is great for machines but not for living beings. In order to efficiently and successfully carry out data analysis people should be provided with charts, diagrams, time flows, graphs (for example graphs of event hierarchy and causality) etc.

Most existing monitoring systems (see 2) have built-in means of presenting gathered data in human-readable form [18]. These kinds of features have become a standard for such applications.

1.3.5. Extensibility

Resources that are part of grid network may be very varied. It is virtually impossible to create monitoring infrastructure that covers all of them. Not only there are many different models of given resource types (for example CPUs may be *Intel i7* or *AMD Opteron*). New types of resources may appear during the lifetime of monitoring system. They also have varied "nature". Aforesaid CPU resource would be a fabric resource (a physical component). Other resources may be logical (local network, subnets etc.) or "soft" (processes, tasks).

This diversity creates a need for monitoring framework to be extensible in matter of resources it can detect, recognize and measure. Thanks to that it could be customized and adapted to specific grid network and its capabilities.

1.3.6. On-line operating

It most cases state of monitored resource changes in time, often very frequently. For example, current CPU user time may change rapidly within seconds.

Because of this monitoring data should be obtained and delivered on-line or in best case in real-time (the latter is very hard to achieve). This impacts efficiency of *probing* (process of obtaining information about state of resource) and transporting this data from measured nodes to interested parties. Both of these operations have to be very efficient to ensure that monitoring data is delivered in sensible time, before it becomes useless as state of examined resource has changed. Moreover, data should be updated frequently, providing end user with most recent information all the time.

Not meeting this requirement may result in having outdated data that has nothing to do with real situation on measured node. Some other consequences of failing to provide current data were highlighted in 1.1.

1.3.7. Low overhead

Operation of acquiring monitoring data (it will be called *probing* from now on) may affect probed resource [28; 14]. Probing available bandwidth on network link is a very good example. In this case some packets have to be sent over the examined link. This causes two problems. Firstly, sent packets reduce available bandwidth, so measurement would not be absolutely accurate. Secondly, probed network link is usually used by ordinary tasks run on grid nodes (migrating data for instance). Probing interferes with these tasks and slows them down. Similar problems occur when probing CPU load (probing consumes CPU time) or process activity or state (additional code to report process state slows them down). The same applies to monitoring jobs that are being invoked on the Grid. Application instrumentation [29] injects additional code to original program and may harm its performance.

So it is clear that special effort should be made to minimize the degree that monitoring application interferes with resources and normal tasks. This can be done, amongst other things, by optimizing probing process (lower CPU time consumption) and reduce amount of data sent over the network (low bandwidth usage, see 6.8.1 and [24]).

1.3.8. Interoperability

Interoperability is one of basic characteristics of Grid networks (see [26] chapter 3). It ensures that all participants of VO use same standard, open mechanisms for authentication, authorization, resource access, data exchange etc. This openness must span across whole grid network to enable fluid and dynamic VO creating and changing.

Monitoring infrastructure must comply with this assumption. Without it, monitoring frameworks would hider VO creation and collaboration. Therefore, it would either become less usable or highly violate Grid nature and assumptions.

For a monitoring infrastructure to be interoperable at least two requirements should be met:

1. **open protocols** - in fact, interoperability vastly depends on protocols. They are implementation-independent and define data manipulation na format without imposing specific solutions. Well defined protocol allows diverse implementations and envirinments to work together. Monitoring infrastructure should define protocols for monitoring data transport and subscription, error handling and query statements. They should be well known and defined to make implementing third party extensions and modules possible.

   These definitions should use existing protocols in grid networks, or in networks in general (HTTP, TCP).

2. **universal resource identification** - as aforesaid, grid resources may vary, so their identification within given actual organization. Disks, cpus, processes are identified in different manners not only across single organization but also within single nodes. For example, UNIX operating systems identify disks as block devices and processes with integer numbers. Some resources don't even have identifiers. Network link between two hosts is a good example. To enable interoperability, this identification must be unified, by building an abstraction layer above "native" signatures or simply using available virtual resource identification identification that is available in Grid network and used by other applications.

1.3.9. Ease of installation and deploying

Modern grid networks are vast, connecting tens of computing clusters each containing sometimes hundreds of computing nodes (see TeraGrid [1], one of the largest scientific Grid networks). Installing parts of monitoring middleware on each of them manually would be mundane, to say the least. This, in turn, may result in mistakes and failures during deployment process. Moreover, overly complicated installation and configuration may discourage from using given technology even if it is robust, efficient and secure.

Monitoring software should contain some utilities that would help deploy its parts over the network and relieve humans from this task as much as possible.

1.3.10. Integration with existing monitoring software

Virtually in every network there is some kind of monitoring software that is well adapted to nature and "quirks" of the environment it is working in. It would be very desirable to use this software in Grid monitoring. This would reduce the extensibility (1.3.5) and deploying (1.3.9) problems. Unfortunately, this can be very hard due to variety of used protocols, resource representations and architectures.

1.3.11. Security

It is not very obvious that problem of security applies to monitoring middleware in large extent. One has to keep in mind though that monitoring data sent over the network very often contains names and states of processes running on probed nodes. This creates a serious threat as unauthorized parties may intercept data containing information about network structure and its state. Such information may be very useful to prepare actions that would disturb normal activity of a network. Therefore special care should be taken to ensure that monitoring data is not accessible to unauthorized third parties.

1.4. Thesis organization

This work can be viewed as consisting of three parts. First one (chapters 2 and 3) concerns the general problem of monitoring grid networks and applications. It defines requirements for monitoring systems in distributed environments and describes the potential contribution of Complex Event Processing to monitoring solutions. A short overview of existing solutions when it comes to monitoring distributed systems is also given.

The second part (chapters 4 and 5) is an introduction to distributed CEP in monitoring in general: it defines the problem, describes potential issues and provides some solutions and proposals for them.

Finally, the third part (chapter 6 and 7) describes the development of GEMINI-2 monitoring framework, a sensor module in particular. Requirements for the sensor module are given and implementation details with regard to previously mentioned problems and solutions are presented. Moreover, features that support and help introducing distributed CEP in GEMINI-2 are outlined. Finally, evaluation tests are carried out with special emphasis on influence a sensor may have on environment.

This thesis is organized in following way. Chapter 1 is an introduction to this work, containing some background information and motiviation and goals. Chapter 2 contains an overview of existing monitoring systems for distributed environment with analysis of their functionality in terms of on-line data acquisition. Chapter 3 presents the general concept of Complex Event Processing and the way it can be used in monitoring frameworks for distributed environments. Chapter 4 highlights some issues concerning resource handling when it comes to on-line monitoring of distributed systems. Chapter 5 discusses the problem of distributed approach to CEP with possible problems involved and solution drafts for some of them, architectural patterns and examples. Chapter 6 describes introduced modifications in GEMINI-2 frameworks, including designed and implemented sensor module, and evaluation information. Chapter 7 presents some performance tests results of new GEMINI-2 components and use case regarding storage monitoring. Finally, chapter 8 contains summary including future steps in GEMINI-2 development.

2. Overview of existing monitoring systems

In this part several existing monitoring systems are mentioned and described. In each case a short overview of architecture is presented. Moreover, an emphasis is put on their compliance with functional and non-functional requirements mentioned in 1.3 and ability to support on-line grid monitoring.

2.1. GridICE

GridICE [8] is relatively new Grid monitoring infrastructure. It was designed an implemented exclusively for Grid networks in order to address its unique requirements.

GridICE architecture consists of five layers:

- Measurement - responsible for gathering resource metrics (simple and composite). It also defines abstraction of resource identification and hierarchy.
- Publisher - responsible for publishing gathered data to consumers. It creates common interface for monitored data access. GLUESchema [10] was used for data definition which enables users to perceive resources through GIS.
- Data Collector Service - stores data for retrospective analysis and is responsible for detection of new resources and disappearance of monitored ones
- Detection/Notification and Data Analyzer Services - notifies about certain events through various notification means (e-mails, SMS etc) and provides various analysis, reports and statistics
- Presentation Service - web-based graphical user interface presenting monitoring information in concise way

The main advantage of GridICE is its integration with Grid protocols and data format. It uses GLUESchema, provides a common interface for data access (second layer) and integrates with other existing monitoring applications such as Nagios [31]. Thanks to Data Collector Service it also detects new resources and can handle their disappearance. Presence of presentation layer is an additional plus. Therefore, it well meets some requirements mentioned in 1.3.

Unfortunately, it cannot provide fine grained data in an on-line manner. Using GIS and GLUESchema for data delivery can result in quite big messages that would cause significant overhead of network links. Moreover, resource detection based on periodical polling GIS. Period is usually about a day so this system cannot respond quickly to changes in network.

2.2. Ganglia

Ganglia [40] was designed for monitoring distributed systems, including Grid networks or even planetary-scale systems. It's architecture is hierarchical. Leaf nodes represent examined clusters. Each higher node represents a group of lower nodes, creating a cluster federation.

Significant emphasis has been put on performance issues. Ganglia causes low overheads when it comes to CPU (special measurement algorithms) and bandwidth usage (multicast used for transporting monitoring data). Test results presented in [40] are very optimistic. Therefore it can be used in high-performance clusters. Moreover it seems to be robust as it applies heartbeat signals between nodes.

On the other hand Ganglia does not take advantage of trusty and open Grid protocols such as GIS. Therefore its interoperability is reduced when it comes to Grid environments. It also does not have proper data storage functionality.

2.3. R-GMA

R-GMA, or Relational Grid Monitoring Architecture [21; 22], is based on Grid Monitoring Architecture. Its implementation is based on relational data model. Architecture consists of three types of components:

- Producer - performs measurements and publishes monitoring data
- Consumer - accepts monitoring data from producers. Additional operations on data are available, such as joining data from multiple producers and further publishing
- Registry - is a directory service that holds information about avaliable producers and their location. Each producer after initialization registers itself in the Registry. Consumers perform a lookup in Registry to obtain information about Producers they want to receive data from.

Almost all operations in monitoring data, such as publishing, registering in Registry, querying data are expressed in SQL. Producers, Consumers and Registry components maintain their own databases to store received data, produced data and information about producers respectively. For example, registering producer in Registry is expressed as `CREATE TABLE` clause (to publish schema of monitoring data), while requesting data from priducers is simply a `SELECT * FROM` clause. According to [21], "R-GMA creates the impression that you have one RDBMS per Virtual Organisation".

Thanks to usage of relational data model, this system can provide variously grained monitoring data. The actual content can be arbitrarily defined by `SELECT` clauses. Therefore, R-GMA meets the data granularity related requirements (1.3.2). On the other hand, using relational databases makes it harded to provide on-line monitoring data.

2.4. Inca2

Inca2 [45] aims at providing monitoring data on user-level grid functionality. Therefore, the particular measurements are launched from standard user account. Every user can define tests that will be used by the system to determine the health of examined distributed system.

The architecture of Inca2 consists of following elements:

- Reporters - eecutable programs responsible for monitoring data acquisition. Results are written to an XML file
- Agent - centralized configuration and management of the system. Responsible for dispatching reporters
- Depot - stores monitoring data that can be used to generate reports and history views. Based on Hibernate ORM
- Consumer - presents data to end user. It is a web application running on web server.

Additionally, Inca2 provides a reporters repository containing reporters. These reporters are ready to use in Inca2 deployments, making it easier and faster.

Above, in conjuntion with the fact that reporters can be relatively easily implemented added and plugged in into the system makes Inca2 very extensible and easy to deploy. Moreover, it can present both historical and current data with proper visualization. The fact that the tests are user-level grid applications makes it more secure and aware of the environment.

However, Inca2 does not provide very recent data with frequent updates. It delivers current data in form of summaries of test results. Moreover, it is hard to acquire monitoring data on single instances of very fine grained resources.

Nevertheless, the list of successful deployments that includes TeraGrid [1], and GLEON [33] indicate that it is an effective and useful monitoring platform.

2.5. Conclusion

All of listed solutions are mature and have been successfully used in various distributed environments. Probably the most notable is Ganglia, which have been installed in vast number of noted distributed systems (list of them is available on Ganglia web page).

However, none of them is able to provide on-line monitoring data in efficient way. For example, mentioned Ganglia in most cases presents snapshots of resource state over longer time period, such as 1 hour. Therefore, there still exists a room for monitoring system providing on-line data. Such system would be very attractive as it could present gathered data in a form of animated charts in real-time-like manner. It also would be useful to solve some of the issues mentioned in 1.1.

3. CEP technology in monitoring

In this chapter a problem of monitoring resources in grid environment using CEP will be generally discussed. Firstly, the idea of Complex Event Processing will be briefly presented. Secondly, main features of Complex Event Processing will be mentioned with emphasis on their application in monitoring. Then an overview of problems that must be addressed when using CEP in distributed environment will be discussed.

This chapter contains many definition and concept explanations that are required to understand further parts of this Thesis.

3.1. Idea behind Complex Event Processing

Referring to [39], Complex Event Processing (or CEP) is a computing that performs operations on *complex events*. These complex events are built or created as a result of processing simpler events. An event (more precisely an event object) in terms of CEP is a representation of real event ("anything that happens" - according to [39]) recorded for the purposes of complex processing. Authors of [43] proposed a more specific definition of event: "*event* as an object that is a record of an activity in a system."

Events are sometimes also called *messages*. In this document both names will be used interchangeably.

Figure 3.1 shows interaction and relationships between main elements of CEP concept. These elements are discussed below.

3.1.1. Event processing

All events, both simple and complex, regardless of their type and source come to one *event cloud*. Event cloud is a set of event objects that is usually unbounded in terms of time, quantity and event types. Relationships, such as timing, causality and explicit ordering are maintained between events in this cloud. These relationships make it possible to carry out a detection of specific patterns in event distribution. The activity of finding these patterns is called *event processing*.

The results of event processing are *complex events*. They may be seen as indicators of some specific correlations in set of simple events. On the other hand, they may be subject to processing by some other patterns, resulting in more complex and abstract events.

An example of event processing may be detecting storage nodes in storage cluster that are currently receiving and saving data. In this case simple events include current CPU user and system time, current load on all network links that nodes are connected to and current disk capacity of each node. If for a given period (for example 15 seconds) CPU time, network load are greater than average during last 5 hours on other nodes and disk capacity on considered

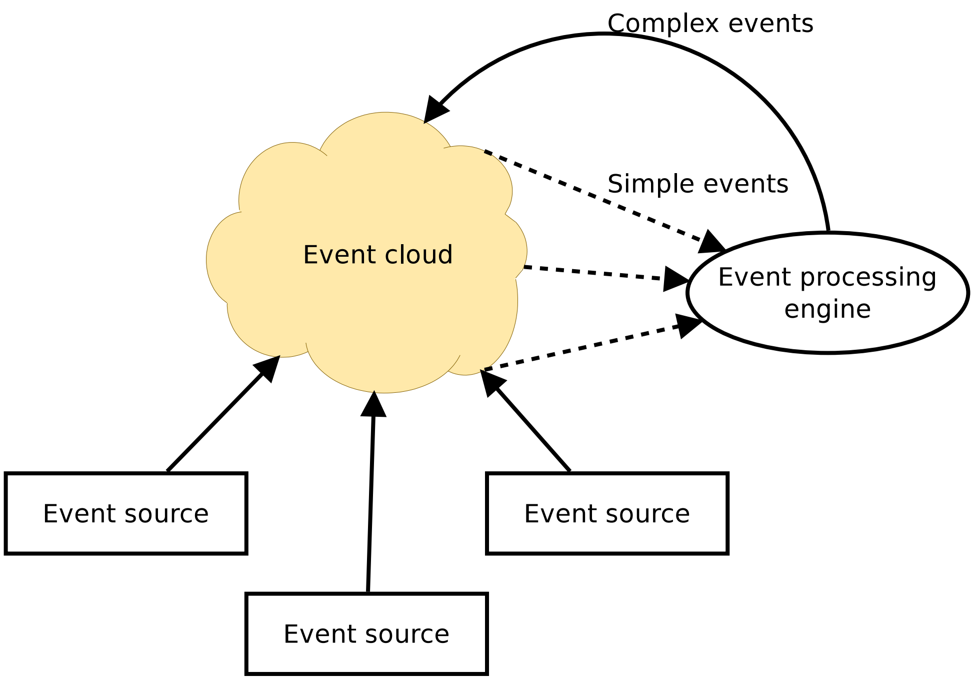


Figure 3.1: The CEP concept. Event sources produce information about particular events (simple events) and throw them into one event cloud. The contents ot this cloud is subject to processing, which aims to detect more complex information about events.

node is increasing then complex event should be created stating that this node is accepting data to store.

3.1.2. Event sources

Event sources (also called emitters - see [39]) are those elements of CEP environment that produce simple, atom events. The products of event sources come directly into event cloud and are subject to processing.

Examples of event sources include:

- physical sensors, such as RFID sensors [50], digital thermometers, photo-cells, infrared sensors, battery capacity sensor
- software sensors, like CPU system/user/idle time, disk capacity

Emitters are by definition distributed over the environment they are installed in (i.e. mentioned photo-cells or RFID sensors). This fact makes CEP perfect solution for distributed systems. Event sources may be installed on separate machines or devices. Their architecture does not have to be known by the event processing engine or third parties interested in receiving events as the only thing one should care about is the format of event objects. According to [20] CEP has already become a paradigm for development of distributed applications.

3.2. CEP applications

Despite being a quite new idea CEP has already been applied in many systems that work in various fields of economy and industry. Below are some examples of successful CEP applications. They show that this concept is not only a mere subject of scientific studies but also a working, mature solution that proves useful in real-life, deployed systems.

3.2.1. CEP and business processes

Business processes are event driven by definition. Certain events, such as creating an account, paying check, withdrawal etc. cause business process to transit to other state and trigger other events. In economy it is often important to discover specific patterns in those events. Market rate changes is a good example. This is where a problem emerges as there may be thousands of events per second coming from many different sources in different locations. Business process based architectures have not been designed to cope with such tasks. This is where CEP becomes useful. According to [38] CEP has already been applied to Business Process Management based systems. Authors mention *front running* [1] detection or monitoring loan processes as real-life CEP use cases. Another example, fraud detection, is presented in [41].

3.2.2. Industry

Industry has been vastly computerized over the last few decades. Production lines are managed by computers connected by specified network fabrics. The management itself can be viewed as exchange of events between computers and robots taking part in production process. An event may cause transition to another state of production line, contain statistical data or indicate an error.

For fabrication line management it is crucial to know what, where and why something is happening all the time. In case of failure, knowledge about what caused it is crucial for recovering from it. Analyzing the situation is hard as all of produced events come from various elements of complex system.

Using CEP to detect causality or hierarchy between events could solve the problem. Indeed, [37] gives an example of complex event processing used in silicon chip fabrication line. According to this paper, such lines "consist of several hundred computers communicating across a middleware layer". Event causality detection and hierarchy definitions are presented. All these strongly support the thesis that CEP can be successfully applied in industry.

3.3. CEP attributes in monitoring

Previous points proved that CEP can be used in different fields such as economy and industry. This section covers the problem of applying CEP to monitoring purposes in distributed applications. In other words, it shows what benefits may be derived from applying it to distributed environment monitoring systems. To answer this, CEP main features and attributes are discussed in scope of this particular field.

3.3.1. Simple and complex metrics

As mentioned in 1.3.2, monitoring infrastructure should be able to handle varied metrics granularity. In fact, each layer in distributed system (see 3.3.2) produces event with different range of information, data and semantics. For example, an application middleware layer may

[1] illegal practice of a stock broker

send events regarding state of whole transactions while persistence layer may inform about specific requests to database that are part of more complex operations.

In most cases data and granularity of metrics is fixed for each layer or element of distributed system and cannot be changed on demand. This is a problem because of two reasons:

- consumer may want to receive metrics with different granularity than given layer offers. For example, one could wish to receive information on how many select statements have been issued to DBMS within the last 2 days, while the DBMS provides information only about single statements (no counting operation available).

- consumer may wish to receive metrics that no layer or element of distributed environment offers but can be obtained as a product of existing ones. For example, one may wish to obtain data about average usage of all processors within some arbitrary time period.

The first of mentioned problems could be solved by adding appropriate aggregate types (for example sums, counting, maximum) for all event types that each distributed system element produces. However, such approach has one serious drawback. It may lead to enormous increase in number of event types that are present in monitoring environment. The second problem is really a problem of defining new event types on demand. In this case new event types exist only for the sake of single request that they are concerned with. When this request is outdated, they are no longer needed. In fact, the first of mentioned issues may be seen as special case of this one.

Both of these problems may be solved by CEP's ability to create complex events. In this case every event source would have a set of well defined event types assigned. Each event should be simple, namely it would be fine grained and indivisible. This way CEP event processor could construct virtually any kind of event from simple events, satisfying consumer's requests.

One may argue that consumer could just request all kinds of events that can be provided and extract data from them after they are received. This solution yields another problem, which is discussed in 3.3.4.

3.3.2. Event correlation

It is well known that contemporary distributed systems consist of at least several different layers. Each layer has well defined interface and responsibilities. They don't know anything but the interface of layers they directly communicate with. An OSI-ISO network layer model is a good example here. Physics layer communicates with link layer to receive data to be sent over physical connection and to pass received data.

Each layer may trigger events. Those may indicate errors or contain status and health information. Very often those events occur because some other events occurred in one of lower layers. For example, errors in TCP packets may be caused by faulty physical layer. However, in most cases none of these events contain any kind of information about what caused them. To make the matters even more complicated, a kind of layer hierarchy may be perceived. For example, almost every application working in application layer of the OSI model has an internal layer architecture. Persistence, middleware layer, presentation, client come to mind. Again, each of those may trigger events indicating errors or ordinary special actions (such as writing data to database).

Most monitoring infrastructures only collect data from those events. They are stored in databases or in log files. These kinds of representations are flat, without any information about event hierarchy or causality. To make the matters worse, different event types are usually kept in different ways. Application events may be stored in log files, database events may be in database and network and transport layer events may be stored in routers and servers. This makes it virtually impossible to detect any kind of causality between different levels of abstraction which is essential in diagnosing errors and recovering from them.

CEP’s event cloud concept may be useful here. If all events from all layers and sources where put into one bucket, detecting any relationships between them would be a lot easier. Of course, event cloud does not have to be physically one entity. It may be distributed over network nodes, servers, routers and applications. The main point is to be able to view all the events globally, regardless of their type or source.

Just creating an event cloud is just a part of success. Some kind of way of detecting relationships between events is still needed. As mentioned beforehand (see 3.1.1, CEP technology does this out-of-the-box.

3.3.3. On-line processing

One of the main problems of research described in this Thesis is on-line processing of monitoring data. CEP technology can be a powerful tool to solve this issue. In fact, CEP engines process events in on-line manner by nature. Each message that arrives to a processing engine is processed right away. Because usually events are small portions of data, the whole operations takes very short time and next message can be handled.

Moreover, CEP often relies on temporal properties of events, such as creation time. Therefore, the processing of single event cannot take long. If it did, the time-related property values would be outdated when data leaves the engine. As a result, existing implementations of CEP are very efficient.

3.3.4. Data reduction

In 1.3.7 it has been stated that monitoring infrastructure should influence the measured object as little as possible. This is especially hard when it comes to network bandwidth. The more accurate and fine grained monitoring metrics are the more bandwidth they use as a lot of simple events have to be sent frequently. Similarly, reducing bandwidth overhead usually entails receiving less accurate and up-to-date data.

This problem is not very significant when dealing with one consumer. However, when dealing with more consumers (tens or hundreds, each requesting its own set of events) or with one consumer but requesting a lot of frequent events it becomes noticeable. It is worth pointing out that even infrequent events but arriving in large bursts can be harmful as they cause peaks in bandwidth utilization.

Existing monitoring systems usually deal with this problem by decreasing frequency of sent data. For example, Ganglia (2.2) provides aggregated values over arbitrary past period. In most situations this is sufficient.

However, in at least two cases consumer can get all data, without any restrictions on granularity and without noticeable network load. These are the following situations:

- sometimes consumer would like to get some little portion of data on one single resource instance. Specific CPU user time may be an example here. In such cases amount of data transfered over network link is very small (in fact, sending CPU user time is sending one 32-bit integer number). Therefore, all required data can be sent, without any aggregations of limitations

- consumer may be interested only in filtered events. That is, only events with particular values are relevant. For example, one may want to get only information about those CPU's which have been extensively used (user time above 0.8). Number of events that meet desired criteria may be so small that sending them all over the network would not be sensible in terms of bandwidth utilization. In fact, filtering may reduce this case to the above one as only few resources may comply with filter.

The first case is simple: all monitoring data should be sent. Actually, CEP can be helpful here with its flexible approach to event types (see 3.3.1).

The second one is a bit more tricky. Usually, in existing monitoring infrastructures monitoring data filtering is done on the receiving end (R-GMA [21] is an exception here as consumer may specify its request using SQL language). Therefore, producers may send a lot of events that will be discarded by consumer right away and in fact pollute network with needless data.

Applying CEP in proper way to handle event filtering may solve this problem. Filtering itself is done by applying event patterns. Unfortunately, just applying any CEP implementation will not get rid of too much monitoring data in network. The point is to make CEP filter events as close to their source as possible.

One may consider a simple example: a producer is connected through network link to consumer. Producer wants to receive only those events that concern first of two installed CPUs and have user time value greater than 0.7. A CEP implementation may behave in two ways in terms of pattern recognition:

- filter and recognize complex events in producer and then send them to consumer
- filter at consumer's machine, just before the consumer receives data

In both cases results are equal from consumer's point of view: CEP properly filters and finds patterns in events. However, in terms of bandwidth utilization efficiency first solution is clearly better. Therefore, well configured and appropriate CEP implementation may be useful in reducing network links load. More information on this topic can be found in chapter 5.

Such case when data is filtered at source and reduced amount of it is sent over the network will be called *data reduction* in further part of this document.

3.4. Conclusion

Complex Event Processing concept appears to be very interesting solution for event-oriented applications. In fact, it has been designed for such solutions. However, many of its features and aspects described before seem to be useful in monitoring applications as well. Therefore, further part of this paper will discuss the problem of applying this technology to monitoring infrastructure in a way that supports efficient monitoring data processing and delivery.

4. Resource handling

The main subject of monitoring of any distributed application are resources. Their nature, characteristics, properties, identification etc. can be very diverse, depending on many factors such as operating system, vendor, family and so on.

In order for monitoring infrastructure to fulfill its tasks it has to work on all of these varied resource types and create unified view of them [9]. If each resource type was described or identified by different means (such as XML files, plain text files, operating system registry entries etc.) it would be very hard for humans to issue any kinds of requests for monitoring data about those resources and virtually impossible for computers to analyze it.

In general, as far as resources are concerned monitoring infrastructure has to perform all following actions:

1. discover – detect existing resources in system that monitoring infrastructure is working in.

2. describe – create uniform view on all discovered resources, presenting their capabilities, properties and characteristics. In other words, monitoring infrastructure has to create some kind of abstraction layer over native resource representation to unify any differences in it.

3. publish – send information about resources to other systems or users

This chapter covers the last two points of this list. Resource discovery is very system specific and hard to describe in general. It is presented more thoroughly in 6.8. Moreover, a problem of uniform, consistent resource identification in distributed environment is discussed. Among other things, its connection with CEP in monitoring is given.

One of the purposes of this chapter is to present decisions that have been made regarding resource identification, registering and event mapping that have been made during research. This information is very helpful to understand further chapters.

4.1. Resource examples

Following sections refer to several resources as examples. In order to clarify any inaccuracies and avoid misunderstandings below is list of them with short description.

- **CPU** - Central processing unit (or processor) - hardware element that carries out instructions. Most modern CPUs may have several cores. Similarly, each distributed system node may be equipped with several CPUs

- **network link** - single network connection between two nodes (hosts). Usually is bidirectional (full duplex). May be dynamic in terms of routing (different paths) and therefore involved network standards (Ethernet, FDDI) provided that the two hosts remain mutually accessible

- **Hard disk drive** - hardware device for storing data.
- **Process** - instance of computer program being executed. A process may be in several different states. In most cases it's executed in environment provided by operating system

4.2. Resource description

As aforesaid, each resource that monitoring infrastructure can probe may have different type, behavior and description. Resources description means defining these properties for each resource instance.

This section presents set of identified aspects of resource description and examples of ways of handling them. It is focused on describing resources in monitoring applications and is by no means complete in general.

4.2.1. Hierarchy of resources

Having a set of unordered, not systematized resources would be very disadvantageous. Operations such as identifying (consumer wants to have information about specific CPU) and searching (consumer wants to have information about all hard drives) would be very slow when performed on such "bag" of elements. Therefore some kind of resource systematization is required.

A solution to this problem proposed in this paper and referred to in further sections is a resource hierarchy. An example of such systematization is shown in figure 4.1. Hierarchy is built based on a "belongs to" relation between resources. In figure 4.1 it is shown with arrows. Thus, a CPU resource belongs to particular Host, Process belongs to Operating system it is being executed in and so on. One may say that Host is parent of given process instance and Operating System is parent of specific Process instance.

Such systematization has number of advantages:

- simplicity; it is easy to establish a "belongs to" relation between resources. In fact, very rarely this relation will span between several machines (clusters, distributed operating systems and network links come to mind as examples of resources that concern multiple nodes). Most resources will belong to resource on same machine. This reduces the amount information that has to be exchanged between nodes in distributed system in order to establish discussed relationship.
- greatly helps in identifying resources (see 4.3.5)
- enhances construction of requests to monitoring system. Querying all CPUs of given host or all cores of given CPU is natural in this structure. In fact, resources can be viewed on many levels of abstraction (single cluster, single host, single cpu).

However, this solution also has some serious drawbacks:

- structure is incoherent. This is caused by the fact that the "belongs to" relation is not coherent. Some resources do not have parents. in figure 4.1 *Network link* resource is "dangling". It cannot be assigned to one single host as it refers to two hosts. On the other hand host cannot be assigned to one network link as it may be connected to many of them

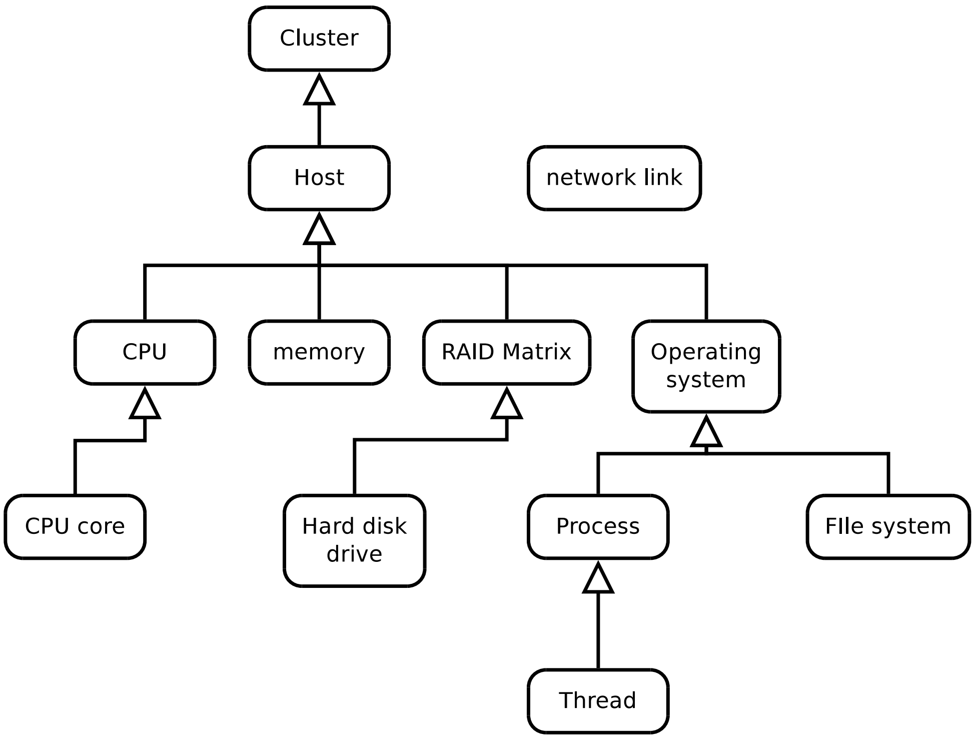


Figure 4.1: Example resource hierarchy. The arrows represent a "belongs to" relation between resource instaces. In this case, a given CPU belongs to a host it is installed on while the file system belongs to an operating system it is working in. A Network link is "dangling" because it does not belong to single instance of any other resource.

- the "belongs to" relation is ambiguous. Sensibility of relations presented in figure 4.1 can be easily undermined. For example, in case of distributed operating system it cannot belong to one single host (rather hosts should belong to this operating system). Similarly, hard drive does not always belong to a matrix and may be connected directly to the mainboard.

While first of mentioned disadvantages is not critical, the other one is rather serious. Assuming that relationships between resources are created when the resources are discovered by sensors (producers) there is a danger of incoherence between created hierarchies. One producer may decide that process should belong to operating system while other that user should be an owner. As a result monitoring environment may end up with two resource instances of same type (see 4.2.3) that are located in very different locations in hierarchy. Therefore, when applying this solution great care should be taken to ensure relationship conformity.

Regardless of mentioned drawbacks hierarchy as a way of systematizing resources will be used in further research as it proved to be useful in other aspects.

4.2.2. Nature of resources

Each resource may behave differently in terms of lifetime, frequency of state changes [34] etc. These attributes, being very abstract and fluid, usually cannot be included in properties. Such set of attributes of a resource that cannot be described in systematized way will be called nature of the resource.

Nature may compromise many aspects of a resource. Here are some examples:

- physical (mainly hardware) or virtual (mainly software - processes, operating systems, applications)
- lifetime length - is it permanent (always present in environment) or transient (appears and disappear frequently from environment). For example, processes are created and killed very frequently compared to CPU's or hard disks which do not disappear very often from environment
- there are kinds of resources that may be called *phantom*. Their main attribute is that they can not be discovered in advance. Network link may be an example. Discovering all network links in environment that monitoring infrastructure works in would be very hard therefore although they exists they cannot be kept in any kind of registry. [1]

Although nature cannot be described explicitly, it's knowledge is crucial for monitoring infrastructure to successfully serve it's purpose. For example, it is very obvious that all resources that have long lifetime should be kept in some kind of registry. CPUs, hard drives etc. can be rather easily discovered and information about them stored in one place. Such registry is useful for handling requests as monitoring system can detect which resources they concern and decide which locations system shouldquery for specific data. Moreover, having a registry of resources would enable data consumer to have an overview of all elements of distributed system that it can gain information about and therefore avoid situations when it has to send requests "blindly".

However, problems appear when dealing with short-lifetime resources, such as jobs [13]. Adding them to registry every time they appear in system and removing them each time they disappear would be very inefficient. For example, in typical multi-user operating systems several process start or die every second. This is true even more when it comes to threads. Moreover, short intervals between resource appearing and disappearing may cause any request regarding it invalid in the moment it comes to the producer. A following example may be considered: monitoring infrastructure detects that a process has started. It publishes it to all consumers. One of them, seeing it decides that it want's to know current processor usage of this process and sends proper request to producer. Meanwhile however considered process ends and producer receives request that regards non-existent resource.

When dealing with short-lifetime resources monitoring system can behave in one of a few ways:

- use minimum retention time: keep every resource in registry for some specified minimum time period. If resource disappears before this time expires, keep it anyway (with some kind of "done" status) and remove it after that time. Some kind of information that resource is expired is required here.
- treat such resources as *phantom* resources (see below). In this case resources are neither kept in registry nor they are published

Handling mentioned before *phantom* resources also poses a problem. If they cannot be discovered how can they be probed? Moreover, how can requests regarding them be handled? One of possible solutions is to treat them as on-demand resources. That is, they are discovered and created when requests regarding them arrives to producer. Referring to mentioned example of network link, producer may try to discover it only when consumer specifies request for particular resource. Since request contains unambiguous identification of resource (or it should

[1]One may argue that discovering existing network topology in advance is possible. In fact, some solutions for this problem exist, such as [35]. However, in order to work efficiently they require support from data-link and network layers which is not always available. Therefore, for this paper network links will be considered phantom.

as consumer want's to have information about particular element) it is easy to discover that specific instance of network link. An obvious drawback of that solution is that consumer has to know that given resource really exists. In case of faulty request referring to non-existent element producer should return error message.

If nature of given resource is known in advance during system design and implementation then adapting it to handle specific behaviors of this resource is quite easy. However, when system is to handle unknown types of resources or it is known that new types of resources will appear in system after it has been designed and deployed then it has to detect the nature of resources.

Detecting short-lifetime resources is not very difficult. System may observe the frequency of their appearance and disappearance from environment. If it is above arbitrary value then software may decide that given resource type should be treated as transient.

Specifying which resources are *phantom* is virtually impossible during runtime. The decision would have to be made at the moment they are discovered, without any previous information about their nature. Therefore, such resources should be identified during design and implementation. However, if monitoring system is modular and have special modules for gathering data for each resource type, these can decide whether resource is *phantom*. This way whole system can be adapted by simply adding new modules.

4.2.3. Resource type

Among all resources existing in distributed environment it is possible to distinguish specific classes or types (CPU, Hard drive, Host). These types specifies how resources are treated by monitoring system. In fact, they are very similar to object classes in object oriented programming: they describe in general resource capabilities and sets of properties. In other words they systematize and categorize resource instances in addition to hierarchy mentioned in 4.2.1.

In general, resource type help to:

- define how resource should be handled by monitoring system; in particular, it may specify resource nature (see 4.2.2)
- characterize resource in terms of properties (see 4.2.4). That is, each resource type has specific set of those
- specify event types that are suitable for given resource (see 4.4)
- identify resource instances in set of resources. Specifying resource type when trying to refer to particular instance greatly reduces amount of objects that have to be searched. For example, when trying to refer to single CPU in whole distributed environment without passing type (CPU in this case) would make monitoring system search through all discovered resources. That would be inefficient, to say the least.

Implementing resource typing is very easy. Almost always there are some kinds of "probing ends" that specialize in gathering data on specific resource type. As a matter of fact, it is hard to imagine monitoring system without any kind of resource classification.

4.2.4. Properties

One of elements of resource description are properties. Their main function is to provide consumers or any client of monitoring infrastructure information about resource state and capabilities. Table 4.1 contains examples of resource properties.

As stated in 4.2.3 a set of properties is defined by resource type. The types specify names and optionally data type of property. For example, CPU times (user/system/idle) may be written as real numbers (value from 0.0 to 1.0) or in percents (0% to 100%). Resource instance itself has specific values of all properties assigned.

Two types of properties can be distinguished in terms of their value dynamics in time:

- **static** properties are those, whose values do not change during resource lifetime. They are assigned when resource is detected by monitoring system.
- **dynamic** properties values change over time. Their actual values have to be measured periodically by monitoring system. In fact, these are object of concern of monitoring infrastructures. Their value affect actual distributed system state.

The *static* column in table 4.1 defines which properties are static.

Identifying static properties is very important for monitoring infrastructure efficiency. Such properties do not have to be probed and sent over the network as frequently as dynamic ones. Moreover, they can be held with associated resources in some kind of registry. Consumer could send a request to this registry for values of static properties it wants to know. This way the whole probing and measuring infrastructure would not be burdened with resolving such queries.

It is also possible to define a hierarchy in resource properties. In [36] authors proposed a classification of metrics for network based resources. In fact, if one analyzes properties of network link in table 4.1 a hierarchy is quite clearly visible: there are three subtypes of *bandwidth* property (*maximum*, *available*, *utilized*) and two subtypes of *loss* (one-way and *roundtrip*). Such classification has following advantages:

- it makes it easier to send more specific request for monitoring data. Consumer may specify that it wants to receive whole bandwidth information or just one type of it (say, available). It may reduce probing overhead as only requested data would be extracted
- simplifies human-perspective on properties. Tree views with collapse and expand functions would be very suitable here
- simplifies adding new property subtypes; such modification could remain transparent for all consumers that rely only on super-property

Resource type	Property name	Static	Description
CPU	vendor	✓	Name or ID or processor vendor
	frequency	✓	Maximum frequency this processor can work with
	working frequency	✗	frequency this CPU is currently working with. Modern CPUs can often change efficient frequency to save power.
	user time	✗	% of time that CPU spent in user tasks
	idle time	✗	% of time that CPU was idle
	system time	✗	% of time that CPU spent in system tasks (such as IO)

Resource type	Property name	Static	Description
Hard Disk Drive	vendor	✓	name or id of hard disk vendor
	rpm	✓	maximum speed that this disk can rotate at
	capacity	✓	maximum capacity (in bytes)
	space occupied	✗	number of bytes utilized on disk (in bytes)
	cluster no.	✓	number of clusters
Process	PID	✓	process id
	command	✓	command that invoked the process
	working directory	✓	full path to working directory of the process
	memory usage	✗	number of bytes of memory this process allocated
	CPU usage	✗	% of overall CPU time this process used
	threads	✗	list of threads that have been started by given process
	user	✓	ID of user that executed given process
	parent	✓	PID of parent process
Network link	hop number	✓	number of servers/routers that on the way between terminal nodes
	maximum bandwidth	✓	maximum achievable speed on this link
	available bandwidth	✗	bandwidth that is achievable considering current network load
	utilized bandwidth	✗	bandwidth that is currently used by any traffic on this link
	loss one-way	✗	% of packets lost in one direction (sender to receiver)
	loss round-trip	✗	% of packets lost in communication sender- receiver- sender
	MTU	✓	maximum transmission unit; largest protocol data unit that link layer can handle

Table 4.1: Properties for example resources

4.3. Resource identification

One of the main functions of monitoring system is answering to requests for data on given resource. In general, any consumer has to be able to select single CPU from hundreds or thousands scattered across network. In order to meet this requirement, system must be able to identify which resource exactly the request concerns.

Identifying resource in distributed, heterogeneous environment is not a trivial thing. One must overcome several problems. These are discussed in 4.3.1.

The main subject of this section is to propose an identification system for resources on grid environment with additional emphasis on its cooperation with Complex Event Processing. This proposal is supposed to solve problems mentioned in 4.3.1 and meet requirements specified in 4.3.2.

4.3.1. Problems

It is true that resource are already identified somehow by operating systems. They may be called *native* identifiers. For example, UNIX family systems assign numbers to CPUs, IDs to processes, special files to hard drives etc. However, these identifiers remain unique only within scope of single machine that given OS operates on. Almost every UNIX machine has hard disk drive identified as `/dev/sda`. Therefore, this "native" identification does not work in environment that consists of more that one UNIX machine.

Moreover, different operating systems identify resources that they manage in different ways. Windows using capital letter to designate disks and partitions compared to block devices and directories UNIX is a good example here. These kinds of differences of "native" identifiers renders them inefficient in heterogeneous environment such as grid networks.

Finally, some resources don't even have their *native* identifiers. Network links, clusters of nodes, cpu cores are usually the case. Still, they can be subject of monitoring requests and need to be specified somehow.

Clearly, mentioned problems cause a need for additional, unified identification system to emerge.

4.3.2. Requirements for proper identification

Heterogeneous and distributed nature of grid networks impose several requirements on resource identification system. They are named and discussed below.

- unambiguity – resources have to be identified uniquely within scope of whole grid network. That is, if they are to have an id assigned, it should be unique in whole environment.
- extensibility – identification system has to be able to handle new types of resources that may appear in monitored environment. This is somewhat a derivative of monitoring system requirements described in 1.3.5.
- versatility – any type or resource must be able to be designated by considered identification system. A situation, where a special identifiers have to be assigned to some small group of peculiar resources is unacceptable as it would lead to needless complication and confusion.
- human readability – while this is not a critical feature, human readable resource identification could help in satisfying the requirement discussed in Section 1.3.4. In fact, many native identifiers are understood by humans (drive letters in Windows, block device names in UNIX, host names)

First of mentioned requirements is crucial. Failing to meet it makes the whole identification idea pointless.

4.3.3. Possible solutions

Taking into account the requirements mentioned in 4.3.2 several solutions can be proposed:

- user numerical ids: this is very similar to artificial primary keys in relational databases. Each resource can have a unique number assigned. Clearly, this solution meets all of requirements named in 4.3.2.
- use native identifiers. The problem of ambiguity of native ids can be overcome by assigning a unique prefix that comes from host. It may be host's IP address or name. For example, one could specify a `/dev/sda2` drive by `192.168.5.43./dev/sda2`.
- identify by static properties. In this case, consumer specifies values of properties that desired resource should match. For example, a request may contain vendor property with value "Intel", frequency with value "2 GHZ" and number of cores equal to 2. Monitoring system shoud match all CPUs that are made by Intel, have 2 cores and work with maximum frequency of 2 GHz.

First solution is very good in terms of meeting mentioned requirements. Numbers are universal and can be used with any type of resource. Moreover, using GUID (Globally Unique Identifier) would make it easy to assign unique IDs. However, the fact that the numbers are hardly human-readable (especially GUIDs) makes this solution less attractive.

When it comes to the second one, it creates unique IDs at relatively low price. Unfortunately, it cannot handle resources that have no native identifiers.

The last one is quite promising. It successfully solves the problem of extensibility and handling any resource type by using properties that are assigned to all kinds of resources. However, addressing by properties is usually not unique. There may be several instances of resource with same property values. It may be hard to find such a set of them that would uniquely define an element that consumer is interested in. Moreover, in order to use this kind of identification consumer must know all static property in advance to construct proper query. Therefore, it would have to obtain data about all existing resources in monitoring environment, resulting in big amounts of data being sent over the network. Finally, comparing set of properties can be takes longer than simply comparing an id. This is especially true when hierarchical properties are involved. In spite of a number of drawbacks this proposition may be an interesting addition to primary identification.

Apparently, none of mentioned solutions is satisfying enough. Therefore, another identification system was designed. It is described and discussed in 4.3.5.

4.3.4. Connection with CEP

Chapter 3 discussed the general possibility of applying CEP for monitoring purposes. One of conditions that have to be met is that resources are identified in uniform, consistent way. This can be justified by a number of facts.

Each simple event in CEP-based monitoring infrastructure concerns one single resource instance. All these events are to be put into one common event cloud. As mentioned before, within this cloud various operations can be performed. Most important ones from the perspective of this paragraph are correlations. If each of these events, staying in common event cloud, referred to resource it concerns in a different way (for example, events produced by hosts operating on Windows had references to disks noted with capital letters while UNIX-based hosts used mount directories) finding a correlation within them would be very hard (referring to given example, correlating all events that regard system partition would be difficult).

Moreover, because all the events "live" in same space name, without unique identification conflicts may occur. For example, on each machine CPUs are uniquely marked with integer

Resource type	Discriminator	Example
Process	PID	34434
CPU	Cpu Number	1
Disk drive or partition	name of device in operating system	*C:* for Windows, */dev/sda5* for Unix
Host	IP Address	192.168.2.39
Network link (between two nodes)	ip addresses of two ends	192.168.2.69-192.168.5.33

Table 4.2: Discriminators for example resources

number. However, in aforesaid event cloud these numbers are no longer unique. Without any change in identification, this cloud would contain more than one event referring to CPU number 1 causing incorrect correlations.

Unique and uniform identification systems is also important for following reasons:

- it creates a common "resource space" that correspond very well to event cloud. All resources can be thrown into one "bag" and still can be identified. Moreover, client may search for resources within this "bag".
- filtering operations on events are easier. Client could specify that it wants to receive data on events referring to particular id
- it makes data reduction more effective (see 5.6 and 5.4).

Therefore, a unique and uniform identification is essential for monitoring systems based on CEP approach.

4.3.5. Identification proposition

Proposed identification system takes advantage of hierarchy resource systematization described in 4.2.1. The main idea is that the whole resource context ordered in hierarchy can be viewed as forest data structure (set of trees). Identifying single node in such structure is nothing more than specifying a path from tree root to desired node.

Figure 4.2 presents an example of contents of resource registry. There are two hosts, each equipped with two CPUs. CPUs of the first one have two cores, while the other's are simple, one-core processors.

In order to point to second core of the first CPU of first host one may specify a path:

$$host \rightarrow cpu \rightarrow core$$

However, such path is unambiguous. There's no information about which host and which CPU instance it concerns.

In order to point to specific instances within single resource type (in this case given host among presented pair of hosts, or single CPU within given host) native identification can be used. Native identifiers that are used to discriminate resource instances among children of same parent will be called *discriminators*. If no such identifier is available (ref. 4.3.1, artificial discriminator can be created. Table 4.2 contains discriminators for some example resources.

After adding discriminators the considered path in resource tree has following form:

$$host[192.168.2.31] \rightarrow cpu[1] \rightarrow core[2]$$

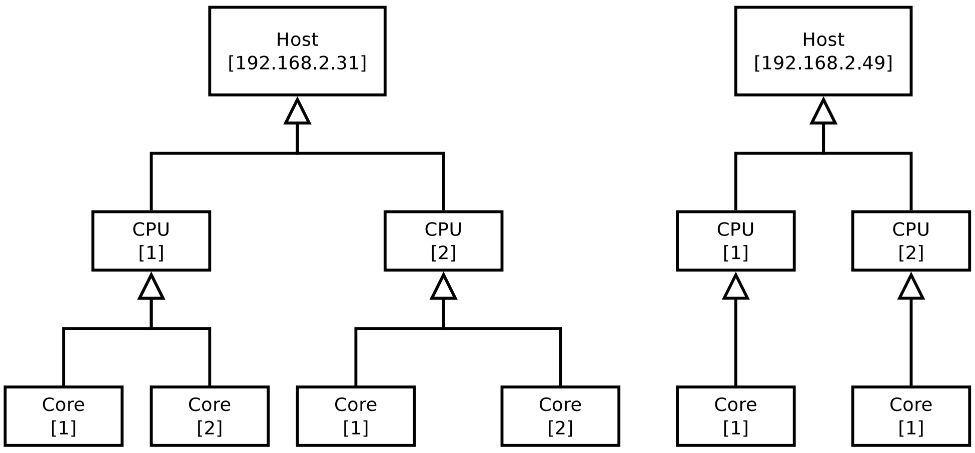


Figure 4.2: Example resource registry content. There are two Host type resource instances with CPU resources assigned to those nodes that they are installed on. Resources of same type can be identified by their native identification. In this case, it is an IP address for hosts and number for CPUs.

This time the identifier is unambiguous and uniquely points to desired resource instance. In general, resource id can be build with following algorithm:

1. determine the root of tree that considered resource instance belongs to. Attach its type and proper discriminator to the beginning of the path
2. go down the selected tree to selected resource instance and attach each intermediate resource's type and discriminator

It can be shown that presented identification system meets requirements mentioned in 4.3.2. First of all, it is unambiguous, provided that both relations in resource hierarchy and root of the tree can be determined uniquely. Versatility comes from the fact every resource have type and discriminators can be created easily for any kind of resource (in extreme situations plain number may be assigned). Therefore, there are no problems in creating single path node for any kind of resource. Moreover, new resource types can be handled added by just specifying their type and discriminator. Finally, identifiers created in this way are quite well readable.

This solution also has some special advantages:

- unambiguity is obtained at relatively low price. Virtually no data has to be exchanged between distributed producers. This is achieved by taking advantage of native identifiers which are already unique at given scope. IP addresses that discriminate hosts are a great example here.
- hierarchical nature of identifier makes it easier to identify groups of resources and thus make it easier to create certain types of requests. For example, specification of all CPUs of given host could have following format:

  $$host[192.168.2.31] \rightarrow cpu[*]$$

  where $*$ means *any*. However, to take advantage of this possibility additional problems should be solved when it comes to event manipulation in Complex Event Processing.

The identification system that is presented in this section is used in further research. Due to technical reasons, such as event marshalling and parsing, different notation is applied with dots instead of arrows. Therefore, in further part of this document resource IDs will be written similarly to the following example:

$$host[192.168.2.31].cpu[1].core[2]$$

4.4. Resource to event mapping

When considering resources in monitoring system that uses Complex Event Processing one more problem has to be discussed: how they should be viewed in terms of their relation to corresponding event messages. In other words, a question has to be asked whether there should be exactly one event type assigned to one resource type or should this relation be more loose.

In fact, four different kinds of relationships between resources and event types can be distinguished:

- **one-to-one**: every resource type has exactly one type of event assigned. Preferably, this event contains all information about resource, including static and dynamic properties.
- **one-to-many**: each resource may have a unique set of event types assigned. For example, sensor that takes care of probing CPU may create messages containing only dynamic and only static properties. Each event type is assigned to exactly one type of resource.
- **many-to-many**: same as one-to-many except that some event types may correspond to more than one resource type. A message containing vendor information may be an example as it can be applied to virtually any kind of resource.
- **many-to-one**: there is one single message format that carries information about any type of resource

Depending on which of these four types of relationships is chosen consumers will perceive particular resources differently. In case of one-to-one relation, there is no need to specify desired message type in requests. In fact, in this case consumer sees resource quasi-directly, without caring about message types. In following two solutions consumer specifies message type it wants to receive rather than resource. Thus, message content is decoupled from resource properties. Moreover, both these relationship types make it possible to separate static properties from dynamic ones by creating different message types for them. This, in turn makes it possible to reduce network traffic as described in 4.2.4.

Last of mentioned relationship types requires a generic event message to be designed. Such message would have to contain a list of property names with their values and datatypes for proper interpretation. One of its advantages is fairly large size after serialization, as information all mentioned information (names and datatypes) would have to be sent.

From presented solutions many-to-many relationship has been chosen for further research as it is simple in implementation while being versatile enough.

5. Distributed CEP

Chapter 3 discussed the way that main concepts of CEP can be used in monitoring infrastructures. However, the real usefulness of this technology in such applications is closely related to the implementation of the CEP idea. This chapter presents a distributed approach to CEP with special emphasis on applying it in monitoring infrastructures for efficient, on-line operating.

First, the architecture of distributed CEP oriented system is presented with its advantages and disadvantages. Secondly, the problems that have to be faced when designing such systems are presented along with solution suggestions. Finally, the benefits of using this architecture are mentioned.

5.1. EPL statements

Before any further research is presented, one must first become familiar with defining event patterns in CEP. They are used to define complex events and can be specified as statements in special language called Event Programming Language (or EPL). There are many dialects of EPL depending on implementation of CEP. The research that is presented in this Thesis uses Esper [4] CEP implementation. Therefore, Esper EPL dialect [3] will be referenced as an example of EPL abilities and features. It is worth noting that these operations are not strictly tied to EPL. They can be easily expressed in any other language.

This sections covers only a few aspects of EPL capabilities that are crucial to understand the problem of distributed CEP that is discussed in this chapter. Full reference can be found in [3]. Reader should refer to it in case of any inconsistencies or vaguenesses in following text.

5.1.1. Event streams

EPL statements operate on data structures called event streams. An **event stream** is a named, ordered, infinite sequence of single events of same type. The order in event stream is based on event creation time. The older events (created earlier) are first in the stream. The infinity comes from the fact that event sources (see 3.1.2) in most cases produce events continually, without permanent breaks. The name of the stream is arbitrary, given by system users or system itself. It is used to distinguish event streams carrying the same event types but coming from different sources.

An event stream may be created in one of two ways:

- in event sources, by producing given event type. In this case, all events of same type automatically form single stream

- by applying event patterns and creating complex events. Complex events that are result of event pattern form new event stream.

Event streams may be joined, split or filtered. Stream joining means forming single event stream from two or more other streams. Getting events containing process pid and command line with information about user that invoked this process, provided separate events about processes and users are generated, may be an example here.

Stream splitting is an oposite operation to join. It extracts information from one event stream and forms two or more new ones. For example, two separete consumers may want to have information about CPUs. However, one of them wishes to receive events about only one of them while other one about all of them. Thus, two separate event streams have to created: one containing event that concern only specific CPU and other containing all events.

Filtering streams involves rejecting events that do not satisfy some arbitrary condition. For example, one may spacify that only events that have a given value greater than 10 should be involved in further processing.

5.1.2. Sliding windows

Stream of events that arrives to CEP engine is by definition infinite. Performing any kind of group operations on such data structure is impossible.

Sliding windows solve this problem by selecting a limited subset of incoming events. The criteria by which events are put into windows can be divided into two categories.

- **time-based** - events are selected by the time of their creation (arrival to CEP engine). For example, a window may contain only events from last 3 minutes. If at some point any event turns out to be older than specified time period it is removed. Simultaneously, new events that have just arrived are put into window. Therefore, time window "slides" through event steam with regard to time.
- **size-based** - windows that have limited event capacity. If window is full and new event arrives, the oldest of events is removed to make room for the new one. Thus, windows slides with regard to size.

Each window can be additionally filtered by arbitrary condition. For example, one may want to have only events from last 4 minutes that come from producer with given name.

5.1.3. Aggregation functions

Aggregation functions in EPL work in similar ways as in SQL applied in relational databases. They are invoked on group of events. This group is usually specified by windows (group is equal to all events in window). The result is a new complex event containing the outcome of aggregation function.

Examples of such functions include:

- **sum** - returns sum of given field of group of events.
- **average** - arithmetic average of given property of group of events
- **maximum value** - returns largest of of values taken from specific property of event group
- **element count** - number of non-empty elements

In fact, the syntax and semantics in EPL of these functions is very similar to their counterparts in SQL.

5.1.4. Output control

EPL enables user to control the number of events that are generated by the statement at one time and frequency of these outputs. For example, it is possible to specify that 1, 10 or any arbitrary number of all accumulated events should be released each given time interval passes (for example: 10 seconds, 2 hours etc.). This feature is provided by the `output` clause.

5.1.5. EPL examples

Below are some examples of EPL statements along with description of their results. They may be useful to grasp the EPL syntax and semantics and understand some event pattern detection capabilities of CEP.

Statement	`select avg(cpu.idleTime) from CpuInfo.win:time(2 sec)`
Description	Calculate average idleTime from CpuInfo messages from last 2 seconds
Statement	`select max(cpu.idleTime) from CpuInfo.win:lengh(200)`
Description	Get maximum idleTime from last 200 CpuInfo messages
Statement	`select avg(cpu.idleTime) from CpuInfo.win:time(2 sec)`
Description	Calculate average idleTime from CpuInfo messages from last 2 seconds
Statement	`select avg(cpu.idleTime) from CpuInfo(userTime > 0.5).win:length(100)`
Description	Calculate average idleTime from CpuInfo messages from last 2 seconds that have userTime above 50%. It is worth pointing here that only those events that meet this condition will be put into length window and occupy space in it.
Statement	`select avg(cpu.idleTime) from CpuInfo.win:time(4 sec) output last every 2 seconds`
Description	Calculate average idleTime over a time window of 4 seconds. Every 2 seconds return the latest result.

5.2. System architecture

In 3.1 a general idea of CEP was presented. However, the concept of CEP can be implemented in many ways, using different types of architectures.

The simplest and most corresponding to CEP idea is centralized architecture. An example of such architecture is shown in figure 5.1.

The arrows in this figure represent event streams coming from event producers and from CEP engine.

Event producers are implementation of event sources (see 3.1.2). They collect produce single simple events and send them to one well known location called **monitor**.The event messages themselves are usually sent over the network.

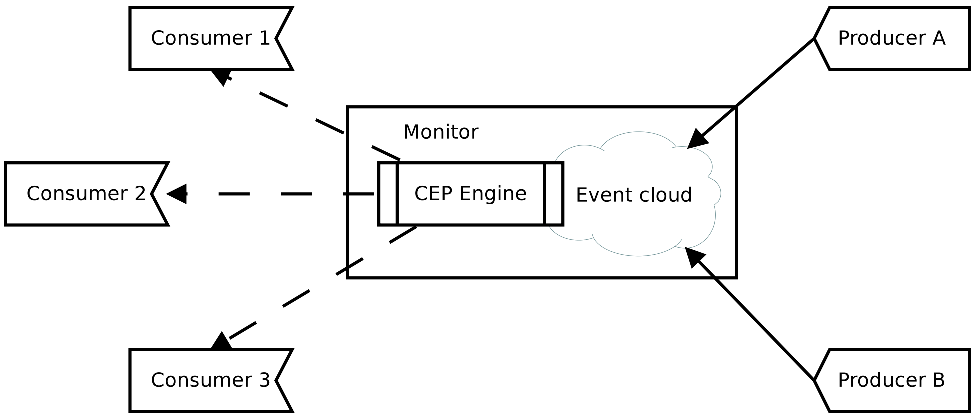


Figure 5.1: Centralized approach to Complex Event Processing. All producers are sending events to one single physical event cloud handled by one CEP engine. All consumers receive complex events from the same engine.

In order to perform event processings, a monitor has a CEP engine installed. The **CEP Engine** is part of the system that deals with event pattern detection in incoming events. This is a heart of whole CEP-based application. It accepts EPL statements, compiles them and constructs state automata (as far as Esper implementation is concerned) that are used to detect complex events. Therefore, a CEP Engine consumes and aggregates simple events to produce complex ones. Those are sent by the monitor to consumers. The detection and pattern matching are performed in one place on all arriving events. In this manner the centralized CEP corresponds to the general CEP idea very well. There is one single event cloud that contains all events that exist in the environment and one Event Processing Engine that carries out all operations on them. In other words, the centralized CEP architecture is a nearly direct implementation of concept presented in figure 3.1.

The event consumers are applications that accept events produced by the CEP Engine. In most cases they also specify event patterns, usually written in EPL, that they are interested in. Event consumers do not have to know anything about producers or the way they are handled. They only have to know the definitions of available event types in the system.

A typical scenario in centralized CEP architecture may look as follows:

1. all producers start sending events to CEP Engine. They work all the time, producing events periodically and therefore creating streams

2. a consumer sends a request to the monitor in the form of an EPL statement, defining event patterns it wants to have detected. The monitor passes the statement to its CEP engine

3. the CEP engine receives statement and starts detecting complex events in arriving event streams. These events are then sent by the monitor to the consumer that sent the request before

All operations regarding EPL compilation and event detection (in point 3) are provided by CEP implementation. Therefore, when implementing centralized model of CEP, one has to deal only with request protocols (from consumer to CEP engine) and event transportation.

Unfortunately, presented solution has a number of disadvantages:

- one single central point makes this kind of architecture very vulnerable to breakdowns. If monitor fails, the whole system will stop working

- if there are many producers and many consumers the monitor may be significantly loaded in terms of CPU and memory usage (event pattern detection and event storing for aggregation purposes). Although most CEP implementations (including Esper) can handle hundreds of thousands of events in one second, the communication interface (usually network) and processes concerned with event serialization may become a bottleneck.

- the network links between the monitor and event producers may be significantly loaded as they carry all the simple events existing in system environment. It also makes data reduction (see 3.3.4) impossible

These factors, especially the first an the last one, make discussed architecture less usable for monitoring purposes. First of all, the monitoring infrastructure that is discussed in this Thesis is supposed to work in grid environemnt which should be robust and accessible. Secondly, as stated in 1.3.7 such system should affect the performance of whole network as little as possible. Therefore, a different approach to CEP architecture should be taken.

In figure 5.2 an alternative architecture is presented. Instead of one single monitor and one event cloud there is a number of them, connected together. The consumers still send their request to and receive events from one monitor and view the whole system just as if it was centralized. However, the mechanism of collecting simple events and handling event cloud is very different.

First of all, events created by producers do not come to final processing directly. They are subjected to several intermediate processings carried by other CEP engines. Each CEP engine operates on separate event cloud formed by events carried by streams that are sent to this specific engine. Similarly, each engine views arriving streams uniformly, regardless their source. In other words, at stream receiving point it has no meaning whether it comes from CEP engine or directly from event source. For example, in figure 5.2 engine 1 receives three streams: directly from producer D and two that are results of processing by engines 2 and 3. Similarly, each intermediate engine sees engines that receive streams from it as ordinar consumers that can issue requests. For example, in figure 5.2 the engine 1 is viewed by engine 2 as consumer, in the same way as engine 1 views consumer 1, 2 and 3. Therefore, if producers can accept requests just like monitors, a communication uniformity exists and the actual complexity of architecture is hidden from each monitor.

In order for this architecture to work, each EPL statement that is sent between intermiediate CEP engines has to be properly related to a request that arrives from consumer to the front-end CEP engine. These partial statements are constructed as a result of analyzing original one. In addition to this, a new EPL statement is created that will be used to process incoming events. Such process will be called **request distribution** as it creates sub-statements that should be distributed over other engines. It is extensively described in 5.3. The request distribution is carried out in monitors.

Typical usage scenario, from issuing a request from consumer to receiving events by it, in distributed architecture presented in figure 5.2, may look like this:

1. just as in centralized architecture, all producers send events to assigned CEP engine periodically

2. consumer 1 issues a request that concerns events produced by producer Producer A, C and D. Consumer does not have to know what types are produced by which sources. It just has to know that those types exist in system

3. Engine 1 examines the request and realizes that it has to send sub-statements to engine 2, 3 and producer D.

4. Engines 2 and 3 receive requests and both start sending events coming from producer A and C accordingly

5. Engine 1 receives events from engines 2 and 3 and uses its assembly statement (see 5.3.1) to construct a final result: events requested by consumer and send them to this consumer

Obviously, this scenario is more complicated than the one in centralized architecture. There are several steps of statement processing and additional process of request distribution. Moreover, some other problems have to be solved when implementing this solution. These are discussed in 5.5. However, by the price of complexity some important advantages can be obtained:

- there is no single central point. That means no point is extensively loaded with event processing. Instead, the whole processing task is distributed obver a group of nodes. Moreover, failure of one CEP engine does not necessarily mean that the whole system must stop. While the front-end monitor (monitor 1 in figure 5.2) is a kind of single point of communication between consumers and other monitors the system could be adapted to its failure by implementing a fallback mechanism that would reroute all request issued to this monitor to other ones. The implementation details of such solution will not be discussed in this work. Moreover, not all consumers will be always connected to same monitor. As a result, failure of one front-end monitor does not mean detachment of all consumers from monitoring service.

- scalability; by taking advantage of the communication uniforimity between monitors, producers and consumers, any producers or consumer can be replaced by monitor. Therefore, the area of network covered by monitoring infrastructure can be easily extended and performance of event processing increased.

- data reduction support (see 3.3.4).

5.2.1. Nomenclature

Following sections of this chapter refer to some concepts concerning distributed CEP architecture. Below are their definitions:

- **partial producer** - a producer that from point of view of particular monitor creates events that will be processed by an assembly statement. For example, in figure 5.2 in case of monitor 1 partial producers are: monitor 2, monitor 3 and Producer D. Partial producers will also be called *direct producers* as they are those producers that send events directly to given monitor

- **assembly monitor** - monitor that is using an assembly statement to process incoming events. Each partial producer has its assembly monitor which is the monitor it sends its events to. In figure 5.2 monitor 1 is assembly monitor for monitor 2, monitor 3 and producer D, while monitor 2 is assembly for producer A and B.

- **front-end monitor** - monitor that receives requests directly from consumers. The consumers see the whole CEP-based monitoring infrastructure as it was only this one monitor. In other words, front monitor hides the complexity of monitoring infrastructure from the consumer. A front monitor exists in context of given consumer. Thas its, a monitor that serves this role for some client may be only intermediate one (on processing path) for requests sent from some other consumer.

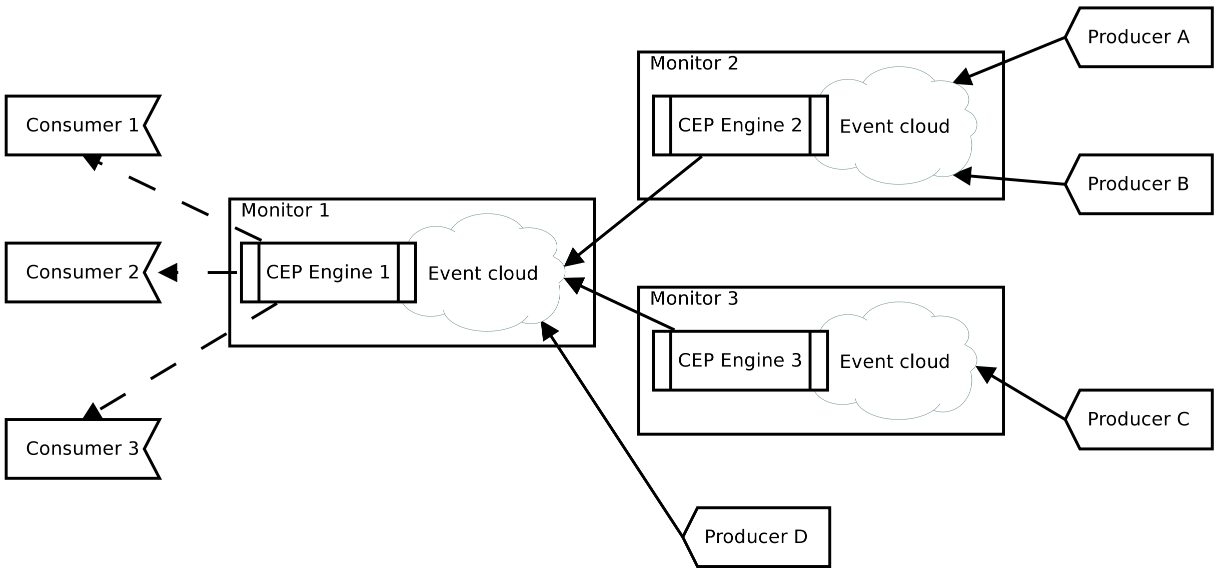


Figure 5.2: Distributed approach to Complex Event Processing. The event cloud is distribtued over multiple CEP engines. Each producer sends event to one designated event cloud. Results of processing performed by one CEP engine may be an input for another one. Consumers may receive events from arbitrary monitor.

- **intermediate monitor** - monitor that performs processing that is a part of resolving of given request but is not a front-end monitor. In figure 5.2 monitor 2 and monitor 3 are intermediate ones from the point of view of monitor 1 when resolving requests from any of shown consumers.

5.3. EPL request distribution

As stated in 5.2, distributed CEP requires a process of analyzing EPL statements and dividing them into sub-statements. In this section this particular problem will be thoroughly discussed.

5.3.1. The problem

The main idea of request distribution is to detect parts of EPL statement that can be resolved independently by different monitors (more specificly: separate CEP engines).

Figure 5.3 shows how request distribution process interacts with other parts of distrbuted CEP architecture presented in 5.2. An EPL request sent from consumer is subjected to request distribution process before it gets to CEP engine. This process creates two types of result. One is an EPL statement called **assembly statement** that will be used by CEP engine to process streams coming from producers. The other are partial requests that are to be sent to individual monitors that should carry out sub-processing.

Below is an example of request distribution. First, the original statement that is sent from consumer:

```
select avg(CpuInfo.userTime) as avgTime from CpuInfo;
```

The result of applying this statement in CEP engine would be an average user time of all events containing data about CPUs existing in the environment that system is working in.

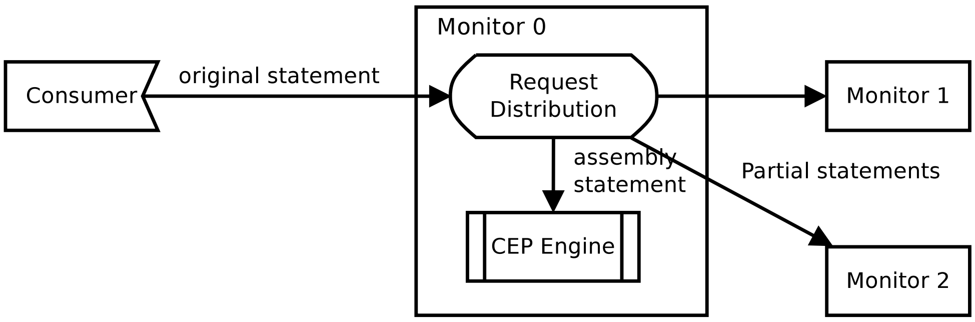


Figure 5.3: A place of request distribution in distributed CEP. The statement received by Monitor 0 is subject to distribution that is supposed to detect parts that can be delegated to other monitors. The results of this process are partial statements that are sent to designated monitors (those that should take part in resolving request) and an assembly statements that will be used to process the results received from them.

The calculations of this average can be distributed over multiple monitors by first summing the usertTime in intermediate CEP engines and calculating average of these sums in final monitor. Therefore, the partial statements would be:

```
select sum(userTime) as s, count(userTime) as c  from CpuInfo;
```

and the assembly statement:

```
select sum(s)/sum(c) as avgTime from
   partialStream.win:length(numberOfPartialProviders);
```

The form of partial statement is obvious: it counts sum and number of given factors to make it possible to calculate an average later. The assembly statement is more complex. It uses length window to limit the number of events that aggregation functions are working on to equal to number of direct providers of events (other monitors that perform partial processing). If there was no limitation, the average would be calculated using invalid data set that could include too many events or old events. In 5.3.2 specific patterns of distributions are discussed in details.

It is worth noting that presented assembly and original statements produce same type and format of events (including name of parameters, in this case *avgTime*). This is one of conditions for a request distribution to work properly. In general, the requirements are:

- format of events that are result of assembly statement must match the format of events that would be produced by plain original statement. This way transparency of request resolving from the consumer point of view is preserved
- the overall outcome of assembly statement must be same as of the original statement. That is, the event sequence and content must be same as those of original statement being processed in centralized architecture

Following parts cover the problem of statements distribution in more general way.

5.3.2. Distribution patterns

Not every EPL can be effectively distributed. For example, trivial statements can be considered, such as:

```
select * from CPUInfo;
```

It simply requests every single event message concerning CPUs. There is no place where the distribution could be performed. Obviously, in this case the intermediate monitors would be simply passing all individual events, just as the assemblying monitor [1].

In order for a statement to be suitable for request distribution, at least one of following operations must be present in it:

- aggregate function - aggregating could be performed on intermediate monitor (closer to the source of events) or distributed over multiple ones (just as in example in 5.3.1)
- time or length windows - windows can be partially filled on intermediate monitors
- grouping - the *group by* clause in EPL, discussed in next parts of this chapter
- filtering - the *where* clause in EPL or stream filtering. Filtering can be placed as near the origins of the events as possible to prevent unwanted messages being sent over the network

Meeting the above requirement does not necessarily mean that a statement can be properly distributed. The question whether given EPL statement can or cannot be beneficially broken into sub-statements is an advanced problem. However, some general patterns of EPL statement distribution can be identified. They are presented in table 5.1.

Every assembly statement refers to `assemblyStream`. This is the name of the stream that is formed by events coming from producers as a result of processing partial statements.

No	Request	Distribution
1	`select avg(value) as [alias] from [stream];`	Partial statements: `select sum(value) as s, count(value) as c from [stream]` Assembly statement: `select sum(s)/sum(c) as [alias] from assemblyStream.std:unique(producerSpec);`
2	`select count(value) as [alias] from [stream];`	Partial statements: `select count(value) as  from [stream]` Assembly statement: `select sum(c) as [alias] from assemblyStream.std:unique(producerSpec)`
3	`select max(value) as [alias] from [stream];`	Partial statements: `select max(value) as mx from [stream];` Assembly statement: `select max(mx) as [alias] from assemblyStream.std:unique(producerSpec);`

[1]While statements similar to presented one can not be distributed in sense presented in this work, the results created by them can be buffered on intermediate monitors. In some cases events arriving from multiple producers may flood the monitor. Therefore, each monitor could normalize the flow of events by releasing only latest of them with given frequency.

No	Request	Distribution
4	**select** [groupAttribute], aggregate(**value**) **as** [alias] **from** [stream] **group by** [groupAttribute];	Partial statements: **select** [groupAttribute], aggregate(**value**) **as** [alias] **from** [stream] **group by** [groupAttribute]; Assembly statement: **select** [groupAttribute], [alias] **from** assemblyStream;
5	**select** attribute **as** [alias] **from** [stream] **output** [quantity] every [frequency]	Partial statements: **select** attribute **as** [alias] **from** [stream] **output** [quantity] every [frequency/n] Assembly statement: **select** attribute **as** [alias] **from** assemblyStream **output** [quantity] every [frequency];
6	**select** [aggregate] **as** [alias] **from** [stream].win:**time**([**time** spec])	Partial statements: same **as partial** for normal aggregate but over **time** window with [**time** spec] Assembly statement: same **as** assembly for given aggregate
7	**select** [aggregate] **as** [alias] **from** [stream].win:length([length spec])	Partial statements: same **as partial** for normal aggregate but over length window with length [length spec]/producerCount Assembly statement: same **as** assembly for given aggregate

Table 5.1: EPL distribution patterns

Pattern 1 concerns statements that contain calculating an average of specific property of incoming events. This mechanism is portrayed in figure 5.4. Each event contains sum (*s*) and count (*c*) values calculated at partial producer and event producer specifier (*P*, each letter refers to different producer) for reference purposes. When event arrives to assembly monitor, it is put into an `unique` window (new events are marked with gray color), triggering computation of total average. If there was no `unique` window, an effect of "double aggregation" would occur. The assembly monitor would remember all received events and calculate average from all events sent so far. Because these event already contain aggregated "historical" values, the outcome would be invalid. For example, without the `unique` window, the fourth event (second from producer A) would be appended to the whole set. As a result, the average would be calculated from all four partial events, where fourth event already contains all the information from the first

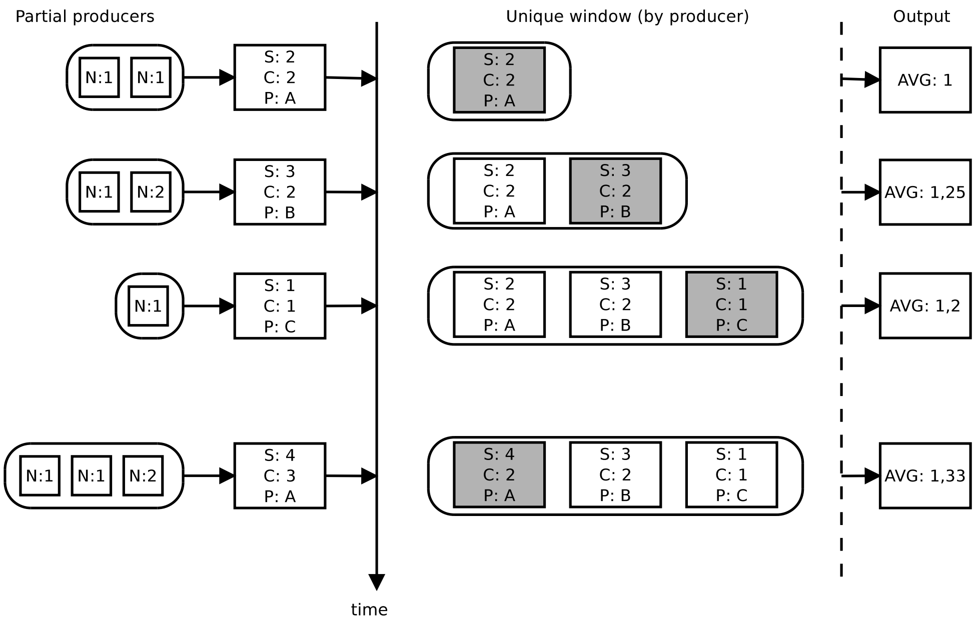


Figure 5.4: Example of distributed average calculation using unique window. Gray events are those that are updated by incoming ones. After each received event an actual result is produced. Each producer (on the left) aggregates its events on it's own and sends only aggregation results to monitor. The overall outcome is computed using those partially aggregated values.

event (from producer A). When using this kind of distribution, some synchronization problems that may occur. They are presented in 5.5.

In pattern 2 a `count` aggregation function is distributed. It is very simple: each intermediate monitor counts its events and sends the outcome to an assembly monitor. There the outcomes from each producer are summed to give final value. Again, the `unique` view is applied in the assembly statement to prevent overlaying old and new data.

Pattern 3 covers the `max` aggregate function (the `min` function is analogous). It is distributed by finding maximum of maximum values sent from other procucers/monitors. In other words, each monitor calculates its own greatest value and sends it to its consumer. The assembly statement uses `unique` window for the same reasons as in `avg` distribution.

The above three patterns show that each aggreate function has to be considered individually in terms of request distribution. However, the general principle is the same: partial results are aggregated and assemblied using the `unique` window.

A grouping operation is considered in pattern 4. This pattern is very architecture dependent. If the group by attribute value is different for each producer, then the statement can be naturally distributed over them. Simply, every monitor performs aggregation on events it produces. For example, if the original statement was:

```
select hostname, avg(userTime) as avgTime from CpuInfo group by hostname
```

where `hostname` attribute can be unambigously mapped to producer (one producer per host), then the partial statement would be:

```
select hostname, avg(userTime) as avgTime from CpuInfo group by hostname
```

In fact, the `group by` clause is virtually obsolete here due to the fact that all events coming from single producer would be in same group. However, it is required for the statements to be compilable. The attributes that are outside any aggregation function (`hostname` here) has to be in the `group by` clause, just as in SQL. The assembly statement is very simple:

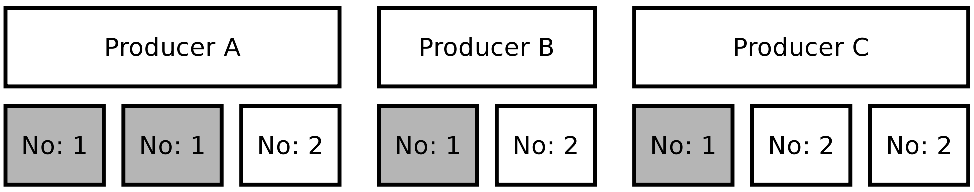


Figure 5.5: Example of an event group to producer layout for the group by clause. Each producer creates events that belong to one of two groups defined by the `No` property (`group by No`; a group corresponding to events with No value of 1 is marked with gray). As a result there are six virtual producers (2 groups for each one of three physical producers).

```
select hostname, avgTime from assemblyStatement;
```

However, when the `group by` clause concers attributes that are cannot be directly connected to individual producers, the distribution problem of such statement becomes complex. The aggregation cannot be delegated as a whole to other monitors because they operate on groups that are formed at the assembly monitor. Therefore, it must be distributed in the same way as in previous patterns. The assembly should include grouping the incoming events in the same way as the original statement would do it. However, when constructing assembly statement it must be taken into account that the groups created by the EPL statement are split over multiple physical producers. This problem is presented in figure 5.5. Events are grouped by the `No` property. Therefore, there are two logical EPL groups (group 1 is marked with gray color) divided between three producers. Each of them sees its part of group as a whole. That is, producer A considers two events as a whole group 1 and one event as a group 2. For each of these group aggregation is calculated separately. In the end, the "gray" results from producer A must be combined with "gray" results from producer B and C. In order to handle the aggregation properly, the assembly monitor must perceive each group in producer as separate producer. That is, considering the situation in figure 5.5 there are six "virtual" producers: white A, gray A, white B, gray B, white C, gray C. For example, for original statement as follows:

```
select number, max(userTime) as mxTime from CpuInfo group by number
```

where `number` is simply an integer identifying a CPU within single host (different CPUs on different machines may have the same number), the partial statements would be:

```
select number, max(userTime) as mx from CpuInfo group by number
```

The assembly statement is following:

```
select number, max(mx) as mxTime from
    CpuInfo.std:unique(producerSpec,number) group by number
```

The last statement, compared to plain `max` function distribution was modified with additional parameter to the `unique` window. This parameter corresponds to `group by` clause arguments and causes the `unique` window to distinguish groups within producers. Presented modification is valid only for `max` and `min` operations. Again, each aggregation function must be considered separately.

Statements in pattern 5 contains `output` clause. Distributing such requests has virtually no benefits. However, the `output` clause often appears with other EPL constructions therefore it is important to analyze it. In fact, distribution of the `output` clause is very simple. Both partial and assembly statements preserve the quantity specifier. However, the partial ones have the frequency reduced by some arbitrary factor `n`. Usually, is may be 2 or less, but to lesser than 1. It is needed because of possible lack of synchronization between those producers. This problem is discussed more thoroughly in 5.5.2.

Pattern 6 concerns the distribution of statements with aggregation functions over time window. Such statements can be distributed in similar way as plain aggregation statements. The only difference is additional time window in partial statements with same time interval as in original statement. The justification for such solution is simple. The plain aggregation statements can be viewed as having an infinite time window. This window is in fact applied implicitly in each partial statement for plain aggregation functions. Therefore, setting a limit on window in original statement influences windows in partial statements. As an example, following statement with `max` function on time window can be considered:

```
select max(userTime) as maxTime from CpuInfo.win:time(100 sec)
```

the partial statement is:

```
select max(userTime) as mx from CpuInfo.win:time(100 sec)
```

and assembly statement:

```
select max(mx) as maxTime from assemblyStream.std:unique(providerSpec);
```

Statements in pattern 7 use aggregation function computed over a length window. Similarly as in case of time window pattern, the assembly statement remains unchaged compared to assembly statements for particular aggregation function. However, the partial statements use length window with length equal to original size divided by number of producers taking part in distribution. The reason for this is that the original length window collects a limited number events from all producers. If length window is divided by number of produers, this limitation is preserved as each event received by the assembly monitor will contain information about $\frac{windowLength}{producerNumber}$ of all partial events provided by all producers. This value, multiplied by number of received events at given moment (which is equal to number of producers) gives the length of original window. For example, following original statement:

```
select avg(userTime) as avgTime from CpuInfo.win:length(50);
```

computes average user time from last 50 `CpuInfo` events. The assembly statement for this case is:

```
select sum(s)/sum(c) as avgTime from
   assemblyStream.std:unique(producerSpec);
```

The partial statement, assuming that there are 5 partial producers, is following:

```
select sum(avgTime) as s, count(avgTime) as c from CpuInfo.win:length(10);
```

The assembly statement will accept one event from each partial producer, which corresponds with 10 `CpuInfo` events. Because there are 5 producers, the final average will be calculated using 5 partial aggregation events (the ones containing `avg` and `count`) which corresponds to 50 `CpuInfo` events. One may argue that the `count` function is not necessary here as the actual number of events is known in advance because of the length window. However, is was kept to make this statement compatible with distribution of `avg` without length window.

5.3.3. Distributing stream joins

As mentioned in 5.1.1, EPL statements may contain join operation on event streams. Presence of this operation in EPL statement complicates the distribution process. In fact, it may cause the distribution virtually impossible.

Three following situations can be distinguished when dealing with statement containing stream joins:

- all events in all streams that are part of the join operation are provided by the same producer
- at least one of the streams is provided by other producer than the rest of the streams
- all event types involved in join operation can be provided by all producers

First two cases are easy to resolve. In the former one a statement can be distributed as if no join was present. The joined streams can be treated as one stream for the purpose of distribution. The latter case cannot be distributed. The statement requires events from two or more separate sources thus virtually no operations can be executed at any of the producers as there is no access to all required events.

When it comes to the third point the statement can be transformed only if the distribution of particular streams taking part in the join to the producers is regular. That is, if one of producers creates more events of given type than the other, distribution is probably not possible. For example, following statement produces events that contain average cpuTime for those processes, which were invoked by same user on two machines:

```
select avg(H1.cpuTime + H2.cpuTime) from ProcessInfo(host='192.168.2.31')
   as H1, ProcessInfo(host='192.168.2.35') as H2 where H1.user = H2.user
```

In this case, two producers can provide same event types (`ProcessInfo`). However, the first producer (at host with ip address 192.168.2.31) may have less processes invoked from given user (or none) than the other one. Moreover, the join itself filters events in a way that cannot be resolved at the producers separately because there is a correlation between streams coming from two different sources. Therefore, this statement cannot be distributed properly and has to be processed as is by receiving simple events from producers. Processing the `where` clause may be required here to determine whether correlation between streams exists.

On the other hand, if the there is no correlation as above and joined streams can be distributed regularly within producers, the statement distribution can be performed. For example, following statement:

```
select avg(P.cpuTime) as avgCpuTime, U.name as uname from ProcessInfo as P,
    UserInfo as U  where P.user = U.name group by U.name
```

can be distributed because the streams `ProcessInfo` and `UserInfo` do not correlate outside single producer (assuming that user names are unique for all producers, that is a given user name can appear on only one host). Thus, the assembly statement for this case would be following:

```
select sum(s)/sum(c) as avgCpuTime, uname from
   assemblyStream.std:unique(producerSpec);
```

and the partial statement:

```
select U.name, sum(P.cpuTime) as s, count(P.cpuTime) as c from ProcessInfo
   as P, UserInfo as U where P.user = U.name group by U.name;
```

Unfortunately, determining whether the correlation between joined streams exists is difficult. It requires knowledge of the meaning of properties that are involved in the `where` clause and finding out whether they concern events coming from different sources (such as the `host` property in the example above specifies the producer). This problem is discussed more thoroughly in 5.4. In practice, only statements containing join on plain event streams (without filtering) and without the `where` clause should be distributed.

5.3.4. Mixing patterns

Patterns presented in 5.3.2 cover only some very specific EPL statement types. In fact, the real, usable EPL statements rarely contain only one aggregation function or only `output` clause without aggregation functions. In most cases, these features are combined in one statement. In fact, finding a way of distributing every EPL statement can be a challenge. However, several most common combinations can be rather easily resolved. They are mentioned below:

- `output` clause and any other EPL feature combine rather flawlessly. This is because the `output` clause does not affect the processing and matching itself (conditions, filtering, time windows etc.) and can be simply attached to existing partial and assembly statements. Therefore, distributing statement with an `output` clause should first distribute the statement without it and then attach possibly modified `output` (see 5.1 to created assembly and partial statements

- aggregation functions with expressions as arguments, such as example average of sum of two property values. These are distributed as presented in table 5.1 with single argument replaced by expression. For example, following statement:

  ```
  select avg(userTime*idleTime*systemTime) from CpuInfo.win:time(200
     sec);
  ```

  can be distributed as normal `avg` statement with the single argument replaced by the multiply expression.

- multiple aggregation functions in one statements can be considered separately. That is, proper parts of a `select` clause should be constructed for each of them independently and then composed to form partial and assembly statements. For example, following statement:

  ```
  select avg(userTime) as avgUT, max(idleTime) as maxIT from
     CpuInfo.win:time(200 sec);
  ```

  would result in an assembly statement as:

  ```
  select sum(s)/sum(c) as avgUT, max(m) as maxIT from
     assemblyStream.std:unique(producerSpec);
  ```

  and following partial statements:

  ```
  select sum(userTime) as s, count(userTime) as c, max(idleTime) as m
     from CpuInfor.win:time(200 sec)
  ```

5.4. Handling partial producers

Many of request distribution patterns presented in table 5.1 referred to number of partial producers. This is where another problem emerges when it comes to request distribution: the assembly monitor has to know how many producers should take part in the process. More specificly, following questions must be answered:

- which producers should take part in request distribution. That is, which of them should receive partial requests.

- which assembly statement an incoming event sent by partial producer should be assigned to

The first point is very obvious and is an extenstion of the partial producers quantity problem. In order for assembly monitor to determine which producers should take part in distribution two pieces of information have to be gathered: what types of events is produced by each producer and what event types are involved in original statement.

The first one in easy to establish. Each monitor should hold a producer registry with producer addresses and provided simple event types. For base producer (not monitor, the one that actually only creating events) the produced event types are simply those that this producer can create in probing process. For any producer that receive events from other ones, the set of event types that it can provide must include also the events offered by sub-producers. For example, in figure 5.2 the monitor 2 would announce event types from producer A and B, while monitor 1 from monitor 2, monitor 3 and producer D (one has to keep in mind that each monitor treat base producers and other monitors that send streams to it uniformly).

The problem which event types are in involved in original statement is more complex. In order to answer this question, the original statement has to be examined. Beside the `from` clause (which explicitly contains streams that are involved) also `outer join` has to be taken into account. For example, a following statement can be considered:

```
select user.id, cpu.number, cpu.userTime from CpuInfo.win:time(200 sec) as
   cpu left outer join UserInfo as user on user.hostname = cpu.hostname;
```

It refers to two event types: `CpuInfo`, which is stated in the `from` clause and implicitly to UserInfo mentioned in the `outer join` clause. Therefore, the partial statements should be sent to all producers that provide either the `UserInfo` or `CpuInfo` events.

Moreover, besides the syntax of the statement, property values could be examined to determine which producers really should contribute to distribution. This is especially true when the `where` clause and window filtering are concerned. The condition in these clauses may effectively limit the events to only those coming from a few particular producers. Using resource identifiers as in statement below is a classic example of such situation:

```
select avg(userTime) from CpuInfo(resourceId =
   'host[192.168.2.34].cpu[1]').win:time(100 sec);
```

This is a typical case of average pattern (number 1 in table 5.1) except that it uses a condition in window specification to refer to only one specific resource: a first CPU on host with ip address 192.168.2.34.

It would the be best to send partial statements only to those producers that can provide events about this particular resource. However, in order to do that the assembly monitor would have to know two things: what resources each producer can provide events about and which properties determine the particular producer (it may be `hostname`, `deviceName` or as in previous example `resourceId`) and how their values are mapped to producers. The first requirement is easy to meet: the monitor should hold a resource registry (see 4.2.4) containing all resources handled by system and information which producers can provide information on them (thus, reference to mentioned before provider registry must exist). The second one is more complex. The monitor must know which properties can be considered as producer "discriminator". In other words, a mapping of pair (property name, property value) to producer is needed. Resource identifier described extensively in 4.3.5 is a good example of property that can be easily mapped to producer. Well defined format of this identifier that is used by all parts of monitoring system makes it possible to determine exact instance of resource by parsing the property content. Therefore, using such identification may greatly improve the statement distribution. When

Property name	Discriminating values
hostname	192.168.2.31
deviceName	/dev/sda5 /dev/sda6 /dev/sda1

Table 5.2: Example of discrimating producer event property values

it comes to other properties, such as `hostname` or `deviceName`, producer could publish them along with set of values that concern it. For example, a particular producer installed on host with IP adress 192.168.2.31 may send "distriminating" property values to a consumer as in table 5.4.

Of course, any discriminating properties received from one producer may "overlap" those from the other ones. For example, each producer may provide information about device named `/dev/sda5`. It is up to assembly monitor to determine whether given property uniquely points to single producer.

If window filtering exists and no information can be gathered on how this filtering maps to producers (meaning the used properties cannot be recognized by system in any meaningful way), the partial statement has to be sent to all available producers. This is not a great disadvantage as unwanted events (that do not match the condition) will be discarded sooner or later (in worst case at the assembly monitor). However, issuing requests to producers that will not effectively participate in gathering and processin data is a kind of inconvenience as it may lead to a unnecessary request flooding caused by monitors passing statements to all of their producers. This is especially true as far as very complex architectures are concerned, with developed producer-consumer tree.

The second of the problems mentioned in the beginning of this sections is connected with the fact that processing of single original statement spans across more than one CEP engine. In the centralized architecture the events created by any kind of producer at any location were simply put into one event cloud. As long as EPL statements did not refer explicitly to the origins of events (for example in the `where` clause to get only events from desired location), the CEP engine did not need any information aboout it. Therefore, event messages did not have to contain such data.

However, in case of distributed CEP the processing of single statement spans across mutiple CEP engines. That raises the need of correlation between individual monitors. This is visible, among others, in distribution patterns presented in 5.3.2. In some of them a property named `producerSpec` appeared. This is nothing more than a unique logical address of a producer that created given request. This address, or identifier, is assigned in arbitrary way by the monitoring system. There are only two requirements that it must meet:

- uniqueness: any producer (monitor is also a producer) must be able to be unambigously identified by any monitor that take part in statement distribution
- well-known: any monitor must know all identifier of its direct producers. Knowledge of addresses od remote producers is not required as their existence is hidden from non-direct consumers). The means by which producers announce their identifiers is is not significant. It is also possible that the assembly monitor assigns the addresses to the producers.

In order for the statement distribution to work properly, each event must be marked with this identifier of its creator (a correlation identifier, see *Correlation Identifier* integration pattern in [2]).

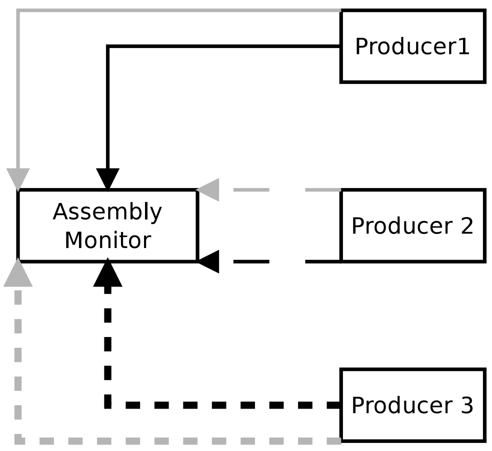


Figure 5.6: Example of event addressing by request and producer. Arrows symbolize event streams: gray ones concern distribution of original statement A, black - statement B. Streams coming from different producers are distinguished by arrow style (dotted, dashed and solid)

A different solution of the event provider recognition problem exists. Each of the producers could send events through different channel "physical channel" (such as JMS queue). In this case, the assembly monitor would know that events coming from given channel were sent by specific producer. However, this solution has a number of drawbacks, from among which the most serious is simply the multiplicity of required channels. Moreover, each of them would be utilized in relatively small degree by carrying events from only one source. Therefore, this solution is unfavourable.

Figure 5.6 illustrates a case where same assembly monitor cooperated with three different producers in distribution of two different requests (statements). Such situation can be very common as consumers may issue more than one request to the system. All three producers participate in both distributions. Therefore, the monitor receives two event streams from each producer. Both contain events with the same correlation identifier (described before). However, each stream concerns different request. Therefore, some kind of additional association between event streams and a distribution request that they concern is needed. This association must meet following requirements:

- producer identification independency: this results from the fact that multiple producers can take part in same request (gray streams in figure 5.6) and vice versa: every producer may participate in many request distribution (all producers in 5.6 contribute to resolving two requests)

- uniqueness and coherence: both producer and consumer must unambigously associate given stream with same distribution request

One of potential approaches to mentioned problem would be taking advantage of the way that Esper CEP implementation [4] (used in this research) handles streams. Namely, it is possible to create event stream based on event types by specfying that all events with given type should be put to same stream. Because most statements that are result of distribution do create very singular event types, it hints that these types could be used to associate requests with particular streams. Unfortunately, a situation where distribution of two different statements yields different partial statements that create same event types (streams) cannot be ruled out. For example, the two following statements result in same event format:

```
select avg(userTime) from CpuInfo.win:time(2 seconds);

select avg(userTime) from CpuInfo.win:length(100);
```

The difference is quite subtle: type and parameters of windows. However, the values in created statements would be very different and mixing them in one stream would result in unreliable monitoring data.

Therefore, an explicit request-event association is required. It can be established is similar way as the producer identification - by adding an identifier to each event. Every message should contain an ID (usually a number) that associates it with specific request. The identifier of given statement distribution should be sent to producer by the assembly monitor with partial statement. In this way every producer will be able to bind products of this partial statements with distribution request that this statement takes part in resolving. Again, it is theoretically possible to have separate event channel for each request. However, this is even more disadvantageous than in case of channel per producer approach as in this case channel would have to be transient since each request may be cancelled by consumer or expire (for example due to consumer failure). Moreover, using channels for both request resolving and producer identification would require vast number of them as each producer would have to maintain separate channel for each request it handles.

When it comes to the assembly monitor, it first separates events by their request id and inserts them to proper assembly statement. Then the statement itself takes advantage of producer ID contained in each event (see `producerSpec` in patterns in table 5.1). Therefore, the event-to-request resolving is transparent from the CEP engine's point of view.

Summarizing, two independent, complementary addressing means are needed: by event producer (correlation ID) and by request given event belongs to. Each event should carry both of these addresses.

5.5. Event synchronization

CEP approach is inherently time-based. Events can be ordered, grouped and released basing on their creation time. This can be done using widows and output controlling in EPL statements.

All these mechanisms work very well as far as centralized architecture is concerned. The event creation time is assigned the moment the event arrives to the CEP engine. Therefore all event messages are sychronized as they are sequentially ordered by the same clock.

However, when it comes to the distributed architecture of CEP event sychronization becomes a problem. In fact, most distributed applications have to cope with such difficulties. The reasons for the lack of synchronization between events can be various and include:

- unsychronized clocks - each machine in distributed environment may work with slightly different time set in its clock. This causes differences in possible timestamp values in events

- various load on different machines - the nodes of grid network that are more loaded may produce and process events slightly slower than the task-free ones

- network delays - different network routes between event source and event consumer have different length and Round Trip-Time, therefore even if two events are created simultaneously by two separate producers they may reach the destination in significantly different moments

The synchronization problem affects many aspects of distributed application. In this section an attention is focused on its connection with CEP. Several aspects are discussed.

5.5.1. Data accuracy

The fact that in distributed CEP events arrive to destination at different times than they would in centralized approach may often affect the accuracy of data they carry. By term **accuracy** the compatibility of the outcomes of centralized and distributed CEP architecture are understood. In fact, the centralized CEP architecture, as being very simple and close to the CEP idea itself (although very inefficient), can be treated as a kind of point of reference for testing the validity of distributed architecture implementation.

The violation of this accuracy can be especially seen when using time and length windows in EPL statements that are processed by distributed environment. Figure 5.7 illustrates the problem of time window. The events with dashed lines represent a situation where no delay between event creation and arriving to the CEP engine exist. That is, events are thrown into the engine right after the moment they are created. This somewhat reflects a situation where events are created and processed on same machine and no network delays occur. The solid-line events correspond to more realistic, delay-burdened architecture such as transporting events over the network or long intermediate processing (like in distributed CEP). In both cases same time window is shown (with same time-length). It is clearly visible that the difference in interval between consecutive events influences the behaviour of window aggregation. While without any external delays the window can hold up to three events, when the interval between the arrival two consecutive events increases, the capacity of the time window declines. In figure 5.7 the presented time window spans accross three events in the ideal case (no delays) and only two when there is some kind of hold off.

To clarify this problem even more, one may consider following case: a time window with span of 30 seconds (that is, it will hold events from last half minute) and events being produced every 5 seconds. This window can hold up to 7 of shose events (one current and 6 more from the past). However, if the interval between them increases by 10 milliseconds, the window will keep 6 events at most (the 7th will be too old by $30 - 6 \cdot 5,01 = 0,06$ seconds).

When it comes to the length windows, unlike in case if time windows, their sensitivity to lack of event synchronization is visible when more than one producer is sending events to an EPL statement that utilizes such windows. Figure 5.8 presents a situation where two producers are sending events which are put into length window afterwards. If there is no delay on event arrival the results of processing with length window are valid. That is, each event that is created later "pushes" the oldest one out of the window, Therefore, the window slides through the stream of events properly. However, if any of events is delayed, it may disturb the operation of length window. In figure 5.8 events provided by Producer A arrive with certain delay. As a result, the time order between events at receiving point that was originally kept in the length window is violated. Event 4 and event 2 are kept together in window even though event 3 occured between them.

Such event mixing in may render results of some EPL statements unreliable. For example, a following EPL statement can be considered:

```
select avg(userTime) from CpuInfo.win:length(20);
```

Assuming that there are only two producers installed on two different machines, each creating one event message every second, this statement will result in average user time of two CPUs from last 10 seconds (2 events per second · 10 seconds = 20). However, if events from one producer are late, the events coming from the other producer will start to dominate the

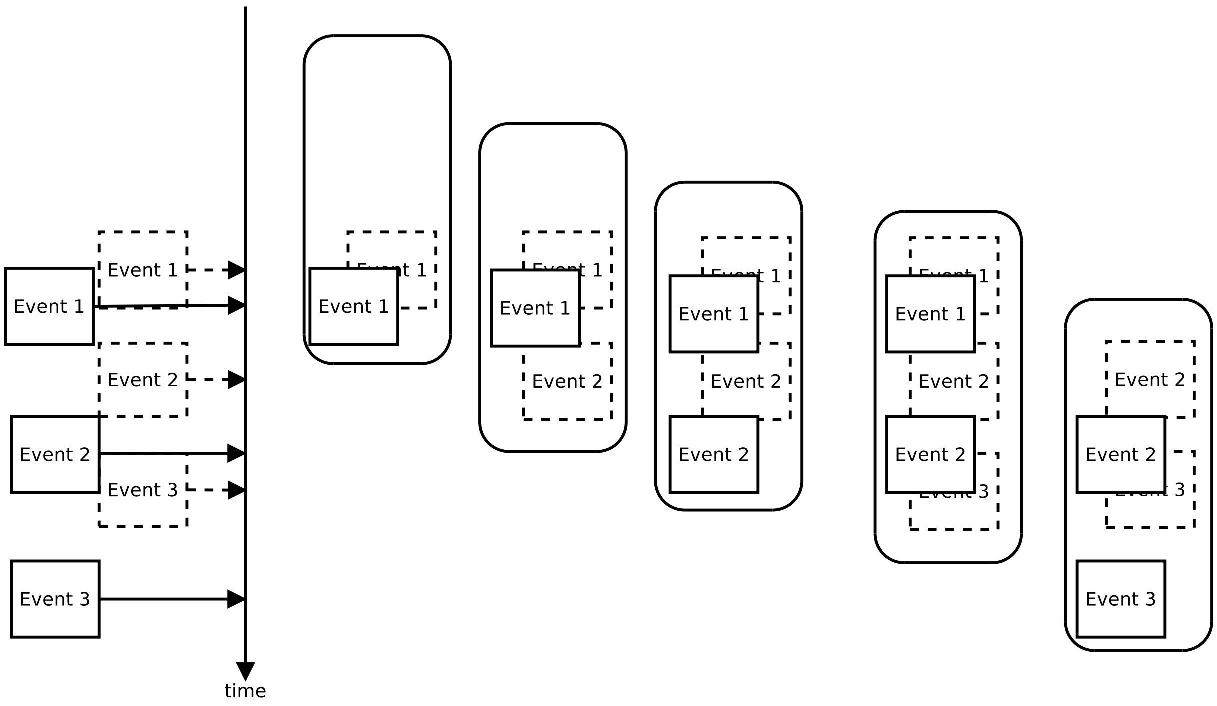


Figure 5.7: Lack of synchronization affecting accuracy of a time window based processing. Boxes with dashed lines represent events that arrived without delay. Solid-line boxes portray events that arrive with slight delay.

length window. As a result, the average will be counted using the data from the other producer in greater part, becoming a somewhat weighted average. Unfortunately, this will happen without the consumer knowledge. As a result, the actual data will be interpreted incorrectly.

5.5.2. Output time control

Output control based on time intervals described in 5.1.4 can also be vulnerable to problems with event synchronization, especially the delays. If EPL statements contains the `output` clause stating that a group of events should be released every speified time interval and any of partial events that should take part in processing arrive after this interval because of delay, the result event may contain less accurate data.

For example, the following statement releases last of result events every 5 seconds:

```
select avg(userTime) from CpuInfo.win:time(30 minutes) output last every 5
    seconds;
```

Assuming that the `CpuInfo` event is produced every 1 second, if there is no delay each result event should be created after 5 new partial events have arrived and put into the time wimindow. However, if the last of these 5 events arrives event just a little bit after 5 seconds, it will not be counted in current processing.

All of mentioned problems concern on-line monitoring data processing. That is, data is gathered and sent to consumer contantly, with small intervals. In fact, the smaller the interval, the greater the significance of the delay as it is more probable that certain event will be created soon enough to arrive before its delayed predecessor from other producer. If events are sent rarely, with interval specified in minutes or hours, it is virtually not possible that a delay would cause their reordering at receiving end.

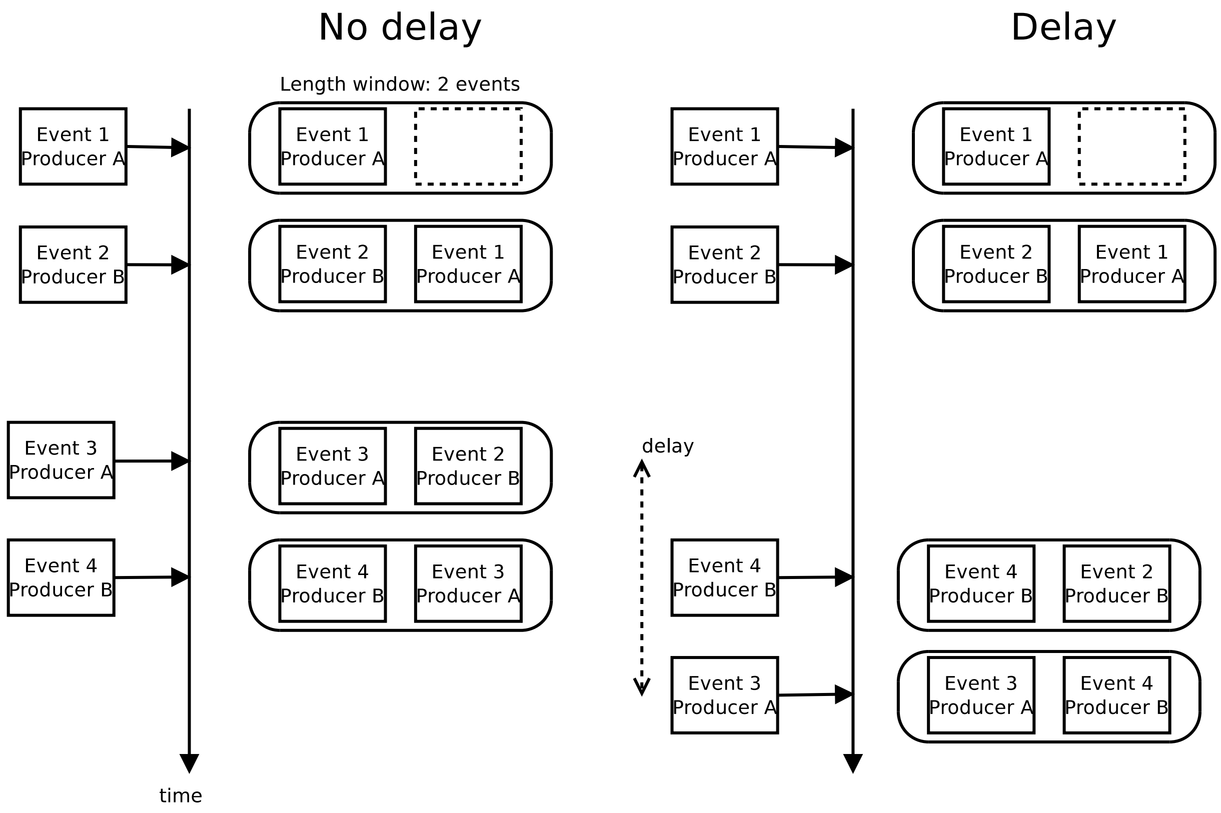


Figure 5.8: Lack of synchronization affecting length window based processing accuracy. Left part refers to situation without delay, right - with delay on events from producer A. A delay in Event 3 arrival causes the order of events in length window to be violated, as Event 4 and 2 are simultaneously placed in the same window (compare with left side of this figure).

5.5.3. Error accumulation

One may argue that discussed synchronization problems in time and length windows also occur in centralized architecture of CEP and that’s true. In fact, most applications of such system would be in distributed environment, where event messages are sent over the network and can be delayed. However, these factors become more important in distributed CEP architecture as there are many steps of event processing and therefore many points where data incosistency may occur. Moreover, these incosistencies may accumulate at each successive monitor. This accumulation occurs in several cases. First of all partial processing at intermediate monitors takes more time that simply passing an event to next node to reach the central monitor. This causes additional delay in event transporting and causes events to reach the consumer much later than in case of centralized CEP. Secondly, the output time control problem can become very serious in distributed environment. Figure 5.10 helps to illustrate this. Monitor 1 and 2 use an EPL statement using same time output control with interval 5 seconds. Each producer creates events every 1 second continously. Events in stream 3 are late by 50 milliseconds due to network delay and slightly longer processing of monitor 2 (later initialized CEP engine causes all processing to be shifted in time). If both monitors start counting down the output timers simultaneously (which is quite possible to some extent, as both statement may be part of same statement distribution process), monitor 2 will output the event at the same moment monitor 1 closes the output batch. Therefore, the event sent by monitor 2 will not reach monitor 1 on time to be included in current processing. As a result, the monitor 1 in following batch will use old event (the one that have arrived just after closing previous batch). This is more serious because monitor 1 will in fact use outdated data (by 5 seconds) from both producer 1 and 2 (that was aggregated by monitor 2), rendering the whole outcome unreliable and probably useless for consumer. This effect is shown in figure 5.9. Such chain of outdates may span across multiple monitors, causing vast amount of resulting events to contain unreliable data. One of possible solutions to this problem is to use smaller output intervals the closer monitor is to original producer. Referring to above example, monitor 2 could use time interval of 3 or 2 seconds. This would cause at least one of its events to arrive in time and reduce the amount of data in aggregated event that is outdated. This solution is used in distribution patterns presented in table 5.1.

5.5.4. Evaluation

Many problems presented before that are connected with distributed CEP are very hard or even impossible to overcome. In fact, it is virtually impossible to get rid of delays caused by network or event processing on monitor. Synchronizing clocks or CEP engine initialization on particular nodes of distributed system to make them work simultaneusly is too complicated compared to potential benefits (clocks are usually inconsistent by 1 second at most).

However, all of these problems are significant only in some specific cases. First of all, the interval between two consecutive events created by producers matters. In most cases it is specified in seconds or minutes, which is much greater than potential delay caused by network and processing at monitors together. The latter is usually hundreds of milliseconds at most. Secondly, a even if a single case of data inaccuracy, as described in 5.5.1 occurs, its significance is the smaller the smaller the interval between production of simple events. This is because the more frequent simple events are the smaller their validity time as they are considered outdated after new event is created. As a result, the possible lack of synchronization and data unreliability caused by it have little consequence because in short time following event will correct the

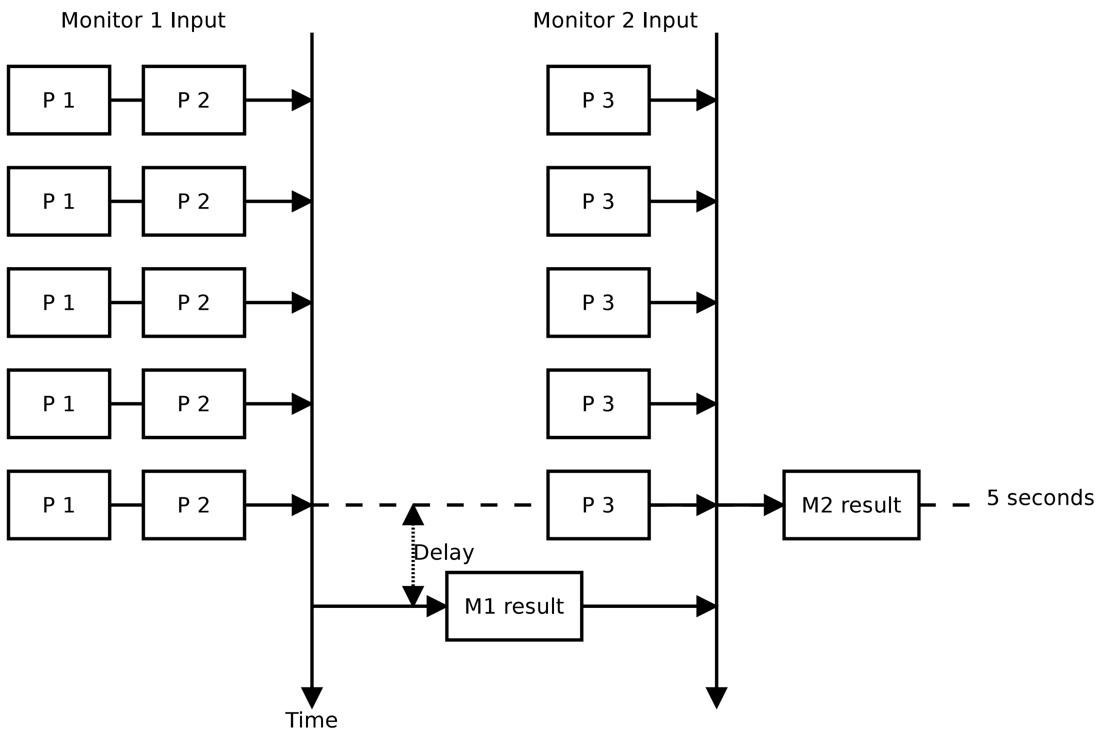


Figure 5.9: Effect of outdated data when using output time control. Monitor 1 and 2 use statements with output clause with same time interval (5 seconds). Event produced by monitor 2 that is result of this statement arrives to monitor 1 with some minor delay and is used only in next batch. Events P1, P2 and P3 are created by producers 1, 2 and 3 respectively.

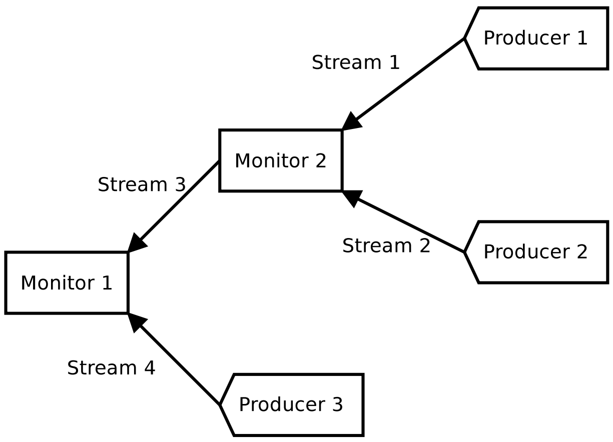


Figure 5.10: Example architecture to illustrate the problem of late event error escalation when using controlled output. Each monitor introduces a delay caused by the event processing and data serialization/deserialization operations. As a result, events created by Producer 2 will arrive later to Monitor 1 that those created by Producer 3 even if total delay of network links used by stream 3 and 2 is same as those utilized by stream 4.

error. Therefore, the smaller the event creation interval the smaller the effect of the errors and the greater the interval the smaller the probability that the lack of synchronization will occur.

One of the factors that may have a big impact on the distributed CEP reliability is mentioned in 5.5.2 accumulation of error. In fact, if the distributed CEP based monitoring infrastructure is complex and developed, containing many monitors, error accumulation is a real threat to data accuracy. Therefore, the actual placement of the monitors should be thoroughly considered to minimize the length of the path from consumer to first producers (counted in monitors).

It also should be kept in mind that in very few cases the consumer would want to receive very current data. Most consumers will be interested either in some kind of averages over the specified past time to know the overall health of a system or momentary peaks in value of specified properties (such as network utilization) to detect or prevent system failures. On the other hand, if consumer really requests for on-line, actual data this is usually only for informative purposes, for example to display current CPU user time. Therefore, minor inconsistencies in data should not seriously impact on the usability of on-line monitoring infrastructure based on distributed CEP.

5.6. Benefits

All the problems presented above straightforwardly state that distributed CEP comes at price. In fact, of the are very hard or impossible to solve. Therefore, it is important to analyze potential benefits of this solution.

The main purpose of applying distributed CEP to monitoring system is reducing the bandwidth utilization. This can be achieved by limiting the number of events transmitted over the network. This in turn can be done by carrying out the event processing, such as aggregation and output control, as near the event source as possible.

5.6.1. Beneficial EPL statements

In section 5.3.2 it was said that not every EPL statement can be distributed. Indeed, some statements are so simple that there are no points where distribution could take place. However, the sheer fact that statement can be distributed does not mean that the distribution would be beneficial in any way. For example, following statements can be easily transformed according to patterns presented in 5.1:

```
select avg(userTime) from CpuInfo.win:time(100 sec);
```

This statements calculates an average which can be distributed over multiple producers. However, according to Esper implementation, each time a `CpuInfo` event arrives, the average will be recalculated effecting releasing an new event containing actual value of the average. In other words, for each incoming event there is being released one. The same thing applies to all other aggregation functions: they are calculated again when new event arrives to the statement. Therefore, the assembly and partial statements for such functions will behave in same way: releasing an event after new one arrives. The overall conclusion is that the number of events resulting from distributed statements is equal to the amount of messages sent when using traditional, centralized approach. Therefore no data reduction takes place and distributed CEP loses its advantage.

However, such statements as presented above have little usefulness in real-life monitoring systems. Knowledge of the very current value of the average cpu user time over the past 100 seconds is rarely critical for estimating the health of the system or preventing errors. Especially as it may be burdened with errors and inaccuracies (see 5.5.

A more real-life EPL statement example that could be useful in monitoring infrastructure is presented below:

```
select avg(userTime) from CpuInfo.win:time(100 sec) output last every 50
    seconds;
```

This statement is very similar to previous one. There is only one difference: it uses the `output` clause. This change causes the events to be released every 50 seconds. As a result, the consumer would know the average of CPU user time from last 100 seconds with 50 seconds interval. Although the very actual data is not available, this statement is still useful for estimating current load of CPUs. Moreover, it is very beneficial when it comes to distribution. Assuming that `CpuInfo` events are created every 1 second (which is very frequent for monitoring purposes), this statement outputs one event for every 50 arriving. Therefore, the reduction of data is significant here. If centralized CEP approach was applied, this reduction would concern only the channel from the front monitor to the consumer. All the events created each 1 second at every producer have to be sent by them to central CEP engine, literally flooding the network. However, if statement distribution is used according to table 5.1 each producer would output events at specific frequency (distribution pattern for the `output` clause uses decreasing frequency for each consecutive distribution). As a result, the amount of events transmitted between monitors taking part in processing a statements can be reduced significantly. The exact reduction factor is implementation dependent. For definiteness, architecture in figure 5.2 may be considered. All producers (A, B, C and D) can create `CpuInfo` events with interval of 1 second. Assuming output frequency divisor n is equal 2 (see 5.1), if one of the consumers issued a request containing above EPL statement, the monitor 2, 3 and producer D would receive requests to output events at rate 1 per 25 seconds (50 seconds divided by 2). Then, monitor 2 and 3 would issue another requests to producer A, B and C with output interval od 12 seconds (25 seconds divided by 2 rounded down). Therefore, events between monitor 1 and its direct producers flow at rate $\frac{1}{25s}$ and between monitor 2 and 3 and their partial producers at rate $\frac{1}{12s}$. This is a significant improvement compared to the centralized CEP where events between the producers and the central monitor would be sent at rate of $\frac{1}{s}$. Such speed is not very significant for network links. However, if there were more requests issued by consumers, the number of event created by each producer could be much greater. The request distribution makes it possible to reduce traffic in this case at least 12 times.

Presented example shows that statements using the `output` clause can be very beneficial when it comes to data reduction. In fact, every EPL statement that contains timed (*output every time interval*) or quantified (*output every amount of events*) has great chances to be distributed in an effective way.

However, one must take into account the quantity specifier in the `output` clause telling the CEP engine how many events should be released when the time or quantity condition is met. For example, the `output all` clause causes all gathered events since last output to be released. This means that although the `output` clause is present, there is no real reduction as each accumulated event triggered by incoming message will be sent eventually. The only difference in comparison to a case without output control is that here the data is released in bursts causing the network links to be utilized in greater extent periodically.

Therefore the general rule that tells whether a given statement beneficial for request distribution or not is:

$$\frac{N_{received}}{N_{created}} > 1$$

where $N_{received}$ and $N_{created}$ are number of events received and created respectively by the statement during some arbitrary time. In short, statement should produce less events than it receives.

Statement distribution can reduce the number of events not only by moving the aggregation closer to the source. It can also help to detect producers that are obsolete in resolving particular statement and discard unwanted events sooner. If the CEP engines where installed not only in intermediate monitors but also in the mere sources of event, they could fitlered out many messages right away. A request for statistics of one particular CPU existing in system is a classic example. If centralized CEP was used, all the nodes existing in the environment would have to send their messages containing information about CPU. The data would be filtered only in central monitor. This would result in a lot of unwanted data being sent over the network and then discarded at the CEP engine. However, if part of event analyzing and filtering was done at the source, the CEP engines installed at those nodes that are not of interest of the consumer would be rejected right away and never sent. Also, each intermediate monitor may discard some of the events, gradually reducing the number of data being transmitted. Of course, the amount of data being reduced and the question whether it is possible or profitable depends on the specific request and usually cannot be determined by the system. Below are some examples of EPL statements distribution of which will result in some data reduction.

```
select avg(userTime) from CpuInfo.win:time(100 sec)  where hostName =
   'gandalf' output last every 5 seconds;
```

Obviously, this statement concerns only messages send from host named *gandalf*. No other events are needed to resolve it and those produced by sensors installed on other nodes will be discarded. If such statement was sent to other host than the one specified in the `where` clause, it will not create any events.

A more complex example may include a network presented in figure 5.11 composed of two clusters, each holding a group of disk arrays. An example EPL statement that could be used by the consumer to examine the health of such system is presented below:

```
select ControllerInfo.name, max(usedSpace) from DiskInfo.win:time(1 hour),
   ControllerInfo where DiskInfo.hostName = ControllerInfo.name and
   ControllerInfo.clusterId='DIANA' group by ControllerInfo.name
        ControllerInfo.name output last every 5 seconds.
```

This statement will result in events containing maximum used space from all disk arrays belonging to single disk controller (the `group by` clause) from the DIANA cluster. The maximum will be calculated from events that were created in last 1 hour. The up-to-date value will be released every 5 seconds. Each disk controller will send its events to the corresponding agent. Both controllers in DIANA cluster will be requested by their agent to provide the `ControllerInfo` and `ProviderInfo`. As a result, the controllers will request the events from the disk arrays (assuming that proper sensors are installed) The agent itself will process them according to the statement and sent result to the Front Monitor.

Controllers in the SARAH cluster will also be requested for events, despite the fact that they are useless for this statement. This is because of the way `join` constructions should be resolved(see 5.3.3). However, the CEP installed SARAH cluster controllers will discard the events as soon as they are associated with the `ControllerInfo` events and identified as coming from the inadequate cluster. As a result, no useless events will be transmitted outside the controller domain (controller and disk arrays handed by it).

This kind of data reduction (by discarding unwanted events) is a natural consequence of using distributed CEP. No additional operations or processing are needed. The native features

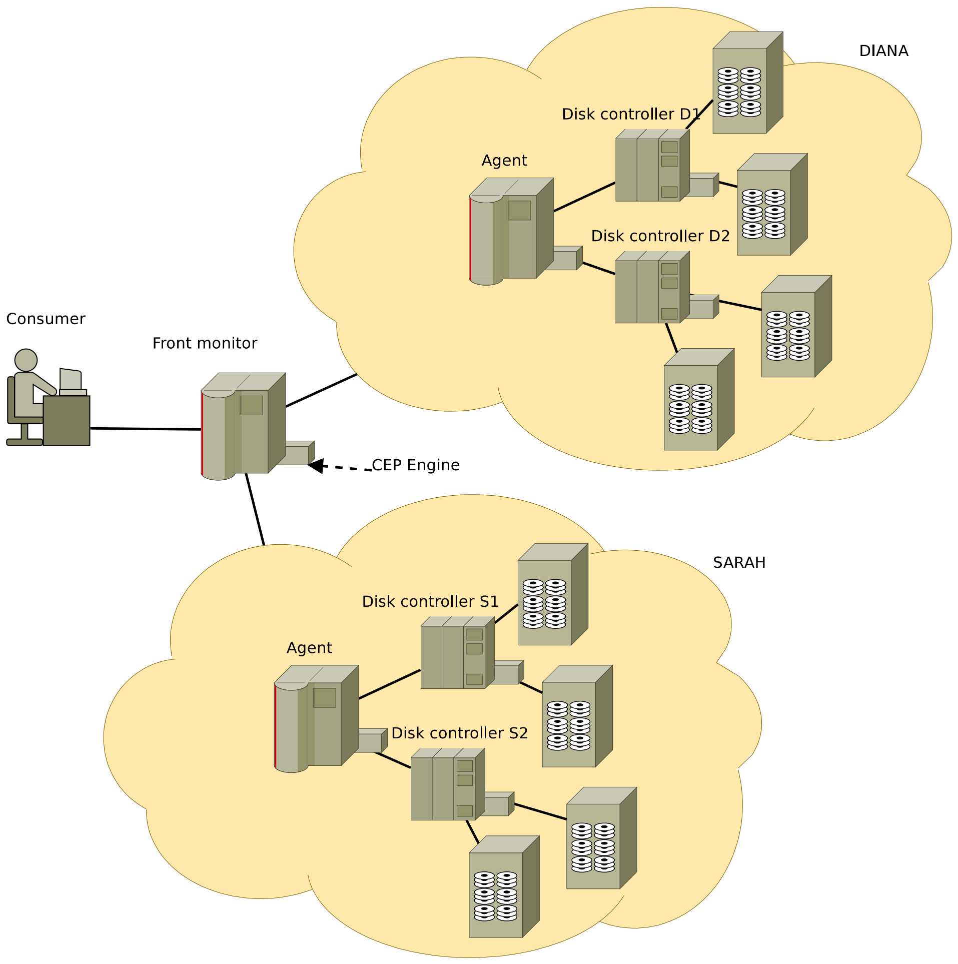


Figure 5.11: Example distributed system architecture being subjected to monitoring. The two clusters, SARAH and DIANA communicate with the front-end monitor through their agents. Each Disk controller, the agents and and the front monitor have CEP engines installed (boxes in right lower corner). The consumer may send request to and receive events from the Front monitor.

of CEP such as event pattern matching and filtering are used here and do all the job. The key is to take advantage of them as close to the event source as possible.

5.6.2. Balanced memory usage

Except for the data reduction mentioned before, using distributed CEP may also bring a number of some minor benefits. One of them is balancing the memory usage. This feature results from the fact that typically each monitor does a part of aggregation, filtering large groups of events etc. All the aggregation functions presented in 5.1.3 require all the data involved in calculation be held in memory. Because there is no one large groups of events held in memory of one node but rather a couple of smaller groups, each stored on different host, memory usage caused by aggregating events is distributed over multiple monitors. As far as centralized CEP is concerned, the node that the CEP engine works on could be loaded with vast amount of events to be kept in it's memory. This can be clearly seen in case of EPL statements that use timed output control with very large interval (hours or days). In such situations, all the events have to be stored until the specified time passes and the event batch can be released. For example, a following EPL statement can be considered:

```
select * from DiskInfo.win:time(3 days) output all every 2 days;
```

In the this case, all events have to be kept in memory as all of them are required to be released after 2 days. If all of events required for resolving this statement were to be stored in single node (centralized CEP), significat amount of memory would have to be used (counted in megabytes, see 7.1.1). On the other hand, in case of distributed CEP, each monitor would store a part of these events and release them after some relatively long period of time (2 days divided by some factor resulting from timed output control distribution).

In addition to aforesaid factors, some CEP implementations, such as Esper, optimize resolving statements so that only absolutely necessary events are kept in memory.

6. GEMINI-2 Development

This chapter presents an implementation sensor module for GEMINI-2 [17] and modification that have been introduced to monitor module. It also describes the way that sensor for GEMINI-2 interacts with monitor and other parts of the system and how its architecture and some applied solutions influence the overall usability and support distributed CEP approach.

First some background is given, presenting information regarding GEMINI-2 framework that is essential to understand further parts. After that a description of changes introduced to monitor component of the system under consideration is given. It is followed by information about architecture and implementation details of sensor component that was created during research described in this paper. Finally, some .

6.1. Background information

This section is dedicated to overall GEMINI-2 architecture, solutions introduced in it and some missing features. This information is needed to understand further parts that concern research on and development of GEMINI-2.

6.1.1. GEMINI-2 overall architecture

High-level GEMINI-2 architecture is presented in figure 6.1. According to this architecture, sensors communicates with monitor in order to subscribe in and send events to it. Monitor is supposed to send requests for events to the sensors. The interaction between monitor and sensor is discussed in details in 6.6. The sensor component presented in this diagram is in fact a simple stub. It generates events containing data on first discovered CPU on machine that it is installed on. There is no CEP engine implemented in it, therefore the CEP approach is purely central (the only CEP engine is installed in monitor).

Moreover, not all of presented monitor parts were finished. In particular, the sensor manager, discovery services and all parts regarding mutators were still under development when works described in this Thesis began. Some of them have been implemented during this research to enable evaluation and feasibility study. The details are described in 6.3.

All presented components are implemented using Java programming language with minor support of other technologies and native code.

6.1.2. Endpoints

To define and configure communication channels between distributed elements GEMINI-2 uses a concept of endpoint. An endpoint is a broker between actual transport system (JMS, WebSerivces, plain TCP) of messages with given component (client, monitor, sensor). Each endpoint represents either consumer or a producer. As the name suggests, producer can send

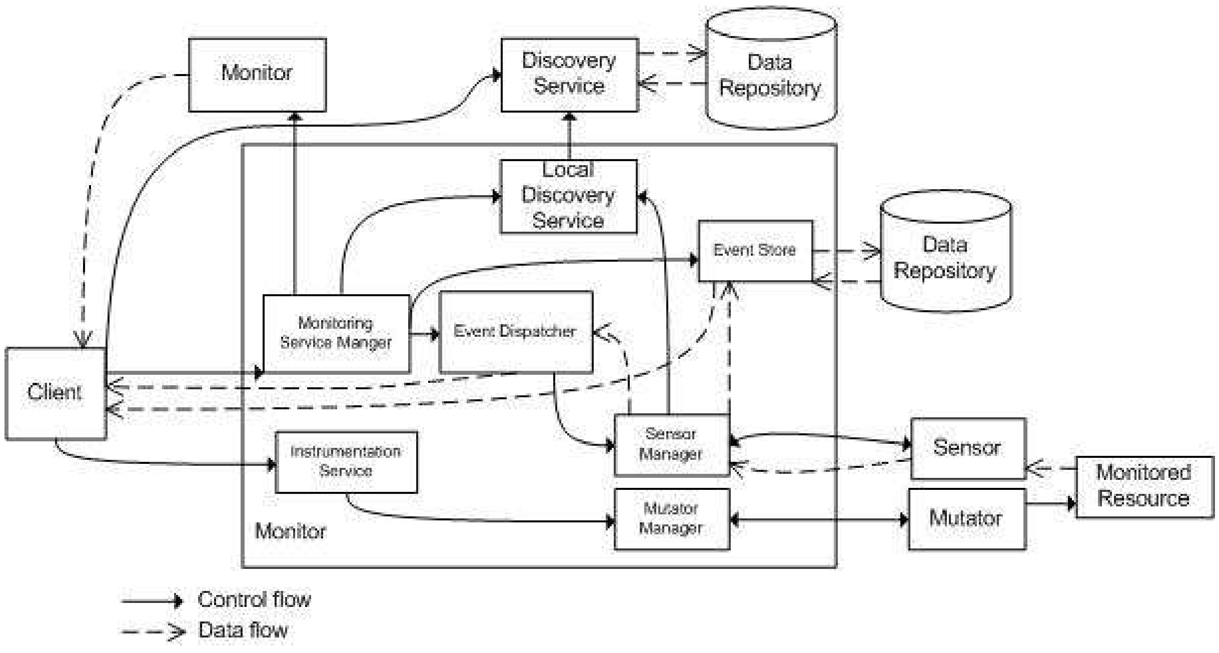


Figure 6.1: GEMINI2 architecture. Presented diagram is taken from GEMINI-2 documentation site [5]. It contains the target architecture. Some of features are not implemented yet.

messages through the channel while consumer can read them. This separation of roles of endpoints implies that channels defined by them are unidirectional. Every endpoint has a set of messages assigned that it can send to or receive from channel.

Two endpoint types were supported when works described in this Thesis began:

- event endpoint - used to send or receive events containing monitoring data.
- monitoring service manager endpoint - used to send requests by client and receive them on monitor side. Channel associated with this endpoint carries request and control messages.

More information about endpoints is available in [5].

6.1.3. The Esper CEP

The GEMINI-2 infrastructure uses the Esper CEP implementation for event processing and matching. It is an open-source (GNU GPL) component written entirely in Java. Below are some key features and concepts of this software that are significant from the point of view of works described in this Thesis. Full description and documentation of this product can be found on its webpage (`http://esper.codehaus.org/`).

Event types

Esper uses a number of ways to define event types. Most important ones are:

- plain Java class - each Java class can be treated as definition of separate event type. As a result, any Java object can be sent as event and processed by the CEP engine
- map - uses a mapping of property names to data types to define event type. Each event will contain a set of properties with given names with values of types defined in this mapping. In other words, event type contains set of property names and their types

Field	Value
Name	"Cpu"
Property names	user time
	system time
	idle time
	number
	core count
	max frequency
Discriminator	number
Static properties	max frequency
	number
	core count

(a) CPU metadata

Field	Value
Type name	"CPU"
Property values	0.7
	0.2
	0.1
	1
	2
	2000

(b) CPU instance

Table 6.1: CPU Resource representation

A crucial thing about event types is that the Esper CEP has to know all types involved in given EPL statement before this statement is compiled and used by the engine. Moreover, new event types can be added at runtime. The implications of these facts will be discussed in following sections of this work.

Creation of event streams

A stream of events in Esper can be created in one of two ways:

- explicitly, by inserting events from other stream using the `insert to` clause
- implicitly, by simply sending events to the engine. This will result in stream of events named after the type of sent events.

The above facts imply that the initial (first) streams are created only by sending event to the CEP engine. Moreover, every event stream can contain only events of the same type that is known in advance by the engine.

6.2. Resource representation

In chapter 4 a general idea of resource handling in monitoring infrastructure was discussed. Here a concrete implementation of resource representation in GEMINI-2 is presented.

Each resource is characterized by two kinds of data: its metadata (corresponds to resource type) and concrete instance data (or just instance). The resource metadata specifies the set of properties, information about which of them are static (see 4.2.4) and the discriminator (see 4.3.5). A given instance of resource carries only the type name and the property values. For example, a particular CPU is represented by structures as in table 6.2.

Cpu instance refers to its type by *type name*. Such data distribution into two structures is useful because of couple of reasons:

- it reduces the amount of data that has to be kept in memory or in database. In fact, for each resource instance only property values and type name are stored. The common data for resources of same type is held once.

- makes it easier to publish new resources types discovered by sensor module. Only resource metadata has to be sent to monitors.
- it corresponds very well to relational data model, therefore it makes it easier to build relational database based resource registries that would hold all resources discovered in system. The reference through the type name can be seamlessly converted to foreign key in relational data model.
- size of events that concern resources is reduced as only type name and property values are contained in them.

Still, some drawbacks exist:

- all elements of monitoring infrastructure must hold coherent sets of resource metedata. If any of monitors lacks some resource metadata, it will not be able to process any data (such as events) that refers to the unknown type
- the order of property names and values must be the same in metadata and instance data

The second problem can be easily overcome by proper implementation. The first one is solved by keeping all data about resources and their metadata in one place called **resource registry**. This registry is referenced by all monitors to update the resources or to get information about them.

In current implementation the resource registry is kept in memory using plain Java classes. However, thanks to proper interface abstraction the design is adapted to using more sophisticated implementations, such as Hibernate based persistence.

6.3. Monitor modifications

In order to make the cooperation between monitor and sensor as presented in 6.6 possible, a number of modifications had to be introduced into the monitor module.

First of all, the original monitor implementation lacked a sensor manager module. As a result, neither the subscription of sensors in the monitor nor sending requests to sensors was possible. The sensor could only send events to the predefined monitor. In order to solve this problem, a minimal functionality of sensor manager was added to the monitor. This includes:

- ability to receive subscriptions from sensors. Every sensor should be able to introduce itself in designated monitor with proper initialization data
- sending requests to sensors. Monitor should be able to control what and how events are produced by the sensor. In particular, it should be able to send an EPL statements that sensor should use for event generating.
- a sensor registry containing information about each sensor that has introduced itself to the monitor. This is needed to properly handle request distribution (see 5.4).

In order for the first feature to operate, an additional endpoint type was required that would handle the messages with initialization data that are sent from sensor to monitor. Therefore a `EventProviderControllerEndpoint` was added. The producer of this endpoint is sensor's `MonitorAgent` (see 6.7). On the consumer side is the

`EventProviderControllerListener` that receives the subscription messages from sensors and takes care of their proper initialization.

The sensor requesting mechanism was implemented in the same way as the client requesting to monitor. That is, from the sensor point of view monitor is a client just as a real client to the monitor. Such solution has two advantages. Firstly, it reuses already created and tested solution. Secondly, it is a step towards the unification of communication interfaces between event providers and event consumers. Thanks to that it is even possible that client would issue a request directly to the sensor. Such situation could be possible if monitor returned a client's address for request channel in subscription response to sensor (see 6.6). For details about the client-side mechanisms regarding event request handling reader should refer to the GEMINI-2 development site [5].

In order for a monitor to keep information about subscribed sensors, an Event Provider Registry stub was added. It holds an identifier that has been assigned to sensor (see 6.6 associated with following data:

- a transport address that this sensor listens on for requests for events
- a client stub that has been assigned to given sensor for requesting events
- list of event types that this sensor can provide.

All of this data is needed for request distribution and proper sensor management. The fact that considered element is called an Event Provider Registry indicates that it is adapted to hold information about any kind of event provider, not just sensors. Again, this is a small step towards the unification of comminication interfaces between GEMINI-2 infrastructure components.

As aforesaid, current Event Provider Registry implementation is just a stub that holds all information in Java Virtual Machine memory. It has some obvious drawbacks, including the vulnerability to virtual machine crashes and data locality (accessible to only single monitor). However, it can be relatively easily replaced with more sophisticated solutions, for example some based on JNDI to make it available globally and persistence technologies such as Hibernate to prevent it from data loss caused by JVM crashes.

Another element that was introduced to GEMINI-2 monitor is the **resource registry**. It contains information about resources discovered by sensors and received with resource update messages (see 6.6.3). The resource registry is composed of two elements: resource representation provider and resource context. The first one is responsible for holding resource types, as described in 6.2. Each entry is identified by resource type name. The resource context contains specific information about each resource instance: its identifier, static property values and optionally parent resource identifier. Using the last one, resource context maintains a parent-child relationship between resources, making it possible to fetch data in more sophisticated way that just by resourceId, such as retrieving all CPUs installed on given host. Just like event provider registry, current resource registry is only a stub that will be replaced with real implementation in the future.

Finally, a request distributor component was introduced to monitor. Its main role is to construct assembly and partial statements for distributed CEP. It is discussed thoroughly in 6.9.2.

The monitor architecture, after all introduced modifications is presented in figure 6.2.

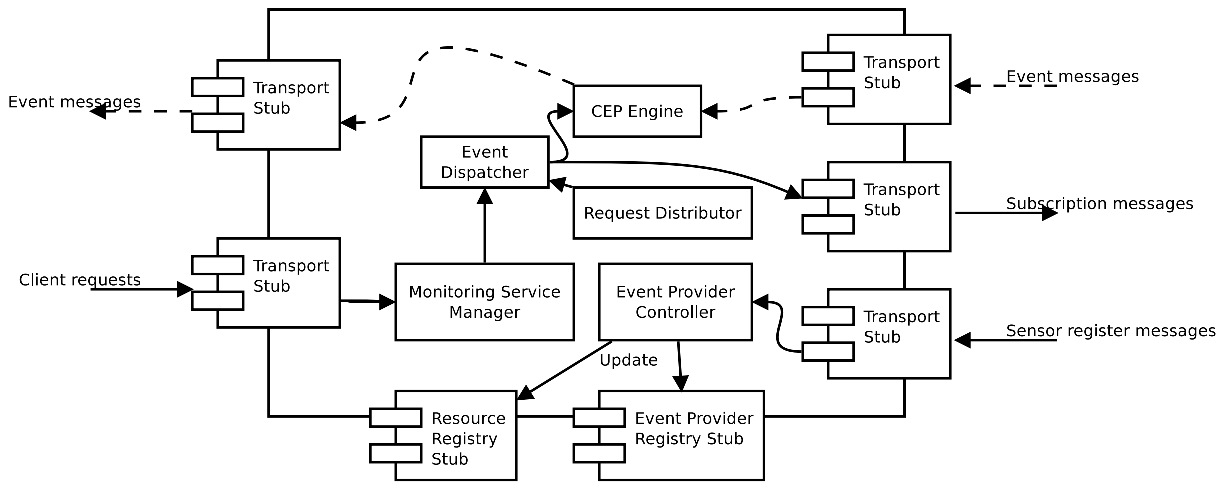


Figure 6.2: Diagram presenting monitor architecture in current state. Dashed arrows represent flow of monitoring data events while solid ones represent contol messages and signals.

6.4. Sensor environment

Figure 6.3 presents a environment in which GEMINI-2 Sensor component works. In fact, it is a part of GEMINI-2 high-level architecture widh some additional details and without elements that do not interact with sensors directly, such as event store.

Each sensor is deployed on separate node of measured distributed system and it is supposed to handle all resources installed on this node. Therefore, there is only one sensor per node. There is also a possibility that sensor was installed on the same node as monitor. Resources are handled by sensor using Sampling Modules (see 6.8).

Each sensor is assigned to exactly one monitor. On the other hand, monitor can manage multiple sensors, usually located in same subnetwork or some kind of domain. The managing itself boils down to sending requests for events and canceling them (see 6.6).

6.5. Sensor features

The sensor module design and implementation is supposed to meet following requirements:

1. modularity - sensor should be modular in order to be able to handle multiple types of resources

2. automatic discovery of existing resources - sensor installed on given machine should discover existing resources and publish information about them to corresponding monitor

3. installation of new resource types at runtime - there should be no restart required to introduce new resource type

4. ability to receive EPL statements from monitor and use Esper CEP engine for data reduction

5. the overhead inflicted by working sensor on the machine it is installed on should be as small as possible

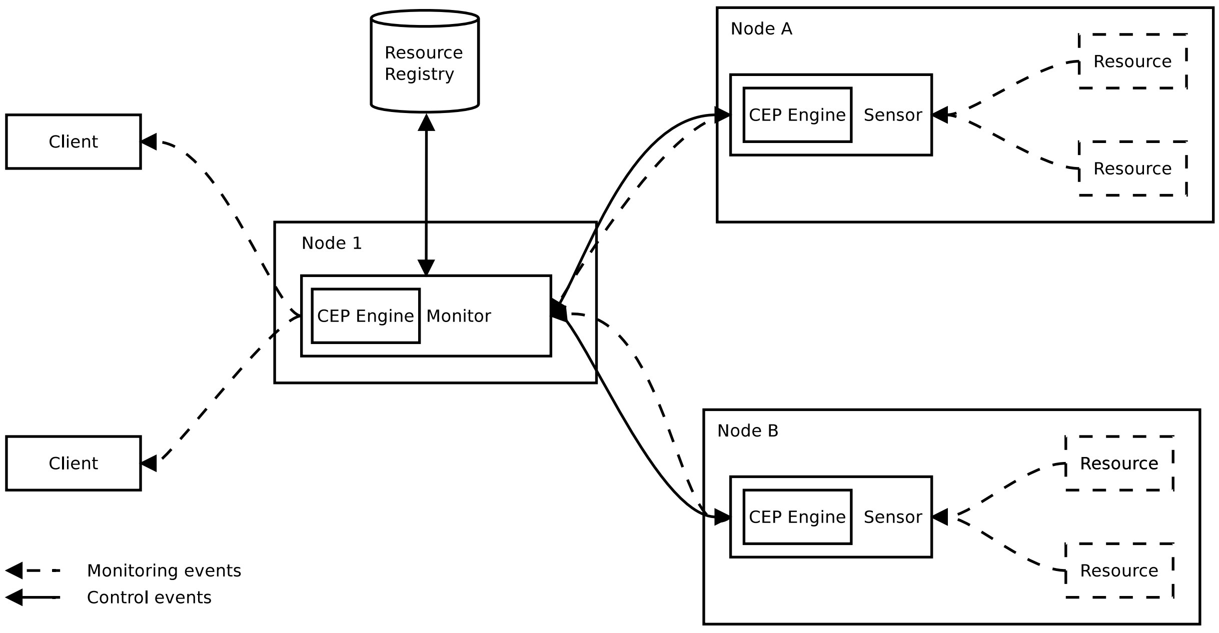


Figure 6.3: Sensor working environment. Monitor and each sensor are installed on different network nodes. The elements marked with dashed lines (such as resources) represent entities that are not a part of GEMINI-2 framework but interact with it.

The modularity was achieved by introducing Sampling Modules (see 6.8). They can be invoked and stopped by sensor module and installed an runtime by using the SPI [7]. Each sampling module is dedicated to one specific resource type and therefore "knows" how to discover, handle and probe it. Therefore, the discovery of resources is a sampling modules responsibility.

Sensor can control the functioning of the sampling modules to optimize the mentioned overhead. In particular, this is done by:

- starting sampling module only when there is any demand for events it creates. That is, any received EPL statement refers to events created by given sampling modules
- stopping sampling module when no statements refer to events it provides
- modifying the interval at which events are created by sampling module

The last feature is not yet implemented as it requires extensive analysis of EPL statements sent to the sensor.

Before sensor initializes any of sampling modules, it first has to know which ones should be invoked. The sensor resolves which subscribes in sampling modules for events in one of two ways:

- by passing a resource identifier pointing to resource that should be subject of probing. For example, sensor tells the CPU sampling module to measure only one specific core of given CPU. This mode is called *smart mode*.
- by requesting sampling module to gather all possible information about all resources it discovered. In this case, CPU sensor would probe all cores of all installed CPUs. This mode is called *plain mode*.

Sensor always tries to extract resource identifier from received EPL statement first. If this is impossible, it uses the second subscription mode.

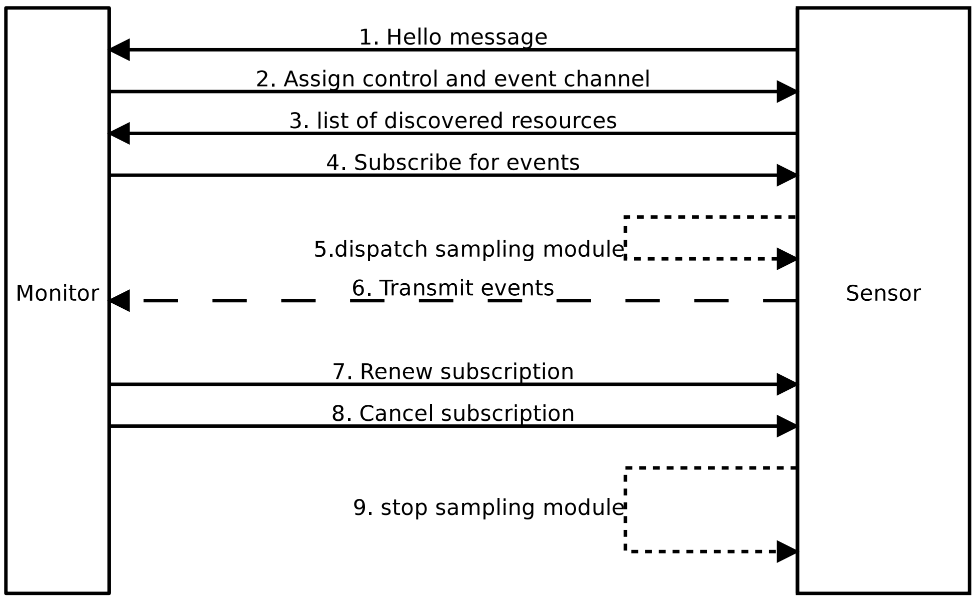


Figure 6.4: Sensor-monitor interaction diagram

6.6. Sensor-monitor interaction

As stated in 6.1 sensor cooperates with monitor in terms of providing events and accepting requests. This section discusses the details of interaction between sensor and corresponding monitor.

6.6.1. Monitor discovery

Current implementation does not contain any kind of discovery of monitor [48]. It is assumed that all sensors that should work with given monitor know its address when they are started. However, it is possible to implement some kind of discovery and lookup service for monitors similar to (or even based on) JNDI. Each monitor should register in well known registry when it starts, providing transport type and address that should be used by sensors for their registration (see 6.6). The location of this registry should be well known and available to all sensors and monitors. Therefore, a directory-service-based solution could be an answer to this problem [47].

Still, the issue how sensor should choose proper monitor to connect to remains. Monitors can be assigned to subnetworks they are supposed to gather data from. In this case, sensor should pick monitor that is in its own subnetwork.

Regardless of applied solution, the monitor discovery process should end up with each sensor having a one designated monitor it can work with.

6.6.2. Communication channels

There are three separate channels used to pass on messages between monitor and sensor. The implementation of these channels is based on endpoints available in GEMINI-2 framework [17]. Each of those three channels is used for different purpose:

- registration channel (direction from sensor to monitor) - used by sensor to send hello messages to register itself in monitor and to send updates about discovered resources. See

6.6.5.

- control channel (direction from monitor to sensor) - used by monitor to send requests for messages to sensor.
- event channel (direction from sensor to monitor) - used by sensor to send monitoring events to monitor.

More specific details of how these channels are used are described in 6.6.5.

6.6.3. Control messages

In order to cooperate monitor and sensor exchange a number of control messages known by both entities. These are listed in table 6.2.

Fields	**Data type**	**Comment**
Hello		
preferredTransportURL	string	transport url that sensor wishes to be used by monitor to send requests
preferredTransportType	string	same as above, but concerns transport type
eventTypes	list	list of event types that this sensor can provide
HelloResponse		
assignedTransportURL	string	transport url that has been picked by monitor to send requests to sensor
assignedTransportType	string	same as above, but concerns transport type. This field is ignored if assigned-TransportURL is not empty (URL already contains the transport type)
assignedId	string	id that monitor assigned to subscribing sensor
errorCode	integer	code of error that occured during subscription process. If this field is empty, everything went fine
errorMessage	string	optional error description for passed error code. This field is empty if no error occured
ResourceUpdate		
providerId	string	id of sensor sending this message
resources	list	list of resources discovered by sensor sending this message. The details of format of this list are discussed in 6.6.3
SubscribeRequest		
statementLanguage	string	Name of language that subscription statement is written in. Currently only EPL is supported
subscriptionStatement	string	the statement that should be used by sensor to detect significant events

Field	**Data type**	**Comment**
resourceId	string	identifier of given resource. Format of identifier is compliant with identifier described in 4.3.5
parentId	string	identifier pointing (equal) to `resourceId` property of parent resource
properties	list	list of resource properties. Each property has name and optionally value (suitable for static properties)
discriminators	list	list of property names that discriminate resource (see 4.3.5)
typeName	string	name of resource type. See 6.2

Table 6.3: Resource record content used in resource update message

Fields	**Data type**	**Comment**
subscriptionId	string	desired id of subscription. With this field monitor may suggest the id that should be used for new subscription
transportURL	string	transport URL that monitor will use for control channel
transportType	string	transport type that monitor will use for control channel
RenewSubscription		
subscriptionId	string	Id of subscription to be renewed

Table 6.2: Control messages

Messages **SubscribeRequest** and **RenewSubscription** were already present in GEMINI-2 framework and were used in client-monitor interaction. In fact, monitor-sensor communication is very similar to the one being discussed here.

The **ResourceUpdate** message holds list of resource in special format. Each entry contains serialized information about single resource. The content of these records are presented in table 6.6.3. Basing on on these records, the message receiver (a monitor in this case) can reconstruct whole resource hierarchy known by the sender (sensor). This is done using the fields `parentId` and `resourceId`. The resource that has no parentId becomes the root of the hierarchy.

6.6.4. Subscriptions

Sensor uses the same subscription mechanism for requests coming from monitor as the one used for handling client requests in monitors in GEMINI-2 framework. Each subscription associates the EPL statement received in **SubscribeRequest** message and event channel endpoint that should be used to transport events created by the mentioned statement. The idea of subscriptions is presented in figure 6.5.

Every subscription has a finite time to live. When it expires, the subscription is canceled automatically. The **RenewSubscription** messages can be used to prolong it. Likewise, subscriptions can be canceled before they expire by sending **Unsubscribe** message to the request recipient.

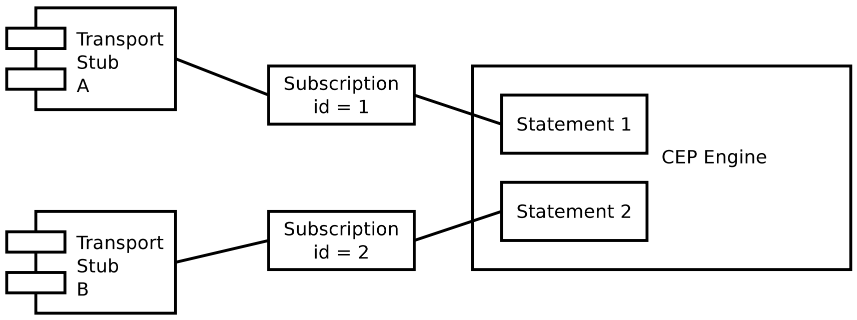


Figure 6.5: A subscription concept. Each subscription instance has a unique ID and uniquely associates an EPL statement installed in the CEP engine with transport stub that should be used to send results created by this statement.

There is one significant difference between sensor and monitor subscription mechanism. Monitor usually generates subscription ID and sends it back to the client. Sensor never does that. It always uses the ID sent in subscribe request message. The reason for this is the need to maintain the association between subscription in sensor and monitor. This link is required to handle event addressing, which is described in 5.4. Moreover, subscription requests sent to sensor are usually a result of request received by monitor. Those can expire or can be canceled by the client. In such case monitor should cancel all subscriptions that were issued to sensors as a result of original one. If subscription IDs in sensor were assigned arbitrarily they would not be associated with any subscription in monitor and canceling obsolete sensor subscriptions would not be possible.

6.6.5. Communication sequence

Figure 6.4 presents an typical message exchange sequence between monitor and sensor, starting from sensor registering in designated monitor.

The hello message is sent by the sensor right after it has started and chosen proper monitor. When receiving it, monitor checks the desired transport type and URL of control channel and decides whether it will use them or pick new ones. After that, a unique id for new sensor is generated and the information regarding given sensor, that is the address of communication channel and identifier, is put into event provider registry. Finally, a response message is sent back to sensor, with generated sensor id and actual transport parameters.

Sensor remembers received ID for future use and creates its end of control channel. Then, it sends to monitor first **ResourceUpdate** message with received ID through the registration channel. Upon receiving it, monitor updates resource registry. This message it generated by sensor periodically to inform monitor about any changes and signalize that sensor is still alive. At this point monitor knows all new resources and data types that can be provided by the new sensor and is ready to send requests to it.

When such request is received by the sensor, it analyzes the statement included in it and invokes proper sampling modules (if needed). It also adds the received statement to the CEP Engine and associates its endpoint of event channel with transport parameters contained in registration request (monitor provides an address that it listens on for events). At this point, the event channel is established and as soon as first event is generated by the CEP engine it will be sent to monitor.

Because the subscriptions are expiable (see 6.6.4 monitor sends **RenewSubscription** messages periodically to the sensor in order to keep the subscriptions valid.

When given subscription in monitor expires it sends the **Unsubscribe** message to sensor to inform it that EPL statement associated with it is no longer valid. At this point, sensor checks if sampling module started before is still needed (some other subscriptions may still take advantage of it). If not, it is stopped.

Presented sequence is in considerable part same as interaction between client and monitor. This concerns the creating, renewing and canceling subscriptions and receiving events. This compatibility is significant because of future possibility to unify monitor-monitor and monitor-sensor communication which is crucial to satisfy the communication uniformity proposition mentioned in 5.2.

6.7. Sensor architecture

Figure 6.6 presents detailed view of sensor architecture. The dashed lines represent flow of events containing monitoring data while solid ones symbolize flow of control messages and signal exchange between components.

The transport stubs are responsible for passing events between remote endpoints of GEMINI-2 framework. They isolate the higher layers of monitoring infrastructure from specific implementation of transportation over the network. Each transport stub handles one designated transport method or protocol. Currently there are four types of transport stubs available: web services (CXF implementation), JMS (Java Message Service), plain TCP and within Virtual Machine. The first three were already part of GEMINI-2 system when work described in this paper began. The last one was introduced later to make testing and evaluation easier.

The transport methods implemented in GEMINI-2 are unidirectional. That is, they enable sending information only in one direction. Returning a result by the receiving end is possible but communication is always initiated by only one side. Because of that, sensor module uses three transport channels, each represented by one endpoint connected to a transport stub. These stubs are:

- event transport stub - used to *sent* event objects released by the CEP engine to monitor
- request transport stub - used to *receive* requests from monitor that manages given sensor containing EPL statements.
- control transport stub - used to *send* messages containing set of resources discovered by this sensor and some control messages such as hello message used to initialize sensor-monitor interaction (see 6.6

Each transport stub is defined by type that defines protocol and concrete technology that will be used to transport data (mentioned JMS and web services are transport types) and an address. The format and meaning of the latter is specific for each type. For example, TCP would use hostname and port number, while JMS requires only a queue name. The concrete implementations of transport (web services, TCP, VM) can be changed in sensor configuration. By default, JMS is used to transport events from sensor to monitor and web services are used for requests and control transport.

All request received by designated transport stub are preprocessed to extract data required for subscription (see 6.6). This includes: EPL statement, subscription ID and transport type and address desired by request sender (usually monitor). This data is passed to to **Event Dispatcher**. This component performs the subscription process itself. This includes following steps:

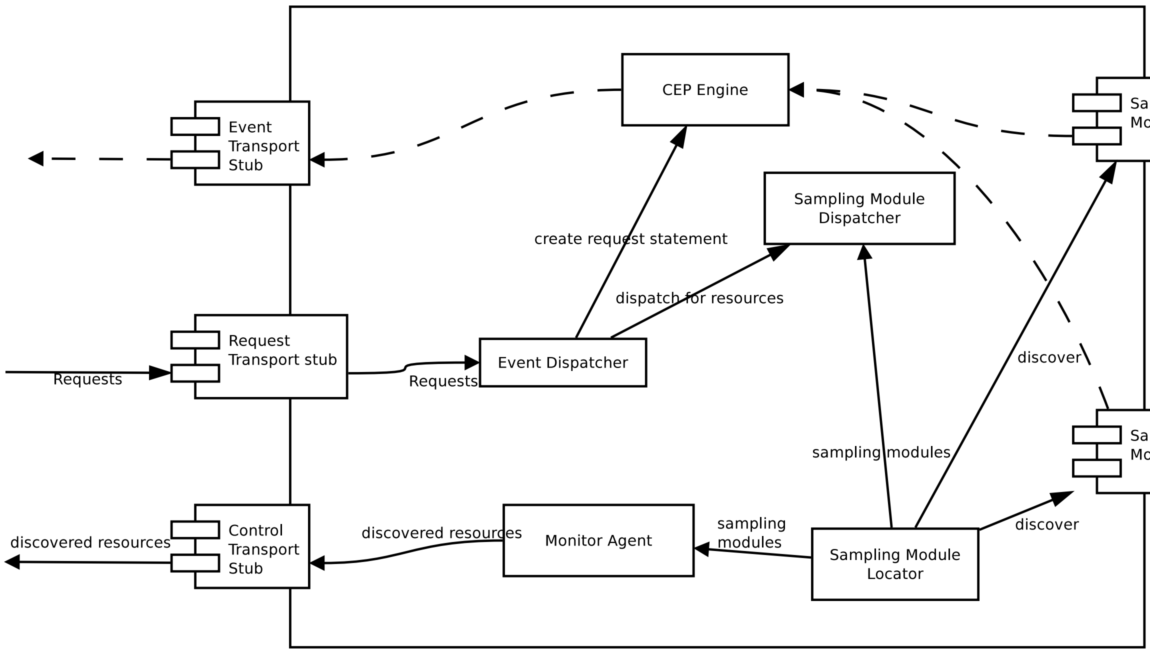


Figure 6.6: Sensor architecture. Dashed lines represent monitoring data events flow while solid ones stand for signals and control messages. Sampling modules are responsible for gathering information about particular resources and can be plugged-in at runtime.

1. assign a unique subscription id; if subscription message contains desired id, try to use it. If not, or passed ID cannot be used, generate a new one
2. determine transport type an address (for example JMS and specific queue name) that will be used to send events to. Subscriber may sent desired transport parameters but sensor is not obliged to use them and may use arbitrary transport type and address.
3. request **Sampling Module Dispatcher** to invoke proper sampling modules in order to "feed" the received EPL statement with events produced by them
4. register EPL statement in installed **CEP Engine**

As a result of subscription a response message is sent back to the subscriber containing actual subscription ID (which, as written before may but does not have to be same as ID desired by subscriber) and actual transport type and address (similar behavior as in case of subscription ID). Event dispatcher also takes care of subscription renewal and cancellation.

As stated before the task of **Sampling Module Dispatcher** is to invoke proper sampling modules. Which ones of them should be started is determined by the EPL statement from request. If statement concerns a resource that is handled by given sampling module or it refers to event type produced by this sampling module (see sampling module initialization modes in 6.5) then the module is invoked. In order to know what sampling modules are available, Sampling Module Dispatcher queries Sampling Module Locator for actual list of them. This component is plugged into the Event Dispatcher as a listener (*Observer* design pattern) and is notified about each subscription.

Sampling Module Locator is used to detect available sampling modules for given sensor. Current implementation uses SPI [7] to load sampling modules at runtime. JAR files with sampling module implementations should be put in designated directory known by the locator. A special thread checks this directory for any new JARs. If such files are found and they contain sampling modules, which is determined using SPI, they are made available to the sensor. Also, when new sampling module is detected, the Sampling Module Dispatcher is informed about this fact so that it can initialize sampling modules in advance for future use. Other implementation, that uses fixed set of sampling modules (not changed at runtime) is also prepared and was added for testing purposes.

Monitor Agent is used for interaction with Monitor that given sensor is assigned to. It gathers information about detected resources and sends it to the monitor. It is also responsible for proper initialization of communication between sensor and monitor. More information on this subject is available in 6.6.

The **CEP Engine** (Esper implementation is used) is somewhat a heart of a sensor. It receives events produced by the sampling modules and filters, processes and matches them using registered EPL statement received in requests. The results are sent to the monitor through the event transport stub. However, as mentioned in 6.1.3 to work properly it has to know the definition of all simple event types (which are usually Java objects) in advance. To satisfy that condition, same solution as in monitor implementation is applied. A message definition registry is used that contains all event types used by monitoring infrastructure. Each Java object representing simple event is marked with special annotation that specifies whether message is used for control purposes (such as request messages) or is a plain data message (containing monitoring data). These annotations are assigned using AspectJ (aspect-oriented Java extension).

Using CEP Engine in sensor improves data reduction (see 3.3.4) as obsolete events are discarded immediately at the creation point.

6.8. Sampling modules

Sensor handles resources using sampling modules. Each sampling module specialized in discovering and measuring one single resource type and provides one single event type concerning this resource.

All sampling modules implement a specific interface that is known by the sensor module. Through this interface the sensor can control the behavior of a sampling module and change its state. Figure 6.7 presents available states for every sampling module and transitions along with signals that invoke them.

The **Created** state is the initial state for every sampling module. It is triggered when the module is discovered and created by sensor. No specific actions are performed when transiting to this state.

The initialization takes place when sampling module is needed for the fist time. At this point it detects available resources for the first time and performs setup required to probe resources.

After sampling module is **initialized** it can be started. A **working** state means sampling module is probing resource and creating events at given interval. This interval can be changed to the most suitable value for current needs (see 6.5). A started sampling module can also be stopped when there is no need for events created by it. The start-stop cycle can be performed any number of times. Finally, sampling module can be destroyed when it is no longer needed.

Two sampling modules where implemented for evaluation purposes. They are presented in following part of this section.

6.8.1. Network bandwidth sampling module

This sampling module handles the network links between two hosts and measures available bandwidth between them. As stated in 4.2.2, network links are *phantom* resources. Therefore,

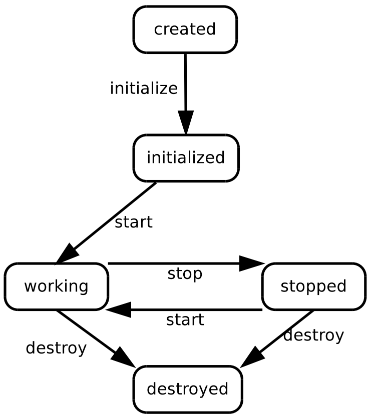


Figure 6.7: Sampling module state diagram.

this sampling module does not perform any resource discovery operation (it shows the discovered resources set as being empty). In fact, it would be virtually impossible to discover all existing network links.

Instead, it extracts the desired network link from the resource identifier received from sensor. That implies that plain mode requests (see 6.5) are not possible when it comes to this sampling module as no information about which exactly network links should be probed would be available.

For bandwidth measuring, this sampling module uses a packet train method [32]. In order for it to work, one end of network link must have a particular client installed, and other a special server. When initialized, this sampling module invokes its own server. Therefore, all nodes that have a sensor installed with this module can be taken into account for measuring purposes.

In brief, to measure bandwidth, client sends a couple of packets (packet train) to the server with well defined interval. To improve the accuracy and avoid problems caused by lost packets, more than one train is sent. The server, after receiving packets, passes them back to client. The possible traffic congestion and network delay on examined link will cause the interval between received packets to be greater than between the ones sent by the client in the beginning. Therefore, bandwidth is estimated based on this interval difference.

The main advantage of packet-train method is that it overloads the network traffic very slightly. In order for it to work, a few dozens of packets at most are required. This is very little compared to other methods such as Iperf [6] that for measuring purposes literally flood the network with thousands of packets in order to utilize all available bandwidth for short period of time. For example, measuring bandwidth on localhost (about 6.8 Gbits/s) sending more than 50 000 packets was required of total size around 1 GB. Packet train method used only 32 packets of total size of 10 KBytes.

On the other hand, the estimates based on interval differences between packets may not be as accurate as traditional methods. However, this is not a big disadvantage when it comes to on-line monitoring. The current available bandwidth value will most likely fluctuate in greater range than the estimation error.

6.8.2. CPU sampling module

Sampling module for measuring CPUs uses SIGAR (System Information Gatherer And Reporter) - a Java library originally created for the needs of Hyperic monitoring system. It provides unified API for gathering system information. SIGAR provider ways of detecting CPUs installed in system as well as gathering information about them.

The CPU Sampling Module collects data about all CPUs and their cores available in system. CPUs that do not have multiple cores are treated as having single explicit core for uniformity. Gathered static properties:

- maximum frequency - frequency at which processor can work
- CPU number - number at which CPU is visible in operating system
- core count - number of cores

For each core, CPU sampling module measures following dynamic properties:

- idle time - % of time that core was idle
- system time - % of time that core spent executing system calls
- user time - % of time that core spent executing user level calls

6.9. Distributed CEP support

Full implementation of distributed CEP described in chapter 5 is very complicated and goes beyond the subject of this work. However, a number of features and modifications were introduced to both sensor and monitor modules to support distributed CEP and make it easier to evaluate in terms of feasibility and efficiency, as well as install it in the future. This section describes them in detail.

6.9.1. CEP Engine in sensor

First implementation of GEMINI-2 framework had a CEP engine installed only in monitor. This engine processed events coming in constant streams from sensors. Sensors in turn were simply periodical monitoring data generators. Therefore, the architecture corresponded to traditional centralized CEP.

The installation of CEP Engine is a step towards the distributed CEP and data reduction. Events can now be filtered at source effectively reducing amount of data being sent to monitor. The engine in sensor operates accordingly to EPL statements received from monitor. These statements are associated with requests sent by clients to monitor. Therefore, they are part of processing one original request issued by client. In other words, processing of single request is done in at least two distributed points: sensor and monitor.

6.9.2. Request distributor stub

The most difficult thing in implementation of distributed CEP is creating an algorithm that would produce partial and assembly statements from original one. Some patterns were presented in 5.3.2 that may be used to develop one.

To evaluate distributed CEP and prepare monitor for using such algorithm, a request distributor stub was introduced in it. It uses is a simple interface that defines a service for generating EPL partial and assembly requests as well as identifiers of providers that should take part in distribution (see 5.4). In addition to this it returns a set of event types involved in partial statements and assembly statements. This is important because of the facts discussed in 6.1.3 and 6.9.4.

Current implementation of this interface simply contains a predefined set of distribution patterns, that is a projection of original EPL statements to a triple consisting of assembly statement, partial statements and set of involved sensors. The set of original statements is finite and has to be defined directly in configuration.

Although this solution is trivial it is still useful because of two reasons:

- it enables testing, evaluation and even usability of distributed CEP. However, the last one is very limited as client has to pick a specific statement from a set of available ones

- it can be easily replaced with real implementation. Most of features related with distributed CEP have been implemented to support it.

6.9.3. Event addressing

As written in 5.4, distributed CEP introduces the need for proper event addressing: by source and by request. Implementing such addressing requires support from both monitor (as event processor) and sensor (as event source). Solution for the addressing by request problem was already described in 6.6.4. For marking events with their source, a provider identifier is required. This identifier is identical with sensor id mentioned in 6.6.5. Therefore, solving the addressing-by-source problem is as simple as adding a sensor id to each event produced by it.

Current implementation marks all events coming from sensor with both request ID and sensor ID (correlation Identifier). As a result, whole event addressing problem is solved and results of partial statements can be properly assembled.

6.9.4. Complex event handling

Adding a CEP Engine to the sensor seriously affected the types of events coming from it. In prior implementation sensor produced messages of types from predefined set (such as `CpuInfo` or `NetworkBandwidth`) that was well known to monitor. Therefore, it was possible to define all event types in advance in monitor's CEP.

However, the existence of CEP engine in sensor involves the dynamically changing set of possible event types. This is because sensor produces complex events which have contents and data dependent on received EPL statement. For example, client may specify any set of properties from given event stream, along with aggregate function such as `avg` that qualifies for distribution. As a result, sensor would probably produce events containing all of the requested properties and components needed for computing the average. It is impossible to predefine such event types. Therefore, these types have to be extracted from EPL statements (see 6.9.2).

There is one more problem with dynamic event types. All predefined events that are known to both sensor and monitor are plain Java objects that are generated by Protobuf (see 6.10). They are serialized and deserialized automatically. However, when it comes to aforesaid dynamic event types, there is a need of a generic way of converting them to protobuf message that can handle virtually any type of event. To solve this problem an additional Protobuf message was added that contains a list of property names with their values. It can be easily converted to map-based-type event (see 6.1.3) and used by CEP engine in monitor. Explicit marshalling and demarshalling (in addition to operations provided by the protobuf-generated code) were also implemented.

6.10. Used technologies

Previous sections already mentioned a some of technologies used by GEMINI-2 framework with its new modifications. Below is a cumulative list of all more significant solutions in GEMINI-2.

- **Protobuf** - protocol buffers used, among others, by Google. Protobuf enables easy definition of messages that are sent over the network. Serialization and deserialization mechanisms are created automatically. Message definitions are compiled to a number of programming languages, including Java. Main advantages of protobuf-based messages is their small size and fast serialization.
- **Esper** - CEP engine Java implementation. Already described in 6.1.3
- **Spring** - J2EE application framework. Used by GEMINI-2 to assemble modules from components and smaller beans and for configuration.
- **ActiveMQ** - JMS implementation. Used by GEMINI-2 for JMS transport implementation (see 6.7)
- **Sigar** - an open source library used by Hyperic monitoring system. It contains a number of useful Java wrappers for standard system calls that can be used for gathering data on various hardware resources. Used by CPU Sampling Module (6.8.2).

6.11. Configuration

GEMINI-2 uses Spring framework to assemble and configure its components, monitor and sensor in particular. Configuration is kept in XML files, usually one per component.

This section describes the configuration of sensor and monitor. Almost all of elements described here were already implemented in GEMINI-2 framework when works described in this Thesis began. They are described for reference.

Each configuration for sensor and monitor contain a gemini context tag. It is a heart of GEMINI-2 component configuration. It defines the endpoints, transport protocols with implementation and modules that are attached to them. In other words, it describes an interface of GEMINI-2 component (sensor or monitor) that it can connect with with other elements of infrastructure. Below is an example of Gemini context from sensor configuration:

```
<geminiContext xmlns="http://cyfronet.edu.pl/gemini2/schema/spring"
        id="samplesensor">

        <endpoint uri="event:producer:sensor">
                <transport uri="jms:queue:esperQueue" />
        </endpoint>

        <endpoint uri="msm:consumer:subscriptionListener" >
                <transport uri="jms:queue:requestQueue"/>
        </endpoint>

        <endpoint uri="epc:producer:sensorSubscriber" >
                <transport uri="ws:http://localhost:9000/msm" />
        </endpoint>
</geminiContext>
```

This gemini context contains three endpoints. Each endpoint is defined with specific URI. The URIs are composed of three parts separated with colon. First one specifies the type of channel. Currently, there are three types of endpoints available. In fact, these are implementations of channels described in 6.6.2:

- `event` - channel used for transporting events
- `msm` - Monitoring Service Management channel. Used to send or receive EPL requests
- `epc` - Event Provider Control - used by sensor to subscribe in monitor and send **ResourceUpdate** messages.

As stated in 6.6.2, each channel is unidirectional. The direction of channel is specified by the second part of URI. A given endpoint (belonging to a gemini-context holder, like monitor or sensor) may be either consumer or producer. In the example above, sensor is producing to event channel (this is obvious as sensor's job is to produce events), reading from the `msm` channel (receiving requests from monitor) and producing to the `epc` channel (sending registration requests and resource update messages).

The third part of URI specifies a name of Spring bean that will be using a given endpoint. The exact interface that this bean must implement is specific to the endpoint type. More information is available in [5].

Each endpoint has a list of possible transports assigned. In given example, only one transport per endpoint is defined but more can be added. Similarly to endpoints, transports are also defined with URIs composed of parts. The first part specifies the type of transport, or in other words the protocol. The meaning and format of second part of the URI is type dependent.

Besides defining communication interface, the Gemini context is also important because it imposes the rest of configuration. In fact, both monitor and sensor configuration is built around it in order to satisfy dependencies for the beans that are attached to endpoints. Because the other parts are ordinary Spring beans with traditional Spring bean configuration and definition, they will not be listed here.

7. Evaluation

The main purpose of this chapter is evaluation of concepts, solutions and design decisions that were presented in previous parts of this Thesis. Firstly, a performance test results of GEMINI-2 framework are given with emphasis on new sensor module behavior. They cover memory usage and bandwidth utilization caused by monitoring activity. Then, an extensive use case that concerns applying GEMINI-2 framework to monitoring storage devices in distributed environment.

7.1. Performance

This section is devoted to performance tests of distributed CEP . The goal is to determine how data reduction caused by CEP engine installed on sensor may limit bandwidth utilization and at what are memory costs of conducting event processing on sensor. The latter aspect is important because it may affect the performance of node that sensor is installed on. In most cases, these nodes would have other tasks to execute. Additionally, processing time of single event was measured.

For these performance tests a simple infrastructure consisting of one monitor and one sensor connected to it was used. Both monitor and sensor were installed on the same machine communicating with JMS queues for events and internal Virtual Machine for transport. Monitor as well as sensor were working in the same instance of JVM. To generate events an artificial sampling module was use that sent events with specified interval.

One may question the credibility of results obtained on such simplified infrastructure. However, when it comes to the memory usage only the part caused by event accumulation is taken into account. Because monitor does not aggregate events, it just passes them directly to client stub, only sensor's accumulation affects increased memory usage. Moreover, the network utilization still can be measured with respect to processing of single request. Obviously, when sensor is processing more requests, the event stream created by it will be greater and will consume more bandwidth.

7.1.1. Memory usage

A couple of scenarios were tested to measure memory usage. Each scenario defines a statements sent by client and partial statement received by sensor. The latter is the main point of interest as it affects the resource usage by sensor. Each scenario also uses different event generation interval. Usually it is 1 second.

In each case the used memory was measured using the built in Java methods: `totalMemory` and `freeMemory` of the `java.lang.Runtime` class. The value was calculated as a difference of outcomes of these two methods. Moreover, the garbage collector was invoked before each reading to make sure only really needed memory is taken into account.

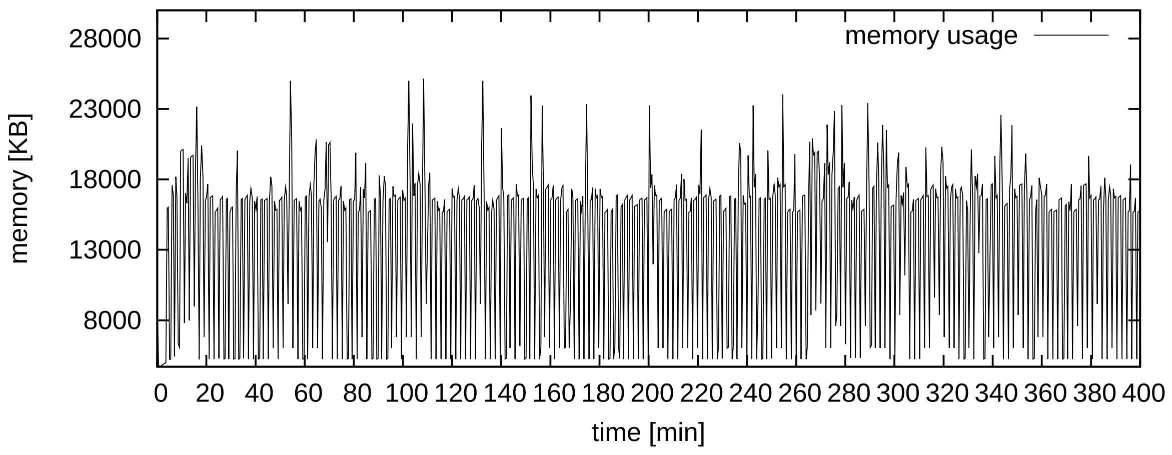


Figure 7.1: Memory usage scenario 1 results

Scenario 1

Original statement:

```
select * from CpuInfo
```

Partial statement:

```
select * from CpuInfo
```

Events were generated with 1-seconds intervals. This scenarion was used to simulate a case where no event processing takes place in sensor. The statements simply pass the events to the monitor and then client. Presented plot (figure 7.1 show that amount of used memory was fairly constant. The visible fluctuations can be bounded with constant minimum and maximum values.

Scenario 2

Original statement:

```
select avg(userTime) as au from CpuInfo.win:time(4 hours) where providerId
   = ’sensor1’ output last every 1 hour
```

Partial statement:

```
select avg(userTime) as au from CpuInfo.win:time(4 hours) output last every
    1 hour
```

In this scenario events were generated with one second interval. Figure 7.2 presents outcomes of measurement. Fluctuations visible in the figure are probably caused by temporary objects being created for processing purposes. Still, gradual increase of the minimum value of used memory is quite clear. It is caused by the fact that the window in partial statement is filling with events. When its full (after 240 minutes) the minimum usage becomes relatively constant as no more events can enter the window. They just replace the old ones.

Scenario 3

Original statement:

```
select * from CpuInfo.win:time(2 hours) where providerId = ’sensor1’ output
    all every 2 hours
```

Partial statement:

```
select * from CpuInfo.win:time(2 hours) output all every 2 hours
```

Events were generated every 1 second. Unlike in scenario 1, memory usage increases gradually and drops after every 120 minutes. This corresponds to the partial statement which contains

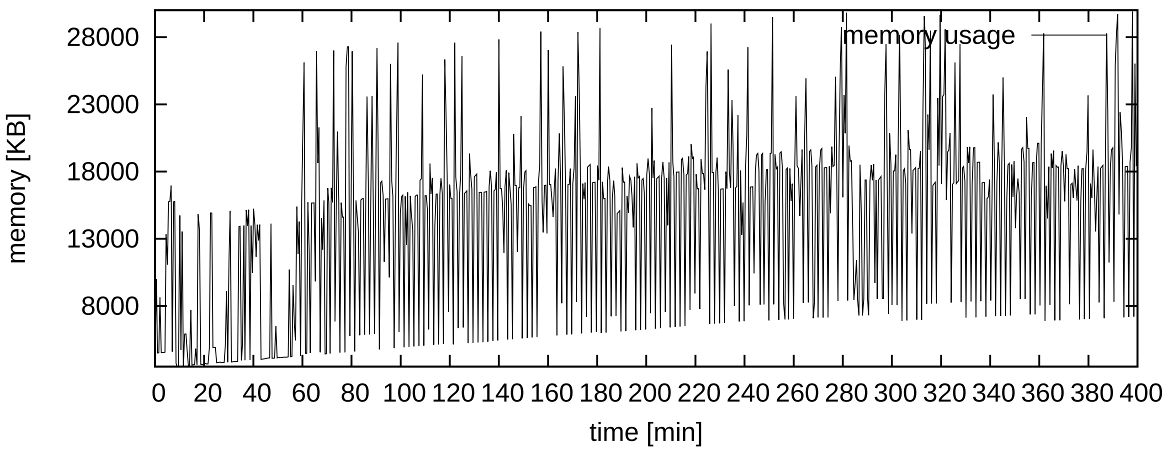


Figure 7.2: Memory usage scenario 2 results

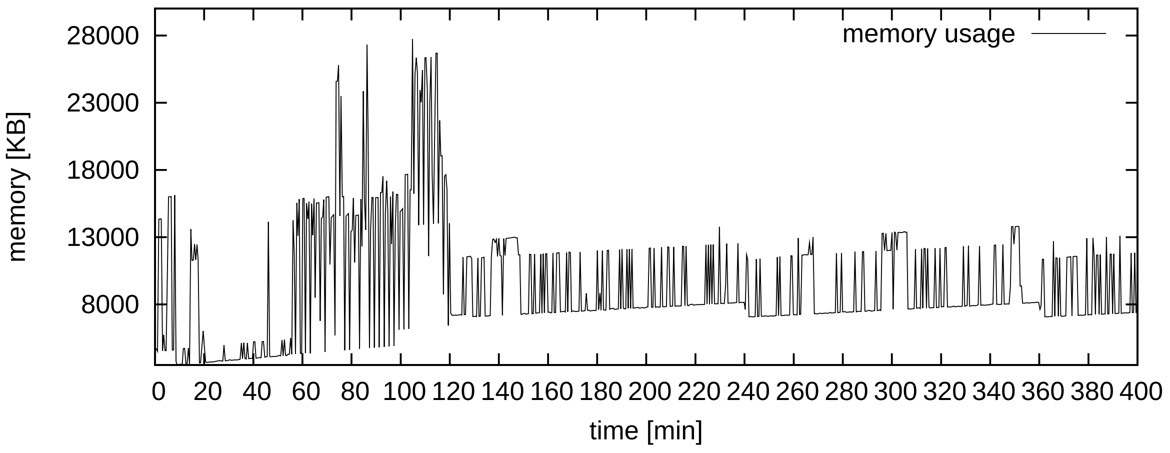


Figure 7.3: Memory usage scenario 3 results

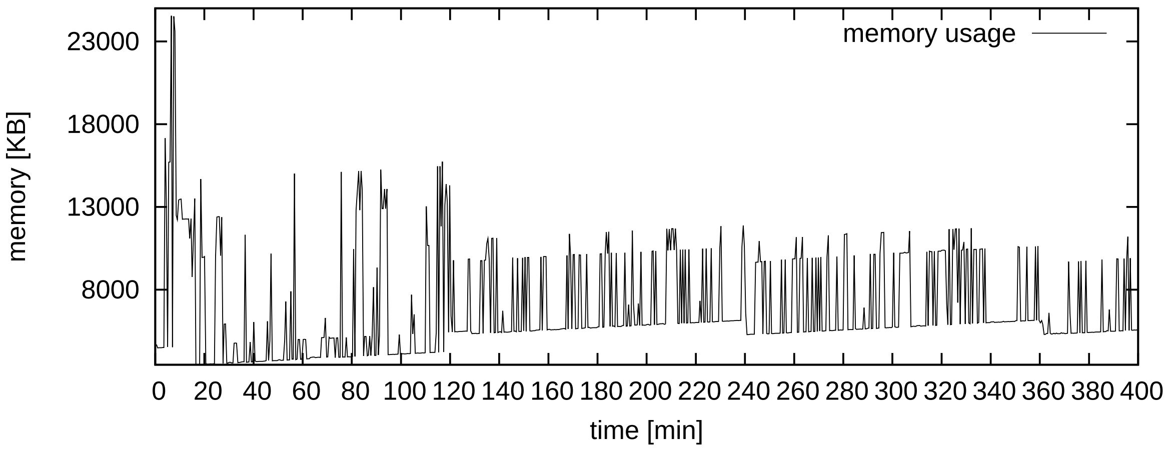


Figure 7.4: Memory usage scenario 4 results

an `output` clause that instructs CEP Engine to release all events accumulated in window after 2 hours.

Scenario 4

Original statement:

```
select * from CpuInfo where providerId = 'sensor1' output all every 2 hours
```

Partial statement:

```
select * from CpuInfo output all every 2 hours
```

Again, events were generated every 1 second. This scenario is very similar to previous one. The only difference is lack of the time window. The results are also very similar. Memory used to store events is freed every 120 minutes after gathered events are released.

Scenario 5

Original statement:

```
select * from CpuInfo.win:time(2 hours) where providerId = 'sensor1' output
    all every 2 hours
```

Partial statement:

```
select * from CpuInfo.win:time(2 hours) output all every 2 hours
```

This scenario uses the same statements as scenario 3. Unlike in previous cases, in this one events were generated every 10 milliseconds. The purpose of this scenario was to check the memory usage under extremely large stream of events. The memory usage is clearly greater that in previous cases and rises quickly up to nearly 93 MB. Similarly to previous scenarios, when events are released memory used to store them is freed.

All of presented scenarios clearly indicate that memory usage increases when statements with output control or time windows are processed. Therefore, it should be taken into account when installing on nodes that are also utilized for other tasks. However, the overhead caused by given statements was not very big. It did not exceed 2 MB. The exception was scenario 4 but it used unusually dense stream of events. In other words, the memory usage caused by event processing itself seems to be acceptable.

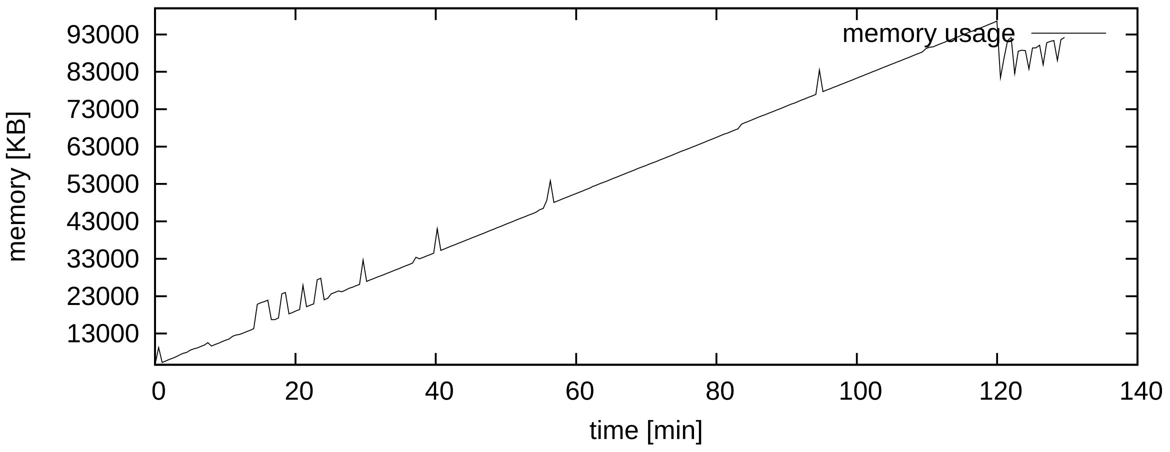


Figure 7.5: Memory usage scenario 5 results

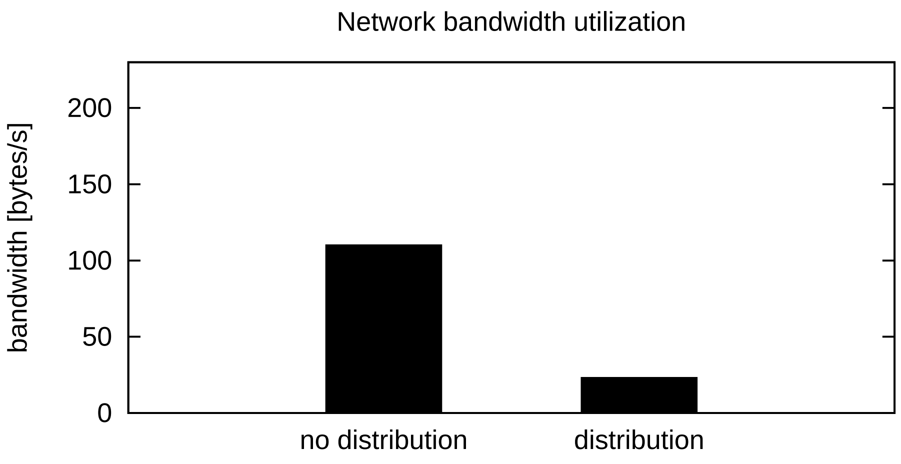


Figure 7.6: Network utilization scenario 1 results

7.1.2. Network utilization

Similarly as in case of memory usage tests, network utilization was also measured in reference to a few scenarios. In each of them a single EPL statement was examined in two modes: with and without distributed CEP. In all cases events were produced with 1-second interval. For the distributed CEP mode a partial statement that was prepared in accordance with patterns shown in 5.3.2 and was issued to sensor is given. Event messages were sent with TCP transport implementation. In order to get the amount of bytes sent over the local loopback an **iptraf** program was used. The results are presented below.

Scenario 1

Original statement:

```
select avg(usertime) as au from CpuInfo.win:time(10 seconds) output last
    every 10 seconds
```

Partial statement:

```
select sum(usertime) as s, count(usertime) as c from CpuInfo.win:time(10
    seconds) output last every 5 seconds
```

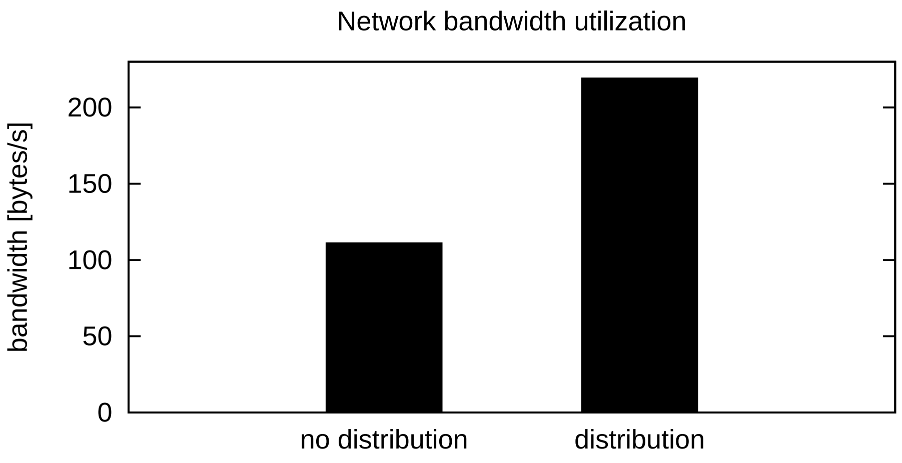


Figure 7.7: Network utilization scenario 2 results

In figure 7.6 results of scenario 1 are presented. The difference of bandwidth usage is clearly visible and is in favor of distributed CEP. In fact, utilization of network of the latter is about five times lower. This is clearly caused by the `output` clause in the partial statement that releases events every 5 seconds instead of one.

Scenario 2

Original statement:

```
select avg(usertime) as au from CpuInfo.win:time(10 seconds) output last
   every 10 seconds
```

Partial statement:

```
select sum(usertime) as s, count(usertime) as c from CpuInfo.win:time(10
   seconds) output last every 5 seconds
```

Figure 7.7 shows results for scenario 2. This one presents a very unfavorable statement distribution. It causes twice as much network utilization as standard solution. This is caused by the fact that the time window creates new result every time an events enters it or leaves. This happens more often than each 1 second. Scenario 2 shows that not every distribution is favorable and it has to be considered whether a statement should be processed in this way or not. More information about benefits from statement distribution is discussed in 5.6.1.

Scenario 1 clearly show the benefits of distributed CEP and data reduction that comes with it. Although presented bandwidth utilization values are not very high (around hundred bytes per second), they concerns single event stream. If sensor was emitting more event streams as a result of processing multiple requests, which is not an uncommon case as single sensor handles many resource instances, the utilized bandwidth would be many times higher. In such case, the profit of data reduction would be much greater.

Scenario 2 may look discouraging. Not only implementation of distributed CEP poses many problems, it can actually make the network performance worse when not used correctly. However, one should keep in mind that statement used in this scenario is quite obsolete in real applications. It is anticipated that most statements used could be profitably or at least harmlessly distributed as in scenario 1.

7.1.3. Event processing time

Time of processing single event by monitor was measured. It is defined as time between reception of event sent by sensor and releasing of event that was created as a result. Measured value ranged from 15 to 90 milliseconds, depending on CPU load of machine.

This value is significant because of the reasons presented in 5.5. Each monitor introduces a slight delay in event processing and may cause eventual data inaccuracies. Therefore, this value should be taken into account when designing the layout of monitors in monitored network.

The processing time depends very slightly on CEP engine performance, which in case of Esper is very good. Most overhead is introduced by transportation mechanisms and related serialization (especially JMS).

7.2. Storage device monitoring

This section is devoted to using GEMINI-2 framework to measuring parameters of various storage devices working in distributed environment. It is intended as a case-study for implemented changes in GEMINI-2 and simple proof of concept for resource handling and description mechanisms. I also presents a process of designing a sampling module to handle new type of resource.

Presented information is very general. This is because GEMINI-2 is still under development and implementation and configuration details may change in near future. Therefore this chapter should not be perceived as some kind of manual for implementing sampling modules for measuring storage devices. These kinds of information are available in [5].

7.2.1. Background information

Three kinds of storage systems are taken into account: disk arrays, hierarchical storage management (HSM) and ordinary local disks.

Local disks are simply single hard disk drives that are visible directly by the operating system. They do not involve any sophisticated hardware or software.

Disk arrays are set of hard disks that are used to store data in such way that it is more secure and can be more efficiently read compared to single disk. Several models of data layout over disks are available called RAID (Redundant Array of Independent Disks) levels. Each level specifies whether and how data is backed up and how redundant it is. RAID levels are designated with integer numbers. There are 5 standard levels of RAID. Disk arrays usually communicate with operating system through special hardware controller. There is also a possibility to create software RAID matrix.

Hierarchy Storage Management is a system that automatically migrates data between different levels of storage device types. Each level consists of same kind of storage devices. In most cases there are three levels. On top there are higher-cost devices, such as hard disk drives. They keep frequently accessed data. Lower are optical disk that are cheaper but less efficient. Finally, at the bottom of hierarchy are tape drives that can store significant amounts of data but suffer from very long access times. The whole idea of HSM is to migrate data between these levels in most efficient way, considering data access time and storage costs. For the need of this chapter two-level HSMs will be considered.

Name	**Data type**	**Static?**	**Description**
Local disks			
device name	string	✓	name under which device is visible in operating system. For example, disks in UNIX are named sda1, hda2 etc.
Total capacity	integer[bytes]	✓	maximum number of bytes that given disk can hold
free capacity	integer[bytes]	✗	number of bytes that can be used to write data on disk
average read transfer	integer[bytes/s]	✗	number of bytes that can be read in one second from disk on average. Although this is not a static property by nature, it after enough number of estimates it becomes quite stable and does not change over time and can be considered as such
average write rate	integer[bytes/s]	✗	same as above but concerns writing data
current read transfer	integer[bytes/s]	✗	number of bytes that are currently being read in one second
current write transfer	integer[bytes/s]	✗	same as above but concerns writing bytes
working	boolean	✗	true means the device is reading or writing data at the moment

Table 7.1: Common properties for storage devices

Name	**Data type**	**Static?**	**Description**
RAID level	integer	✓	number defining the level of RAID
disk number	integer	✓	number of disk that array consists of

Table 7.2: Unique properties of disk array resource

7.2.2. Storage devices properties

In order to design a sampling module properties of storage devices that will be measured have to be found and defined. All resources considered in this chapters have some common purpose and functions. They are all used to store data and provide at least reading and writing features. Therefore, a number of common properties can be identified. These are listed in table7.2.2. Each one has a static/dynamic qualification and optionally a description.

Moreover, HSM systems and disk arrays have some unique properties. They are presented in tables 7.2.2 and 7.2.2.

7.2.3. Resource type definition

After the properties have been found and described, resource type can be defined, according to 6.2. Metadata for each of discussed three resource type is shown in tables 7.2.3. The fields in these types correspond to static properties presented previously. For this reason the description for them is omitted because meaning is the same as in 7.2.2.

Presented information can be used by GEMINI-2 sensors to define resources and send information about them to monitors and by whole framework to define event types that will

Name	**Data type**	**Static?**	**Description**
high watermark	float	✓	% of disk space filled after which data migration to lower levels of HSM begins. Although this property is not strictly static it usually changes very infrequently and sometimes can be treated as such.
low watermark	float	✓	% of disk space filled data migration invoked by high watermark stops. Similarly to previous property, this one also can sometimes be treated as static
libraries	listof libraries	✓	list containing information about tape libraries that HSM system includes. Each library is defined by name and contains a number of tape drives. Each drive can be empty or contain a tape.
number of premigrated files	integer	✗	number of files that have been selected for migration and have been moved to special migration directory on disk (premigration state)
size of premigrated files	integer[bytes]	✗	same as above but concerns total size of these files
number of migrated files	integer	✗	number of files that have been migrated to lower level of HSM hierarchy
size of migrated files	integer	✗	same as above but concerns total size of these files

Table 7.3: Unique properties of disk array resource

contain information about these resources state. Typically, a monitoring data event contains values of all non-static properties. For example, as far as disk array monitoring events is concerned, event would contain values for `freeCapacity`, `averageReadTransfer`, `averageWriteTransfer`, `currentReadTransfer`, `currentWriteTransfer` and true/false value for `isWorking`. Therefore, both event and resource definition are ready.

The fact that some common group of properties have been identified in 7.2.2 suggest using some kind of inheritance mechanism for resource metadata. HSM, Local disk and Disk Array resource types could have a common supertype: Storage Device. Unfortunately current implementation of resource handling in GEMINI-2 does not support such constructions.

7.2.4. Event types definition

Based on resource type definitions established in previous section event types that will carry information about particular resources can be defined. They are presented in table 7.2.4. Most fields in those events are self explanatory or correspond to fields in resource types therefore their description is omitted.

Unlike resource types, presented event type set do take advantage of the fact that some common properties can be identifier among storage devices. Events `StorageDeviceDynamicParameters` and `StorageDeviceStaticParameters` can be used to transport data that concerns any device used to store data. Other events detail this information. Messages concerning the same resource instance can be associated using `resourceID` as they will have same value in this field.

Static and dynamic properties have been put into separate events. Such solution helps avoiding situation, where a lot of dynamic properties are measured while consumer requests only single static one. In presented solution, consumer can explicitly request for static properties only.

7.2.5. Sampling module implementation

After events content are defined and resource properties described, one may start to implement sampling modules. One sampling module per resource type will be required. Each of them will produce events with type corresponding to measured resource. The implementation details will not be discussed here. Measuring local disk parameters is usually fairly easy. However, getting information about HSM systems strongly depends on their producer and operating software. Same applies to discovery of those resources.

During design and implementation of sampling module one has to pay attention to number of things:

- time required to perform single measurement must not be long. Typical probing times range 1-5 seconds

- sampling module should rely on native measurement utilities (operating system tools, hardware vendor tools etc.) as much as possible to ensure efficiency and measurement accuracy

Sampling module should be packaged in JAR file with the main class that handles measurement specified in META-INF directory accordingly to SPI specification [7]. Thanks to that Sampling Module Locator (see 6.7) will be able to detect and initialize it. Finally, the whole jar should be placed in designated directory that holds all available sampling modules.

Field	**Value**
Name	"HSM"
Property names	deviceName
	totalCapacity
	freeCapacity
	averageReadTransfer
	averageWriteTransfer
	currentReadTransfer
	currentWriteTransfer
	isWorking
	lowWatermark
	highWatermark
	libraries
Discriminator	deviceName
Static properties	deviceName
	totalCapacity
	lowWatermark
	highWatermark
	libraries

(a) HSM metadata

Field	**Value**
Name	"DiskArray"
Property names	deviceName
	totalCapacity
	freeCapacity
	averageReadTransfer
	averageWriteTransfer
	currentReadTransfer
	currentWriteTransfer
	isWorking
	RAIDLevel
	diskCount
Discriminator	deviceName
Static properties	deviceName
	totalCapacity
	RAIDLevel
	diskCount

(b) Disk array metadata

Field	**Value**
Name	"Disk"
Property names	deviceName
	totalCapacity
	freeCapacity
	averageReadTransfer
	averageWriteTransfer
	currentReadTransfer
	currentWriteTransfer
	isWorking
Discriminator	deviceName
Static properties	deviceName
	totalCapacity

(c) Local disk metadata

Table 7.4: Resource metadata for storage devices

Field	Datatype
resourceID	string
parentID	string
totalCapacity	integer[bytes]
deviceName	string

(a) StorageDeviceStaticParameters

Field	Datatype
resourceID	string
parentID	string
freeCapacity	integer[bytes]
averageReadTransfer	string
averageWriteTransfer	string
currentReadTransfer	string
currentWriteTransfer	string
isWorking	boolean

(b) StorageDeviceDynamicParameters

Field	Datatype
resourceID	string
parentID	string
libraries	list of Library
highWatermark	integer
lowWatermark	integer

(c) HSMStaticParameters

Field	Datatype
resourceID	string
parentID	string
diskCount	integer
RAIDLevel	integer

(d) DiskArrayStaticParameters

Table 7.5: Event types for storage devices

Field	Datatype
resourceID	string
parentID	string
hostName	string

(a) AgentInfo

Field	Datatype
resourceID	string
parentID	string
agentHostName	string
hostName	string

(b) ControllerInfo

Table 7.6: Additional event types for example purposes

7.2.6. Example

This part contains an example of distributed CEP based monitoring framework. Its main purpose is to provide a clear overview of how concepts presented in this Thesis regarding mentioned approach (especially in chapter 5) may work in real solution.

The example refers to architecture presented in figure 5.11. All presented EPL statements use event types defined in 7.2.4. In addition to them, events presented in 7.2.6. They concern information about agents and disk controllers respectively.

Following assumptions are made:

- that the infrastructure is up and running, that is all required connections between components are established, and no errors occur during the processing.

- the sensors installed at each disk array produce events periodically, sending a constant stream of events to disk controllers.

- the event property `resourceID` uniformly defines event producer. This fact will be needed in statements with the `group by` clause

- disk array resource belongs to (is child of in resource hierarchy of) disk controller resource.

Below are actions, in order, that are taken by each component of the architecture.

1. Consumer sends request to the front monitor containing a following EPL statement:

   ```
   select ai.hostName as agentId, avg(da.freeCapacity) as avgFreeSpace
      from StorageDeviceDynamicParameters.win:time(1 minute) as da,
      ControllerInfo as ci, AgentInfo as ai where da.parentID =
      ci.resource Id and s.agentHostName = AI.hostName group by
      ai.resourceID output last every 10 seconds
   ```

   With this statement consumer attempts to obtain information about the trend in used disk space for each disk array in both clusters

2. the front monitor receives the request and assigns an id to it (needed for event addressing - see 5.4) *usedSpace*. It also examines the statement contained in the request and establishes following facts:

   - Two event types are involved in the statement: `StorageDeviceDynamicParameters` and `ControllerInfo`
   - There is a `group by` clause that "splits" the events in accordance to the sources. In other words, each group comes from different source
   - there is an `avg` aggregation function involved

3. Basing on the information above, the front monitor sends following requests to cluster agents:

   ```
   select ai.hostName as agentId, avg(da.freeCapacity) as avgFreeSpace
      from StorageDeviceDynamicParameters.win:time(1 minute) as da,
      ControllerInfo as ci, AgentInfo as ai where di.controllerId =
   ci.resourceID  and ci.agentHostName = ai.hostName output last every 10
     seconds
   ```

   simultaneously, a following statement is used as assembly one:

   ```
   select * from aggregateStream where requestId = 'usedSpace'
   ```

   The `aggregateStream` is an event stream composed of events received from the agents. The type of the events that will be contained in this stream is determined when the partial requests are constructed. The `where` clause assures that only events associated with this particular request will be used to compute final result. Apart from partial statement, also request id is sent to agents.

4. Both agents receive their requests. They read the request id contained in them and remember it for future use. They analyze the statement and, just like the front monitor, construct their own partial statement:

   ```
   select ci.agentHostName as agentId, sum(da.freeCapacity) as
      freeSpaceSum, count(da.freeCapacity) as freeSpaceCount from
      StorageDeviceDynamicParameters.win:time(1 minute) as da,
      ControllerInfo as ci where da.parentID = ci.resourceID group by
      ci.resourceID output last every 5 seconds
   ```

   and assembly statement

```
select ai.hostName as agentId,
   sum(ag.freeSpaceSum)/sum(ag.freeSpaceCount) as avgFreeSpace from
   aggregateStream.std:unique(providerId) as ag, AgentInfo as ai where
    ai.hostName = ag.agentId and requestId = 'usedSpace' group by
   ai.hostName output last every 10 seconds
```

In both statements the `group by` clause is added only to make them compilable (events have to be grouped by the selected fields, just like in SQL). Technically, the grouping is done by splitting the original statement to partials Again, request sent to disk controllers contain the request id.

5. Disk controllers receive the statements and install them in their CEP Engines. After 5 seconds, their CEP engines generate first events. Before they are sent to the agents, request id and provider id are added to them.

6. When agents receive events, they put them into their CEP engines with proper assembly statements installed. Their output control tells the engine to release events every 10 seconds. After that time (assuming that at least one event provided by disk controller arrives before which should occur because of 5-second interval in their statements). As soon as any event leaves the CEP engine, requestId and providerId are added to it.

7. finally, the front monitor receives the first event. Just like in other cases, it is put into monitor's CEP engine. When event the engine releases event, it is passed to the consumer.

Obviously, presented scenario does not cover all aspects of distributed CEP. For instance, the problem of establishing communication channels between providers and consumers, event type recognition from statements and statement distribution mechanism itself still need to be resolved. Some of these are discussed in chapter 6.

Moreover, it has to be admitted that brought up EPL example is fairly complex. In order to distribute it in a way as above, the distribution algorithm would have to analyze the join relations between three event types. Fortunately, real-life statements probably would use such constructions fairly rarely.

8. Summary

Presented work made a significant step in GEMINI-2 framework development towards more efficient and complete monitoring infrastructure. Particularly, a problem of distributed CEP was defined and then examined and evaluated using mentioned system. It proved to be relatively complex to design and implement. A number of potential problems regarding applying this technology in monitoring systems was identified and described. Moreover, some directions were given that could help implementing it. Although GEMINI-2 does not fully support distributed CEP yet, a number of improvements were introduced to bring it closer to complete functioning of this technology.

The test results are promising as far as network utilization as well as memory consumption of monitoring infrastructure are concerned. This indicates that the development of distributed CEP support in GEMINI-2 may be beneficial and make this system an efficient monitoring platform that is able to compete with other solutions in this field.

These conclusions indicate that the main goal of this Thesis regarding applicability and evaluation of distributed CEP in grid monitoring was reached. When it comes to the other problems mentioned in 1.2 they were also solved. A sensor module was designed and implemented, making GEMINI-2 more extensible and capable. In addition to this, some basic concepts regarding resource handling were defined, described and introduced to discussed monitoring system.

Following sections take some other issues regarding work described in this paper. These include the fulfillment of requirements for monitoring system and plans for the future when it comes to the development of GEMINI-2.

8.1. Functionality compliance

Section 1.3 defined a set of requirements that a monitoring infrastructure for distributed environments should meet. The implementation of GEMINI-2 sensor component and modifications introduced to monitor helped some of them to be met by GEMINI-2 framework. They are listed below, with short justifications:

- **diverse data granularity** (1.3.2) - CEP technology provides varied data granularity support out of the box. User can define any pattern and aggregate any types of events
- **extensibility** (1.3.5) - architecture of sensor that utilizes easily replaceable sampling modules definitely contributes to overall extensibility of GEMINI-2 framework
- **low overhead** (1.3.7) - development towards distributed CEP support by introducing a CEP engine in sensor and using efficient measuring techniques (such as packet-train method to probe bandwidth) reduce the overall overhead that GEMINI-2 may have on environment it is installed in.

- **on-line operation** (1.3.6) - the CEP technology itself is on-line oriented in nature.

Still, a number of requirements awaits resolving. Most important ones include security, data storage and ease to deploy.

8.2. Future challenges

In chapter 6 it was mentioned that some features of GEMINI-2 framework are not implemented yet. Below is a more complete list of parts that need development.

- implementation of request distributor - this is probably the most challenging feature to be implemented of all mentioned. The analysis of EPL statement in order to extract data and operations that can be distributed poses many difficulties. They were mentioned mainly in 5 but also in this chapter. Request distributor is crucial for full distributed CEP support.

- improve security - test results presented in 7.1.1 clearly showed that long time windows cause significant memory usage. This may be a security vulnerability as malicious clients could send requests containing really long windows measured in hours or event days. Therefore, additional analysis of received statements ins required. It could be conducted by the request distributor along with distributed statement creation.

- design monitor discovery mechanism - some kind of monitor discovery is required for sensors to automatically initialize. Currently, they must have an address of monitor they are supposed to work with predefined. That makes automatic initialization and configuration of sensors very hard or impossible. This problem is discussed more thoroughly in 6.6.1.

- resource type inheritance mechanism - currently there is not possibility to define any relation between resource types. As resource type definition example presented in 7.2.2 showed, a mechanism for resource type inheritance could be useful to avoid redundant property entries. Moreover, it would help organize resources in more sophisticated way (right now they form a tree hierarchy), thus making search operations more efficient and flexible (for example, fetching all resources of given type and its subtypes).

- implement proper resource registry - as mentioned before, currently resource registry is just a stub that is used for evaluation purposes. For fully functioning monitoring infrastructure a more sophisticated component is needed. The requirements for it include:

  – generally accessible by all elements of GEMINI-2 infrastructure: monitors and clients. Monitor need it to update resource data and sometimes fetch information needed to distributed EPL statement that concerns given resources over sensors. Clients would be able to obtain data about static properties of resources without involving monitors in the process.

  – persistence - data contained resource registry should be preserved during failures of single monitors or even whole monitoring infrastructure. Usually the layout of hardware resources does not change rapidly in distributed environments so most resources would still be up to date when system is initialized after failure

  - efficiency - some resources , such as processes, do appear and disappear from distributed environment often. Therefore, frequent data updates issued to resource registry are probable. Moreover, clients may also request data from registry very frequently. Resource registry should be prepared to handle all these operations efficiently.

- unify interfaces between GEMINI-2 components - current interfaces allow monitor to request data only from sensors and accept requests from clients only. In order to enable a fully functional architecture as suggested in 5.2, a unification of these interfaces is needed. Each monitor should be able to register in other monitor just like sensor does now. Similarly, each monitor should be able to request data from other monitor just as client does it with regard to monitor and monitor with regard to sensor. In other words, each monitor should be able to work as a event consumer (client) and event producer at the same time [47]. Some work in this field been done in this area by applying the request interface used by client to monitor (see 6.3).

- introduce instrumentation - current implementation of GEMINI-2 framework does not allow user to probe state of processes. A system of process instrumentation is needed to satisfy this need. A certain solutions are presented in [15], [49] and [16]

Glossary

consumer	A computer, subsystem or any other element of distributed system interested in receiving monitoring data., 22
node	Single machine that works in network, usually part of a cluster, 11
probing	Activity of sensor or any other device/software that is responsible for handling resources that consists in gathering monitoring data about resource instances. For example, CPU probing would be extracting information about its user, system and idle time., 12
producer	Any element of distributed environment that gathers and sends monitoring data to consumers. It may be any kind of sensor, application etc., 23
resource	Any element of distributed environment that can be subjected to measurements and monitoring. This include hardware (CPUs, hard disk drives, memory, nodes, network links), software (processes, applications, operating systems) or other (databases)., 11

Bibliography

[1] Teragrid. URL `http://www.teragrid.org`.

[2] *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[3] Esper EPL Reference. URL `http://esper.codehaus.org/esper-2.0.0/doc/reference/en/html/epl_clauses.html`.

[4] *Esper Reference Documentation*.

[5] GEMINI-2 Development Site. URL `http://chomik.cyfronet.pl/trac/gemini`.

[6] *OpenSS7 IPERF Utility Installation and Reference Manual*.

[7] Java Service Provider. URL `http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html#Service%20Provider`.

[8] S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G.L. Rubini, G. Tortone, and M.C. Vistoli. Gridice: a monitoring service for grid systems. *Future Generation Computer Systems Journal*, 2005. doi: http://dx.doi.org/10.1016/j.future.2004.10.005.

[9] Sergio Andreozzi, Massimo Sgaravatto, and Maria Cristina Vistoli. Sharing a conceptual model of grid resources and services. *CoRR*, cs.DC/0306111, 2003.

[10] Sergio Andreozzi, Stephen Burke, Laurence Field, Steve Fisher, Balazs KÃşnya, Marco Mambelli, Jennifer M. Schopf, Matt Viljoen, and Antony Wilson. Glue schema specification version 1.2. Technical report, 2005.

[11] Author. Towards a Framework for Monitoring and Analyzing QoS Metrics of Grid Services. In *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference*, 2006.

[12] F. Azzedin, M. Maheswaran, and Man. Manitoba Univ., Winnipeg. Integrating trust into grid resource management systems. In *Parallel Processing, 2002. Proceedings. International Conference on*.

[13] Zoltan Balaton and Gabor Gombas. Resource and Job Monitoring on the Grid.

[14] ZoltÃąn Balaton, Peter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. Comparison of Representative Grid Monitoring Tools. Technical report, Computer and Automation Research Institute of the Hungarian Academy of Sciences, .

[15] ZoltÃąn Balaton, PÃľter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. From Cluster Monitoring to Grid Monitoring Based on GRM, .

[16] Bartosz Balis, Marian Bubak, Wodzimierz Funika, Tomasz Szepieniec, and Roland Wismüller. An infrastructure for grid application monitoring. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 41–49, London, UK, 2002. Springer-Verlag. ISBN 3-540-44296-0.

[17] Bartosz BaliÅŻ, Bartosz Kowalewski, and Marian Bubak. Leveraging complex event processing for grid monitoring, 2008.

[18] K. Balos, D. Radziszowski, P. Rzepa, K. ZieliÅĎski, and S. ZieliÅĎski. Monitoring grid resources:jmx in action. *TASK Quarterly : scientific bulletin of Academic Computer Centre in Gdansk*, 8, 2004.

[19] Peter Brunner, Hong-Linh Truong, and Thomas Fahringer. Performance Monitoring and Visualization of Grid Scientific Workflows in ASKALON.

[20] Alejandro Buchmann and Boris Koldehofe. Complex event processing. *it - Information Technology*, 51, 2009. doi: 10.1524/itit.2009.9058.

[21] Rob Byrom, Laurence Field, Steve Hicks, Manish Soni, and Antony Wilson. Relational grid monitoring architecture (r-gma), 2003.

[22] A.W. Cooke, A.J.G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S.M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. OâĂŹCallaghan, and J. Ryan. The Relational Grid Monitoring Architecture: Mediating Information about the Grid. *Journal of Grid Computing*, 2004.

[23] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003. ISSN 1570-7873. URL `http://dx.doi.org/10.1023/A:1024000426962`. 10.1023/A:1024000426962.

[24] Tiziana Ferrari and Francesco Giacomini. Network monitoring for grid performance optimization. *Computer Communications*, 27(14):1357 – 1363, 2004. ISSN 0140-3664. doi: DOI:10.1016/j.comcom.2004.02.012. URL `http://www.sciencedirect.com/science/article/B6TYP-4C1CCBW-2/2/407a151e8acc095cbbb34f1bbda50edb`. Network Support for Grid Computing.

[25] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002. URL `http://www-fp.mcs.anl.gov/\~{}foster/Articles/WhatIsTheGrid.pdf`.

[26] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid.

[27] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35:37–46, 2002. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2002.1009167.

[28] Dan Gunter, Brian Tierney, Keith Jackson, Jason Lee, and Martin Stoufer. Dynamic monitoring of high-performance distributed applications. *High-Performance Distributed Computing, International Symposium on*, 0:163, 2002. ISSN 1082-8907. doi: http://doi.ieeecomputersociety.org/10.1109/HPDC.2002.1029915.

[29] Petr Holub, Martin Kuba, Ludeek Matyska, and Miroslav Ruda. Grid infrastructure monitoring as reliable information service.

[30] Adriana Iamnitchi and Ian Foster. On fully decentralized resource discovery in grid environments. In *International Workshop on Grid Computing*, 2001.

[31] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-716-2. doi: http://doi.acm.org/10.1145/1272680.1272685.

[32] Andreas Johnsson. On the comparison of packet-pair and packet-train measurements. *SNCNW2003*, 2003.

[33] Timothy K. Kratz, Peter Arzberger, Barbara J. Benson, Chih-Yu Chiu, Kenneth Chiu, Longjiang Ding, Tony Fountain, David Hamilton, Paul C. Hanson, Yu Hen Hu, Fang-Pang Lin, Donald F. McMullen, Sameer Tilak, and Chin Wu. Toward a Global Lake Ecological Observatory Network.

[34] Wei Li, Zhiwei Xu, Fangpeng Dong, and Jun Zhang. Grid resource discovery based on a routing-transferring model. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 145–156, London, UK, 2002. Springer-Verlag. ISBN 3-540-00133-6.

[35] Bruce Lowekamp, David O'Hallaron, and Thomas Gross. Topology discovery for large ethernet networks. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 237–248, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383078.

[36] Bruce Lowekamp, Brian Tierney, Les Cottrell, Richard Hughes-Jones, Thilo Kielmann, and Martin Swany. A hierarchy of network performance characteristics for grid applications and services, 2004.

[37] Davic C. Luckham and Brian Frasca. Complex Event Processing in Distributed Systems. Technical report, Program Analysis and Verification Group Computer Systems Lab Stanford University, 1998.

[38] David Luckham. SOA, EDA, BPM and CEP are all Complementary. *complexevents.com*, 2007.

[39] David Luckham and Roy Schulte. *Event Processing Glossary*. Event Processing Technical Society, 2008.

[40] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004. ISSN 0167-8191. doi: DOI:10.1016/j.parco.

2004.04.001. URL `http://www.sciencedirect.com/science/article/B6V12-4CMHWWX-2/2/b6b44ba67c732867d1c3881c510b2953`.

[41] Brenda M. Michelson. Event-driven architecture overview. Technical report.

[42] D.A. Reed and C.L. Mendes. Intelligent Monitoring for Adaptation in Grid Applications. In *Proceedings of the IEEE* .

[43] David B. Robins. Complex Event Processing. In *CSEP 504*, 2010.

[44] Uwe Schwiegelshohn, Rosa M. Badia, Marian Bubak, Marco Danelutto, Schahram Dustdar, Fabrizio Gagliardi, Alfred Geiger, Ladislav Hluchyi, Dieter KranzlmÃijller, Erwin Laure, Thierry Priol, Alexander Reinefeld, Michael Resch, Andreas Reuter, Otto Rienhoff, Thomas RÃijter, Peter Sloot, Domenico Talia, Klaus Ullmannu, Ramin Yahyapoura, and Gabriele von Voigt. Perspectives on grid computing. *Future Generation Computer Systems*, 2009.

[45] Shava Smallen, Kate Ericson, Jim Hayes, and Catherine Olschanowsky. User-level grid monitoring with inca 2. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 29–38, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-716-2. doi: http://doi.acm.org/10.1145/1272680.1272687.

[46] T.S. Somasundaram, R.A. Balachandar, V. Kandasamy, R. Buyya, R. Raman, N. Mohanram, S. Varun, and Chennai Anna Univ. Semantic-based grid resource discovery and its integration with the grid service broker. In *Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on*. doi: 10.1109/ADCOM.2006.4289861.

[47] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, Martin Swany, and the Grid Performance Working Group. White Paper: A Grid Monitoring Service Architecture (DRAFT). .

[48] Brian Tierney, Brian Crowley, Dan Gunter, Jason Lee, and Mary Thompson. A Monitoring Sensor Management System for Grid Environments, .

[49] Hong-Linh Truong, Thomas Fahringer, and Schahram Dustdar. Dynamic instrumentation, performance monitoring and analysis of grid scientific workflows. *Journal of Grid Computing*, 3:1–18, 2005.

[50] C. Zanga and Y. Fana. Complex event processing in enterprise information systems based on rfid. *Enterprise Information Systems*, pages 3–23, 2007.

[51] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163 – 188, 2005. ISSN 0167-739X. doi: DOI:10.1016/j.future.2004.07.002. URL `http://www.sciencedirect.com/science/article/B6V06-4DH2G0J-1/2/0ffdfce1044b5862dfea24d1d5dc51f8`.