Latency Minimization of Order-Preserving Distributed Event-Based Systems

Latenzminimierung sortierender verteilter Ereignisverarbeitungssysteme

Der Technischen Fakultät

der Friedrich-Alexander-Universität Erlangen-Nürnberg

zur

Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

Christopher Mutschler

aus Nürnberg

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg.

Tag der mündlichen Prüfung:	**19.02.2014**
Vorsitzende des Promotionsorgans:	**Prof. Dr.-Ing. habil. Marion Merklein**
Gutachter:	**Prof. Dr. Michael Philippsen**
	Prof. Dr. François Bry

“If I can’t handle events,
I let them handle themselves.”
– Henry Ford

For Sylvia and Emilie.

Abstract

Nowadays sensors are increasingly deployed in many kinds of applications and deliver streams of data that they continuously collect. This data is significant as it provides important information in real time. However, without a fast processing of this information the sensors only provide a pointless stream of data. Hence, there is a need for automatic processing to extract meaningful information within an acceptable amount of time. This thesis describes techniques that allow a fast analysis of streaming data.

Real-time Locating Systems (RTLSs) or Radio Frequency Identification (RFID) systems provide several thousand position events per second. Event-based systems (EBSs) meet the high performance requirements and are a powerful technique for a reactive analysis of such data streams. Detection algorithms are divided up into several comparatively small event detectors (EDs), become inherently scalable through distribution, and are easy to maintain because of their reduced software complexity. Such event detectors communicate by messages, i.e., events, over an event processing middleware and are hierarchically linked to detect the final events of interest.

Since partial results, i.e., events, are generated at different points in the system they are no longer timely synchronized. However, the algorithms implemented in the event detectors assume a timely ordered event stream as they try to detect interaction patterns. It is never a viable solution to process events out of order. This puts a significant workload to the underlying middleware. A-priori estimations of reordering parameters cannot include runtime information about object and system behavior, and thus the event loads, and must hence be set too conservatively in order to avoid system failures caused by ordering mistakes. But this often results in high detection latencies.

This thesis describes how to optimally adapt to variations in the observed environment to minimize detection delays at runtime. We show how out-of-order events are transparently reordered with low latency at each node so that event detectors may process them in a correct order. A speculative processing exploits unused system resources and reduces detection latency to a

minimum. We further present a technique to migrate event detectors between nodes at runtime and show how to optimize their detection latency introduced by networking delays in a distributed system environment. Our system is scalable as the number of trackable objects and measurement sensors grows.

The methods presented in this thesis compare very well with methods proposed so far. We show that we reduce latency of distributed event-based systems adaptively by integrating available system resources dynamically to fit the performance requirements at runtime for a continuously changing environment. At the same time the semantics of event detector implementations remain untouched. Our system deals with any type of delays, does not need to be parameterized a-priori, and is fully scalable. The author is not aware of any published method performing significantly better.

Zusammenfassung

Sensoren und andere miniaturisierte Messgeräte finden sich in immer mehr Anwendungsbereichen. Diese generieren Datenströme, welche interessante Informationen in Echtzeit enthalten. Allerdings bestehen solche Datenströme lediglich aus einer sinnlosen Flut an Informationen, solange sie nicht geeignet analysiert werden. Um aus den Daten in vertretbarer Zeit wertvolle Informationen extrahieren zu können, müssen die Ströme automatisch verarbeitet werden. Die vorliegende Arbeit beschreibt Techniken, um aus Stromdaten zur Laufzeit und mit geringer Latenz Ereignisse erkennen zu können.

Echtzeitlokalisierungssysteme oder RFID-Systeme stellen mehrere Tausend Objektpositionen pro Sekunde zur Verfügung. Ereignisbasierte Systeme sind eine Möglichkeit, diese Daten automatisch zu verarbeiten, da sie die Anforderungen an die Performanz erfüllen – sie bieten eine mächtige Methode um die Daten reaktiv zu verarbeiten. Algorithmen zur Detektion von Ereignissen können so über mehrere sog. Ereignisdetektoren verteilt werden und sind folglich inhärent skalierbar. Zusätzlich werden sie aufgrund reduzierter Softwarekomplexität leichter wartbar. Ereignisdetektoren kommunizieren dabei über Ereignisse, welche über eine spezielle Diensteschicht (*engl.* Middleware) verteilt werden. Diese verknüpft die Ereignisdetektoren in geeigneter Weise miteinander und koordiniert die Datenflüsse, sodass die finalen Ereignisse detektiert werden können.

Da Berechnungen nun entkoppelt voneinander stattfinden, werden partielle Ergebnisse (Ereignisse) nicht mehr in zeitlich korrekter Reihenfolge an den Eingängen der Ereignisdetektoren empfangen. Dies erhöht die Synchronisationsarbeiten innerhalb der Diensteschicht. Da die Ereignisdetektoren eine zeitlich korrekte Sortierung der Nachrichten annehmen, ist es daher keine Alternative, einzelne Ereignisse in nicht zeitlich korrekter Reihenfolge zu verarbeiten. Allerdings sind das Objekt- und Systemverhalten und somit die Ereignisraten vor der Laufzeit nur unzureichend bekannt, was bedeutet, dass Sortierungsparameter – wenn sie vorher abgeschätzt werden – zu konservativ gewählt werden müssen. Dies hat hohe Detektionslatenzen zur Folge, was

zwangsläufig dazu führt, dass Parameter zur Laufzeit gemessen und gesetzt werden müssen.

Diese Dissertation beschreibt, wie sich ein Ereignisverarbeitungssystem idealerweise zur Laufzeit an die aktuellen Gegebenheiten anpassen kann, um die Detektionslatenzen zu reduzieren. Sie zeigt, wie man zeitlich falsch sortierte (*out-of-order*) Ereignisse zur Laufzeit an jedem Rechenknoten sortiert, ohne dabei Änderungen an den Ereignisdetektoren vornehmen zu müssen. Spekulatives Verarbeiten einzelner Ereignisse kann darauf aufbauend dazu verwendet werden, verfügbare Rechenkapazitäten zur Latenzreduktion zu verwenden. Die vorliegende Arbeit beschreibt weiterhin, wie man Ereignisdetektoren zur Laufzeit zwischen Rechenknoten migriert, um somit die Verteilung der Ereignisdetektoren im Netzwerk latenzminimierend zu optimieren. Das resultierende System ist hoch skalierbar, um der steigenden Anzahl an lokalisierbarer Objekte und Sensoren auch zukünftig gerecht zu werden.

Die vorliegenden Methoden erbringen im Vergleich zu allen bisher entwickelten Techniken sehr gute Ergebnisse. Sie können die Latenz in verteilten ereignisbasierten Systemen adaptiv reduzieren, indem verfügbare Ressourcen hinzugenommen und die Performanz so dynamisch an die Anforderungen angepasst wird. Gleichzeitig bleiben Implementierungsdetails der Ereignisdetektoren unberührt. Das präsentierte System muss nicht konfiguriert werden, behandelt verschiedenste Sorten von Verzögerungen und ist äußerst skalierbar. Methoden und Techniken, die besser als die hier präsentierten sind, sind bis dato nicht bekannt.

Acknowledgements

This work has been put together in collaboration between the Fraunhofer Institute for Integrated Circuits (IIS) and the Chair for Programming Systems (Computer Science 2) at the University of Erlangen-Nuremberg. In this regard I want to give special thanks to Prof. Dr. Michael Philippsen who has himself proven as a very profound supervisor that contributed much to the relevance of the contents of this thesis and its sharpness in elaboration.

I would like to thank Nicolas Witt, Stephan Otto, Thorsten Edelhäußer, Christoffer Löffler, and Thomas Jakobs for their help and support in developing software components that are used for the implementation of the methods presented in this thesis.

Acknowledgments also go to the DEBS (Distributed Event-Based Systems) community for their wide interest in the 2013 ACM DEBS Grand Challenge [49], whose task was to analyze the position stream data of the RedFIR locating system in a 8 vs. 8 player match on a half-sized playing field. This work helped in re-identifying early problems and issues that are raised when processing high data rate position data streams from Real-time Locating Systems. Especially the accepted papers [16, 60, 71, 74, 97, 137] provided strong and clever solutions for data stream analysis in soccer applications. The work of Matthias Völker, Holger Ziekow, and Zbigniew Jerzak in developing the Grand Challenge description and paper [49] is very much appreciated. I also like to thank Prof. Dr. François Bry for his initial encouragement to set up the competition, his valuable input at the 2012 DEBS PhD workshops, and his willingness to serve as the second reviewer of this thesis.

Finally I like to take the opportunity to thank my wife Sylvia for her constantly recurring support, help, kindness, and encouragement. Without all this support this thesis would not have been what it is. Thank you for bearing all these hard times.

Christopher Mutschler
Erlangen, 2014

Contents

Abstract i

Zusammenfasssung iii

Acknowledgements v

List of Figures xii

List of Algorithms xiii

1 Introduction 1

1.1 Event-Based Systems 2
1.2 Research Objectives 6
1.3 Thesis Structure 8

2 Preliminaries 11

2.1 Fundamental Definitions 11
2.1.1 General Definitions 11
2.1.2 Time Model Semantics 14
2.2 System Architecture and Implementation 17
2.2.1 Overview 17
2.2.2 Implementation 18
2.2.3 The Event Detector API 20
2.3 Related Work 22
2.3.1 The Borealis Stream Processing Engine 23
2.3.2 Esper 24
2.3.3 IBM InfoSphere Platform 25
2.4 Reference system details 25

2.4.1 The RedFIR Real-time Locating System 26
2.4.2 The Position Data Streams 29
2.5 Summary 29

3 Dynamic K-slack Buffering 31

3.1 Event Ordering in Streaming Systems 31
3.2 Slack-based Event Buffers 33
3.2.1 Motivation 34
3.2.2 The K-Slack Approach 35
3.2.3 Problem Definition 37
3.3 Self-Adaptive Ordering Units 39
3.3.1 Derivation of *clk* 39
3.3.2 Derivation of *K* 40
3.3.3 The *clk*-setting dilemma 41
3.3.4 Event Ordering Units 42
3.3.5 Formal Proof 45
3.3.6 Summary and Further Improvements 47
3.3.7 Implementation Details 48
3.4 Evaluation 51
3.4.1 Delay Discussion 52
3.4.2 Application Performance 56
3.4.3 Discussion 58
3.4.4 Summary 59
3.5 Initialization 59
3.5.1 Iterative Delay Calculation 60
3.5.2 Semi-Configured Delay Estimation 60
3.5.3 Summary 64
3.6 Related Work 64
3.6.1 Event Processing Systems 65
3.6.2 Methods 66
3.7 Summary 67

4 Speculative Event Processing 69

4.1 The latency/overload-dilemma 69

4.2 Related Work 72
4.2.1 Buffering Techniques 72
4.2.2 Speculative Techniques 73
4.3 Motivation 74
4.4 Speculative Processing 76
4.4.1 Event Emission and Replay 77
4.4.2 State Recovery 81
4.4.3 Event Retraction 86
4.4.4 Runtime α-adaptation 89
4.5 Implementation Details 92
4.6 Evaluation 94
4.6.1 Latency reduction 95
4.6.2 Runtime α-adaptation 96
4.6.3 Resource Consumption 97
4.6.4 Evaluation of Retraction Techniques 99
4.6.5 Discussion 100
4.7 Conclusion 101

5 Reallocation in Distributed Event-Based Systems 103

5.1 Runtime Migration of Event Detectors 103
5.1.1 Introduction 104
5.1.2 Related Work 106
5.1.3 Runtime Migration 108
5.1.4 Evaluation 113
5.1.5 Summary 116
5.2 Runtime Optimization 116
5.2.1 Introduction 117
5.2.2 Related Work 119
5.2.3 Runtime Latency Optimization 120
5.2.4 Solution 121
5.2.5 Heuristics 122
5.2.6 Evaluation 128
5.2.7 Conclusion 134
5.3 Summary 135

6 Evaluation and Discussion 137
6.1 Evaluation of Event Definition Language Approaches 137
6.2 Event Ordering Use-Case Example 144
6.2.1 Proximity detection 146
6.2.2 Player hits ball detection 149
6.2.3 Code complexity analysis 154
6.2.4 CPU consumption 155
6.2.5 Detection latency analysis 156
6.3 Runtime Migration Use Case Evaluation 157
6.4 Summary 160

7 Conclusion and Future Work 161
7.1 Conclusion 161
7.2 Related Fields of Application 163
7.3 Future Work 165

References 169

List of own Publications 189

List of Acronyms 194

A Source Code Excerpts 195
A.1 Lock-Free Ringbuffer 195
A.2 Acceleration Recognition Detector 196
A.3 Active Ball Detector 197
A.4 Proximity Detector (w/o ordering middleware) 198
A.5 Proximity Detector (w/ ordering middleware) 199
A.6 Player Hits Ball Detector 201
A.7 Non-deterministic Finite Automaton 202

Index 209

List of Figures

1.1 EPTS reference architecture for event-based systems. [110] 2
1.2 Typical structure of an event-based system. [103] 3
1.3 Exemplified detection hierarchy for a shot on goal event. . 5

2.1 The logical structure of an event instance [57]. 15
2.2 Distributed publish/subscribe EPS. 17
2.3 The EventCore middleware architecture. 19
2.4 Sample code for an event detector. 21
2.5 The signal processing chain of the RedFIR system. 26
2.6 A glass model of the ball's transmitter and inductive charger. 27
2.7 The position stream data provided by the RedFIR locating system. 30

3.1 Example for `A!BC`. 34
3.2 Out-of-order event stream. 35
3.3 K-slack on an event stream. 37
3.4 Sorting window over event stream. 37
3.5 Out-of-order event stream with clk_A. 39
3.6 Setting *clk* by an ordered event set with different delays. . . 41
3.7 Event ordering unit for event detector `D`. 43
3.8 Iterators for ordering units. 50
3.9 Position jitter delay. 53
3.10 Distribution of K. K values between i and $i + 1$ show event detectors of level i, sorted by growing K. 55
3.11 Event throughput for threads. 57
3.12 Delays and K. 58
3.13 Iterative delay update. 60
3.14 Composition of sub-delays. 62

4.1 Automatically controlled camera system. 70

4.2 Distributed publish/subscribe EPS with speculation. 74
4.3 Out-of-order examples. 76
4.4 K-slack's event insertion and AEP. 79
4.5 Speculative ordering unit with $\alpha = \frac{1}{3}$. 80
4.6 Speculative ordering unit with snapshot recovery and $\alpha = \frac{1}{3}$. 84
4.7 Event retraction on reception of `B4`. 87
4.8 Latencies of (speculative) buffering. 95
4.9 Adaptation of α. 97
4.10 CPU loads with varying α. 98

5.1 Latencies in a distributed EPS. 104
5.2 Event detector migration. 108
5.3 Steps of the cooperative handover. 110
5.4 Delay $\delta(e)$ of event `e` before migration, $\delta'(e)$ after migration; d_f is the forwarding sub-delay. 111
5.5 Delays and K after state transition. 114
5.6 Event hierarchy for an *offside* rule violation. 117
5.7 The way PSO combines weighted forces. 126
5.8 Topology of virtual network environment. 129
5.9 Subscription patterns of event detectors. 130
5.10 Dynamic latency adaption achieved by GA, CS, PSO and CS/PSO. 132

6.1 Arrival time stamps of player and ball positions. 144
6.2 Event hierarchy for player hits ball. 146
6.3 Proximity detector: `callback` function. 147
6.4 Proximity detector: `correlate` function. 148
6.5 State machine for the player hits ball detector. 150
6.6 Player hits ball event detector. 151
6.7 Received events printed in order of their arrival time. . . . 152
6.8 Software complexities and LoC of the event detectors. . . . 154
6.9 Evaluation results when applying migration. 159

List of Algorithms

3.1 Pseudo code for the ordering unit (without λ, and without reduction of K) . . . 44
3.2 Iterative Delay Calculation . . . 61
3.3 Pseudo Event Propagation . . . 63
4.1 Adding a newly received event e. . . . 82
4.2 Event emission, replay, and buffer purge. . . . 85
4.3 α-adaptation. . . . 91
5.1 Delay Adaption and Echo Cancellation. . . . 112
5.2 The Genetic Algorithm. . . . 122
5.3 The Cuckoo Search algorithm. . . . 124
5.4 The PSO algorithm. . . . 125
5.5 The CS/PSO algorithm. . . . 127

1. Introduction

Nowadays, sensors are increasingly deployed and their streams of significant data can be used to predict earthquakes [12], tsunamis [89], and panics [89]. Due to the miniaturization of these sensors and their decrease in energy consumption they will get even more deployed in many kinds of applications in the near future [21]. Consider, for instance, RFID-systems in warehouses [14], surveillance systems [128], and sports monitoring applications [129]. The on-the-fly analysis of those sensor streams can provide important high-level information in near real-time.

However, the massive amount of sensor stream data can only hardly be analyzed manually any longer. It must instead be processed automatically to draw valuable information in a reasonable amount of time. Systems for the analysis of streaming data are as manifold as their application requirements. Consider the previously mentioned RFID-based warehouse management system. The major focus of this kind of application is to provide a simple programming interface so that users, i.e., logistics specialists, can quickly implement detection rules by themselves. The sensor data rate is usually low and high notification rates play a minor role [94, 103]. A second kind of application is distributed processing on sensor nodes as part of a sensor network. As communication between sensor nodes requires considerable amounts of energy it must be avoided wherever it is possible, and as memory is only available sparsely [134] storing of data should also be avoided. Both of these applications aim towards a similar direction but differ significantly in their requirements, their workloads, and their desired response times.

In contrast, this thesis focuses on applications that demand high-performance, distributed event processing with low detection latencies. Consider, for instance, a Real-time Locating System (RTLS) that provides several thousand position sensor events per second. We can use such an RTLS to track players and balls in soccer games and extract meaningful events and situations[1] automatically [129]. The number of applications and clients using

[1] A situation is an event occurrence that might require a reaction. [57]

such a technology is sheer unlimited. If we manage to detect events such as ball possession, passes, and shots on the goal automatically within a few milliseconds, we can provide this information to the media, referees, camera systems, etc. Live feedback to a trainer can further help to directly survey qualitative performance data of his or her players [46].

1.1. Event-Based Systems

The massive data load of sensor systems is a challenge to the scalability of any system that is used to extract meaningful information. Hence, approaches such as event-based systems (EBS) recently gained much interest. The aim of event-based systems (EBSs) is to filter and aggregate events, i.e., special occurrences of interest, and to iteratively transform them into higher level events until they reach an information granularity that is appropriate for an end user application or for triggering some action.

Consider the system presented in Figure 1.1. It adapts the idea of the *Event Processing Technical Society* (EPTS) reference architecture [110]. Sensor data originates from the outside, and is injected into the EBS as raw events. Components (event detectors) take events from the event bus and generate new events that may be an input for another component. The system may

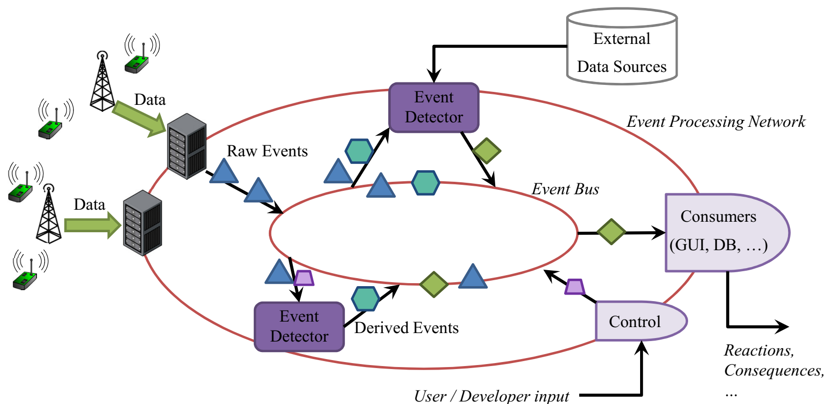


Figure 1.1.: EPTS reference architecture for event-based systems. [110]

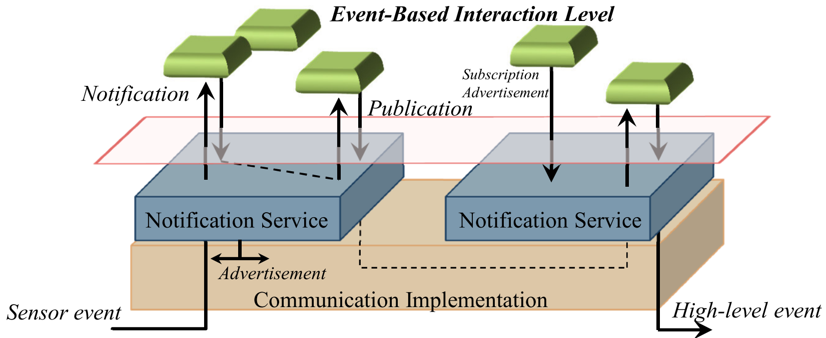


Figure 1.2.: Typical structure of an event-based system. [103]

also use external databases for information enrichment or user-defined input for reasoning. Results may be visualized on a graphical user interface (GUI) or used to automatically trigger some action. For instance, a camera surveillance system may automatically focus crucial points in an observed environment. This can help, for instance, security personell to survey critical areas and locations more efficiently [44].

Event-based systems provide extreme scalability and are a well-suited technique to analyze high data rate streams. Moreover, event-based components also exhibit advantages concerning the application's software complexity. Since there is (usually) only a low degree of coupling between software components, i.e., a message coupling [58], the software complexity tends to be low, making it both highly maintainable and flexible.

Figure 1.2 shows the interaction in a distributed event processing system (EPS). Usually event detectors (event-based components, producers and consumers)[2], i.e., the (green) boxes on top in Figure 1.2, are implemented either as black box units, i.e., for instance in programming languages such as Java or C/C++ [1, 75], or with database-like query languages [4, 10, 51, 87]. The former types make use of an API to communicate with other event detectors across the connected notification service. The latter types are usually directly loaded into the EPS without the necessity of being compiled previously. They

[2] In classic Data Stream Management Systems (DSMS) event detectors are also denoted as *continuous queries* or *operators*. [110]

are converted and processed at runtime within an extra unit embedded into the notification service [135].

Typical event-based systems build on a publish/subscribe model that implements the following basic functionalities (more detailed information can be found in Section 2.1).

Subscription. Subscriptions are initially issued by the event detectors (to the middleware) and by event-based middlewares (to the network) to express an intent to receive certain messages, i.e., events, in the future. Whenever subscribed events are generated, the event detector will receive these events for further processing.[3]

Notification. Notifications are used by the EPS to callback the event detector with a concrete data element, i.e., with an event, that is either received over the communication channel or by some other event detector that is connected to the same middleware. The physical source of the notification usually remains unknown to the event detector.

Publication. A publication is an initial issue of an event detector and signals a general provision of data elements, i.e., events of a particular type. Whenever the event detector detects an event of this type it generates an event, i.e., the computerized representation of the *real* event, and sends it to the EPS middleware.

Advertisement. Both the event detectors and the EBS use advertisements to disseminate the information on data provision. Whereas event detectors use advertisements to signal the event types that they provide, the event-based middleware aggregates all the advertisements from its event detectors and disseminates a single advertisement to the other event-based middlewares. Advertisements are optional components depending on the semantics of the EPS.

Although a common ground definition for event processing does not exist, see Section 2.1.1, the basic structure and functionality is the same in any event-based system. Event detectors work on an event-based interaction level, see Figure 1.2, subscribe to particular event types, i.e., event streams, process incoming events, and publish other events, visualize results, or trigger

[3] There are actually two meanings to the word *event*. It can either refer to the actual occurrence, i.e., something that really has happened, or the object that contains information about that happening in order to identify/refer it. We use *event* for both meanings as its actual meaning can always be derived from the context.

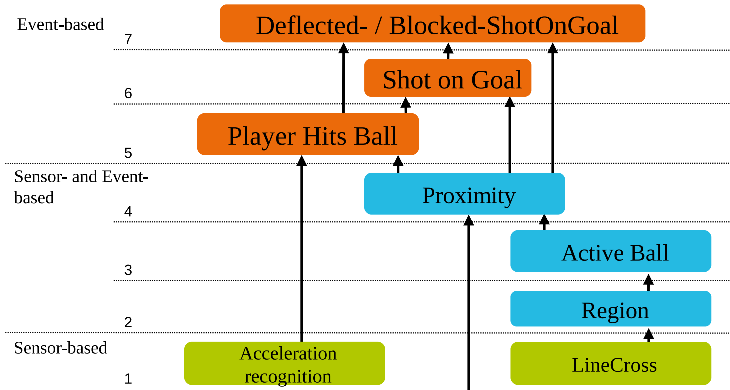


Figure 1.3.: Exemplified detection hierarchy for a shot on goal event.

some action as a result. Event-based systems are usually implemented on top of generic publish/subscribe-systems or -middlewares, whereas other implementations, such as libraries, are also possible. These systems or middlewares manage network and local communication (issues), data dissemination and integrity, routing, and much more depending on the requirements of the application domain [103].

As a result, event detectors do not need to care about system aspects, detector allocations, communication, or system loads. Complex detection tasks can easily be split up into several sub-tasks that are linked together to detect the top goal, i.e., the high-level event of interest. Consider the exemplified event detection hierarchy for a *deflected/blocked shot on goal* as shown in Figure 1.3. Event detectors at the lower levels receive and mostly filter out the high data rate position streams. Events are generated along the event hierarchy until the top event, i.e., a deflected shot on goal, is finally generated.

However, depending on the position data rate, the detection of proximity on Layer 4 may already consume all the processing power of an entire machine, no matter how efficiently it may have been implemented. Hence, the detection of events that rely on proximity (in ball sports, there exist dozens of them) becomes infeasible on a single machine and must instead be distributed over several machines. Since event-based system components are loosely coupled, and since the underlying middleware manages the commu-

nication, it is easily possible to form a highly scalable system by distributing event detectors across a network without changing the detector implementation itself [5]. Optimally the event detector's internal implementation does not need to take care about the physical location where other event detectors are running.

1.2. Research Objectives

Of course, the above mentioned powerful features of event-based systems only come as a trade-off between complexity and problematic challenges. Those include for instance, impact analysis for the implementation of change requests [113] or performance-related development issues [136]. But essentially, the distribution of event detectors over a network results in non-trivial synchronization problems of the particular detection units. Since partial results, i.e., events, are generated at different points in the system they are no longer received in a correct timely order at each event detector. However, event detectors usually detect interactions or merge events and assume a timely ordered event input.

Reordering of those events without considering latency in greater detail is naive: we just need to wait a sufficiently long time and postpone events until we can be confident (if we at least can) that they are in order. However, for most of the applications such assumptions are impractical because they introduce far too high detection delays. The solution is to configure reordering parameters to delay events manually and to provide an ordered event stream to the event detector.

Those parameters depend on the system load, the number of sensors or tracked transmitters, and the object behavior within the observed environment. Hence, they are hardly predictable, and if they are over-estimated they limit the detection latency, the throughput, and the system performance and scalability. The idea is to let the middleware system derive those parameters automatically at runtime. Thus, the reordering complexity is kept out of the event detector's implementation and the events are only delayed for a little amount of time.

However, known techniques that try to solve those issues exhibit a number of drawbacks. Since events experience different delays in the distributed environment (for instance there is a detection, a network, and a processing

delay) they must be dynamically reordered at runtime so that event detectors are provided with an ordered event input stream. If stateful event detectors process events out-of-order they may get stuck in invalid states and/or generate events that actually did not occur (false-positives), or do not detect events that actually occurred (false-negatives). We denote both of these cases as system failures because it can no longer be assured that the algorithms work as they have been expected to by the system engineers. System failures must hence be precluded at any time.

This dissertation addresses the problems of low-latency runtime event ordering without a-priori knowledge. To tackle the above mentioned issues, we can split up the challenge into a number of building blocks that can be solved separately. These building blocks are described as follows.

Self-Configuration of ordering parameters. To configure the ordering parameters of an event-based system and to reach a minimal latency we need to measure the particular event delays at runtime. Ordering buffers are then dynamically allocated, and the out-of-order events are reordered under-the-hood to provide a totally ordered event sub-stream to the respective black-boxed event detectors. In order to avoid incorrectly ordered events that are introduced by variations in the event stream delays we must use safety margins to trade detection latency with out-of-order events. Moreover, in order to avoid a recurring need to start delay measurements from scratch when restarting the system, we have to calibrate ordering parameters at system startup.

Utilizing system resources for latency reduction. With the ordering approach we usually buffer the events long enough to be confident that the event streams are properly in order. Additional CPU power and memory resources cannot be used to reduce detection latencies as the latency is bound to the buffer sizes. In contrast, speculative middlewares process any event speculatively, and retract events and revert states if speculation was wrong. This can exploit additional CPU power. However, speculation causes retraction cascades that inevitably lead to system overloads and hence system failures. The aim is to efficiently combine both techniques in order to utilize available system resources for speculation that remain unused when using conservative buffering middlewares.

Distributing event detectors over the network. System scalability is a consequence of the ability to distribute work across several nodes. But when we distribute event detectors and middlewares over a network there arise several new challenges. First, the allocation of event detectors over the network is crucial to gain performance benefits. If the allocation is poor the system may even perform worse. Second, efficient allocations may also become poor when CPU and event loads change. Event detectors must then be reallocated at runtime to fit the performance requirements of the application.
Such issues are bound to distributed environments and have (to the best of the author's knowledge) not sufficiently been tackled before.

1.3. Thesis Structure

The remaining chapters are organized as follows. Chapter 2 provides basic definitions and semantics, clarifies the disparities of definitions, provides details about the system architecture of our reference system, and surveys related work that is too general to be mentioned in one of the specific sections in the upcoming chapters but which is nevertheless important and relevant for the contributions presented in this dissertation. It also provides technical details of our evaluating system and the available sensor data.

Chapter 3 shows how to provide a timely ordered stream of events with low-latency event buffering to black-boxed event detectors. At the same time, the introduced latency is minimal in respect to the events' delays and no a-priori information must be provided. We formally prove the correctness of the algorithm and further give two methods to calibrate the system parameters at startup time in order not to start from scratch by reusing information from previous system runs. An evaluation with micro benchmarks proves that the methods we present work well on position data stream of a Real-time Locating System in a soccer application.

Chapter 4 describes an approach that exploits unused system resources by introducing speculation. The technique reduces latency of slack-based buffering approaches as those presented in Chapter 3. We show that buffering systems that only exhibit lazy CPU loads can be optimized so that latency reduces to only 20-30% of its original value. At the same time, the system stays fully functionable, and system failures caused by CPU or memory over-

load are avoided. To provide the best possible degree of speculation for the current event stream and system load the parameters are continuously calibrated at runtime. An evaluation shows that our method is very efficient and also compares more than well with other existing approaches that have been published.

Chapter 5 focuses on latency optimization when running event-based systems in a distributed context. Section 5.1 shows how to migrate stateful event detectors between nodes at runtime while the EPS is processing events. At the same time all event detectors continue processing and the ordering constraints are kept valid. Section 5.2 uses this migration to optimize the overall detection latency in a distributed environment by minimizing network delays. Micro benchmarks prove that our migration performs well compared to existing approaches and that our optimization adapts the detector allocation very well when event loads and latencies change at runtime.

Chapter 6 provides a high-level evaluation of our event processing system and the methods presented in this dissertation. We first analyze the results of a global competition where we called for innovative solutions for the analysis of our position data streams. We show that basic assumptions we made in the previous chapters really hold. Next, we compare the efficiency of an event hierarchy implementation that assumes unordered event streams with an implementation that uses our ordering approach. We take a closer look at detection latency, resource consumption, and code complexity and show that our approach keeps a nice balance between system performance and source code maintenance.

Chapter 7 sums up the main contributions of this dissertation, lines out similar applications and use-cases where the techniques and methods presented in this thesis can be used for, and provides directions for future research.

2. Preliminaries

This chapter provides fundamental information that builds the basis for the technical content we present in this dissertation. After introducing the specific definitions, semantics, and time model assumptions (Section 2.1), we present information about our middleware system (Section 2.2). We provide a brief high-level overview on related work (Section 2.3) where we mention and discuss related systems that cannot be focused on from their high-level point of view in the particular technical parts of this thesis. The chapter concludes with a detailed description of the Real-time Locating System (RTLS) and the provided sensor data streams that we later use for the evaluation of the presented system and methods.

2.1. Fundamental Definitions

We divide this section into two main parts. The first part focuses on general definitions of keywords to provide a common ground of understanding whereas the latter describes the time model assumptions used throughout this thesis.

2.1.1. General Definitions

Although we may repeatedly define some keywords that have already been used, we will provide a deeper and a more concise level of detail in the following.

Events and notifications. According to Etzion et al. [56] an event has one of three possible meanings: it is either (1) anything that happens or that is contemplated as a happening, (2) a state of change of anything, or (3) a somehow detectable condition that can trigger notifications. We stick to a general view and denote an event to be the notification of some

arbitrary state change of a component in an observing system [103].[1] In contrast, a notification can be considered as a specific instance of an event that may also carry some additional data and that is generated by a particular observing component, see Section 2.1.2. A description of the data elements that are attached to events can for example be modelled in a (semi-)structured fashion such as JSON or XML [103], or may also be omitted.

Event-based programming. The term *event-based programming* is used when one or more software components within a system process data in response to the reception of one or more event notifications. Sometimes, this is also denoted as event-driven architecture (EDA) [57].

Event detector. The software components implemented by application developers are denoted as event detectors. They act as producers and consumers of notifications and only interact with the event-based system middleware. Producers are components that publish notifications, i.e., that send (or emit) events. This is sometimes denoted as operator firing [116]. Consumers receive notifications delivered to them by the notification service and react appropriately. They are also unaware of their specific communication partners – they have no knowledge of their actual notification producers and only issue subscriptions to describe the kinds of notifications they are interested in [103].
Through the rest of this thesis we always assume that an event detector can act as both, a producer and a consumer, which is a generally known assumption [57, 110]. Then the event detector reacts to both incoming notifications and observed internal state changes, and the resulting computation may always lead to a newly published event notification [103]. Sometimes those components are also called *event processing agents* [57]. As a result, the event detectors are connected in a data-flow type graph (event detector hierarchy, operator graph) [116], with each detector receiving data from and forwarding results to other software components [5].
An event detector's implementation only accesses its own state and is completely encapsulated. In turn, its own state is not accessed by other

[1] As mentioned before, we use the term of *sending* an event when we mean the computerized representation of a *real* event as the reader may deduce the correct meaning from the context.

software components. The only way of interaction between event detectors is communication via events. The decision whether to publish a state change, i.e., to send an event, or not is made by the event detector's internal computation and is the core part of its functional implementation [103]. Typical types of event detectors include filtering, pattern detection, or event transformation [57].

Stateful event detector. Stateful event detectors hold some internal state to decide whether to publish an event notification or not. They can often be compared to stack state machines, i.e., finite state machines (FSM) or deterministic finite automata (DFA) with memory attached to it. Stateful event detectors are, for instance, deduplication filters that eliminate duplicate events from the event stream. In contrast, a conventional filter can often be implemented as a state-less event detector as it only applies static rules. An event detector can be considered stateful if the way it processes events is influenced by more than one input event [57].
But there are methods that assume no internal state as they put all the state information into the event, i.e., event-centric approaches [116]. However, it is often not convenient or even possible (deduplication filter) to do so. Hence, we always assume that an event detector has an internal state and that we do not have *direct* access to it from within the event processing middleware (as the only way of interaction with the detector is communication via events).

Subscriptions and advertisements. A subscription describes (a set of) notifications a consumer is interested in. Consumers register their interest usually at their own startup to the EBS middleware, see Section 2.2.3.[2]
Advertisements are periodically issued by producers to declare the type of notifications they are willing to send prospectively. We encapsulate that functionality by the middleware, which collects all the advertisements from the event detectors to disseminate a single advertisement message through the network. Similar to traditional software components the advertisements comprise the information about a component's output interface [103]. Neither the producer nor the consumer software

[2] However, this paradigm changes as real-time reconfiguration becomes a hot topic recently. Runtime and real-time reconfiguration and their issues are out of the scope of this thesis.

component need to care about advertising as this is achieved by the EBS middleware, see Section 2.2.

Event notification service. The event notification service is considered as the mediator in an event-based system and is used to decouple the producers from the consumers. The notification service alone is responsible for the transmission of notifications between the event detection components, and must deliver every event that has been published [103]. Event distribution in its simplest form may be achieved by a log file to which producers write to, and from which consumers read from [57]. The notification service is usually implemented as a publish/subscribe interface, which is described more clearly in Section 2.2.

CEP vs. EBS vs. EPS. Literature is riddled with different and contradicting definition of Complex Event Processing (CEP), Event-Based Systems (EBSs), and Event Processing Systems (EPSs). As some authors define CEP to be a combination of sequence operators and query languages that implement a data-flow paradigm [131, 135] others employ a more general view and define it to be the *processing of continuously arriving events with the goal of identifying meaningful patterns* [10, 114]. Interestingly, one of the earliest work in that area by Luckham et al. [94] even applies a more general definition of CEP and identifies it as *a set of tools and techniques that can be used to analyze and control the complex series of interrelated events*.
Throughout this thesis we stick to the latter CEP definition and use either of the terms equally to identify the combination of event processing middleware and event detection components.

2.1.2. Time Model Semantics

The time model we assume in this dissertation is that sensor events are time-stamped from the same discrete time source before they are sent to the network for processing. This requires synchronization of all system units that directly communicate with the sensors. However, this is not a great loss of generality because applications that require a low detection latency usually have the means to time-stamp sensor events when they are generated. For

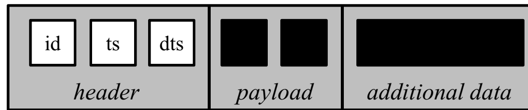


Figure 2.1.: The logical structure of an event instance [57].

instance, in warehouse applications, the RFID-readers may synchronize over LAN, time-stamp the sensor readings accordingly, and push the data packets as sensor events to the network. GPS-based systems may use the time stamps provided by the satellites [67]. In a Real-time Locating System the microwave signals of transmitters are extracted by several antenna units that are synchronized over fiber optic cables [129].

We use the following terminology throughout the rest of this thesis:

Event type, instance and time stamps. An event type is a specification for a set of event objects that have the same semantic intent and the same structure [57]. An event type defines an interesting occurrence and is identified by a unique ID. An event instance is an instantaneous occurrence of an event type at a distinct point in time.[3] An event detector receives an event instance as a notification. It can be a primitive (sensor) or a composite event. An event has three time stamps: an occurrence, a detection, and an arrival. All time stamps are in the same discrete time domain according to our applied time model. An event appears at its occurrence time stamp ts, or just time stamp for short. It is detected at its detection time stamp dts. At arrival time stamp ats the event is received by a particular EPS node. The occurrence and the detection time stamps are fixed for an event at any receiving node whereas the arrival time stamp may vary at different nodes in the network. Figure 2.1 denotes the logical structure of an event instance. The header information on the left side is fixed and is set for each event instance in particular. The payload information is set by the event detector that generates the event and is always part of event instances from the same type. The open content part (additional data) is a variable data part that may or may not be used by particular event instances.

[3] If events have a duration we set the occurrence time stamp to the beginning of the duration.

Out-of-order event. Consider a stream of events $e_1, e_2, \cdots, e_n$ that is sorted ascending to their arrival time stamps: $e_k.ats < e_{k+1}.ats$, $(1 \leq k < n)$. An event e_j is an out-of-order event if there is an event e_i with $1 \leq i < j \leq n$ and $e_i.ts > e_j.ts$. [87]. In other words, out-of-order events are events that occur before others but are received after them.
For instance, consider an event stream sorted by arrival time stamps *ats*: `A1` (*ts*=1, *ats*=2), `B4` (*ts*=4, *ats*=5), `C8` (*ts*=8, *ats*=8), `B7` (*ts*=7, *ats*=9), `A10` (*ts*=10, *ats*=11). Then, `B7` is an out-of-order event since it is received after `C8` but occurs before `C8`. Hence, the sequential semantics [116] in the event stream are no longer satisfied.

Event stream. The input of an event detector is a potentially infinite event stream that usually is a subset of all events, holds at least the event types of interest for that detector, and may include some irrelevant events as well.

Event lifetime. In most event-based systems (and especially those that deal with correct event ordering) there is some kind of notation concerning the lifetime of an event. The lifetime of an event is considered as the maximal time the event has an influence on window-based streaming operators such as multi-joins. The lifetime is often implicitly set by the maximum size of the applied window operators over the incoming event stream. Window operators are used to process a sliding part of the event stream as if it were static data. [94, 103]

System and wall clock time. In event-based systems wall-clock, system, and application times are usually different. In this thesis, we do not access the wall-clock time due to application semantics and performance-related reasons, see Chapters 3 and 6. Under this assumption, we derive the application-time of our system by deriving a local clock from the incoming event stream. Application time and system time are described by this local clock, whereas the wall-clock time remains both unknown and unused. We only use it to formally prove the correctness of our algorithms.

Pseudo event. Pseudo events carry the basic information of real events (ID and time stamps) but are not processed by the event detectors. The receiving detector, resp. the notification service, only uses such a pseudo event for configuration purposes, see Chapters 3, 4, and 5.

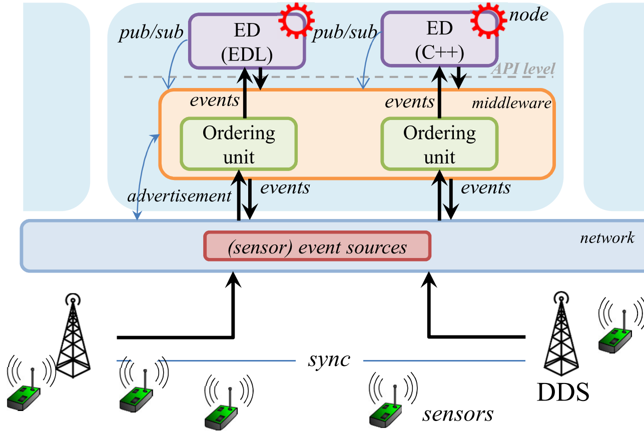


Figure 2.2.: Distributed publish/subscribe EPS.

2.2. System Architecture and Implementation

In this section we describe the distributed event processing system that has been used to implement the presented contributions from an architectural point of view. While Section 2.2.1 focuses on a high-level understanding about the system behavior, Section 2.2.2 digs into the details from an implementation point of view, before Section 2.2.3 describes the application programming interface (Event Detector API) for application developers.

2.2.1. Overview

Our runtime system is depicted in Figure 2.2. It consists of several data distribution services (DDS) that collect sensor data (for example an antenna that collects RFID readings), and several nodes in a network that run the same event processing middleware. On top there are event detectors spread over the network. An event detector and the middleware exchange subscriptions, publications, and necessary control information. The middleware does not have any knowledge about the complex event pattern that is implemented in the detector (that can either be implemented in a native programming language [75] or some EDL [135]), and the detector does not have any knowledge about

the distribution of other event detectors and the runtime configuration, i.e., they are completely decoupled.[4] At startup the middleware does not have any knowledge about event delays but just notifies other middleware instances about event publications and subscriptions (advertisement) [103]. For each detector the middleware also provides a personal ordering unit for the incoming event stream. The middleware dynamically adapts the buffer sizes of the ordering units by means of the methods described in Chapters 3 and 4. The middleware is therefore generic and encapsulated, and does not incorporate the application-specific complex event definition that is implemented in the detectors.

2.2.2. Implementation

Our event processing middleware is called *EventCore* and is implemented in about 17,000 lines of C++ code. On top of that event detectors are implemented in around 16,000 lines of C++ code. Our implementation makes use of various well-known concepts from generic publish/subscribe middleware systems [103] but also includes some application-specific optimization techniques. The middleware is used to take over any administrative tasks, comprises a notification service, and also schedules the event detectors for their processing of events.

Figure 2.3 shows coherences and control flow information in the EventCore. Dashed lines denote the process spaces of the interacting components. Whenever an event or raw event, i.e., sensor data, arrives at the middleware, it is immediately inserted into a lock-free ring-buffer[5] that is located in a global shared memory. In the literature this is also referred to as the *preparation task* [110]. The ring-buffer has a fixed size and exploits compare-and-swap mechanisms to avoid the usage of locking data structures. As a result, the buffer works on a best-effort basis: if the event data rate is too high the buffer cannot provide enough space for all data and the system will crash. However, this is not as big a disadvantage as it looks at first glance. No matter how large we choose the buffer – if the data rate is too high the system will eventually fail at some point as the event detectors are not able to detach events from the buffer fast enough – an early symptom of a system overload.

[4] Decoupled components do not depend on a particular processing or course of action being taken by another components [57].

[5] Our implementation is similar to the Disruptor pattern [126] that is available for Java.

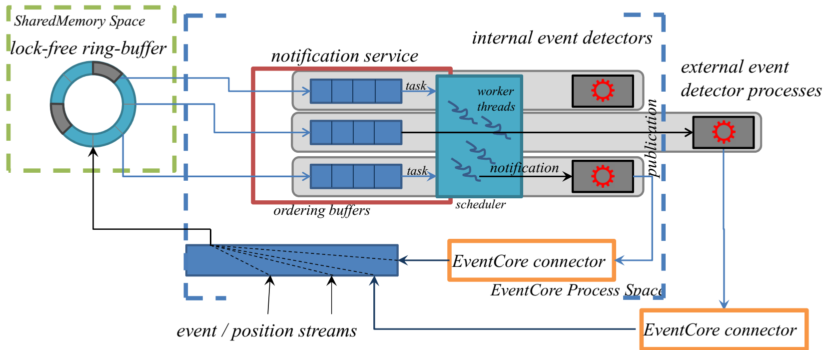


Figure 2.3.: The EventCore middleware architecture.

The buffer returns an index to the element as soon as the event has been successfully inserted into the memory. This index is used as a reference to the current event so that the event's data must only be stored once and can be referenced several times from the ordering buffers. The notification service sets up such an ordering buffer per event detector and fills its buffer with index tuples referencing the elements that are received (either locally by other event detectors or over the LAN). The notification service also synchronizes its local clock to the events' time stamps by means of the method presented in Chapter 3.

Whenever the notification service emits events from ordering buffers, it hands them over to a (dedicated) scheduling unit that has pre-allocated worker threads to process the derived tasks in a FIFO[6] order. These worker threads either (1) directly invoke the callback function implemented in the event detector (in case that the internal event detectors are directly linked to the middleware) or (2) sends the event's index from the ring-buffer to the event detector over a message passing interface (in case of external event detector processes). Whether event detectors run in the same process space of the middleware or not does not influence the detector's internal implementation as they simply implement an interface that is provided by the event detector API, see Section 2.2.3, and use a connecting element for communication with the runtime system. However, the big advantage of internal event detectors is

[6] First In First Out.

their performance. Since the detector is executed in the same process space the latency is considerably lower, task switches are omitted, control channels for inter-process communication are not necessary, and the middleware can take care of task scheduling itself. The event detectors we use in this thesis are always internally linked.

A last but important feature to mention is sequence numbering [110]. Sequence numbers are used to label the events in an incrementing fashion before they are sent to the network, separately for each event type. This is useful for a couple of reasons. First, depending on the network topology and the detector hierarchy events may be received twice. Sequence numbers help to identify and eliminate events that have already been received. Second, events are disseminated through the network over multicast as this significantly reduces transmission time and network load in contrast to other event dissemination methods like event brokers [103] or message-oriented middleware like the Java Message Service (JMS) [57]. Since multicast is based on UDP there is no guarantee that an event makes its way from the source to its destinations. If we want to establish a safe communication path sequence numbers can be used to identify pending acknowledgements and retransmitted events in multicast environments.

2.2.3. The Event Detector API

To illustrate the encapsulation of the event detector implementations and to motivate the problematic issues of incorrectly ordered event streams we briefly describe the application programming interface (API) provided by our EPS that can be used by the developers to set up a hierarchy of event detectors. Our API completely shields the event detector from the underlying protocols used by the middleware.

Figure 2.4 shows the example of an event detector as it can be implemented and used with the *EventCore* middleware. At instantiation the event detector signals its interest for event notifications on the three event types `EVENT_A`, `EVENT_B`, and `EVENT_C`, and also signals that it will generate events of type `EVENT_D`. This example and its detector implementation is also used in Chapters 3 and 4. The implementation of the `callback` function is the only mandatory one. This function is invoked whenever the middleware's notification service takes an event from the ordering buffer of the detector and emits it to the event detector, i.e., when a worker thread takes an event and calls

```
1  #include <eventcore/eventdetector/IEventDetector.hpp>
2
3  class SampleEventDetector
4    : public IEventDetector, public Serializable {
5    public:
6
7      EventDetectorSample(EventCoreConnector* _connector)
8        : connector(_connector), data(0) {
9        // initiate subscriptions; config. flags optional.
10       connector->listen(EVENT_A, EC_FLAG_UNSORTED);
11       connector->listen(EVENT_B);
12       connector->listen(EVENT_C, EC_FLAG_SECURE);
13
14       // signal publication.
15       connector->publish(EVENT_D);
16     }
17
18     // core implementation.
19     void callback (const Event& event) {
20       if (event.id == EVENT_A) {
21         data = new int[event.data.GetAttribute("size")];
22       } else if (event.id == EVENT_B) {
23         data[event.data.key] = event.data.value;
24         Event e2(EVENT_D, event.ts);
25         // send a new event to the middleware.
26         connector->Send(e2);
27       } else if (event.id == EVENT_C) {
28         delete[] data;
29       }
30     }
31
32     void serialize(Serializer& serializer) const {
33       serializer.serialize(data); // serialize state.
34     }
35
36     void deserialize(const Deserializer& deserializer) {
37       deserializer.deserialize(data); // deserialize state.
38     }
39
40   private:
41     int* data;
42 };
```

Figure 2.4.: Sample code for an event detector.

the event detector's callback function. The core functionality is implemented in/from here. The `serialize` and `deserialize` functions are optional and are only required if the `Serializable` interface is implemented by the detector. The implementation of these functions is mandatory to achieve the functionality that is used in Chapter 4 where event detectors are required to provide a snapshot and to restore from a snapshot on demand.

However, this small event detector sample already causes some problems if naive assumptions are specified. Consider the application code implemented in the callback function in more detail. On reception of event `EVENT_A` we initialize `data` as an array of integers. The size of the array is provided as additional information in the notified event. Whenever `EVENT_B` is received we write some data provided by the event into the internal state `data` and send an event afterwards. As soon as we receive `EVENT_C` we purge the internal state data.

This example code works correctly if the causal order of the events is always correct, i.e., if we can be sure that `EVENT_A` inevitably leads to a later `EVENT_C`, but with the possibility that some optional event `EVENT_B` happens in between them. We provide examples in the upcoming chapters.

A problem occurs when `EVENT_B` is received without having received an event `EVENT_A` before. However, although the application's sequential assumptions may be correct it may happen that the event detector receives the events out-of-order. The event detector's application code results in a segmentation fault as a result of memory corruption, and the system fails afterwards.

Implementing the application code in a more robust and fail-save way sounds good at first glance but results in high safety work overhead in its development. In Chapter 3 we show how to provide correctly ordered input event streams to the event detector by withholding the early and late events from the detector as long as necessary. The application developer does not need to care about consistency issues raised by incorrect event order. At the same time latency is kept minimal. In Section 6.2 we also show that event detectors that themselves care about event ordering are highly complex.

2.3. Related Work

We will present a detailed discussion of the related work for each technical core component of this dissertation in the upcoming chapters. How-

ever, at these points we cannot focus on available systems from a high-level point of view or consider these systems in a larger focus and application domain. There exist a number of academic systems like StreaMIT [82, 125], STREAM [11], TelegraphCQ [33, 96], Cayuga [27, 28, 53, 54], Aurora [142], Medusa [142], SASE [3, 87, 135], Sienna [30, 77], and Hermes [111] among several others. Moreover, stream processing has meanwhile also emerged in some commercial systems including SAP Sybase [9, 124], TIBCO StreamBase [117], and Software AG Apama [2] (recently sold by Progress Software Inc.). In this section we list systems that try to address similar applications or technical challenges as we tackle in this thesis. We particularly focus on the Borealis Stream Processing Engine (Section 2.3.1), Esper (Section 2.3.2), and IBM InfoSphere/System S (Section 2.3.3).

2.3.1. The Borealis Stream Processing Engine

The Borealis Stream Processing Engine [1] is the only other EBS we know that also combines all the solutions we consider in this thesis: stream revision processing, (partial) event ordering, and distributed task optimization.

Its stream revisioning by Cerniack et al. [34] uses so-called time-travels and re-processes the events that are buffered when revisions occur. However, for predominant out-of-order events almost all generated events must be revised, which often triggers a cascade of revisions along the detector hierarchy, see Chapter 4. As this puts pressure on the memory management, Borealis limits the memory for revision processing and drops packets that cannot be revised to implement an inherent load shedding. Though, as we have seen in Section 2.2.3 the corrupt states (as a result of missing events) can make the event detectors and hence the entire system fail. Borealis uses (partial) ordering in a way that events must be sent from a distinct *send* task that is used to send events to the next downstream SPEs (stream processing engines) in order, which however, may not guarantee an in-order arrival of all events at the downstream SPEs because it may also receive events from other sources. In contrast, we dynamically determine the minimal buffer sizes that avoid such partial ordering failures and deep revision cascades. For the details of our out-of-order event processing and a comparison to specific related work, see Chapters 3 and 4.

Whereas the Borealis box-sliding operator migration [34] can only move event detectors to the lower level event detector's node or to the higher level

event detector's node (because it is not order-preserving), we present a technique that can migrate event detectors between arbitrary nodes, independent from the detector hierarchy. More details and a more thorough discussion of related work in the area of migration can be found in Section 5.1.2.

Xing et al. [139] add dynamic load distribution to Borealis. Their greedy approach migrates filter operators pair-wise and minimizes end-to-end latency by lowering load variance and by increasing load correlations. They periodically collect CPU load statistics and either a one- or two-way algorithm migrates the operators. Whereas their greedy approach is likely to just optimize towards a local optimum and does not consider network latencies, our technique migrates several event detectors at a time and obtains a significant benefit in performance that is closer to the global optimum, see Section 5.2.

2.3.2. Esper

Esper [22, 70] is a Java-based complex event processing system that has been introduced in 2007 and that is still under active development. Esper provides a complex event processing engine with a powerful and native interface. Queries can be split up to form sub-queries that can build on each other to detect a high-level event. The Esper Processing Language (EPL) features basic operators such as windows, contextual partitioning, aggregation, and expressions. An interesting feature of Esper is that it not only processes queries implemented in EPL but also supports seamless integration of Java-defined queries that can be induced into the CEP engine. Hence, the capabilities of Esper are not limited by the expressiveness of its query language: the high-level EPL provides fast development of queries while the low-level Java implementation provides the needed flexibility. Existing work also shows that Esper is a suitable framework for the implementation of sensor stream data processing systems [31, 60, 71], see Section 6.1 for more details.

However, Esper not only comes with basic abilities for distributed out-of-order processing. Order preserving delivery is either achieved by timeouts or by spin-locking among the event listeners. The former is naive as the event latencies must be defined a-priori, which results in high detection latencies. The latter is more sophisticated but (1) does not work for negative patterns (see Section 3.2.1), and (2) does no longer work efficiently when event detection is distributed among several nodes as we must implement locks at inter-node level.

2.3.3. IBM InfoSphere Platform

IBM's InfoSphere Platform [23, 69], formerly developed as System S [72], is a large-scale distributed complex event processing (CEP) platform combined with its Stream Processing Language (SPL) [66], formerly known as SPADE [63, 123]. System S has first been described by IBM in 2005 [8]. The InfosSphere middleware comprises a Stream Processing Core (SPC) [8, 72] as its runtime that uses queries specified by SPL as the front-end for the data-flow specification. The InfoSphere runtime is commonly deployed on a cluster of compute nodes and can scale to thousands of nodes [23, 69]. The runtime provides an abstraction of processing elements (PEs) connected to each other via data streams, and manages the distribution of the PEs across the cluster. InfoSphere also includes solutions for operator scheduling [109], failure recovery [130], etc.

The Stream Processing Language (SPL) fills the gap between the high level development language and the low-level data flow support provided by the middleware's runtime. SPL is compiled and linked together with the data flow description that is then loaded into the runtime [116]. SPL supports data types, arbitrary user-defined logic, and stateful operators. A recently developed method named Fission [116, 119] is used together with operator profiling to decide on the operator placement in the network.

InfoSphere also considers out-of-order event arrival. The event ordering that is used in InfoSphere is based on pulses [116] that are equivalent to punctuations [86, 127]. However, especially for negative event patterns, as we will show in Chapter 3, such a mechanism is highly prohibited due to performance related issues, see Section 3.6.

2.4. Reference system details

This section gives a detailed overview of our evaluating system. Section 2.4.1 provides an appropriate understanding of the RedFIR locating system [129] from a computer scientist's point of view, while Section 2.4.2 gives detailed information about the data that is provided by the RedFIR system in real-time.

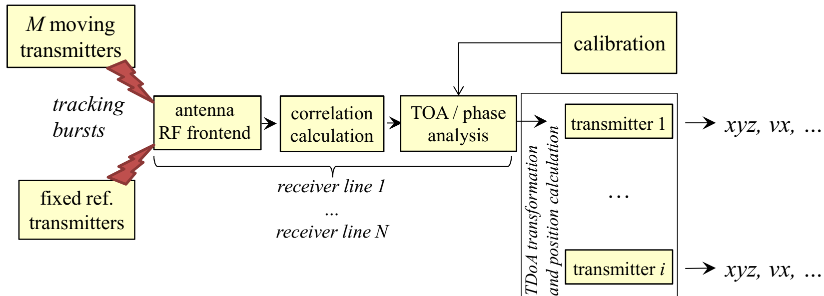


Figure 2.5.: The signal processing chain of the RedFIR system.

2.4.1. The RedFIR Real-time Locating System

The RedFIR tracking system is a Real-Time Locating System (RTLS) based on time-of-flight measurements, where small transmitter integrated circuits (ICs) emit burst signals [129].

A centralized unit processes these microwave signals and extracts time of arrival (ToA) values. ToA values are the basis for time difference of arrival (TDoA) values, from which x, y, and z coordinates are derived. In the following, we describe the technical aspects of the RedFIR system installed in the main soccer stadium in Nuremberg, Germany, where most of the data used in this thesis originates from.

The RedFIR system operates in the globally license-free ISM (industrial, scientific, and medical) band of 2.4 GHz and allows a usage of around 80 MHz. Miniaturized transmitters (61×38×9 mm) use this available bandwidth to generate short broadband signal bursts together with identification sequences. The RedFIR locating system is able to receive an overall of 50,000 of those tracking signal bursts per second.

Figure 2.5 illustrates the signal processing chain of the RedFIR system. The system distinguishes fixed reference transmitters for configuration purposes, i.e., six transmitters located at the corners and the lateral ends of the midline, from M moving transmitters, i.e., the balls[7] or the mobile transmit-

[7] RedFIR tracks up to 12 balls simultaneously.

Figure 2.6.: A glass model of the ball's transmitter and inductive charger.

ters carried by the players. All transmitters emit tracking burst signals, which are received by N receiving antennas that are located on top of the flood light poles and the roofs of the stands. The installation in the soccer stadium in Nuremberg provides 12 antennas that receive signals from up to 144 different transmitters. The balls emit around 2,000 tracking bursts per second whereas the remaining transmitters emit around 200 tracking bursts per second.

For each of the 12 receiver lines, dedicated FPGAs correlate the individual burst sequences, and a CPU analyzes the resulting ToA and signal phase measurements. The receivers are synchronized so that the RedFIR system can determine time difference of arrival (TDoA) values out of the ToA values for each particular tracking burst between pairs of receivers. This allows the RedFIR system to avoid taking the time of transmission into account and thus alleviates the need to synchronize the transmitters. The RedFIR system can calculate the positions using only the timing information from the receivers. With the given number of receiving units N=12, the system derives N-1=11 TDoA-value streams.

Burst signals can be reflected by metallic surfaces as well as the playing field. This means that the receiving antennas might receive a given signal on multiple paths, which might negatively impact the quality of measurements. To avoid using the wrong propagation path an unscented Kalman filter (UKF) monitors several propagation paths at the same time and chooses a reliable line-of-sight (LOS) path for the tracking signal. Hence, at any time the most likely propagation path is used for positioning and ToA calculation [104].

In a final step the TDoA streams and the carrier phase signals serve as inputs to extended Kalman filters (EKF), i.e, one per transmitter, to calculate the x, y, and z coordinates as well as other spatial information, see Section 2.4.2. The EKF uses hyperbolic positioning in order to calculate intersection points between measurements. The position information is further improved by using measurements of the carrier phase signal together with the ToA measurements – this allows to increase the accuracy for measurements of relative movements.

The miniature transmitters themselves are splash-proof (in case of player transmitters) or integrated into the soccer ball. Hence, they are charged by an inductive principle. Figure 2.6 shows the charging cradle for a ball (charging units for the small transmitters are only a few centimeters long) and the suspension of the transmitter within the ball. Both ball and player transmitters can be fully charged within a few hours. The batteries run for over 3 hours,

i.e., more than enough for a game with interruptions, stoppage time, and extra time. Moreover, the transmitter are application-specific integrated circuits (ASIC) and can be used as arbitrary waveform generators (AWG) operating at 2.4 GHz, i.e., they are re-programmable.

2.4.2. The Position Data Streams

Figure 2.7 gives an impression on the data set that is produced by the RedFIR locating system. The data streams for the individual object transmitters are all embedded in a single network stream. Each position packet usually comprises ten subsequent position measurements. Section 3.4.1 provides further information on the position packaging.

The position in the three axis are resoluted in millimeters, with (0,0,0) being the midpoint of the game field. The values of the velocities and accelerations are normed by their amount and need to be transformed prior to any processing, see [49] for more details.

Usually each player is equipped with at least two sensors – one per each foot, located in the near of the shinguard. However, besides the obvious locations like hands and gloves, further sensor positions like the head or the back are possible for deeper reasoning and pattern recognition.

2.5. Summary

This chapter provided basic information and definitions that are important to understand the technical contents of this dissertation. We first defined the most important terms and defined the time model semantics in more detail. We also illustrated our runtime system in which we implemented the technical features we describe in the upcoming chapters. We finally covered the related work in the field of streaming systems which provides similar functionality as our runtime system and concluded with a technical description of our reference sensor system and its streaming data.

```
id|      time        |   position      |abs. vel./acc. | vel. comp.   | acc. comp.   |QoL...|
---------------------------------------------------------------------------------------------
                     |   x     y     z |  |v|     |a|  | vx   vy   vz | ax   ay   az |qx ...|
                     |                 |               |              |              |      |
11 23720517802523129 -15566 -3827 -930 2030696 8555312 7367 6616 1392 6290 7773   -6 1  ...
11 23720522924485922 -15560 -3817 -929 2075876 8630651 7236 6765 1366 5749 8181  -24 1  ...
11 23720528046448437 -15556 -3819 -927 1912220 5633798 7477 6515 1278 6033 7968 -322 1  ...
11 23720533168410881 -15551 -3816 -927 1872036 4711167 7540 6464 1162 5971 7992 -677 1  ...
11 23720538290373275 -15546 -3808 -918 1865926 4286281 7356 6625 1404 4521 8917 -194 1  ...
11 23720543412335262 -15542 -3803 -922 1854093 3918456 7295 6688 1430 3720 9281 -109 1  ...
11 23720548534297582 -15538 -3797 -925 1850818 3417717 7444 6507 1494 5458 8376  199 1  ...
11 23720553656260069 -15532 -3789 -921 1889021 3612784 7450 6457 1670 6326 7701  813 1  ...
      .
      .
      .
```

Figure 2.7.: The position stream data provided by the RedFIR locating system.

3. Dynamic K-slack Buffering

Event-based systems (EBSs) are used to detect and analyze meaningful events in surveillance, sports, finances and many other areas. As we described before, with rising data and event rates and with correlations among these events, sequential event processing becomes infeasible and needs to be distributed. However, existing approaches cannot deal with the ubiquity of out-of-order event arrival that is introduced by network (and other types of) delays when distributing EBS, and as we have seen, an order-less processing of events may result in a system failure and must hence be avoided in any case.

This chapter presents a low-latency approach based on K-slack [135] that achieves ordered event processing on high data rate sensor and event streams without a-priori knowledge. Incoming events are no longer instantly processed but inserted into a buffer. These *slack buffers* are dynamically adjusted in their sizes to fit the disorder in the streams without using local or global clocks. The middleware transparently reorders the event input streams so that events can still be aggregated and processed to a granularity that satisfies the demands of the application. On a Real-time Locating System (RTLS) our system performs accurate low-latency event detection under the predominance of out-of-order event arrival and with a close to linear performance scale-up when the system is distributed over several threads and machines.

3.1. Event Ordering in Streaming Systems

As mentioned before EBSs provide a scalable and powerful technique to process high data rate sensor streams [122]. But due to performance related issues the processing of those events is no longer possible on a single machine. The solution is to distribute event algorithms (event detectors), across several machines. However, it is not as simple to distribute event detectors beyond one machine. Events are generated at different points in the network

and are no longer timely synchronized. If the sensor data rate is high, out-of-order events become predominant. As a consequence, naively distributed event detectors process events incorrectly and generate wrong results.

Eliminating these ordering issues by finding an optimal distribution of event detectors over the available nodes is not always possible. Moreover, it would not solve the problem because of application-specific delay types, such as back-setting delays. Back-setting delays occur when events are generated with time stamps that are earlier than the time stamps of the events that cause them. Consider again the event detector hierarchy from Figure 1.3. For instance, to detect the shot on goal in Layer 6, the angle of the shot must be right. However, we only know the angle after a few more position events have been received and processed. Another example is the filtering of values. Consider that we process sensor events provided by a noisy thermometer. To generate a smoothly filtered temperature value per time unit we need prospective sensor readings to balance the amplitudes between the measurements correctly. Thus, both events have a back-setting delay and can only be inserted into the event stream long after they have actually happened.

We therefore focus on the distributed derivation of optimal K-values for the K-slack algorithm. K-slack transparently buffers and reorders events before they are processed by event detectors.

Unfortunately, existing work is insufficient because it either lacks support for distributed processing or for correct event ordering. Established methods also fail for negative patterns, i.e., waiting for certain events *not* to occur in between other events, since they assume fixed buffer sizes. Such patterns are also denoted as *absence-patterns* or *non-events* [57]. Timing offsets need to be measured dynamically in relation to dedicated event sets in order to enable an in-order processing of events.

The remainder of this chapter is organized as follows. We motivate the problem in Section 3.2.1 and introduce K-slack in Section 3.2.2 before we present the details of our novel runtime ordering solution:

- Section 3.2.3 formalizes the problem and proves that the original K-slack approach does not work for hierarchical event detectors in a distributed system.

- Section 3.3 shows how to make K-slack work by ordering event streams with dynamically adjusting slack buffers under the absence of a global and local clock. Our middleware solution does not need a-priori knowl-

edge, and interesting events are still generated with low latency such that, for example cameras, can take immediate action on the live output of the system. At the same time, the application developer does not need to specify delays as most delays either depend on the number of event streams or the object behavior so that they can hardly be estimated before runtime anyway. If they are estimated, overall detection latency may be too high. For instance, an offside in soccer must be detected as quickly as possible to blow the whistle. Also our solution does not restrict the detector implementation since the middleware reorders the event streams transparently.

- We formally prove the correctness of our dynamically adjusting K-slack approach by a mapping to wall-clock times in Section 3.3.5.
- Since initialization parameters of the algorithm at system startup cannot be defined a-priori, Section 3.5 describes two methods to properly estimate them at runtime.

Section 3.4 evaluates our method under real-life conditions when it is used to analyze position stream data from a Real-time Locating System (RTLS) in a sports application. Soccer events suffer from different types of delays, for which we discuss the influence on both the system stability and the correctness of the system output. We also show, that our system can handle massive out-of-order event arrival. Slack buffers for event ordering are dynamically (re-)sized and are optimal in respect to detection latency. Section 3.6 provides a discussion on related work before Section 3.7 closes this chapter.

3.2. Slack-based Event Buffers

Event detectors often assume a total order on the incoming event stream, which means that they rely on the fact that the order in which they receive events reflects the events' time stamps. However, in practice and in distributed event processing environments, out-of-order events are predominant because of two reasons. First, machine or partial network failures or intermediate services such as routers or translators may introduce delays. Second, the workloads of the processors that run event detectors vary.

It is difficult and error-prone to implement event detectors that can process out-of-order events. Especially as developers often have no clue about the

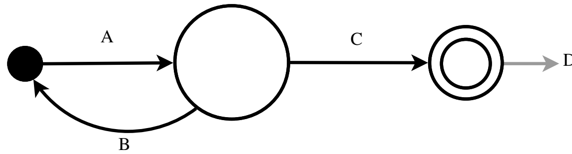


Figure 3.1.: Example for A!BC.

timing delays their code may face at runtime. Systems that provide a high-level Event Definition Language (EDL) to manage out-of-order events [59] often restrict the expressiveness of event definition, see Section 6.1 for details. In contrast, our solution implements a hybrid approach: expressive event detectors can still be implemented in a native programming language with the assumption of ordered events because the middleware transparently reorders out-of-order events under the hood. Hence, the middleware does not have knowledge of the patterns that are implemented in the detectors.

3.2.1. Motivation

Without an ordering unit, event processing may fail. Consider the following example. To detect that a player kicked a ball, we wait for the events that a ball is near the player and then, that the ball is kicked, i.e., a peak in acceleration. Between the two events there may not be the event that the ball leaves the player, because in that case the ball would just have dropped to the ground. More formally: if we receive event A (near) and subsequently C (acceleration peak) and not B (not near) in between, we generate event D. A very similar application is the book shelf-reading example that is often used in RFID-based complex event processing (CEP) [87]: a book is stolen from a bookstore, if it is taken out of the shelf (A), and afterwards passing the sensed exit (B) before it is put back into the shelf (C). Figure 3.1 gives a finite state automaton for this, i.e., event D. To simplify, we leave out the differentiation of transmitter IDs for player identification, resp. the unique numbers that identify different books. This event detector is similar to the sample implementation provided in Section 2.2.3.

In the first part of the event stream in Figure 3.2 the events C with time stamp 1 (C1 for short) and B3 are received too late. With the occurrence time

stamp 1, the in-order placement of C1 would be between A0 and A2, and with the occurrence time stamp 3, the in-order placement of B3 would be between A2 and A4, but the arrival time order is different. A detector that ignores ordering, incorrectly detects D out of A2/C1 and cannot detect D out of A4/C5.

Certainly this problem only arises when events are merged. However, this is necessary because we split computation across several event detectors and must iteratively summarize preliminary results (events).

Unfortunately, in general it is impossible to distribute a set of event detectors so that events are always received and processed in correct order without buffering because of two reasons. First, publication and subscription dependencies between event detectors in the processing hierarchy cannot always be mapped onto a networked structure. The hierarchy is a special case/instance of a non-planar graph. Second, even if such a mapping is possible there are delays that are application-specific and introduced by the event detectors themselves (back-setting delays) so that reordering would still be necessary.

In Section 3.4 we show detailed measurements of event delays from the detector in Figure 3.1. A and B are delayed up to 180ms whereas C has only 5ms delay. Without ordering these events, the detector cannot always work correctly.

3.2.2. The K-Slack Approach

The well-known K-slack algorithm [15] deals with out-of-order events. It uses a buffer of length K to delay an event e_i for at most K time units (K must be known a-priori). K-slack has shown significant reduction of the runtime state (the number of buffered data elements) when executing queries over streams. Although K-slack has been developed for non-distributed and single-threaded streaming applications it can be used in distributed environments. Given some local clock clk, Li et al. [87] buffer an event e_i at least as

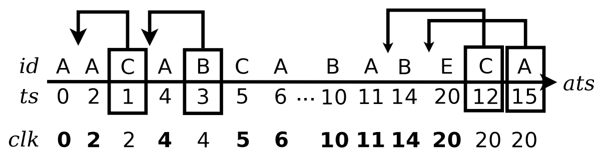


Figure 3.2.: Out-of-order event stream.

long as $e_i.ts + K \leq clk$. As there is no global clock in a distributed reactive system, each node synchronizes its local clock according to *the largest time stamp seen so far* on any incoming event, see the *clk*-line in Figure 3.2. Bold numbers indicate an update of *clk*. The rectangles show events that are out of order and the arrows point to their appropriate positions in the event stream. Note, that the right hand side of the above inequation (also) depends on the stream of incoming events e_i.

Recall that the example event detector for `D` builds on the events `A`, `!B`, and `C`. K-slack waits for K time units before generating event `D`, because only then there cannot be a late event `B`. For the first part of the event stream in Figure 3.2, an a-priori value of $K = 3$ works. Event `A` does not have a delay (its time stamp is equal to *clk*, `A`'s delay is 0). The first `C1` event arrives while *clk* is 2 but before *clk* is set to 4. Its delay is at most $clk\text{-}ts$=4-1=3$\leq K$. `B3` arrives while *clk* is in $[4;5]$. Hence, $K = 3$ also suffices with `B`'s maximal delay of $clk\text{-}ts$=5-3=2$\leq K$.

Figure 3.3 shows the internal semantics of the ordering unit for the given event stream from Figure 3.2. The current local clock *clk* is depicted on top. Bold values indicate updates of *clk* whereas grey values are intermediate clock values that are skipped as no event is received with that time stamp. Particular events arrive (depending on their own delay) later than other events, see the small diamonds. Hence, the early arriving events are delayed appropriately, see the straight (red) lines. These straight (red) lines end as soon as the K-slack inequation holds. This is depicted with the big diamonds. In the given example `A0`, `A2`, and `A4` could have been emitted at time 3, 5, and 7, i.e., one time unit earlier than K-slack emits them. But since the ordering unit has no knowledge about the intermediate clock values it has to buffer the events longer, see the dotted (blue) lines. Then, the dashed (blue) arrows pointing to the bottom depict the emission of the event to the event detector. Hence, the lower an event's (natural) delay is, the longer K-slack needs to delay it additionally in order to compensate the delay differences among the different event types.

The ordering unit in Figure 3.4 implements the K-slack approach with $K_D = 3$ for the given event stream. It applies a sliding window to the input stream, delays the events according to their time stamps, and produces an ordered output stream of events. The dark-grey area shows the events that are emitted when updating *clk* to 7, the brighter-grey area shows the events that remain in the buffer. The size of the sliding buffer is not defined by the num-

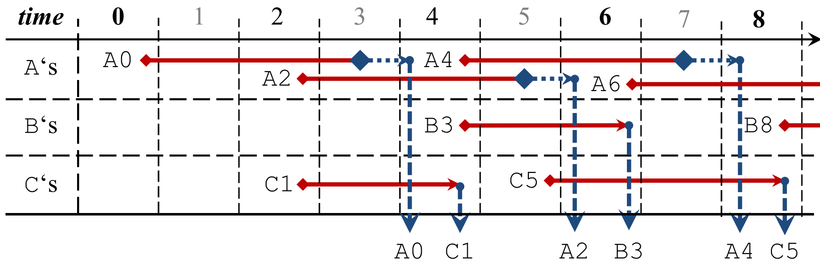


Figure 3.3.: K-slack on an event stream.

ber of events but varies with both *clk* and the event time stamps. With K-slack we can also deal with back-setting delays because it is simply modeled as a part of an event's overall delay. The event is ordered as if it would have a real delay.

3.2.3. Problem Definition

A single fixed a-priori K does not work for distributed, hierarchical event detectors. K-slack takes K time units to generate `D` out of `A`/`C` as the ordering unit has to wait at least until clk=`C`.$ts+K$ before `C` can be emitted to the detector. The detector then may set `D`.ts:=`C`.ts, i.e., `D` takes the occurrence time stamp of `C`. An event detector on a higher layer that waits for `D` and that only buffers for K time units, may miss `D` because `D`.$ats \geq$`D`.$dts \geq$`D`.$ts+K$. Waiting times (must) add up along the detector hierarchy.

An individual K-value per event detector solves the problem. Such a K_n must at least be set to a value larger than $max(K_{n-1})$, that is larger than the maximal delay of all the subscribed events (because then each detector buffers

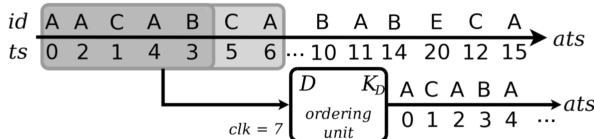


Figure 3.4.: Sorting window over event stream.

its events longer than the lower levels delay their events). If all K_n are then sufficiently large, K-slack works properly and provides sorted event streams. Although this sounds good at first glance, conservative and overly large K-values result in large buffers and therefore long latencies for hierarchical event processing, and must be avoided for the types of applications that we target.

Hence our aim is to find K-values that are as small as possible, but as large as necessary. This is both difficult and application- and topology-specific. Since we have no a-priori knowledge of any of them, event delays and hence K-values, can only be found by runtime measurements. Recall the basic inequation of K-slack: $e.ts + K \leq clk$. In contrast to the original K-slack approach we not only need to derive *clk* from the event stream but we also need to extract K from the event stream. However, since K now also depends on *clk* as we have discussed in Section 3.2.2 there arise two problems.

1. **clk grows unexpectedly** ⇒ a previously determined K-value is too small. Consider Figure 3.2 again. On reception of `E20`, *clk* is set to 20. With K_D calculated from the previous value of *clk*, the event sequence `A11/B14/C12` does not make the event detector generate `D` because at *clk*=20 `C12` arrives too late for the current buffer size. `A11` and `B14` are emitted with reception of `E20` and then `C12` is processed out-of-order.

2. **K grows unexpectedly (or is still unknown, 0)** ⇒ a previously determined K-value is too small. In Figure 3.2 the maximal delay of `C` is set to 3=4-1 as soon as `A4` arrives. The maximal delay of `B` is 2=5-3. Hence, for `D` the suitable K is the maximum of the worst delays of all its input events, namely $K_D = \max(0, 2, 3) = 3$. Assume now that between `A6` and `B10` another `B` with time stamp 8, and between `B10` and `A11` another `C` with time stamp 7 arrives (that is later than it was expected to arrive). Then we would retrofit the value of K_D to 11-7=4. Although the detector should have fired for `A!BC` (time stamps 6-7), it did not generate the `D` because at that time, the old value of K=3 has still been in place and `A6/B8` have already been processed.

3.3. Self-Adaptive Ordering Units

We first describe the steps for correct derivation of both *clk* and *K*, then show how to better estimate *K*, before we present the algorithm in pseudo code, give an example, and prove the correctness of the algorithm.

3.3.1. Derivation of *clk*

The problem of unexpected *clk* changes can be fixed by no longer setting the clock to the largest time stamp seen so far *on any incoming event*, but to only use one or more designated event types for setting it. While the above definition of out-of-order events has only identified events that are late, we now also use the *clk*-values to postpone events that arrive too early. This brings us to a new definition of out-of-order events.

Out-of-order event. Consider an event stream e_1, e_2, $\cdots$ e_n as before. Assume that *clk* is only set by events of type ID. Then event e_j is ***out-of-order*** if there do not exist e_i, e_k, with $e_i.id{=}e_k.id{=}$ID and $e_i.ats \leq e_j.ats$ so that $e_i.ts{\leq}e_j.ts{\leq}e_k.ts$ from now on.

In Figure 3.2 an unexpected change of *clk* can be avoided if the clock is set only on incoming events of, for instance, type `A`. See the clk_A line in Figure 3.5. Straight rectangles and arrows depict late events and their appropriate position in the event stream while the dashed rectangle and arrow depict an early event and its appropriate position in the event stream. `C12` arrives while clk_A is still 11 and fits for the current size of *K*. At this point, it is irrelevant if the picked *clk*-setting event type is used by the event detector because it is only used to adjust the right hand side of the above inequation, that is *clk* in $e_i.ts{+}K{\leq}clk$. `B`, `C` or some other event type that is embedded in the stream

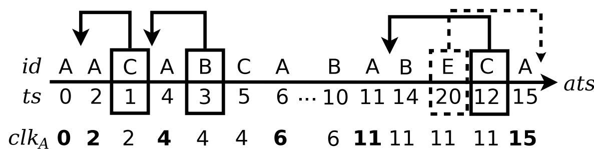


Figure 3.5.: Out-of-order event stream with clk_A.

work equally well. By avoiding sudden changes of K, early events are postponed until K has been updated and will no longer make the detector fail.

The remaining question is which event type to pick for setting *clk*. The higher the occurrence frequency of the picked event type is, the fewer events need to be postponed, the smaller are the resulting K-values, and the better are the measured delays. If there is a choice of event types, the one with the more stable and fixed delay is preferable, because it better reflects the real time. If otherwise the events of a certain type vary in their arrival times, *clk* does not behave smoothly. For better clock update frequency, instead of using just one event type, it is possible to use a set of event types for *clk*-setting, provided that those event types have the same absolute delays. It is not sufficient to pick a set of event types that form an ordered sub-stream, see Section 3.3.3. In Section 3.3.6 we hint how it is possible to set *clk* by events with different delays.

3.3.2. Derivation of K

With a given stream of incoming events e_i and a sufficiently stable *clk*, the key idea is to perform the runtime delay measurements by comparing an event's occurrence time stamp with its arrival time stamp, and to use this knowledge about disorder to derive a suitable K. Recall that for an event detector d the proper K_d is calculated as the maximal event delay of all subscribed events: $K_d = max_j \left[\delta(e_j)\right]$. The maximal delay of an event is $\delta(e_j)$=$e_k.ts$-$e_j.ts$, where e_k is the next event that updates *clk*.[1]

However, the above mentioned problem of a suddenly increasing K is still open. If K is too small, we miss events or process them out-of-order, and only afterwards increase K to be suitable for future event delays. There are two counter-measures. First, we can avoid detection errors due to rare sudden increases of K with an added safety margin, that is a slightly larger K. Instead of fixing K after an error has occurred, we overfit K according to the expected variation of the delays, computed from all recent delay measurements of e_i and the standard deviation. The added safety margin is the product of their standard deviation and a scaling factor λ is defined by the system architect to trade latency for better detection probability, see Section 3.3.6.

[1] In the distributed system the wall-clock arrival time stamps are unknown. Hence, the arrival time stamps are set according to *clk*. Due to the fact that we use sensor events as stable *clk* update events these time stamps are quite accurate and hence the derived K values work.

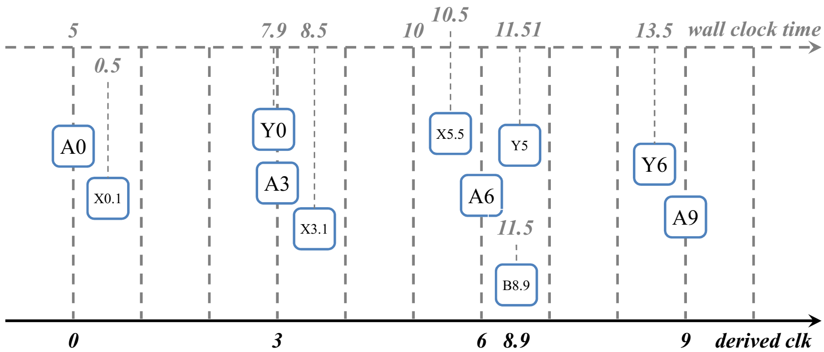


Figure 3.6.: Setting *clk* by an ordered event set with different delays.

The second counter-measure works for event detectors further up the hierarchy. They receive an advance notice of an upcoming delay change. Consider an increased K-value at a certain event detector. Up to now, this fact remains unknown to all subscribing event detectors further up the detector hierarchy. The upper level detector will only notice a changed and potentially too large delay when the subscribed event is actually generated. Then the upper level K may be too small to avoid misdetection and retrofitting of K. It is better to warn the upper level detector in advance. Hence, whenever a K increases, we notify all subscribers by sending a pseudo event with a suitable time stamp (see Section 3.5.2 for details) so that they can modify their K-values, if necessary. The recipient only uses such a pseudo event for configuration purposes.

3.3.3. The *clk*-setting dilemma

In the following we briefly show that setting *clk* by several event types is insufficient if those event types only form an ordered sub-stream (instead of having equal delays). Consider the example event stream in Figure 3.6. The rectangles denote events and their corresponding occurrence time stamps (the arrival time as well as the wall clock time increase to the right). The grey wall clock time stamps are not directly accessed by the system and are only used to show why ordering will eventually fail. The *derived clk* denotes the currently set *clk* value. In relation to the wall clock time the *real* delay of `A`

is 5 time units and that of B is 2.6 time units. But as we have no access to the wall clock time we do not know that their delays are different.

Although events A and B form an ordered sub-stream (A0, A3, A6, B8.9, and A9) the system will not order the events correctly when setting *clk* by both event types. At the beginning with A0, X0.1, Y0, and A3 we have the first measurements, derive the maximal delay of X=2.9 and Y=3, and set K=3. This information remains valid with the upcoming set of events X3.1, X5.5, and A6. But when B8.9 arrives *clk* is set to 8.9, the buffered X5.5 satisfies the K-slack inequation (5.5+3≤8.9), and is emitted before Y5 is received slightly afterwards. Although the delay measurements are correct the algorithm incorrectly orders the X and Y events.

The core problem is that we conduct delay measurements in respect to an event type that has a higher delay (in that case A with a delay of 5 time units) than that event type that is later used to set *clk*. As soon as we set *clk* by an event type with lower delay (in that case B that has a delay of 2.6 time units) the algorithm falsely relates the buffered elements and emits events too early for the current delay measurements.

If events are sorted by a clock that does not reflect the real time, postponing events by measured delays may not be appropriate. We will also notice this in the formal proof in Section 3.3.5. The algorithm may then work correctly for this subset of event types but may stumble again until all measurements are sufficiently correct.

But high data rate sensor events (ideally from one source) with precise time stamps are excellent candidates to set *clk*. They then make the dynamic K-slack work when there is no global clock in a distributed system. Moreover, it serves as a general solution in stream applications and improves K-slack behavior in general. In Section 3.3.6 we show how to use event types with different delays to set the internal clock properly.

3.3.4. Event Ordering Units

The resulting middleware has both a stable clock *clk* for the right hand side and a sufficiently good K for the left hand side of our inequation. These values are used to setup the required event ordering unit that turns an out-of-order stream into a sorted input for the detector, see Figure 3.4. The ordering unit is a black box that is mounted between the original event stream and

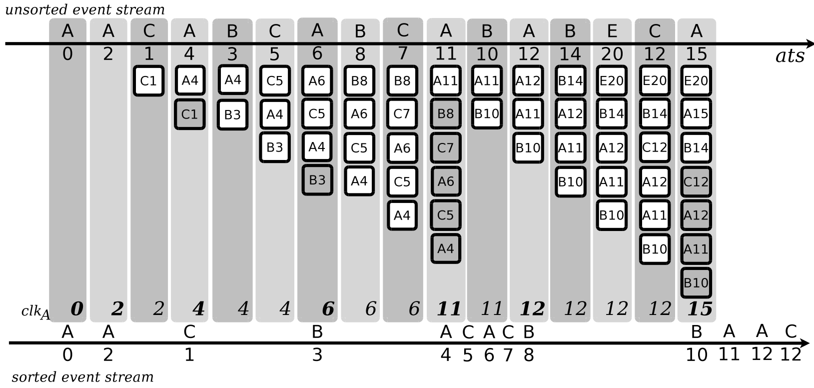


Figure 3.7.: Event ordering unit for event detector D.

the input of the event detector so that the event detector can assume sorted input. Note that there is usually more than one event detector per middleware and machine, each of which has a specific ordering unit with a suitable and detector-specific K that only picks the subscribed events from the main event stream. Algorithm 3.1 shows the pseudo code for the implementation of such an ordering unit. In order to focus on the crucial points we skip the K-overfitting and reduction with λ.

Figure 3.7 shows such an event ordering unit for the input stream of Figure 3.2. *clk* is set whenever an A is received, see the bold values in the clk_A line. At the beginning when A0 and A2 are received, there are no measurements and K is still 0, which means that both events are immediately passed to the output stream and are not delayed (they fulfill $e_i.ts{+}K{\leq}clk_A$). When C1 is received, it is pushed to the buffer and waits until A4 updates clk_A. As the delay for C1 is 3=4-1, we set K=3 and relay C1 ($e_i.ts{+}K{=}1{+}3{\leq}clk_A$ =4). A4 is buffered at least until clk_A equals 7=4+K. With A6 the maximal delay of B3 is 3=6-3, K holds, and B3 is passed to the output stream.

The output stream is a sorted sequence of events with a minimal delay. Whenever an input event is received (pseudo events are ignored), it is sorted into a buffer according to its occurrence time stamp. If out-of-order events are rare, insertion sort of new events usually is just a simple and fast push to the head, i.e., top, of the buffer. Whenever *clk* is updated and $e_i.ts{+}K{\leq}clk$ holds

Algorithm 3.1: Pseudo code for the ordering unit (without λ, and without reduction of K)

```
Data: InputEvent e, EventDetector d, DelayCalculationList lst, K_d,
      clkEvents
if e.id ∈ d.GetSubscriptions() then
    d.inputBuffer.InsertionSort(e);
    lst.add(e);
if e.id ∈ {clkEvents} then
    clk ⟵ e.ts;
    d_max ⟵ 0;
    // calculate new delays
    for Event e_tmp : lst do
        d_tmp ⟵ clk − e_tmp.ts ;                    // delay of d_tmp
        delays [e.id] .add(d_tmp);
        d_max ⟵ (d_tmp > d_max) ? d_tmp : d_max;
    if d_max > K then                              // check K-increase
        K ⟵ d_max;
        // ts=clk-K
        propagatePseudoEvent(d.id, K, clk − K)
    while Event e_tmp ⟵ d.inputBuffer.front() do         // event
    relaying
        if e_tmp.ts + K ≤ clk then
            if e_tmp.isNoPseudoEvent then
                d.relay(e_tmp);
            d.inputBuffer.popFront();
        else
            return;
```

for some tail events e_i in the buffer, we emit those e_i to the output stream and process the next input event. In the startup phase of the example, only C1 remains out of order since we did not measure a delay yet. In Section 3.5 we describe two techniques to improve startup behavior.

3.3.5. Formal Proof

We now formally prove the correctness of our dynamic K-slack approach. Recall that K_d is defined as the maximal delay of all events e_i that are subscribed by an event detector d. To prove the correctness of our distributed K-slack approach, we must prove that for every event detector d and for any time clk, the following two properties hold:

1. **Delay Measurement Property.** The *real* delay (with respect to its occurrence wall-clock timestamp) of an event e_j is less than or equal to the sum of the real delay of e_k, $e_k.id =$ `ID` (events of type `ID` are used to set clk), and the calculated maximal delay of e_j. More formally: under the precondition that $\delta(e_j) = \delta(e_j|clk \leftarrow e_k)$, i.e., that the delay of e_j is estimated by comparing it to e_k (that sets clk), we must prove that the real delay of e_j

$$\hat{\delta}(e_j) \leq \hat{\delta}(e_k) + \delta(e_j). \tag{3.1}$$

   $\hat{\delta}(e_j)$ denotes the real delay of e_j according to wall-clock time. The wall-clock time is the real time that is not used by our method but only applied here to prove the correctness. In other words, the real delay of e_j must be less than or equal to its maximal delay $\delta(e_j)$.

2. **Event Ordering Property.** For any pair of events e_i and e_j we have to prove that if $e_i.ats > e_j.ats$ and $e_i.ts < e_j.ts$ (e_i occurred before e_j but is received later) the left side of the K-slack inequation for event e_j does not hold before e_i is received, i.e.,

$$e_i.ats \leq e_j.ts + K. \tag{3.2}$$

   In other words, the time at which e_j is relayed to the event detector is later (larger) than the time at which e_i is received so that the ordering unit can insert e_i and properly reorder e_i and e_j.

To prove the second property we assume that delays have already been sufficiently measured and that K is stable.

Proof of Property (3.1), Delay Measurement:

Recall that the maximal delay of an event e_j is defined as $\delta(e_j|clk \leftarrow e_k) = e_k.ts - e_j.ts$, with the restriction that the wall-clock arrival time wc (which we have no access to) of the events is $wc(e_j)<wc(e_k)$. When we insert our delay calculation into the property of inequation (3.1) we get

$$\hat{\delta}(e_j) \leq \hat{\delta}(e_k) + e_k.ts - e_j.ts$$

Moreover, since the real delay of e_j is $\hat{\delta}(e_j) = wc(e_j) - e_j.ts$, $e_{j/k}$ can be reformulated as

$$e_{j/k}.ts = wc(e_{j/k}) - \hat{\delta}(e_{j/k}), \tag{3.3}$$

which is the events' time stamps expressed in terms of the wall-clock time. This gives

$$\begin{aligned} & \hat{\delta}(e_j) \leq \hat{\delta}(e_k) + \left[wc(e_k) - \hat{\delta}(e_k)\right] - \left[wc(e_j) - \hat{\delta}(e_j)\right] \\ \Leftrightarrow \quad & \hat{\delta}(e_j) \leq wc(e_k) - wc(e_j) + \hat{\delta}(e_j) \\ \Leftrightarrow \; & wc(e_j) \leq wc(e_k), \end{aligned}$$

which is always true under the assumption me made at the beginning, that e_k is received after e_j. □

Proof of Property (3.2), Event Ordering:

In order to prove that no event detector receives out-of-order events when setting K to the maximum of all event delays $\delta(e_n)$ we show that for any pair of events $\langle e_i, e_j \rangle$ we reorder it correctly. As the property then holds for any pair of subscribed events this proves the property for the whole set of events. Formally, we need to prove that $e_i.ats \leq e_j.ts + K$ holds:

$$\begin{aligned} e_i.ats &\leq e_j.ts + \max_n \left[\delta(e_n)\right] \\ e_i.ats &\leq e_j.ts + \max\left[\delta(e_i), \delta(e_j)\right]. \end{aligned}$$

Since $e_j.ts > e_i.ts$ but $e_j.ats < e_i.ats$ and Property (3.1), it follows that $\delta(e_i) > \delta(e_j)$. We can simplify the inequation:

$$e_i.ats \leq e_j.ts + \delta(e_i).$$

From Property (3.3) follows that

$$\begin{aligned} \hat{\delta}(e_i) &= wc(e_i) - e_i.ts \\ \hat{\delta}(e_i) &\le e_i.ats - e_i.ts \\ e_i.ts + \hat{\delta}(e_i) &\le e_i.ats, \end{aligned}$$

which we can replace in the inequation:

$$\begin{aligned} &e_i.ts + \hat{\delta}(e_i) \le e_i.ats \le e_j.ts + \delta(e_i) \\ &e_i.ts + \hat{\delta}(e_i) \le e_j.ts + \delta(e_i). \end{aligned}$$

Since $\delta(e_i) \ge \hat{\delta}(e_i)$, which follows from the proof of Property (3.1), and $e_i.ts < e_j.ts$, which is the precondition, the inequation is always true. □

Hence, based on the proof of Property (3.1) that the maximal delay $\delta(e_i)$ is larger or equal than the real delay of e_i (which means that waiting for $\delta(e_i)$ time units suffices to receive a late e_i), the proof of Property (3.2) shows that input event sets are always reordered correctly.

3.3.6. Summary and Further Improvements

This section described why out-of-order event processing is critical and why existing approaches fail for distributed event processing. We have presented our extended K-slack approach with dynamically adjusting slack buffers. As the event detector distribution and system topology are unknown a-priori, K-slack cannot be parameterized properly at compile time. Therefore we measure event delays and extract a local clock out of the event stream, and configure our algorithm to properly estimate the disorder at runtime to provide sorted event streams with lowest latency. The presented solution can also cope with back-setting delays whereas most other methods would inevitably fail.

However, there are two measures that can improve the approach and also give it greater stability. We hint the basic ideas behind them in the following.

Estimation of the safety margin. Currently we describe the safety margin of an ordering unit's K by the standard deviation of the subscribed events. Although this has no mathematically profound background it

works well for the event streams we process. However, it may be a better idea to derive the distribution of the events' delays, i.e., it may be a mixture of a normal distribution introduced by processing and a heavy-tailed distribution introduced by the variable network delays [88], and to adjust the safety margin and the λ in respect to that combined distribution.
However, since the speculation we propose in the upcoming chapter also solves this problem and reduces latency we did not deeper investigate on the perfect combination of K, λ, and the standard deviation.

Using multiple event types to set *clk*. The presented method works well for high data rate sensor streams as their update frequency is very high. However, if those data streams are not available or a sufficiently large set of events with equal delays cannot be found the presented method cannot measure the events' delays properly and evaluates the K-slack inequation less frequently. This introduces additional latency.
A solution is to use several event types with different delays to set *clk*. Consider the stream of events `A0`, `A3`, `B2`, `A6`, `A9`, `B20`. *clk* is set by events of type `A`. With the knowledge we gain from the combination `A3/B2/A6` we know that the maximal delay of `B` (in respect to that of `A`) is 4. At the same time, we know that its minimal delay is 1: as we received `A3` before `B2` we can conclude that the delay of `B` must be at least 1 time unit. Hence, from the reception of `B20` we know that we can set *clk* to 21 as `B` has a delay of at least 1 time unit, i.e., if we received an `A` event at this exact moment, it would have an occurrence time stamp of at least 21.
Hence, although `A` and `B` do not have equal delay and do not form an ordered subset we can use both event types to set *clk* based on the relative measurements between them.

3.3.7. Implementation Details

In order to increase the efficiency and hence the event throughput and detection latency of the provided techniques our implementation is slightly different. In the following we briefly describe the differences and provide some justifications for them.

Sensor stream insertion. The ordering units take incoming events and insertion sort them into their ordering queues. Since there are usually several ordering units per node a recently received event is inserted in more than one ordering queue. While this does not affect the correctness of the proposed algorithms in general it nevertheless has two disadvantages.

First, the event's data is stored many times. To get away from this we insert the event's data into the global (intra-node) ring-buffer, see Section 2.2, and only insert a reference to the event's data into the ordering buffers.

Second, as an event type is often subscribed by several event detectors the effort for insertion sort is introduced several times per event. While this also does not affect the algorithm's correctness it may introduce significant runtime overhead if both the data rate and the occurrence frequency of out-of-order events are high. For instance, the high data rate position streams are usually inserted into several ordering buffers. However, since these buffers also contain different other types of events insertion effort cannot be *reused*.[2]

Figure 3.8 shows the alternative implementation for ordering high data rate position streams. We create an individual ordering buffer for this type of events (see the rectangle on the right). Each ordering unit that subscribes position streams then holds a pointer (or a currently processed maximal time stamp), see *NextPos*, that points to the next element in the external ordering buffer (position stream buffer). Whenever the ordering unit later checks the K-slack inequation for event emission it must compare both time stamps: the one of the ordering unit's head element and that of the *NextPos* pointer in the position stream buffer. The ordering unit then emits the event with the lower time stamp. Events that are emitted from the ordering unit's buffer are purged while the events from the position stream buffer remain until no ordering unit requires them any longer.

This puts more CPU consumption on the time of emission (instead of the insertion) but helps to drastically reduce the overhead of sorting the sensor event stream twice.

[2] Actually, insertion sorting of the sensor event stream in each ordering unit overloads the system yet with only four ordering units and is hence highly prohibited.

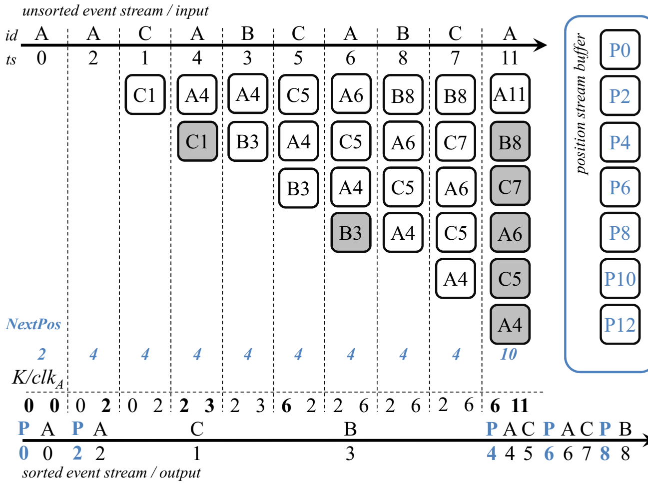


Figure 3.8.: Iterators for ordering units.

Ordering unit separation. The insertion of new elements into the ordering units' buffers is initially triggered from within the receiving threads. Hence, it is inevitable to separate event processing tasks from insertion tasks (unless we want to block receiving). Hence, whenever we insert a new event that may cause an update of *clk* we identify the events that may now be processed, i.e., that satisfy the K-slack inequation, and hand them over to a distinct scheduling unit. This scheduling unit employs several worker threads that take events from a ready-list and invoke the event detectors' callback functions.

Event queue time barrier. A further issue arises in the statically sized ring-buffer into which we insert the events' data. The buffer works as efficient as possible and does not apply any synchronized areas or locks by exploiting compare-and-swap techniques, see Appendix A.1. A problem is that new elements are always inserted at the current write index. This index does (1) not necessarily point to the beginning of a previously inserted event, and (2) may override an event that is still referenced from an ordering unit's buffer. If the ordering unit later accesses the element in the buffer it fails since it has been (partially) overwrit-

ten. This may happen when buffers run with very large K values. The solution is either to choose the buffer size large enough or to employ a more sophisticated buffer that would imply worse throughput/latency performance instead.

3.4. Evaluation

For the evaluation we have analyzed position data streams from a Real-time Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. The RTLS tracks 144 transmitters at the same time at 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Each player is equipped with four transmitters, one at each of his limbs. The sensor data consists of absolute positions in millimeters, velocity, acceleration, and Quality of Location (QoL) for any direction [129], see Section 2.4.1 for more details.

Soccer needs these sampling rates. With 2,000 sampling points per second for the ball and a velocity of up to 150 km/h, two succeeding positions may be more than 2cm apart. Soccer events such as pass, double pass, or shot on goal happen within a fraction of a second. A low latency is required so that a hierarchy of detectors can help the human observer, for example a reporter, or a camera systems that should smoothly follow events of interest, to instantly work with the live output of the system.

We present results from applying our event processing system and our algorithms on position data streams from the stadium. Our platform consists of several 64-bit Linux machines, each equipped with two Intel Xeon E5560 Quad Core CPUs at 2.80 GHz and 64 GB of main memory that communicate over a 1 Gbit fully switched network. For our tests we organized a test game between two amateur league football clubs and processed the incoming position streams from the transmitters.[3]

Section 3.4.1 analyzes delay and timing issues. We discuss how certain delay types affect event detectors from different hierarchy levels. Although out-of-order events are predominant and certain events are significantly delayed our technique nevertheless derives optimal K-values. Section 3.4.2 evaluates the system performance and its robust and accurate event detection. We focus on measurements of a specific event detector and show how delays of input

[3] FIFA rules do not allow a continuous operation in premier league matches.

events have influence on the self-adapting ordering units of the middleware. We prove that we estimate an optimal K at runtime with the result of smaller buffers and less latency. Overall we achieve a close to linear performance scale-up for distribution over several machines.

3.4.1. Delay Discussion

High data rate event processing cannot afford to ignore out-of-order events. This section quantitatively analyzes where delays come from and shows that they are significant and vary between events. Delays add up until most events arrive out-of-order. As mentioned before, the total delay of an event is a sum of sub-delays: position jitter delay, ordering delay, processing delay, network delay, and back-setting delay. The position jitter delay is only introduced once whereas the latter are introduced for every event separately. These delays are characterized below.

Position jitter delay.

RTLS data usually has a variation in the time it takes to send position data from a source to a destination (jitter). In other words, the delay of positions varies and the jitter defines the degree of this variation. Consider two positions p_i and p_j with $p_i.ats<p_j.ats$. The jitter delay d_j of position p_j is

$$d_j(p_j) = \max_i \left[(p_i.ts - p_j.ts) + (p_j.ats - p_i.ats), 0\right],$$

which is the maximal difference of the occurrence time stamps normalized with the difference of the associated arrival time stamps. For instance, positions p_i and p_j with p_i.ats=13, p_j.ats=14, $p_i.ts$=12, and $p_j.ts$=9 have a jitter delay $d_j(p_j)$ of (12-9)+(14-13)=4, as p_j was delayed for 4 time units more than p_i. There are no negative jitter delays.

We recorded position arrivals in our RTLS at 70% of the maximal system capacity, which is with 36,000 positions per second. Consider the *single positions* plot in Figure 3.9. Positions from ball transmitters usually have a jitter delay below 5ms, caused by routers and different transmission lane lengths. Positions from other transmitters have jitter delay above 5ms because for them, the position data is extracted from the microwave signal with lower priority. Some position events even have a jitter delay of up to 100ms.

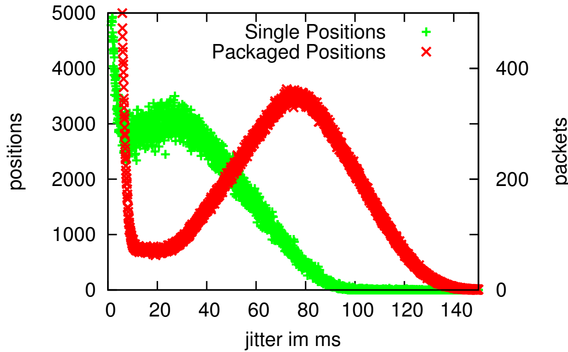


Figure 3.9.: Position jitter delay.

An RTLS usually packages ten positions of a particular transmitter into one packet. Packaging adds delay to all but the last position in the package and hence adds jitter delay. For the low priority positions this adds up to 145ms, see the *packaged positions* plot in Figure 3.9.

To provide a stream of sorted sensor events to the event detectors, event ordering units must set their K at least to the maximal position jitter delay.

Ordering delay.

On top of that an ordering delay is added as events have to be postponed to guarantee a correct detection, see Section 3.3. The ordering delay of an event is the time needed so that $clk{<}e.ts{+}K$. Both K and the time we need to postpone certain events grow with the hierarchy level. For instance, to detect a deflected/blocked shot-on-goal, we need to subscribe the shot-on-goal event (around 180ms delay), the player-hits-ball event (around 145ms delay), and the proximity event (around 145ms delay). Since the shot-on-goal event has the largest delay, the other events need to be buffered for at least an additional 35ms to be properly ordered.

Ordering delays can vary from a few milliseconds to hundreds of milliseconds. Figure 3.10 shows the distribution of optimal K-values for the soccer application up to hierarchy level 9. On higher hierarchy levels K-values are much larger but subscribed events occur only seldom. Note that the drop at

hierarchy level 2 is due to the fact that our soccer application uses some event detectors that do not rely on low priority transmitters.

Processing delay.

The processing delay is the time that is needed to detect an event based on a particular input event (the runtime of the event detector). The value of this delay depends on both the complexity of the algorithm and the system load. For most event detectors, the processing delay is relatively low compared to the other delay types. For instance, the proximity detection runs for about 0.2ms on a newly arrived position packet with 10 positions. The CPU is fully loaded once it has to process 5,000 packets. Event detectors for more complex analytics run longer but are not triggered that often. Usually an EPS splits events over several event detectors and forms a processing hierarchy so that the processing time/delay of a single detector is often negligible.

Network delay.

The network delay is the time that is needed to send an event from one node to another. This delay is influenced by network load, bandwidth and topology, and publisher/subscriber distribution. In a local network, this delay is less than a millisecond. If we have a more widely distributed network or wireless LAN, delays may reach values of tens of milliseconds.

Back-setting delay.

An event may be set to an earlier time stamp than the time stamp of the events that trigger its detection. For instance, a shot on goal cannot be detected (and told from a cross pass) before the ball actually leaves the player. Only then the direction of the shot can be estimated accurately. Hence, with a minimum radius of 1 meter and a shot velocity of 70 km/h, the direction of the shot is detected 51ms late, implying a back-setting of 51ms. In our soccer application, we add back-setting delays whenever correct decisions cannot be made instantaneously, which is in about ten percent of the event detectors. In our test matches most of the back-setting event detectors added a back-setting delay between 50ms and 150ms. There are some event detectors that even add a back-setting delay of several seconds.

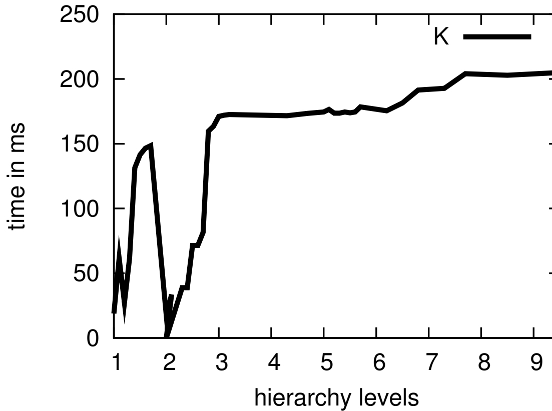


Figure 3.10.: Distribution of K. K values between i and $i + 1$ show event detectors of level i, sorted by growing K.

From Section 3.3 we know that event delays are used to select K-values. On lower levels of the detector hierarchy, where position data is processed, delays are dominated by the position jitter delay. On higher hierarchy levels other delay types gain more importance. Position jitter delay is only introduced at the lowest level, whereas other types of delay add up along the hierarchy, see Figure 3.10.

As a result of the delays and the resulting K-values, in our system over 95% of all events arrive out-of-order. Even when sensor events are excluded, still 9% of all events arrive out-of-order. Out-of-order events are thus predominant. Depending on the delays, the hierarchy, and the event load, ordering the event streams consumes up to 20% of the available CPU power.

The K-values must be as small as possible to reduce latency. Besides the fact that we may not even be able to predict many of the above delay types at all, manually pre-selecting a fixed (increase of) K is not a viable option. For reasons of reliability any manual approach would need to significantly overfit the K-values. For the K's of the first level (which is the position jitter), K-values around 500ms per detector are reasonable because the system could become fully charged and packets would get lost. For any higher hierarchy

level, additional 25ms are needed to compensate for worst-case network, system inconsistencies, and detector runtimes that usually also depend on the event load that we cannot estimate any better. Even without any back-setting delays, the K-values of the highest event detectors are at least around 1 second. In contrast, Figure 3.10 shows that our technique finds much smaller K-values suitable for the actual system state instead of a worst-case scenario. We achieve latencies that are less than a fifth of the manual ones.

Such a wide spread of delay types as presented here is not unique to position sensor streams. For instance, it is also seen in financial market data where stock markets are timely synchronized by the backend but the data is distributed over the internet. In RFID systems the situation is similar, as many readers collect data and send them to a centralized server for analysis.

3.4.2. Application Performance

Our middleware implementation takes about 17,000 lines of C++ code to implement the basic functionality and event ordering. On top of it, we implemented 70 different event detectors in 15 levels from simple line events up to complex technical scenarios (also in C++). On average the size of an event packet is about 200 Bytes. The largest is about 4 KBytes.

We first give a throughput analysis of our system before we illustrate the performance of our event ordering approach.

Event Throughput Scalability.

In contrast to many other EPS such as SASE [135] or Cayuga [53] our system can use threads and can be distributed across several nodes to run in parallel. For a benchmark of our system we performed different geometric calculations and processed position information from our localization system. Figure 3.11 shows the system performance when distributing the processing over several threads. We achieve a close to linear performance scale-up of the event throughput with respect to the number of available threads. We reach a plateau for 8 threads or more since this is the number of physical cores. The performance scale-up for the distribution over several machines strongly depends on the actual allocation of event detectors and their processing hierarchy. However, even in the worst scenario we achieve a performance scale-up when distributing across several machines.

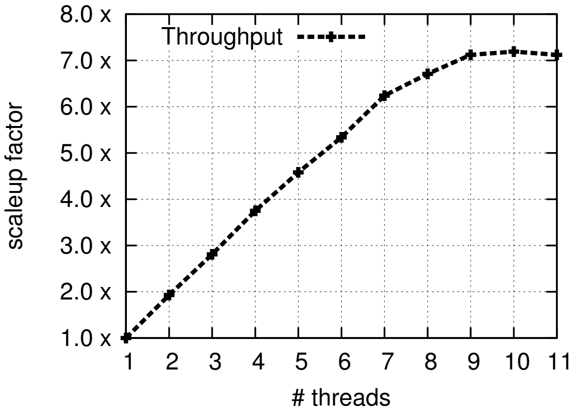


Figure 3.11.: Event throughput for threads.

This performance benchmark is a baseline that shows that our system is efficient on ordered input and is scalable in the number of threads and nodes as the number of trackable objects and sensors grows. The main contributions are that is also works well for massive out-of-order events, which we present next. However, if the event ordering does consume too much processing time, we can add additional machines and distribute event detectors efficiently.

Detector Reliability and Latency.

To evaluate the event ordering units we recorded the delays of incoming events for the *Player Hits Ball* event detector, see Figure 3.12. These events are *Is Near*, *Is Not Near*, both oscillating between 5 and 45 ms delay, and *Ball Acceleration* with a delay of 1-2ms. We took out the single positions jitter for simplification. Other detectors of the soccer application behave similarly. The technique presented in Section 3.3 to select a suitable K-value at runtime correctly orders over 95% of all events even without any a-priori knowledge, see the straight black line. There are rare points at which K is too small and may cause a misdetection, see the crosses, before it is increased. These points are not necessarily a miss-detection as they only mark increases in K. Miss-detection only occur when events with lower time stamps are

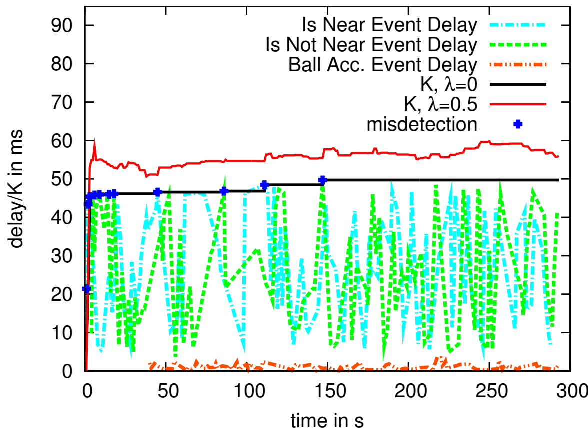


Figure 3.12.: Delays and K.

buffered and emitted before a new event increases K with a lower time stamp than the previously emitted event. In the present case no miss-detection occurred.

However, with an overfitted K and an added safety margin of λ=0.5 the detector works considerably better (upper line): not a single K misses the maximal event delay, the event detector is always supplied with ordered events and is fully reliable.

The result shows that our method chooses K as large as necessary to fit the maximal delay of subscribed events so that we generate a totally ordered event input stream. At the same time, K is only barely above the necessary maximal delay. Small K-values result in small buffers on higher level event detectors and hence lower latencies for event generation. In total, this detector has a latency of up to 59.8ms whereas a manual K-selection by an expert would cause a latency of >500ms. Only 8ms (=13.4%) are caused by the overfitting. It is worthwhile to trade this slowdown for perfect detection quality.

3.4.3. Discussion

We demonstrated quantitatively that a manual K-selection causes a considerably higher detection latency (>8.4×). Our middleware system can work in a network and performs event ordering without any a-priori knowledge. It adaptively finds the best K's that minimize buffer costs and latency at runtime. Event detectors hence see transparently pre-sorted event streams as their inputs and are therefore easy to implement.

Unfortunately, there is no related work that makes use of a similarly large real world setup. Existing work cannot be used for a quantitative comparison because it either lacks support for distributed processing, or for correct event ordering when dealing with negative patterns or back-setting delays.

We deliberately did not benchmark CPU and RAM consumption in more detail as they are not the bottlenecks. In the application area that we target, there is usually sufficient RAM and the fluctuations in the event delays are not so large that CPU power would make a significant difference for the event reordering. The crucial point to reduce latency is to find the minimal time for which events need to be delayed such that event streams can correctly be reordered. Hence, more CPU power or RAM will not reduce the detection latency. We present a technique to use additional CPU and RAM resources to reduce latency by using speculation in the next chapter.

Our technique improves K-slack by adding only little processing overhead. We only need to derive event delays and K-values at runtime to dynamically configure K-slack in an optimal way.

3.4.4. Summary

We first zoomed into the sub-delays of the total event delay, discussed their quantities, and argued that out-of-order events are the rule in event stream processing and that efficient low-latency solutions are highly required. In contrast to the related work, our middleware system can work in a network and performs event ordering without any a-priori knowledge. It adaptively finds the best K's that minimize buffer costs and latency at runtime. Event detectors hence see transparently pre-sorted event streams as their inputs and are therefore easy to implement.

3.5. Initialization

With knowledge about the expected delay of events, the middleware can order them. However, at startup the middleware does not have this knowledge and stumbles, see C1 in Figure 3.7. Here are two ways to correctly initialize the delays.

3.5.1. Iterative Delay Calculation

In many applications we either can run and restart a system several times, or we have some pre-recorded sensor streams that can be employed to calibrate the correct configuration. Figure 3.13 depicts this strategy. The idea is to start from an initial configuration c_0 (= values for all K's) and to derive c_1 (a proper configuration for the event detectors on level 1) by running the system and measuring sensor event delays directly. The application will fail because of stumbling detectors, but at the end of this run all events on the first level have suitable K-values that are used when the system is restarted. Restart i, $i > 1$, hence derives c_i with correctly measured delays for the event detectors on levels $\leq i$. Algorithm 3.2 converges after at most n iterations, where n is the highest hierarchy level. We assume that there are no cycles in the detection hierarchy (that would also cause problems with event ordering).

Iterative delay calculation either requires that there is sufficient training data available, or that the system can be restarted over and over in the environment of use. Moreover, network and CPU loads need to remain stable for any two runs.

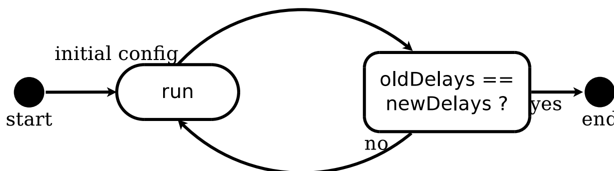


Figure 3.13.: Iterative delay update.

Algorithm 3.2: Iterative Delay Calculation

```
Data: SensorStream p, Configuration conf(null),
      EventDetectionEngine engine
engine.setConfiguration(conf);
for i ← 1 to engine.MaxHierarchyLevel() do
    engine.run(s);
    if conf.delays equals engine.delays then
        break;
    else
        engine.setConfiguration(engine.delays);
return conf;
```

3.5.2. Semi-Configured Delay Estimation

There are cases in which the Iterative Delay Calculation cannot be applied or it takes too long to process the whole (maybe large) training data for n times.

We therefore show a way to initialize the middleware from a different configuration c_{old} instead of starting from scratch. The idea of the *Semi-Configured Delay Estimation* is to combine delay information of c_{old} with further runtime measurements to derive valid K-values for the current configuration. To implement this we zoom into the total delay of an event and find it to be the sum of sub-delays: sensor jitter delay d_j, ordering delay d_o=K, network delay d_n, processing delay d_p, and back-setting delay d_b (recall Section 3.4.1 for more details). Only the first three sub-delays d_j, d_o, and d_n depend on the underlying network topology, that is the distribution of event detectors, and are therefore most definitely different from c_{old}. The other two sub-delays remain stable between c_{old} and c_{new}. Usually d_p is uncritical compared to the other sub-delays because it has a relatively small influence on the total delay. The back-setting delay d_b is application-specific and hence it does not depend on the configuration.

Similar to the Iterative Delay Calculation, we start from the lowest level of the hierarchy, where input events depend directly on sensor data. Instead of processing real training data, we circumvent the event detectors and emit pseudo events. Therefore time consuming processing of events is not required. Pseudo events carry the same information as real events (ID and time stamp) but are not processed by event detectors.

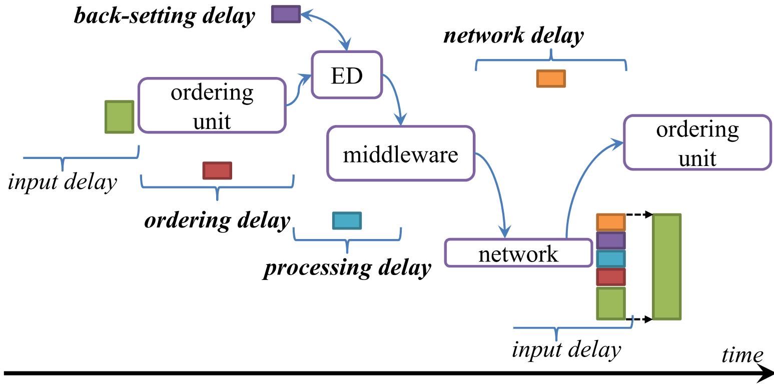


Figure 3.14.: Composition of sub-delays.

Figure 3.14 shows the composition of the delay of an event by its sub-delays and how the particular delays are influenced by the event detector hierarchy. An event already has a delay (*input delay*) at its reception. The *ordering delay* is the first type of delay that is added on top of the event's delay and denotes the time an event is kept back by the ordering unit to guarantee a correct ordering to the event detector. Certainly, this delay depends on the input delays of the other subscribed event types. As soon as the event is emitted to the detector the *processing delay* is added on top (which is usually comparably small). Before an event detector generates an event and sends it to the middleware it may also set the time stamp of the event to some previous point in time. This *back-setting delay* takes no real time but is added instantaneously by reducing the event's time stamp. When an event detector generates an event and the middleware sends it to another node, an upper event detector's ordering unit receives this event only after an additional *network delay* (if there is any network communication) has been added on top. Finally, this upper level ordering unit measures the event's delay (input delay) without having access to its sub-delay components.

We assume that sensor data is already received and we measure d_j directly. Then, we initially generate pseudo events for events that would be emitted by event detectors of the first hierarchy level. The jitter delay d_j is directly measured from the sensor stream at each node and is equivalent to K/d_o at

higher hierarchy levels since it reflects the delay differences of the sensor events. The processing and back-setting delays d_p and d_b are taken from c_{old}. To make sure that the pseudo events have the same delay as on real sensor event streams, we must ensure that $clk{=}e.ts{+}d_j{+}d_p{+}d_b$ and fake $e.ts$ appropriately. In other words, we set $e.ts$ as if we would postpone the processing by d_j, process it (d_p), and set it backwards by d_b. We then emit the pseudo event. When the subscribing event detectors receive this pseudo event, d_n has been added implicitly because the event was passed over the network (if necessary). Hence, the pseudo events have similar delays as the real events from training data.

Algorithm 3.3 iteratively traverses the hierarchy and propagates all pseudo events through the network. After they have reached the highest hierarchy level, K values for all detectors are reasonably set. There is no need for further iterations.

3.5.3. Summary

This section presented two techniques for correct system initialization that improve startup behavior: *Iterative Delay Calculation* and *Semi-Configured Delay Estimation*. The latter is as precise as the available knowledge on back-setting and processing sub-delays.

Both initialization techniques break when ordering in cycles exist within the event hierarchy. However, when an event hierarchy requires ordered event delivery within a cycle then ordering itself is ill-defined as this cannot be established (unless events are subscribed unsorted).

3.6. Related Work

As already discussed, related work in the field of event processing is manifold since requirements are diverse. Section 3.6.1 gives an overview of recently emerged EPS. We focus on RFID-based systems because their requirements are most similar to ours when considering different data sources and out-of-order events. Section 3.6.2 then focuses on methods and techniques from other contexts.

Logical clocks by Lamport et al. [84] cannot be applied since the total order provided by local clocks is ambiguous on different nodes. The *happened-*

Algorithm 3.3: Pseudo Event Propagation

```
Data: Configuration c_old, c_new, Clock clk, Jitter d_j
begin
    for EventDetector ed : GetFirstLevelDetectors() do
        for Event e_i : GetPublishedEvents(ed) do
            d_p←c_old.GetProcessingDelay(e_i);
            d_b←c_old.GetBackSettingDelay(e_i);
            e_i.ts←clk-d_j-d_p-d_b;
            Send(e_i);
    while ReceivePseudoEvent(e) do
        c_new.SetDelay(e.id, clk − e.ts);
        for EventDetector ed : GetDetectorsBySub(e) do
            if c_new.AllMeasured(ed) then
                for Event e_i : GetPublishedEvents(ed) do
                    d_o←c_new.GetOrderingDelay(ed);
                    d_p←c_old.GetProcessingDelay(e_i);
                    d_b←c_old.GetBackSettingDelay(e_i);
                    e_i.ts ← clk-d_p-d_b-d_o;
                    Send(e_i);
end
```

before clause only works for events that have a common equal origin. If the origin is different, i.e., if the events have not been triggered or generated by the same source, or if no central coordinator is established the total order provided by the local clocks may be incorrect.

Vector clocks [101] enhance logical clocks so that not only pairs but many different events can derive the *happened-before* clause. However, both methods apply completely different time model semantics: the happened-before clause is strictly defined to take the logical increment counter in both methods into account. This counter actually reflects the detection time stamp of an event and not its original occurrence time stamp. Moreover, both methods are not well-suited for *scheduling* of events as they cannot derive the point in time an event can assume that no other event is received with a lower time stamp.

We avoid such problems by measuring the relative delays of events on each node at runtime. Ordering units withhold events from detectors as long as necessary for a total order. Hence, there is neither a need for retraction of events or for restoring of detectors, nor are there any fixed synchronization points. Moreover, we reduce the complexity of the detector implementation as they can be implemented without any consideration of event delays.

3.6.1. Event Processing Systems

SASE [135] is an event processing engine for RFID-readings that is also the foundation of follow-up work [3, 92]. SASE works with Nondeterministic Finite Automata (NFA) generated from event queries. Although Li et al. [87] solve some problems of out-of-order event arrival, SASE fails when it is distributed over several machines.

Another EPS for RFID-readings is Cayuga [53]. Work built on top of Cayuga is demonstrated in [27, 54]. Cayuga uses a timing mechanism built on priority queues and *epochs*. As in SASE, Cayuga assumes that events are delayed for at most K time units. K is defined a-priori. In an epoch it only processes events with a time stamp from that epoch. Further work by Brenna et al. [28] distributes Cayuga over a network. In contrast to our work there is no back-setting and events may not cross epoch boundaries.

The Complex Event Detection and Response system CEDR [20] comes with a query language to express a wide range of event patterns, comprising temporal correlation and negation. The language also has consistency levels for latencies and out-of-order events. CEDR handles out-of-order events by retracting incorrect output and by adding correct, revised output in turn. However, if out-of-order events are predominant, almost all generated events must be retracted, often triggering a cascade of retractions along the detector hierarchy. This poses non-trivial challenges to the memory management and to preclude retractions we have to accept high latencies, see also Chapter 4 for more details.

We avoid such problems by measuring the relative delays of events on each node at runtime. Ordering units withhold events from detectors as long as necessary for a total order. Hence, there is neither a need for retraction of events nor for restoring of detectors, nor are there any fixed synchronization points. Moreover, we reduce the complexity of detector implementations as they can be written without considering event delays.

3.6.2. Methods

O'Keeffe et al. [106] analyze the influence of communication errors in addition to timing uncertainties due to the lack of a global clock in distributed systems. Their complex event language can declare detection policies to specify how to deal with errors. The authors address different tasks than we do: their events have time intervals rather than time stamps due to clock uncertainties, their event load is significantly lower, and they do not consider timing issues introduced by distributed processing as critical.

Brito et al. [29] speculatively process events in parallel. This is achieved by means of an underlying Software Transactional Memory (STM) infrastructure. The basic approach is different from ours in that the authors assume that stateful event detectors need to be executed sequentially. Although the presented STM method achieves good results for parallel execution on multicore systems, it would suffer when it is distributed because buffering and committing of events would no longer be efficient.

Fodor et al. [59] split complex events into a set of binary goals of subpatterns. Goals are chained and a complex event is detected whenever the top goal is reached. They do not cope with additional delays and also assume that events are processed on a single machine. Signalling those goals over a network presumably degrades performance and also introduces delays. Moreover, in order to form those goals, events cannot be arbitrarily defined as they must be implemented in an EDL with limited expressiveness.

GauthierDickey et al. [61, 62] focus on event ordering in peer-to-peer games for massively multiplayer online games (MMOGs). They introduced NEO, a low-latency event ordering protocol that prevents players from cheating in an untrusted environment and also abandons the need for a client/server infrastructure. Time is divided into rounds of fixed length that are an implicit upper bound for the delays. NEO does not fulfill our requirements because it simply discards events that are too old for the current round.

Tucker et al. [127] and Li et al. [86] use special annotations embedded in data streams, called *punctuations*, to specify the end of a subset of data. These serve as window delimiters and time markers, and inform that no event will be generated with a lower time stamp. However, for negative patterns, an event detector must fire a punctuation at each sensor event, otherwise upper level detectors will buffer and hold the processing. The introduced network load is not tolerable. Also timed punctuations do not work in application-specific

event detectors due to the lack of knowledge about the runtime configuration and would introduce latency.

Chandramouli et al. [32] permit stream revisions by using punctuations. They give an insertion algorithm for out-of-order events that removes invalidated sequences. Since our system is highly distributed, removing invalidated sequences is not possible. Events that need to be invalidated may already be consumed/processed on other nodes.

Srivastava et al. [120] model stream time differences by wall-clock dependencies, and define clock-skews between data sources. For any data arrival they set heartbeats on each stream. A heartbeat at wall-clock time c is the maximal application time stamp τ such that all tuples arriving from that stream after time c must have a time stamp larger than τ. By using the local clock and by generating heartbeats, time constraints can be defined with high accuracy. This also makes their method costly (slow) and limits the scalability of the system. With 150 detectors (streams) and 50,000 sensor events per second we would insert 7.5 million heartbeat events per second.

3.7. Summary

The presented methods achieve reliable, low-latency, distributed event processing of high data rate sensor streams even under the predominance of out-of-order events. Event delays are measured, the middleware adapts itself at runtime and postpones events as long as necessary to transparently put incoming event streams in order for any application-specific event detectors. No a-priori knowledge of event delays is needed. The presented system works well on a Real-time Locating Systems (RTLS) in a soccer application.

The performance (in terms of latency reduction) is limited by the derivation of *clk*. If clock updates are rare, the detection latency of events increases. To tackle that, we gave a basic idea how to exhaustively exploit the derivation of *clk* and therefore *K* by setting the internal clock by several event types that have different delays. We refer that idea to be subject to future work. However, a second idea is to loosen the strict and conservative buffering approach and to open up for speculative processing and retraction of out-of-order events. This will be the key idea of the upcoming chapter.

4. Speculative Event Processing

We have shown that distributed event-based systems can be used to detect meaningful events with low latency in high data rate event streams. However, both known approaches to deal with the predominant out-of-order event arrival at the distributed detectors have their shortcomings: buffering approaches as the one presented in Chapter 3 introduce latencies for event ordering and stream revision approaches [20, 29, 32, 133] may result in system overloads due to unbounded retraction cascades.

This chapter presents a speculative processing technique for out-of-order event streams that enhances typical buffering approaches. In contrast to other stream revision approaches this novel technique encapsulates the event detector, uses the buffering technique to delay events but also speculatively processes a portion of it, and adapts the degree of speculation at runtime to fit the available system resources so that detection latency becomes minimal.

The speculation technique performs very well compared to existing approaches on both synthetical data and real sensor data from the Real-time Locating System (RTLS) with several thousands of out-of-order sensor events per second. Speculative buffering exploits available and unused system resources and accomplishes a latency reduction of 40% on average.

4.1. The latency/overload-dilemma

Many applications also demand event detection with minimal delays [122]. For instance, the distributed EBS contributed in this thesis detects events to steer an autonomous camera control systems to points of interest, see Figure 4.1. This obviously requires low detection latencies in order not to lag behind.

To process high rate event streams, an EBS usually spreads the computation over several event detectors, linked by publish/subscribe to build an event detection hierarchy. These event detectors are distributed over the available

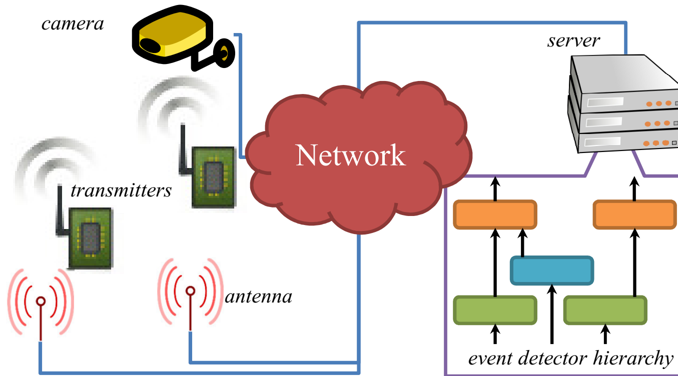


Figure 4.1.: Automatically controlled camera system.

machines. Events arrive at the distributed detectors out-of-order because of various types of delay, see Section 3.4.1. As we have seen, ignoring the wrong order causes misdetection. Event detectors themselves cannot reorder the events with low latency because in general event delays are unknown before runtime. Moreover, as there are also dynamically changing application-specific delay types (like for instance a detection/back-setting delay) there is no a-priori optimal assignment of event detectors to available nodes. Hence, in a distributed EBS the middleware deals with out-of-order events, typically without any a-priori knowledge on the event detectors, their distribution, and their subscribed events.

Buffering middleware approaches withhold the events for some time, sort them, and emit them to the detector in order, see Chapter 3. The main issue is the size of the ordering buffer. If it is too small, detection fails. If it is too large, it wastes time and causes high detection latency. Note that waiting times add up along the detection hierarchy, see Section 3.4.1. The best buffer sizes are unknown and may depend on some dynamic, unpredictable behavior. In addition, there is no need to buffer events that cannot be out-of-order or that can be processed out-of-order without any problems. Buffering middlewares are the basis of reliable event detection, but they are also too costly for many types of events and do not benefit from faster CPUs as they are bound by the waiting times.

Speculative middlewares, the other approach to cope with out-of-order event arrivals, speculatively work on the raw event stream. As there is no buffering, this is faster. Whenever an out-of-order event is received, falsely emitted events are retracted and the event stream is replayed. The effort for event retraction and stream replay grows with the number of out-of-order events and with the depth of the detection hierarchy. This is a non-trivial challenge for the memory management, may exhaust the CPU, and may cause high detection latencies or even system failures. In contrast to buffer-based approaches, a stronger CPU helps, but the risk of high detection latencies remains.

To combine the advantages of both worlds, we propose to add a novel speculative processing to a buffering EBS. The most important requirements are: (1) The middleware can neither exploit the events' semantics nor their use by the event detectors because both of them are highly application-specific. Hence, methods for strong synchronization of event detectors are no viable option. (2) In spite of the speculation, the buffering middleware must keep event detection reliable, i.e., false-positive or false-negative detection must be avoided to prevent system failures. Hence, it is no option to use imprecise approximative methods or to discard events that would cause a system overload.

Our key idea is to use buffering to sort most of the events but to let a snapshot event detector speculatively and prematurely process those events that will be emitted and processed soon. Event detectors are restored whenever a replay occurs. The degree of speculation is adapted to suit the available CPU resources, ranging from full speculation on an idle CPU to plain buffering on a busy CPU. The technique presented here works without knowledge on internal event semantics, can be used for any publish/subscribe-based buffering middleware, and does not use query languages or event approximation.

The rest of this chapter is organized as follows. Section 4.2 reviews related work. Section 4.3 motivates the speculation approach, before we present the main contributions in Section 4.4: a speculative event processing and snapshot recovery, two methods to efficiently retract events across the detection hierarchy, and a greedy method to adapt the amount of speculation at runtime to optimize detection latency by efficiently exploiting unused system resources. Section 4.6 evaluates our methods before Section 4.7 concludes.

4.2. Related Work

As we know of no attempt to fully combine buffering and speculation in an EBS, we discuss both fields in turn.

4.2.1. Buffering Techniques

As we have explained above, buffering is needed for reliable event detection but it also introduces latency. Below we sketch some known buffering EBS and show that for each of them an added speculative component can help to improve latency.

To deal with communication errors and timing uncertainties caused by the lack of a global clock in distributed systems O'Keeffe et al. [106] use time intervals instead of event time stamps. Buffered events are only emitted when time intervals safely overlap. Therefore, as latencies are high, an added speculative component that lowers the upper interval margin can reduce detection latency.

Punctuations [86, 127] are special annotations embedded into data streams to indicate (1) the end of a subset of data, and (2) that no event will be generated with a lower time stamp. For negative patterns like `A!BC`, where no `B` shall occur in between `A` and `C`, a buffering unit must use timed punctuations that introduce latency because firing a punctuation on each event will exhaust the system. An added speculation unit can help because punctuations can also be fired speculatively, so that events can be consumed earlier.

Srivastava et al. [120] model stream time differences and clock-skews between data sources. Their heartbeat stream synchronization among buffers introduces latency. Speculation can help by processing events before the heartbeats actually occur or by emitting the heartbeats prematurely.

SASE [3, 87, 135] and Cayuga [27, 28, 53, 54] both are event processing engines for RFID-readings that work with Nondeterministic Finite Automata generated from event queries. They assume that events are delayed for at most K time units, K to be set a-priori. Although both systems use query languages they can be applied for arbitrary middlewares. But their conservative and a-priori configuration causes high detection latencies in their delay buffers. Speculation can help by processing events before they have been buffered sufficiently long, i.e., for K time units.

4.2.2. Speculative Techniques

Known EBS speculation techniques are either based on query languages and window operators, are imprecise and approximate events, are unreliable, or use a-priori knowledge for their configurations. As they do not fulfill all the above-mentioned requirements for publish/subscribe middlewares, known speculation techniques cannot be used as add-ons to a buffering system.

Approximative techniques [10, 18, 85, 100] use partial or imprecise events to generate the most likely query results first and refine the results incrementally on corrected and more precise events. To limit latency and resource consumption, some of the authors only retract events that have a strong impact on the generated output or use partial events to restore the correct state for the replay. However, as all those systems use query languages and are imprecise, they are unsuitable as a generic speculation component.

Another way to limit the degree of speculation (and hence latency and resource consumption) is to automatically discard older and unprocessed events from their queues and to process more recently received events instead [115], i.e., a form of implicit load shedding. However, this is only applicable to state-less event detectors. In general, event detector states depend on the discarded events so that detection fails if they are simply dropped.

A commit action resolves the speculation (the essence of transaction systems) into a reliable event detection step [29, 133]. But every out-of-order event increases the number of transactions, i.e., the degree of speculation, and results in potential CPU overload and hence system failure. That is why buffering EBS should stay away from transaction speculation.

Window query approaches [92] pipe events through operator graphs that are constructed from queries. The degree of speculation is set by limiting historical operator states. Unfortunately, window operators (such as multi-joins) need event detector definitions in a particular query language for operator graph construction and are hence not general purpose.

Complex Event Detection and Response (CEDR) [20] speculatively generates events from event streams that also may contain revision tuples enhanced with time stamp corrections. Upon a revision event arrival, CEDR only adjusts the time stamps of the generated event instead of retracting the event. But this brakes stateful event detectors that may already have consumed the generated event with the old time stamp, and may have (falsely) generated an event because of the old time stamp.

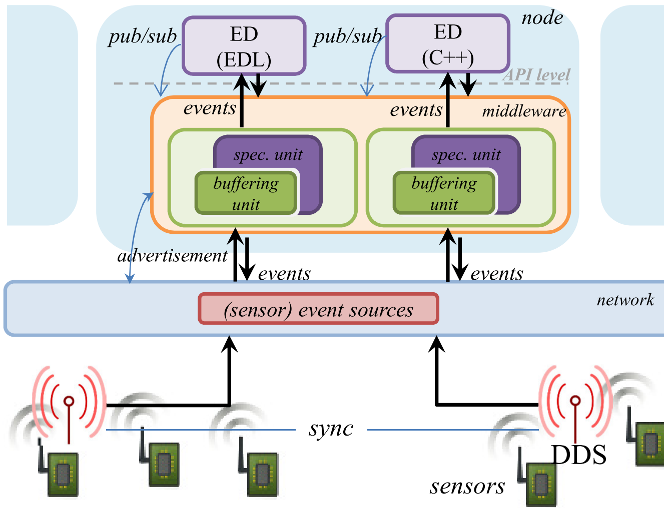


Figure 4.2.: Distributed publish/subscribe EPS with speculation.

Chandramouli et al. [32] limit speculation either by sequence numbers or by *cleanse*. The receiver can use the former to deduce disorder information in the rare cases when particular events are generated at stable rates. The latter only works for a punctuation-based environment, which must incorporate the event definition to limit query windows by setting the punctuation to the latest event time stamps of the event detector. Since the middleware technique cannot access this information, it cannot be used as a generic buffering extension.

4.3. Motivation

In this section, we illustrate the assumptions of the generic buffering in the hosting EBS that we extend with our novel speculative technique. Although Section 4.4 will build on top of this buffering, the novel speculation can easily be adopted by other buffering EBSs because our speculation is general purpose and quite generic.

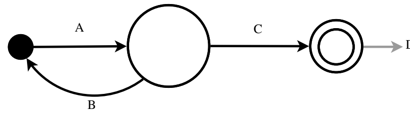
As can be seen in Figure 4.2, the new EBS consists of several data distribution services (DDS) that collect sensor data (for example an antenna that collects RFID readings), and several nodes in a network that run the same event processing middleware. The middleware creates a reordering buffer per event detector (ED). This component is now wrapped with the new speculation unit. The middleware deals with all types of delays such as processing and networking delays or detection delays and does not need to know the complex event pattern that the detector implements (either in a native programming language [75] or in some EDL [135]). Still the detector does not need to know on which machine other event detectors are running nor their runtime configurations. The application code of the event detector assumes that the events are received in correct order with respect to their occurrence time stamps. At startup the middleware has no knowledge about event delays but just notifies other middleware instances about event publications and subscriptions (advertisement) [103]. The middleware is therefore generic and encapsulated.

Since our speculative buffer asks the event detector to provide and to restore snapshots, event detectors have to provide additional functionality, if this is not provided by the programming language.[1] However, in many cases this can easily be achieved as snapshots are handled transparently to the (application) code in the event detector that is used to process the event stream. The application code does not need to care about data synchronization or side-effects as the middleware assures that no events are being processed by the detector while taking or restoring a snapshot.

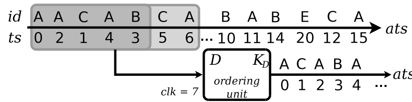
Even if we use minimal K-values for all the detectors across the hierarchy, the resulting combined latencies may be unnecessarily high and a speculative processing may be the better option.

Assume the example in Figure 4.3(b) needs a minimal K_D=3. This delays the detection of any event in upper processing hierarchies by at least 3 time units since event `D` can only be detected with 3 time units delay. However, assume that events of type `B` are rare. Then it may be advantageous not to delay the detection of `D` until we preclude the occurrence of `B`, but to retract a false-detection of `D` in the rare cases when `B` actually occurs. For the event stream in Figure 4.3(b) we can hence detect `D` out of `A4`/`C5` before *clk* is set to 8 (`A4` is emitted at *clk*=7 whereas `C5` must wait at least until *clk*≥8). If there

[1] In Java we could make use of the built-in serialization and in `C#` we could exploit reflection to provide that feature both generically and natively.



(a) Example for `A!BC`.



(b) Sorting window over event stream.

Figure 4.3.: Out-of-order examples.

is a `B` to cancel the detection of `D` later we retract the falsely generated `D`. Hence, the event detector that is used to detect `D` generates preliminary events that can be used to trigger event detectors on higher levels with lower latency.

Hence, the key idea is to combine both techniques and to let the speculation unit wrap a K-slack buffer to process a portion of events prematurely.

4.4. Speculative Processing

Added speculation results in improved detection latency if there are no out-of-order events at all, because nothing that an ordering unit emits and a detector generates has ever to be retracted from further up the detector hierarchy. But the more out-of-order events there are and the deeper the detection hierarchy is, the more complex becomes the retraction work as more memory is needed to store the detector states and as more CPU time is needed to perform the retraction. Hence, in naive speculation approaches, the cost of purging the effects of false speculation can easily outweigh its beneficial effects and can easily increase the latency beyond what pure non-speculative buffering would have caused.

Therefore, the amount of speculation must be limited so that the CPU and memory are used at full capacity on the one side, but without getting exhausted on the other side. System parameters must be deduced at runtime and the speculative ordering units must be continuously adapted to the current system and event load. From now, we use the following terminology.

Event emission and replay. The ordering unit (within the middleware) emits the events to the event detector to process them. An event is emitted *prematurely* if it is emitted before K-slack (pure buffering) would emit it. The speculation component must be aware that premature emission of an event may result in a costly retraction cascade and stream replay while non-premature emissions do not. When an event detector produces an event and sends it to the middleware we refer to it as *generating* an event. Whenever the ordering unit detects a miss-speculation the event stream is *replayed*. In those cases particular (premature) events are emitted repeatedly to the event detector, see Section 4.4.1.

Event retraction. Whenever the ordering unit detects a miss-speculation, events may have been mistakenly processed and need to be retracted. At this point we have to consider two different issues. First, events have been emitted to the event detector in a wrong order and the event stream is immediately replayed. Hence the event detector processes some events twice, which is no viable solution as this may result in a system failure. We solve that by *snapshot recovery*, see Section 4.4.2. Second, the event detector may have generated events based on the incorrectly ordered event input. These events may already have been inserted into the ordering units' buffers of upper level event detectors or may even have been emitted prematurely (causing a ripple effect to upper level detectors). Hence, events must be retracted throughout the entire detector hierarchy, see Section 4.4.3.

4.4.1. Event Emission and Replay

Our speculative event processing technique extends K-slack buffering approches. It puts most of the input events in order but it does not buffer them as long as required for a perfectly correct order. Hence, instead of buffering an event e_i for K time units, we only buffer e_i as long as

$$e_i.ts + \alpha \cdot K \leq clk, \qquad \alpha \in [0; 1], \tag{4.1}$$

with a new attenuation factor α. The attenuation factor is used to adjust the speculation component in the ordering unit. The larger α is, the fewer events

are emitted prematurely, i.e., α=1 is essentially a K-slack without speculation. Smaller values for α switch on speculation. For α=0, there is no buffering/ordering at all because the inequation always holds (except for events with negative delays[2]). For instance, a classic event ordering unit with K=5 and α=1 will emit an event with ts=20 that is received at *clk*=22 not before *clk* is at least 25. Only then e_i.ts+K=20+5≤25=*clk*. Pure buffering middlewares will not just emit the events but they will also purge them from the buffer. In the example with K=5 but with α=0.6 the speculative buffer prematurely emits the event already at *clk*=23 (20+0.6·5=23) and leaves the event in the buffer. With $\alpha \leq 0.4$ emission is even instantly at *clk*=22. The event is purged from the buffer as soon as *clk*≥25.

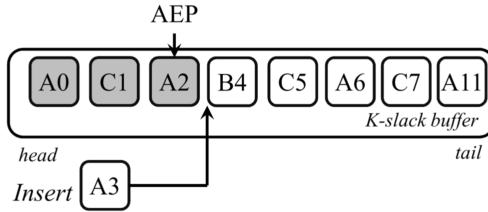
With speculation, events are emitted but no longer instantly purged from the buffer. They may be needed for the event stream replay later. Hence, the K-slack buffer is enhanced with an *already emitted pointer* (AEP) that is a reference to the last element in the buffer that has already been emitted speculatively so far. Next, instead of starting from the buffer's head, the ordering unit starts to evaluate the event time stamps from the event referred to by AEP.

A newly arriving event is inserted into the sorting buffer according to its time stamp. For instance, in Figure 4.4(a) and 4.4(b) `A3` is inserted between `A2` and `B4`. The events from the buffer's head to AEP (shown in grey) have already been emitted. Depending on α, *clk*, K, and AEP there are two possible cases.

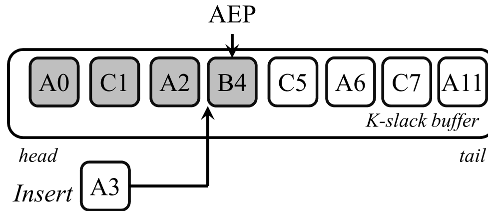
e_i.ts ≥ e_{AEP}.ts. If the time stamp of the recently inserted event e_i is equal or larger than that of the AEP, we insert e_i *behind* the AEP. We can infer that no false-speculation has occurred, and hence we do not need to retract or replay any events. In Figure 4.4(a) `A0` to `A2` have been prematurely emitted, and `A3` is not missing in the stream that is already on the go.

e_i.ts < e_{AEP}.ts. If the time stamp of the new event is smaller that that of the AEP we need to retract the events we falsely emitted to the event detector. Moreover, we also have to retract the events the event detector falsely generated out of falsely emitted events. The former retraction is handled by the technique we introduce in Section 4.4.2. The latter type of retraction is handled by means of the methods that we describe

[2] This may be the case if different events are used to set *clk*.



(a) Inserting A3 behind AEP.



(b) Inserting A3 in front of AEP.

Figure 4.4.: K-slack's event insertion and AEP.

in Section 4.4.3. The AEP is set to e_i, and the events from the new AEP to the old AEP are replayed. In Figure 4.4(b) the event B4 must be retracted, and A3 can be emitted prematurely before replaying B4.

Whenever *clk* is updated the speculative buffer emits all the events that fulfill inequation (4.1). In contrast to dynamic K-slack we do not wait for *clk* to be updated but also evaluate each event that is received, and if it fulfills the inequation we prematurely emit it immediately. Events are purged only if the regular non-speculative K-slack buffer would purge them. Only then we can be sure that we no longer need them for upcoming replays.

Figure 4.5 shows how the speculative buffer for the event detector from Figure 4.3(a) works with an attenuation factor of α=1/3 and an initial K=0. The orientation of the buffer is now vertical, the head is on top. Events of type A are used to set the internal clock clk_A, and K is adjusted by means of dynamic K-slack, see the bold values on top of the sorted event stream. Dashed events will be purged from the buffer at this time step. Prematurely emitted events are in grey. The AEP, although not explicitly visualized to avoid

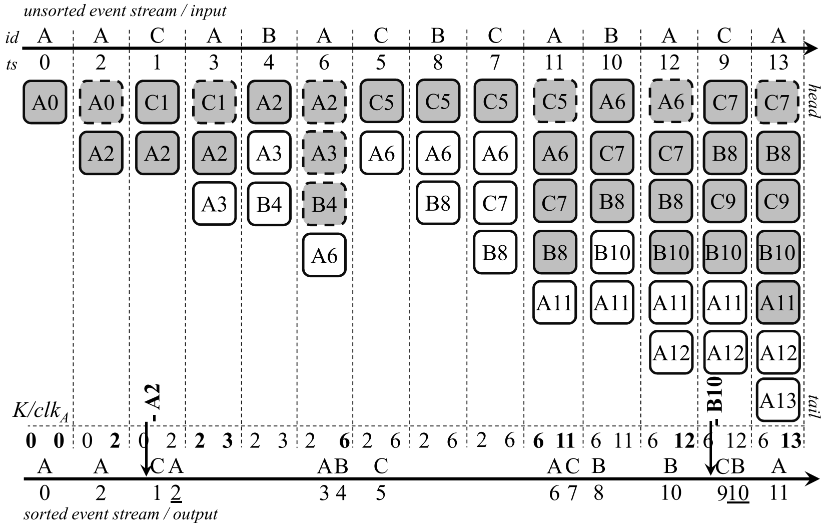


Figure 4.5.: Speculative ordering unit with $\alpha = \frac{1}{3}$.

confusion, points to the lowest grey event. Events that need to be *reversely* processed, i.e., that need to be retracted from the detector, are shown with negative events and with arrows that point to the output stream. Replayed events in the output stream are underlined.

At the beginning, we have no measurements of event delays, so we emit `A0` and `A2` (speculatively) because both $0+0 \leq 0 = clk$ and $2+0 \leq 2 = clk$ hold. With reception of `A2` we can purge `A0` from the buffer. When `C1` is received, we detect that we incorrectly emitted `A2` before, and replay the correctly ordered sub-stream, i.e., we emit `C1`, and again `A2`. The `-A2` event is implicitely handled by the technique we describe in Section 4.4.2. K is updated to 2 as soon as `A3` arrives (because the maximal delay of `C1` is 3-1=2 due to dynamic K-slack). `A3` is not yet emitted, because $e_i.ts{+}\alpha{\cdot}K{=}3{+}\frac{1}{3}{\cdot}2{=}3.67{>}3{=}clk$. `C1` can now be purged as $K{=}2$ tells us that there cannot be out-of-order events e_i with $ts{<}3{-}K{=}1$. We insert `B4` into the buffer. With `A6`, we evaluate the currently buffered events, emit `A3` ($3.67{\leq}6{=}clk_A$) and `B4` ($4.67{\leq}6{=}clk_A$), and purge the buffer by erasing each event that satisfies $e_i.ts{+}K{\leq}clk$, i.e., `A2`, `A3`, and `B4` that are now safe (by means of the rules of pure K-slack). Although

clk is not updated on reception of `C5`, we can prematurely emit `C5` because it fulfills $e_i.ts+\alpha\cdot K=5+\frac{1}{3}\cdot 2=5.67\leq 6=clk$, and since `A6` has not been processed before, we do not need to replay. `B8` and `C7` are both queued until reception of `A11`. We then process `A6`, `C7`, and `B8`, and purge `C5`. With `A12` we prematurely emit `B10` because $e_i.ts+\alpha\cdot K=10+\frac{1}{3}\cdot 6=12\leq 12=clk$. We insert `C9` in front of the AEP, i.e., between `B8` and `B10`, and thus detect another false speculation. We relocate the AEP and replay `C9` and `B10`. Hence, although we reduced detection latency, i.e., the size of the ordering buffer, by around 66% we only miss-speculated two times within this example.

Whenever an event detector generates an event out of a prematurely emitted event, the middleware attaches the original K-value difference (we will call it K^+ later) so that the ordering units of higher level event detectors can calculate the event delay, and hence their new K-value, appropriately, as speculation leads to lower delay measurements on higher hierarchies.

Algorithm 4.1 gives the pseudo code for the insertion of an out-of-order event and the possibly necessary AEP relocation while Algorithm 4.2 gives the pseudo code for the speculative emission, i.e., the replay of events. We describe the details of both algorithms after we introduce the state recovery below as this is an essential part of the algorithms.

4.4.2. State Recovery

If speculation was too hasty the speculative ordering unit relocates the AEP and replays the event stream. However, although in this case the ordering unit signals that events have been mistakenly *emitted* (to the detector) and *generated* (by the detector), and revises incorrect events by means of the retraction methods that we describe in Section 4.4.3, the event detector that processes some of the emitted events may still be in a wrong state due to false-positive events. Hence, for a replay the internal variables of the event detector must be reverted to the state they had before the event detector processed the first incorrect premature event to avoid inconsistencies. This is what we denoted as *reverse* processing before.

However, such a state recovery is difficult because of three reasons. First, since the ordering middleware transparently handles out-of-order events the event detector does not even know that an ordering unit exists. Second, even if the event detector knows that there is a speculative ordering unit, and it processes retraction events to revert its state, it nevertheless has no clue about

α and K, and hence how many states it needs to keep for later state recovery. Third, in many cases retraction cascades, which are the core reason why speculation must be limited, can be interrupted earlier and resolved faster. But this is only possible from within the ordering middleware, see Section 4.4.3.

Our solution is to let the middleware trigger both state backup and recovery. This avoids side-effects by out-of-order events in the detector and the application programmer does not need to care about retraction events or recovery. On demand, the event detector has to be able to provide all the data that is necessary to later on be restored to this snapshot (if this is not provided

Algorithm 4.1: Adding a newly received event *e*.

```
Data: Event e, OrderingBuffer buffer, Mutex m, WorkerThread
      workerThread.
begin
    UpdateK(clk-e.ts+e_i.K^+);                // update K if needed
    while !m.acquireLock() do                 // lock buffer
        workerThread.interrupt();
    BufferIterator it ← buffer.tail;
    repeat
        it ← it.previous;
        if it.GetTime() ≤ e.GetTime() then
            buffer.insertAfter(it, e);
            break;
    until it equals buffer.head ;
    if e.GetTime()<buffer.head.GetTime() then
        buffer.push_front(buffer.head, e);
    if AEP.ts > e.ts then                     // it.next is snapshot
        Event s ← it.next.pop();
        s.SetTime(e.GetTime());               // adjust ts
        buffer.insert_before(it, s);          // move snapshot
        buffer.emit(s);                       // re-init detector
        buffer.SetAEP(it);                    // relocate AEP
    m.releaseLock();
    workerThread.wakeUp();
end
```

by the programming language). The key idea is to ask the event detector for a snapshot whenever a premature event e_i is going to be emitted and to insert this snapshot as an exceptional event e_s with $e_s.ts$=$e_i.ts$ into the ordering buffer, directly in front of the prematurely emitted event e_i. The snapshot then represents the state that has been in place before e_i was prematurely emitted.

Whenever events are replayed to an event detector, the detector switches back to an earlier state as soon as it receives such an exceptional event e_s encapsulating that earlier state. The middleware only emits the first/earliest buffered snapshot event to the detector and skips the remaining snapshots. During a replay, event detectors are also asked for snapshots, and existing snapshots in the ordering buffer are replaced by new ones. Snapshot events are purged from the buffer using standard K-slack like any other events (as the time stamps also denote their lifetimes).

Figure 4.6 shows the ordering unit of Figure 4.5 with snapshot processing. On top of each emitted grey event there is a special snapshot event in the buffer (denoted by s_{ts}) holding the state of the detector before the premature event was processed. Consider the replay situation at reception of `C1` in column 3. The snapshot $\underline{s}_2$ that has been taken when `A2` has been emitted speculatively (column 2) is still in the buffer. For the replay, this snapshot is emitted first, its time stamp is set to 1 (as it is now the state that has been in place before `C1` has been speculatively processed), followed by `C1` and `A2`. We can re-label s_2 to s_1 and take a new snapshot s_2 before we replay `A2` and insert the new snapshot event between `C1` and `A2`. The procedure is similar when `C9` arrives.

If two subsequent snapshots do not differ, we just store a reference to the prior snapshot. Another space optimization is not to take a snapshot on each premature event emission. This saves space but increases the necessary CPU consumption when the event stream is replayed as an appropriate snapshot may only be found earlier in the buffer. Nevertheless speculation needs extra storage. The space needed grows with the degree of speculation and depends on the complexity of the event detector's state space.

Algorithms 4.1 and 4.2 show how the speculation works. We use worker threads that iterate over the ordering units and that are used by the event detectors for event processing (Algorithm 4.2). While such worker threads may be busy with processing events, a new event may be received, and Algorithm 4.1 is called upon reception. For any event we receive, we first calculate its delay and update K by means of classic K-slack, if necessary. Note that as events

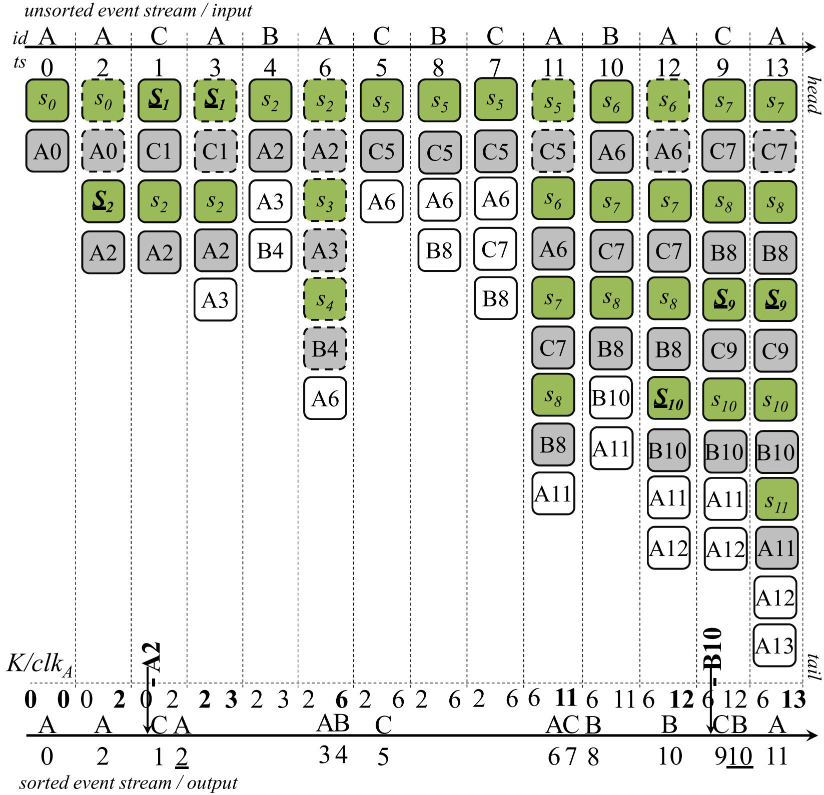


Figure 4.6.: Speculative ordering unit with snapshot recovery and $\alpha = \frac{1}{3}$.

may have been generated prematurely by the lower level detector we have to add K^{+}, i.e., the premature emission time that we attached to the event, on top of the event's delay. Next, since this event may be out-of-order, Algorithm 4.1 acquires the lock on the ordering unit's buffer, i.e., the worker thread that is used to process an event with the event detector will stop *after* this event, inserts the (out-of-order) event at its correct position into the buffer, and reinitializes the event detector and relocates the AEP, if necessary. With its termination it triggers the worker thread to proceed as all of them might be idle at that time.

Algorithm 4.2: Event emission, replay, and buffer purge.

```
Data: OrderingBuffer buffer, Mutex m, Clock clk, AEP.
begin
    while true do
        if m.acquireLock() then
            while AEP ≠ buffer.tail() do
                CheckAndBreakOnInterrupt();
                if AEP.GetTime() ≤ clk - α·K then
                    SnapshotEvent s ← MakeSnapshot();
                    buffer.insert_before(AEP, s);
                    buffer.emit(AEP);
                    AEP ← AEP.next();
                    if AEP.isSnapshotEvent() then
                        AEP ← AEP.next;
                else
                    break;
            // buffer purge by K-slack constraints
            while buffer.head().GetTime + K < clk do
                if buffer.head equals null then
                    break;
                if buffer.head() not equals AEP then
                    buffer.pop_front();
                else
                    break;
            m.releaseLock();
            if not relocated(AEP) then
                m.sleep();
        else
            m.sleep();
end
```

The worker threads in Algorithm 4.2 are not only triggered on AEP relocations but also by *clk* updates, and are also used to purge the obsolete events from the buffer. The worker thread tries to acquire the lock (non-greedy)

and checks, starting from the current AEP position, if particular events may be prematurely emitted/processed, while snapshots are skipped. Afterwards, K-slack is used to clean the buffer.

4.4.3. Event Retraction

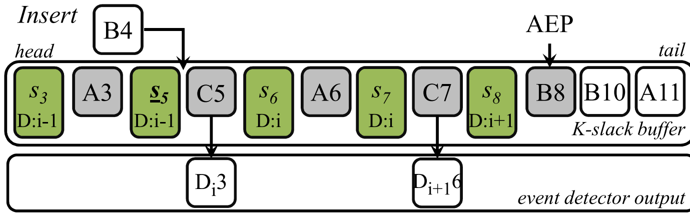
If some event is missing in the speculatively emitted stream, we restore the snapshot of the subscribing event detector and replay the event stream. What remains open is that this event detector may itself already have generated events based on the incomplete event stream. These events must be retracted/eliminated from event streams subscribed by detectors on higher hierarchy levels. Since this may lead to a heavy retraction cascade we must limit the degree of speculation.

Consider the example event detector of Figure 4.3(a) and the speculative buffer in Figure 4.7(a). For this example we assume that the event detector numbers the generated `D` events. The event detector has already speculatively processed the grey events `A3` to `B8` and has already generated D_i3 out of `A3/C5` and $D_{i+1}6$ out of `A6/C7` when an event `B` with time stamp 4 arrives. The ordering unit detects a miss-speculation, and since the subscribing event detector itself has incorrectly generated D_i3, we must restore the event detector's state, replay `C5` to `B8`, and retract D_i3 from the streams of higher level event detectors.

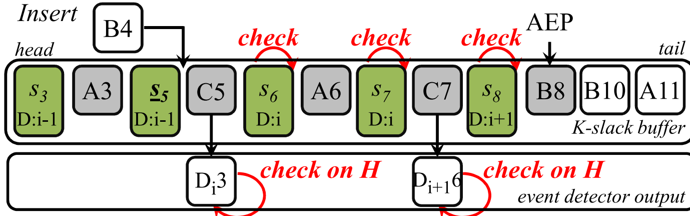
Moreover, $D_{i+1}6$ may be wrong as well because of two reasons. First, the event detector may not have reached the `D`-state in presence of `B4`, because of some internal state variables that are not shown. Second, even if it would reach the `D`-state then instead of $D_{i+1}6$ it should have produced D_i6. Hence, in addition to be able to replay event streams the middleware must also be ready to invalidate events that have been generated based on prematurely emitted events.

The key idea to restore a correct state at a higher level event detector `H` is to send a retraction event that identifies the events that have incorrectly generated so that `H`'s ordering unit can fix that and replay the event stream.

Below we present two techniques to handle event retractions across the detection hierarchy: *Full Retraction* and *On-Demand Retraction*.



(a) D3 and D6 generated prematurely.



(b) On-Demand Retraction.

Figure 4.7.: Event retraction on reception of B4.

Full Retraction

The key idea of *Full Retraction* is to instantly retract all events that may have been generated incorrectly as soon as the AEP is relocated. For this purpose, an event detector's ordering buffer must not only store the prematurely emitted events and the snapshots but must conceptually also hold a list of events that the detector has generated from the prematurely emitted events, i.e., D_i3 and $D_{i+1}6$ in the example. When an out-of-order event is inserted into the buffer, we first collect all events that may have been incorrectly generated, and send a (conceptually) retraction event for each of them to the ordering unit of each subscribing higher level event detector H. When this ordering unit receives such a retraction event it purges this event from its buffer, and performs its own retraction and replay. Hence, a retraction event reuses the replay and snapshot recovery used for out-of-order events. For instance, in Figure 4.7(a) we insert B4 between A3 and C5, instantly send retraction events for $-D_i3$ and $-D_{i+1}6$, tell the event detector to restore the appropriate snapshot, and start the replay.

Although retraction of every single incorrectly generated D event would work, there is a more efficient way to achieve the same effect. The idea is to exploit an event counter i that the ordering unit attaches to the detector's state as it has been done by the event detector before.[3] Instead of sending several retraction events (-D_i3 and -D_{i+1}6 in the example), it is sufficient just to send the event *counter* D:i-1 to the upper level detector. This detector can then purge all D events from its buffer that have a larger counter.

But the event counter not only helps to reduce the number of retraction events that need to be sent to higher level detectors. With the counters stored in the states, there is no longer a need to keep lists to the generated events. In the example, there is no need to store the lists C5->D_i3 and C7->D_{i+1}6. Instead the counter values are piggybacked to the states s. This reduces the necessary footprints.

The advantage of full retraction is that the ordering units of higher level event detectors purge retracted events from their buffers and replay their event streams immediately. If the event detector's state changes and/or the prematurely generated events differ, full retraction works as efficient as possible. Full retraction is essentially the state-of-the-art in the related work.

On-Demand Retraction

With full retraction and its purging of generated events, the detectors have to perform their detection work again. This consumes CPU cycles. But consider state-less detectors that will generate exactly the purged events again. It is obvious that for those detectors most of the retraction work is wasted. The efficiency of full retraction and the achievable degree of speculation strongly depend on the internal structure of the event detector and its generated events.

The key idea of *on-demand retraction* is not to send the retraction events immediately upon AEP relocation. Instead we replay the event stream and only retract events if snapshots change and/or if events are not generated again during replay. In more detail, whenever we emit events during replay we check if the following two properties hold:

(1) Snapshots are equal. If we replay the event stream, and the snapshots do not differ we can abort the replay process. Because the upcoming

[3] The time stamps cannot be used as event counter because the ordering middleware has no influence on their generation and they are not said to be increasing.

premature events in the replay cause the same snapshots and hence generate the same events both the snapshots and the previously generated premature events remain valid.[4]

(2) Generated events are equal. The events and their counters that are generated again during replay are marked as updates, and the ordering unit of the higher level event detector `H` checks if the previously generated premature event equals the recently generated update event. If it does, the ordering unit of `H` does not reinsert the new event, does not relocate the AEP, and hence does not trigger an unnecessary retraction cascade.

Figure 4.7(b) shows on-demand retraction for the example. When `B4` is inserted into the buffer, the event detector is reset to use the snapshot $\underline{s}_4$=$\underline{s}_5$, and works on the replayed events. The event detector will reach some state s_5' after processing `B4` speculatively (not shown in Figure 4.7(b)). If $s_5'=\underline{s}_5(=\underline{s}_4)$, i.e., if the state of the event detector is not affected by `B4`, we can abort replay and retraction because all the subsequent snapshots and the prematurely generated events will remain unchanged.

However, if the state of the event detector is affected, we replay the event streams, and whenever an event is generated, we set the *update* flag before we send it to the ordering unit of `H`. The ordering unit of `H` then checks the updated event with respect to equality and discards it if there is no change.

If the event detector is state-less or events that cause a replay do not change much of the output, on-demand retraction considerably reduces retraction work that would be introduced by full retraction across the detector hierarchy. See also the evaluation in Section 4.6.

4.4.4. Runtime α-adaptation

Speculation uses additional system resources to reduce event detection latency. The remaining question is how to set the attenuation factor α that controls the degree of speculation. The ideal value of α results in best latencies but also avoids exhausting the available system resources and hence system failures. However, a static α-value hardly results in good latencies and resource utilization for the overall system runtime. Hence, due to the varying

[4] We assume that event detectors solely process events emitted by the middleware. Other input, e.g., system calls or file handles, are not allowed.

system and event loads, α must continuously be adapted over time to fit the currently available resources to the current event load.

We now present a runtime α-adaptation algorithm that achieves this goal. It is safe because when either no runtime measurements are available or when a critical situation occurs, e.g., a CPU load that is higher than some threshold, α is (re-)set to its initial value α_0=1 in order to prevent a system failure. Moreover, α-adaptation only has a local effect because α only effects replay and retraction of a single ordering unit. The CPU load on machines that run other detectors are not affected much because the speculative buffer of an upper level event detector only inserts and/or retracts events but does not start a replay (which in general depends on its own α).

The key idea of our runtime α-adaptation is to use a control loop similar to the congestion control mechanism known from the transmission control protocol (TCP) [6]. Congestion control tries to maximize throughput by doubling the data rate, i.e., the congestion window size that holds the number of to-be-acknowledged packets, at each time unit. When the data rate becomes too high for the network link, data packets are timed out because packets are probably lost in the network, and the window size is reduced to 1. The maximal window size is saved and a new iteration begins. The window size is doubled again until it reaches half the window size of the previous iteration. Afterwards the window size is incremented until again packets are lost and the next iteration starts over.

To adapt that idea, we use α as the congestion window size, and the CPU workload as a load indicator. Whenever we evaluate the CPU load we adjust the value of α. To measure the CPU load accurately, the middleware repeatedly (after each time interval t_{span}) sums up the times that event detectors need for event processing, i.e., t_{busy}, and relates it to the sum of the times the worker threads have available, t_{span}. The resulting busy factor b_c is

$$b_c = \frac{t_{busy}}{t_{span} \cdot \#workers}.$$

For instance, with one worker thread and a time interval t_{span}=0.5s and an accumulated t_{busy}=0.41s the busy factor b_c is 0.41s/0.5s=0.82. This means that 82% of the available resources are used and that about 18% of the system resources are still available (assuming that no other processes interfere with the EBS). The busy factor grows with decreasing values of α, see also Section 4.6.2.

To adjust α we specify a target zone $[b_l; b_u]$, i.e., an interval of the *lower* and the *upper* target values for b_c. If the busy factor is below b_l, CPU time is available and α is decreased by halving its value. This is similar to doubling the congestion control window size. α is the inverse of the window size and halving α increases the degree of speculation. If the busy factor grows above b_u, CPU time becomes critical, and the current α is kept (called α_{best}) before α is set to its initial value α_0=1. From there α is again halved until its value is about to be set below the bisection line (1-α_{best})/2. From then on α is lowered in small steps of α_s to slowly approach the target zone where the best latencies are achieved. Algorithm 4.3 gives the pseudo code for the α-adaptation.

Our evaluation shows that a busy target interval of [0.8; 0.9] in combination with α_s=0.05 works reasonably well. Of course, b_c is not only affected by the choice of α. In burst situations or when a detector reaches a rare or slow area of its state space, t_{busy} may peak. In such cases, the runtime α-adaption reacts appropriately by resetting α and hence by giving more resources to the detector. The presented algorithm only works properly if the delays in the

Algorithm 4.3: α-adaptation.

Data: α, *lastMinimum*, **bool** *slowmode*, b_l, b_u, b_c
begin
  if $b_u < b_c$ **then** `// reduce speculation`
    lastMinimum ← α;
    α ← 1;
    slowmode ← false;
  if $b_c < b_l$ **then** `// increase speculation`
    α_{target} ← 0;
    if *slowmode* **then**
      α_{target} ← α-0.05;
    else
      α_{target} ← $\alpha/2$;
      if *(1- lastMinimum)/2* > α_{target} **then**
        slowmode ← true; `// goto slow mode`
        α_{target} ← α - 0.05;
    α ← α_{target};
end

network are normally distributed (what they usually are) or if the distribution is heavy-tailed (as the sensor events in our locating system). Otherwise, the adaptation function needs to be adjusted to fit the distribution.

4.5. Implementation Details

In order to increase efficiency we implemented the presented techniques in a slightly different manner. Similar to Section 3.3.7 we describe the major differences and provide justifications.

Separation of ordering unit and scheduler. In order to stop the worker threads that are currently used for event detectors that process events triggered from a single ordering unit, we need to combine the scheduler and the ordering unit (contrary to the implementation in the previous chapter). This is because we need to invalidate each event that is prematurely emitted but not yet processed. These events need to be deleted from the scheduler's ready list if the ordering unit's AEP is reset. This is only possible if the ordering unit has direct access to the elements in the scheduler. Otherwise the ordering unit would also require sophisticated data structures to quickly identify the events that need to be removed from the scheduler's ready-list that also holds events from other ordering units and event detectors. Basically, the implementation of the speculative buffering is now similar to that in Algorithms 4.1 and 4.2.

Snapshotting of event detectors when receiving position elements. As we illustrated in Section 3.3.7 we need to take special care about the ordering of the high data rate position sensor streams. The performance issues of the insertion sort of the events are still present. The solution was to order the position stream events individually within an external ordering buffer. While this was the best escape to fit the performance requirements it results in a problem concerning the inserted snapshots in the ordering units. The snapshot and recovery is based on the fact that both events and the detector's snapshots are inserted into the same ordering buffer. But as the position stream events are located in an external ordering buffer, which is accessed by several ordering units, we cannot just insert the snapshot events there.

The solution is to insert pseudo events into the ordering units. These events are not emitted to the detector but are only used to hold a pointer to the position streams' ordering buffer, and to insert the snapshots at their appropriate positions in front of the pseudo events. At the same time the way we create those pseudo events can be used to trade memory for processing time: if we only insert a pseudo events every few position stream emissions we can save space needed for snapshots by processing more position stream events than necessary in case of replays.

Unsorted event streams. The event detector API, see Section 2.2.3, enables the the developer to subscribe particular event types out-of-order. This means, that events of these types are not inserted into the buffer but bypass the ordering unit. Hence, they are directly provided to the event detector without delay.
However, this causes problems if the event detector is reset as a result of AEP relocation. Out-of-order events that have been subscribed out-of-order, may have already been processed by the event detector. If the event detector is now reset to some previous state, i.e., a state that has been active before these unsorted events have been processed, the out-of-order events remain unnoticed in the upcoming replay (since they have not been inserted into the ordering unit's buffer). For the event detector it looks like these events did never actually happen. On the other side, we cannot simply process the events again (by default) as we do not know if the events are already part of the state that is restored. To provide a consistent way of processing we omitted the functionality of subscribing events out-of-order.

Memory corruption. A critical situation occurs when the event detector's memory is corrupted by incorrect event order. Consider the example event detection from Section 2.2.3. When `EV_B` is processed before `EV_A` the event detector accesses invalid memory space and the system fails. In the event detectors that are implemented for our middleware system this does not happen. However, since we may not assume that by default the solution is to set up a signal handler that catches the exception and resets the system state. We may want to stop speculative processing for event detectors that are prone to segmentation faults as a cause of premature event emission.

4.6. Evaluation

For the evaluation we have analyzed the position data streams from our Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. We used the same setup with 144 transmitters and 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Each player has four transmitters, one at each of his limbs. The sensor data consists of absolute positions in millimeters, velocity, acceleration, and Quality of Location (QoL) for any direction [129], see Section 2.4. The sensor data arrives out-of-order as shown in Section 3.4.1.

Since soccer needs these sampling rates and events happen within the fraction of a second a low latency is required so that a hierarchy of detectors can help the human observer, for example a reporter, or a camera system that should smoothly follow events of interest, to instantly work with the live output of the system or to automatically trigger actions.

We present results from applying our event processing system and our algorithms on the position data streams from the stadium. Our platform consists of several 64-bit Linux machines, each equipped with two Intel Xeon E5560 Quad Core CPUs at 2.80 GHz and 64 GB of main memory that communicate over a 1 Gbit fully switched network. For our tests we organized a test game between two amateur league soccer clubs and processed the incoming position streams from the transmitters.

For all benchmarks we replay position stream data in our lab's computing cluster. To focus on the experimental results more clearly, we perform all the work on just one machine. Our event processing middleware, i.e., the methods for speculative processing, pub/sub-management, etc., take around 17,000 lines of C++ code. On top of that we implemented over 70 event detectors with more than 16,000 lines of C++ code that are used to detect more than 750 different event types. The event detection hierarchy has 15 levels, and we replay a snippet of the event stream from the soccer match. The duration of the test event stream is 65 seconds and consists of 2.2 million position events plus 25,000 higher-level events that are generated by the event detectors (not including prematurely emitted events or retraction events). The data stream also incorporates some burst situations. The average data rate of the processed data streams is 2.67 MBytes/sec.

We let the event detectors work on the data streams, and discuss the generated results. For all our experiments we use only one worker thread to

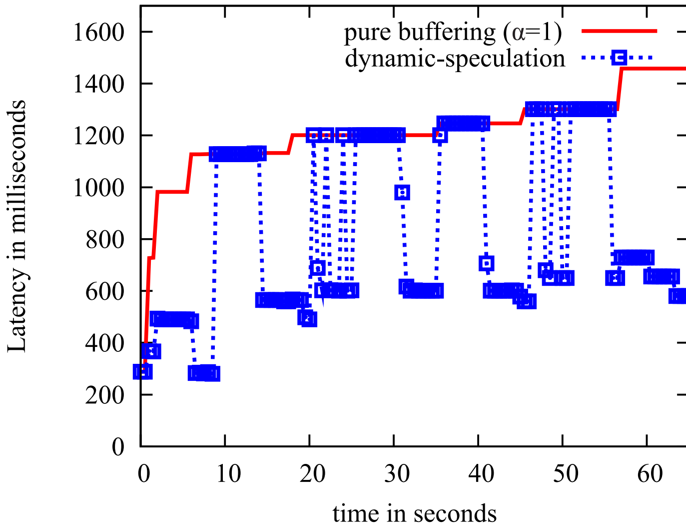


Figure 4.8.: Latencies of (speculative) buffering.

make the results more clear and to avoid side-effects. Section 4.6.1 shows that speculative processing can considerably reduce both latencies and detection delays. Section 4.6.2 evaluates the α-adaptive speculation in detail. Section 4.6.3 and 4.6.4 provide measurements on the resource consumption during the event stream replay versus full speculative and pure buffering approaches.

4.6.1. Latency reduction

Figure 4.8 shows that added speculation can significantly reduce latency of buffering middlewares. For the benchmark we measured the latency of the *pass* event detector. This detector subscribes to 6 different event types and detects a (unsuccessful) *pass* event.

When we replay the event data stream to a pure dynamic K-slack buffering (α=1, straight line) it updates K upon ordering mistakes and finally ends up with a detection latency of 1458ms at the end of the stream replay. The average latency for the 65 seconds is 1276ms. In contrast, our dynamic α-

adaptation reaches a much smaller detection latency (dotted line). At the beginning, α starts at α_0=1 and around 290ms of latency. As the CPU is not fully loaded α-adaptation switches on speculation. This brings detection latency down to 280ms, where pure buffering already has 1128 ms latency. Note that the latency of pure buffering increases much more than that of dynamic speculation as α-adaptation outweighs the growth of the K-slack buffer. At several points the event streams, and hence the busy factor b_c burst. That causes α to back off, leading to a higher detection latency again. Afterwards, α-halving resumes and the latency approaches its minimum again. The average latency of the dynamic speculation is below 800ms, i.e., about 40% better than what pure buffering can achieve.

These results show that our speculative buffering technique strongly reduces the detection latency of events at runtime. Throughout the entire 65 seconds the CPU load was tolerable, and at the critical burst situations α-adaptation can avoid system failures, see Section 4.6.2. Hence, camera movements can be triggered about 40% faster than with pure buffering techniques. The latencies of other event detectors behave similarly.

We did not show the latency of full speculation (α=0) because it consumes too much CPU power and causes event processing to fail. See Section 4.6.3 for more details. Other variants with static α-values would just result in latencies that represent the original latency divided by the static α-value.

4.6.2. Runtime α-adaptation

In the above measurements there are several bursts where α has to back off due to high system loads. To discuss the performance of our α-adaptation in more detail, let us zoom into the first 20 seconds of the event stream, see Figure 4.9.

With a busy factor target zone [0.8; 0.9], we start from α=1 (straight black line). We evaluate and (possibly) halve α every t_{span}=0.5 seconds, and as a result the busy factor b_c (dashed red line) grows. After 7 seconds α is 0.125, we cease halving, and b_c stays in its target zone between 0.8 and 0.9. For a while, both the busy factor b_c and the CPU load (dotted blue line) fluctuate within the target zone before a growing frequency of incoming events requires more and more detection work. After 8.5 seconds into the benchmark b_c reaches 0.95 (the CPU load at that time was at 91%), α is immediately reset to 1, and as a result both b_c and the CPU load drop instantly. Then α is evaluated

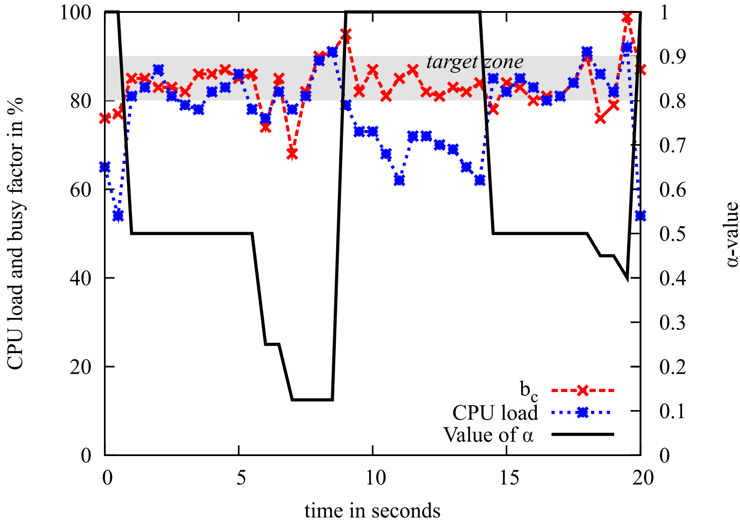


Figure 4.9.: Adaptation of α.

and after 16 seconds α-halving starts over. But this time halving stops at the bisection line (1.0-0.125)/2=0.43. From there α takes small steps of α_s=0.05 per t_{span} to bring b_c and the CPU load back into the target zone.

The α-adaptation algorithm not only decreases α to efficiently use the available CPU power but also rapidly stops speculation if the system is almost overloaded in burst situations, to avoid system failure and to absorb the bursts. Moreover, b_c is a sufficiently good indicator of the CPU load. Only if the CPU load is low, b_c slightly overestimates because of task switches and thread sleeping.

4.6.3. Resource Consumption

To measure the resource consumption we replay the event stream snippet three times. Figure 4.10 shows the resulting CPU loads. We recorded the CPU loads for pure K-slack buffering (α=1, straight red line), dynamic α-adaptation (dotted blue line), and fully speculative processing (α=0, dashed purple line).

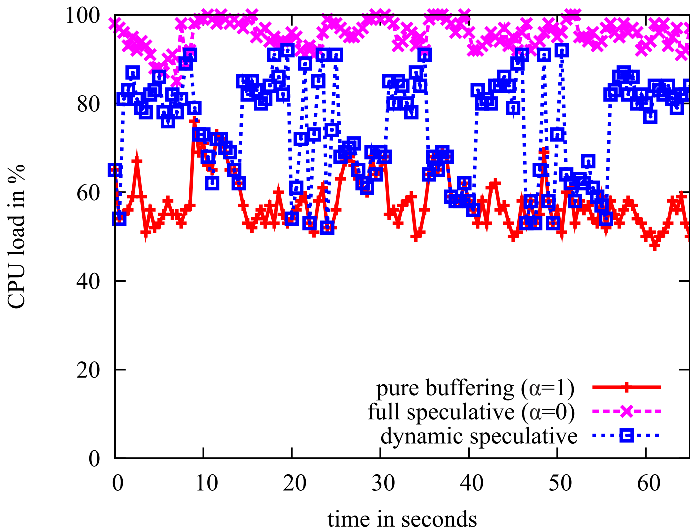


Figure 4.10.: CPU loads with varying α.

Pure buffering exhibits a comparatively lazy CPU load of 50-70% for the entire 65 seconds. That is because events are buffered for a sufficiently long time and the event detectors neither receive retraction events nor are they asked for snapshots or get restored. In contrast, full speculation causes a high CPU consumption above 90% for the entire 65 seconds. In the benchmark the CPU consumption reaches 100% a few times. This is prohibitive because event detection then takes longer than it should. The resulting higher delays cause a ripple effect since the K-values of the detectors further up the detection hierarchy are not prepared to deal with the extra delay and purge events or process them out-of-order. Hence, detectors may get stuck in invalid states and event processing fails. Even worse, there is another reason for system failure: when event processing is slowed down, the queue of incoming events that await being processed often outgrows the available buffer space.

Figure 4.10 also shows that to achieve the good detection latencies discussed in Section 4.6.1 the dynamic speculation makes reasonable and efficient use of the available resources. The CPU load stays within a non-critical target zone that gives the dynamic α-adaptation enough room to react to bursts

and to avoid system failures. Hence, if detection latencies in an application are too high, one simply needs to use a stronger CPU. Then there is more room for speculation and hence latency reduction.

Speculation also needs more main memory. On our benchmark, the pure buffering took only 1,120 KBytes of buffer space, averaged over the 65 seconds. There are two reasons for the demand being that small. First, only the events but not the detectors' snapshots need to be stored. And second, since an event is usually input to more than one detector we store it only once and insert a reference to it into the ordering units. In contrast, as an event detector's state has an average size of around 800 Bytes, full speculation required 21,100 KBytes for an event detector's buffer on average. Dynamic speculation is better and only took 14,850 KBytes on average, i.e., with only 14 MBytes of additional memory per detector we reduce latency by 40%.

Throughout the 65 seconds we took around 4,500 snapshots. An ordering unit holds 365 valid snapshots in its buffer on average. States of lower level event detectors are considerably smaller (below 50 bytes) than those from higher level detectors (one unique detector has a state size of 800 KBytes). Lower level detectors are snapshot more frequent than higher level detectors since the event load is considerably higher.

4.6.4. Evaluation of Retraction Techniques

To evaluate the two retraction techniques we replay the real event stream from the first half of the soccer match (45 minutes). For each of the two retraction techniques we record the number of events that are prematurely generated and retracted from a *player-hits-ball* event detector.

On-demand retraction is more efficient and in general also helps reduce latency. Over 45 minutes, full retraction affected 5,623 ball hit events, whereas on-demand retraction only had to work on 735 events. With full retraction, the resulting retraction cascade affected 117,600 events across the hierarchy, compared to 12,300 events for on-demand retraction. To explain the advantages of on-demand retraction, more than the event counts need to be considered. Compared to full retraction it is more costly for on-demand retraction to process snapshots and to check for state changes. More work takes more CPU power and α-adaptation selects a lower degree of speculation that results in a higher latency. On the other hand, the retraction events of full retraction are quicker to deal with – but there are so many of them.

On-demand retraction works best for event detectors with small states or with states that do not change much (or not for every single event). For example, event detectors that directly process sensor data or low-level events. For such detectors, on-demand retraction can (1) abort the retraction cascade from the bottom of the event hierarchy, and (2) check detector state changes quickly because snapshots are small or not even necessary.

But there are rare event detectors that work better with full-retraction. The *pass* detector is an example. First, it changes its state on almost every event received. Thus, all its checking of snapshots and of generated events is useless work. Second, full retraction only affects 819 events. On-demand retraction can only reduce that number by 27 which is not enough to amortize its cost.

In total, choosing on-demand retraction *for all detectors* is better than full retraction. In the benchmark averaged over all detectors the α-value of full retraction was 0.81 whereas on-demand retraction only achieved an average α-value of 0.69. Nevertheless, on-demand retraction achieved a 15% better detection latency than full retraction. It is future research and an optimization problem of its own to automatically deduce and assign the best retraction technique *for each individual detector* at runtime. That might improve performance even more.

4.6.5. Discussion

We first showed that our speculative buffer with dynamic α-adaptation considerably reduces latency. We zoomed into the details and proved that our method exploits available system resources progressively until we reach a predefined level. We analyzed the resource consumption of our approach and compared it to full-speculative as well as buffering approaches and proved we achieved a perfect trade-off between latency reduction and resource consumption.

We deliberately did not present results on the efficiency of taking and restoring snapshots. While memory consumption has shown to be moderate but tolerable, the time for taking snapshots was too small in our benchmarks to matter at all. Hence, there is also no need for more sophisticated mechanisms like transactional memories etc. However, depending on the event detectors' states and the events this may be an option for improvement for other applications.

4.7. Conclusion

The speculative buffer extension presented in this chapter achieves reliable and low-latency event processing under the predominance of out-of-order events. Any buffering middleware can use the speculation to process buffered events earlier in order to reduce detection latency by exploiting available CPU and memory resources whereas conservative buffering approaches can natively not use them. The speculation does not need any a-priori knowledge of event delays nor the internal description of the event detectors and the system adaptively adjusts the degree of speculation at runtime.

An evaluation of the presented methods on position data stream from a Real-time Locating Systems (RTLS) in a soccer application shows that our dynamic speculation outperforms other speculative and buffering techniques. On average, it reduces latency by 40%.

Future work will refine the α-adaptation to incorporate event loads, latencies, and out-of-order delays per event detector more specifically as it is currently only set for each node individually. Moreover, we will investigate approaches that automatically deduce and assign the best retraction technique for each individual event detector at runtime. See Section 7.3 for more details.

5. Reallocation in Distributed Event-Based Systems

By means of the low-latency ordering techniques described in Chapters 3 and 4 the EPS detects events with low delay without a need for a-priori configuration of ordering parameters.

Although the previously mentioned approaches have specifically been developed to tackle problems of distributed systems they also solve the equivalent problems for a single-node processing. Moreover, for a concise and clear evaluation of these methods we even focused on single-node experiments to make the results more clear.

This chapter explicitly focuses on latency optimization and reliability issues that arise in distributed contexts. For that we first describe a novel method to migrate event detectors at runtime while preserving the low-latency ordering constraints in Section 5.1. Using this runtime migration we are able to adjust poor detector-to-node allocations or free a node that needs to be shut down for maintenance. To reduce latency that is introduced by naive assignments of event detectors over the available nodes we analyze the detector dependencies and data loads at runtime, and migrate detectors to improve performance in Section 5.2. Section 5.3 closes this chapters and draws conclusions.

5.1. Runtime Migration of Event Detectors

Runtime migration has been widely adopted to achieve several tasks such as load balancing, performance optimization, and fault-tolerance. However, existing migration techniques do not work for event detectors in distributed publish/subscribe systems that are used to analyze sensor data. Since low-latency time-constraints are no longer valid they reorder streams incorrectly and cause erroneous event detector states.

This section presents a safe runtime migration for stateful event detectors that respects low-latency time-constraints and seamlessly orders input events correctly on the migrated host. Event streams are only forwarded until timing delays are properly calibrated, the migrated event detector immediately stops processing after its state is transferred, and the processing overhead is negligible. On our Real-time Locating System (RTLS) we show that we can efficiently migrate event detectors at runtime between servers where other techniques would fail.

5.1.1. Introduction

Runtime migration of system components is the method of choice for load balancing, performance optimizations, fault tolerance etc. [91]. But existing solutions do not work well for distributed event processing systems that are used to analyze high data rate sensor streams with low latency. In such systems input data streams usually have a data rate of several thousand events per second, sources may be arbitrarily spread, and events arrive massively out-of-order.

Consider the distributed system in Figure 5.1. An event detector that runs on host N_3 subscribes to four events, for instance sensor readings, namely `A`, `B`, `C`, and `D`, each generated at a different point in the distributed environment. For simplification, these events are detected with zero delay. However, for further processing on host N_3 they will be received with different delays. For

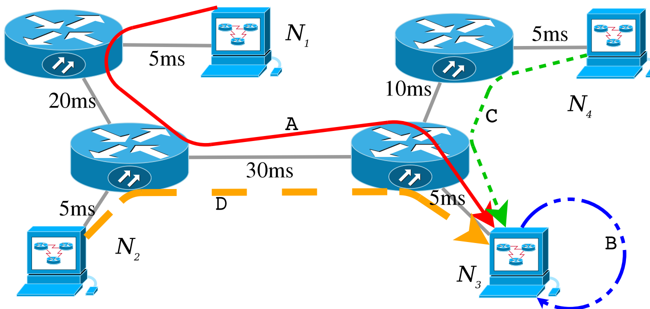


Figure 5.1.: Latencies in a distributed EPS.

instance, A (5ms+20ms+30ms+5ms=60ms) and D (5ms+30ms+5ms=40ms) have much higher delay than C (5ms+10ms+5ms=20ms). B has the lowest delay (0ms) since it is detected by a sensor reading device that is directly connected to the same host and does not travel through the network. The event detector on N_3 measures these delays at runtime and reorders the events into a totally ordered event input stream. The size of the reordering buffer is selected as low as possible to guarantee event detection with lowest latency, see Chapter 3. The generated event is detected with at least 60ms delay and may be subscribed by other detectors on another host for further processing.

To migrate this event detector from N_3 to N_4 typical migration approaches take a snapshot of the state of the event detector, transfer it, activate it on N_4, and terminate it on N_3. The problem is that the state includes the measured event delays and the size of the reordering buffer, both of which are affected by the migration. After migration B now has a non-zero delay, the delays of the events A and B increase by at least 10ms, and C does no longer have a significant delay (since now it is a local event). Hence, the previously measured event delays for correct stream reordering are no longer valid, and although the migration itself worked the system fails since the event reordering is corrupt which often results in incorrect states. The buffers for the reordering units of the affected event detectors are too small for correct ordering, i.e., The K-values. Moreover, in a hierarchy of event detectors an upper level event detector that subscribes the event generated by the migrated event detector may also see out-of-order events as its delay may also have grown.

Other typical migration approaches are based on stream forwarding, leave the reordering unit behind on N_3, and forward the correctly ordered events to the migrated event detector on the target host N_4 until it properly calibrated its own reordering unit. If certain events only occur sparsely, forwarding may be needed for a long time before the new host itself can reorder the event streams. The networking overhead would be high.

In this section we present a technique that transparently reorders out-of-order events at the new host where existing approaches inevitably fail. For that we meet the following requirements. First, the forwarding of events must be kept at a minimum to avoid high network load. Second, the old and the new event detector may not run in parallel and the old event detector must immediately be shut down to reduce processing load. Third, after the state has been transferred the new host must derive correct parameters for its ordering unit to achieve correct event detection.

The rest of this section is organized as follows. Section 5.1.2 reviews related work. Section 5.1.3 then presents the details of our migration algorithm. We carefully consider delay differences and guarantee an in-order event processing at all times. We evaluate our method under real-life conditions in Section 5.1.4 and discuss the runtime migration when processing position sensor data from a Real-time Locating System (RTLS) in a sports application.

Runtime migration is needed for this use case. Assume that we locate players in a soccer game and we apply event-based processing on the position streams. Events such as ball hits, goals, or fouls are automatically detected by the EPS and used to assist referees or control camera control systems. Soccer rule violations such as handball, fouls, etc. are punished more severely if players are inside the penalty area. There we not only need to process the players' position events in more detail but the event detectors are computationally intense as, for example, they (try to) derive the players' intentions. Hence, CPU loads get unbalanced and the system fails if EDs are not migrated soon enough, see. Sec. 6.3

5.1.2. Related Work

Recent related work on runtime migration is mainly found in the area of virtual machines (VMs).

CR/TR-Motion [91] uses checkpoint/recovery and trace/replay to achieve a fast migration of VMs. Checkpoints from the source VM are recovered at the destination, and call traces from the source are replayed so that both machines are consistent. However, the authors do not forward data and do therefore not consider that the order of incoming commands may be different at the new host.

Bradford et al. [26] deal with the transfer of a local persistent VM state. After migration, network connections are redirected to the new host and commands from old connections are forwarded. The old VM is shut down as soon as all the old clients are gone. However, in contrast to our approach, both machines not only have to run in parallel while commands are being forwarded, but the order is ignored in which commands are received over the network.

MOSIX [19] is a cluster management system that supports interactive processes and resource discovery for workload distribution. As MOSIX migrates processes and redirects system calls it has the same disadvantages as the previous approaches.

Pipelined State Partitioning (PSP) [132] time-slices stateful operators, i.e., multi-way window-based join operations, and then distributes the fine-grained states over a cluster to form a virtual computation ring. The states are relocated if CPU loads are unbalanced. However, although the operator itself is stateful, the states are not. Moreover, the authors assume that input streams are equally ordered on each host.

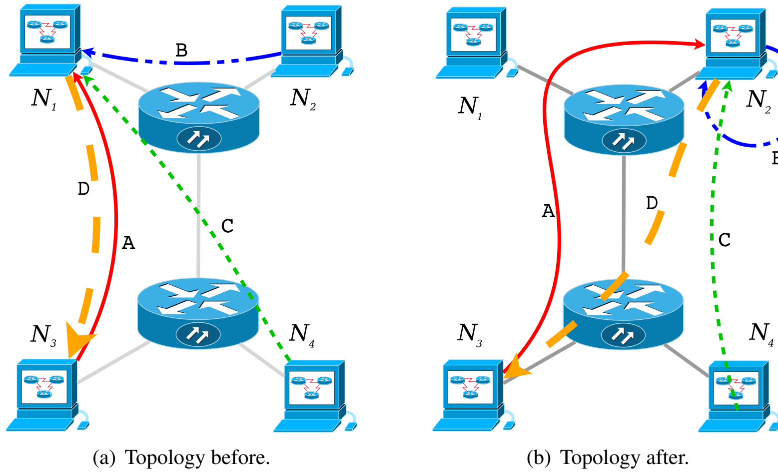
Liu et al. [90] combine *state spill* and *state relocation*, and use decision making to use one of them. However, both do not work if events need to be reordered.

Endler et al. [55] provide a comparison of various handover techniques for mobile devices from which *new/old domain service* ensures a total order. However, both methods forward the complete event stream until the new service takes over.

Koldehofe et al. [81] and Ottenwälder et al. [107] use migration plans to migrate operators at predefined points in time, and hence minimize delay and network traffic in distributed systems. While they do not introduce any downtime or overhead processing their ordering adaptation is based on time markers that are periodically embedded into the stream. Event detectors need to buffer any events between markers for a correct order between pairs of events. This introduces high latency.

Most of the previous approaches do not consider the order of incoming commands and/or data explicitly. That is because usually the source of the commands, i.e., the user's workstation, is static and commands are still received in correct order. However, if we deal with multi-user VMs, problems may occur if two users try to modify the same file. At the original VM, user A's command may be received first, whereas at the migrated VM, user B's command will be first. The VMs are then out-of-sync. Approaches like CR/TR-Motion would repeat the recovery and replay process in such unlikely situations. However, for event detection such situations are very likely and migration would probably result in endless recovery and replay cycles.

Notice that the migration of an event detector not only affects the input delays of the migrated event detector but may also affect the input delays of upper level event detectors that subscribe events generated by the migrated event detector although the subscribers do not participate in the migration. Sudden increases of K may lead to out-of-order processing on higher level event detectors. The migration algorithm presented here also addresses this problem.



(a) Topology before. (b) Topology after.

Figure 5.2.: Event detector migration.

5.1.3. Runtime Migration

As an event detector stores status information, its migration requires to send the state and to use it for initialization of the new event detector. This is similar to virtual machine migration, see Section 5.1.2, and is not discussed here any further.

Consider the network topology depicted in Figure 5.2. Before migration (Figure 5.2(a)) an event detector runs on host N_1 and subscribes three events `A` (published by N_3), `B` (published by N_2), and `C` (published by N_4). It generates event `D` (subscribed by N_3). When we migrate this event detector from N_1 to N_2 (Figure 5.2(b)), the sources of the subscribed events remain the same, but their delays most definitely change. The delay of `B` at the new host N_2 shrinks, because after migration `B` is a local event. The delays of `A` and `C` may or may not shrink. Note that the delay of `D` at N_3 may also change even though the subscriber at N_3 does not participate in the migration.

Unless delays shrink for all involved events, a naive migration is likely to fail because the migrated event detector (or any event detector on a higher

level of the hierarchy) no longer sees the subscribed events in correct order as its K-sized ordering buffer that worked well on the old host is too small for the new host.

As discussed before naive migration approaches cannot be used to guarantee ordered input events for the migrated event detector. Because often CPU overload or buffer overflow trigger migration we cannot run the new event detector concurrently to the old one until the new one has configured its K-value. Also forwarding the reordered events from the old to the new host is prohibitive as it may cause high network loads and processing overheads for a long time if particular events occur sparsely.

In the following, we present an algorithm that migrates event detectors at runtime and initializes their K-values according to the timing delays at the new host. The introduced latency is negligible. Never during the migration there are two event detector instances that consume CPU time. And most importantly, both the old and the new event detector instances as well as upper level event detectors see in-order events at any time. The old event detector stops as soon as its state has been copied, and the networking overhead is minimal.

Cooperative Handover

The key migration idea is a cooperative handover in which the new host not only subscribes the necessary input events from their sources, but the old host also forwards those events to the new host. The new host can then derive the correct order by combining delay information from events that arrive along two paths. Our algorithm consists of two different, interleaved steps: (1) migration and (2) delay adaption and echo cancellation. Figure 5.3 depicts a sequence diagram of the cooperative handover. Below we explain the two steps in detail by taking up the example from Figure 5.2.

Step 1: Migration. When we migrate an event detector from N_1 to N_2, we first need to move the event detector's state. However, the movement must fulfill certain requirements. First, as any downtime of the event detector may add delay to the generated events it must be as short as possible. Second, since the event detector is continuously processing events, we cannot just terminate it and move it. Instead, when we close an event detector on one machine, we need to restore the *correct* state on another machine.

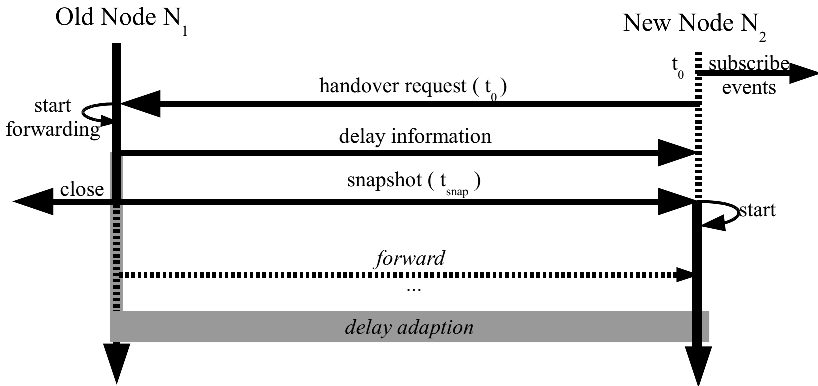


Figure 5.3.: Steps of the cooperative handover.

To implement the migration at first (t_0) the new event detector on the target host N_2 subscribes the required events A, B, and C, and then sends a handover request to N_1. N_1 responds with two packets. The first packet holds the current delay information for each subscribed event. For example, let N_1 responds with the delays δ(A)=30ms, δ(B)=10ms, and δ(C)=20ms. The packet also holds the current time stamp ts so that N_2 can calculate d_f=clk-ts, i.e., the sub-delay of forwarding an event to N_2. Let d_f be 5ms for our example.

The second packet is the snapshot of the event detector. Since local clocks may vary between both machines, N_1 must ensure that N_2 buffers all events so that it can recover the correct state from this snapshot. If N_1 did not process an event since t_0, its current state is used for the snapshot and t_{snap}=t_0. Otherwise t_{snap} is set to the occurrence time stamp of the last processed event. After sending the snapshot, N_1 terminates the event detector as N_2 will take over.

For instance, at t_0=clk=100 N_2 sends the handover request to N_1. As N_1 is continuously processing events, it may already be busy with an event with time stamp 180. When the handover request arrives N_1 takes the snapshot of the event detector, sets t_{snap}=180, sends the packet to N_2, and terminates the event detector. N_2 sets the state of the event detector and processes any buffered events with a time stamp above 180 (only).

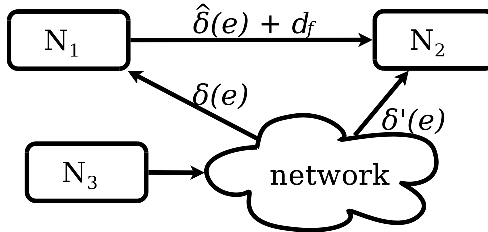


Figure 5.4.: Delay $\delta(e)$ of event e before migration, $\delta'(e)$ after migration; d_f is the forwarding sub-delay.

Step 2: Delay adaption and echo cancellation. With the above migration step we can correctly move a running event detector from N_1 to N_2. Nevertheless, at N_2 event delays and therefore suitable K-values may be different, as the delays of the subscribed events may have changed. Instead of starting with a fresh K'_D=0, it is initialized according to the delay information received from N_1 in the first packet, i.e., K'_D = max(δ(A)+d_f, δ(B)+d_f, δ(C)+d_f) = max(30+5, 10+5, 20+5) = 35ms.

As shown in Figure 5.4 the new host receives all events twice. An event e reaches N_2 directly with a delay $\delta'(e)$ and it also reaches N_2 with a delay $\delta(e) + d_f$ because it is forwarded by N_1. As soon as N_2 receives an event along the direct route, it can update its K by using $\delta'(e)$ instead of $\delta(e)$+d_f. For instance, a delay of δ'(A)=25ms for the first directly received A reduces K_D to max(**25**, 15, 25)=25ms.

While receiving events twice is advantageous for initializing K and for setting up the reordering unit, echoed events would pose problems for the event detector. To make sure that the event ordering unit only sees an event once, one of them needs to be dropped so that only one event is in the ordered input stream of the event detector. The echo cancellation works as follows: for each event type there is a first time when the new host sees both an event and its echo. Before that moment, events with lower delay are passed along the ordering unit (and late events are dropped).[1] Afterwards, as soon as the direct

[1] Actually, these events have the same sequence number and hence they are dropped by the middleware anyway.

Algorithm 5.1: Delay Adaption and Echo Cancellation.

```
Data: ED_new, d_f, DelayList delays, EventList echo
begin
    for delay d : delays do
        d ← d + d_f;
    ED_new.K ← K ← max(delays);                          // set K
    Event e;
    while e.receive() do
        if e.id ∉ ED_new.GetSubscriptions() then
            continue;
        if e.isDirect() then
            delays.at(e.id) ← clk-e.ts;
            K_n ← max(delays);
            if K_n ≠ K then
                ED_new.K ← K ← K_n;                       // retro-fit K
                NotifyDelayChange();
            StopForward(e.id);
        if echo.contains(e) then
            echo.erase(e);                                // duplicate.
        else
            echo.add(e);
            ED_new.pushToOrderingUnit(e);
end
```

event is received, to get rid of the echos at the end of the transition phase, N_1 is notified that this event type does not need to be forwarded any longer. Algorithm 5.1 gives the (abbreviated) pseudo code of the delay adaption and echo cancellation.

The problem of sudden increases of K for event detectors on upper levels in the processing hierarchy is solved by pseudo events. An event detector's ordering unit emits a pseudo event when its K increases. The pseudo event carries the new delay (with the faked time stamp, see Section 3.5) to the upper level event detectors so that they can update their buffer sizes well before they receive an event that may otherwise be too late. Pseudo events are not used for event detection but only for resizing buffer sizes further up the hierarchy.

This new cooperative handover performs safe migration of event detectors and simultaneously ensures a total event order for the new event detector. Event detectors are copied and immediately shut down because the migrated event detector iteratively calibrates its optimal K.

5.1.4. Evaluation

We have again analyzed the position data streams from the Real-time Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. To compare the efficiency of the presented approach we first compare our cooperative handover to classic migration techniques. In Section 6.3 we present a use-case evaluation and a sample scenario that prove that runtime migration is needed.

Comparison with classic migration

For a comparison we replay the recorded test match data and process it in our lab's virtual environment (an ESXi server with a cluster of VMs, each with a 2 GHz Dual Core CPU, 2 GB of main memory, and 1 GBit virtual network communication configured to simulate a real networked environment).

In this setup, we migrate an event detector for detecting a *pass*. It subscribes four different event types and emits the pass event. Other detectors behave similarly.

Approaches that ignore the order of events [26, 92] only copy the snapshot and inevitably fail. Approaches that run in parallel [26, 90] until all delays are correctly measured are less efficient than those that use stream forwarding [55]. We denote the latter as *classic* approaches and compare the migration presented here with them.

Misdetection avoidance

A classic migration would snapshot the event detector on one host, ship it to and re-start it with the current buffer size on the target host, and subscribe to all the necessary events (this is better than an initial buffer size of K=0). This only works well if all the subscribed events arrive at the new host earlier than they used to arrive before migration. If events take longer, the buffer is too small and the event detector will fail because it processes events out of order.

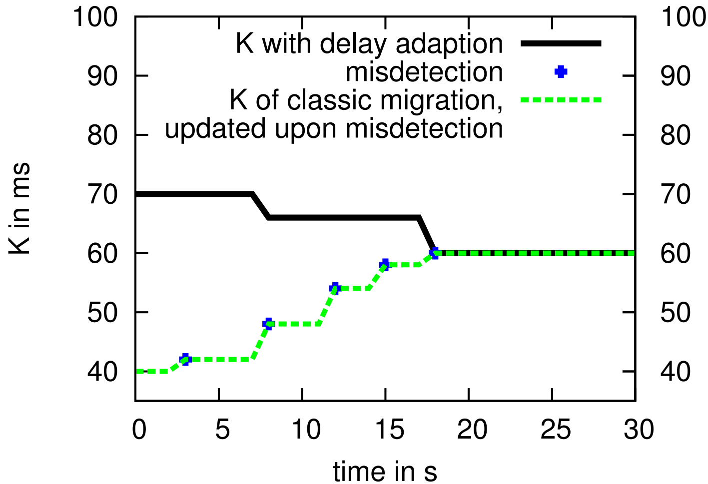


Figure 5.5.: Delays and K after state transition.

To demonstrate that this problem does occur in practice and that our cooperative handover can deal with this issue, we replay the test match data twice.

Figure 5.5 shows the K-values of the event detector migrated in the classic way (first replay, dotted line, K starts at 40ms). This event detector fails 5 times within the first 17 seconds after migration. Whenever it fails, K is increased to prevent future missdetections (as dynamic K-slack buffering would do). One event type first shows up after 30s but did not increase K.

In comparison, in the second replay our cooperatively migrated event detector subscribes to the events twice. It receives the events on the direct path (as the naively migrated event detector does) and a forwarded copy from the old site which includes the forwarding sub-delay of 30ms. Hence, our technique makes the event detector start with a higher K-value (40+30=70ms) at the beginning. And whenever direct events show up early, the K-value is lowered. As shown in Figure 5.5 our novel technique avoids missdetections as the K-value is always large enough. Moreover, K of the migrated *pass* event detector is just 17% too large at the beginning (70ms instead of 60ms) and melts down quickly. This is a small price tag for perfect detection.

Bandwidth and shipping cost reductions

An idea to make the classic migration avoid missdetection is to leave the event ordering unit behind on the old host and to forward the ordered event stream to the new host where the events can then be processed in order. On the target host we only measure the event delays of directly received events, and let a new ordering unit take over as soon as K is properly derived.

In the example of Figure 5.5, the classic migration with event forwarding sorts the events on the original host for 30 seconds with a delay of K=40ms before they are forwarded to the target host with an additional forwarding delay of d_f=30ms. The total delay on the target host is 70ms. This goes on for 30s, before the migrated event detector can switch to the newly configured reordering unit (K=60ms). This results in the same correct detection that our cooperatively migrated detector achieves, but without the early meltdown of K, i.e., with a higher accumulated latency.

Moreover, latency is not the only disadvantage of the classic migration with event forwarding. The main problem is the network bandwidth that event forwarding consumes. Forwarding of all events for the first 30s takes 13,337 packets, just for this single pass detector. In contrast, our novel cooperative migration can stop sending events of a certain type, as soon as one event of that type has reached the migrated event detector along the direct path. In the example, a total of only 51 packets need to be sent, i.e., cooperative migration can save 99.6% of the network bandwidth and shipping overhead of the classic technique with event forwarding. For other event detectors that we see in practice there are even larger savings. For some of the event detectors of the soccer application it takes as long as 5 minutes before at least one event of every type has appeared, i.e., before the classic technique can stop forwarding of all events. The results shown here generally hold for other event detectors because the number of forwarded events is limited by the number of subscriptions. Each type is only forwarded until the new host received it once. Hence, a statistical consideration is not necessary. The migration only has the overhead of the K-slack buffer at the old node which is negligible.

The overhead of the correct but inefficient classic migration with event forwarding is prohibitive since migration is triggered often because of bandwidth or load bottlenecks, see Section 6.3 for more details.

5.1.5. Summary

In distributed publish/subscribe-based systems there arise needs to migrate event detectors from one host to another at runtime due to various reasons such as load balancing or performance optimization. The presented method migrates stateful event detectors at runtime while low-latency time constraints are kept valid. To guarantee a correct event order we use delay information from the old host and runtime measurements on the new host to calibrate the event input buffers of the ordering units. The introduced network overhead is negligible and the method works well on a Real-time Locating Systems (RTLS) in a soccer application. The algorithm also solves migration issues known from sensor networks. Event detectors can be considered to run on sensor nodes and events need to be transferred in the network.

The next section shows how to optimize the detector distribution using heuristics. This reduces the detection latency in the distributed system environment.

5.2. Runtime Optimization

As we have seen event-based systems (EBS) are a good choice to efficiently process massively parallel data streams. In distributed event processing the allocation of event detectors to machines is crucial for both the latency and efficiency, and a naive allocation may even cause a system failure. But since data streams, network traffic, and event loads cannot be predicted sufficiently well the optimal detector allocation cannot be found a-priori and must instead be determined at runtime.

This section describes how evolutionary algorithms (EA) can be used to minimize both network and processing latency by means of runtime migration of event detectors. Moreover, it qualitatively evaluates the algorithms on synthetical data streams in a distributed event-based system. We show that some EAs work efficiently even with large numbers of event detectors and machines and that a hybrid [64] of Cuckoo Search [140] and Particle Swarm Optimization [78] outperforms others.

5.2.1. Introduction

Think of autonomous robots that play soccer and locate their positions [76] or in general applications where computation is spread over components connected with significant network latencies (e.g. sensor networks [134] or wide area publish/subscribe systems [103]). For instance, the robots form a wireless ad-hoc network to run both distributed decision making algorithms and rule violation detection. The algorithms are implemented as a distributed EBS, and events and sensor readings can be transmitted and processed on any robot. For instance, a high-level tactical rule violation such as *offside* can be defined by composing events from lower levels, see Figure 5.6. Event detectors are spread over the available computing nodes and the robots iteratively aggregate and process events until they detect the top-level event. Especially for tactical decisions but also for the detection of rule violations a low latency processing is crucial and event detectors must be allocated carefully across the network. Similar use-cases appear in military maneuvers or in autonomous coordination of (Mars) robots.

Rarely there is an optimal allocation of event detectors to the distributed processing units because the robots, i.e., the processing components, are mo-

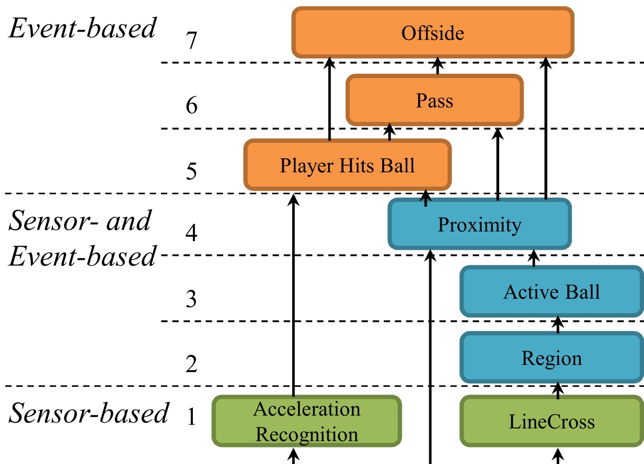


Figure 5.6.: Event hierarchy for an *offside* rule violation.

bile or communication may be instable, they constantly interact with each other, experience continuously changing tasks and latencies, and event loads are unknown a-priori.

Hence, an allocation must continuously adapt to the system environment and data load in order to be optimal. Unfortunately, not only is finding an optimal allocation known to be NP-hard, but also as good allocations quickly turn deprecated, slow exact algorithms are useless. If the optimizer takes too long to allocate a large number of event detectors to the available nodes, event loads may already have changed again. The optimal allocation is a moving target.

Greedy approaches are insufficient as they only improve an allocation locally, are likely to get stuck in local optima, and only obtain small performance benefits.

This section exploits the fact that sensor data, events and performance measurements can be (wirelessly) broadcast to a central unit. We show how Evolutionary Algorithms (EA) can be used to derive good solutions for the latency optimization problem and to produce migration commands that are sent to the respective nodes to migrate their event detectors. EAs are well-suited for the moving target problem. They terminate their search after a while, pick the currently best solution, and migrate the event detectors accordingly.

We evaluate different heuristics for the optimization of event-based systems and show why some of them are or are not suited for event-based systems. The requirements for such heuristics are twofold. First, since networking delays are high (for wireless transmissions) and since we face a hierarchical communication pattern, the primary goal is to minimize network transmissions. Second, global knowledge on event loads can be harvested from all robots/hosts and should be used for better results. Event stream data is continuously collected at a centralized point and a search heuristics is triggered to calculate an optimal allocation for the distributed system. Event detectors are then migrated between nodes to achieve a better performance in terms of latency.

The rest of this section is organized as follows. Section 5.2.2 reviews the related work. Section 5.2.3 then formalizes the optimization problem before Section 5.2.5 presents the optimization heuristics. We evaluate the heuristics in Section 5.2.6, discuss the results when processing synthetic event data streams, and show that a Cuckoo Search / Particle Swarm Optimization (CS/PSO) hybrid outperforms the other heuristics.

5.2.2. Related Work

Related work for runtime latency optimization is found in various areas such as distributed operating systems, sensor networks, and event-based systems. But most of them optimize other criteria or do not meet the above mentioned requirements.

Xing et al. [139] and Balazinska et al. [17] periodically collect CPU load statistics and use a greedy algorithm to migrate pairs of filter operators in the Borealis EBS [1]. Because their filter operators run for many seconds, they only consider CPU load but do not address network latency at all.

Plan-based approaches [4, 112, 138] statically calculate an optimal allocation a-priori but do not adapt event detector placements at runtime to changing event loads.

Liu et al. [90] address main memory shortage and split operator states for long running, non-blocking queries such as multi-joins. However, they focus on splitting large database operations and do not consider network latency as critical. Threshold-based techniques [141] also only optimize CPU and memory loads but do not optimize networking latency.

Lakshmanana et al. [83] split event processing agents into several *strata* that run in parallel before they profile for load balancing. But since they just can migrate few agents they only approach local optima.

With projecting event queries onto distributed hash tables [65, 108], queries can be divided into sub-queries and loads can be moved between hosts that are near to save communication cost. These approaches optimize locally and may also result in worse allocations if event detectors form a detection hierarchy.

Zhou et al. [143, 144] focus on local load balancing in publish/subscribe systems where communication is not tightly coupled. In contrast, in our application domain the communication paradigm is tightly coupled and the event loads are dynamic.

The distributed Lagrangian Rates, Greedy Populations (LRGP) [95] algorithm optimizes resource allocation by adapting flow rates and consumer admissions. However, reducing consumer admission to control the resource allocation is no viable solution because they drop events which may result in event detection failures.

Babcock et al. [13] adaptively optimize operator scheduling to minimize memory consumption and throughput latency. However, they replace the or-

der in which stream operators are applied to a number of data packets from multiple input streams on a single machine. This is an orthogonal type of optimization as we optimize the placement of operators in a network.

MOSIX [7, 79] is a distributed cluster operation system that proactively migrates processes at runtime to optimize the performance. Decisions are based on a theoretical cost model that incorporates CPU time and memory consumption of jobs, collects node statistics from the network, and (re-) assigns jobs. However, the communication costs *between* jobs are not optimized.

Khanna et al. [80] use a genetic algorithm (GA) to generate an optimal number of cluster heads in sensor networks to minimize communication cost, and hence to maximize the lifetime of the sensor net. However, in sensor nets the requirements usually differ because sensor nodes transmit data towards a sink and only rarely communicate with each other to perform distributed tasks.

Ant Colony Optimization (ACO) [118] is a popular method for network-based optimization. However, those technique are mainly used to optimize network routes or to provide alternative routes for end-to-end or multiple-end communications. Since we essentially want to optimize a complex multi-hop scenario, known ACO methods cannot be applied.

5.2.3. Runtime Latency Optimization

The presented runtime latency optimization moves event detectors to improve latencies. As event streams cannot be predicted sufficiently well, plan-based approaches do not work for online optimization. As greedy approaches often end up in local optima we apply heuristics to optimize the detector distribution at runtime. This requires continuous monitoring of all event stream statistics in a centralized optimization master (OM) component. The OM collects the number of transmitted events for each `ID`, the measured network latencies to other nodes, and the number of generated events. The OM periodically triggers one of the heuristics of Section 5.2.5.

Problem Formalization

Let n be the number of event detectors, m the node count. An allocation is a three-tuple consisting of an $m \times n$ matrix X with $x_{i,j}$=1 iff detector i runs on node j, an $n \times n$ matrix L where $l_{i,j}$ is the latency between hosts i and j, and

an $m \times m$ matrix T where $t_{i,j}$ is the number of events transmitted from node i to j. The goal is to minimize

$$\varphi(X, L, T) = \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{k=1}^{n} x_{i,k} \cdot x_{j,k} \cdot \left(c_t \cdot t_{i,j} + c_l \cdot l_{i,j}\right)$$

which describes the cost of the current detector distribution under the runtime measurements L and T, weighted by c_t (emphasis on network transmissions) and c_l (emphasis on latency).

5.2.4. Solution

An optimization based on heuristics needs (1) a formal encoding of possible solutions (individuums), and (2) a fitness function that grades the solution's performance.

Individuum. A detector i runs on host x_i in an event detector allocation $\vec{X} = (x_1, x_2, \cdots, x_n)^T$, with $x_i \in [1; m]$.

Fitness function. The fitness function is used to grade a possible solution and projects it onto a number. The higher the quality $f(\vec{X})$ of a solution $\vec{X}$ is, the fitter is the individuum in the evolutionary algorithm. Our fitness function incorporates network latencies and traffic, and is again weighted by the parameters c_t and c_l:

$$f(\vec{X}) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} \cdot c_t \cdot t_{i,j} - \sum_{i=1}^{n} \sum_{j=1}^{n} b_{i,j} \cdot (c_t \cdot t_{i,j} + c_l \cdot l_{i,j}).$$

$a_{i,j}$ is 1 if event detectors i and j run on the same host, i.e., $x_i = x_j$, and 0 otherwise. Contrary, $b_{i,j}$ is 1 if detector i and j run on different hosts, and 0 otherwise. The left part of the fitness function sums up the *cost savings* by events that are transmitted locally between event detectors and that must not travel through the network. The right part sums up the cost for detector dependencies that cause network traffic. Hence, $f(\vec{X})$ grows with locally transmitted events and decreases with events that are subscribed by hosts with higher latency or with more generated events. Due to limited CPU power we skip and discard solution vectors that would overload hosts.

Algorithm 5.2: The Genetic Algorithm.

```
Data: fitness function f(X⃗)
individuums ← generate initial set of individuums;
while (t < MaxGeneration) or (!stop-criterion) do
    foreach (individuum X⃗_i) do
        evaluate fitness f(X⃗_i);
    sort all allocations X⃗ by fitness value;
    select two higher ranked allocations and perform crossover;
    randomly select allocation, replace subsequence, replace if
    better;
return best allocation
```

5.2.5. Heuristics

This section evaluates four heuristics to dynamically optimize event detector allocations: a Genetic Algorithm (GA) [68], a Cuckoo Search (CS) [140], a Particle Swarm Optimization (PSO) [78], and a hybrid form of CS and PSO [64]. Below we shortly describe their principles and the guts of their application to the runtime optimization problem.

Genetic Algorithm (GA)

The classic GA is based on fundamental mechanisms of the theory of the natural evolution and the continuous change of DNA sequences to improve a species' fitness.

A population consisting of d distributions $\vec{X}$ reproduces for k generations. In each generation we apply three operations: (1) selection, (2) crossover, and (3) mutation. Algorithm 5.2 provides the pseudo code of the genetic algorithm.

(1) Selection. We evaluate the fitness of each solution of the current generation and sort them by their calculated fitness values. We use a fraction p, with $0 < p < 1$, of the highest ranked solutions in the following step.

(2) Crossover. We choose two solutions, $\vec{X}_a$ and $\vec{X}_b$, randomly from the top fraction p of the current generation. We randomly choose a subsequence $(x_i, x_{i+1}, .., x_j)$, $1 \leq i < j \leq m$ from both $\vec{X}_a$ and $\vec{X}_b$, and exchange this sequence between the solution vectors. We evaluate the fitness of the resulting

distributions $f(\vec{X}'_a)$ and $f(\vec{X}'_b)$, and if they are better than $f(\vec{X}_a)$ and $f(\vec{X}_b)$, we replace them.

(3) Mutation. To overcome local optima and to generate new solutions, we randomly choose a distribution $\vec{X}_c$ among the population for mutation and replace a randomly chosen subsequence of its solution vector with valid random values. We determine the fitness of the resulting distribution $\vec{X}'_c$, and if it is better, we replace $\vec{X}_c$ with $\vec{X}'_c$.

Cuckoo Search (CS)

Cuckoo Search [140] is an iterative and evolutionary algorithm that manipulates an initial set of d solutions over a given number n of generations. CS uses (1) the parasitic breeding behavior of some species of cuckoo birds and (2) the characteristics of the random walk of the common fruit fly's method to determine its route for exploring feeding grounds.

(1) Breeding behavior. The population of solutions is represented by d bird's nests (containers), each of which holds a solution $\vec{X}$. To stay in the metaphor of CS, these nests are owned and used as breeding locations by other species of birds. A cuckoo bird reproduces by laying its own egg into one of these nests and lets the victimized bird breed it. The victims either hatch the egg, evict it, or abandon their own nest. How a cuckoo generates an egg is discussed in (2).

We implement these options as follows. In each generation, the cuckoo generates a new egg $\vec{X}'$, and if it represents a better solution with a higher fitness value $f(\vec{X}')$ than the egg $\vec{X}_j$ of a randomly chosen nest j, it replaces $\vec{X}_j$. Otherwise $\vec{X}'$ is discarded. Furthermore, in each iteration a fraction p of eggs with low fitness values is abandoned and replaced by randomly generated solutions/eggs in new containers/nests.

(2) Egg generation. After randomly selecting a nest i the cuckoo takes the egg $\vec{X}_i$ and performs a Lévy Flight in the solution vector. The Lévy Flight is the common fruit fly's method for determining its route, i.e., a special random walk, and we use it to explore the possible allocations. The random walk shifts some of the dimensions of vector $\vec{X}_i$:

$$\vec{X}' = \vec{X}_i + c \cdot Levy\vec{F}light(\alpha).$$

The step length is determined by a heavy-tailed random distribution that has an infinite variance with an infinite mean. The factor c is used to weight the random walk, and α is the Lévy distribution's parameter. This walk varies from local exploration steps to steps far out of the current proximity and is performed on the given solution (egg) $\vec{X}$ from the randomly chosen nest j. The result is evaluated and compared to another randomly chosen nest's egg i as described in step (1). This avoids local optima and explores new locations more quickly.

Due to the discrete and interdependent characteristics of the representation of the event detector allocation in $\vec{X}_i$, we do not shift all dimensions of $\vec{X}_i$ at once. A change caused by a random walk in one dimension can have a ripple effect on the quality of the allocation of other event detectors to their host machines. Therefore it is necessary to limit the impact of the random walk in our scenario. Hence, we randomly select a small subset of dimensions of $\vec{X}_i$ and perform the random walk on each of them. The step length of the random walk is also important. It influences not just its own dimension but possibly several more because event detectors subscribe events from other detectors.

The result $\vec{X}'$ of the random walk is then used as a new cuckoo egg. It is evaluated according to the rules described in step (1) and, depending on the result, placed into the nest j chosen in step (1).

Algorithm 5.3: The Cuckoo Search algorithm.

Data: fitness function $f(\vec{X})$, Lévy Parameter α,
unfit fraction p;
nests ← generate initial set of d allocations;
while *(t < MaxGeneration) or (!stop-criterion)* **do**
    i ← nests.getRnd();
    Allocation $\vec{X}'$ ← $\vec{X}_i$.LévyFlight(α);
    j ← nests.getRnd();
    if $f(\vec{X}') > f(\vec{X}_j)$ **then**
        $\vec{X}_j \leftarrow \vec{X}'$;
    sort all allocations $\vec{X}$ in nests by fitness value;
    replace the fraction p of the nests with a low fitness value with random new and valid ones;
return *best allocation*

The fraction p of eggs to abandon at the end of each generation is also important for the algorithm's performance. It is essentially a global search with a high randomness, and not a random walk. However, as the probability is small to create good eggs by chance we get better results with a small fraction p.

Algorithm 5.3 shows the pseudo-code of the CS algorithm. The stop criterion can be any condition, for instance a well chosen maximal number of iterations in which the best solution did not change significantly. We leave that out for simplification.

Particle Swarm Optimization (PSO)

PSO applies techniques from the swarm behavior of some animals/particles, especially birds and fish. The pseudo-code is given in Algorithm 5.4. A population of d animals/particles, each a representation of a solution $\vec{X}$, flocks towards an optimal state. To implement the swarm behavior, it is necessary to know the velocity $v_i(t)$ at time t of each particle i and the globally best particle $\vec{X}_{g_best}$ found so far, evaluated by $f(\vec{X})$. Furthermore, we need the personal best position $\vec{X}_{p_best_i}$ of each particle i. A particle i then moves through the solution space into a direction computed from these values as shown in Figure 5.7. Operating on the set of particles, a time-discrete loop performs the following three steps.

Algorithm 5.4: The PSO algorithm.

Data: fitness function $f(\vec{X})$
Allocation $\vec{X}_{g_best}$;
swarm ← generate initial set of d allocations;
while *(t < MaxGeneration) or (!stop-criterion)* **do**
  foreach *(particle $\vec{X}_i$)* **do**
    evaluate fitness $f(\vec{X}_i)$;
    update $\vec{X}_{p_best_i}$ and/or $\vec{X}_{g_best}$;
    $\vec{v}_i(t+1)$ ← update velocity $\vec{v}_i(t)$;
    $\vec{X}_i(t+1)$ ← update position $\vec{X}_i(t)$;
return $\vec{X}_{g_best}$

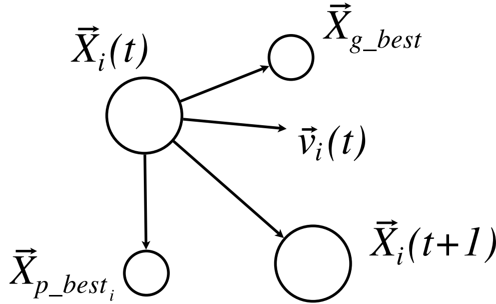


Figure 5.7.: The way PSO combines weighted forces.

(1) Fitness evaluation for all particles' current positions. If necessary, $\vec{X}_{g_best}$ and $\vec{X}_{p_best_i}$ are updated.

(2) Update velocities. Based on these new values we calculate the velocity $\vec{v}_i(t+1)$ as a weighted sum of the current speed and the direction to the personal and the global best particles, see Figure 5.7:

$$\vec{v}_i(t+1) = K \cdot [\vec{v}_i(t) + c_1 \cdot r_1 \cdot (\vec{X}_{p_best_i} - \vec{X}_i(t)) + c_2 \cdot r_2 \cdot (\vec{X}_{g_best} - \vec{X}_i(t))].$$

We calculate the velocity separately for each dimension of a solution vector $\vec{X}_i$ and use a constriction factor model proposed by Clerc et al. [50]. This model is fast to calculate, performs similarly, and is a good alternative to the inertia weight method. The factor c_1 is used to weight and prioritize the local optimum, c_2 is used to weight and prioritize the global optimum, and r_1 and r_2 are random values between 0 and 1 that are used to let the particles escape from local optima to explore new areas. K is the constriction coefficient and satisfies

$$K = \frac{2}{\left|2-(c_1+c_2)-\sqrt{(c_1+c_2)^2-4(c_1+c_2)}\right|}.$$

(3) Update positions. We derive the new valid position for each particle from a sum of its old position and its new velocity:

$$\vec{X}_i(t+1) = \vec{X}_i(t) + \vec{v}_i(t+1).$$

Cuckoo Search / Particle Swarm Optimization (CS/PSO)

The hybrid of CS and PSO [64] combines swarm intelligence and Lévy Flights to benefit from both concepts. The random walk of CS helps the algorithm to escape from local optima while the swarm intelligence of PSO accelerates the overall convergence by letting the cuckoos collaborate. The pseudo-code is given in Algorithm 5.5.

The key idea of the algorithm is that not a single cuckoo bird performs Lévy Flights on a randomly chosen egg but a swarm of cuckoo birds is used to find new solutions. Furthermore, we also have a population of bird's nests,

---

Algorithm 5.5: The CS/PSO algorithm.

Data: fitness function $f(\vec{X})$, Lévy Parameter α,
unfit fraction p;
Allocation $\vec{X}_{g_best}$;
nests ← generate initial set of d allocations;
swarm ← generate initial set of e allocations;
while *(t < MaxGeneration) or (!stop-criterion)* **do**
  cuckoo ← swarm.getRnd();
  Allocation $\vec{X'} \leftarrow \vec{X}_{cuckoo}$.LévyFlight($\alpha$);
  n ← nests.getRnd();
  if $f(\vec{X'}) > f(\vec{X}_{cuckoo})$ **then**
    $\vec{X}_{cuckoo} \leftarrow \vec{X'}$;
  if $f(\vec{X'}) > f(\vec{X}_n)$ **then**
    $\vec{X}_n \leftarrow \vec{X'}$;
  sort all allocations in mesh by fitness value;
  replace the fraction p of the mesh with a low fitness value with random new and valid ones;
  foreach *(cuckoo's allocation $\vec{X}_i$)* **do**
    evaluate fitness $f(\vec{X}_i)$;
    update $\vec{X}_{p_best_i}$ and/or $\vec{X}_{g_best}$;
    $\vec{v}_i(t+1) \leftarrow$ update velocity $\vec{v}_i(t)$;
    $\vec{X}_i(t+1) \leftarrow$ update position $\vec{X}_i(t)$;
return $\vec{X}_{g_best}$

---

each of which holds a solution $\vec{X}$. In contrast to CS each cuckoo bird in the swarm represents a solution $\vec{X}$ as well. The swarm of cuckoo birds follows the rules defined by the PSO algorithm. In each generation the swarm's birds move as defined by the PSO algorithm, and a single randomly selected cuckoo additionally performs a Lévy Flight on its allocation to create a new egg $\vec{X'}$. This is done in order to escape from local optima and to speed up convergence. This way, the cuckoo's chance to put a better egg into its victim's nest improves.

A time-discrete loop performs the following three steps.

(1) Cuckoo selection. In every generation we randomly choose a cuckoo and its allocation $\vec{X}_{cuckoo}$ from the swarm and perform a Lévy Flight on its solution vector to create a new solution $\vec{X'}$. We evaluate the fitness of the result $\vec{X'}$ and replace $\vec{X}_{cuckoo}$ if the former is better.

(2) Nest selection. Next, we randomly choose a nest n. The fitness $f(\vec{X}_n)$ is evaluated and if it is worse than $f(\vec{X'})$, we replace $\vec{X}_n$ with $\vec{X'}$, otherwise we discard $\vec{X'}$.

(3) Selection of fittest. At the end of each iteration a fraction p of nests with low-fitness distributions is replaced by random new ones and the cuckoo swarm's positions and velocities are updated, as are the personal and global best solution vectors.

To adapt this algorithm to our specific scenario, we again perform the Lévy Flight only on a subset of dimensions of a solution $\vec{X}$. This again improves the local search due to the interdependent nature of the dimensions of $\vec{X}$. By limiting the random walk we prohibit overly random new solutions $\vec{X'}$, as a change in one dimension can have a ripple effect on the quality of the allocation of other event detectors to their host machines. Furthermore, we also update the cuckoos' personal bests and the swarm's global best if a Lévy Flight results in an allocation with better fitness. Additionally, to simplify the parameterization of the swarm's behavior, we again use the constriction factor model to update the swarm's velocity. Analog to CS, CS/PSO works best with a small value for the fraction p.

5.2.6. Evaluation

To evaluate the four heuristics we created a benchmark scenario consisting of 30 event detectors that process synthetic event data streams on 6 nodes. These 6 VMs (LinuxContainer instances, each with a 2 GHz CPU core, 1GB of main

memory and 1GBit virtual network) run on a server with 2 Intel Xeon X5675 Six Core CPUs and 64GB of main memory, and are linked by ns-3, a network simulator [105], to implement the virtual topology shown in Figure 5.8. Every 60s the synthetic event detectors, symbolized as colored shapes in Figure 5.8, change their subscription patterns. They switch between the two situations shown in Figure 5.9. The numbers denote the event detector IDs. The different colors and shapes denote the *optimal* allocations of event detectors to the 6 nodes, if not more than 5 event detectors may run per node. The arrows show the direction of events detected/published. The optimal allocations are different for the two phases. They have been determined a-priori by means of an exhaustive search.

To evaluate the heuristics, we feed a synthetic event stream to the root event detector `0`, and analyze the flood of events along the event hierarchy. The heuristics then minimize the number of events that are sent over network links with higher latency by migrating event detectors to nodes with lower latency, or by grouping highly dependent event detectors to the same node. Therefore, the heuristics minimize the *average latency* between any two collaborating event detectors, as more events are routed locally on the processing node or via links with lower latencies. The average latency is the arithmetic mean of all latencies of the event detectors' subscriptions over the network, measured at runtime. Local subscriptions have a latency of 0.

We evaluate the heuristics' performances and convergence behaviors by comparing the average latencies they achieve to the optimal allocations. We discuss each algorithm's performance separately and present only their es-

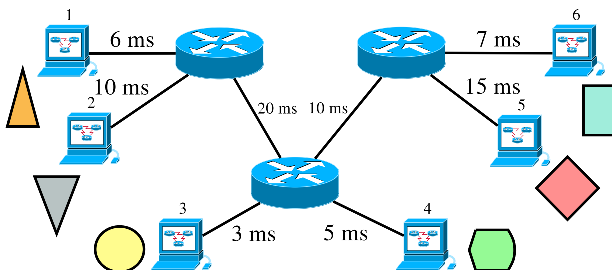
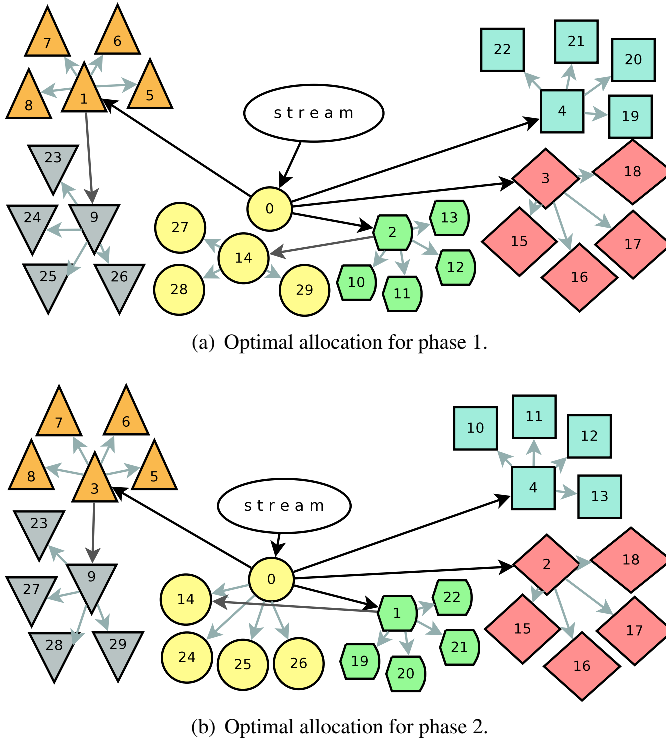


Figure 5.8.: Topology of virtual network environment.



(a) Optimal allocation for phase 1.

(b) Optimal allocation for phase 2.

Figure 5.9.: Subscription patterns of event detectors.

sential evaluations for clarity. The number of iterations/generations for all heuristics is set in a way to limit the runtime to 3 seconds. This is needed for our low-latency event based system with changing dependencies and workloads. Additionally, before each optimization the heuristics measure the event count and latencies for 2 seconds. In Figure 5.10 the straight black line shows the average latency of the optimal allocations found a-priori (Figure 5.9). The second straight line in each of the diagrams in Figure 5.10 shows what the heuristics can achieve when configured with the parameter set that works best. In addition we show the performance that the heuristics achieve with other parameter sets. Every 60 seconds the subscription pattern of the event detectors

switches between the phases. While this does not affect the optimal/manual allocation, the heuristics face a peak of latency before they dynamically adapt. We start the measurements after 50s because initially detectors are distributed randomly.

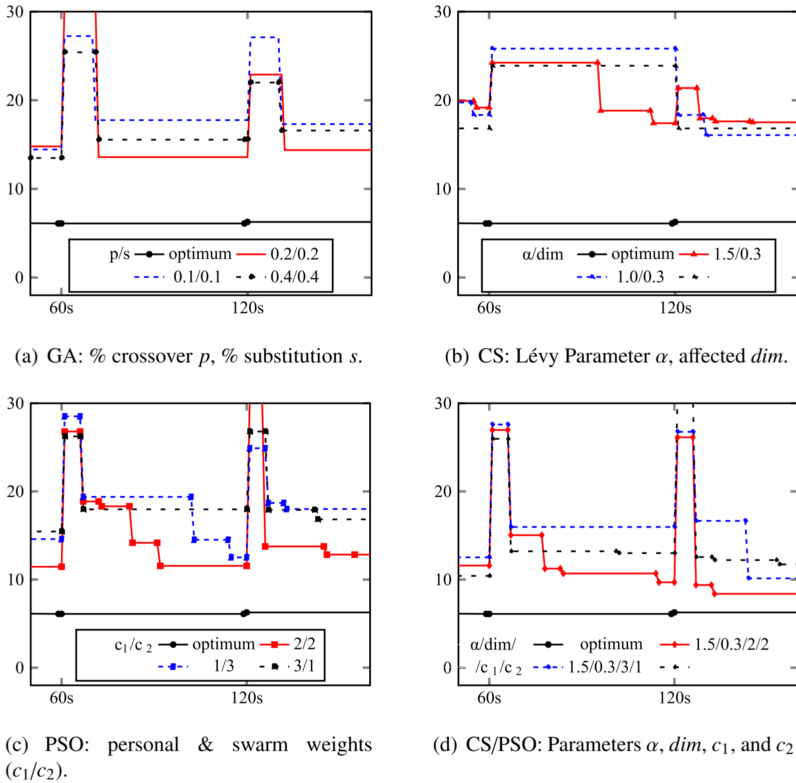
Genetic Algorithm (GA)

The parameters of the GA are (1) the percentage p of the solutions that perform crossover and (2) the percentage s of a subsequence of the solution vector $\vec{X}$ that is exchanged between two randomly chosen solutions $\vec{X_a}$ and $\vec{X_b}$ in the crossover step. We use a fixed population with a size of 80 solution vectors as recommended in [68]. Figure 5.10(a) shows the average latency of all event detectors that the GA achieves with three parameter sets p/s. With p=0.2 and s=0.2 the GA outperforms other parameter sets. The number of generations has been set to $35,000$.

After the first phase change, the GA with the best parameters improves the event detector allocation by 56%, i.e., the average latency drops from 30.8ms to 13.6ms. This is still considerably slower than the optimal average latency (6.1ms). After the next phase change the best GA again needs to run twice before it adapts. It achieves an average latency of 14.4ms which is more than twice the optimum (6.27ms). Although the GA keeps monitoring and searching, it does not find better allocations. Therefore there are no reallocations of event detectors and no extra dots on the line in the diagram. Other parameter sets, e.g., p=0.4, s=0.4, exhibit a slower convergence and end up with slower latencies. With higher values of the parameters p or s, the GA often misses good solutions that are close to the population's solution vectors. With lower values the GA stagnates at local optima and does not explore new solutions.

Cuckoo Search (CS)

CS has two relevant parameters (1) the Lévy Flight's distribution α that influences the step length of the random walk. With higher values for α the CS is more likely to generate small variations only instead of far leaps. Parameter (2) is the fraction of dimensions *dim* on which the Lévy Flight is performed. Preliminary experiments suggest that the fraction p of abandoned nests is not significant. The reason is that randomly generated results with a high fitness are unlikely. We set the population size to 15 nests [140].



(a) GA: % crossover p, % substitution s.

(b) CS: Lévy Parameter α, affected dim.

(c) PSO: personal & swarm weights (c_1/c_2).

(d) CS/PSO: Parameters α, dim, c_1, and c_2.

Figure 5.10.: Dynamic latency adaption achieved by GA, CS, PSO and CS/PSO.

Figure 5.10(b) shows the performance of the CS algorithm. Of all evaluated parameter sets, the parameter set α=1.5 and dim=0.3 converges best. Unfortunately it takes 95 seconds to find an event detector allocation that slightly improves the average latency. Just before the phases change again, the best found allocation has an average latency of 17.4ms, which is far from the optimum (6.1ms).

One other parameter set in the diagram shows the effect of a smaller Lévy parameter α (1.0 instead of 1.5). Due to the resulting far leaps the solution space exploration misses nearby good allocation vectors. Potentially better allocations around local optima are not explored efficiently. Even if the effect is not as drastic after the second phase change, the resulting latencies are still worse than what the GA found. The other variation of parameters, i.e., α=1.5 and *dim*=0.1, exhibits similarly bad results. They suggest that small values for *dim* should be avoided. The CS with the parameter set that adapts best to the system's dynamics still performs considerably worse than the GA.

Particle Swarm Optimization (PSO)

Recall that the the velocity formula made use of each particle's local/personal best position and the swarm's global best solution, weighted by c_1 and c_2 respectively. We evaluated PSO for different sets of these weights, using a population of 80 particles and 27,000 iterations, which preliminary experiments suggested as suitable numbers.

Figure 5.10(c) shows that the best PSO with equal weights almost continuously optimizes the allocations after the first phase change until it settles at an average latency of 11.5ms, which is close to the optimum (6.1ms). After the second phase change depending on the parameter set, PSO improves the event detectors' allocations after just one run by between 24% and 54%. With equal weights c_1 and c_2 PSO reaches an average latency of about 13ms while the parameter sets with unmatching weights end up around 18ms.

With unmatching weights, the swarm's particles either stagnate at their respective local best allocation ($c_1 > c_2$) or the swarm quickly degenerates to the global best ($c_1 < c_2$). PSO only works well if the personal weight c_1 and the swarm weight c_2 match, e.g., c_1=c_2=2.

PSO achieves much better latencies than both the GA and the CS (11.5ms in the first phase and 12.8ms phase 2). It also exhibits better convergence.

Cuckoo Search / Particle Swarm Algorithm (CS/PSO)

The hybrid CS/PSO combines the CS and the PSO parameters. As before we use 15 nests and we set the swarm's population size to 80. Since the hybrid is computationally slightly more complex than PSO, only 25,000 iterations fit into the 3s optimization window.

Figure 5.10(d) shows the performance of the best parameter set we found and two additional sets that we use to explain the results. After the first phase changes, the best CS/PSO already reduces the average latency to 13.2ms in its first run. Eventually it finds an allocation with 9.7ms average latency, just 3.6ms above the optimum of 6.1ms. After the second phase change, it performs even better.

As with the PSO, the weights c_1 and c_2 ought to be equal to maintain a balance between stagnation and degeneration. Small steps in the Lévy Flight applied to more dimensions work better than large leaps on fewer dimensions. The PSO's swarm, as a part of the hybrid algorithm, is well suited to conduct global searches. The cooperation of the swarm's particles speeds up global searches towards (local) optima. The local search around allocations near the local optimum is improved by the Lévy Flights. By slightly varying solution vectors, the CS/PSO hybrid explores immediate neighbors, even if the allocations' fitness is close to an optimum.

In comparison, the CS/PSO hybrid algorithm clearly outperforms the other heuristics. Whereas the GA and CS stay far away from the optimum, PSO and CS/PSO both approach the optimal detector allocation. As Figure 5.10 shows, the CS/PSO hybrid performs better than the PSO heuristics, and it converges sooner and gets closer to the optimum.

5.2.7. Conclusion

Evolutionary algorithms can be exploited to optimize the allocation of event detectors in a distributed event-based system at runtime. In our system a centralized unit collects runtime information and latencies, triggers the heuristics, and sends migration commands to the respective nodes.

We evaluated four heuristics and showed that they all reduce network latencies and hence enhance the system's average detecting latency. The Particle Swarm Optimizer outperforms both a Genetic Algorithm and a Cuckoo Search. A combination of Cuckoo Search and Particle Swarm Optimization performs best. Multi-swarm optimization may further improve the heuristic.

5.3. Summary

In this chapter we delved into specific issues that arise in *distributed* event-based systems. With a runtime migration that considers low-latency ordering constraints we are able to move event detectors between nodes to avoid system overload or to optimize performance. Our novel runtime migration outperforms other techniques that have proven their efficiency in related areas such as distributed file systems or virtual machine migration. Our runtime optimization uses global data and event load statistics, triggers heuristics to calculate a lower-latency allocation set for the event detectors, and commands the particular nodes to initiate a migration of event detectors. This allows to dynamically optimize event detector allocations to the current event loads and network latencies.

6. Evaluation and Discussion

In previous chapters we introduced specific solutions for technical problems and demonstrated their efficiency by means of evaluations that zoomed into the details of the particular methods. While these micro benchmarks better explain the performance and internal behavior of our techniques they still leave some open questions that we would like to address in this chapter.

This chapter is organized as follows. Section 6.1 focuses on the evaluation of Event Definition Languages for the detection of events out of position data streams. We discuss and draw conclusions from the results we obtained from the 2013 ACM DEBS Grand Challenge [49] where we carried out a sophisticated competition for the analysis of the position data streams that we also used for the evaluations throughout this thesis. Section 6.2 presents a use-case evaluation of the detection of a *player hits ball* event out of a small detector hierarchy comprising *proximity* and *ball hit* events. This use case evaluation shows that an event ordering under the hood results in smaller source code that is easier to maintain. The higher detection latency of the buffering middleware approach in contrast to an in-detector-ordering pays off while events are traversing the event detector hierarchy. Section 6.3 shows a practical example why runtime migration is necessary in distributed event-based systems. Event detectors that run on 3 nodes suddenly become unbalanced due to unpredictable behavior of located objects, and one node suffers a system overload. Migration helps to load work off the overloaded node so that event processing stays fully functionable. Section 6.4 concludes this chapter.

6.1. Evaluation of Event Definition Language Approaches

As mentioned earlier there are basically two contrary approaches for the implementation of event processing systems: we could either make use of event definition languages (EDLs) and quickly implement event detectors on a high

level of abstraction, or we apply programming languages such as Java, C/C++, or C#, and operate on a lower level that achieves higher performance and flexibility. The former method has the advantages that we can reach a working system implementation quickly, that we do not need to worry about complex source code whose maintenance may require a significant amount of time, and that we avoid serious programming mistakes more easily. The resulting code is hence easy to maintain, compact, and safe.

In contrast to the EDL-based approach an implementation in a programming language may provide a considerably better performance. That is because we avoid a runtime interpretation of intermediate code. In addition to that, the compiler may massively optimize the source code before runtime for maximal execution speed under consideration of the underlying hardware architecture. On the downside, programming language implementations are highly complex and need sophisticated programming skills for their development. This also results in fewer potential application developers for the system programming. Another disadvantage is the time needed for the implementation: source code implementations must usually be tested thoroughly as errors can happen easily. This can significantly extend the time-to-delivery cycles.

Approaches that use event definition languages together with a custom compiler or even a just-in-time (JIT) compilation to avoid intermediate code are settled somewhere between the two approaches. Depending on the language the compiler may also optimize the source code for performance. The more similar the language is to a low level programming language, the more possibilities for optimizations open up. But with that also comes error-prone low level implementation work.

Hence, an evaluation that compares source code and query language implementations must depend on the application domain in order to draw valuable results. In fact, the results even depend on the particular event detectors. Evaluation criteria vary from expressibility of queries over execution speed to latency and throughput.

For the analysis of the position data streams we used throughout this thesis, an event processing system must at least provide a reliable event detection, i.e., without false-positive or false-negative events. A second criterion is latency: to perform actions as quickly as possible all the events must be detected with the lowest possible latency. However, as there are so many event processing systems out there, see Section 2.3, an evaluation on the applica-

bility of event processing systems and their different techniques results in a tremendous effort. Hence, we created a representative test data set with the RedFIR RTLS [129] and let two amateur football clubs play 8 vs. 8 on a half-sized soccer field in the main soccer stadium in Nuremberg, Germany. We formulated a set of events to be detected, and carried out a grand challenge competition in the research community [49] to tackle the problem and to present novel systems and approaches for the real-time processing of this kind of sensor data.

In particular, we defined four queries that represent a wider spectrum of application scenarios that involve continuous query processing. Each of the requested queries addresses a specific challenge for the analysis of the data. We briefly describe the queries below.

(1) Running analysis. The goal of the running analysis query is to quantify and track how well each of the players moves on the playing field. The running analysis aims for an efficient implementation of continuous window-based analysis of different running intensities of both the players and the teams. While the analyzing component should emit the current states continuously, it should also emit statistics that span 1, 5, 10, and 20 minutes, as well as for the whole game. The update frequency is required to be at least 50 Hz. This query poses a challenge towards updating internal state information quickly, i.e., continuous query aggregation.

(2) Ball possession. The goal of the ball possession query is to calculate the current ball possession, and statistics for each of the players and for each team. It is important that sub-events (like a proximity detection, a ball hit detection, or an active ball detection) are merged appropriately to detect the ball possession. This query hence requires a functionality to detect and reuse occasional events in the event processing system.

(3) Heatmap. The goal of the heatmap query is to calculate statistics about the presence of each player in a given region of the playing field. The heatmap query addresses performance and event reusability. The EPS should emit heatmaps for different time windows, i.e., 1, 5, 10, and 20 minutes as well as the whole game. The heatmaps should be generated for each player and for each team with different grid layouts laid over the playing field. In total the system must provide 20 heatmaps in different granularities that must be updated at least once per second.

(4) Shot on goal. The purpose of the shot on goal query is to detect that a player hits the ball in an attempt to score a goal. This query poses a challenge to the flexibility and expressibility of queries in the EPS. At the time of the ball hit, the query must wait for some time for the ball to leave the player in order to properly estimate the ball's trajectory. If the ball's trajectory then indicates that the ball may reach the goal area within the next 1.5 seconds the system should detect a shot on the goal. This query requires the passive waiting for events to happen and to define a complex trajectory function that can hardly be implemented by a continuous query in an EDL. However, such complex functions are not as unusual as it seems at first glance since they are often needed for location-based sensor data processing.

In the following we line out the solutions that have been submitted to the Grand Challenge committee, and discuss the overall outcome. In total, there have been 15 full paper submissions, from which only 6 high-quality solutions could be selected for acceptance. This Grand Challenge has drawn more interest than the previous two ACM DEBS Grand Challenges. In the following we sum up the overall impression of the solutions and interpret the results. We show that all accepted submissions concluded that high-level event definition languages are either not flexible enough to express the queries completely, or that they cannot meet the low latency requirements.

Jacobsen et al. [71] propose the BlueBay Soccer Monitoring Engine, a custom C++ solution. They also present two fully functional systems implemented in Esper [70] and Storm [99]. While they provide the functionality to induce custom functions and hence the needed flexibility for query expression, the authors conclude that STREAM [11] and StreamIT [82] are not suitable for this task because of limitations concerning I/O, stateful operators, and library support. Both systems lack direct support for user-defined functions to implement finite-state machine behavior and trajectory estimation. BlueBay provides the flexibility to trade off between processing throughput and processing latency. On an Intel Xeon 3.2 GHz QuadCore machine with 6 GB of memory BlueBay achieves a throughput of up to 790k events per second, i.e., 58× real time, when running in multi-threaded mode, and up to 364k events per second when running in non-threaded mode. The end-to-end detection latency ranges from 0.005ms to 56ms, depending on the desired throughput capabilities. The heatmap emission takes a few milliseconds on average and maximal 75ms (when the system aims for maximal throughput).

The SPRINT Stream Processing Engine by Wu et al. [137] is also a custom C++ solution. It makes use of novel strategies such as lock-free ring-buffers, frame-based sliding windows [98], and dynamic parallel computation by using several threads on multi-core architectures. The lock-free ring-buffer uses a fixed amount of memory and compare-and-swap (CAS) technique. It is essentially a basic version of the ring-buffer that we implemented in the Event-Core, see Section 2.2. A load shedder is used to avoid ring-buffer overwrites in case of congestion. However, while this feature avoids memory corruptions at the time of event insertion, congestion may still lead to a system failure as stateful event detectors get stuck in invalid states. On a 4-core Intel machine with 2 GB of main memory running CentOS 5.8 SPRINT achieves a throughput of 133k events per second (without load shedding). With an increasing load shedding rate the throughput rises up to 143k events per second. However, with a load shedding rate of 100% no output is generated as any input event is discarded. The authors conclude that a load shedding rate of up to 75% does not have a significant influence on the correctness of the results. However, load shedding changes the result of accurate intervals within the running analysis.

Jergler et al. [74] propose a custom Java-based solution that makes heavy use of ring-buffers for inter-communication between computational tasks, and that trades off between spin-waiting (active) and locking (passive) to reduce detection latency. Their Java-based solution is slower than the C++ implementations from other solutions. The main reason for that are garbage collection cycles and costly context switches between the user threads. However, the authors' solution is still efficient enough and manages a low-latency stream processing. They evaluate their system on different CPUs and hardware setups from which the best configuration on an Intel Core i7-2530M DualCore CPU at 2.9 GHz achieves an end-to-end detection latency of 0.018ms for 87.5% of all generated events. The end-to-end latency for the 99.805% percentile is at 0.145ms (best on an Intel Xeon processor). Their best configuration replays the provided test data set 48× faster than real time, which is equivalent to a throughput of 661k events per second.

Enorm [97] is a stream processing system based on MapReduce [52] that shares computation among streaming windows. Latency and throughput cannot keep up with other solutions as MapReduce has initially not been designed for the low latency processing of sensor data streams. As in the previously mentioned systems the detection tasks are implemented in a program-

ming language in order to provide the needed flexibility. The authors evaluate Enorm in the Amazon EC2 cloud with 1, 4, and 8 dedicated instances. The average end-to-end detection latency is 1,524 milliseconds and the throughput ranges from 0.8 MB/sec. (1 instance) to 5.2 MB/sec. (8 instances). Although Enorm falls far short behind the other solutions the evaluation shows that Enorm scales with an increasing number of nodes.

The TechniBall System [60] comprises *streams* [24, 25], a Java-based description language for sources, sinks, and processors together with Esper [70] as its CEP engine. TechniBall achieves the necessary performance by making the trade-off between low latency processing and query language processing. Whenever low latency or numerical efforts are required TechniBall uses the streams framework, whereas Esper kicks in for high-level reasoning tasks that do not require such big performance boosts. The authors evaluate TechniBall on an Intel Core i7-3770 at 3.4 GHz and with 16 GB of main memory. Without any processing of the data streams TechniBall reads 922k events per second; with processing query 1 and 3 together it achieves approx. 500k events per second; and with query 2 and 4 together it also results in approx. 500k events per second. Unfortunately, to process all the queries together, internal queues in TechniBall need to be shared among the query processors, which results in a lower throughput of 285k events per second. The end-to-end detection latency is between 0.002ms and 0.003ms (without emitting the events to other applications or command line but only for integrating them into the local statistics that are located in the heap).

Badiozamany et al. [16] present the EPIC (Extensible, Parallel, Incremental, and Continuous) data stream management system (DSMS). In order to provide the required high performance EPIC allows for executing user-defined functions next to their query language over the incoming streams. Queries run in different OS processes to allow for parallel execution. The authors evaluate EPIC on an Intel Core i7-2730QM CPU at 2.6 GHz with 8 GB of main memory running Windows 7 64-bit. The authors process the entire 60 minutes game in 20 minutes (all queries together), i.e., 3× real time, whereas queries 1, 2, and 4 may be processed in only 5 minutes. The end-to-end detection latency of EPIC is in order of hundreds of milliseconds.

All accepted solutions make use of a programming language implementation either to express queries properly or to achieve the required low-latency processing. From that we conclude that EDL-based approaches alone can-

not replace the functionality provided by programming language implementations. For the analysis of high data rate position sensor streams (or maybe generally spoken *spatial* data) EDL-based approaches lack the level of flexibility that is needed for the implementation of queries and for properly defining detection tasks. Another conclusion is the usage of programming language approaches for performance reasons. When comparing the efficiency of the proposed solutions in terms of detection latency and throughput, the most efficient solutions are the ones that are implemented in C/C++ as this is the most efficient way for the implementation of the EPS. Java-based solutions are slightly less efficient but still aim for event processing in orders of milliseconds which is totally acceptable for most of the tasks.

To compare our runtime system implementation to the submitted solutions of the grand challenge we also implemented the four queries for the EventCore, and measured the event throughput and end-to-end detection latency on an Intel Core i7-M620 CPU at 2.67 GHz and 8 GB of main memory. In total we used 13 event detectors to create a total number of 46 different event types. When we execute all the queries, and if the events are already located in the main memory our EventCore system processes 499k input sensor events per second in a non-threaded processing mode, i.e., it processes the input data 33× real time. Multithreading does not yet pay off its management overhead and results in less throughput as there are not so many event detectors and complex calculation tasks yet. This result is totally acceptable and is well within the balance of the submitted solutions. The average end-to-end detection latency of the four demanded queries is below 0.002ms for a 99.25% percentile, i.e., measured from inserting a sensor event into the EventCore system to the point when data is incorporated in the event detector's heap space. These results show that our custom streaming system is highly optimized for the processing of our sensor stream data, and that it can easily keep up with other presented stream processing systems.

However, considering detection latency in that way was not the initial intention for the development of our system. Our system dynamically resizes event buffers to cope with out-of-order events. For the grand challenge we provided a pre-sorted file of position events that can simply be loaded and replayed in order to focus on the novel ideas and strategies for analyzing position sensor data in general. Moreover, none of the presented solutions provides full support for distributed processing whereas our system scales efficiently across several nodes.

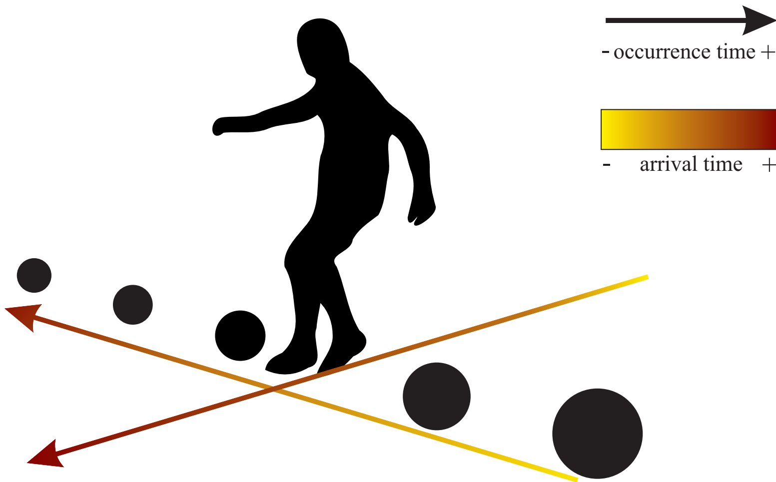


Figure 6.1.: Arrival time stamps of player and ball positions.

Summarizing, we can conclude (1) that our ordering method presented in this thesis has correctly assumed that EDL-based approaches restrict the expressibility of even detectors too much in general, (2) that C/C++ is the way to go for a fast and custom event processing system solution for high data rate requirements, (3) that the components our system is composed of, i.e., lock-free ring-buffers etc., are also commonly found in the provided solutions, and (4) that our system can easily keep up with the other presented solutions in terms of both throughput and latency.

6.2. Event Ordering Use-Case Example

This section gives a practical example that shows that runtime reordering is necessary. Assume that we want to detect that a player hits a ball, see Figure 6.1. We expect that the ball has some velocity and that the player's and ball's trajectories cross each other at the time the ball is kicked. We split this task into sub-tasks, and define the sub-events `BALL_DIRCTN_CHANGED`

(see Appendix A.2), i.e., that the ball experiences an acceleration peak, and PROXIMITY_IN/PROXIMITY_OUT (see Section 6.2.1), i.e., that the ball enters or leaves the proximity of a player. The *PlayerHitsBall*-detector (Section 6.2.2) then uses these event types to compose the PLAYER_HITS_BALL event. This has also been the running example in the Chapters 3 and 4 to motivate event ordering. In the following we show how to detect all the necessary events and how to cope with out-of-order events. We also compare the code complexity, CPU consumption, and detection latency of our middleware-based ordering approach to an event detector hierarchy that cares itself about ordering within the event detectors, see Sections 6.2.3, 6.2.4, and 6.2.5.

A small hierarchy that is used to detect the PLAYER_HITS_BALL event is shown in Figure 6.2. Since there are usually several balls tracked simultaneously we take the recognition of the active ball out of the proximity detector as this can be accomplished by a simple filtering operator.[1] However, the influence of that detector on the event ordering is negligible.

But although the event detector hierarchy only consists of 3 (without the *Active Ball* detector there would only be 2) levels there arise ordering issues at several points. As the *Ball Direction Changed* event detector on level 1 does not need to take interactions between the sensors into account but just analyzes the position streams of the balls, it will generate the events with almost negligible additional latency, i.e., an ordering is irrelevant here.

However, the *Proximity* event detector on level 2 needs to relate different position tags with each other, i.e., the tag of the active ball to the tags of all the players. As shown in Section 3.4.1 this introduces remarkable latency. Consider Figure 6.1 again. The arrows show the trajectories of the player (from top right to bottom left) and the ball (from bottom right to top left). The color gradient of the arrows encodes the arrival times of the particular position events: the arrival time increases as the color gradient turns from bright (yellow) to dark (red). The essence here is that the positions of the ball are received at lower (arrival) time stamps than the positions of the player's transmitter. At the time the trajectories cross each other in real life we also receive the ball's position it has at the crossing, i.e., we receive its positions instantly. The arrival time of the ball's position at the crossing is gray (orange). However, we receive the player's position that s/he has at the crossing

[1] We implemented the detector a bit differently as a conventional filter operator would essentially generate another high data rate event stream that would have an influence on the measurements.

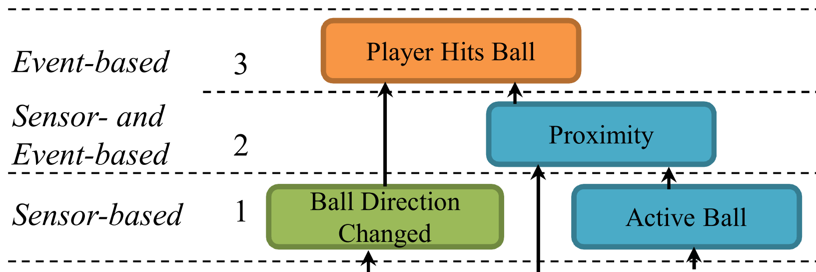


Figure 6.2.: Event hierarchy for player hits ball.

much later: we receive the position of the player sensor not before the arrival time already turned darker (red).

In the following we show details about event delays in a real system environment and show how to implement the event detectors in both ways: with and without an event ordering middleware. We point out that the implementation of event detectors without event ordering middlewares requires considerably more source code, has a higher software complexity, and causes higher CPU load than the implementation of event detectors for ordering middlewares. This section provides source code for the *Proximity* and the *Player Hits Ball* event detector. The source codes for the *Ball Direction Changed* and the *Active Ball* detectors can be found in Appendix A.2 and A.3 as they are not discussed in greater detail here. We also moved parts of the *Proximity* detector that are irrelevant for the discussion to Appendix A.4. The *Proximity* detector that uses the ordering middleware can be found in Appendix A.5.

6.2.1. Proximity detection

The *Proximity* detector subscribes the `ACTIVE_BALL` event that is used to internally set the currently active ball, i.e., the play ball, and the position data streams. The *Proximity* detector should send a `PROXIMITY_IN` event as soon as the current ball gets close to a player, i.e., if the distance between the ball's center and a player's transmitter becomes less than one meter. As soon as the ball leaves the proximity of a player the detector should generate a `PROXIMITY_OUT` event. Both events should carry the minimal time stamp of the state changes. In other words, the proximity events are based on two

```
1  typedef MAP_IT map<int, list<Position> >::iterator;
2
3  void Proximity::callback(const Event* event) {
4    Position& pos = event->position[0];
5    if (event->id == ACTIVE_BALL) {
6      active_ball = pos.obj_id;
7      return;
8    }
9    positions[pos.obj_id].push_back(pos);
10   if (pos.obj_id == active_ball) {
11     for (MAP_IT iter : positions) {
12       if (iter->first != obj_id) {
13         correlate(iter->first);
14       }
15     }
16   } else {
17     correlate(pos.obj_id);
18     PurgeObsoleteBallPositions();
19   }
20 }
```

Figure 6.3.: Proximity detector: `callback` function.

positions from different transmitters with different time stamps. The events should carry the smaller time stamp of the two as its occurrence time stamp. The event detector implementation can be found in Appendix A.5.

However, this naively implemented detector that only holds the most recently received position per transmitter does not detect the proximity events properly for unordered event streams.[2] The problems are the varying delays among the position data streams. In general, positions of the balls arrive considerably earlier than the ones of the players.[3] As the positions are packaged and the high data rate streams are prioritized the ball events are received up to one second earlier than a player's position that has an equal occurrence time stamp. An event detector that ignores the order may not detect a proximity event although it may have happened and may also generate a proximity event based on transmitters that have not actually been in the proximity of the ball.[4]

[2] Unless we use the detector in combination with an ordering middleware.

[3] Nevertheless, we may also receive player positions earlier than ball positions.

[4] Consider Figure 6.1 again. If the ball crosses a player's trajectory shortly behind him the colors, i.e., the arrival times, become equal at the *intersection* point. We receive the ball's position at the crossing when we receive player's position s/he had before.

```
void Proximity::correlate(unsigned int obj_id) {
  std::list<Position>& player = positions[obj_id];
  std::list<Position>& balllist = positions[active_ball];

  list<Position>::iterator ball_it = balllist.begin();

  // search the first position in which player[0] fits
  while(!player.empty()) {
    Position& playerpos = player[0];
    while (playerpos.ts < ball_it.ts) {
      ++ball_it;
    }
    --ball_it;

    //found nearest ball position with lower time stamp.
    int64_t time = Near( ball_it, playerpos );

    if (states[obj_id] == false && time != -1) {
      states[obj_id] = true;   // now near.
      Event e(PROXIMITY_IN, time);
      connector->Send(e);
    }

    else if (states[obj_id] == true && time == -1) {
      states[obj_id] = false;  // now un-near
      Event e(PROXIMITY_OUT, time);
      connector->Send(e);
    }
    minimal_timestamp = time;
    player.pop_front();
  }
}
```

Figure 6.4.: Proximity detector: `correlate` function.

Without an external ordering unit the event detector has to buffer the positions and must timely correlate them in order to relate them appropriately. Figure 6.3 and 6.4 show source code excerpts of the *Proximity* event detector: `callback` is invoked on event reception and `correlate` is used to relate the positions of the transmitters. The remainder of the detector's source code can be found in Appendix A.4. The event detector subscribes the position data stream and the active ball events and publishes `PROXIMITY_IN`

and PROXIMITY_OUT events. Consider the callback function provided in Figure 6.3. Whenever the event detector receives the active ball event it only uses it to set the internal active_ball variable to the new transmitter id (line 5 to 8). But if the currently received event is a position event we push it into a buffer that holds this transmitter's positions (line 9). At this point we have to distinguish two cases. First, that we receive a position of the currently active ball (line 10). Then we have to check all the other transmitters for a change of their proximity status (lines 11 to 15) by calling correlate (line 13) for each pair of transmitters. Second, that we receive a player's position (line 16). Then we only have to check whether the proximity status of this particular transmitter changes (line 17) as this does not affect the other player transmitters. PurgeObsoleteBallPositions (called in line 18) iterates over the buffered ball positions and removes the ones that are no longer needed for correlation. Obsolete player positions are purged by correlate.

The correlate function provided in Figure 6.4 takes the id of the recently received position and correlates the transmitter's buffered positions to that of the ball's positions. For that we iterate over the buffered ball positions until we find the first ball position that has a larger time stamp than the time stamp of our oldest player position (player[0]). If the transmitters are *near* but their former state has been *not near* (line 18), or the transmitter is *not near* but has been *near* right before (line 24) we observe a state change and generate the appropriate event. We can remove the player's oldest position since we correlated it to the appropriate ball position.

The detector we designed in this section can detect the proximity between transmitters and the ball as soon as the particular position events show up. The implementation does not introduce latency by active waiting for positions because we correlate the positions immediately at the time of their arrival. The implementation can cope with arbitrary event delays.[5] In Section 6.2.3, 6.2.4, and 6.2.5 we analyze code complexity, resource consumption and latency, and show the price we have to pay for that low latency detection.

6.2.2. Player hits ball detection

The player hits ball event detector is the first event detector in our hierarchy that is no longer working on high data rate sensor streams but instead only on derived events. Since the detector only subscribes to the events provided

[5] Of course, main memory is limited but the algorithm itself works correctly.

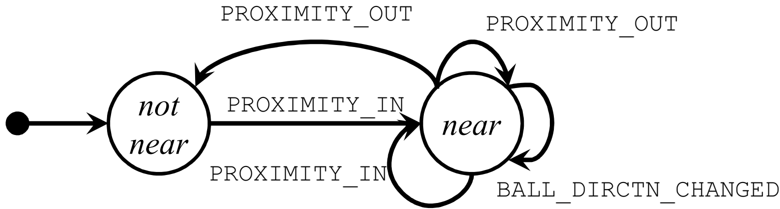


Figure 6.5.: State machine for the player hits ball detector.

by the *Proximity* detector, i.e., PROXIMITY_IN and PROXIMITY_OUT, and the *BallDirectionChanged* detector, i.e., BALL_DIRCTN_CHANGED, the processing load is already significantly reduced. The player hits ball detector uses these three events as input and generates the PLAYER_HITS_BALL event whenever a PROXIMITY_IN event is directly followed by a BALL_DIRCTN_CHANGED event. But if we receive PROXIMITY_OUT in between the BALL_DIRCTN_CHANGED event may just be caused by the ball dropping on the ground or hitting a barrier and does not let the detector generate the PLAYER_HITS_BALL event.

Figure 6.5 shows a non-deterministic finite automaton describing the internal behavior of the event detector. The detector basically implements two stages. Either it is in the *not near* state, which means that currently no player is near the ball, or the *near* state, which means that at least one player is in proximity of the ball. Hence, the detector holds a list to save all the transmitters that are near the ball. A subsequent ball hit then triggers the detector to generate the PLAYER_HITS_BALL event for all the players that may have caused it (as we cannot be sure which player it has really been).

Figure 6.6 shows the source code for the player hits ball implementation. The header file that contains data structures and other implementations used here can be found in Appendix A.6. For compactness we use a (non-) deterministic finite automata interface, see Appendix A.7, and just define functions that are called when we traverse between states. In the constructor the detector signals to subscribe the given events and the provision of the PLAYER_HITS_BALL event. The detector is implemented as a finite state machine as described in Figure 6.5 and consists of two states: `not near`, i.e., `StateA`, meaning no transmitter is currently in proximity of the ball, and `near`, i.e., `StateB`, meaning that at least one transmitter is near the ball. We define four transitions, i.e., `connect` (line 13 to 16), among the

```
1  #include "PlayerHitsBall.hpp"
2
3  PlayerHitsBall::PlayerHitsBall(EventCoreConnector* _con)
4    : connector(_con) {
5    connector->listen(BALL_DIRCTN_CHANGED);
6    connector->listen(PROXIMITY_IN);
7    connector->listen(PROXIMITY_OUT);
8    connector->publish(PLAYER_HITS_BALL);
9
10   StateA = new State("not␣near");
11   StateB = new State("near");
12
13   connect(StateA, StateB, PROXIMITY_IN, &PlayerProxIn);
14   connect(StateB, StateB, PROXIMITY_IN, &PlayerProxIn);
15   connect(StateB, StateB, BALL_DIRCTN_CHANGED, &Successful);
16   connect(StateB, StateA, PROXIMITY_OUT, &PlayerProxOut);
17   InitDFA(StateA);
18 }
19
20 bool PlayerHitsBall::PlayerProxIn(const Event* event) {
21   PlayersInProximity.push_back(event->positions[0]);
22   return true; // go to state StateB.
23 }
24
25 bool PlayerHitsBall::PlayerProxOut(const Event* event) {
26   PlayersInProximity.remove(event->positions[0]);
27   if (PlayersInProximity.empty())
28     return true; // leave state and go to StateA.
29   else
30     return false; // do not leave the state.
31 }
32
33 bool PlayerHitsBall::Successful(const Event* event) {
34   Event e(PLAYER_HITS_BALL, event->positions[0].ts);
35   e.positions.push_back(event->positions[0]);
36   e.positions.push_back(PlayersInProximity);
37   connector->Send(e);
38   return true; // irrelevant.
39 }
40
41 void PlayerHitsBall::callback(const Event* event) {
42   this->update(event); // check states and transitions.
43 }
```

Figure 6.6.: Player hits ball event detector.

```
1  PROXIMITY_IN  [#26]  @7564115828239289 @7564116333287819
2  PROXIMITY_IN  [#24]  @7564255028073944 @7564259551084405
3  BALL_DIRCTN_CHANGED  @7564302548017380
4  BALL_DIRCTN_CHANGED  @7564312148006616
5  PROXIMITY_OUT [#24]  @7564394227909809 @7564398751376446
6  PROXIMITY_OUT [#26]  @7564394227909809 @7564395693549119
7  BALL_DIRCTN_CHANGED  @7564504147779787
8  BALL_DIRCTN_CHANGED  @7564547347728870
9  BALL_DIRCTN_CHANGED  @7564691347557302
10 BALL_DIRCTN_CHANGED  @7564772947460756
11 BALL_DIRCTN_CHANGED  @7564787347444662
12 BALL_DIRCTN_CHANGED  @7564806547420849
13 PROXIMITY_IN  [#28]  @7564778227454210 @7564782913997403
14 PROXIMITY_OUT [#28]  @7564922227286291 @7564922924005943
15 BALL_DIRCTN_CHANGED  @7565315346824394
16 BALL_DIRCTN_CHANGED  @7565358546772867
17 BALL_DIRCTN_CHANGED  @7565790546265309
```

Figure 6.7.: Received events printed in order of their arrival time.

states and assign an evaluation function to these transitions. For instance, in line 13 we define that if the current is `StateA`, i.e., no player is near the ball, and we receive a `PROXIMITY_IN` event we invoke `PlayerProxIn`. If `PlayerProxIn` returns true (here this is always the case, see line 22) we walk the transition to `StateB`. If a transition function returns false we remain in the currently active state. Hence, we do not need to declare the transition from `StateB` to `StateB` on an `PROXIMITY_OUT` event since this transition is handled implicitly by the return value of the transition connected in line 16. We only walk the transition from `StateB` to `StateA` if `PlayerProxOut` returns true, i.e., if the `PlayersInProximity` list becomes empty. Otherwise there are still players near the ball and we remain in `StateB`. If we are in `StateB` and we receive `BALL_DIRCTN_CHANGED` we invoke `Successful`, which generates the event, fills in the appropriate data, i.e., the data provided by the `BALL_DIRCTN_CHANGED` event and any player that is currently in proximity, and sends the event to the middleware. The return value of the `Successful` transition function is irrelevant here (since it is implemented as a self-transition).

However, the event detector defined here misses some `PLAYER_HITS_BALL` events and may also detect false-positives because we again experience or-

dering issues. We know that the *proximity* detector delays the detection of PROXIMITY_IN and PROXIMITY_OUT events as we had to bring together the information of both transmitter types. For the proximity detector we simply buffer the positions, correlate them as soon as they show up, and purge them when they have been correlated.

But it is not as easy for the player hits ball event detector as we receive PROXIMITY_IN and PROXIMITY_OUT events next to BALL_DIRCTN_CHANGED events. Figure 6.7 shows the events printed to the command line interface (CLI) in the order of their arrival time stamps. The BALL_DIRCTN_CHANGED events are generated by the *Ball Direction Changed* detector depicted in Appendix A.2, and both the PROXIMITY_IN and PROXIMITY_OUT events originate from the event detector we implemented in Section 6.2.1. The latter carry the two time stamps of the participating transmitters of the ball and the player. The first time stamp is the occurrence time stamp, which is the earlier one of both. While we receive the first few events in a correct order (until line 12 the order is correct) we receive the PROXIMITY_IN event in line 13 out of order. The time stamp of the PROXIMITY_IN event is smaller than that of the BALL_DIRCTN_CHANGED event from the previous line #12. Their time difference is around 28 milliseconds. Using the naive implementation we proposed above, we do not receive the ball direction changed event between the two proximity events, and do hence not generate the player hits ball event as we are in StateA, i.e., *not near* when we actually receive the two BALL_DIRCTN_CHANGED events in lines 11 and 12. But actually, the time stamps tell us that the player hits ball event has actually happened.

The remaining open point is, that we need to postpone the processing of BALL_DIRCTN_CHANGED events for some *n* time units so that we can arrange them appropriately between the two proximity events. Only then the event detector walks through the correct transitions and fires for the correct PLAYER_HITS_BALL event. Another implication is that we may not send the PLAYER_HITS_BALL event directly on reception of a BALL_DIRCTN_CHANGED event as we may receive a late PROXIMITY_OUT event afterwards. Thus, to prevent a false-positive detection we also need to wait for some *n* time units to ensure that the event actually happened.

Hence, we do not only need to withhold events that lead to the creation of events, but we also need to withhold events in general to not walk invalid paths in the state machine (there are state machines that are far more complex than this example for which we cannot implement revisioning as easily).

Event detector	McCabe complexity	LoC
ActiveBall	14	37
BallDirectionChanged	8	24
InOutRegion	24	46
Proximity (w/ ordering)	**45**	**186**
Proximity (w/o ordering)	**11**	**40**
PlayerHitsBall	7	40

Figure 6.8.: Software complexities and LoC of the event detectors.

We need to withhold the processing of any event at least for *n* time units. To make that work correctly, *n* must be set to the maximal variation in the event delays to order the incoming events appropriately. `BALL_DIRCTN_CHANGED` events are detected with almost no delay whereas the proximity events are detected with a maximal delay of the delay variation in the position streams (plus some processing delay and overhead). Hence, at this point the event delays sum up to the value our ordering middleware delays the events. From within the event detector we now have no chance to order the events with lower latency as we need to guarantee that no late `PROXIMITY_OUT` event is generated.

Although event-based systems are not intended to, a strong synchronization between the two event detectors could solve the problem on a single machine. But this is inefficient if the event detectors run on different machines, see Section 3.6. Hence, the latency we saved for the detection of the proximity events is wasted for the detection of so-called negative event patterns like the one implemented in the *Player Hits Ball* event detector. In other words, the detection latency we saved for the proximity events vanishes at the detection of the `PLAYER_HITS_BALL` event and the end user will not notice any difference in the latency between the two implementations.

6.2.3. Code complexity analysis

In the following we want to take a closer look at the resulting code complexities. The implementations of both event detector hierarchies, i.e., the

one with in-detector ordering and the one with in-middleware ordering, are mostly equal – except for the implementation of the *Proximity* detector (and of course the *Player Hits Ball* detector – but here we have no chance to implement an efficient in-detector ordering). To measure code complexity we use `cppncss`[6] in version 1.0.3 using the command line arguments

```
./cppncss x86_64 -r -v -x -k *.cpp *.hpp > report.xml
```

to derive the McCabe cyclomatic complexity (CCM) [102] of the event detectors.[7] The CCM is defined by the number of linearly independent paths through a program's control flow graph, and implicitly defines the minimum number of test cases to traverse all possible paths through the source code.

The measurement results are depicted in Figure 6.8. The cyclomatic complexity of the event detectors that do not (need to) care about unordered event input is 12.8 on average. However, the *proximity* event detector that cares itself about event ordering has a complexity of 45, which is nearly four times as much as the complexity of the other detectors. This must result in four times more testing work in order to guarantee a similar software quality than its simpler version. We can draw similar results when we consider the lines of source code we need to for implementation. The event detectors that do not (need to) care about event ordering have 37.4 lines of C++ code on average (we only consider the `*.cpp` files without their header files). However, the *proximity* detector that cares itself about event ordering is implemented in 186 lines of C++ code, i.e., nearly 5 times as much as its simpler version that uses an ordering middleware.

These results clearly show that it is worth implementing the event ordering in the middleware instead of within the event detectors. If the event ordering is implemented and tested thoroughly there this then results in event detector source code that is easier to maintain, compact, and safe.

6.2.4. CPU consumption

Source code complexity affects the time and management of system development. Another measurement for performance and efficiency is the resulting

[6] `http://cppncss.sourceforge.net/`

[7] Certainly, CCM is only to a limited extent significant for the analysis of complexity in event-based systems but there is no other (more significant) measure that has been established yet.

CPU consumption, i.e., the performance of the event detector implementation. To measure this we replay a soccer match and measure the CPU loads over time for both event detector hierarchies.

When we replay the soccer position stream, the ordering middleware exhibits a comparably lazy CPU load of only 35-40% over the whole game. That is because the sensor events are buffered and ordered, and the event detectors have no further processing overhead for consuming these events. In contrast, when we turn off the ordering in the middleware and let the event detector hierarchy handle the ordering itself, the CPU load is higher, i.e., 71% on average. That is because the overhead of buffering and insertion sorting the events is lower than the correlation processing within the proximity event detector.

These results show, that an ordering middleware not only causes lower software complexity but also achieves a higher throughput than an in-detector ordering of events can achieve. However, throughput is only one side of the coin. The other side are the achievements in detection latency. Typically, throughput and detection latency compensate each other, and in many cases, we have to make a trade-off between throughput and latency. In the next section we will shed light onto that.

6.2.5. Detection latency analysis

In this section we focus on the resulting detection latency. While software complexity and CPU consumption of the ordering middleware approach outperform that of the in-detector ordering, the detection latency of an in-detector ordering may be better, see the implementation of the *Proximity* event detector in Section 6.2.1. The ordering middleware buffers all the sensor events until they are in order and then emits the events to the event detector in order. Instead, the in-detector ordering approach takes the unsorted event stream and correlates the events as soon as they show up. Hence, the event detector may generate the `PROXIMITY_IN` and `PROXIMITY_OUT` events with lower latency as it does not need to wait until K is satisfied but only until the particular events can be matched, which is – in many cases – earlier than K.

To analyze the detection latency we measure the delay with which we generate the `PROXIMITY_IN` and `PROXIMITY_OUT` events. We replay the event stream in our replica cluster, i.e., only packaging delay without transmitter prioritization. Most of the proximity events are generated by the detector

with a delay of less than 100ms. Only 10% of the events are generated with a delay of more than 100ms, and only occasional events are generated with more than 800ms delays. This is because most of the out-of-order events are not so late in compared to some other events (especially in replay mode that diminishes some out-of-order events). Instead, the proximity detector that leaves the ordering behind to the event-based middleware introduces a static latency of around 950ms. That is because the middleware measures the event delays and sets the K-value to the maximum of the measurements, i.e., to the maximal variation. Some position events arrive so late (and also lead to the creation of proximity events). Hence, the ordering detector only introduces a latency that is only a fraction of the latency that the non-ordering detector introduces.

However, these are only the resulting measurements for the *Proximity* detector. If we take a closer look at the latency of the `PLAYER_HITS_BALL` events the advantages pay off. As we have elaborated in Section 6.2.2 the *Player Hits Ball* event detector has no chance to implement an efficient in-detector ordering for the negative event pattern, i.e., it needs to wait the worst-case time to preclude the occurrence of any late `PROXIMITY_OUT` events. In fact, the worst case time the detector needs to wait is the same time the ordering middleware would withhold the events from the *Player Hits Ball* detector. Hence, a *Player Hits Ball* event detector that implements an in-detector ordering generates the events with the same latency as the detector using the ordering middleware approach.

However, more important is the fact that (in this example) the user does not care about the proximity events at all as they are only used internally to generate the more high-level event of interest, i.e., the `PLAYER_HITS_BALL` event. For the end user in-detector ordering (at this point) has no advantage but only causes higher CPU loads and needs more efforts for its implementation. The only disadvantage we have to accept when using ordering middleware approaches pays off as the events traverse the event detector hierarchy to form the top level events.

6.3. Runtime Migration Use Case Evaluation

This section provides a use case example to show that runtime migration is needed. Soccer rule violations such as handball, fouls, etc. are punished more

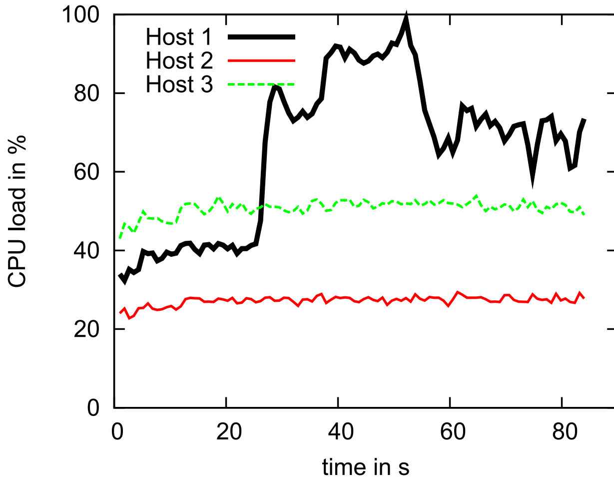
severely if players are inside the penalty area. There we not only need to process the players' position events in more detail but the event detectors are computationally intense as, for example, they (try to) derive the players' intentions. Hence, CPU loads get unbalanced, events are no longer properly ordered, and the system fails if event detectors are not migrated soon enough.

The real environment in the stadium consists of several 64-bit Linux machines, each with two Intel Xeon E5560 Quad Core CPUs at 2.80 GHz and 64 GB of main memory that communicate over a 1 GBit fully switched network.

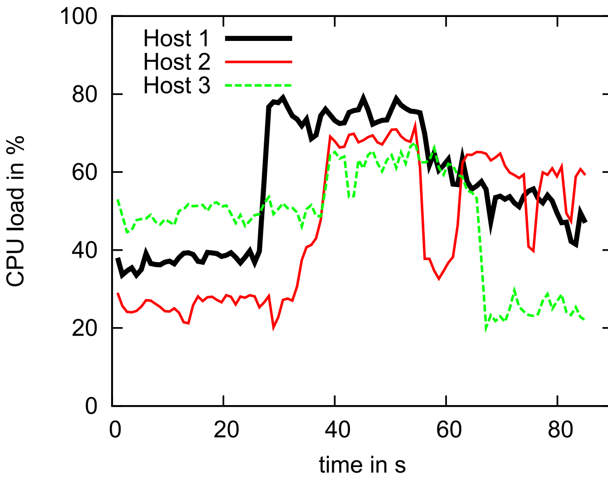
For the evaluation we pick up the penalty area use-case described in Section 5.1.1. Figure 6.9(a) shows a timeline of the CPU load on three hosts. At the beginning, event detectors are somehow distributed over the machines with a rather balanced load. After about 25 seconds, the load on host 1 rises rapidly because after a corner kick many players run into the penalty area. This triggers many event detectors and increases CPU consumption. Without migration, CPU 1 gets fully loaded and the system fails as it can no longer perform event detection although the other two machines run at a moderate load. Note that in Figure 6.9(a) the system fails at after about 53 seconds when it hits 100% load. The drop of the load afterwards is an artifact of the use case. Due to the missdetections of some event detectors, other event detectors missdetect as well and reach simpler and invalid states. The system keeps misbehaving for a long time after the peak. With migration, there is no peak and no such misbehavior. Migrating event detectors to hosts 2 and 3 smoothes load distribution, see Figure 6.9(b).

Hence, when CPU loads get unbalanced and event processing becomes critical for the performance, our technique transparently migrates event detectors that continue to process events correctly. In the test matches, it only took a few ms to migrate an event detector completely. The longest forwarding time we have seen for the migrated event detector was 2.1 seconds. On average, the data size of a migrated event detector state was 81 Bytes. The largest was 183 Bytes (player positions, time stamps of previous events).

If event detectors are migrated with techniques that run event detectors in parallel [26, 90] the loads on nodes 2 and 3 increase as in Figure 6.9(b), but node 1 behaves as in Figure 6.9(a). With classic migration based on stream forwarding [55] we see similar CPU loads as in Figure 6.9(b). However, the difference is the immense networking overhead that lasts for a long time.



(a) CPU loads without migration.



(b) CPU loads with migration.

Figure 6.9.: Evaluation results when applying migration.

6.4. Summary

This chapter focused on basic concepts of our approach and evaluated it on a broader basis.

First, we presented the results we have drawn from the ACM 2013 DEBS Grand Challenge where we carried out a competitive challenge based on our position data streams. The results confirmed our approach that event definition language based approaches cannot fit the requirements for position data stream analysis completely. Hence, a dynamic event ordering approach must also work for event detectors implemented in programming languages such as C/C++ or Java and may not use the query language functionalities to synchronize the detection components.

Second, we lined out an example for out of order event processing without an ordering middleware. With a simple and small hierarchy we illustrated different ordering issues and showed that the management of out-of-order processing from within the event detector is a hard and complex way to go as the software complexity and the code basis significantly increase. We have also shown that the higher latency introduced by buffering middlewares (in contrast to event detectors that care themselves about event ordering) becomes negligible when events traverse the event detector hierarchy.

Third, we gave a practical example why runtime migration is needed. While we have shown that our runtime migration works efficiently in the previous chapter we have shown here how we can avoid a system failure in a distributed event processing system by migrating event detectors at load peaks to load work off from overloaded event processing nodes. As a result, the system stays fully functionable and event processing continues with the usual performance and low latency.

7. Conclusion and Future Work

This thesis dealt with the challenges of low-latency event processing of high data rate sensor streams. Our presented system and methods do not require data specific a-priori configuration and do not restrict the implementation of event detectors.

The key motivation of this work was to minimize the end-to-end detection latency of events in order to trigger actions such as focusing in autonomous camera systems quickly or to present valuable information in a short amount of time. For this purpose we investigated dynamic event reordering techniques combined with a speculative processing of events to massively reduce the buffering times. Combined with solutions for challenges arising out of distributed contexts this results in fast and accurate event detection for interactive systems.

This chapter summarizes the results and contributions, highlights similar fields of applications for the methods presented in this thesis, and concludes this dissertation by pointing out directions for future research.

7.1. Conclusion

We introduced (distributed) event-based systems and worked out their advantages when dealing with complex applications that demand high scalability and maintenance. Event-based systems are usually implemented on top of publish/subscribe-systems, link detection components hierarchically, and turn the result iteratively into a top-level event of interest. At the same time, sub-information in terms of events can be reused by other detection units. Afterwards, we have shown that time synchronization and configuration are mandatory, and revealed the downside of event-based systems. These configuration, reordering, and as a result, resource utilization challenges, have been the central motivation for the development of this dissertation.

The basis of our solution is a runtime reordering technique that reorders the incoming event streams with lowest possible latency for black-box event detectors. This approach enhances the existing K-slack [135] algorithm that is well-known in event-based systems. The key idea is to deduce ordering parameters by runtime measurements and to use these parameters to relate the time stamps of the incoming events. The buffers we use to delay the events are not fixed in both their buffer size and the time intervals that they cover. As a result event detectors can be implemented without knowledge about clock skews and incorrect event order. We have also shown that this reduces their source code complexity and hence their maintenance cost massively. An initialization of reordering parameters for different system configurations can be used to correctly order event streams from the beginning of the system start.

As available CPU and memory resources cannot be used to reduce detection latency in buffering middlewares we presented a generic solution that applies speculative event processing on top of buffering middleware approaches. The novel speculation approach cannot only be used for a variety of buffering techniques but can also adaptively adjust its degree of speculation. Other existing speculative approaches inevitably lead to a system overload caused by unbounded retraction cascades. Retraction propagates throughout the entire event detector hierarchy and is not limited to an upper bound. Our technique uses event emission and replay combined with snapshot recovery and event retraction to take out the speculation complexity from the event detectors and to avoid any side-effects like memory corruptions. Our adaptation algorithm uses runtime measurements of CPU loads and adaptively fits the degree of speculation. This makes our approach so efficient: by using available system resources at runtime the evaluation has shown that we can further reduce the detection latency of any buffering approach by an average of 40%.

Finally, we complemented our contributions by focusing on event-based systems running in distributed contexts. An efficient allocation of event detectors over the available nodes is crucial for the runtime efficiency (in terms of latency) of the detection system. As this is usually not pre-configurable properly we introduced a technique to migrate event detectors at runtime. At the same time low-latency ordering parameters remain valid and any downtime of event detectors is avoided. With this runtime migration we can build an optimization unit that uses runtime event load measurements to optimize the detector allocation. This reduces introduced network load and detection latencies.

7.2. Related Fields of Application

Throughout this thesis we made use of our soccer application to motivate the development of our methods, and to show their efficiency by experiments. However, our contributions catch a broader field of applications that goes beyond soccer and related ball sports such as basketball, hockey, american football/rugby, or volleyball, where the connection between the requirements is quite obvious. As many other CEP systems, we leveraged knowledge about our physical system environment and developed properly working methods to tackle problems that arise when processing our data streams.

In the following we point out some related fields of applications where the methods we presented in this thesis can also be applied to. We show that our solution is more than a custom problem solver for a specific application field, and give an impression of the scope that our solution covers.

Automotive and driver assistance. Automotive applications include some promising use cases for our solution. As the technologies in the automotive sector are evolving so rapidly we can think of several applications.
First, consider the technological and regulatory achievements in car-to-X communication, the technological advances for seamless human-machine interfaces (HMI), and the efforts to reduce damage to persons with the help of congestion and hazard detection. However, the sensor streams of several cars, the human factors, i.e., the human behavior, and the sensors embedded in the infrastructure are not yet combined for deeper reasoning. Combining this information properly and quickly can help to make traffic much safer.
Second, consider fleet management of car rental organizations. By using GPS and other sensors in the cars we can gather information of the entire fleet at any time. Hence, the car rental organization can, for instance, optimize the utilization, detect maintenance issues, and automatically trigger actions such as changes in bookings and maintenance appointments.

Betting systems. Betting operators such as *bet365* (`http://www.bet365.com`) and *bwin* (`http://www.bwin.com`) use heavily distributed architectures to process streaming data from different sources, i.e., sensors, and publish different data streams to various endpoints, i.e., actors. The

sources include events originating from the betters, i.e., purchases of bets, and event data that is manually induced by jockeys that observe the games (for instance to temporarily stop the acceptance of bets after significant changes or impacts like goals). Events are published to web-clients, smartphone apps, or RSS-feeds.
The requirements and fundamental conditions of betting systems are similar. We see a timely synchronized backend, i.e., the hosting datacenter at the betting operator, and distributed sensors that generate events based on events and time stamps originating from that backend. Hence, the methods presented in this thesis may be used to reduce processing latency and resource consumption in the distributed system located in the backend data center.[1]

Emergency management. Emergency management is among the most popular research topics of modern urban city management. An essential challenge is to detect, to manage, and to react to hazardous situations like panics or situations that are dangerous, unsafe, or hazardous to people's health. Given a variety of different input sensors, i.e., camera signals, cellular footprints, traffic schedules, audio signals etc., it is possible to automatically detect problematic situations, and to intervene properly at an early point in time.
The methods presented in this thesis can help to dynamically optimize the distribution of detection tasks and to order the events originating from different sources. Especially in those emergency cases a low latency processing is very important.

Smart grid. Modern smart grid applications include distributed sensing at several levels including energy consumers, electricity grids, energy storage, and the points where energy is actually generated, i.e., the energy producers. As the energy stores cannot hold unlimited amounts of energy, as the network can only be loaded with a limited amount of energy, and as the producers' energy production varies the smart grid must continuously monitor the states, and must adapt to current cir-

[1] The betting example is also similar to financial applications in general. For instance, in live-trading banks continuously emit sell and buy trading offers that are rejected and/or accepted. Our methods can help to provide a faster update for price calculation and hence better offers to customers. However, we do not further focus on that as banks usually do not provide any insights to their internal data processing.

cumstances quickly. Hence, the essential challenge of mastering the smart grid is not in the distribution of the sensors or the actors, but in an intelligent processing of these data streams and in deriving proper reactions to changes.
As sensor data can be time-stamped at the point it is inserted into the network of the smart grid we can apply the methods we presented in this thesis. Local processing and adaptation is placed near the actual sensors and actors implicitly as the latency would increase when they are placed farther away. Sensor information is gathered and processed by particular event detectors, and events to control the smart grid are emitted to the actors in the smart grid. Sensor readings arrive in the processing units out-of-order and are reordered by our methods. Latency for adapting and controlling the smart grid can hence be minimized.

Sensor networks. Sensor networks draw interest in many areas as their applications range from (simple) environmental monitoring to complex pattern recognition tasks. The primary goal of implementing sensor networks is the maximization of its lifetime. As sensors are usually as small as possible, batteries must kept small and energy consumption must be minimal. Hence, as communication is the main cause of energy consumption most of the necessary processing should be performed on the particular sensor nodes.
The methods we presented can help to minimize the overall detection latency, which also minimizes the network latencies, and hence network communication itself. Moreover, sensor nodes can aggregate information and send these aggregates as our method dynamically reorders them prior to any further processing.

7.3. Future Work

This section points out directions for future research based on the results that we have achieved in this dissertation. After that, we show outstanding issues for the development of event-based systems that we experienced while implementing event detection systems. According to Etzion et al. [57] we can distinguish two major themes within the area of event processing. The first is the event-driven architecture that uses events for processing. That is

the theme that we mainly addressed in this dissertation. The second theme is about the processing operations we perform on events, i.e., the development of the event detector hierarchy. Some problems that are sufficiently well analyzed and studied for conventional software development are not yet well studied for loosely coupled event-based or publish/subscribe-systems in general.

Runtime α adaptation. The degree of speculation and hence the reduction of latency crucially depends on how α is adjusted at runtime. Basically, the α-adaptation must incorporate the distribution/variation of event delays to fit the speculation degree properly. Future research should be directed to deduce the distribution of event delays at runtime, to fit an appropriate α-adaptation function to this distribution, and to adapt α by using this dynamic function.
A second optimization for the α-adaptation is to use separate α values per event detector. Here is room for research on how we should choose α's throughout the detector hierarchy: we could either choose smaller values on lower hierarchy levels in combination with larger values on higher hierarchy (more speculation for high data rate events) or choose them vice versa.

Retraction. Apart from the α's also the retraction technique has an influence on how far we can go with speculation. That is because retraction is a crucial component in the speculative process when it comes to the retraction cascades. We presented two different retraction techniques. However, it is not easy to choose the appropriate retraction method per event detector. Although it may be a good starting point to let the systems engineer choose the technique, an automated method could also incorporate dependencies between the event detectors, their state sizes, and their affinity to change their states on particular input events. Such an approach may hence be much more efficient and result in a higher degree of speculation.

Pause partial event detector hierarchies. When the speculation technique detects a load peak we currently reset α in order to prevent system overload and failure. In contrast to this conservative strategy we could also stop parts of the detector hierarchy and give the CPU power to a subset of event detectors that benefit from lower detection latency.

As soon as the load peak is over the partially stopped detector hierarchy can be reactivated and process the events that have been buffered so far. We can, for instance, delay the processing of event detector whose events are subject to long buffer times at higher level event detectors without any problems.

Decentralizing the optimization of event detectors. The optimization of event detection latency builds on a central optimization master (OM) that recurringly triggers phases consisting of data collection, optimization, and migration. However, such a centralized approach may not be optimal for each type of application. Assume situations that lack a central component that has access to the event streams of all nodes because of mobility issues or missing links in sensor networks. In those cases a decentralized and non-optimal solution might be a better way to go. It is subject for future research to find a good online algorithm for optimization that only takes local information for the optimization of the global latency problem.

Besides the above mentioned research directions we also found open points while developing event-based systems applications, i.e., the event detectors that use the event processing middleware. In the following, we line out some software engineering issues that we identified as not sufficiently well studied yet.

It is an unresolved issue and always a subjective decision whether to split computation, i.e., an event detector, into several smaller event detectors. The basic question behind that is whether the resulting performance increases as a results of the additional possibilities for parallel execution, or if the resulting additional management overhead (scheduling queues, buffers, etc.) outweighs the parallelization and distribution benefits. Moreover, metrics we use to classify the performance of an event-based system may vary among maximal input throughput, maximal output throughput, minimal average latency, minimal maximal latency, and latency leveling [57]. Although there exists theoretical [136] and practical [5] work that shows the demand for solutions and also early results we think that the available work only hardly applies to general publish/subscribe-based systems as many information that is used by previous work is not available before runtime.

In the recent 20 years the way we develop software has massively changed. Application developers use tools for quality and complexity measurements,

rule violations, and other code analysis tools. But such tools can hardly be used for event-based systems. Although they may successfully analyze code, the resulting measurement of, for instance, complexity is in no sense representative. In this exact example, the detector hierarchy must be incorporated in order to get reliable results. Another example is how to manage a refactoring process. Indeed, there are tools that detect weak software code but they are only of little use for event-based systems. A refactoring inherently includes a reorganization of pub/sub-dependencies and code shifts. Only little attention was paid to these software engineering topics yet (to the best of the author's knowledge).

A recurring issue in event-based systems is still debugging and testing. In conventional testing frameworks both data and processing paths can be traced and even be predicted at some points. A static analysis can be enhanced by dynamic (runtime) test results to generate a holistic test report that developers can use to enhance their software. As this becomes cumbersome for parallel and distributed applications it becomes a lifetime's work for event-based systems as the detection components are only loosely coupled. A static analysis is not possible at all because in many cases the event publications and subscriptions and hence the dependencies are derived at runtime. Therefore it is difficult to provide quantitative and qualitative results for a test report up to now.

Bibliography

[1] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, CIDR 2005, pages 277–289, Asilomar, California, USA, 2005.

[2] Software AG. Software AG Acquires Apama, `http://www.softwareag.com/corporate/Company/apama.asp`, last accessed 2013-07-25.

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM International Conference on Management of Data*, SIGMOD'08, pages 147–160, Vancouver, Canada, 2008.

[4] Mert Akdere, Ugur Çetintemel, and Nesime Tatbul. Plan-based Complex Event Detection across Distributed Sources. *Proceedings of the VLDB Endowment*, 1:66–77, 2008.

[5] Shoaib Akram, Manolis Marazakis, and Angelos Bilas. Understanding and Improving the Cost of Scaling Distributed Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 290–301, Berlin, Germany, 2012.

[6] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP with Larger Initial Windows. *ACM SIGCOMM Computer Communication Review*, 28(3):41–52, 1998.

[7] Lior Amar, Amnon Barak, Ely Levy, and Michael Okun. An On-line Algorithm for Fair-Share Node Allocations in a Cluster. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pages 83–91, Washington, DC, USA, 2007.

[8] L. Amini, H. Andrade, F. Eskesen, R. King, Y. Park, P. Selo, and C. Venkatramani. The Stream Processing Core. Technical report, IBM T. J. Watson Research Center, New York, New York, USA, 2005.

[9] Mihnea Andrei, Xun Cheng, Sudipto Chowdhuri, Curtis Johnson, and Edwin Seputis. Ordering, Distinctness, Aggregation, Partitioning and DQP Optimization in Sybase ASE 15. In *Proceedings of the 2009 ACM International Conference on Management of Data*, SIGMOD '09, pages 917–924, New York, New York, USA, 2009.

[10] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Retractable Complex Event Processing and Stream Reasoning. In *Proceedings of the 5th International Conference on Rule-based Reasoning, Programming, and Applications*, pages 122–137, Fort Lauderdale, Florida, USA, 2011.

[11] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26:2003, 2003.

[12] Ilhan Aydin, Mehmet Karakose, and Erhan Akin. The Prediction Algorithm Based on Fuzzy Logic Using Time Series Data Mining Method. *Proceedings of World Academy of Science: Engineering and Technology*, 51:91–98, 2009.

[13] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, 2004.

[14] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21th ACM Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, USA, 2002.

[15] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting K-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems*, 29(3):545–580, 2004.

[16] Sobhan Badiozamany, Lars Melander, Thanh Truong, Cheng Xu, and Tore Risch. Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 325–330, Arlington, Texas, USA, 2013.

[17] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM International Conference on Management of Data*, SIGMOD '05, pages 13–24, Baltimore, Maryland, USA, 2005.

[18] Magdalena Balazinska, Yongchul Kwon, Nathan Kuchta, and Dennis Lee. Moirae: History-Enhanced Monitoring. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR 2007, pages 375–386, Pacific Grove, California, USA, 2007.

[19] Amnon Barak, Amnon Shiloh, and Lior Amar. An Organizational Grid of Federated MOSIX Clusters. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '05, pages 350–357, Cardiff, United Kingdom, 2005.

[20] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR '07, pages 363–374, Pacific Grove, California, USA, 2007.

[21] Alessandro Bassi and Geir Horn. Internet of Things in 2020: Roadmap for the Future. In *RFID Working Group of the European Technology Platform on Smart Systems Integration (EPoSS)*, 2008.

[22] Thomas Bernhardt and Alexandre Vasseur. *Esper: Event Stream Processing and Correlation*. O'Reilly, 2007.

[23] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proceedings of the 2010 ACM International Conference on Management of Data*, SIGMOD '10, pages 1093–1104, New York, New York, USA, 2010.

[24] Christian Bockermann. The Streams Framework, `http://www.jwall.org/streams`, last accessed 2013-07-30.

[25] Christian Bockermann and Hendrik Blom. The Streams Framework. Technical Report 5, TU Dortmund University, 2012.

[26] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 169–179, San Diego, California, USA, 2007.

[27] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A High-Performance Event Processing Engine. In *Proceedings of the 2007 ACM International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, Beijing, China, 2007.

[28] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed Event Stream Processing with Non-deterministic Finite Automata. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 3:1–3:12, Nashville, Tennessee, USA, 2009.

[29] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative Out-of-Order Event Processing with Software Transaction Memory. In *Proceedings of the 2nd ACM International Conference on Distributed Event-Based Systems*, DEBS '08, pages 265–275, Rome, Italy, 2008.

[30] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[31] Patrick Cerny. Evaluation und prototypische Implementierung eines Parsers für Ereignisbeschreibungssprachen in Echtzeitlokalisierungssystemen. Bachelor Thesis, Friedrich-Alexander-University Erlangen-Nuremberg, 2013.

[32] Badrish Chandramouli, Jonathan Goldstein, and David Maier. High-Performance Dynamic Pattern Matching over Disordered Streams. In *Proceedings of the VLDB Endowment*, volume 3, pages 220–231, Singapore, 2010.

[33] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM International Conference on Management of Data*, SIGMOD '03, pages 668–668, San Diego, California, USA, 2003.

[34] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*, CIDR 2003, Asilomar, California, USA, 2003.

[35] Christoffer Löffler, Christopher Mutschler, and Michael Philippsen. Evolutionary Algorithms that use Runtime Migration of Detector Processes to Reduce Latency in Event-Based Systems. In *Proceedings of the 2013 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 31–38, Torino, Italy, 2013.

[36] Christopher Mutschler. *Apparatus and Method for Transferring Event Detector Processes*. Patent Application PCT/EP2011/069159, 2011.

[37] Christopher Mutschler. *Apparatus and Method for Transmitting a Message to Multiple Receivers*. Patent Application PCT/EP2011/069169, 2011.

[38] Christopher Mutschler. *Apparatus, Method, and Computer Program for Processing Out-of-Order Events*. Patent Application EP13153525, 2013.

[39] Christopher Mutschler and Michael Philippsen. *Apparatus, Method and Computer Program for Migrating an Event Detector Process*. Patent Application PCT/EP2011/069159, 2012.

[40] Christopher Mutschler and Michael Philippsen. Learning Event Detection Rules with Noise Hidden Markov Models. In *Proceedings of the 2012 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 159–166, Erlangen, Germany, 2012.

[41] Christopher Mutschler and Michael Philippsen. Towards a Distributed Self-Optimizing Event Processing System for Realtime Locating Systems (RTLS). In *DEBS PhD Workshops, 6th ACM International Conference on Distributed Event-Based Systems*, Berlin, Germany, 2012.

[42] Christopher Mutschler and Michael Philippsen. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *Proceedings of the 27th International Conference on Parallel and Distributed Processing Symposium*, pages 1133–1144, Boston, Massachusetts, USA, 2013.

[43] Christopher Mutschler and Michael Philippsen. Dynamic Low-Latency Distributed Event Processing of Sensor Data Streams. In *Proceedings of the 25th Workshop on Parallel Systems and Algorithms*, PARS 2013, pages 5–14, Erlangen, Germany, 2013.

[44] Christopher Mutschler and Michael Philippsen. Reliable Speculative Processing of Out-of-Order Event Streams in Generic Publish/Subscribe Middlewares. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 147–158, Arlington, Texas, USA, 2013.

[45] Christopher Mutschler and Michael Philippsen. Runtime Migration of Stateful Event Detectors with Low-Latency Ordering Constraints. In *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 609–614, San Diego, California, USA, 2013.

[46] Christopher Mutschler, Nicolas Witt, and Michael Philippsen. Do Event-Based Systems have a Passion for Sports? In *Proceedings of the*

7th ACM International Conference on Distributed Event-Based Systems, DEBS' 13, pages 331–332, Arlington, Texas, USA, 2013.

[47] Christopher Mutschler, Nicolas Witt, Michael Philippsen, and Stephan Otto. *Apparatus and Method for Synchronizing Events*. Patent Application PCT/EP2011/069160, 2011.

[48] Christopher Mutschler, Norbert Franke, Daniel Wolf, and Nicolas Witt. *Apparatus and Method for Analyzing Sensor Data*. Patent Application PCT/EP2011/069166, 2011.

[49] Christopher Mutschler, Zbigniew Jerzak, and Holger Ziekow. The DEBS 2013 Grand Challenge. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pages 289–294, Arlington, Texas, USA, 2013.

[50] M. Clerc and J. Kennedy. The Particle Swarm - Explosion, Stability, and Convergence in a Multidimensional Complex Space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[51] Gianpaolo Cugola and Alessandro Margara. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, Cambridge, United Kingdom, 2010.

[52] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[53] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 627–644, Munich, Germany, 2006.

[54] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR '07, pages 412–422, Pacific Grove, California, USA, 2007.

[55] Markus Endler and Vera Nagamuta. General Approaches for Implementing Seamless Handover. In *Proceedings of the 2nd International Workshop on Principles of Mobile Computing*, pages 17–24, Toulouse, France, 2002.

[56] Opher Etzion and Jeffrey M. Adkins. Tutorial: Why is Event-driven Thinking Different from Traditional Thinking about Computing? In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 269–270, Arlington, Texas, USA, 2013.

[57] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2011.

[58] Ted Faison. *Event-Based Programming - Taking Events to the Limit*. Apress, 2006.

[59] Paul Fodor, Darko Anicic, and Sebastian Rudolph. Results on Out-of-Order Event Processing. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, pages 220–234, Austin, Texas, USA, 2011.

[60] Avigdor Gal, Sarah Keren, Mor Sondak, Matthias Weidlich, Christian Bockermann, and Hendrik Blom. Grand Challenge: The TechniBall System. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 319–324, Arlington, Texas, USA, 2013.

[61] Chris Gauthier-Dickey, Virginia Lo, and Daniel Zappala. Using n-Trees for Scalable Event Ordering in Peer-to-Peer Games. In *Proceedings of the 15th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 87–92, Washington, DC, USA, 2005.

[62] Chris Gauthier-Dickey, Daniel Zappala, Virginia Lo, and James Marr. Low Latency and Cheat-Proof Event Ordering for Peer-to-Peer Games. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 134–139, Cork, Ireland, 2004.

[63] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, Vancouver, Canada, 2008.

[64] Amirhossein Ghodrati and Shariar Lotfi. A Hybrid CS/PSO Algorithm for Global Optimization. In *Proceedings of the 4th Asian Conference on Intelligent Informatics and Database Systems*, pages 89–98, Kaohsiung, Taiwan, 2012.

[65] Anastasios Gounaris, Christos A. Yfoulis, and Norman W. Paton. An Efficient Load Balancing LQR Controller in Parallel Database Queries under Random Perturbations. In *Proceedings of the 18th International Conference on Control Applications*, pages 794 –799, Saint Petersburg, Russia, 2009.

[66] Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar, Ggiuliano Losa, Mark Mendell, Howard Nasgaard, Robert Soul'e, and Kun-Lung Wu. SPL Streams Processing Language Specification. Technical report, IBM Research, New York, New York, USA, 2009.

[67] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and Elmar Wasle. *GNSS - Global Navigation Satellite Systems: GPS, GLONASS, Galileo, and more: GPS, GLONASS, Galileo & more*. Springer Vienna, 2008.

[68] Young-Chang Hou and Ying-Hua Chang. A New Efficient Encoding Mode of Genetic Algorithms for the Generalized Plant Allocation Problem. *Journal of Information Science and Engineering*, 20(5):1019–1034, 2004.

[69] http://www.dbms2.com. IBM System S Streams, aka InfoSphere Streams, aka Stream Processing, aka Please don't call it CEP, `http://www.dbms2.com/2009/05/13/ibm-system-s-infosphere-streams-processing`, last accessed 2013-07-26.

[70] EsperTech Inc. Esper Complex Event Processing Engine, `http://esper.codehaus.org`, last accessed 2013-07-26.

[71] Hans-Arno Jacobsen, Kianoosh Mokhtarian, Tilmann Rabl, Mohammad Sadoghi, Reza Sherafat Kazemzadeh, Young Yoon, and Kaiwen Zhang. Grand Challenge: The BlueBay Soccer Monitoring Engine. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 395–300, Arlington, Texas, USA, 2013.

[72] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, New York, USA, 2006.

[73] Thomas Jakobs. Optimierung von verteilten Ereignisverarbeitungssystemen durch Umverteilung von Ereignis-Detektoren zur Laufzeit. Bachelor Thesis, Georg-Simon-Ohm Fachhochschule Nürnberg, 2012.

[74] Martin Jergler, Christoph Doblander, Mohammedreza Najafi, and Hans-Arno Jacobsen. Grand Challenge: Real-time Soccer Analytics Leveraging Low-Latency Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 307–312, Arlington, Texas, USA, 2013.

[75] Zbigniew Jerzak and Christof Fetzer. BFSiena: A Communication Substrate for StreamMine. In *Proceedings of the 2nd ACM International Conference on Distributed Event-Based Systems*, DEBS '08, pages 321–324, Rome, Italy, 2008.

[76] Pieter P. Jonker, Jurjen Caarls, Wouter J. Bokhove, Werner Altewischer, and Ian T. Young. RoboSoccer: Autonomous Robots in a Complex Environment. In *Proceedings of the 3nd International Conference on Image and Graphics*, pages 47–54, Hefei, China, 2002.

[77] John Keeney, Dominik Roblek, Dominic Jones, David Lewis, and Declan O'Sullivan. Extending Sienna to Support more Expressive and Flexible Subscriptions. In *Proceedings of the 2nd ACM International Conference on Distributed Event-Based Systems*, DEBS '08, pages 35–46, Rome, Italy, 2008.

[78] James Kennedy and Russell C. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, Perth, Australia, 1995.

[79] Arie Keren and Amnon Barak. Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster. *IEEE Transactions on Parallel and Distributed Systems*, 14(1):39–50, 2003.

[80] Rahul Khanna, Huaping Liu, and Hsiao-Hwa Chen. Self-Organization of Sensor Networks Using Genetic Algorithms. In *Proceedings of the IEEE International Conference on Communications*, pages 3377–3382, Istanbul, Turkey, 2006.

[81] Boris Koldehofe, Beate Ottenwälder, Kurt Rothermel, and Umakishore Ramachandran. Moving Range Queries in Distributed Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 201–212, Berlin, Germany, 2012.

[82] Kimberly Kuo, Rodric M. Rabbah, and Saman Amarasinghe. A Productive Programming Environment for Stream Computing. In *Proceedings of the 2nd Second Workshop on Productivity and Performance in High-End Computing*, 2005.

[83] Geetika T. Lakshmanan, Yuri G. Rabinovich, and Opher Etzion. A Stratified Approach for Supporting High Throughput Event Processing Applications. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 5:1–5:12, Nashville, Texas, USA, 2009.

[84] Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, 1978.

[85] Chuan-Wen Li, Yu Gu, Ge Yu, and Bonghee Hong. Aggressive Complex Event Processing with Confidence over Out-of-Order Streams. *Journal of Computer Science and Technology*, 26(4):685–696, 2011.

[86] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-Order Processing: A New

Architecture for High-Performance Stream Systems. In *Proceedings of the VLDB Endowment*, volume 1, pages 274–288, Auckland, New Zealand, 2008.

[87] Ming Li, Mo Liu, Luping Ding, Elke Rundensteiner, and Murali Mani. Event Stream Processing with Out-of-Order Data Arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, pages 67–74, Toronto, Canada, 2007.

[88] Jörg Liebeherr, Almut Burchard, and Florin Ciucu. Delay Bounds in Communication Networks With Heavy-Tailed and Self-Similar Traffic. *IEEE Transactions on Information Theory*, 58(2):1010–1024, 2012.

[89] Annie Liu, Michael Olson, Julian Bunn, and K. Mani Chandy. Towards a Discipline of Geospatial Distributed Event Based Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 95–106, Berlin, Germany, 2012.

[90] Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. Optimizing State-Intensive Non-Blocking Queries using Run-time Adaptation. In *Proceedings of the 23rd International Conference on Data Engineering Workshop*, ICDE '07, pages 614–623, Istanbul, Turkey, 2007.

[91] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live Migration of Virtual Machine based on Full System Trace and Replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 101–110, Garching, Germany, 2009.

[92] Mo Liu, Ming Li, D. Golovnya, E.A. Rundensteiner, and K. Claypool. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *Proceedings of the 25th International Conference on Data Engineering*, ICDE '09, pages 784–795, Shanghai, China, 2009.

[93] Christoffer Löffler. Netzwerksimulation- und -optimierung von verteilten Ereignisverarbeitungssystemen. Bachelor Thesis, Friedrich-Alexander-University Erlangen-Nuremberg, 2012.

[94] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 2001.

[95] Cristian Lumezanu, Sumeer Bhola, and Mark Astley. Utility Optimization for Event-Driven Distributed Infrastructures. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, ICDCS' 06, pages 24–33, Lisboa, Portugal, 2006.

[96] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM International Conference on Management of Data*, SIGMOD '02, pages 49–60, Madison, Wisconsin, USA, 2002.

[97] Kasper Grud Skat Madsen and Li Su Yongluan Zhou. Grand Challenge: MapReduce-Style Processing of Fast Sensor Data. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS' 13, pages 313–318, Arlington, Texas, USA, 2013.

[98] David Maier, Michael Grossniklaus, Sharmadha Moorthy, and Kristin Tufte. Capturing Episodes: May the Frame Be With You. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 1–11, Berlin, Germany, 2012.

[99] N. Marz. Storm - Distributed and Fault-Tolerant Realtime Computation, `http://www.storm-project.net`, last accessed 2013-07-30.

[100] Anurag S. Maskey and Mitch Cherniack. Replay-Based Approaches to Revision Processing in Stream Query Engines. In *Proceedings of the 2nd International Workshop on Scalable Stream Processing Systems*, pages 3–12, Nantes, France, 2008.

[101] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, 1988.

[102] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[103] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, Berlin, 2006.

[104] Thorsten Nowak and Andreas Eidloth. Dynamic Multipath Mitigation Applying Unscented Kalman Filters in Local Positioning Systems. *International Journal of Microwave and Wireless Technologies*, 3:365–372, 2011.

[105] nsnam.org. The ns-3 Network Simulator, `http://www.nsnam.org`, last accessed 2013-07-22.

[106] Dan O'Keeffe and Jean Bacon. Reliable Complex Event Detection for Pervasive Computing. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 73–84, Cambridge, United Kingdom, 2010.

[107] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 183–194, Arlington, Texas, USA, 2013.

[108] Lin Ouyang and Qing-ping Guo. A Dynamic Load Balancing Technique of Distributed Stream Processing System. In *Proceedings of the 2nd International Conference on Future Generation Communication and Networking Symposia*, pages 52–57, Hainan Island, China, 2008.

[109] Sujay Parekh, Kirsten W. Hildrum, Deepak Rajan, Joel L. Wolf, and Kun-Lung Wu. Characterizing, Constructing and Managing Resource Usage Profiles of System S Applications: Challenges and Experience. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1177–1186, Hong Kong, China, 2009.

[110] Adrian Paschke, Paul Vincent, Alex Alves, and Catherine Moxey. Tutorial on Advanced Design Patterns in Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 324–334, Berlin, Germany, 2012.

[111] Peter R. Pietzuch and Jean Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 611–618, Washington, DC, USA, 2002.

[112] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 49–60, Atlanta, Georgia, USA, 2006.

[113] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Impact Analysis for Distributed Event-Based Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 241–251, Berlin, Germany, 2012.

[114] Balan Sethu Raman. Event Processing Solutions in the Enterprise - a Waypoint on the Path to the Warehouse or the Launchpad for new Analytics Solutions (Industry Keynote). Keynote Presentation at the 6th International Conference on Distributed Event-Based Systems, Berlin, Germany, 2012.

[115] Esther Ryvkina, Anurag S. Maskey, Mitch Cherniack, and Stan Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 141–143, Atlanta, Georgia, USA, 2006.

[116] Scott Schneider, Martin Hirzel, and Buğra Gedik. Tutorial: Stream Processing Optimizations. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 249–258, Arlington, Texas, USA, 2013.

[117] Naomi Seyfer, Richard Tibbetts, and Nathaniel Mishkin. Capture Fields: Modularity in a Stream-Relational Event Processing Langauge. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*, DEBS '11, pages 15–22, New York, New York, USA, 2011.

[118] Kwang Mong Sim and Weng Hong Sun. Ant Colony Optimization for Routing and Load-Balancing: Survey and new Directions. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 33(5):560–572, 2003.

[119] Robert Soulé, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic Expressivity with Static Optimization for Streaming Languages. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 159–170, Arlington, Texas, USA, 2013.

[120] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *Proceedings of the 23rd ACM Symposium on Principles Database Systems*, PODS '04, pages 263–274, Paris, France, 2004.

[121] Stephan Otto, Thorsten Edelhäußer, Nicolas Witt, Matthias Völker, David Voll, and Christopher Mutschler. *Apparatus, Method, and Computer Program for Providing a Virtual Boundary*. Patent Application EP13156961, 2013.

[122] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 Requirements of Real-time Stream Processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[123] Toyotaro Suzumura, Toshiaki Yasue, and Tamiya Onodera. Scalable Performance of System S for Extract-Transform-Load Processing. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 7:1–7:14, Haifa, Israel, 2010.

[124] Meyer Tanuan. Using Sybase WorkSpace to Build Service Oriented Architecture (SOA) Applications Quickly. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 848–849, Montreal, Canada, 2007.

[125] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIT: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, United Kingdom, 2002.

[126] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. High Performance Alternative to Bounded Queues for Exchanging Data between Concurrent Threads, `http://code.google.com/p/disruptor`, last accessed 2013-07-17.

[127] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[128] Roberto Vezzani and Rita Cucchiara. Event Driven Software Architecture for Multi-Camera and Distributed Surveillance Research Systems. In *Proceedings of the 2013 IEEE International Conference on Computer Vision and Pattern Recognition Workshops*, CVPRW '10, pages 1–8, 2010.

[129] Thomas von der Grün, Norbert Franke, Daniel Wolf, Nicols Witt, and Andreas Eidloth. A Real-time Tracking System for Football Match and Training Analysis. In *Microelectronic Systems*, pages 199–212. Springer Berlin, 2011.

[130] Rohit Wagle, Henrique Andrade, Kirsten Hildrum, Chitra Venkatramani, and Michael Spicer. Distributed Middleware Reliability and Fault Tolerance Support in System S. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*, DEBS '11, pages 335–346, New York, New York, USA, 2011.

[131] Di Wang, Elke A. Rundensteiner, and Richard T. Ellison, III. Active complex event processing over event streams. *Proceedings of the VLDB Endowment*, 4(10):634–645, 2011.

[132] Song Wang and Elke Rundensteiner. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology*, pages 299–310, Saint Petersburg, Russia, 2009.

[133] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines through Client Speculation. In *Proceedings of*

the 6th USENIX Symposium on Networked Systems Design and Implementation, pages 245–260, Boston, Massachusetts, USA, 2009.

[134] Georg Wittenburg, Normal Dziengel, Christian Wartenburger, and Jochen Schiller. A System for Distributed Event Detection in Wireless Sensor Networks. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, pages 94–104, Stockholm, Sweden, 2010.

[135] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-Performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM International Conference on Management of Data*, SIGMOD '06, pages 407–418, Chicago, Illinois, USA, 2006.

[136] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing Stateful Operators in a Distributed Stream Processing System: How, should you and How Much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 278–289, Berlin, Germany, 2012.

[137] Yingjun Wu, Qian Lin, David Maier, and Kian-Lee Tan. Grand Challenge: SPRINT Stream Processing Engine as a Solution. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 301–306, Arlington, Texas, USA, 2013.

[138] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 775–786, Seoul, Korea, 2006.

[139] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 791–802, Tokyo, Japan, 2005.

[140] Xin-She Yang and Suash Deb. Cuckoo Search via Lévy Flights. In *Proceedings of the 2009 World Congress on Nature & Biologically Inspired Computing*, pages 210–214, Coimbatore, India, 2009.

[141] Alex King Yeung Cheung and Hans-Arno Jacobsen. Dynamic Load Balancing in Distributed Content-Based Publish/Subscribe. In *Proceedings of the 7th International Conference on Middleware*, pages 141–161, Melbourne, Australia, 2006.

[142] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *Bulletin of the Technical Committe on Data Engineering*, 26(1):3–10, 2003.

[143] Yongluan Zhou, Karl Aberer, and Kian-Lee Tan. Toward Massive Query Optimization in Large-Scale Distributed Stream Systems. In *Proceedings of the 9th International Conference on Middleware*, pages 326–345, Leuven, Belgium, 2008.

[144] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *Proceedings of the 2006 International Conference on On the Move to Meaningful Internet Systems*, pages 54–71, Montpellier, France, 2006.

List of own Publications

Conference Proceedings

Parts of the contents of this dissertation have previously been published in conference proceedings. Those contributions can be found in [35, 40, 41, 42, 43, 44, 45, 46, 49].

Filed patents

Parts of this dissertation have been filed as patents. The patent applications are registered under [36, 37, 38, 39, 47, 48, 121].

Student Thesis

Parts of this dissertation have been supported and co-worked by some of my students. I would like to thank Thomas Jakobs [73], Christoffer Loeffler [93], and Patrick Cerny [31], whose contributions supported the development of this thesis.

List of Acronyms

ACK	Acknowledgement
ACO	Ant Colony Optimization
AEP	Already Emitted Pointer
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AWG	Arbitrary Waveform Generator
CAS	Compare and Swap
CEDR	Complex Event Detection and Response
CEP	Complex Event Processing
CLI	Command Line Interface
CPU	Central Processing Unit
CS	Cuckoo Search (Algorithm)
DDS	Data Distribution Service
DEBS	Distributed Event-Based System
DFA	Deterministic Finite Automaton
DSMS	Data Stream Management System
EA	Evolutionary Algorithm
EBP	Event-Based Programming

EBS Event-Based System

ED Event Detector

EDA Event-Driven Architecture

EDL Event Definition / Description Language

EKF Extended Kalman Filter

EPL Esper Processing Language

EPS Event Processing System

EPTS Event Processing Technical Society

FIFA Fédération Internationale de Football Association
International Federation of Association Football

FPGA Field Programmable Gate Array

FSM Finite State Machine

GA Genetic Algorithm

GPS Global Positioning System

GPU Graphics Processing Unit

GUI Graphical User Interface

HMI Human Machine Interface

IC Integrated Circuit

ID Identificator

ISM Industrial, Scientific, and Medical

JIT Just-in-time (compilation)

JMS Java Message Service

JSON JavaScript Object Notation

LAN	Local Area Network
LOS	Line of Sight
MMOG	Massively Multiplayer Online Game
MOM	Message-oriented Middleware
NFA	Non-deterministic Finite Automaton
OM	Optimization Master
OS	Operating System
PE	Processing Element
PSO	Particle Swarm Optimization
Pub/Sub	Publish/Subscribe (Pattern)
QoL	Quality of Location
RAM	Random-Access Memory
RFID	Radio-Frequence Identification
RPC	Remote Procedure Call
RSS	Really Simple Syndication (formerly: Rich Site Summary)
RTLS	Real-time Locating System
SN	Sensor Network
SOA	Service-Oriented Architecture
SPC	Stream Processing Core
SPL	Stream Processing Language
STM	Software Transactional Memory
TCP	Transmission Transport Protocol

TDoA Time Difference of Arrival

ToA Time of Arrival

UDP User Datagram Protocol

UKF Unscented Kalman Filter

VM Virtual Machine

XML Extensible Markup Language

A. Source Code Excerpts

A.1. Lock-Free Ringbuffer

```
#include <Queue.hpp>
#include <stdint.h>

namespace bipc = boost::interprocess::ipcdetail;
namespace queue {
  Queue::Queue(size_t _max_size, unsigned char* _mem) :
    max_size(_max_size) , mem (_mem) { }
  Queue::~Queue() { }

  uint32_t Queue::AddElement
          (const uint32_t* buf, size_t length) {
    uint32_t idx = 0, new_idx = 0, cur_idx = 0;
    do {
      cur_idx = *((uint32_t*)mem);
      if ( (cur_idx + 4 + length + 4 + 4 > max_size) ) {
        new_idx = 0;
        idx = bipc::atomic_cas32(  (uint32_t*)mem,
                                   length+4, cur_idx);
      } else {
        new_idx = cur_idx + length + 4;
        idx = bipc::atomic_cas32(  (uint32_t*)mem,
                                   new_idx, cur_idx);
      }
    } while(idx != cur_idx);
    *((uint32_t*)&mem[8+idx]) = length;
    memcpy(mem + idx + 8 + 4, buf, length);
    return idx;
  }

  DataElement Queue::operator[] (uint32_t idx) const {
    unsigned char* ptr = mem + 8 + idx;
    return DataElement(ptr + 4, *((uint32_t*)ptr));
  }
}
```

A.2. Acceleration Recognition Detector

```
1  #include <eventcore/eventdetector/IEventDetector.hpp>
2
3  class  BallDirectionChanged {
4  public:
5    BallDirectionChanged(CoreConnector *_con)
6      : connector(_con) {
7      connector->listen(EVENT_POSITION);
8      connector->publish(BALL_DIRCTN_CHANGED);
9      filter.setMaxWindowSize(10); // 10 pos. in filter
10     filter.setPeakThreshold(55000); // 55 m/s^2
11   }
12
13   BallDirectionChanged::callback(const Event* event) {
14     foreach (Position p : event->positions) {
15       uint64_t peak_time = 0;
16       if (filter.add(p, &peak_time)) {
17         Event e(BALL_DIRCTN_CHANGED, peak_time);
18         connector->Send(e);
19       }
20     }
21   }
22
23 private:
24   Global::MeanFiler filter;
25 };
```

A.3. Active Ball Detector

```
1  #include <set>
2  #include <eventcore/eventdetector/IEventDetector.hpp>
3
4  class ActiveBall {
5  public:
6    ActiveBall(CoreConnector *_con)
7      : connector(_con) {
8      connector->listen(EVENT_POSITION);
9      connector->publish(ACTIVE_BALL);
10         last_active_ball = -1;
11   }
12
13   ActiveBall::callback(const Event* event) {
14     Position& pos = event->position[0];
15     unsigned int id = pos.obj_id;
16     if (!Global::GetBallTransmitters().contains(id)) {
17       return;
18     }
19     bool in = Global::BoundingBoxPlayingField(pos);
20     if (in && !balls_in_field.contains(id)) {
21         balls_in_field.add(id);
22         if (balls_in_field.size() == 1 &&
23             last_active_ball != id) {
24           last_active_ball = id;
25           Event e(ACTIVE_BALL, pos.ts);
26           connector->Send(e);
27         }
28       }
29     }
30     if (!in) {
31       balls_in_field.remove(id);
32     }
33   }
34
35 private:
36   std::set<unsigned int> balls_in_field;
37   int last_active_ball;
38 };
```

A.4. Proximity Detector (w/o ordering middleware)

```
1   #include <map>
2   #include <list>
3   #include <eventcore/eventdetector/IEventDetector.hpp>
4
5   class Proximity {
6   public:
7     Proximity(EventCoreConnector* _con) : connector(_con) {
8       connector->listen(EVENT_POSITION, EC_FLAG_UNSORTED);
9       connector->listen(EVENT_ACTIVE_BALL, EC_FLAG_UNSORTED);
10      connector->publish(PROXIMITY_IN);
11      connector->publish(PROXIMITY_OUT);
12      minimal_timestamp = 0;
13    }
14
15    void Proximity::callback(const Event* event);
16
17    void Proximity::correlate(unsigned int obj_id);
18
19    int64_t Near(Position& p1, Position& p2) {
20      if (abs(p1.x - p2.x) > 1 || abs(p1.y - p2.y) > 1) {
21        return -1;
22      }
23      else if (sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2))<1) {
24        return (p1.ts < p2.tx) ? p1.ts : p2.ts;
25      } else {
26        return -1;
27      }
28    }
29
30    void PurgeObsoleteBallPositions(void) {
31      std::list<Position>& ball = positions[active_ball];
32      while(ball.front().ts < minimal_timestamp) {
33        ball.pop_front();
34      }
35    }
36
37  private:
38    std::map<unsigned int,list<Position> > positions;
39    std::map<unsigned int,bool> states;
40    unsigned int active_ball;
41    int64_t minimal_timestamp;
42  };
```

A.5. Proximity Detector (w/ ordering middleware)

```
1  #include <map>
2  #include <list>
3  #include <eventcore/eventdetector/IEventDetector.hpp>
4
5  class Proximity {
6  public:
7    Proximity(EventCoreConnector* _con) : connector(_con) {
8      connector->listen(EVENT_POSITION);
9      connector->listen(EVENT_ACTIVE_BALL);
10     connector->publish(PROXIMITY_IN);
11     connector->publish(PROXIMITY_OUT);
12   }
13
14   void Proximity::callback(const Event* event) {
15     Position& pos = event->positions[0];
16     positions[pos->obj_id] = pos;
17     if (event->id == EVENT_ACTIVE_BALL) {
18       active_ball = pos.obj_id;
19     } else if (pos.obj_id == active_ball) {
20       for (MAP_IT iter: positions) {
21         if (iter->first != obj_id) {
22           correlate(iter->first);
23         }
24       }
25     } else {
26       correlate(pos.obj_id);
27     }
28   }
29
30   void correlate(unsigned int obj_id) {
31     int64_t time = Near(positions[active_ball],
32                         positions[obj_id]);
33
34     if (states[obj_id] == false && time != -1) {
35       states[obj_id] = true;
36       Event e(PROXIMITY_IN, time);
37       connector->Send(e);
38     } else if (state[obj_id] == true && time == -1) {
39       states[obj_id] = false;
40       Event e(PROXIMITY_OUT, time);
41       connector->Send(e);
42     }
```

```
43      }
44
45      int64_t Near(Position& p1, Position& p2) {
46        if (abs(p1.x - p2.x) > 1 || abs(p1.y - p2.y) > 1) {
47          return -1;
48        }
49        else if (sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2))<1) {
50          return (p1.ts < p2.tx) ? p1.ts : p2.ts;
51        } else {
52          return -1;
53        }
54      }
55
56    private:
57      std::map<unsigned int, Position> positions;
58      std::map<unsigned int,bool> states;
59      unsigned int active_ball;
60    };
```

A.6. Player Hits Ball Detector

```
#include <map>
#include <list>
#include <eventcore/eventdetector/IEventDetector.hpp>

class PlayerHitsBall {
public:
  PlayerHitsBall(EventCoreConnector* _con);

  ~PlayerHitsBall(void) {
    delete StateA;
        delete StateB;
  }

  void callback(const Event* event);

  bool PlayerProxIn(const Event* event);
  bool PlayerProxOut(const Event* event);
  bool Successful(const Event* event);

private:
  State* StateA;
  State* StateB;

};
```

A.7. Non-deterministic Finite Automaton

```
1  #include <vector>
2  #include <eventcore_headers/Types.hpp>
3  #include <eventcore_headers/Serializable.hpp>
4  #include <eventcore_headers/Serializer.hpp>
5  #include <DetectorConnector.hpp>
6
7  class DetectorConnector;
8  using namespace std;
9
10 template <class T>
11 class NFA {
12   public:
13   class State {
14     public:
15       State(const char* _name, void (T::*in)
16         (const Event*), void (T::*out)(const Event*))
17         : In (in), Out(out), name (_name) { }
18
19           void (T::*In)(const Event* event);
20       void (T::*Out)(const Event* event);
21       const char* name;
22     };
23
24   class Transition {
25     public:
26     Transition(State* _source, State* _target,
27       unsigned int _event_id, T* _obj,
28       bool(T::*_fn)(const Event*))
29       : source(_source), target(_target),
30         event_id(_event_id), obj(_obj), fn (_fn) {
31     }
32
33     State* source;
34     State* target;
35     unsigned int event_id;
36     T* obj;
37     bool (T::*fn)(const Event*);
38   };
39
40   void InitDFA(State* start) {
41     this->start = start;
42     this->current = start;
43   }
44
```

```
45    void update(const Event* event) {
46      for (uint32_t i = 0; i < transitions.size(); i++) {
47        if ((transitions[i]->source == current) ) {
48          if ((transitions[i]->event_id == 0)  ||
49            (transitions[i]->event_id == event->id)) {
50            if (transitions[i]->fn != NULL) {
51              if (CALL_MEMBER_FN(transitions[i]->obj,
52                transitions[i]->fn)(event) == false) {
53                continue;
54              }
55            }
56            if (current->Out != NULL) {
57              CALL_MEMBER_FN(transitions[i]->obj,
58                             current->Out)(event);
59            }
60            current = transitions[i]->target;
61            if (current->In != NULL) {
62              CALL_MEMBER_FN(transitions[i]->obj,
63                             current->In)(event);
64            }
65            break;
66          }
67        }
68      }
69    }
70
71    void connect(State* source, State* target,
72      uint32_t event_id, bool (T::*test)(const Event*) ) {
73      Transition* trans = new Transition(source, target,
74                                         event_id, (T*)this, test);
75      transitions.push_back(trans);
76    }
77
78    bool isInState(State* state) {
79      if(state == current) {
80        return true;
81      } else {
82        return false;
83      }
84    }
85
86    ~DFA() {
87      for (unsigned int i = 0; i < transitions.size(); i++) {
88        delete transitions[i];
89      }
90      transitions.clear();
```

```
91      }
92
93      void serialize(Serializer& s) {
94        s.serialize(start);
95        s.serialize(current);
96      }
97
98      void deserialize(Serializer& s) {
99        s.deserialize(start);
100       s.deserialize(current);
101     }
102
103     private:
104       State* start, *current;
105       vector<Transition*> transitions;
106   };
```

Index

A

a-priori 35, 36, 47, 116
a-priori knowledge 31, 33, 38
acceleration 34
access 13
action 2
advertisement 13, 18
aggregate 2
aggregation 24
algorithm 39
antenna 17, 28
API 3
application developer 33
application-specific 35, 38
arrival time 40
arrival time stamp 40
ASIC 29
AWG 29

B

back-setting delay 32, 35, 47
behavior 44
buffer 36
buffering 35
burst 26

C

candidate 42
CEP 14, 24, 34
change 39
client 1
clock
    global 32, 36, 42
    internal 42
    local 32, 35
code 34
communication 118
compile time 47
complex event pattern 17
computation 35
configuration 16, 18, 26, 41
conservative 38
consumer 12
correctness 33, 39, 44
correlation 31
counter-measure 40
CPU 28
critical 47
Cuckoo Search 116

D

data
    sensor 118
DDS 17
definition 39
delay 31, 42, 44
delay
    back-setting 32, 35, 47
dependency 35

detection error 40
detection probability 40
detector hierarchy 37, 41
detector-specific 43
developer 33
disorder 40, 47
distributed 31, 47
distributed system 40, 42
distribution 32

E

EA 116
EBS 2, 14, 31, 116
EDL 17, 34
efficiency 48, 116
encapsulated 18
energy 1
EPL 24
EPS 14
error-prone 33
Esper 24
estimation 33
event
  pseudo 16, 41
  sensor 40, 42
event definition 34
event detector . . . 12, 13, 31–33, 37, 39, 40, 45, 47
event stream 43
event-based programming 12
expression 24
expressiveness 34

F

failure 33
filter 2
finances 31
FPGA 28
frequency 40

G

generic 18
global clock 36, 42
granularity 31
graph 35
greedy 118

H

heuristics 118
hierarchy 35, 37
hybrid 34

I

IC 26
implementation 43
in-order placement 35
individual 37
inequation 39
initialization 33
input stream 36
insertion sort 44
interface 24
ISM 26

J

Java 24

K

K-slack 31, 32, 38, 42, 47
K-value 37
Kalman filter 28
knowledge
  global 118

L

latency 1, 33, 38, 40, 47, 48, 116
local clock 35
logistics specialist 1
LOS 28

M

machine 31, 43
mapping 35
MapReduce 141
measurement 40
measurement
    performance 118
media 2
memory 1
middleware 17, 31, 32, 34, 42

N

network
    ad-hoc 117
    sensor 1
network structure 35
node
    sensor 1
notification 12
notification service 14
NP-hard 118

O

object behavior 33
offside 33
on-the-fly 1
optima
    local 118
optimizer 118
ordering unit 18, 36, 42, 43
out-of-order 31–33, 40, 42, 47

P

parameter 33
pattern 34
peak 34
performance scale-up 31
precise 42
processor 33
producer 12
programming language 34
pseudo code 39, 43
pseudo event 16, 41, 44
PSO 116
publication 17, 35

Q

quality 28
query 35

R

reactive 36
real-time 13
Real-time Locating System 1
reconfiguration 13
RedFIR 26
reduction 35
register 13
reordering 35
retrofit 41
RFID 1, 17
ring-buffer 49
robot 117
router 33
RTLS 1, 26, 31, 33
rule violation 117
runtime 13, 33, 34, 47
runtime measurement 38
runtime state 35

S

safety margin . . . 40
scalability . . . 2
scaling factor . . . 40
sensor event . . . 40, 42
sensor reading . . . 117
sequence . . . 44
sequential . . . 31
single-threaded . . . 35
situation . . . 1
slack buffer . . . 31, 47
sliding window . . . 36
sports . . . 31, 33
stability . . . 33
stable . . . 40
standard deviation . . . 40
startup . . . 44
stateful . . . 13
stream . . . 35
stream
    event . . . 31, 118
    input . . . 36
    sensor . . . 1, 31
stream application . . . 42
sub-stream . . . 41
subscription . . . 13, 17, 35
surveillance . . . 1, 31
synchronization . . . 32
system
    camera . . . 2
    distributed . . . 42
    event-based . . . 2
system architect . . . 40
system failure . . . 31
system topology . . . 47

T

TDoA . . . 26
technology . . . 2
thread . . . 31
throughput . . . 48
time
    response . . . 1
time stamp . . . 33, 40, 41
time unit . . . 35
time-of-flight . . . 26
timing delay . . . 34
timing offset . . . 32
ToA . . . 26
topology-specific . . . 38
total order . . . 33
transform . . . 2
translator . . . 33
transmission . . . 118

V

variation . . . 40

W

wall-clock . . . 33, 40
warehouse . . . 1
workload . . . 33

Curriculum Vitae

NAME, VORNAME	Mutschler, Christopher
GEBURTSDATUM	22. August 1986
GEBURTSORT	Nürnberg
STAATSANGEHÖRIGKEIT	deutsch
FAMILIENSTAND	verheiratet

09/1992 – 07/1996	Grundschule Cadolzburg
09/1996 – 07/2005	Wolfgang-Borchert-Gymnasium Langenzenn
09/2002 – 10/2006	Telefonist (Andreae-Noris-Zahn AG)
05/2003 – 10/2007	Kundendienstannahme (Aventi Automobile AG)
10/2005 – 05/2010	Studium der Informatik an der Friedrich-Alexander-Univ. Erlangen-Nürnberg
06/2007 – 02/2009	Softwareentwickler (Continental)
10/2008 – 03/2009	Vorlesungsbetreuung (Lehrstuhl Informatik 9)
11/2008 – 04/2009	Studienarbeiter (Elektrobit Automotive GmbH)
06/2009 – 11/2010	Wiss. Hilfskraft (Fraunhofer IIS)
05/2010	**Diplom in Informatik**
11/2010 – 11/2013	Wiss. Mitarbeiter am Lehrstuhl für Informatik 2 (Programmiersysteme)
2014	**Promotion**

Christopher Mutschler
Erlangen, 2014