UNIVERSITÄT BASEL

PAN - A P2P Approach for Scalable Complex Event Detection in Distributed Data Streams

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Databases and Information Systems (DBIS) Group
http://dbis.cs.unibas.ch/

Examiner: Prof. Dr. Heiko Schuldt
Supervisor: Ivan Giangreco, M.Sc.

Lukas Probst
lukas.probst@unibas.ch
09-050-402

August 8, 2014

UNI
BASEL

Acknowledgments

First, I would like to thank Prof. Dr. Heiko Schuldt for the opportunity to write my master thesis in the DBIS group and his great feedback in our meetings. Special thanks go to my supervisor, Ivan Giangreco. His feedback, proofreading and especially our regular controversy and inspiring discussions were very helpful. In addition, I want to thank all the other nice people who supported me with feedback and thought-provoking impulses.
Moreover, I would like to take this opportunity to thank my family, my friends and especially my girlfriend Lea for supporting me during the last labor intensive months.

Abstract

In the last decade, the number of data streams and the volume of streamed data has increased enormously. With this trend, the importance of detecting complex events in data streams in real-time has increased as well. Solving this problem is important for many economical as well as entertainment (e.g., sport analyses) use cases.

In this thesis, we present PAN (P2P Analysis Network). PAN is a generic real-time complex event detection system which is able to analyze multiple distributed input data streams and handle several client requests.

In order to be scalable, PAN distributes its workload onto several workers hosted on peers in a P2P network, which are combined to a workflow. This general idea is not novel but used by many distributed complex event processing (CEP) systems. However, PAN uses a *pull-based* - instead of the common *push-based* - publish/subscribe approach to connect the workers and thereby inverts the workflow definition direction. This fundamental difference enables the dynamic extension of the workflow at runtime without changing the existing workflow. In consequence, PAN is able to handle clients as sinks of a workflow and balance the load onto multiple publishers. This makes PAN scalable not only in terms of data but also w.r.t. the number of client requests.

Evaluations based on an extended version of the ACM DEBS 2013 Grand Challenge scenario confirm that the PAN approach works well, i.e., that it is possible to combine the workers of a real-time complex event detection system to a workflow by means of a pull-based publish/subscribe system.

Table of Contents

Abbreviations

CEP	Complex Event Processing
DEBS	Distributed Event-Based Systems
DEBS'13	7th ACM International Conference on Distributed Event-Based Systems
FIFA	International Federation of Association Football
FPGA	Field Programmable Gate Array
P2P	Peer to Peer
PAN	P2P Analysis Network
REST	Representational State Transfer
W2W	Worker to Worker
WTT	WeakTrueTime

Figure Legend

General Communication Distinction:

→	Intra-PAN Communication
⇢	Inter-PAN Communication

Communication in exemplary Figures:

🟦 ⇢	Real-World Sensor Data Input Stream
🟫 →	Forwarded Real-World Sensor Data Stream
🟩 →	(Intermediate) Statistical Data Stream between two Workers
🟩 ⇢	Statistical Data Stream between Worker and Client
🟥 ⇢	Client Request (no Stream)

Symbols:

◉	(Simulated) Sensor
🧍	Client
✹	Algorithm

Abbreviations:

LL_A2	Left Leg Player A2 (i.e., SENSOR47)
RL_B1	Right Leg Player B1 (i.e., SENSOR61)
BP_A1	Ball Possession Player A1
BP_A	Ball Possession Team A
HM_B1	Heat Map Player B1
W1	Worker 1

Acknowledgements:

The soccer field graphic in our figures are based on a Wikipedia offside graphic[1]. We want to thank the Wikipedia user NielsF for creating this graphic and publishing it under the "copyleft" license.

[1] NielsF's offside graphic: http://de.wikipedia.org/wiki/Datei:Offsidelarge.svg (07.08.2014)

1 Introduction

In the last ten years, the number of data streams and the volume of streamed data has increased enormously. With this trend, the importance of detecting complex events in data streams in real-time has increased as well. Solving this problem is important for many economical as well as entertainment (e.g., sport analyses) use cases. Hence, more and more solutions for this problem arise (e.g., Amazon Kinesis [1]).
However, in many cases, these systems lack an evaluation base. For this purpose, the DEBS Grand Challenge series was introduced. The main purpose of these challenges is to provide an evaluation base for Complex Event Processing (CEP) systems. The ACM DEBS 2013 Grand Challenge [2, 3] defines a scenario for real-time event detection in a soccer match. More precisely, the task is to generate several continuous data streams with statistics about the ongoing soccer match (according to specified queries) given a single input data stream. To make the scenario more realistic and generic, we modify it by introducing two extensions. First, we change the setting from one input data stream to multiple distributed input data streams. Therefore, we simulate each transmitter as an independent sensor which generates and sends its own data stream. Second, we further introduce the problem of handling and answering multiple parallel client requests.
In the context of the ACM DEBS 2013 Grand Challenge six solutions ([4], [5], [6], [7], [8] and [9]) were published. Although, most of these solutions are generic, they do not face our proposed scenario extensions. None of the solutions can handle multiple distributed input data streams. Moreover, most solutions further disregard client requests or provide only rudimentary solutions. In addition, most solutions are not scalable. In fact, only a single solution is evaluated, and thus shown to be able to run, in a distributed manner.
In this thesis, we propose PAN (P2P Analysis Network). PAN is a scalable generic solution for real-time complex event detection in distributed data streams. PAN's architecture is based on the workflow-based architecture idea proposed by Jergler et. al. [6]. Jergler et. al. connect different workers with non-blocking ring buffers and thereby generate a workflow consisting of a sequential and parallel arrangement and connection of workers. Jergler et. al. further state that their idea can be distributed using the publish/subscribe concept. However, they present only a centralized solution and do not describe in detail how the distribution can be performed. Moreover, they do not face our scenario extensions.

With PAN, we pursue Jergler's architecture idea. More precisely, we extend and improve it by means of implementing it in a distributed way and thereby transform the idea into a very scalable approach. In addition, we solve our scenario extensions. PAN distributes its workload onto several workers which are distributed onto multiple peers in a P2P network. These workers are combined to a workflow using a *pull-based* publish/subscribe approach instead of the common *push-based* approach. The major advantage of the pull-based approach is that it changes the workflow definition direction and thereby enables the adaption of the workflow during runtime. As a result, PAN is flexible and scalable in terms of both, data and client requests.
In the end of this thesis, we leverage the extended grand challenge scenario to evaluate PAN's applicability, scalability and performance characteristics.
The remainder of this thesis is organized as follows. Section 1.1 presents the ACM DEBS 2013 Grand Challenge as well as our scenario extensions in detail. The problem statement is given in Section 1.2. Chapter 2 presents and discusses the published solutions for the ACM DEBS 2013 Grand Challenge. We present our solution to the evaluation problem in Chapter 3. Chapter 4 presents PAN. An evaluation of the sensor simulation environment and PAN is given in Chapter 5. Finally, Chapter 6 presents related work and Chapter 7 concludes.

1.1 Motivation Scenario

The motivation scenario for this thesis is based on the ACM DEBS 2013 Grand Challenge. Section 1.1.1 presents the grand challenge scenario, i.e., the setting as well as the requirements and the specified queries, in detail. Although the grand challenge scenario is a remarkably good motivation and evaluation base for real-time complex event detection systems, we further extend it by two aspects for this thesis. Section 1.1.2 presents both extensions and discusses why these extensions are reasonable. The purpose of both extensions is to make the scenario more realistic and, moreover, force our solution to be more generic. Thereby, we put special attention on the system's scalability, distributability and capability to handle distributed input streams and parallel client requests.
The resulting scenario is the motivating example scenario for this thesis. Moreover, the sensor and meta data provided for the ACM DEBS 2013 Grand Challenge are used to evaluate the correctness and performance of our approach.

1.1.1 The ACM DEBS 2013 Grand Challenge

In January 2013, Mutschler et. al. published the description of the ACM DEBS 2013 Grand Challenge [2, 3]. The grand challenge is a part of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS'13)[2]. This challenge is the third of its kind. The main purpose behind these challenges is to provide an evaluation base for CEP systems. In a nutshell, the task in the 2013 grand challenge is to generate several continuous data

[2] 7th ACM International Conference on Distributed Event-Based Systems (DEBS'13): http://www.orgs.ttu.edu/debs2013/ (07.08.2014)

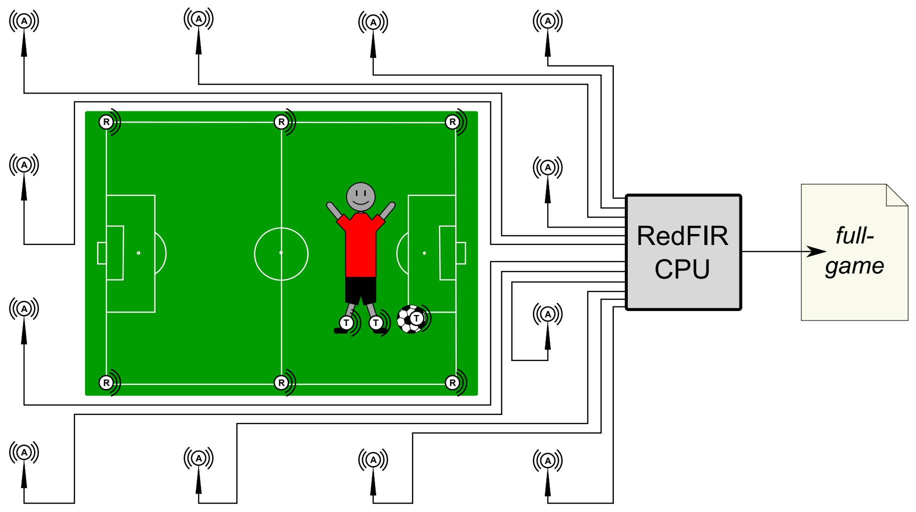


Figure 1.1: RedFIR© Tracking System

streams with statistical data for a soccer match for a given continuous sensor data stream in real time. In the remainder of this section, we will present the setting, the requirements and the specified queries of the ACM DEBS 2013 Grand Challenge.

1.1.1.1 Setting

The dataset of the ACM DEBS 2013 Grand Challenge consists of a test soccer match (60 minutes, 8 vs. 8) of the German Bundesliga club 1. FC Nürnberg in which the position, velocity and acceleration of each player and ball are captured with the highly accurate RedFIR© tracking system[3]. Figure 1.1 illustrates the main components and the setup of this system.
The fundamental idea of this system is comparable to GPS. More precisely, there is a RedFIR transmitter built into each ball and in each of the player's shin guards. The goal keepers are additionally equipped with transmitters near the hands. Each RedFIR transmitter emits a signal in a dedicated interval (ball 2000 Hz, others 200 Hz). Moreover, there are six reference transmitters with known positions, placed at the four corners of the soccer field and at the two crossing points of the halfway line and the sidelines. There are twelve receiving antennas placed around the field. Each antenna is plugged to an FPGA which receives and cleans the signal. All twelve FPGAs are synchronized. A CPU connected to all FPGAs can use the time differences between the arrivals of each emitted signal at the different antennas and the known positions of the reference transmitters to compute the positions of each RedFIR transmitter. The velocity and the acceleration are further computed by the CPU using the positions. Finally, the CPU generates a continuous and live data stream containing the measurements of all transmitters with the following schema:

$$sid,\ ts,\ x,\ y,\ z,\ |v|,\ |a|,\ v_x,\ v_y,\ v_z,\ a_x,\ a_y,\ a_z$$

(e.g., 47, 10634747088949386, 27488, -5849, 161, 57112, 1206195, 5291, -7075, -4683, 8356, -5332, 1316*)*

[3] RedFIR©: http://www.iis.fraunhofer.de/en/bf/ln/referenzprojekte/redfir.html (07.08.2014)

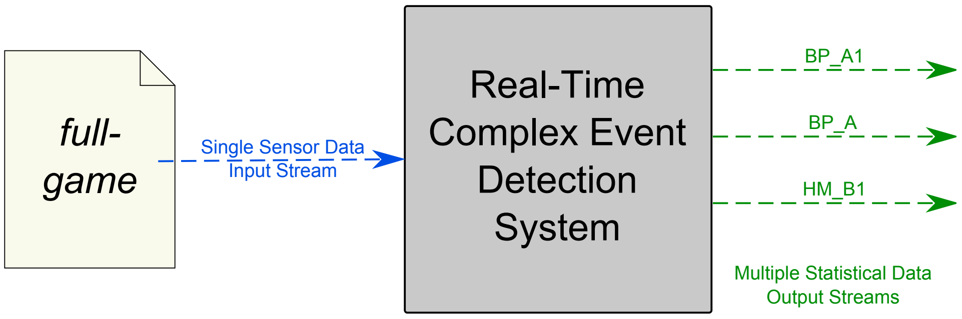

full-game
Single Sensor Data Input Stream
Real-Time Complex Event Detection System
BP_A1
BP_A
HM_B1
Multiple Statistical Data Output Streams

Figure 1.2: DEBS 2013 Grand Challenge Problem Overview Picture

In this 13-tuple *sid* denotes the sensor ID. A mapping from sensor IDs to the players and balls is provided separately. *ts* stores the timestamp at which the data were measured. The triplet $\langle x, y, z \rangle$ contains the position information. $|v|$ and $|a|$ are the absolute values of the velocity and the acceleration, respectively. The remaining values describe the directions of the velocity and the acceleration.
The ACM DEBS 2013 Grand Challenge provides a file (*full-game*) of approximately 4 GB containing the generated data stream, i.e., the measurements of all transmitters. In the challenge, this file is used to generate a continuous data stream as an evaluation base for a real-time complex event detection system. The task is to analyze the sensor data stream using different algorithms and generate statistical data streams which answer various queries w.r.t. interesting soccer events and statistics. Figure 1.2 illustrates an overview of the problem to be solved.

1.1.1.2 Requirements

An important requirement for the solution is that a query must be answered in real-time on-line during the soccer match. Thus, both the analysis of the input stream (sensor data) and the generation of the output stream (statistical data) have to be processed in real-time while new data arrives. Moreover, an additional requirement is, that multiple queries must be answered in parallel. Hence, the complex event detection system has to be able to analyze different events and generate multiple output streams in parallel. Apart from these requirements, there are no further specifications on how the problem should be solved.

1.1.1.3 Queries

The ACM DEBS 2013 Grand Challenge defines which queries have to be answered. Namely, the following queries are required of the complex event detection system:

1. **Running Analysis:** The system should analyze the player movements and generate individual running statistics for each player. The system should be able to generate two different kinds of running statistics:

   a) Current running statistics: This data stream should reflect the current running intensity of the player.

   b) Aggregate running statistics (different window lengths): These data streams should contain information about how long (time as well as distance) the player stayed in a certain running intensity in the past (1 minute, 5 minutes, 10 minutes, 20 minutes or the whole game).

2. **Ball Possession:** The system has to generate ball possession statistics for each individual player and aggregated statistics for the whole teams:

   a) Per player: This stream should include the time of ball possession (for the whole game) as well as the total number of ball hits of this player.

   b) Per team (different window lengths): Each stream has to contain the aggregated time of ball possession for all team members as well the percentage w.r.t. the ball possession of both teams for the duration of the last window length (1 minute, 5 minutes, 10 minutes, 20 minutes or the whole game).

3. **Heat Map:** The system has to calculate statistics about the player presence in different regions on the soccer field. Therefore the system should generate heat map streams for different grid sizes (104 cells, 400 cells, 1600 cells and 6400 cells). Further, these statistics have to be calculated for different time windows (1 minute, 5 minutes, 10 minutes, 20 minutes and the whole game).

4. **Shot on Goal:** The system should be able to detect shots on the goal. For the duration of the shot, i.e., until the ball is blocked or leaves the soccer field, a stream containing the ball position, velocity and acceleration as well as the ID of the player who shot has to be generated.

All streams for query group 1 and 2 have to be generated with a frequency of up to 50Hz. The heap map streams (query 3) should only be updated every second. In contrast to the first three queries which produce streams over the whole duration of the game, the shot on goal query (query 4) only generates a data stream (with the frequency of incoming ball sensor data updates) during a shot.

1.1.2 Scenario Extension

Due to the sensor data (*full-game*) and its detailed elaboration, the ACM DEBS 2013 Grand Challenge scenario described in Section 1.1.1 is a remarkably good example scenario and evaluation base for a real-time complex event detection system. Nevertheless, in this thesis we want to extend the scenario by two aspects, which are presented in the following subsections. Figure 1.3 illustrates the impact of these extensions to the problem overview picture.

1.1.2.1 Multiple Sensor Data Streams

First, in the grand challenge scenario there is only one input data stream containing the positions as well as velocities and accelerations of all objects (i.e., balls, legs and hands). The reason for this is the way the data were captured. In this aspect, we want to deviate

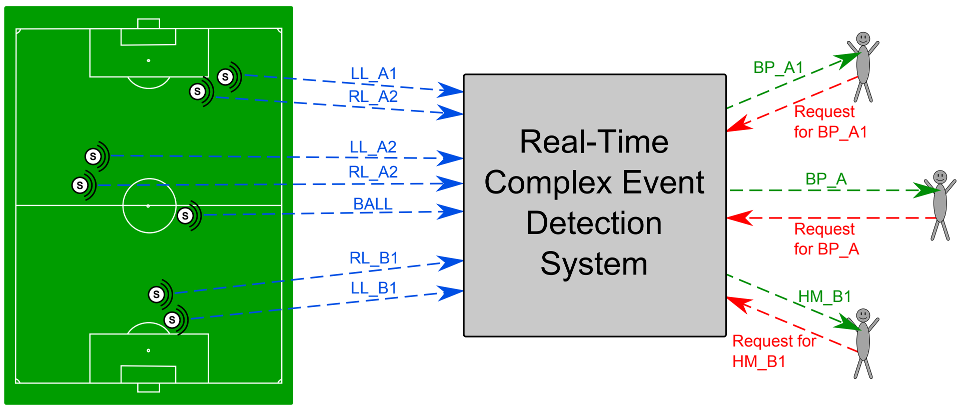


Figure 1.3: Extended Problem Overview Picture

the example scenario and thus the motivation for this thesis from the original ACM DEBS 2013 Grand Challenge scenario. In contrast to only having one input data stream including all measured data for all transmitters, we want to assume that each transmitter is a sensor which measures its position, velocity and acceleration on its own and further produces its own sensor data stream which acts as an input data stream for the real-time complex event detection system.

This modification changes the scenario into a more distributed and thus more generic but also more complex one. Instead of only one, the complex event detection system has to handle several distributed input data streams. In our opinion this modification is very important and reasonable since it forces the system to be more generic. We argue that a system which should be applicable for other scenarios than the above presented grand challenge, should be able to handle and analyze data from several distributed input data streams. Even in the soccer match analysis case (using RedFIR), it is very likely that there are more than one input data stream. For instance, in addition to the single RedFIR data stream, which contains position, velocity and acceleration data, each player could be equipped with a heart rate monitor which generates and sends an additional continuous data stream.

Hence, we simulate all transmitters (i.e., balls, legs and hands) as single sensors which produce their own data streams in order to obtain an example scenario with multiple input streams.

1.1.2.2 Client Requests

Second, there is another interesting topic which is not considered in the ACM DEBS 2013 Grand Challenge. The grand challenge only considers the analysis of the input data stream and the generation of statistical output data streams. The problem description states that it is sufficient to write the output data streams into files or the console output (*stdout*). Hence, the grand challenge totally disregards how clients (e.g., television broadcast companies or also normal persons) can request and access the data streams from the real-time complex event detection system. In this thesis we also face this problem. Thus, the scenario is further enriched with clients requesting, accessing and receiving the generated statistical output streams.

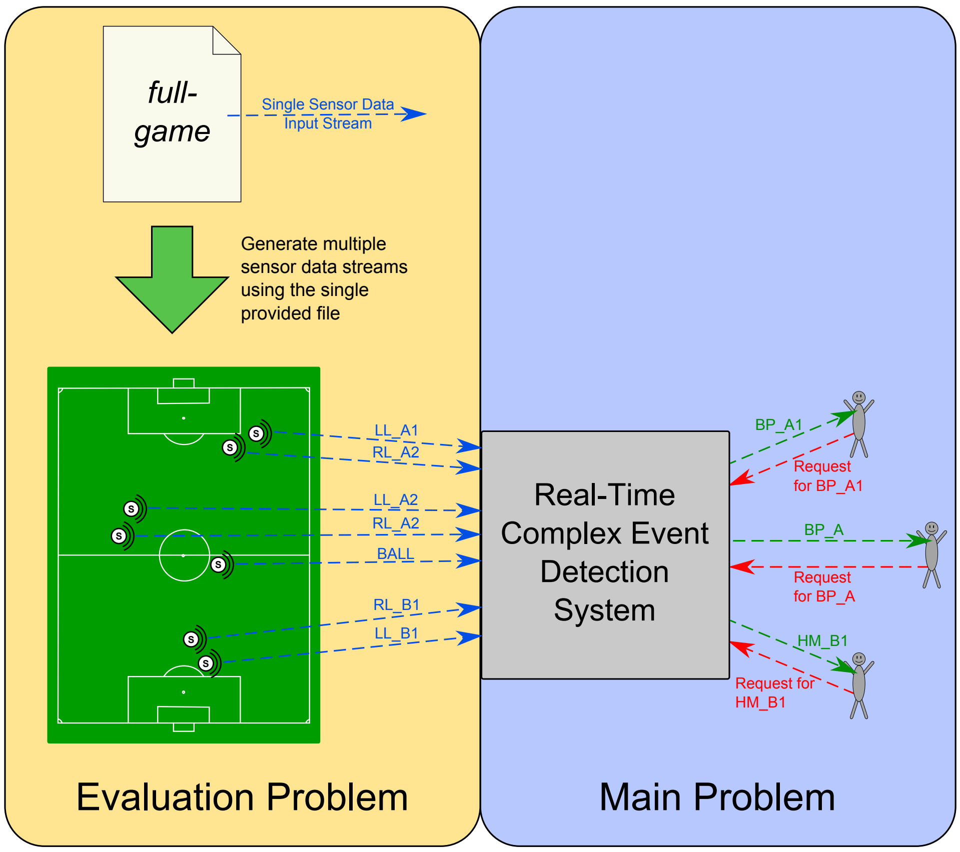

full-game
Single Sensor Data Input Stream
Generate multiple sensor data streams using the single provided file
LL_A1
RL_A2
LL_A2
RL_A2
BALL
RL_B1
LL_B1
Real-Time Complex Event Detection System
BP_A1
Request for BP_A1
BP_A
Request for BP_A
HM_B1
Request for HM_B1
Evaluation Problem
Main Problem

Figure 1.4: Problem Separation

1.2 Problem Statement

Figure 1.4 illustrates the full problem which has to be solved in order to analyze the extended motivation scenario presented in Section 1.1.2. The full problem includes both, generating multiple input data streams using the provided data as well as analyzing these streams in a real-time complex event detection system which further has to be able to handle multiple different client requests in parallel. Hence, it can be separated into two parts: the *evaluation problem* and the *main problem*.

The evaluation problem faces the issue, how to generate the input for evaluating the real-time complex event detection system using the provided data. Since we want our system to be able to detect events in multiple distributed input data streams, we have to generate those streams. More precisely, we want to simulate a live ongoing soccer match in which the players and the balls are equipped with independent sensors producing data streams which are analyzed by the complex event detection system. Thus, the evaluation problem considers how to generate multiple sensor data input streams from the single provided file (*full-game*) in real-time. Chapter 3 describes the evaluation problem in detail and presents our proposed solution as well as its implementation.

However, the main focus of this thesis is to develop a scalable real-time complex event detection system for distributed data streams and thus solving the main problem. It is important to note, that the purpose of this thesis is not to solve the extended version of the

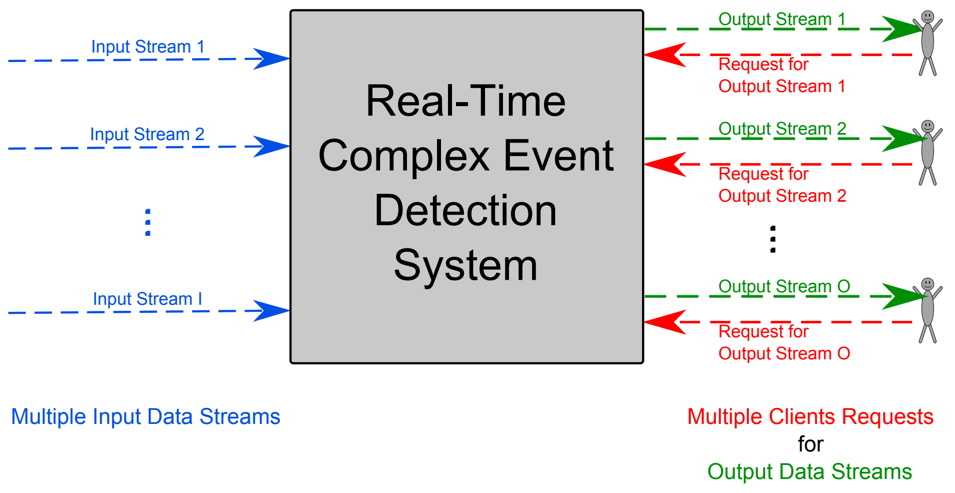

Input Stream 1
Input Stream 2
Input Stream I
Real-Time Complex Event Detection System
Output Stream 1
Request for Output Stream 1
Output Stream 2
Request for Output Stream 2
Output Stream O
Request for Output Stream O
Multiple Input Data Streams
Multiple Clients Requests for Output Data Streams

Figure 1.5: Generic Problem Overview Picture with I Input Data Streams and O Output Data Streams. To simplify the illustration each output data stream is only requested by and sent to one client. In practice, an output data stream can be requested by multiple clients and a client can request several streams.

ACM DEBS 2013 Grand Challenge presented in Section 1.1. Instead, the extended grand challenge is only the motivating example scenario and the evaluation base for this thesis. Rather its purpose is to develop a generic system which can analyze arbitrary events in arbitrary distributed data streams. Hence, the system should not be adapted or optimized for the grand challenge scenario and the scenario specific algorithms used for analyzing the input streams (e.g., ball possession calculation) should be easily exchangeable.
Figure 1.5 illustrates the problem this thesis tackles. The main focus of this thesis is placed on the development of the gray box in the middle, i.e., the real-time complex event detection system. More precisely, the main problem we want to solve is how to handle and analyze multiple distributed input data streams, how to generate several different output data streams (with statistical data) and how to answer client requests with these streams.
Moreover, we suppose that there are situations which cannot be handled by a single machine executing the whole real-time complex event detection system. Even in the grand challenge scenario, there are several situations conceivable in which a single machine is not capable of performing all work. For instance, assume that the number of sensors increases due to more players or new sensor types (e.g., heart rate monitors). Alternatively, assume that new statistics are introduced (e.g., cardioid or endurance statistics) or existing statistics are refined (e.g., finer heat map grids or more sophisticated shot on goal detection algorithms). Each of these small scenario modifications would increase the computational effort and it is very likely that eventually the effort exceeds the computational power of a single machine. But even if the scenario remains the same a single machine can be problematic if the number of client requests increase. For instance, assume that the FIFA wants to provide each television studio and all other interested parties data streams with real-time match statistics of the World Cup final match. This offer would result in a huge number of requests that could not be handled by a single machine. Both, its computational power and its network connection, would become a bottleneck. If one further considers that we want to develop

a generic system and not only a solution for the grand challenge, it is very clear, that our system has so be scalable. We argue that this can only be achieved by distributing the real-time complex event detection system onto multiple machines. Hence, obtaining good scalability and high distributability while avoiding bottlenecks and single point of failures whenever possible are our main targets. In Chapter 4 we present PAN, our P2P approach for scalable complex event detection in distributed data streams.

2 Background

This chapter takes a close look on the solutions of the ACM DEBS 2013 Grand Challenge [2, 3] (see Section 1.1.1), which were published in the Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS'13). While Section 2.1 briefly summarizes each proposed solution, Section 2.2 compares them w.r.t. different issues and tries to identify joint concepts. Finally, Section 2.3 discusses if and how the proposed solutions consider our scenario extensions (see Section 1.1.2).

2.1 Proposed Grand Challenge Solutions

Jacobsen et. al. [4] present not only one but three solutions to the grand challenge. The authors introduce a multi-stage monitoring pipeline consisting of three stages. The first stage named *data collection and dispatching* stage feeds the complex event processing (CEP) engine with the input data stream. The CEP engine performs the *continuous query processing*, i.e., the second stage. The third stage, *visualization and distribution*, faces the problem of providing the statistics to the clients. This is done either by a GUI-based monitoring panel or by a publish/subscribe-based dissemination network. As mentioned above, the paper presents not one but three solutions. More precisely, the authors present three CEP engines for performing the second stage. The authors discuss the applicability of four existing open-source off-the-shelf CEP engines. While they figured out that StreamIT and STREAM are not suitable for solving the ACM DEBS 2013 Grand Challenge, they present solutions based on Esper and Storm. Moreover, the authors develop BlueBay, an event processing engine specialized for analyzing soccer games which is exactly adapted to the demands of the Grand Challenge. The key concept which yields BlueBay's good performance is the *Stream Window*, which is a way of bucketizing in a circular buffer with a fixed length to perform operations in constant time and with constant memory consumption. However, BlueBay only supports limited parallelization which introduces a trade-off between throughput and per-event delay and enables the deployer to distinguish between low-delay online analysis and throughput-optimized offline analysis.

Wu et. al. present SPRINT [5]. SPRINT is a stream processing engine which analyses a single data stream and generates multiple parallel output data streams with statistical data,

i.e., answer multiple continuous queries in parallel. Although SPRINT is explicitly designed for solving the ACM DEBS 2013 Grand Challenge, its architecture is also applicable to other scenarios. Actually, generalizing SPRINT is one of the authors planned future work. SPRINT's architecture mainly consists of three components. First, the data source is read by the *preprocesser* component. The preprocessor further injects the data tuples into an shared lock-free *ring buffer* (LMAX Disruptor library [10]), which acts as a bridge between the incoming data and the query processing. Hence, SPRINT follows the one-producer-multiple-consumer model. The third component is a *group of parallel query processors*, which read the data from the ring buffer. SPRINT supports inter- as well as intra-query parallelism. Inter-query parallelism means, that each of the four queries is performed independently and concurrently with the other queries. In addition, a query can be further parallelized by a partition-and-merge paradigm. This additional parallelism is called intra-query parallelism. Calculating the statistics in the different queries highly benefits from the *frame-based sliding window* concept, which is a less memory consuming alternative to the common sliding-window approach that splits the window into equal-sized intervals. Another noteworthy point is that the authors exploit the fact that coarser heat maps can be generated by aggregating the heat map with the finest granularity (6400 cells). A drawback of the SPRINT approach is, that it uses load shedding to handle different query processing rates and avoid blocking incoming data due to a full ring buffer. Hence, SPRINT's high throughput is achieved by dropping some either fixed or dynamically tuned percentage of the incoming sensor data tuples.
Jergler et. al. [6] present an approach with a special workflow-based architecture. First, the *Event Replaying* module reads the sensor data from the provided file and feeds them into the *Distributer Ring-Buffer* respecting the timestamps (i.e., simulates the input stream). Then the *Parallel Processing* module fetches data from the distributer ring buffer, analyzes the input stream according to the queries and puts the statistics into the *Output Ring-Buffer* from where the statistical output streams can be forwarded to clients. The way the query processing (i.e., the analysis) is done, is the special part of this work: The authors present a workflow-like way of processing the queries. They propose connecting different computation and aggregation tasks with non-blocking ring buffers (LMAX Disruptor library [10]). This yields a sequential and parallel arrangement and connection of different tasks. The paper also includes how the grand challenge queries can be solved using this architecture, i.e., which tasks one has to combine and how. Although the authors only implement a centralized solution, they state that their workflow idea can also be implemented in a distributed way by leveraging the publish/subscribe concept. Moreover, the paper briefly presents the *System Statistics Monitoring* module which in a nutshell is a HTML5 based visualization of the produced statistic streams. Finally, the authors discuss how the event processing (i.e., analysis and statistic computation) can be accelerated by using FPGAs or GPUs.
The paper by Madsen et. al. [7] proposes a MapReduce-style solution called Enorm. Enorm is an extended version of the MapReduce concept which is optimized for processing high frequency input data streams. In contrast to traditional MapReduce, Enorm provides native support for window computations and "sharing common computations among overlapped windows" [7]. While common MapReduce involves only map and reduce functions, Enorm

consists of *map*, *compute* and *consolidate* functions. While compute roughly replaces the common reduce function, consolidate has to be used when sharing common computations is desired. Moreover the authors present a detailed cost model, which should facilitate the decision if sharing is useful. The authors state that "the feature of sharing computation in Enorm is suitable for queries that have multiple windows, high input rates and relatively low output frequency" [7]. A cost analysis unfortunately indicates that sharing is not beneficial for the ACM DEBS 2013 Grand Challenge queries due to their high output frequencies. The authors describe in detail a single Enorm job which answers all grand challenge queries. Moreover, they deploy the Enorm job in the Amazon Elastic Compute Cloud (EC2) to show its scalability.

Gal et. al. present TechniBall [8]. TechniBall is based on a generic solution developed for the INSIGHT European project, whose purpose is to improve the ability to handle emergency situations. The architecture combines the streams framework with the Esper engine. Streams enables the authors to define data flows containing different processors in XML-files. Moreover, the authors are able to implement processors using the streams API. Furthermore, the authors implement "a single custom Esper processor using the streams API, directly including Esper queries into the XML description of the data flow graphs" [8]. Hence, the different processors in the data flow can be implemented either with the streams API, for fast processing simple computations, or with Esper queries for more complex analyses. TechniBall uses this architecture to answer the ACM DEBS 2013 Grand Challenge queries. More precisely, the authors defined a data flow and several processors which reads the input data from one source and prints all statistical output streams to the console (*std-out*). TechniBall uses a state machine for the shot on goal query and sliding windows for aggregated statistics. Furthermore, the authors implement a GameView processor which visualizes the input data (not the statistics) and enrich the console output by query fields and JSON encoding. The authors state that their work "follows a stream-oriented model for event processing" [8] and that the data flow execution can be distributed to several machines when using a Storm topology instead of simply executing it in the streams-runtime. However, the authors neither evaluate this nor present more details how much effort this would induce.

Badiozamany et. al. propose a generic system called EPIC [9]. EPIC is a data stream management system (DSMS) extending the SCSQ system. EPIC's linchpin is its high level query language. All computations, conversions and analyses at the *query processing nodes* as well as the dataflow are expressed by the query language. The two major extensions EPIC introduces are the *Frequently Emitting Windowizer* (FEW) and "user-defined incremental window aggregate functions" [9]. Sliding windows are very important for analyzing the streams and so for answering the grand challenge queries. The query language mentioned above supports windows as first class objects. The authors state, that the problem of normal windows is, that they cannot "emit aggregation results before a complete windows is formed" [9] (e.g., the 5 minute windows cannot emit results after playing only 2 minutes). To solve this problem, the authors introduce the FEW operator, which "decouples the frequency of emitted tuples from a window's slide" [9]. Moreover, EPIC enables a user to define its own windows aggregation functions by implementing three functions (*init*, *add* and *remove*) and registering the aggregation function in the system. In addition to presenting EPIC's generic extensions to the SCSQ system, the authors explain in detail how the ACM DEBS 2013 Grand Challenge queries can be answered with an EPIC query. Their proposed solution heavily leverages the frequently emitting sliding windows and user-defined aggregation functions. Furthermore, the solution exploits the fact that coarser heat maps can be constructed by fine ones. In addition, the authors state that EPIC enables the user to define "user defined parallelization primitives" [9]. The authors even state that "the system can also distribute query processing nodes over several computers" [9]. However, they did neither describe how this can be done nor evaluate a distributed execution. EPIC's intense but easy customizability is its main advantage. The (frequently emitting) windowizer as well as the user-defined incremental window aggregate functions enable using EPIC for various scenarios. The special feature of this solution is that the whole dataflow as well as all computations and functions are defined in a query language (only functions can also be written in external programming languages). This, however, has a serious effect on the throughput as well as delay performance which is much weaker than those of any other proposed solution.

2.2 Comparison

This section discusses commonalities of and differences between the different solutions ([4], [5], [6], [7], [8] and [9]) presented in Section 2.1. If one compares the different solutions, one can make the following observations:

- **Statistic Calculation:** None of the proposed solutions generate all resulting statistical data streams directly from the input sensor data stream. Rather, the single input stream is transformed into several intermediate result streams by means of filtering, extending (with meta data information) or preprocessing. Often this is done in multiple consecutive steps. In addition, query results (i.e., statistical output streams) often base on other query results (e.g., the team ball possession stream depends on the player ball possession streams) or computations are shared between different time windows and heat map resolutions. For instance, [5] and [9] both explicitly state that they exploit the fact, that the more coarse heat maps (e.g., 12×25) can be computed by aggregating the finest heat maps (e.g., 64×100) and thus one only has to calculate and store the heat maps with the finest granularity. Moreover, all solutions have in common that they use some kind of sliding windows at least to answer the aggregated statistic queries. Anyway, sliding windows and aggregations are the major key concepts for the statistic calculations and therefore for answering the queries in all proposed solutions.

- **Generalization:** All presented solutions for the ACM DEBS 2013 Grand Challenge either base on a generic architecture presented in the published papers ([6], [7], [8], [9] and the Esper and Storm solutions in [4]) or can at least be generalized with relatively little effort ([5]). The sole exception is the BlueBay approach in [4] which is intentionally specialized for answering the grand challenge queries for the defined input data stream.
- **Main Architecture Idea:** The main architecture ideas behind the proposed solutions can be separated into three groups. The first group ([6], [8] and [9]) are workflow-based architectures. Although the concrete realization differs and the authors use different terms to describe their architecture, the main idea is roughly the same. [7]'s architecture follows an extended MapReduce concept. In contrast, the third group ([5] and BlueBay in [4]) have rather hardcoded and static architectures with much less flexibility (see also distribution).
- **Ring Buffer:** Another concept which is very prominent in the presented solutions is the ring buffer concept. The main advantage of ring buffers compared to other buffers is their constant length and thus constant memory consumption and absence of memory allocations during runtime. Three solutions (namely [5], [6] and BlueBay in [4]) explicitly state that they use ring buffers. For the other solutions there is no information given which buffers are used, but it is likely that they also use ring buffers for either input, intermediate or output data streams or for storing windows. [5] and [6] both use the LMAX Disruptor library [10] which contains a shared lock-free and thus non-blocking ring buffer.
- **Parallelization and Distribution:** All proposed solutions provide a kind of support for parallelism on a single machine. Hence, they leverage a state of the art multi-core system. Furthermore, all papers evaluated their systems with activated parallelization. Although the performance results are different, the common result is that parallelization introduces synchronization costs which can harm the performance when not considered properly. But, in contrast to single machine parallelism, not all proposed systems can be executed distributed on several machines. The authors of [6], [7], [8] and [9] state, that their solutions can be distributed without too much effort. One important reason for that is that these are the solutions with an workflow-based or MapReduce-style architecture. However, only the MapReduce-style solution [7] is evaluated also in a distributed manner in the Amazon Elastic Compute Cloud (EC2). For the other solutions, heretofore there is only a statement that it is possible to distribute the architecture on multiple machines but no implementation or evaluation is presented in the papers published in the Proceedings of the DEBS'13.

2.3 Scenario Extension Consideration

In this section we inspect if and how the proposed solutions considered our scenario extensions (see Section 1.1.2) and what this means for the extensions. For the two extensions the following conclusions can be drawn:

- **Multiple Sensor Data Streams:** As we expected no presented approach faced the problem of analyzing several distributed input sensor data streams. This is not surprising since it was unambiguously stated that there is only one input data stream containing all sensor data. Hence, this extension noticeably modifies the ACM DEBS 2013 Grand Challenge. Therefore papers handling the extended scenario would not solve the original grand challenge and therefore perhaps would not have been accepted and published. Moreover, this extension complicates the problem a lot.

- **Client Requests:** In contrast to our first extension, the second extension (i.e., handling and answering client requests) is also faced by some solutions published in the Proceedings of the DEBS'13. We suppose the reason for this is that, in difference to our first extension, the second extension does not modify but only extends the grand challenge. [4] solves the problem of handling client requests in the best way. [4] is the only solution which provides both, a visualization (which could be accessed by clients or integrated into television broadcasts) and a publish/subscribe dissemination network (where clients can subscribe for statistical data streams). The solution presented in [6] contains a HTML5 client which visualizes statistics and can be accessed by clients in the web browser. [8] contains a visualization of the incoming sensor data streams (not the statistics) and enriches the output with JSON encoding. This can be interpreted as a very rudimentary solution of the client request problem. The other proposed solutions ([5], [7] and [9]) do not face the client request scenario extension and hence only print their output streams to the console (*stdout*) or to files as required in the ACM DEBS 2013 Grand Challenge problem description. Nevertheless, the fact that two (or even three) out of six proposed and published solutions also face our second scenario extension, emphasizes that this extension is reasonable.

3 Evaluation Problem - Sensor Simulation Environment

This chapter faces the evaluation problem. First, Section 3.1 describes the evaluation problem in detail. That is, it points out the different challenges which have to be solved in order to provide an evaluation base for our real-time complex event detection system. Moreover, in Section 3.2 we present our conceptual solution approaches for the different subproblems as well as the theory which underlies our concepts. Section 3.3 presents our implementation which produces the input data streams for evaluating PAN.

3.1 Problem

As mentioned in Section 1.1.2.1, we modified the ACM DEBS 2013 Grand Challenge scenario. Instead of only one continuous input data stream containing the sensor data of all objects, we want the system to be able to handle and analyze multiple distributed continuous input data streams in parallel. Hence, we cannot simply read the provided sensor data file (*full-game*) and feed the input buffer of our complex event detection system with data tuples from this file w.r.t. the corresponding timestamps, as many solutions published at the DEBS'13 did (see Chapter 2). Instead, we have to simulate each RedFIR transmitter as an independent sensor which measures its position, velocity and acceleration on its own and further generates its own continuous sensor data stream and sends this to the complex event detection system.

Thus, the first problem to be solved is how to simulate real-world sensors. This problem does not have to be solved for developing a scalable real-time complex event detection system for distributed data streams, but for using the data provided by the ACM DEBS 2013 Grand Challenge for evaluating this system. Therefore, we call this problem the *evaluation problem*. The remainder of this section discusses this problem. More precisely, in the following, we will present which problems have to be faced, which questions have to be answered and which tasks have to be solved to simulate the real-world sensors in a way that they generate and send data streams as if they were measuring data of a current soccer match.

3.1.1 Additional Requirements

We require our simulation environment to support both, the simulation of all sensors on a single machine as well as the distribution of the sensor simulations onto multiple devices. The reason for this is that it should be possible to perform two kinds of evaluations.

First, measuring the query delay during the evaluation necessitates executing all sensor simulators on the same machine (see Section 5.2.2). Therefore, all sensor simulators have to run in parallel processes on the same machine. The possibility to execute all sensor simulators on a single machine further enables us to start a new evaluation of our system with minimal effort and costs since the most simple evaluation setup is to execute the sensor simulations as well as the complex event detection system and potential clients on a single machine. For instance, this facilitates quickly testing small changes in our scalable complex event detection system.

Second, it should be possible to evaluate the real-time complex event detection system under real-life conditions. Since in the real-life use case each sensor is a single independent device, we require that also the sensor simulation can be distributed in the same way. According to that, it should be possible to execute each sensor simulator on a single device (e.g., a very weak cloud computing instance).

Moreover, we also want our simulation environment to support all setups between these two extreme cases. This means, that we want to be able to perform the simulation of M sensors distributed on N machines ($N \leq M$).

3.1.2 Sensor Data Separation

The ACM DEBS 2013 Grand Challenge provides only a single file (*full-game*) containing the data of all transmitters. In order to simulate a transmitter as a single independent sensor producing its own data stream, one has to separate the data tuples corresponding to the certain transmitter from this file. These separated data tuples can then be used by the sensor simulator to generate a continuous data stream.

3.1.3 Real-World and Real-Time Simulation

The problem of simulating a sensor which acts like a real-world sensor measuring data during a live soccer match, can be splitted into two subproblems: Simulating the sensor's system properties (*real-world*) and simulating the correct timing, i.e., generating and sending the streams with the same timing as if the soccer game would take place at the moment (*real-time*).

In contrast to the separation which only has to be performed once, the simulation has to be performed again for each evaluation. This is due to the fact, that the simulation is done for generating real-time sensor data streams, i.e., for simulating a live ongoing soccer match which has to be analyzed in real-time by the complex event detection system.

The question of how the data streams are actually sent, or more precisely who the actual receiver of the sensor data stream is (e.g., which component or peer of the scalable complex event detection system), is not faced in this part, since this question also arises in the real-life use case and not only during simulation.

3.1.3.1 Real-World

A real-world sensor which, for instance, measures the position of a ball is a highly mobile wireless and performance weak device. Such a sensor strongly differs from a state of the art personal computer or cloud instance in terms of computational power and network connectivity. However, to be able to simulate the sensors on such machines, the properties of a sensor have to be simulated in the sensor simulator running on an arbitrary machine in order to obtain real-world conditions for our evaluation.
Moreover, it should be possible to simulate different kinds of sensors (e.g., GPS sensors and heart rate monitors) with different properties. Although this is not the case in the ACM DEBS 2013 Grand Challenge scenario, we want our system to be able to handle different sensors. In order to be able to evaluate that, also the sensor simulator has to be able to simulate this heterogeneity.

3.1.3.2 Real-Time

Another problem that we face in this thesis is how to achieve correct timing for the simulation. For instance, it would be problematic if the sensor simulator of the ball sends the data from the 57th minute of play while the sensor simulator for player A1's left shin guard sends data from the 55th minute. Although this example is very extreme, it is very important that the sensor simulators generate data streams with data from the same point of time in the match. Otherwise, the input sensor data streams for the complex event detection system would contain data from different situations in the game. If the time gap between the data is to large, it is impossible for the complex event detection system to perform any proper analyses. Furthermore, the system does not have to be able to handle such problems since this cannot happen when analyzing a live ongoing soccer match in real-time, which in fact is the purpose of the system. Since we do not want to generate new problems with the simulation which cannot happen in the real-life use case, the sensor simulators have to prevent too large time differences between the distributed data streams.
While large time differences do not occur in the real-life use case, small time differences do. For instance, these small differences could be introduced by real-world sensors with different latencies. Accordingly, the event detection system has to be able to handle small time differences. Thus, small time differences between the streams produced by the sensor simulators are acceptable. Finding the threshold which time differences are acceptable for the system, i.e., which maximal time difference has to be guaranteed from the sensor simulators, is one important question to be answered in this thesis.
Hence, guaranteeing that the time differences do not exceed a certain threshold is an important task. This task has to be solved by synchronizing the time of all sensor simulators. If all sensors are simulated on the same machine, i.e., the sensor simulators running in different processes on the same machine, this problem is relatively easy to solve. However, as defined in Section 3.1.1, the sensor simulation should be executable in a distributed manner on multiple machines. In this case, synchronization (i.e., obtaining a global timestamp) is a non-trivial task.
Moreover, even if all sensor simulators are perfectly synchronized, i.e., have exactly the same

global timestamp, the sensor data tuples have to be read from the data and added to the data stream in the moment they were captured during the soccer game. Although the data tuples are enriched with global timestamps (see Section 1.1.1.1), it is not possible to add them at the precise moment unless the sensor simulator checks the data tuple list every picosecond, which is for sure not a good idea. Instead, we will have to define a check period which has to be respected in the above mentioned time difference threshold.

3.2 Theory

In this section, we present our solution approach for the evaluation problem presented in the previous section as well as the theory which underlies our concepts. That is, we present our concept how to simulate real-world sensors. More precisely, we present how we use the data provided by the ACM DEBS 2013 Grand Challenge to generate an evaluation base for PAN, i.e., our scalable real-time complex event detection system for distributed data streams.
In a nutshell, we use the provided file containing all data tuples, to simulate each RedFIR transmitter as an independent sensor which generates and sends its own sensor data stream. As a result, we obtain multiple distributed input sensor data streams for our complex event detection system.
As presented in Section 3.1, we have to perform two steps to achieve this goal. The first step is to separate the data tuples w.r.t. their origins (i.e., the corresponding sensors) into subsets. Section 3.2.1 presents our proposed solution for this step. The second step is the actual simulation of the real-world sensors. Our approach for solving this step is proposed in Section 3.2.2.

3.2.1 Sensor Data Separation

In the ACM DEBS 2013 Grand Challenge there is only one single continuous input data stream containing the data tuples of all sensors. As a consequence, a solution for this challenge, or more precisely a submitted real-time complex event detection system (see Chapter 2) only has to be able to handle and analyze one input data stream. For generating this input data stream, the DEBS'13 committee provides a single file (*full-game*) containing the data from all RedFIR transmitters. Each line in this file reflects a single measurement. Each measurement is stored as a 13-tuple with the following schema (see Section 1.1.1.1 for more details):

$$sid,\ ts,\ x,\ y,\ z,\ |v|,\ |a|,\ v_x,\ v_y,\ v_z,\ a_x,\ a_y,\ a_z$$

(*e.g.,* 47, 10634747088949386, 27488, -5849, 161, 57112, 1206195, 5291, -7075, -4683, 8356, -5332, 1316)

As mentioned in Section 1.2, the purpose of this thesis is not to develop another solution adapted to the ACM DEBS 2013 Grand Challenge. Instead, we want to develop a generic scalable real-time complex event detection system which amongst other things should also support multiple distributed input data streams. In order to still be able to use the grand challenge as an evaluation base, we extended the scenario in a way that it produces multiple distributed input data streams. More precisely, we simulate that each transmitter is an independent sensor which produces its own sensor data stream (see Section 1.1.2.1).

To be able to simulate a sensor, a preparation step is required. We need to separate the data tuples stored in the single provided file (*full-game*) with respect to their origin (i.e., the corresponding transmitter). Since the 13-tuples contain a sensor ID (*sid*, e.g., 47), we can use this ID to separate the file. More precisely, we split the single file into several data tuple subsets named after their sensor ID. Each entry in the new subsets reflects a measurement of the sensor corresponding to transmitter *sid*. For instance, subset 47 contains all data tuples representing measurements of the sensor with *sid* 47 (Left Leg Player A2).
In addition, it would be possible to drop the *sid*, i.e., store and stream 12-tuples instead of 13-tuples to save network bandwidth. Nevertheless, we abandoned this idea, since the sensor ID information contained in each tuple is the easiest way to identify the sensor corresponding to the measurement (i.e., to the position, etc.). Removing it would necessitate storing the source corresponding to each sensor data stream in another way. Moreover, the *sid* is only a small part of a data tuple and thus removing it would not reduce the bandwidth consumption remarkably.

3.2.2 Real-World and Real-Time Simulation

In this section, we present the ideas on how to simulate the sensors with the separated data tuple subsets as well as the basic theory that underlies our concepts. More precisely, we present how we generate multiple distributed continuous sensor data input streams for evaluating our complex event detection approach using the sensor data tuple subsets produced by the sensor data separation. Thereby, we take special care of fitting the correct timing (*real-time*) and present ideas on how to simulate the sensor system properties (*real-world*). Moreover, as presented in Section 3.1.1, we require our sensor simulation environment to run on a single machine as well as distributed on multiple devices (e.g., cloud computing instances). To achieve that multiple or even all sensor simulators can be executed on the same machine, we identify a sensor simulator in the network communication not only by its IP address but additionally by its port. As a result, multiple sensor simulators can be uniquely identified while running on the same machine. This enables both, a network communication between sensor simulators running on different machines as well as between sensor simulators running on the same machine.

3.2.2.1 Real-World

In the scope of this thesis, we perform the evaluation of our approach on state of the art computers and cloud computing instances (e.g., Microsoft Azure cloud instances). These machines are typically well-connected (i.e., low latency, high bandwidth, etc.) and relatively powerful. In contrast, a real-world sensor, which for instance measures the position of a ball or the heart rate of a player, is a highly mobile wireless and performance weak device. To perform realistic evaluations (i.e., evaluate our system under real-life conditions) the sensor simulator has to simulate the system properties of a sensor:

(1) Fluctuating high latencies: A characteristic of wireless and especially mobile devices are their high latencies. Moreover, the latency of a wireless mobile device typically fluctuates a lot.

(2) (Single) Message losses: Due to the sensors high mobility and its wireless connection, it is very likely that some messages are lost during transmission.

(3) Temporal unreachability: It is possible that a wireless mobile devices is temporally unreachable for instance due to signal blocking obstacles.

(4) Message Reordering: Particularly in network connections with a wireless mobile participant, message reordering is very likely occur.

Moreover, the simulation environment has to be able to simulate different kinds of sensors (e.g., GPS sensors and heart rate monitors) with different system properties at the same time in parallel without compiling multiple versions of the sensor simulator. For instance one sensor could have a low latency but be vulnerable for message losses, while another sensor guarantees to be loss free but has a high and fluctuating latency.
However, it is out of the scope of this thesis to develop and implement a sensor simulation environment which is capable to reflect realistic sensor system properties. Instead, we only want to produce input data streams as an evaluation base for our real-time complex event detection system. Since we further argue that it is legal to evaluate PAN first under perfect conditions (in order to avoid corrupting the evaluation results and thus loose trends etc.), we postpone simulating the sensor system properties and evaluating PAN under more realistic conditions w.r.t. the input data streams to future work.

3.2.2.2 Real-Time

Section 3.1.3.1 revealed that the simulation environment has to guarantee that the time difference between two data input streams (Δt_{ij}, where i and j are the sensor IDs of the data streams and $i \neq j$) does not exceed a certain threshold (T). This condition can be mathematically expressed with Equation 3.1.

$$\Delta t_{ij} \leq T \qquad \forall i,j \mid i \neq j \tag{3.1}$$

If this condition does not hold, the sensor simulation environment and thus the evaluation base would introduce problems which cannot occur in live ongoing soccer matches. Since wireless mobile real-world sensors would anyway introduce small time differences due to their different latencies, time differences smaller than the specified threshold are acceptable. T introduces a trade-off between effort and correctness. On the one hand, the threshold has to be realizable without to much (communication) overhead. On the other hand, the threshold has to guarantee that all incoming data streams contain data tuples from the same game situation since otherwise an analysis of the incoming sensor data streams is impossible. Hence, finding the threshold T which perfectly matches our demands is a non-trivial problem. We further suggest, that the perfect threshold depends on the exact scenario, e.g., on the latency variations of the sensors and on the velocity of the events. Therefore it is not possible to find a single specific value for T, which is the best for all scenarios. Instead, one has to experiment with different values in order to find an appropriate value for a specific evaluation setting.

In the remainder of this section, we want to face the problem how to guarantee that the time differences between incoming sensor data streams do not exceed T, or, mathematically, how to guarantee that the condition in Equation 3.1 is fulfilled.
In a nutshell, the key concept to solve this problem is synchronization. The sensor simulators have to synchronize their time in order to generate data streams and thus send data tuples from the same point of time in the match. For synchronizing their time, the sensor simulators have to consider two aspects: The *starting time* and the *time speed*.
The necessity to synchronize the time when the sensor simulators start (*starting time*) to read their sensor files and send data to the complex event detection system is easy to see. The need for considering the *time speed* is a more intricate problem. If all sensor simulators are executed on the same machine, this poses indeed no problem. In this case, all sensor simulators have access to the same clock (on the same machine) and therefore the speed in which the timestamps increase (i.e., the time speed) is the same for all sensor simulators. Thus, the time speed is already synchronized. However, if the sensor simulators are distributed this is not the case. Unfortunately, crystal clocks are not perfect. That means, that the speed in which the timestamp increases and thus the time speed is not the same on all machines. This results in increasing time differences (cf. literature [11]). Therefore, we also need to take the time speed into account in order to obtain the same time on all sensor simulators.
Lamport Clock [11] and Vector Clock [12] are well-known concepts for time synchronization. Lamport introduces the famous happended-before-relation ($\rightarrow$) as well as the logical clock concept. Moreover, Lamport proposes a distributed algorithm which produces a synchronized logical clock fulfilling the (weak) clock condition. Lamport further enriches the partial ordering between the events (happend-before-relation) to a total ordering ($\Rightarrow$) by introducing an arbitrary total ordering between the processes. This protocol is today known as Lamport Clock. The Vector Clock concept extends the Lamport Clock concept in a way that is also fulfills the strong clock condition. That means, that concurrent events are assigned to the same time, which is not true for Lamport Clocks.
Both concepts have in common that they are logical clocks. That means, they yield a causal ordering of the events. However, our sensor simulation environment needs a synchronized physical clock. More precisely, we do not need an ordering but a globally synchronized timestamp. Although, Lamport also presents a method of how to synchronize physical clocks with Lamport Clocks and "derive an upper bound on how far out of synchrony they can drift" [11], we decided not to use one of these concepts for our sensor simulation environment. The reason for this is, that the logical clock approaches assume that the events happened on different machines in the distributed system causally depend on each other. In our simulation environment this is not the case. Instead, the events (i.e., sensor measurements) are independent from each other. The only reason for synchronization is that we want to send all those data tuples at (approximately) the same time, which were also measured at the same time during the soccer match.
For synchronizing timestamps, Google uses the TrueTime [13] approach. TrueTime is the time API of Spanner, Google's latest globally-distributed database. TrueTime uses the known clock uncertainties (κ) to provide each machine with a timestamp which is guar-

anteed to not deviate more than ϵ from the true global timestamp. More precisely, each machine, on which TrueTime is running, holds a time interval $[earliest, latest]$. ϵ, which is the half of the interval's width, is the worst-case local clock time deviation since the last synchronization. Therewith TrueTime perfectly fits our demands. The timestamps simply have to be synchronized so often that ϵ cannot exceeds T.
Unfortunately, TrueTime needs an expensive infrastructure to work. TrueTime uses GPS and atomic clocks as time references. Each *time master* machine is connected to such a time reference device. The reference times are periodically compared between those machines. Each machine in a Google datacenter runs a *timeslave deamon* which periodically contacts a time master to update its local timestamp. The error bound ϵ (and thus the interval width) is 0 immediately after this synchronization and monotonically increases till the next synchronization (i.e., $\epsilon = \kappa \cdot t$).
Since, we do not want to setup such an infrastructure, we introduce *WeakTrueTime* (WTT). WeakTrueTime exploit the fact that we do not need a correct time as a time reference. All we need is an arbitrary reference time. Therefore, we simply choose an arbitrary sensor simulator to be the time master. The time master simply uses its machines clock as the reference time. All other sensor simulators run a timeslave deamon which periodically contacts the time master and updates their time interval in the same way as proposed by Google's TrueTime approach. Moreover, the time master selection does not have to be performed by a complicated leader election algorithm. Instead, an arbitrary sensor simulator can be informed to be the time master with a parameter when started. A more detailed description of WeakTrueTime is presented in Section 3.3.2.
With WeakTrueTime we can solve all synchronization issues presented above. The maximal time difference between a timeslave deamon and a time master ($\Delta t_{masterslave}$) is composed of the starting difference ($\Delta t_{start_{masterslave}}$) and the maximal time interval error bound (ϵ_{max}) (cf. Equation 3.2). The starting difference can be guaranteed to be approximately 0 be specifying a starting timestamp which is guaranteed to be so far in the future (e.g., $60s$) that all sensor simulators synchronized at least once with the time master before the simulation begins.

$$\Delta t_{masterslave} = \Delta t_{start_{masterslave}} + \epsilon_{max} = \Delta t_{start_{masterslave}} + \kappa \cdot t_{syncperiod} \tag{3.2}$$

The time difference between two input data streams (Δt_{ij}) is maximal if the streams are produced by two sensor simulators with timeslave deamons whereof one's WTT timestamp is $\Delta t_{masterslave}$ in the past and the other's WTT timestamp is $\Delta t_{masterslave}$ in the future w.r.t. the time reference (i.e., the machine timestamp of the time master). That is, the maximal time difference between two input data streams (i.e., $\max \Delta t_{ij}$) is composes of two times $\Delta t_{masterslave}$ and the both time periods in which the data tuple set is check for new events ($t_{checkperiod_i}$ and $t_{checkperiod_j}$) (cf. Equation 3.3).

$$\max \Delta t_{ij} = 2 \cdot \Delta t_{masterslave} + t_{checkperiod_i} + t_{checkperiod_j} \tag{3.3}$$

The period in which the tuple set is checked depends on the I/O performance of the machine and on the actual implementation. For instance, $t_{checkperiod_i}$ can be reduced when the

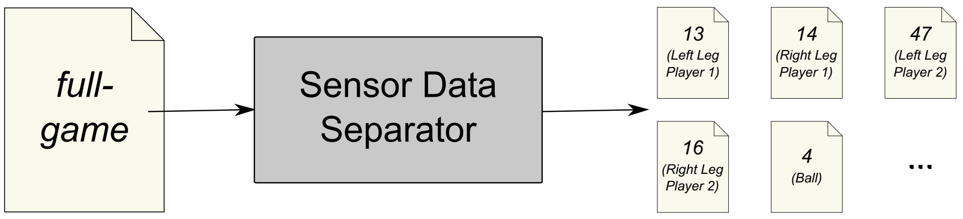


Figure 3.1: Sensor Data Separation Procedure

data tuple set is preloaded from the sensor file into a list in the main memory. Anyway, a checking period below $10ms$ should be possible. In order to guarantee the condition in Equation 3.1, the synchronization period ($t_{syncperiod}$), i.e., the time passing between two synchronizations with the time master, has to be chosen in a way that the maximal time difference between two input data streams does not exceed T (cf. Equation 3.4). Assumed that the periods in which the tuple sets are the same at both sensor simulators (i.e., $t_{checkperiod_i} = t_{checkperiod_j} = t_{checkperiod}$), $t_{syncperiod}$ has to be chosen in a way that Equation 3.5 is fulfilled in order to guarantee that the time differences between two input data stream does not exceed T.

$$\max \Delta t_{ij} \leq T \tag{3.4}$$

$$t_{syncperiod} \leq \frac{\frac{T}{2} - (\Delta t_{start_{masterslave}} + t_{checkperiod})}{\kappa} \tag{3.5}$$

3.3 Implementation

In this section, we present our sensor simulation environment implementation which we use as an evaluation base for PAN. Section 3.3.1 presents the *Sensor Data Separator* whose purpose is to prepare the provided data for the actual sensor simulation. In Section 3.3.2 we present *WeakTrueTime*, a standalone library for synchronizing machine timestamps in a distributed environment. This library is used by our *Sensor Simulator*, presented in Section 3.3.3, which generates and sends the input data streams for PAN.

3.3.1 Sensor Data Separator

As mentioned in Section 3.2.1, the total set of all data tuples has to be separated into several subsets. In a nutshell, we achieve this by separating the single provided file into several files named after the corresponding sensor IDs. Figure 3.1 illustrates this procedure.
The gray box in the middle (i.e., the *Sensor Data Separator*) is implemented very straight forward: The provided file (*full-game*) is read line by line. Each line is parsed and appended to the corresponding sensor file, i.e., to the file named by the *sid*.
Each of the generated files can be used by a sensor simulator to generate the corresponding sensor data input stream for the real-time complex event detection system.

The advantage of saving each data tuple subset in a file, is that the separation procedure only has to be performed once. Once separated, the files can be used for simulating the sensors in all upcoming evaluations. Due to that and since the first version already generated correct files in reasonable time, we relinquish further improving our implementation.

3.3.2 WeakTrueTime

In order to guarantee that all sensor simulators generate and send data streams containing data tuples from the same point of time in the match even if the simulators are distributed onto multiple machines (with different machine clocks), the sensor simulators require a synchronized physical clock. That is, each sensor simulator needs to have a timestamp, which is known to not differ more than a specified threshold from the timestamp of any other sensor simulator.
While Section 3.2.2.2 presented the theoretical advisements regarding this synchronization problem and discussed existing global clock approaches, this section presents how we implement *WeakTrueTime* (WTT).
WeakTrueTime is our approach for synchronizing a global timestamp between several processes on different machines. Since this service is not only applicable in the sensor simulation environment but also in other projects, we implemented it as a standalone Java library and imported it in the sensor simulator project.

3.3.2.1 Weaker Guarantees

Although, the WeakTrueTime approach is inspired by Google's TrueTime [13], the guarantees provided by WeakTrueTime are weaker than those provided by TrueTime. While TrueTime synchronizes the correct UNIX timestamp by leveraging GPS or even atomic clocks, WeakTrueTime simply assumes the time master's machine clock to be correct. Moreover, TrueTime returns a time interval which reflects the current clock uncertainty. Instead, WeakTrueTime only returns a timestamp which is guaranteed to not differ more than T from the time masters machine timestamp. For our purposes, i.e., for solving the evaluation problem, the guarantees provided by WeakTrueTime are sufficient. The sensor simulators do not need to know the correct timestamp but only the same timestamp in order to generate data streams with data from the same point of time. Moreover, the sensor simulators do not care about the current timestamp uncertainty as long as the difference does not exceed a specified threshold.

3.3.2.2 Assumptions

In addition to the weaker guarantees, WeakTrueTime has another limitation. The current implementation assumes that there are no latencies between the synchronizing machines, i.e., between the timeslave deamons and the time master. Hence, we suppose that the current WeakTrueTime implementation will not work properly for synchronizing the timestamp of processes running on wireless connected machines or on machines spread all over the world. However, again, this is not required for solving the evaluation problem since we simulate the sensors on well-connected[4] machines.

3.3.2.3 Architecture

In WeakTrueTime's architecture design we distinguish between the *time master* and the *timeslave deamon* component. Each Java process which takes part in a WeakTrueTime synchronization group, has to create a WeakTrueTime instance either as a timeslave deamon or as a time master. We design WeakTrueTime in a way, that there is only a single time master but arbitrary many timeslave deamons in each WeakTrueTime synchronization group. The machine timestamp of the process hosting this single time master instance is the reference timestamp for the global time synchronization.
The remainder of this section discusses our design considerations and presents the way WeakTrueTime works.

Design Considerations
There are two possibilities how to distribute the reference timestamp in the WeakTrueTime synchronization group. First, each timeslave deamon could periodically send a request to the time master. In this case the time master simply has to answer each incoming request with its current machine time. Second, the time master could be responsible for periodically disseminate its current machine time to all interested timeslave deamons (i.e., broadcast).
The advantage of the first approach is that the time master does not have to know and store the set of timeslave deamons, since it only has to answer requests. Nevertheless, we implement the broadcasting approach for two reasons. First, the broadcasting approach requires less packets and thus a project using WeakTrueTime has less network overhead for time synchronization. Second, if the time master broadcasts its reference time periodically, all timeslave deamons update an equal number of times. Assume the extreme case where there is a timeslave deamon in the synchronization group whose machine clock runs only half as fast as common (i.e., one machine clock second takes two "real" seconds) and all other clocks are perfect normal clocks. If all timeslave deamons would send every 10 seconds a request to the time master, the timeslave deamon with the broken machine clock would actually only sends a request every 20 seconds. Hence, this timeslave deamon would only receive the reference time half as often as the other timeslave deamons do. In difference, if the time master broadcasts the reference time in a specified interval, all timeslave deamons get the reference time equally often even if the time master's machine clock is broken.
The second design consideration we made is, whether to use TCP or UDP. We use UDP, since it is faster and we do not need the guarantees TCP gives. In fact, TCP's guarantees can even harm our system since resending a lost message with an outdated machine time harms our system more than missing a time update.

[4] More precisely, the latencies between two machines hosting sensor simulators in our simulation environments (i.e., in the office or the Microsoft Azure Cloud) are below $5ms$ and thus negligible since the tuple reading period ($t_{checkperiod} = 50ms$) introduces more time differences than disregarding the latencies in WeakTrueTime.

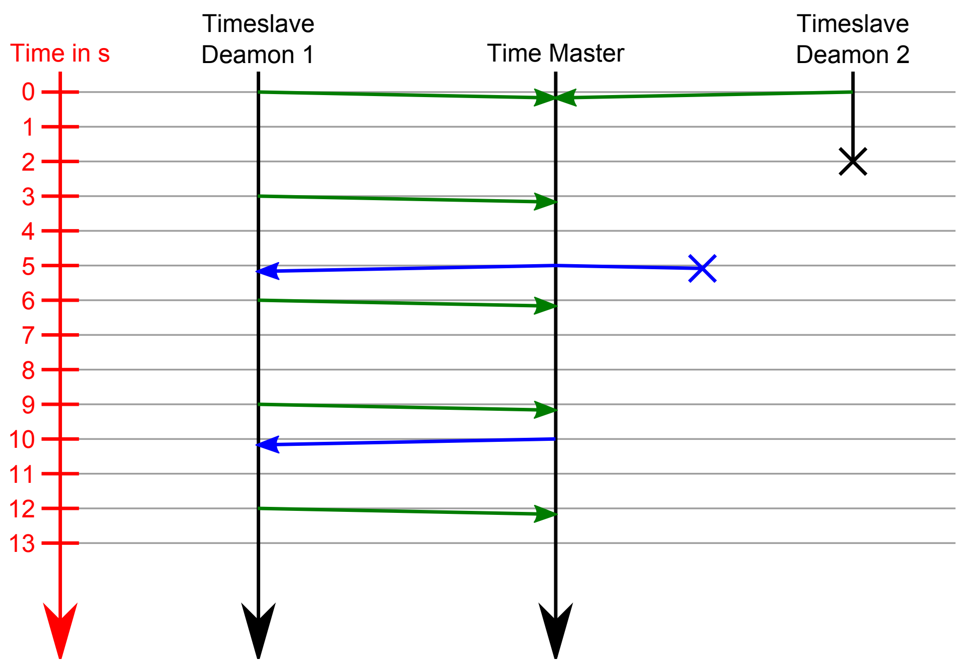


Figure 3.2: WeakTrueTime Communication Example. The green arrows denote `ALIVE` messages and the blue arrows denote messages containing the reference time. Parameters: $t_{aliveperiod} = 3000ms$, $t_{alivetimeout} = 8000ms$ and $t_{syncperiod} = 5000ms$

WeakTrueTime at a Glance

WeakTrueTime works in the following way. Each timeslave deamon periodically (every $t_{aliveperiod}$ milliseconds) sends a dedicated `ALIVE` message to the time master. The time master holds a list of all timeslave deamons (or more precisely their IP addresses and ports) from which it received previously an `ALIVE` message. If the time master did not receive an `ALIVE` message in the last $t_{alivetimeout}$ milliseconds, it removes the corresponding timeslave deamon from its list. All $t_{syncperiod}$ milliseconds the time master broadcasts its current system time to all timeslave deamons in its list. Every time a timeslave deamon receives a time message from the time master, it first checks if the contained timestamp has increased (i.e., is larger than the last received timestamp). This is necessary since we need a monotonically increasing timestamp and in UDP incoming packets can be reordered. If this condition is fulfilled, we use the contained reference timestamp for the WeakTrueTime calculation (see next section). Otherwise, the message is dropped. Figure 3.2 illustrates an example for the WeakTrueTime communication flow.

3.3.2.4 WeakTrueTime Calculation

In this section, we will answer the question how the WeakTrueTime (WTT) is calculated. Since the time master itself always has access to the time assumed to be the correct time reference (i.e., its machine time), it does not have to calculate anything, but simply returns its own machine time as the WeakTrueTime (cf. Equation 3.7).

In contrast, each timeslave deamon has to use the received timestamps from the time master to calculate its WeakTrueTime. From various possibilities, we decided to choose the following way of calculate the WeakTrueTime since we suppose it to be the one with the minimal error. Every time a timeslave deamon receives a new timestamp from the time master, it first checks if the received timestamp is larger than the last received timestamp. This is necessary since UDP does not prevent message reordering and we require WeakTrueTime to be a monotonically increasing clock. If this condition is fulfilled, we use the received timestamp ($t_{received}$) to calculate the difference (Δt) between the timeslave deamon's machine time ($t_{machine}$) and the received one (cf. Equation 3.6). This value can then be used to calculate the WeakTrueTime (cf. Equation 3.7).

$$\Delta t = t_{machine} - t_{received} \tag{3.6}$$

$$WTT = \begin{cases} t_{machine} & \text{at the time master} \\ t_{machine} - \Delta t & \text{at a timeslave deamon} \end{cases} \tag{3.7}$$

The alternative to this approach is to set the received time as the WeakTrueTime (i.e., $WTT = t_{received}$) whenever a time message arrives and in the meantime (until the next time message arrives) increase the WeakTrueTime in the same way as the local machine time ($t_{machine}$) increases. This could be done either by increasing the WeakTrueTime each time when the local machine time increases or by increasing it in arbitrary (maybe not even fixed) intervals by the value by which the local machine time has increased since the last WeakTrueTime update. The problem of the first option is that we do not have a hardware interrupt to properly increase the WeakTrueTime every machine clock second. We only have a software clock and thus not the possibility to guarantee an increment every machine second. Thus, this option must be discarded. The problem of the second option is that the deviation between the value stored as the WeakTrueTime (i.e., WTT) and the real reference time at the time master (i.e., its machine time) is much bigger than those of the delta approach we implemented. Assume the simplified case that the clocks have exactly the same clock speed and there is only a static time shift between the machine clocks (e.g., the machine clock of a timeslave deamon is 20 seconds in the past). Moreover, assume that the alternative approach updates the WeakTrueTime by leveraging the change of its local machine time approximately every 100 milliseconds. Thus, the maximum error when using the alternative approach is 100 ms just in the moment before the WeakTrueTime is updated. In contrast, if one uses the delta approach in this scenario the maximum error is 0.5 ms due to the 1 millisecond quantization of the timestamp. Hence, the delta approach for calculating the WeakTrueTime as presented above has a much better correctness than the alternatives.

3.3.3 Sensor Simulator

In this section, we present how we simulate all RedFIR transmitters as single independent sensors with the correct timing, i.e., how we generate and send the input sensor data streams with the same timing as if the soccer match would take place at the moment. In other words, we present how we solve the evaluation problem and thereby construct an evaluation base for PAN.

More precisely, we describe the *Sensor Simulator* project. The purpose of the Sensor Simulator project is to generate the sensor data input streams for evaluating PAN by simulating the sensors w.r.t. the real-time constraints presented in Section 3.2.2.2.
A single sensor simulator generates and sends the stream of a single simulated sensor using the corresponding sensor file (produced by the sensor data separator) and the WeakTrueTime library. In the remainder of this section, we will present how single sensor simulators can be composed to the whole sensor simulation environment. Moreover, we will present how a single sensor simulator works. Therefore, we first give a rough overview of the Sensor Simulator project architecture. Subsequently, we will present the sensor data stream generation loop as well as the time provider and the timed tuple reader component.

3.3.3.1 Sensor Simulation Environment

As mentioned in Section 3.1.1, we require the sensor simulation environment to support both, simulating all sensors on a single machine as well as distributed on multiple machines. Therefore, we do not implement a single simulation environment project which simulates all sensors. Instead, we implemented the Sensor Simulator project which only simulates a single sensor. In order to obtain the whole simulation environment, multiple of these sensor simulators (one for each sensor) have to be composed.
In practice, this means, that all sensor simulators are started sequentially with different sensor files by a Bash script. In order to ensure, that all sensor simulators simultaneously start to generate and send their data streams, the Bash script defines a starting timestamp (*startingTimestampInMs*).
If one uses WeakTrueTime as the time provider, one of the sensor simulators has to be specified to be the WeakTrueTime master and the starting timestamp has to be far enough in the future to ensure that all sensor simulators have started and at least received one time message from the WeakTrueTime master until the local machine time exceeds the starting time. In our experiments defining the starting timestamp to be 60 seconds in the future was always sufficient. We recommend to run the simulation environment starting script on the same machine as the WeakTrueTime master, since this ensures that the starting timestamp has not already been exceeded in the WeakTrueTime. This could happen if the starting script is executed on a machine whose local machine clock is to far in the past (e.g., 5 minutes).

3.3.3.2 Sensor Simulator Architecture

This section presents the architecture of the Sensor Simulator project and the stream generation procedure. Appendix A.2 shows a class diagram of the project containing all methods and the most important attributes. To start a sensor simulator one has to specify the parameters listed in Appendix B. On startup the sensor simulator first instantiates and starts a time provider. Typically, a `WeakTrueTimeProvider` is used. However, the sensor simulator can also leverage other time provider implementations. Afterwards, a timed tuple reader object is instantiated and initialized. After both components are prepared, a TCP client socket addressed to the stream receiver is initialized. If this was successful, a

Algorithm 1 Sensor Data Stream Generation Loop Pseudocode

```
1: $curMatchPico \leftarrow 0$
2: $halfTimeStartPico \leftarrow matchStartPico$
3: $matchStartMachineMilli \leftarrow timeProvider.getTimeInMs()$
4: while $curMatchPico \leq matchEndPico$ do
5:     $sleep(t_{checkperiod})$
6:     $curMachineMilli \leftarrow timeProvider.getTimeInMs()$
7:     $curMatchPico \leftarrow MachineTimeHelper.generateMatchTimestampInPico(curMachineMilli, matchStartMachineMilli, halfTimeStartPico)$
8:     $List < Tuple > newTuples \leftarrow ttr.readTuplesProducesBeforeOrAt(curMatchPico)$
9:     $sendTuples(newTuples)$
10:    if $skipHalfTimeBreakInSimulation = true$ then
11:        if $curMatchPico > firstHalfEndPico \land curMatchPico < secondHalfStartPico$ then
12:            $matchStartMachineMilli \leftarrow timeProvider.getTimeInMs()$
13:            $halfTimeStartPico \leftarrow secondHalfStartPico$
14:        end if
15:    end if
16: end while
```

`java.io.PrintWriter`[5] (*outputToReceiverHost*) is created to write on the client socket output stream.

After establishing the connection to the stream receiver, the sensor simulator waits for a short while[6] in order to ensure that a potential WeakTrueTime timeslave deamon has received the first time message from the time master. Afterwards, the sensor simulator waits until the time delivered by the time provider instance exceeds the specified starting timestamp (*startingTimestampInMs*). When the delivered timestamp exceeded the specified starting timestamp, the sensor simulator starts its sensor data stream generation loop which keeps running until the end of the simulated match is reached. A detailed explanation of this loop is presented in the next paragraph. After finishing the stream generation the sensor simulator closes its connection as well as the timed tuple reader and stops the time provider instance.

Sensor Data Stream Generation Loop

In this paragraph, we will present the sensor simulator's main loop, i.e., the sensor data stream generation loop. The purpose of this loop is to generate and send the data stream produced by a single sensor (specified by the *filename* parameter) w.r.t. the real-time constraints using the time provider, the timed tuple reader and the TCP client socket. Algorithm 1 shows the pseudocode of the sensor data stream generation loop.

Prior to explaining the pseudocode in detail, we want to explain the semantic meaning of the variable names. All variables whose names end with *Pico* store timestamps in the metric of the sensor data tuple timestamps provided by the ACM DEBS 2013 Grand Challenge (see Section 1.1.1.1). That is, the timestamp is stored in picoseconds and is not related to the UNIX timestamp. In contrast, all variables whose names end with *MachineMilli* are stored in the time delivered by the chosen time provider implementation. Hence, the values are measured in milliseconds and related to the UNIX timestamp.

In the remainder of this paragraph, we will explain the pseudocode in detail:

Before entering the loop, the sensor simulator initializes three variables (see lines 1-3). *curMatchPico* is designated for storing the current match time, i.e., the simulated timestamp. Initializing this variable with 0 ensures that the simulation loop is entered. The next two

[5] http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html (07.08.2014)
[6] In our experiments we found out that 3 seconds are sufficient.

variables are required for calculating *curMatchPico* during the simulation. *halfTimeStart-Pico* stores either the starting time of the match or the starting time of the currently simulated halftime depending on whether the half time break should be simulated or not. In any case, it is initializes with the match starting time in the sensor data tuple metric. All important match timestamps (i.e, the start and end timestamps of both halftimes) are given in literature [2]. *matchStartMachineMilli* stores the time delivered by the chosen time provider implementation when the simulation starts.
The sensor data stream generation loop body (see lines 5-15) is executed until the currently simulated match time (*curMatchPico*) exceeds the provided end timestamp of the match (*matchEndPico*) (see line 4). In the first loop body line (i.e., line 5) the sensor simulator is set to sleep for $t_{checkperiod}$ milliseconds. Hence, with $t_{checkperiod}$ one can control how often the sensor data file is checked for new tuples to send in the stream. We decided to sleep in the beginning of the loop since immediately after starting the simulation no sensor data tuples are measured and thus available yet. After sleeping, the `getTimeInMs()` method of the chosen time provider implementation is called and the result is stored in *curMachineMilli* (see line 6). With this value, the current match time (*curMatchPico*) can be calculated (see line 7). Equation 3.8 and 3.9 show the mathematics used for this calculation. Subsequently, the result is used to fetch the next list of data tuples which has to be sent to the sensor data stream receiver (see line 8). This list contains all those data tuples which were not yet retrieved (and thus sent) and which are produced previous to the current point of time in the simulated match (i.e., *tuple.timestamp* $\leq$ *curMatchPico*). In the next line, these data tuples are sent to the receiver (or in other words appended to the sensor data stream) by printing them to *outputToReceiverHost*.

$$\begin{aligned} \mathit{machineDiffMilli} &= \mathit{curMachineMilli} - \mathit{matchStartMachineMilli} && (3.8) \\ \mathit{curMatchPico} &= \mathit{halfTimeStartPico} + \big(\mathit{machineDiffMilli} \cdot 10^9 \cdot \mathit{speedup}\big) && (3.9) \end{aligned}$$

The remainder of the loop body (see lines 10-15) is required for skipping the halftime break in the simulation. If this is desired, the sensor simulator checks in each loop iteration if the current point of time in the simulation is in the halftime break, i.e., between the end of the first halftime and the beginning of the second halftime (see lines 10-11). If this is the case, the sensor simulator reinitializes the *matchStartMachineMilli* as well as the *halfTimeStartPico* variable (see lines 12-13).

Time Provider Component

The main purpose of the time provider interface is to abstract the sensor simulator from the exact way the current timestamp is produced. Currently, we provide two time provider implementations: The `LocalMachineTimeProvider` and the `WeakTrueTimeProvider`. The `LocalMachineTimeProvider` is a very trivial implementation. It simply returns the current local machine time. In contrast, the `WeakTrueTimeProvider` is more sophisticated implementation which uses the WeakTrueTime library. If the sensor simulation environment is executed distributed on multiple machines, we highly recommend to use the `WeakTrueTimeProvider` implementation to avoid problems introduced by potential machine clock differences. Moreover, it is possible to design new time providers in future work. For instance, one could implement a time provider which fetches the current timestamp from a globally available atomic clock.

Timed Tuple Reader Component

In a nutshell, the timed tuple reader enables us to retrieve all data tuples from a sensor file (or more precisely from a buffer) which are measured before a given match timestamp. This functionality is provided by the `readTuplesProducedBeforeOrAt(timestamp)` method which returns a list containing all data tuples which were not returned yet and which were measured during the match before the point of time specified by the *timestamp* parameter (i.e., $\textit{tuple.timestamp} \leq \textit{timestamp}$). Additionally, the method ensures that a data tuple is not added to the result list if it is measured before the start of the soccer match (i.e., only adds a tuple if $\textit{tuple.timestamp} \geq \textit{matchStartPico}$).
However, instead of reading the sensor data file line by line in this method, the timed tuple reader leverages a buffer. Therefore, it extends the `PreBufferedFileReader<T>` class and uses its `pollElementFromBuffer()` method. The idea of the `PreBufferedFile-Reader<T>` is to read larger blocks of lines from a file into a buffer instead of reading every single line. The purpose of this is to improve the I/O performance. We have decided to implement the `PreBufferedFileReader<T>` as a generic solution and thus abstract from our specific problem (i.e., reading lines from a sensor data file and create tuples).

4 PAN - P2P Analysis Network

The main focus of this thesis is to develop a generic real-time complex event detection system. This system should be able to analyze multiple distributed input data streams and answer several client requests in parallel. Moreover, we require our solution to be scalable w.r.t. the number of client requests as well as with the data, i.e, with the number of input streams (e.g., produced by sensors) and with the complexity and number of statistics.
In this thesis, we propose PAN (P2P Analysis Network). PAN is a generic real-time complex event detection system which is able to analyze multiple distributed input data streams and handle several client requests. Moreover, PAN distributes its workload onto several workers in a P2P network. These workers are combined to workflows using a *pull-based* publish/subscribe approach. As a result, PAN is scalable in terms of both, data and client requests. Figure 4.1 illustrates PAN's position in the overall problem overview picture introduced in Section 1.2.
The remainder of this chapter is organized as follows. Section 4.1 presents the background of our work (i.e., Jergler's workflow-based solution [6]) and how we extend and improve the existing idea. Section 4.2 presents the PAN approach in detail and discusses our design choices. Finally, Section 4.3 gives some information about our implementation.
For illustration purposes, we will use the extended ACM DEBS 2013 Grand Challenge scenario (see Section 1.1) in the explanatory figures (e.g., workflows). Please note, that nevertheless, our approach is generic, i.e., is also applicable for other scenarios depart from the grand challenge.

4.1 Background

PAN is based on the work of Jergler et. al. [6] (see Chapter 2). Jergler et. al. propose a workflow-based architecture in which different workers (computing subtasks of the real-time complex event detection system) are connected with non-blocking ring buffers (LMAX Disruptor library [10]). This yields a sequential and parallel arrangement and connection of the workers (i.e., a workflow). The authors state that their idea can also be implemented in a distributed way by leveraging the publish/subscribe concept. However, they only present a centralized implementation which only exploits multiple cores but not multiple machines

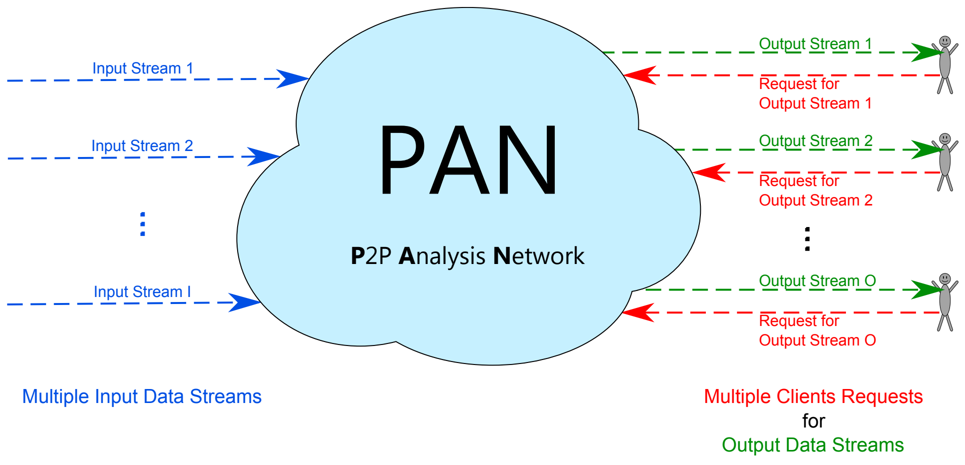

Input Stream 1
Input Stream 2
Input Stream I
PAN
P2P Analysis Network
Output Stream 1
Request for Output Stream 1
Output Stream 2
Request for Output Stream 2
Output Stream O
Request for Output Stream O
Multiple Input Data Streams
Multiple Clients Requests for Output Data Streams

Figure 4.1: PAN's Position in the Generic Problem Overview Picture (cf. Figure 1.5). The cloudy shape denotes PAN's distributed nature. To simplify the illustration each output stream is only requested by and sent to one client and each client only requests a single stream. However, in general, it is possible that an output stream is requested by several clients and each client can request several streams.

and they do not describe in detail how the distribution can be done. Moreover, since Jergler's solution only has a single input buffer (the *Distributor Ring-Buffer*), it is only able to handle and analyze a single input data stream. And last but not least, Jergler's solution does not fully solve the client request problem extension.
With PAN, we continue Jergler's architecture idea. More precisely, we extend and improve Jergler's work by four aspects. First, we implement the workflow-based architecture idea in a way that the workers can be distributed onto several machines (e.g., several cloud computing instances) and thus transform the idea into a very scalable approach. Second, we modify the architecture in a way, that it is able to handle and analyze multiple distributed input data streams instead of a single one. Third, we face the problem of how to answer several different client requests in difference to only providing a single HTML5 client. And fourth, inspired by the XML-file based data flow definition in TechniBall [8] we provide a JSON-file based workflow definition.

4.2 Concept

4.2.1 PAN at a Glance

PAN is built to be scalable in terms of both, data as well as the number of client requests (see Section 1.2). In the PAN approach, we obtain a highly scalable solution by distributing the workload of the real-time complex event detection system onto multiple machines instead of running the full system on a single machine. In doing so, we avoid bottlenecks and single points of failures.
To be precise, PAN distributes its workload onto several workers in a P2P network. As we state in Section 4.1, PAN is based on the workflow-based architecture idea of Jergler et. al. [6]. In Jergler's approach a workflow is composed of several workers (called *task elements*) computing subtasks. Hence, the full workload of the real-time complex event detection

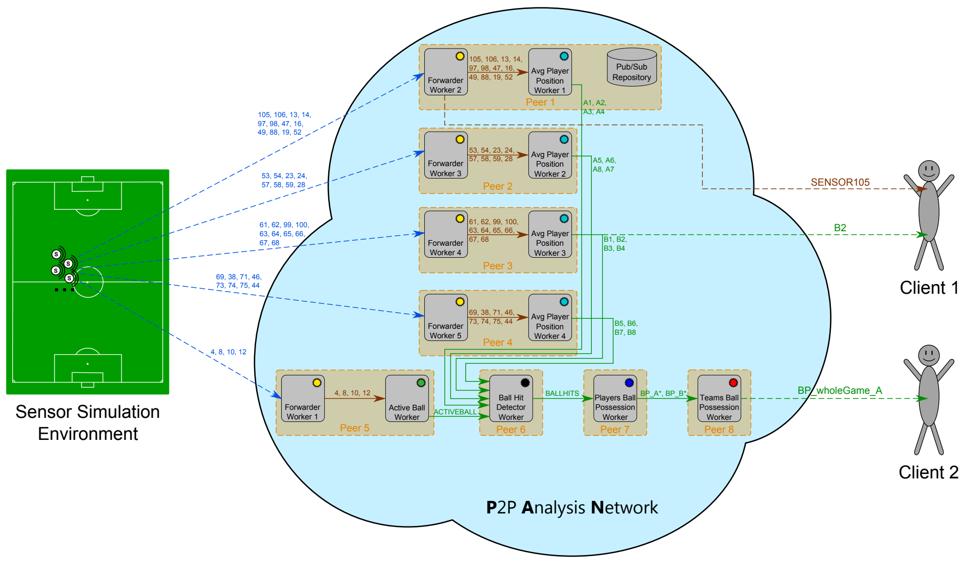


Figure 4.2: ACM DEBS 2013 Grand Challenge PAN Workflow Example. To simplify the figure multiple streams between two workers are illustrated with a single arrow. Moreover, the sensor data streams are abbreviated with their IDs (e.g., *106* instead of *SENSOR106*).

system is splitted onto several workers running on the same machine. In order to distribute the overall workload of the workflow (and thus also the real-time complex event detection system) onto multiple peers, we distribute the workers onto several peers in a P2P network. Figure 4.2 shows an example for such a workflow distribution.

In PAN each worker consumes one or multiple input data streams and analyzes them in one or multiple internal components. Each internal component generates one or multiple streams which are offered from the worker to the other PAN workers and clients. In order to connect these workers and clients, we leverage a publish/subscribe system.

However, we do not use the common *push-based* approach as for instance in OSIRIS(-SE) [14, 15]. In the push-based approach, the publisher is responsible for distributing its output streams to all subscribers. Instead, we use the *pull-based* approach. That means, that a subscriber of a certain stream is responsible for fetching the data from the publisher. A publisher of a certain stream can be retrieved from a publish/subscribe repository which is accessible by all PAN workers as well as from clients outside of PAN.

The major advantage which we obtain by using the pull-base approach is that we gain more flexibility than with the common push-based approach. For instance, this choice enables load balancing since the workflows are easily adaptable during runtime.

In the following sections, we will present in detail how the components of and the concepts used by PAN work and reason our design choices.

4.2.2 Architecture

PAN's main idea is to distribute the workload of the real-time complex event detection system onto several workers which are hosted on peers in a P2P network in order to obtain a highly scalable and flexible solution.

For doing so, we leverage an unstructured P2P network instead of a structured one. In our opinion it is always wise to use a structured P2P network (e.g., Chord [16], Kamelia [17], etc.), if one can benefit from the introduced structure. In particular, we argue that it is a good idea to use a structured P2P network, if the architecture can profit from the `lookup(key)` method or from the routing tables. However, in PAN this is not the case. Therefore, we neglect using a structured network to minimize the communication overhead and increase the freedom of design.

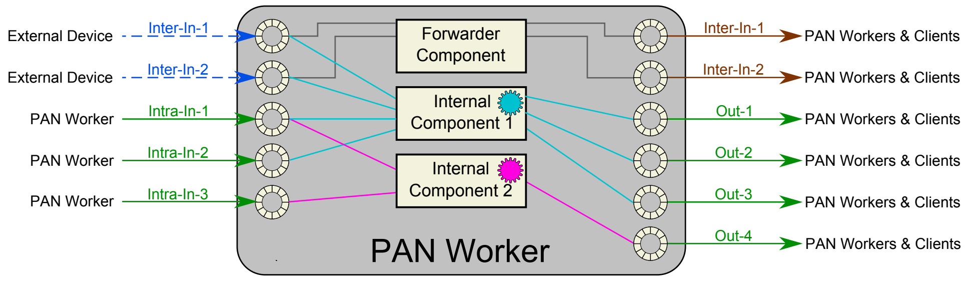
In PAN, a *peer*, i.e., a physical machine or cloud computing instance, can host a single or multiple workers. So, in fact, PAN is rather a W2W (Worker-To-Worker) network than a P2P network. With this design, PAN obtains the maximum degree of flexibility in terms of workflow definition. On the one hand, the whole workflow can be executed on a single machine (when hosting all workers on a single peer) if this machine is very powerful or the workflow has a small workload. On the other hand, a workflow can also be distributed onto thousands of (weak) cloud computing instances hosting workers which compute only small subtasks. We argue that this flexibility is fundamental since PAN is designed to be a generic solution and thus should be applicable for all use cases and deployable on all setups.

4.2.3 Workflow

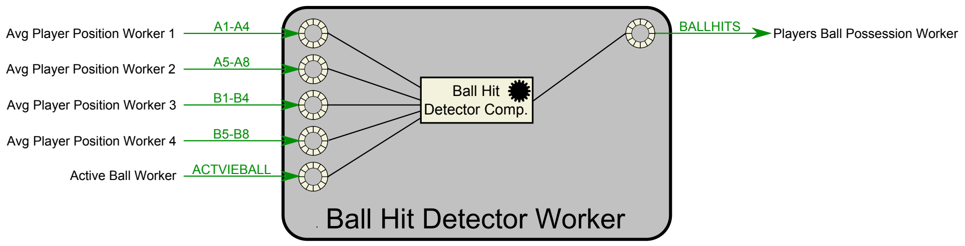
A PAN *workflow* is a sequential and parallel composition of PAN workers. Figure 4.2 illustrates an exemplary PAN workflow, which generates the player as well as the team ball possession statistic streams as defined in the grand challenge specification. For doing so, various intermediate streams (e.g., the average player position and the ball hits streams) are generated as output streams at some workers and consumed as input streams at other workers. For instance, the ball hits stream (i.e., *BALLHITS*) is produced as an output stream at the *Ball Hit Detector Worker* and consumed as an input stream at the *Players Ball Possession Worker*. These data exchange between the workers constructs the intra-PAN workflow. The devices producing the inter-PAN input streams (e.g., simulated sensors producing sensor data streams) are the starting points, i.e., *sources*, of a PAN workflow. Clients consuming the generated (intermediate) output streams are the *sinks* of a PAN workflow.

A static set of PAN workers building the intra-PAN workflow can be defined in JSON. A JSON config file defines the type, the host (i.e., the peer) as well as the expected inter-PAN input streams for each worker. Appendix E.1.3.1 shows the JSON config file for the workflow illustrated in Figure 4.2. The actual connection of the workers is done by means of a pull-based publish/subscribe system which is presented in Section 4.2.5. The clients are not defined in the JSON config. Instead, they can dynamically join the PAN workflow using the publish/subscribe repository and leave it by stop fetching the input streams.

To standardize the inter-worker communication, PAN workers always and only share data via network communication, indifferent if they are hosted on the same or on different peers. Hence, the intra- and inter-peer communication is performed in the same way. This can be achieved again be identifying a worker by a combination of the IP address and a parametrized port. The main advantage of this design specification is that if all communication is done via network, one does not have to perform case differentiations, i.e., check if two communicating workers run on the same peer or not.



(a) General PAN Worker Architecture. 2 inter-PAN input streams, 3 intra-PAN input streams, 3 internal components, 2 forwarded output streams and 4 generated output streams.


Avg Player Position Worker 1
Avg Player Position Worker 2
Avg Player Position Worker 3
Avg Player Position Worker 4
Active Ball Worker
A1-A4
A5-A8
B1-B4
B5-B8
ACTVIEBALL
Ball Hit Detector Comp.
BALLHITS
Players Ball Possession Worker
Ball Hit Detector Worker

(b) PAN Worker Example: Ball Hit Detector Worker (cf. Figure 4.2)

Figure 4.3: PAN Worker Architecture

4.2.4 Worker

The purpose of a PAN worker is to perform a subtask of the whole real-time complex event detection. More precisely, a PAN worker performs one step in the intra-PAN workflow which generates (statistical) output streams using the inter-PAN input streams (e.g., sensor data input streams) in several sequential and parallel steps. In a nutshell, a PAN worker does the following:

(1) First, it receives one or multiple input streams. These input streams can be either inter-PAN input streams (e.g., sensor data input streams) or intermediate intra-PAN data streams (e.g., average player position streams) produced by workers inside PAN.

(2) Subsequently, the worker uses these input streams in one or several internal components. These components can perform complex analyses or simply forward the input streams. In any case, each internal component (and thus the worker) generates one or multiple (intermediate) output data streams.

(3) Finally, the worker offers these output streams for further processing to other workers. Moreover, they can be requested by clients.

Figure 4.3(a) illustrates this process as well as the architecture of a PAN worker in general. To exemplify this general concept, Figure 4.3(b) shows a detailed view on the *Ball Hit Detector Worker* (cf. the exemplary workflow in Figure 4.2).
In the remainder of this section, we will present the PAN worker in detail. Section 4.2.4.1 gives information of how the worker handles its input and output streams, i.e., presents detailed information regarding (1) and (3). Section 4.2.4.2 present how the worker performs

its analysis tasks in the internal components (i.e., (2)). Finally, Section 4.2.4.3 discusses the consequences of the PAN worker's modularization and strict internal component separation.

4.2.4.1 Input/Output

Jergler et. al. use ring buffers in their architecture to connect two workers [6]. More precisely, Jergler et. al. use a single non-blocking ring buffer (LMAX Disruptor library [10]) for each intermediate stream, which is filled by the worker producing the stream and read by the worker consuming the stream. This is possible, since workers are implemented as tasks running on the same machine and thus able to access the same memory.
In contrast, in PAN, workers are distributed onto multiple peers in a P2P network. As mentioned earlier, they only share data via network communication. Hence, the producing worker (i.e., the *publisher*) and the consuming workers (i.e., the *subscribers*) cannot access the same single ring buffer. For instance, in the exemplary workflow the *Ball Hit Detector Worker* producing the *BALLHITS* stream and the *Players Ball Possession Worker* consuming it run on two different peers and thus cannot access the same memory.
We solve this problem, by equipping each worker with its own ring buffer per stream (see Figure 4.3(a)). Thereby, we differentiate between input and output ring buffers. The input ring buffers can only be read as input streams but not filled by the internal components. In contrast, the output ring buffers can only be filled by the internal components as output streams but not read.
We have to differentiate between intra- and inter-PAN input streams. An intra-PAN input stream is an intermediate stream produced by another PAN worker (e.g., *BALLHITS*, *A5*, etc.). In contrast, an inter-PAN input stream (e.g., *SENSOR4*, *SENSOR106*, etc.) is produced by a device outside PAN (e.g., a sensor simulator). In the remainder of this section, we will present how the input as well as the output for these two type of streams is handled by a PAN worker.

Intra-PAN Input Streams

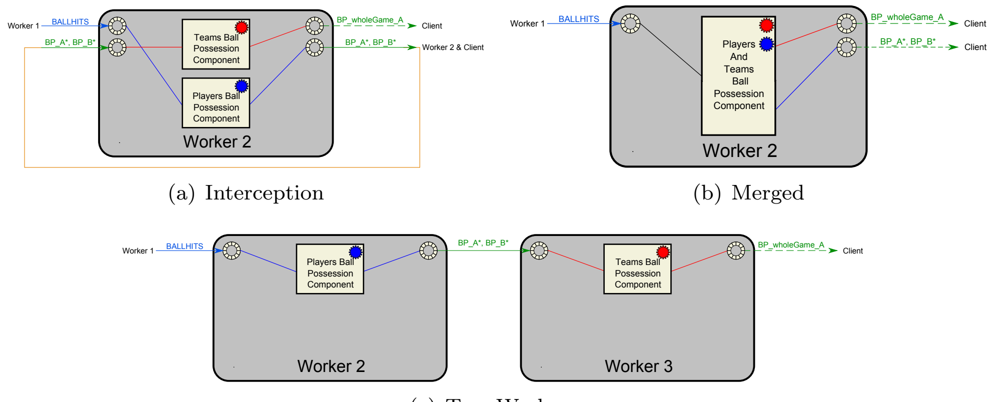
PAN follows a *pull-based* publish/subscribe communication approach. That means, that a consumer (i.e., *subscriber*) has to fetch the data from the producer (i.e., *publisher*). The reasons why we use this approach instead of the common *push-based* model in which the publisher is responsible to disseminate new tuples to all subscribers are presented later in Section 4.2.5.1.
PAN facilitates the pull-based data share model by means of REST-Interfaces. More precisely, each PAN worker runs a webserver which answers predefined REST-Interface requests (see Appendix C.2) using its output ring buffers. This REST-Interface can be used by all other PAN workers to access the workers output ring buffers and fill their own input ring buffers.
In PAN, a worker can either periodically fetch new tuples of a certain stream using the publisher's REST-Interface in a fixed interval in a separate thread or on demand when requested by an internal component. The advantage of the fetch on demand approach is that new tuples are only retrieved if they are needed and thus there is no useless network bandwidth consumption. However, the disadvantage of the fetch on demand approach is

that if an internal component requests new tuples it has to block until the REST request is answered. For evaluating PAN with the ACM DEBS 2013 Grand Challenge scenario we use the automatic periodic fetch approach since for instance the *Ball Hit Detector Component* requires many input streams and is very sensitive regarding timing aspects.

Inter-PAN Input Streams
In addition to the intermediate intra-PAN streams which are produced by PAN workers as output streams and consumed by other workers as input streams for further processing, there is another kind of input streams, i.e., inter-PAN input streams. Inter-PAN input streams are the actual input streams for PAN, i.e., for the real-time complex event detection system. Hence, they are produced by devices outside of PAN. For instance, a sensor data stream (e.g., *SENSOR4*) produced by a sensor simulator is an inter-PAN input stream.
We argue, that we cannot require devices generating the inter-PAN input stream to support our REST-Interface. In fact, we argue that it would be the best to not require anything from these devices since they are not a part of PAN. Any requirement on the devices producing the inter-PAN input streams would be an requirement on the scenario and thus reduce PAN's generality. Since PAN is thought to be a generic solution, we minimize the requirements as much as possible. As a result, the only requirement is, that the inter-PAN input streams are sent to a single predefined (in the JSON config) PAN worker via TCP. Disseminating the inter-PAN input stream to other interested workers is the job of the PAN worker which receives the input stream from its origin.
In order to achieve this job, each PAN worker creates an input and an output ring buffer for each expected (see JSON config) incoming inter-PAN input stream. At runtime, the worker listens for incoming TCP connections, accepts them and binds the incoming TCP streams to the corresponding input ring buffers, i.e., fills the input ring buffers with the incoming tuples. Moreover, a dedicated *Forwarder Component* periodically reads the tuples from each input ring buffer and writes them to the corresponding output ring buffer. As a result, the PAN worker which receives the inter-PAN input streams transforms them to intra-PAN streams and thereby offers them to all other PAN workers.

4.2.4.2 Internal Components

So far, we have only presented how the workers communicate, i.e., share their data (streams). In this section, we will present how the actual real-time complex event detection is performed in the internal components of the PAN workers.
A PAN worker runs one or multiple internal components. Each internal component is performed in a separate thread. Hence, all internal components are performed in parallel.
An internal component can use all input streams for its analysis task. That means, that it is able to read data tuples from all input ring buffers. Moreover, an internal component is able to generate one or multiple (intermediate) output data streams, i.e., fill the corresponding output ring buffers.
In PAN's design, we do not restrict what an internal component does. In fact, a worker can perform complex analyses on the input streams or simply forward them. Up to now, we implemented only two generic internal components: The *Forwarder Component* and


Worker 1
BALLHITS
BP_A*, BP_B*
Teams Ball Possession Component
Players Ball Possession Component
BP_wholeGame_A
Client
Worker 2 & Client
Worker 2
Players And Teams Ball Possession Component
Worker 3

(a) Interception (b) Merged

(c) Two Workers

Figure 4.4: Component Separation Problem

the *Repeater Component*. Both components simply forward their input streams without changing them. The *Forwarder Component* forwards all inter-PAN input stream and is executed on all workers with at least one expected inter-PAN input stream. In contrast, the *Repeater Component* forwards intra-PAN input streams to enable load balancing. This component and our load balancing concept will be presented in detail in Section 4.2.8.
Internal components performing actual analyses depend on the scenario. Thus, implementing them in a generic way is impossible. For evaluating PAN, we implement some components generating (intermediate) statistical output data streams for the ACM DEBS 2013 Grand Challenge. For instance, the *Ball Hit Detector Component* consumes the active ball (*ACTIVEBALL*) as well as all average player position (*A1-A8* and *B1-B8*) streams, detects all ball hits and generates an output stream (*BALLHITS*) containing the timestamp of the latest ball hit and the ID of the player who hits the ball. A list of all internal components is presented in Appendix D.1.

4.2.4.3 Component Separation and Single-Purpose Workers

In theory, PAN's architecture allows to perform multiple tasks by performing multiple internal components at the same worker. That is, for instance, it is possible to perform the player ball possession stream generation as well as its aggregation (to obtain the team ball possession streams) at the same worker by performing two components.
However, a worker's internal components are strictly separated. That means, a component can only use data from an input ring buffer. Hence, the *Teams Ball Possession Component* is not allowed to directly fetch the output of the *Players Ball Possession Component*. Instead, the worker has to receive the players ball possession streams as an input from itself (see Figure 4.4(a)) or the two components have to be merged to a single one producing all output streams (see Figure 4.4(b)).
The advantage of this design choice is that the worker's architecture is cleanly modularized and the way internal components can get input and distribute output data is standardized. However, the consequence is, that we only use single-purpose workers in our evaluation. That means, that in our workflows each worker only performs a single task and thus only

performs a single analysis component or multiple components of the same type (e.g., a single *Player Average Component* for each player). We argue that it makes more sense to split the workload to two subsequent workers, since in this case it is possible to distribute these two workers onto two different peers (see Figure 4.4(c)). The only exception is that, by design, each worker which expects inter-PAN input streams performs a single *Forwarder Component* which forwards all inter-PAN input streams. A list of all PAN workers we implemented is presents in Appendix D.2.

4.2.5 Publish/Subscribe

Up to this point, we have neglected the question of how a PAN worker (or a client) can find a publisher of a certain stream. To solve this problem, we follow Jergler's suggestion [6]. Hence, we leverage a publish/subscribe system and implement the workflow as a "set of subscriptions" [6]. More precisely, we use a *pull-based* publish/subscribe approach. In the remainder of this section, we will present in detail how the set of workers is connected to a workflow and discuss our design considerations.

4.2.5.1 Pull-Based Approach

The general idea how to use a publish/subscribe system to connect workers to a workflow is simple: Each worker has to publish all its output data streams. As a consequence, these streams can be subscribed by other workers. Hence, in PAN, each worker has to publish all its output streams (or output ring buffers) and subscribe all input streams to be able to fill its input ring buffers.
However, the way we implemented the publish/subscribe system is also PAN's unique selling point. The reason for this is that we use a *pull-based* instead of the common *push-based* publish/subscribe approach. In the push-based approach, a publisher is responsible for distributing its output streams to all subscribers. In contrast, in the pull-based approach, a subscriber is responsible for fetching the data from the publisher.
This design choice introduces a major consequence: It changes the workflow definition direction. In the common (i.e., push-based) approach, the workflow is defined from the source to the sink. Thus, a publisher has to know all receivers (i.e., subscribers) of its output streams. In contrast, with the pull-based approach, the workflow is defined backwards, i.e., from the sink to the source. Hence, the publisher does neither have to keep a list of all subscribers of its output streams nor push new data tuples to all of them. Instead, a subscriber asks for a publisher of a certain stream and fetches new tuples on its own.
One can easily see with a small example, that the pull-based approach is the more natural one. Assume, there is a baker who produces bread. In the real-world a baker normally does not have a list of all customers who may want bread and delivers bread to all of them if new bread is available. Instead, a customer searches for a baker who has bread and gets it by itself whenever the customer needs bread.
The major advantage of the pull-based approach is that it is more dynamic and flexible than the push-based approach. First, the pull-based approach enables subscribers to fetch data on demand or with different intervals. For instance, if a subscriber only needs the latest

tuple once per second, it has no benefit from receiving all tuples which are produced with 100Hz at the publisher. Thus, supporting this feature can reduce the traffic enormously, especially since also client requests are implemented by means of subscriptions (see Section 4.2.6). While fetching new data tuples with different intervals (or even on demand) comes for free with the pull-based approach, one has to implement this feature on top of the push-based approach. Moreover, the pull-based approach facilitates adapting the workflow easily during runtime. The reason for this is that the publishers do not need to know the subscribers. Thus, the workflow can be modified by adding new workers without changing anything at the preceding workers. Especially new clients can join the workflow, fetch some data tuples, and leave the system without any changes in the intra-PAN workflow. In fact, the PAN workers do not even notice the joining and leaving clients apart from an increasing or decreasing number of REST requests. Furthermore, this enables load balancing by adding new *Repeater Workers* during runtime if the number of client request grows. More details to this topic are presented in Section 4.2.8.
However, the pull-based approach has also a disadvantage compared to the push-based approach. When using the push-based approach a publisher sends new tuples immediately after generating them. Hence, the subscriber receives the new tuples as early as possible. In contrast, in the pull-based approach there is an additional time gap (Δt) between generating a new tuple and receiving it at the subscriber. This time gap is regulated by the time interval ($t_{fetchInterval}$) in which a subscriber fetches new tuples (cf. Equation 4.1). Unfortunately, this time interval cannot be arbitrarily small, since a too small interval would introduce an enormous number of REST requests at the publisher. Thus, using the pull-based approach increases the query delay, i.e., the time the system needs to generate the output streams using the inter-PAN input streams (see Section 5.2.2), in any case.

$$\max \Delta t = t_{fetchInterval} \tag{4.1}$$

In fact, the $t_{fetchInterval}$ introduces a trade-off between a small query delay and a small network consumption and computational effort. As decreasing the interval decreases the query delay, it also increases the network consumption and the computational effort on the workers hosting the webservers. In contrast, increasing the interval reduces the network consumption and the computational effort but increases the query delay. Finding the perfect fetch interval depends on the scenario, i.e., on the velocity of events.
Nevertheless, we argue that the flexibility introduced by the pull-based approach outweighs this disadvantage since it enables us to fully benefit from the P2P network which underlies PAN.

4.2.5.2 Publish/Subscribe Repository

The remaining question to answer is how to perform a subscription. Both approaches, i.e., the *push-based* as well as *pull-based* one, require the possibility to retrieve the publisher of a certain stream. More precisely, a subscriber needs the contact information (i.e., the IP and port) of the publisher. In the push-based approach, a subscriber uses these contact information to inform the publisher of its interest. In PAN (i.e., the pull-based approach)

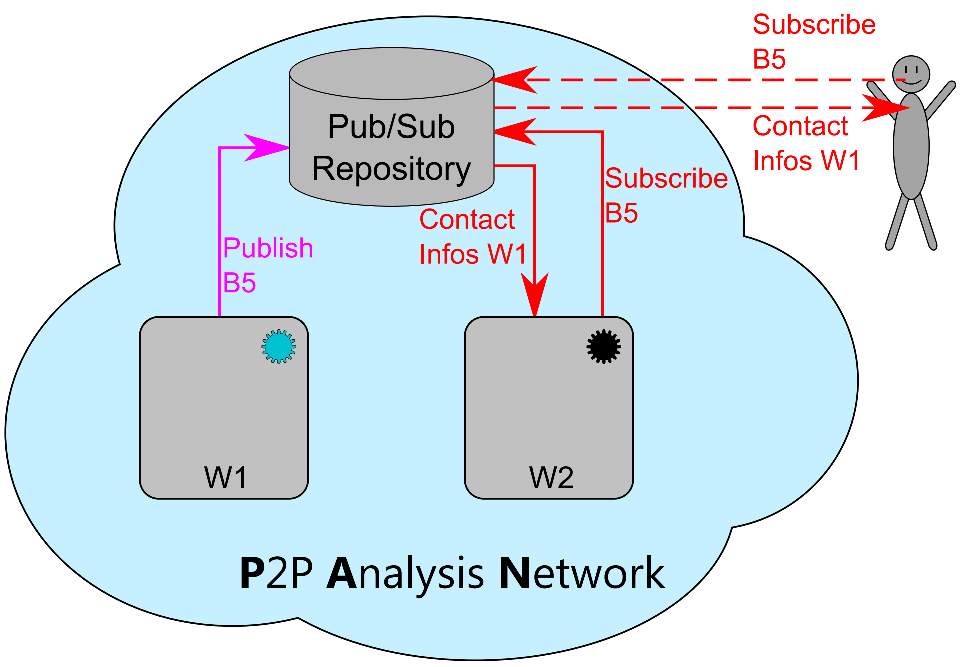


Figure 4.5: Publish/Subscribe Process

the subscriber needs the contact information to fetch data tuples using the REST-Interface (see Section 4.2.4.1).
We implement a central publish/subscribe repository, which stores the publisher contact information for each stream. The repository stores a mapping from the stream identifier to a list of publishers or more precisely to a list of contact informations. The ability to store multiple publishers is required for enabling load balancing (see Section 4.2.8).

$$\textit{Data Stream} \rightarrow \textit{List<Publisher>}$$
$$(\textit{e.g., B5} \rightarrow [\textit{Worker1, Worker3}] = [\textit{1.2.3.4:1234, 1.2.3.5:5678}])$$

As the PAN workers, PAN's publish/subscribe repository runs a webserver which answers predefined REST-Interface requests (see Appendix C.1) for communication purposes. Thus, the publish/subscribe repository is simply a distributed accessible map.
At runtime, each worker has to *publish* all its output streams by means of this REST-Interface. Subsequently, all intra-PAN workers and clients outside PAN can retrieve the contact information of this publisher by *subscribing* the published stream via the REST-Interface. Figure 4.5 illustrates this process.
We implement a central publish/subscribe repository for the first version of PAN. We argue, that this does not harm the PAN approach, as the central repository is currently no bottleneck since each PAN worker only contacts the repository once in the beginning for each of its subscriptions. Later only clients contact the repository once per subscription. All the data transmissions are done in a P2P way between two workers or between a worker and a client. Thereby, PAN's P2P network is comparable to Napster. The publish/subscribe repository is the central component which is only accessed once per subscription to get the contact information. All other communication is done in a P2P fashion.

In our future work, we plan to eliminate this central component to obtain a fully distributed and thus scalable system. We suggest to replace the central publish/subscribe repository by an distributed one. Therefore, it could be beneficial to use a structured instead of an unstructured P2P network and use the possibilities provided by a DHT (e.g., Chord [16]). Alternatively, one could implement the contact information retrieval (i.e., subscription) by means of group communication techniques in the P2P network. In this case, one could remove the publish/subscribe repository and neglect publishing output streams.

4.2.6 Client Requests

Another advantage of the *pull-based* publish/subscribe approach is, that it enables clients to join the workflow as sinks as well as leave it without changing anything in the intra-PAN workflow. Thus, client requests can be implemented by means of subscriptions.
More precisely, a client outside PAN simply acts like a PAN worker to obtain an (intermediate) output data stream. That means, that a client first has to contact the publish/subscribe repository to retrieve the contact information of a publisher for a certain stream (see Figure 4.5). Subsequently, the client can use the received contact information to fetch the data tuples from the publishing PAN worker's output ring buffer just in the same way as other PAN workers do, i.e., by using the REST-Interface (see Section 4.2.4.1).
With this design choice, we also limit the client requests. A client is only able to receive data streams, which are output streams of an arbitrary PAN worker. This includes all inter-PAN input data streams as well as all generated (intermediate) output data streams. For instance, in the exemplary workflow (see Figure 4.2) a client can request the ball possession stream for team A (*BP_wholeGame_A*), the average position stream of player B2 (*B2*) or the sensor data stream of the referee's left shin guard (*SENSOR105*). But, a client cannot perform more complex queries (e.g., "*SELECT playername WHERE ballPossession > 10 %*") unless a PAN worker generates an output stream with results for exactly this query. If a client needs such complex queries, it has to subscribe for all necessary data streams (e.g., the ball possession streams of all players) and implement the query by itself.

4.2.7 Consequences of the REST-Interface Communication Approach

As we presented in the previous sections all communication between PAN workers, clients and PAN's publish/subscribe repository is done via REST-interfaces with JSON objects. This encompasses fetching output streams (i.e., new data tuples) from PAN workers as well as publishing and subscribing streams at the repository. To enable this, each PAN worker and the publish/subscribe repository run a webserver which is able to answer a predefined set of REST requests (see Appendix C) with JSON objects.
The main benefit of using REST-Interfaces for data transfer is that REST-Interfaces are language independent. Hence, it is possible to write clients in all languages which are able to perform HTTP requests. For instance, we implement Java as well as lightweight Python clients for evaluating PAN (see Section 5.2).
However, the disadvantage of REST-Interfaces is that they introduce avoidable communication overhead (i.e., the HTTP Header and the JSON syntax) and computational effort for

running the webserver. Especially in scenarios with traffic intensive workflows, this can be a problem. Hence, we plan to facilitate using binary JSON or even other communication technologies (e.g., Java RMI or SOAP) at least for the intra-PAN communication between PAN workers in our future work.

4.2.8 Load Balancing

So far, we have only presented how we achieve PAN to be scalable w.r.t. the data. Increasing the amount of data, i.e., the number of statistics or the number of input streams, or the complexity of the data, i.e., the complexity of the analyses in the internal components, results in an increase of computational effort. In PAN we accomplish this by distributing the overall workload onto multiple peers in a P2P network. More precisely, we distribute the PAN workers onto multiple peers and connect them with a *pull-based* publish/subscribe approach.

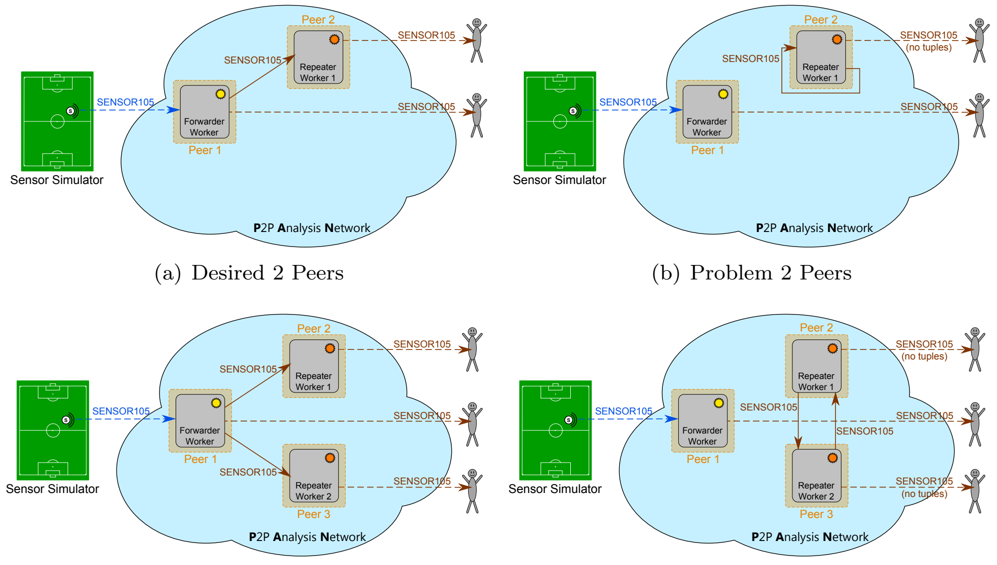
But, as presented in Section 1.2, we further want our system to be able to scale with an increasing number of client requests. Up to now, each stream is only published by one PAN worker. This is either the worker which generates the stream in an internal component (e.g., the *Teams Ball Possession Worker* for the *BP_wholeGame_A* stream) or the receiver and thus forwarder of the inter-PAN input stream (e.g., *Forwarder Worker 2* for *SENSOR105*). In any case, there is only one publisher for each stream.

Assume that for instance in the 2014 FIFA World Cup final match thousands or even millions of clients request the ball possession statistics stream of the German team after shooting the match-winning goal in the extra time. A single PAN worker would not be able to answer all these requests. Both, the network connection and the computational power of the peer hosting this worker would become a bottleneck.

In such situations it is necessary to perform load balancing, i.e., to distribute the client requests for a certain stream to multiple publishers of this stream. As mentioned in the previous sections, PAN enables load balancing. More precisely, the current version of PAN is able to balance the subscriptions for a certain stream onto the set of publishers of this stream and by this distribute the load of the incoming client requests.

In order to be able to create additional publishers of a certain stream, we introduce the *Repeater Component*. The *Repeater Component* simply forwards a given (in the JSON config) set of intra-PAN input streams which are published from another PAN worker without changing them. Therefor, the repeating worker has to perform a subscribe request for the stream at the repository to retrieve the contact information of the publisher, send a publish request to the repository to add itself as a publisher for the stream and repeat the stream by fetching the tuples from the producers output ring buffer and adding them to its own output buffer.

As we mentioned in Section 4.2.5.2, the publish/subscribe repository is able to store a list of worker contact information. Moreover, as a first trivial load balancing we implemented the repository in a way that it returns a random publisher out of the set of publishers for a certain stream. Unfortunately, this is not sufficient to avoid all anomalies which we summarize in Figure 4.6.



(a) Desired 2 Peers (b) Problem 2 Peers

(c) Desired 3 Peers (d) Problem 3 Peers

Figure 4.6: Replicator Problem

Assume a simplistic scenario with a forwarder worker (i.e., *Forwarder Worker*) producing the stream (*SENSOR105*) and a single repeater worker (i.e., *Repeater Worker 1*). Figure 4.6(a) shows the desired workflow for this scenario. However, without further provisions it is also possible that the workflow illustrated in Figure 4.6(b) emerges. This is due to the fact that all PAN workers publish their output streams, before they subscribe the input streams. This is necessary since otherwise a worker cannot subscribe a stream published by another worker. But, this can also result in the repeater worker retrieving itself as the publisher for the stream it want to repeat. The trivial solution for this problem is to forbid a repeater worker to accept itself as a publisher for an input stream. Unfortunately, this is still not sufficient if we extend the scenario by a second repeater worker (i.e., *Repeater Worker 2*). Figure 4.6(c) illustrates the desired workflow for this scenario. However, also the workflow shown in Figure 4.6(d) can arise.

In the current implementation of PAN we solve this problem by means of the *rep* parameter in the publish request. This parameter indicates if the published stream is generated by a repeater component. The publish/subscribe repository stores this information jointly with the contact information of the publisher. Moreover, the subscribe request is enriched with a *norep* parameter. If this parameter is set to true, the repository only returns the original producer of the stream.

In our future work, we plan to replace this boolean model with a level model in order to facilitate hierarchical repeater structures (see Figure 4.7). In this new model, each publisher has a *repeater level* indicating its position in the hierarchical structure. This level is passed with the publish request and stored in the repository. Moreover, the repository is only allowed to return worker contact information of a worker preceding the repeater level of the subscriber.

Furthermore, as we state in Section 4.2.5.1, the flexibility we gain by using the pull-based

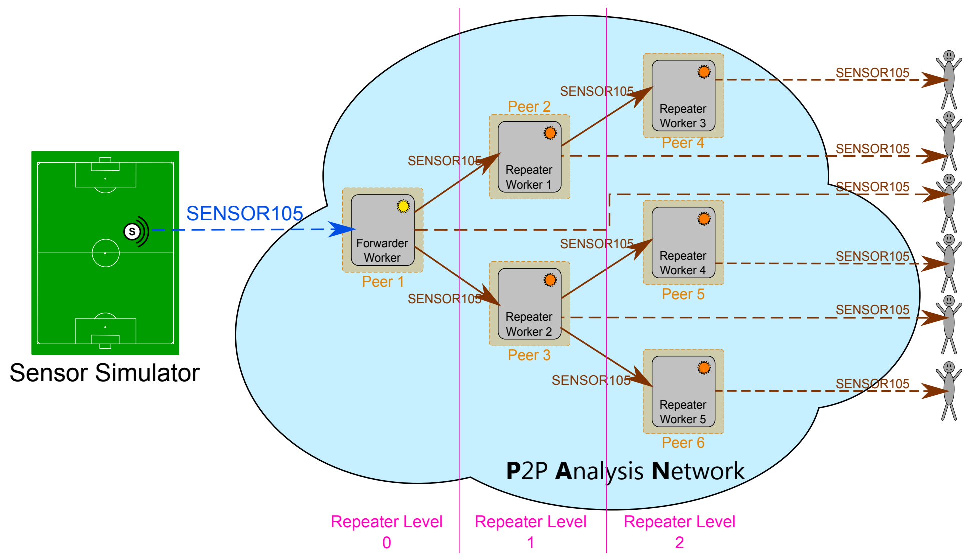


Figure 4.7: Hierarchical Repeater Structure

approach, enables adding new repeaters at runtime. More precisely, it is possible to observe the load in the system and dynamically add new repeater workers if the existing publishers cannot handle all client requests. These new repeater workers could be hosted on new cloud instances which join PAN's P2P network at runtime. If the number of clients decreases, a cloud instance could leave the P2P network and be shutting down. Hence, with the flexibility of the pull-based approach, we can benefit from the pay-as-you-go cloud computing model. We plan to implement such an observer and evaluate PAN w.r.t. its flexibility in our future work.

4.3 Implementation

In this section, we want to give some information about our current implementation of the PAN concept. The purpose of the first prototype is to evaluate if the PAN approach, i.e., distributing a real-time complex event detection system onto multiple workers in a P2P network which are connected to a workflow using a pull-based publish/subscribe system, is feasible.

We decided to implement the first PAN prototype fully in Java in order to obtain a platform-independent prototype. The prototype performs properly on Ubuntu 12.04 as well as on Mac OS X 10.7.5 machines. Moreover, it should also be possible to run a PAN worker or the publish/subscribe repository on a Windows machine. Obviously, the performance of the prototype could be increased by implementing it in C++ or another low level programming language. However, we neglect that for the first prototype and postpone implementing a faster prototype to future work.

In the remainder of this section, we want to give some brief information about the libraries we use in the prototype. Moreover, Section 4.3.4 states which ACM DEBS 2013 Grand Challenge queries (see Section 1.1.1.3) can be answered with the current implementation.

4.3.1 Ring Buffer

As we summarized in Section 2.2, three out of six grand challenge solutions explicitly state that they use ring buffers in their solutions. So we do in PAN. Jergler et. al. [6], i.e., the work PAN bases on, as well es Wu et. al. [5] both leverage the LMAX Disruptor library [10] in their implementation.

As presented in literature [18], the LMAX Disruptor library significantly outperforms all existing (bounded) queue approaches for linking processing stages in pipelines. The reason for this is that the LMAX Disruptor, is designed to address all problems of queues when used for binding producers and consumers. Evaluations confirm that the LMAX Disruptor library is the perfect component for linking processing stages in a pipeline.

Distributing the complex event detection system onto several linked workers (i.e., a workflow) is also a kind of a processing pipeline. Hence, albeit the LMAX Disruptor is designed for linking different threads on a single machine and not different processes distributed on several peers, we considered using the LMAX Disruptor library for implementing the worker input and output ring buffers in PAN.

Unfortunately, it is no option to use the LMAX Disruptor in our PAN prototype since the LMAX Disruptor follows an event-based consuming approach. More precisely, in the LMAX Disruptor approach, a consumer does not decide on its own when to read new entries from the ring buffer (cf. pull-based). Instead, a consumer is notified on new entries (cf. push-based) and has to handle these events in a callback method. This fundamentally disagrees with PAN's design concept, since PAN follows the pull-based approach, i.e., in PAN the subscribing worker is responsible for fetching new data stream tuples from the output ring buffer of the publisher by sending a REST request. In consequence, an event-based consuming approach in which the publisher sends new entries and the subscriber handles received entries is not reconcilable with PAN's design concept. Hence, we cannot benefit from the existing open source LMAX Disruptor library.

Unfortunately, the `java.util.concurrent.ArrayBlockingQueue`[7], which is used as an evaluation reference to the LMAX Disruptor and stated to have "the highest performance of any bounded queue" [18] is not applicable for implementing the ring buffers in PAN. The reason for this is that the `ArrayBlockingQueue` blocks on putting a new element in the queue if the queue is full. In contrast to this behavior, we want to override the oldest element.

Therefore, we decided to implement our own ring buffer which is adapted to fit PAN's needs. More precisely, we build a wrapper around an array. Thereby, we ensure immutability from outside and a proper synchronization. Moreover, we took into account the considerations presented in literature [18].

4.3.2 REST-Interfaces

As we presented in Section 4.2, each PAN worker as well as PAN's publish/subscribe repository run a webserver for answering incoming REST requests. In our prototype implementa-

[7] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html (07.08.2014)

tion we leverage the Jetty library[8] [19]. More precisely, we use Jetty for two purposes. First, we use Jetty's HTTP server to implement our REST request handlers, i.e., the webservers answering incoming REST requests at the repository and at the workers. Second, we use Jetty's HTTP client to perform the requests at the PAN workers, i.e., to perform publish as well as subscribe requests and to fetch data tuples from other workers.
The REST-Interfaces return JSON objects for all requests. The sole exception are the */debug* requests which are answered with a HTML page. In our prototype, we use the Google Gson library[9] [20] for converting Java objects into JSON objects.

4.3.3 Logging

In distributed systems, and especially in the first prototype of a distributed system, logging is indispensable for finding bugs. Therefore, we leverage the Log4j library[10] [21]. More precisely, we use the Log4j library for performing logs in the PAN workers, PAN's publish/subscribe repository and in the sensor simulation environment. Moreover, we bind the existing logging mechanism of Jetty to our Log4j config using SLF4J[11] [22]. During the evaluation we printed all log messages with an error level higher or equal to *INFO* to the console and into dedicated log files.

4.3.4 ACM DEBS 2013 Grand Challenge Workflow

In order to evaluate PAN we implemented internal components and workers for two workflows. The first workflow (*Full Game*, see Appendix E.1), generates ball possession streams for the players as well as for the teams. Thereby, this workflow answers the ball possession queries as specified in the ACM DEBS 2013 Grand Challenge (see Section 1.1.1.3). The second workflow (*Heat Map*, see Appendix E.3) generates streams for the heat map queries in different resolutions for a single player.
We neglect, implementing a workflow which produces streams for all queries in parallel as required of a grand challenge solution for two reasons. First, we only want to use the extended ACM DEBS 2013 Grand Challenge scenario as an evaluation base for PAN. Thus, we relinquished writing internal components for all queries but instead spent the limited time for this thesis on improving PAN. Second, when using the current prototype, such a workflow would require too many cloud instances, since, unfortunately, our resources in this thesis were limited to 14 Windows Azure cloud instances.
Appendix D lists all internal components and workers we implemented for these two workflows. Our implementations are highly inspired by the published ACM DEBS 2013 Grand Challenge solutions (see Chapter 2) and thus no new (and not our own) ideas. Moreover, we suppose that the implementation of our components is not optimal but can be improved. However, as we mentioned above, we only used them as an evaluation based for PAN and thus neglect further improving them.

[8] Jetty version: jetty-9.1.3.v20140225 - 25 February 2014
[9] Google Gson version: 2.2.4
[10] Log4j version: 2.0 rc1
[11] SLF4j version: 1.7.6

4.3.5 Launch and Deployment Scripts

In order to evaluate PAN, we have written Python and Bash scripts for deploying and launching PAN on multiple machines (e.g., Microsoft Azure cloud instances). In a nutshell, these scripts use SCP to deploy the prototype on the machines and SSH to start the workers as well as the publish/subscribe repository.

5 Evaluation

In this chapter, we present how PAN performs in a distributed environment under various conditions. However, prior to this, we will briefly evaluate our sensor simulation environment in order to make sure that we have a proper evaluation base, i.e., proper input data streams. Subsequently, in Section 5.2 we will present and discuss the evaluation results of PAN.

5.1 Sensor Simulation Environment

In order to evaluate PAN it is necessary that we have a proper evaluation base. That means, that we have to make sure, that our sensor simulation environment generates sensor data input streams in the same way as if they were produced by sensors measuring a currently ongoing soccer match.

As we mentioned in Section 3.1.3.2, the most important is to ensure that all sensor simulators generate data streams with data from the same point of time in the match. We further argued, that this can be achieved by guaranteeing that the time difference between two input data streams does not exceed a certain threshold. In this section, we will confirm that our sensor simulation environment (see Section 3.3), i.e., the *Sensor Simulator* using our *WeakTrueTime* library, is able to guarantee this and thus generates input data streams which are a proper evaluation base for PAN.

For doing so, we generate the sensor data streams on a corrupted environment and measure the time difference of the incoming sensor data streams at a dedicated *Debugging Stream Receiver*. More precisely, we simulate all 42 sensors on a single as well as distributed onto two machines. These two machines are connected via ethernet and have a ping around $1.6ms$[12]. In the distributed case, the sensor simulators are equally distributed onto both machines, i.e., each machine simulates 21 sensors. In order to corrupt the distributed environment, we manipulate the machine clock of the second machine, i.e., we set it approximately 20 seconds into the past.

Appendix F.1 shows the configuration of the sensor simulator we use for the evaluation, i.e., lists the values of all configuration parameters. The sole parameter we vary during the

[12] Ping statistics: rtt min/avg/max/mdex = 0.913/1.575/2.923/0.284 ms

evaluation is the *USE_WEAK_TRUE_TIME* variable. That means, that we evaluate the sensor simulation environment with WeakTrueTime[13] as well as the local machine time as the time provider. As a result we obtain four evaluation setups:

(1) Single machine, without WeakTrueTime

(2) Single machine, with WeakTrueTime

(3) Distributed (two machines), without WeakTrueTime

(4) Distributed (two machines), with WeakTrueTime

In our evaluation, we simulate the full game in real-time (i.e., no speedup) for all four setups. That means, that we run the whole sensor simulation environment four times. During each run the debugging stream receiver calculates time difference statistics in a 5 seconds interval. More precisely, every 5 seconds the standard deviation of the timestamps of the last received tuple of all sensor data streams is calculated (cf. Equation 5.1). Moreover, the median difference to mean (cf. Equation 5.2) is calculated. In addition, a moving average with a sliding windows size of 10 is calculated for both statistics.

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})} \tag{5.1}$$

$$\tilde{d} = \underset{i=1,\ldots,n}{\operatorname{median}} |x_i - \bar{x}| \tag{5.2}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, $n = 42$ and x_i = latest timestamp of the ith sensor

Figure 5.1 compares the moving averages of the standard deviations for all four setups. As one can easily see, the time differences in setup (1), (2) and (4) follow approximately the same schema. Up to approximately 23.5 minutes ($\approx 1.4E6ms$) and between the 41th minute ($\approx 2.5E6ms$) and the end of the simulation ($\approx 70min = 4.2E6ms$), i.e., during the second half time, the value fluctuates between 100 and 1000ms. We argue, that this can be regarded as the same situation in the game. As expected using WeakTrueTime as the time provider has no effect in the single machine setup. However, using WeakTrueTime as the time provider compensates the corrupted environment in setup (4). In contrast, in setup (3) the value slightly varies around 10 seconds. This exactly meets our expectations, since 10 seconds is the half of the time difference between both machines. Moreover, this confirms that the WeakTrueTime library works and is required for executing the sensor simulation environment distributed onto multiple machines since one cannot exclude clock differences. Unfortunately, the time difference explodes between the 24th and the 41th minute, i.e., in the end of the first half time and during the half time break. In this time interval, none of the four setups could produce sensor data streams with an acceptable time difference statistic. However, we argue that this is not a problem of sensor simulation environment but of the

[13] The first machine with the correct clock is the WeakTrueTime time master.

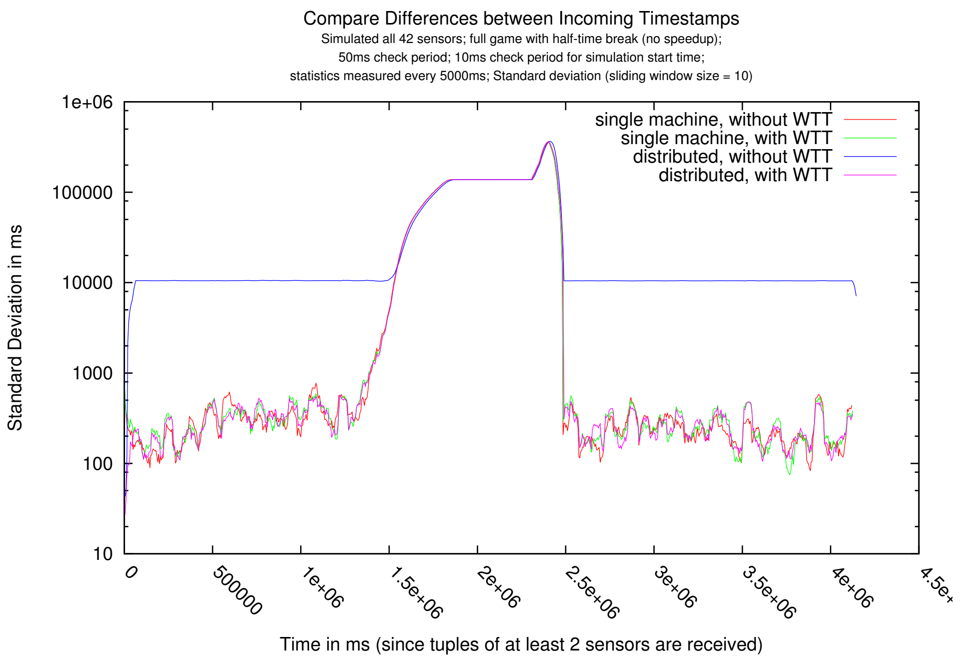


Figure 5.1: Comparison of the time difference of incoming sensor data streams produced by the sensor simulator environment in milliseconds for four different setups. The graph shows the standard deviation (sliding window average) for all four setups.

provided evaluation data. The organizers of the ACM DEBS 2013 Grand Challenge state, that the active ball sensor had problems in the end of the first half time[14]. We suppose that the technical problems even started earlier. Moreover, we suppose that there are no proper sensor data during the halftime.

Appendix G.1 presents additional graphs containing the remaining statistics. In a nutshell, the median difference to mean statistic (cf. Equation 5.2) has the same trend as the standard deviation statistic. However, the median difference is always lower than the standard deviation. This indicates, that the standard deviation is biased by outliers with large time differences.

In conclusion, we state that our sensor simulation environment fulfills the requirements. More precisely, our implementation is able to produce data streams from the same point of time in the game, i.e., with a small time difference, even if it is executed on a corrupted distributed environment. Hence, the our sensor simulation environment can be used as an evaluation base for evaluating PAN.

[14] Quotation: "Towards the end of the 1st half we had technical issues with the locating system so that the last 2.5 minutes are without the active ball transmitter (see without ball above). Hence, the shot on goal and ball possession query cannot produce valuable information for that time." [3]

5.2 PAN

In this section, we will present the evaluation of the main contribution of this thesis, i.e., the evaluation of PAN. In this evaluation we will explore PAN's applicability and performance characteristics. Therefor, we will answer the following questions:

- Where can PAN be deployed? Are there any requirements on the peers (e.g., their CPU power) or on the way they are connected, i.e., on the intra-PAN network properties?
- What are the performance characteristics of PAN? Is PAN able to answer the queries specified in the ACM DEBS 2013 Grand Challenge in real-time? And how long does PAN need to generate the result stream for a certain query, i.e., how large is PAN's query delay?
- Which degree of distribution is useful and beneficial? That is, how many peers are required to perform a certain workflow in real-time? And does it harm PAN's performance to further distribute the workflow?
- Does PAN, or more precisely our implementation of the DEBS specific internal components, generate correct statistical data streams? And are these results consistent and reproducible?
- Does PAN's load balancing feature work? That means, is PAN able to scale w.r.t. an increasing number of client requests?

The remainder of this section is organized as follows. First, Section 5.2.1 presents the settings and the setup for our evaluation and Section 5.2.2 presents our metric for evaluating PAN's performance, i.e., the query delay metric. Section 5.2.3 contains all evaluations we performed on the big exemplary workflow presented in Chapter 4. The evaluation results of the load balancing feature are presented in Section 5.2.4. Subsequently, Section 5.2.5 presents the visualization clients. A concluding discussion of the evaluation results is given in Section 5.2.6.

5.2.1 Setting

In order to evaluate PAN with the largest degree of distribution as possible, we leverage a virtual machine setup (i.e., the Microsoft Azure Cloud). More precisely, each peer in the evaluation workflows is a Microsoft Azure cloud instance[15]. In contrast, the other components, i.e., the sensor simulators and the clients, are performed on a laptop[16]. This is due to the way the query delay is measured (see Section 5.2.2). The sole exception of this segmentation, are the lightweight Python clients in the load balancing evaluation (see Section 5.2.4). These clients are performed on cloud instances which are not in use for hosting PAN workers.

[15] Cloud instance specifications: Small VM (Standard A1), 1 core 1.6GHz CPU, 1.75GB RAM
[16] Laptop specifications: Lenovo ThinkPad W530, Intel Core i7-3820QM CPU @ 2.70GHz, 12GB RAM

The ping between two peers, i.e., cloud instances, is approximately $0.9ms$[17] and the ping from the laptop to a peer in the cloud is around $21ms$[18].
The operating system of the laptop as well as the cloud instances is Ubuntu 12.04 (LTS). In order to modify the network conditions for the intra-PAN network properties evaluation (see Section 5.2.3.3) we leverage two tools, i.e., TC and Wondershaper.
Appendix F presents the configuration of the sensor simulator and PAN. In addition, it contains the client configuration we use for the visualization clients and the query delay client which measures the query delay during the evaluation. More precisely, it lists the values for all important configuration variables. The sensor simulator configuration is the same we use for evaluating the sensor simulation environment (see Section 5.1).
As we state in the previous section, the sensor simulation environment is not able to produce proper input data streams for evaluating PAN in the end of the first half time and during the half time break. Therefore, we simulate only the first 25 minutes[19] of the soccer match in the PAN evaluations. More precisely, we start the simulation at timestamp 10753295594424116 and stop it when the query delay client receives a tuple with a timestamp greater than or equal to 12253295594424116. As a consequence, PAN only receives and analyzes proper inter-PAN input streams during the evaluation. We argue that this is legal since in our opinion it is useful to evaluate PAN first under perfect conditions. Otherwise, the corrupt input data streams may corrupt the evaluation results of PAN and probably even conceal trends or conceptual problems.

5.2.2 Query Delay Metric

In the published ACM DEBS 2013 Grand Challenge solutions (see Chapter 2) the authors mainly use two metrics to evaluate their systems, i.e., the throughput and the query delay. The *throughput* is either measured in analyzed events per second or an event processing speedup value, i.e., how much faster than real-time the system is able to analyze the input stream and generate the output streams. We neglect evaluating the throughput since the current prototype implementation of PAN is only a proof of concept and thus not able to compete the published solutions in terms of throughput performance. In fact, as we will show in Section 5.2.3.2, we need to distribute the exemplary workflow which does not even answer all specified queries at least onto 6 peers. However, in contrast to the published solutions our system is able to scale further in terms of data.
The *query delay* denotes, how long the system needs to calculate and generate a certain statistic, i.e., a certain output stream. In other words, the query delay measures how long the system needs to answer a query. In our evaluation, we use this metric to measure PAN's performance. More precisely, we use this metric to measure how PAN's performance changes with its degree of distribution and with the internal network properties, i.e., to discover PAN's characteristics and limitations. Moreover, we leverage it to evaluate the load balancing feature.

[17] Intra-PAN ping statistics: rtt min/avg/max/mdex = 0.556/0.920/5.118/0.303 ms
[18] Inter-PAN ping statistics: rtt min/avg/max/mdex = 17.642/21.005/182.466/13.643 ms
[19] Simulation length: $12253295594424116ps - 10753295594424116ps = 1.5E15ps = 1500s = 25min$

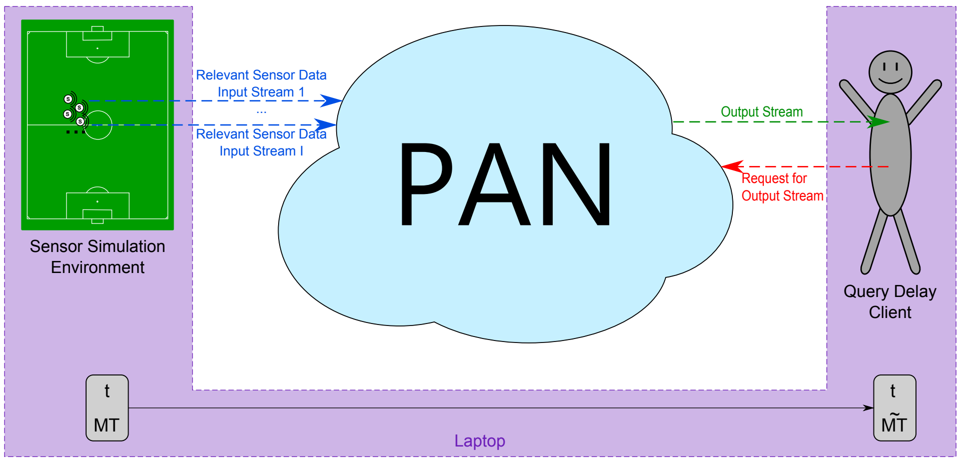

Relevant Sensor Data Input Stream 1
...
Relevant Sensor Data Input Stream l
Sensor Simulation Environment
PAN
Output Stream
Request for Output Stream
Query Delay Client
t
MT
t
$\tilde{MT}$
Laptop

Figure 5.2: Query Delay Metric Setup

Figure 5.2 illustrates the setup for measuring PAN's query delay. The query delay is calculated by means of the machine time (MT) when sending a sensor data tuple at a sensor simulator and the machine time ($\tilde{MT}$) when receiving the corresponding output data stream tuple at the client (cf. Equation 5.3). Hence, since different machines may have different machine timestamps at the same moment, all sensor simulators and the query delay client have to be executed on the same machine.

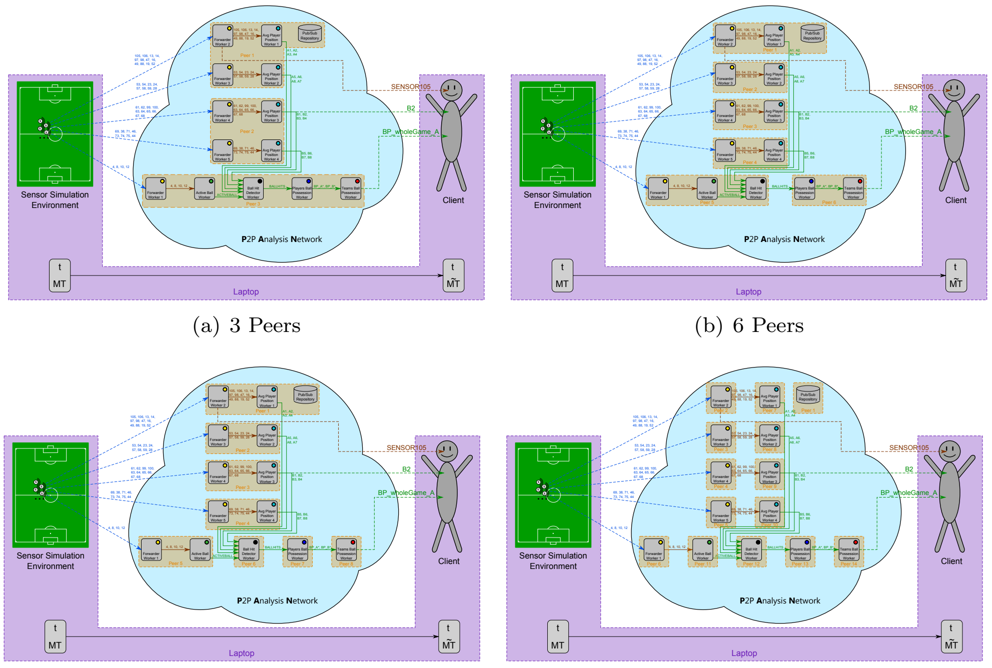
$$QueryDelay = \tilde{MT} - MT \tag{5.3}$$

Each data tuple has a unique match timestamp in picoseconds (t). We use this timestamp to identify the sensor data tuples. More precisely, a sensor simulator which generates a *relevant* sensor data stream logs the machine timestamp for each tuple by writing match-timestamp-machine-timestamp pairs in a dedicated file. Relevant are those sensors which may affect an output data stream for which the query delay is measured, i.e., whose match timestamp may be the resulting event timestamp of an output data stream tuple received at the query delay client at the other end of the workflow. Usually, the event timestamp, i.e., the match timestamp of an (intermediate) output stream, is the maximum timestamp of all input tuples which influence the output tuple. For instance, the event timestamp of the average player position stream can be the match timestamp of all sensors the player is equipped with (e.g., *SENSOR63* and *SENSOR64* for *B2*). Thus, these sensors are relevant for the average player position stream.

The *Query Delay Client* is a Java implementation which uses the Jetty HTTP client to periodically (with max. 50Hz[20]) fetch the latests tuples of all streams (for which one wants to measure the query delay) from PAN, i.e., from PAN workers publishing these streams. As at the sensor simulators, the query delay client logs a match-timestamp-machine-timestamp pair for each received data tuple in a dedicated log file per output stream.

At the end of the simulation, i.e., after 25 minutes, a small Java program (the *Query Delay Calculator*) calculates the query delay for each received tuple. For this purpose, it iterates

[20] The fetching thread sleeps $20ms$ after fetching tuples before it starts again.



(a) 3 Peers (b) 6 Peers

(c) 8 Peers (d) 14 Peers

Figure 5.3: Full Game Workflow with four different Degrees of Distribution. Larger graphs can be found in Appendix E.1.

through each output stream log file and searches for a matching pair in the sensor simulator logs relevant for this stream and stores the resulting query delay (i.e., the difference of the machine timestamps) in a query delay list. In doing so it skips duplicate tuples in the output stream logs, i.e., it calculates the minimal query delay only once per received tuple. Subsequently, we iterate through the query delay list and calculate common statistics as the average, variance, median and so on.
Please note, that our query delay metric measures not only the time PAN needs for analyzing the sensor data input streams and generating the output stream inside PAN but also the time for sending the input stream to the first PAN worker and fetching them from the last PAN worker.

5.2.3 Big Workflow Evaluations

In this section, we leverage the exemplary workflow used in Section 4.2 to explain the PAN concept with different degrees of distribution to evaluate PAN's performance characteristics, its requirements on the environment and its consistency.

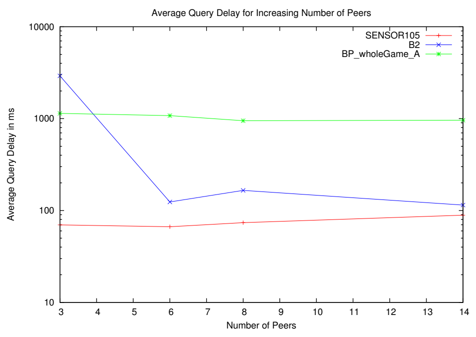
5.2.3.1 Workflow

Figure 5.3 illustrates the *Full Game* workflow with four different degrees of distribution. This workflow consumes all sensor data streams as inter-PAN input streams and generates ball possession streams for all players as well as both teams.
In order to evaluate PAN's performance, we periodically fetch three output streams at the

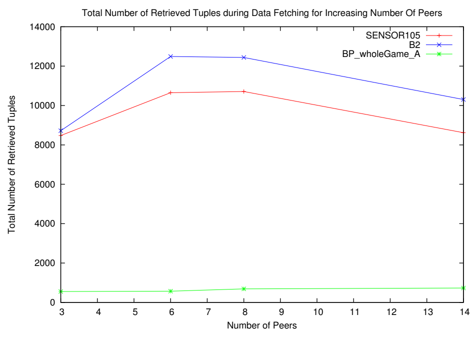
query delay client. Namely, we fetch the sensor data stream of the referee's left shin guard (*SENSOR105*), the average player position stream of player B2 (*B2*) and the whole game ball possession statistic stream for team A (*BP_wholeGame_A*). The reason for choosing these three streams is that they reflect three different kind of streams, i.e., a forwarded input stream, an intermediate output stream as well as a complete statistical output stream defined by the ACM DEBS 2013 Grand Challenge. Hence, they are also produced at different positions (i.e., after a different number of steps) in the intra-PAN workflow. This fact is particularly important for evaluating the influence of the intra-PAN network properties on PAN's performance characteristics.
Apart from the evaluation in which we want to explore the influence of the degree of distribution and find the perfect number of peers for this workflow (see Section 5.2.3.2), we will relinquish evaluating PAN with all four degrees of distribution. Instead, we will use only the 14 peers workflow as illustrated in Figure 5.3(d) in order to evaluate PAN with the maximum degree of distribution. We argue that this is reasonable, since above all PAN is designed to be scalable by distributing the workflow in a P2P network. Thus, a workflow which is distributed as much as possible is the best scenario for evaluating PAN.

5.2.3.2 Degree of Distribution

In the first evaluation row, we vary the degree of distributions, i.e., the number of peers on which the workflow is distributed. In this way, we want to explore the impact of the degree of distribution on PAN's performance. More precisely, we will show that increasing the number of peers can solve computational bottleneck problems. Moreover, we will discuss the perfect number of peers for distributing this workflow in this environment (i.e., on such peers or more precisely Windows Azure cloud instances) and explore the effect of distributing the workflow more than necessary.
In order to evaluate this, we leverage the four different workflows illustrated in Figure 5.3. In a nutshell, we perform a single run, i.e., simulate and analyze the first 25 minutes of the soccer match, for each of these four workflow distributions and measure the query delays during these runs.
Figure 5.4 illustrates the results of this evaluation. More precisely, it shows the average query delay in Figure 5.4(a) and the number of retrieved tuples at the query delay client in Figure 5.4(b). Further graphs as well as a table listing all measured statistics is given in Appendix G.2.1.
While the query delay of *SENSOR105* first decreases (from the 3 peers to the 6 peers setup) but then increases with the number of peers and has its maximum when distributed on 14 peers, the query delay of *BP_wholeGame_A* decreases up to the 8 peer distribution, increases a little in the 14 peer setting but has its maximum in the 3 peer distribution. However, we argue that these are only small fluctuations and the average query delays of the *SENSOR105* and the *BP_wholeGame_A* stream are rather constant in this evaluation row.
In contrast, the query delay of the *B2* stream has a huge value (2923.73ms) in the 3 peers setting which is approximately 20 times higher than the query delay in the remaining three distributions. In these distribution the query delay keeps comparatively constant (i.e., 123.99ms, 165.68ms and 114.72ms). The reason for the huge query delay in the 3 peers



(a) Average Query Delay



(b) Number of Retrieved Tuples

Figure 5.4: Statistics for *SENSOR105*, *B2* and *BP_wholeGame_A* in Increasing Number of Peers Evaluation. More statistics can be found in Appendix G.2.1.

setup is that in this setup *Peer 1* and *Peer 2* each have to generate the average player position streams for eight players. In all other distribution schemes, each peer hosts maximally a single *Average Player Position Worker* and thus only has to generate the average player position streams for four players. This confirms, that PAN is able to solve computational bottlenecks by distributing the workflow.

The number of retrieved tuples which is illustrated in Figure 5.4(b) denotes how many different tuples (i.e., without duplicates) of a certain stream the query delay client received during the evaluation run, i.e., how many query delays are measured. As one can see in this and all subsequent evaluation results, this number is one order of magnitude smaller for the *BP_wholeGame_A* stream as for the *SENSOR105* and *B2* stream. The reason for this, is that the query delay client ignores duplicates. A RedFIR transmitter measures its position, velocity and acceleration with 200Hz. Hence, a sensor data stream and thus *SENSOR105* has a new tuple every $5ms$. The same is true for *B2* since the *Player Average Component* fetches new sensor data tuples all $5ms$ and produces a tuple by averaging these tuples. Hence, for these two streams the fetch interval of the query delay client is the limiting factor. In contrast, the *BP_wholeGame_A* stream only has a new tuple if the *Ball Hit Detector Component* detects a new ball hit which is of course a more rarely event.

While the number of retrieved tuples increases monotonically with the number of peers in the *BP_wholeGame_A* case, there is a different trend for the *SENSOR105* and *B2* stream. For both streams, the number of retrieved tuples strongly increases when increasing the number of peers from 3 to 6 and strongly decreases when decreasing the number of peers from 8 to 14. This shows that increasing the number of peers can increase the number of retrieved tuples. However, this also indicates that increasing the number of peers and thus the degree of distribution can also harm PAN.

In conclusion, we argue that the perfect distribution for this workflow is distributing it on 6 peers as illustrated in Figure 5.3(b). As one can easily see, the 3 peers setup is no option since it is not sufficient for generating all average player positions streams. Regarding only the statistics, the 8 peer setup is also an option. However, we argue that there is no reason for using two additional peers if there is no benefit from using them. And last but not least, we recommend against using the 14 peers setup in practice, since again there is no benefit

```
sudo tc qdisc add dev eth0 root netem delay 10ms
```

Listing 5.1: Bash script for adding an additional delay of $10ms$ to the ethernet network adapter.

but even a disadvantage since the number of retrieved tuples decreases when using 14 instead of only 6 peers. Nevertheless, we use the 14 peers workflow as illustrated in Figure 5.3(d) for the subsequent evaluations for the reasons we have stated earlier.

5.2.3.3 Intra-PAN Network Properties

In the next two evaluation rows, we evaluate the impact of the intra-PAN network properties on PAN's performance, i.e., its query delay. In this way, we explore in which environments PAN can be deployed or more precisely PAN's requirements on the intra-PAN network properties.

In order to do so, we use the 14 peers *Full Game* workflow illustrated in Figure 5.3(d) and simulate worser network conditions for the intra-PAN network than we have in the Microsoft Azure Cloud. More precisely, in the first evaluation row we increase the latency of all peers and in the second evaluation row we limit their bandwidth.

Latency

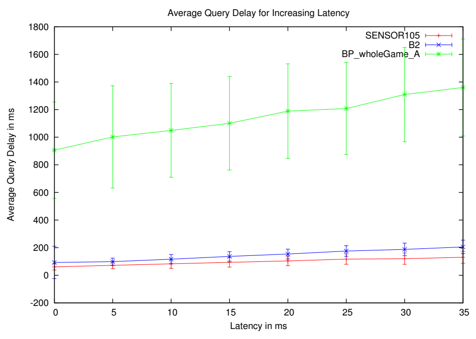
The purpose of the first network properties evaluation row is to evaluate how the peers hosting PAN workers can be distributed w.r.t. their locality. In other words, we explore if all peers have to be positioned in the same building (e.g., cloud compute center) or if they can be distributed in Switzerland, Europe or even on the whole world.

In our evaluation setup all peers are cloud instances in the same Microsoft Azure Cloud region (i.e., *Europe West*). Hence, we simulate the spatial distribution in this evaluation row by artificially increasing the latencies in the evaluation environment. More precisely, we add an additional delay to the ethernet network adapter of each peer by means of the TC tool (see Listing 5.1). Please note, that this additional delay is only added to network communications with other peers and not with the localhost.

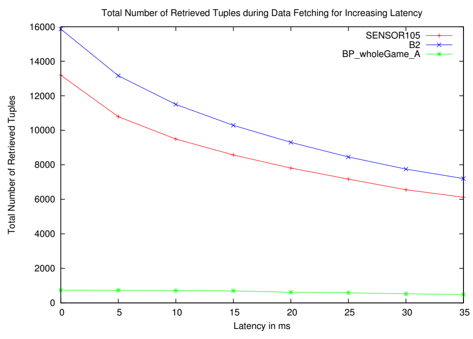
In the evaluation row, we start with an additional delay of $0ms$ (i.e., with the normal setup) and increase it by $5ms$ in each run up to a maximal additional delay of $35ms$. Unfortunately, we are not able to further increase the delay in this evaluation row, since doing so results in problems in our prototype implementation when using the 14 peers workflow. We suppose the reason for this is that the overall latency aggregates to much in the 14 peers workflow. Wo plan to further analyze and fix this problem in our future work.

Figure 5.5 shows the results of this evaluation row. As one can see in Figure 5.5(a) the average query delay increases linearly with the latency. The sole exception of this trend is the *B2* stream in the $5ms$ run. Moreover, the gradients of the curves[21] indicate the position of the PAN worker publishing the corresponding stream in the intra-PAN workflow. More precisely, the more intermediate steps are needed to generate the stream, the higher the

[21] *SENSOR105*: $\frac{130.42-60.74}{35} \approx 1.99$, *B2*: $\frac{205.39-92.47}{35} \approx 3.23$, *BP_wholeGame_A*: $\frac{1360.17-906.31}{35} \approx 12.97$



(a) Average Query Delay and Standard Deviation



(b) Number of Retrieved Tuples

Figure 5.5: Statistics for *SENSOR105*, *B2* and *BP_wholeGame_A* in Increasing Latency Evaluation. More statistics can be found in Appendix G.2.2.

gradient is. These observations exactly match our expectations.
As illustrated in Figure 5.5(b), the number of retrieved tuples decreases when increasing the latency. This general trend meets our expectation. However, the curves are not linear. Although, the decrease of the *BP_wholeGame_A* appears to be linear at first glance it is not (cf. Table G.2). Moreover, the curves of the *SENSOR105* and the *B2* stream are even reminiscent of an exponential decay. To confirm that the number of retrieved tuples decreases like an exponential decay when increasing the latency, we have to perform evaluations with higher latencies in order to explore if this trend continues. As mentioned earlier, this is planned for future work.
In conclusion, we have shown that extending the spatial distribution, i.e., increasing the latencies between the peers, as expected increases the query delay. Moreover, we have shown, that the query delay does not increase dramatical but linear with the latency and that the gradient of the increment depends on the position of the publisher.
In addition, this evaluation confirms that PAN can be deployed onto peers which are spatially distributed. The results show, that distributing the peers in Switzerland or even Europe indeed increases the query delay but is possible. However, at least when using the current prototype, a distribution around the whole world is not possible since the minimal latency (l_{min}) to a peer on the other side of the world is approximately $67ms$[22], but PAN can only cope with latencies up to $35ms$.

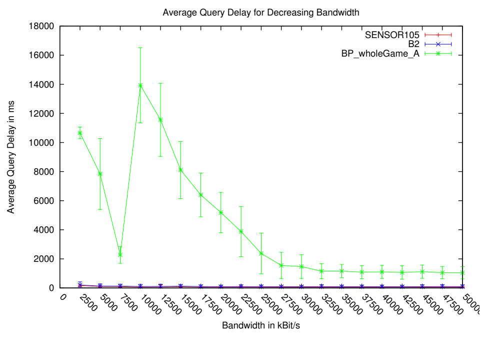
Bandwidth

In the second network properties evaluation row, we evaluate how PAN's performance changes if we limit the bandwidth of the peers hosting the PAN workers and the publish/subscribe repository. For doing so we shape the down- as well as the uplink of each peer using the Wondershaper tool (see Listing 5.2). More precisely, we start with shaping the bandwidth to $50000kb/s$ $(= 6250kB/s)$ and decrease the limit in each run by $2500kb/s$ down to $2500kb/s$ $(= 312.5kB/s)$ in the last run.
Figure 5.6 illustrates the results of this evaluation row, i.e., the average query delay and the

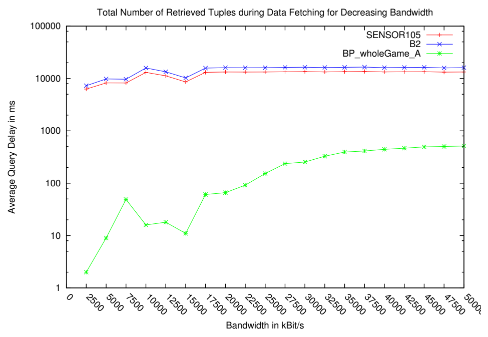
[22] $d_{min} = \frac{P_{earth}}{2} = \frac{40074km}{2} = 20037km \quad \rightarrow \quad l_{min} = \frac{d_{min}}{c} \approx 67ms$

```
sudo wondershaper eth0 40000kbps 40000kbps
```

Listing 5.2: Bash script for limiting the bandwidth (down- and uplink) of the ethernet network adapter to $40000 kb/s$.



(a) Average Query Delay and Standard Deviation



(b) Number of Retrieved Tuples

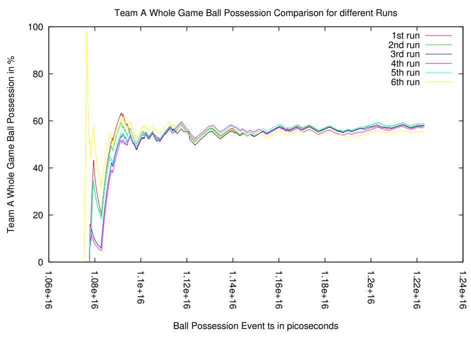
Figure 5.6: Statistics for *SENSOR105*, *B2* and *BP_wholeGame_A* in Decreasing Bandwidth Evaluation. More statistics can be found in Appendix G.2.3.

number of retrieved tuples.

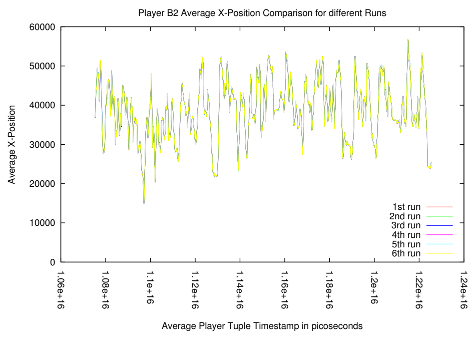
The statistics of the *SENSOR105* and the *B2* stream have exactly the same schema. As long as the bandwidth limit is greater than or equal to $17500 kb/s$ ($= 2187.5 kB/s$), the average query delay and the number of retrieved tuples is approximately constant. The query delay of *SENSOR105* and *B2* fluctuates around $61 ms$ and $90 ms$, respectively, and thus matches the values of the $0 ms$ run in the previous evaluation row. The same is true for the number of retrieved tuples. However, if the bandwidth limit falls below $17500 kb/s$, the statistics change. Unfortunately, the query delay is not inversely proportional and the number of retrieved tuples is not proportional with the bandwidth limit. Instead, the curves show anomalies for bandwidth limits below $17500 kb/s$. More precisely, the query delay increases when decreasing the bandwidth to $15000 kb/s$, decreases down to $10000 kb/s$ and then increases again until the bandwidth limit reaches its minimum. The same trend can be observed in the number of retrieved tuples curve.

The average query delay statistic of the *BP_wholeGame_A* stream shows a similar schema as those of the *SENSOR105* and the *B2* stream but with a different boundary. As long as the bandwidth limit is greater than or equal to $32500 kb/s$ ($= 4062.5 kB/s$), the average query delay keeps constant around $1100 ms$. However, this value does not match the value of the $0 ms$ run in the increasing latency evaluation row. When the bandwidth falls below $32500 kb/s$, the query delay starts increasing exponentially. In contrast to the average query delay, the number of retrieved tuples is not constant for bandwidth limits greater than or equal to $32500 kb/s$, but starts decreasing from begin on. However, the number of retrieved tuples decreases faster when the bandwidth limit is below $32500 kb/s$. We argue, that the values for bandwidths below $15000 kb/s$ are not meaningful and thus can be neglected, since in these runs the query delay client was only able to measure less than 20 query delays. The sole exception is the $7500 kb/s$ case but we suppose that this is only a single outlier.

Apart from some anomalies below the minimal required bandwidth, the observations we



(a) Percentage Component of *BP_wholeGame_A*



(b) X-Position Component of *B2*

Figure 5.7: Output Stream Comparison for six different Runs. More graphs can be found in Appendix G.2.4.

made for the *SENSOR105* and *B2* stream meet our expectations. As long as the bandwidth limitation is greater than or equal to the minimal required bandwidth, PAN's performance does not benefit from having more bandwidth at disposal. Instead, the average query delay and the number of retrieved tuples for the *SENSOR105* and *B2* stream are constant and match those of the $0ms$ run of the previous evaluation. However, if the available bandwidth is below the minimal required bandwidth, PAN's performance collapses. Investigating and solving the issue that the statistics of the *BP_wholeGame_A* stream does not match these schema in all points requires further evaluations and is postponed to our future work.

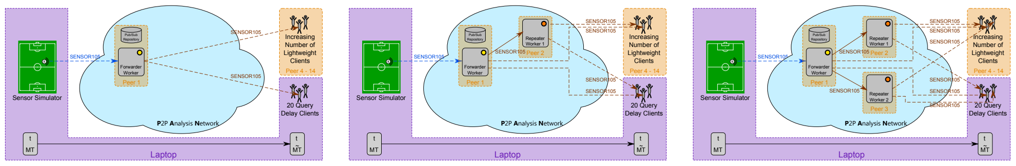
5.2.3.4 Consistency

In this evaluation row, we want to measure if our implementations of the DEBS specific internal components generate correct statistical data streams. More precisely, we want to evaluate if the results are consistent, i.e., if PAN produces the same or at least very similar streams in each run.

In order to evaluate that, we performed six runs without additional latency or bandwidth limitations. During these runs, the *Teams Ball Possession Component* logs the percentage value (and the corresponding event timestamp) of each produced *BP_wholeGame_A* tuple. The same is done for the position of the *B2* stream in the *Player Average Component*. Moreover, we log the player who hit the ball as well as the time difference between the active ball timestamp and the player timestamp in the *Ball Hit Detector Component* for each ball hit.

Figure 5.7 illustrates the results of this evaluation. More precisely, Figure 5.7(a) shows how the ball possession percentage of team A changes during the match. Figure 5.7(b) shows how player B2 moves along the x-axis by illustrating the x-position component of the *B2* stream.

As one can easily see, the team ball possession statistic is not consistent. In particular, in the beginning of the match (i.e., up to the match timestamp $1.1E16ps$) the values distinguish significantly. Although, the differences decline during the match, they persist during the end of the simulation. In contrast, the *Player Average Component* generates a consistent stream. The resulting tuples differ not or only slightly.



(a) 1 Forwarder (b) 1 Forwarder, 1 Repeater (c) 1 Forwarder, 2 Repeaters

Figure 5.8: Sensor Forwarding Workflows. Larger graphs can be found in Appendix E.2.

In consequence, we suppose that the reason for the inconsistencies in the team ball possession statistics are problems in the internal components and not general problems of the PAN approach. As the organizers of the ACM DEBS 2013 Grand Challenge state, a "reliable detection of a ball hit is difficult" [2]. Among other things, the ball hit detection is very time-critical. For instance, deferring the ball hit detection only a few milliseconds can result in identifying a different player as the nearest player and thus as the player who hit the ball. The same is true for intra-PAN time differences, i.e., time differences between the input streams of a PAN worker which are introduced by the intra-PAN workflow and have to be handled by the internal component itself if necessary (e.g., by means of buffering). The problems of the *Ball Hit Detector Component* are illustrated in Figure G.18 and G.19 in Appendix G.2.4. Since ball hits are comparatively rare events and the ball possession streams base on the *BALLHITS* stream, missing or falsely detecting a single ball hit can harm the whole team ball possession statistics. None more so than the beginning of the match. This explains the inconsistencies in Figure 5.7(a). Hence, we argue that one has to improve the DEBS specific internal components instead of the general PAN approach to eliminate the observed inconsistencies.

5.2.4 Stream Repeaters and Load Balancing

This section presents the evaluation of PAN's load balancing feature. The workflow we use in this evaluation as well as the way we changed the evaluation setup presented in Section 5.2.1 in order to obtain an increasing number of client requests will be presented in Section 5.2.4.1. Subsequently, Section 5.2.4.2 presents and discusses the results.

5.2.4.1 Workflow and Setup Modification

The purpose of this evaluation is to evaluate how PAN scales w.r.t. an increasing number of client requests and how its scalability can be improved by means of the load balancing feature, i.e., by means of distributing the client requests onto multiple publishers. Hence, in order to evaluate the load balancing feature we require a workflow in which at least a single stream has multiple publishers.

Figure 5.8 illustrates the workflows we use for evaluating the load balancing feature. In contrast to the *Full Game* workflow, which generates multiple (intermediate) statistical output streams and encompasses several different workers distributed onto multiple peers, these workflows are designed to be as simple as possible. All workflows have in common that they only receive a single inter-PAN input stream (*SENSOR105*) and publish this stream to clients outside PAN. The point in which they differ is the number of publishers of this stream. While there is only a single publisher (i.e., the *Forwarder Worker*) in the single peer workflow illustrated in Figure 5.8(a), the two peers workflow (see Figure 5.8(b)) has an additional *Repeater Worker* which repeats the sensor data stream and thus is an additional publisher of this stream. The three peers workflow contains two *Repeater Workers* and thus provides even three publishers for the sensor data stream. Hence, the load of the client requests is distributed onto one, two or even three peers.

We argue, that it is not only legal but even reasonable to leverage new workflows instead of enriching the *Full Game* workflow with additional publishers. First, in a more complex workflow the probability that side effects corrupt the evaluation results and thus maybe even conceal the real trends increases. Moreover, we need the remaining cloud instances to host the huge number of clients we require for this evaluation.

In the evaluation setup we have presented in Section 5.2.1 and used for all previous evaluations there is only a single query delay client executed on the laptop which purpose is to measure the query delay and thus PAN's performance. We modify this setup by increasing the number of query delay clients which are executed on the laptop to 20. This number is fixed during the following evaluation rows. The query delay statistics are measured by all query delay clients and averaged to obtain a single set of statistical values (i.e., average, median, etc.). Moreover, we host lightweight Python clients whose only task is to subscribe the *SENSOR105* stream in the beginning and fetch the latest data tuple from the retrieved publisher all $20ms$. For doing so the lightweight Python client leverages the urllib2 library[23]. The number of these clients is not fixed but increased during each evaluation row (from 20 to 80). Since the laptop is not able to execute all these clients, the lightweight python clients are equally distributed onto 10 cloud instances which are not in use for hosting PAN workers.

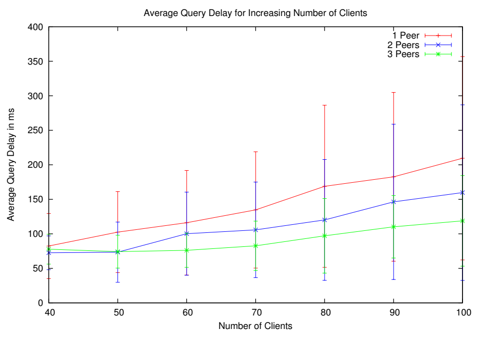
5.2.4.2 Results

In order to evaluate how PAN scales w.r.t. the number of client requests and if the load balancing feature works as expected we perform three evaluation rows, i.e., one evaluation row for each workflow illustrated in Figure 5.8. In each of these evaluation rows, we increase the number of client requests from 40 to 100. More precisely, we increase the number of lightweight Python clients from 20 to 80 by 10 in each run. Hence, overall we performed 21 runs for evaluating the load balancing feature.

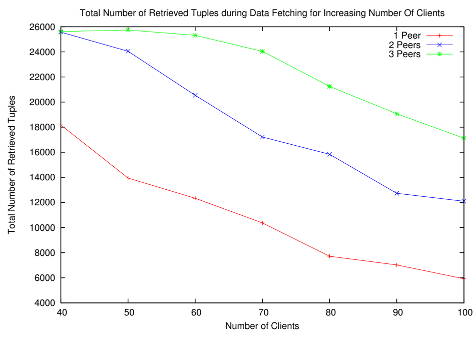
Figure 5.9 illustrates the results for all 3 evaluation rows. Each evaluation row is illustrated as a curve in the graph. If one concentrates on a single evaluation row, one can observe how PAN's performance scales w.r.t. the number of client requests for a constant number of publishers. In a nutshell, the query delay increases and the number of retrieved tuples decreases with the increasing number of client requests. Thus, as we expected, PAN's performance decreases if the load introduced by the client requests increases.

In order to evaluate the load balancing feature, one has to compare the different evaluation rows. As one can see, the more peers publish a stream, the better is the performance. More precisely, the average query delay is always the smallest in the 3 peers workflow and the

[23] urllib2: https://docs.python.org/2/library/urllib2.html (07.08.2014)



(a) Average Query Delay and Standard Deviation



(b) Number of Retrieved Tuples

Figure 5.9: Statistics for *SENSOR105* in Increasing Number of Clients Evaluation. More statistics can be found in Appendix G.2.5.

highest in the 1 peer workflow. Moreover, the number of retrieved tuples is higher the more peers publish the stream. The sole exception of this trend is the 50 clients case in which the average query delay in the 3 peers workflow exceeds the average query delay of the 2 peers workflow. In addition, the gradients with which the query delay increases appeals to be the smaller the more peers publish the stream. In order to make sure that this is true, we plan to perform evaluations with a larger extend (i.e., with more clients) in our future work. But, notwithstanding the above, the evaluation results confirm that PAN's load balancing feature works.

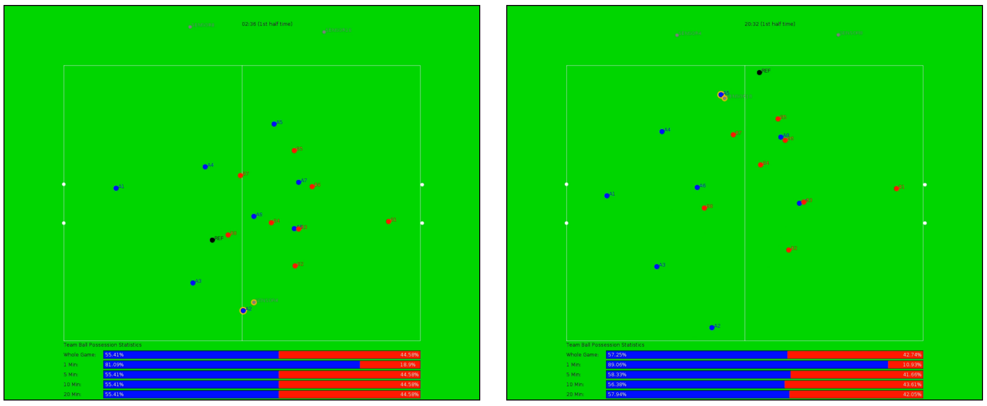
5.2.5 Visualization

In order to answer the last remaining question, i.e., to evaluate that PAN is able to generate not only consistent but also correct statistical data streams for the queries specified in the ACM DEBS 2013 Grand Challenge in real-time, we leverage a visualization. More precisely, we visualize some (intermediate) output streams in Java Processing clients. These clients use the same code to fetch data tuples as the query delay client does. However, instead of logging the arrival time they visualize the result by means of the Processing library[24]. Appendix F.3 lists the values of the most important configuration parameters we used during the evaluation.

5.2.5.1 Full Game

The first visualization client visualizes all players (i.e., their average positions) and balls. Two orange circles highlight the active ball as well the player who actually is in possession of the ball, i.e., the last player who hits the ball. Moreover, bar diagrams at the bottom visualize the current team ball possession statistics for all time windows.
Figure 5.10 shows two screenshots of the client. A video of the whole simulation is available on Vimeo[25]. As one can see in the video, PAN is able to analyze the match in real-time.

[24] Processing library: https://www.processing.org/ (07.08.2014)
[25] Full Game Visualization on Vimeo: https://vimeo.com/album/2972208/video/102009212 (07.08.2014)



Team Ball Possession Statistics

Whole Game:	55.41%	44.58%
1 Min:	81.09%	18.9%
5 Min:	55.41%	44.58%
10 Min:	55.41%	44.58%
20 Min:	55.41%	44.58%

(a) 02:36

Team Ball Possession Statistics

Whole Game:	57.25%	42.74%
1 Min:	89.06%	10.93%
5 Min:	58.33%	41.66%
10 Min:	56.38%	43.61%
20 Min:	57.94%	42.05%

(b) 20:32

Figure 5.10: Full Game Workflow Visualization. Visualizes all players, the active ball and the team ball possession statistics for both teams and all time windows.

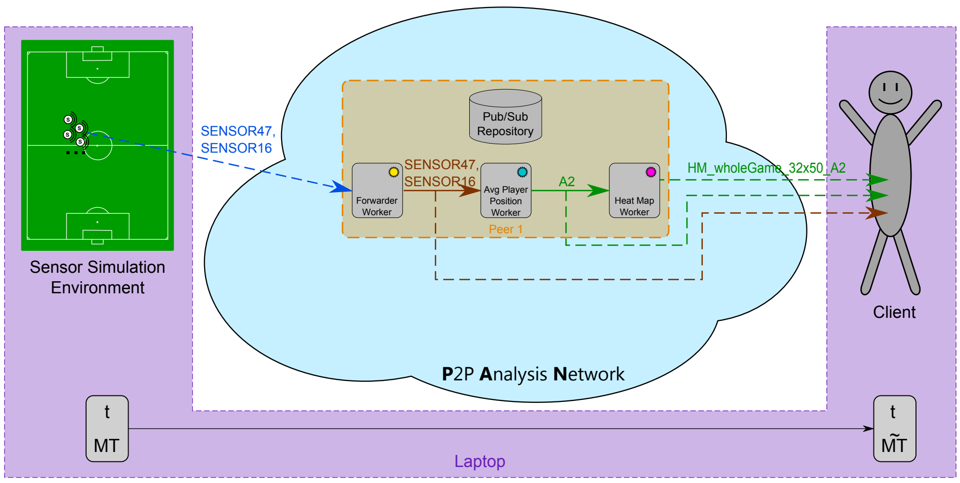


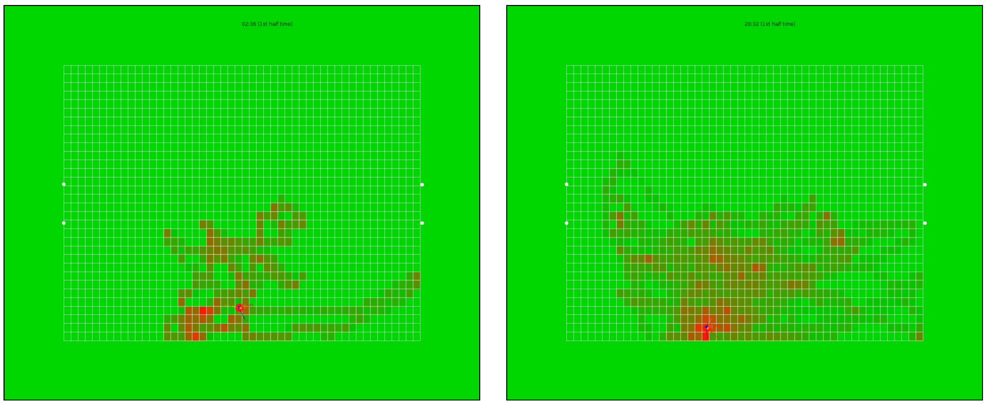
Figure 5.11: Heat Map Workflow for Player A2

That means, it is able to detect the active ball as well as the most ball hits and generate proper team ball possession statistics for different time windows.

5.2.5.2 Heat Map

The second visualization client visualizes the heat map of player A2. More precisely, it visualizes the *HM_wholeGame_32x50_A2* stream, i.e., the heat map of player A2 with the 32×50 resolution for the whole game time window. In order to produce this heat map, we leverage the workflow illustrated in Figure 5.11.
Again, two screenshots of the client are shown in Figure 5.12 and a video of the whole simulation is available on Vimeo[26]. This video confirms, that PAN is able to generate the heat map for a single player on a single peer in real-time.

[26] Heat Map Visualization on Vimeo: https://vimeo.com/album/2972208/video/102604325 (07.08.2014)



(a) 02:36 (b) 20:32

Figure 5.12: Heat Map Workflow Visualization. Visualizes the average position (*A2*) as well as the whole game 32 × 50 heat map (*BP_wholeGame_32x50_A2*) of player A2.

5.2.6 Discussion

Overall, we performed 59 evaluation runs. That means, we have simulated and analyzed the first 25 minutes of the soccer match 59 times under various conditions, in order to evaluate PAN. Using the (query delay) statistics we captured during these runs, we were able to make several observations regarding PAN's applicability and performance characteristics.

First, the evaluation results show that PAN is able to eliminate computational bottlenecks by distributing the workflow onto several peers in a P2P network which are connected by means of a pull-based publish/subscribe system. This is exactly what PAN is designed for. However, the same evaluation row also indicates that increasing the degree of distribution more than necessary can also harm PAN or more precisely its performance. We found out that the perfect distribution of the exemplary workflow (in the cloud environment) is the six peers setup.

Moreover, the evaluation confirms that PAN can be deployed onto peers which are spatially distributed and thus not positioned in the same building (e.g., cloud computing center). We have observed, that extending the spatial distribution (i.e., increasing the latencies between the peers) as expected increases the query delays. However, the average query delay increases not dramatically but linearly with the latency. Furthermore, the gradient with which a query delay increases indicates the position of the publisher in the intra-PAN workflow. As a result, we argue that the workflow can be performed on peers which are spatially distributed in Switzerland or even Europe. Merely distributing the workers onto peers positions in the whole world is not possible when using our current prototype implementation.

Apart from some anomalies, the results of the bandwidth evaluation row meet our expectations. As long as the available bandwidth is greater that or equal to the minimal required bandwidth, PAN's performance does not benefit from having more bandwidth at disposal. However, if the available bandwidth is below the minimal required bandwidth, PAN's performance collapses.

Both visualizations confirm that PAN is able to analyze the soccer match and generate correct statistical output streams for the queries specified in the ACM DEBS 2013 Grand

Challenge in real-time. We further suggest, that the reason for the inconsistencies we have observed in the time-critical streams (e.g., *BP_wholeGame_A*) are not general problems of the PAN approach but problems of the current implementation of some DEBS specific internal components.
In addition, the evaluation results show that, as we expected, PAN's performance decreases if the load introduced by the client requests increases. Hence, PAN is not able to handle an arbitrarily large number of client requests for a certain stream if there is only a single publisher for this stream. However, the evaluation results confirm that PAN's load balancing feature works and thus can be used for solving this problem. Hence, PAN's performance can benefit from its pull-based approach. More precisely, the evaluation confirms that the more peers publish a stream (i.e., the more the load is balanced), the better is the performance. Based on the observations we made, we even suggest that the gradient with which the performance decreases when the number of client requests increases is the smaller the more peers publish the requested stream. We plan to repeat this evaluation row with a larger extend (i.e., with more clients) in order to verify this suggestion.
As we mentioned in Section 5.2.2, the current PAN prototype cannot compete the published grand challenge solutions (see Chapter 2) in terms of throughput or query delay. In consequence, we cannot fulfill Jergler's prediction[27], that the throughput of its architecture could be improved by distributing the workflow.
However, implementing a distributed real-time complex event detection system which outperforms other CEP engines is not our target. Instead, the purpose of this thesis is to proof if the PAN approach works, i.e., if it is possible to distribute the workflow onto multiple peers connected with a *pull-based* approach. Thus, this thesis is a proof of concept.
Moreover, the main focus of the PAN approach (i.e., of using a pull-based system) is not its performance on relatively small workflows, but its scalability and especially its flexibility (during runtime). We argue that the evaluation results confirm PAN's scalability and thus that the first goal could be achieved. Implementing observer systems which utilize PAN's flexibility, i.e., add new repeaters or redistribute the whole workflow during runtime, and evaluating PAN's flexibility by means of these systems is out of the scope of this thesis but planned in our future work.

[27] Quotation: "Although, a distributed publish/subscribe based system would probably provide a higher througput, it may increase the latency at the same time." [6]

6 Related Work

The general idea to distribute a workflow, i.e., its workers, in a P2P network and combine the workers by means of a publish/subscribe system is not novel. Already in the beginning of the 21st century the OSIRIS approach [14] was developed. OSIRIS is a "scalable P2P process management system" [14]. In a nutshell, it executes the steps of a static (predefined) workflow at several distributed service providers. For this purpose, OSIRIS leverages global repositories. OSIRIS-SE [15] extends the OSIRIS approach by enabling it to handle streams. In consequence, OSIRIS-SE is a distributed CEP (Complex Event Processing) system. The authors demonstrated this with a health monitoring application. Besides scalability, OSIRIS(-SE)'s main focus is reliability and fault-tolerance. However, OSIRIS-SE leverages the common *push-based* approach, i.e., a publisher is responsible for disseminating its output stream. As a result, CEP systems based on OSIRIS may achieve a higher throughput than PAN but are less flexible due to the reasons we mentioned earlier in this thesis.

In the last ten years, many research regarding distributed CEP systems was done and published especially in the context of the annual ACM DEBS (Distributed Event-Based Systems) Conference[28]. However, to the best of our knowledge, PAN is the first system which uses a *pull-based* instead of the common push-based approach to distribute the workload of a workflow-based CEP system in a P2P network.

As PAN does in our evaluation scenario also [23] addresses the problem of how to detect events in distributed sensor data streams. However, in contrast to PAN, [23] is specialized on sensor networks. Its main target is to reduce the traffic in the sensor network. For this purpose, the sensors only forward the data tuples which are necessary for answering any user query (i.e., subscription).

Curracurrong [24] is another CEP system for detecting events in sensor networks. But, in contrast to [23], it focuses on the energy efficiency. More precisely, Curracurrong encompasses a query language which tries to find a "good trade-off between productivity, flexibility, and energy efficiency" [24] as well an "energy-efficient operator placement" [24] heuristic. Curracurrong Cloud [25] extends the applicability of the Curracurrong approach towards

[28] ACM DEBS Conferences: http://dl.acm.org/event.cfm?id=RE268 (07.08.2014)

detecting events in cloud environments (e.g., monitoring the load of the cloud instances) instead of only in wireless sensor networks (WSNs). Although Curracurrong's *sense* operator is time-triggered, in contrast to PAN, Curracurrong is no pull-based but a common push-based approach. This is due to the fact, that the sense operator is performed on the sensor (or the cloud instance) itself and thus does not fetch the data via network communication. Moreover, all other Curracurrong operators are event-triggered.

SCTXPF [26] is a platform for distributed complex event detection. As PAN, SCTXPF is generic, scalable and able to analyze multiple distributed input data streams. However, in contrast to PAN, SCTXPF's focus is on high throughput instead of flexibility during runtime. SCTXPF tries to distribute the CEP rules to the available event processors (EPs) (i.e., workers) as optimal as possible. Hence, SCTXPF's linchpin is its "CEP rule allocation algorithm" [26]. This algorithm, distributes the CEP rules onto the EPs in a way that those "CEP rules that shared the same state information were allocated to the same EPs" [26] while at the same time balances the load between the EPs.

Another distributed CEP system is DHEP [27]. The authors of DHEP argue, that state of the art distribute CEP system are not used in industry since they do not provide all features of centralized CEP systems (e.g., "user friendly interfaces" [27]). The DHEP approach solves this problem by connecting existing centralized CEP systems to a distributed CEP system with all features of the centralized CEP systems. Hence, DHEP's main focus is "supporting interoperability between heterogeneous event processing systems" [27]. In order to achieve this, DHEP introduces a "powerful object oriented definition language, that enables efficient, tool-aided designing of big industrial CEP applications" [27]. In contrast, PAN combines small workers to workflows and supports heterogeneity by means of REST-Interfaces. However, both approaches have in common that they are flexible during runtime. As PAN for instance enables adding new repeater workers (on new peers) or removing old ones during runtime, DHEP enables adding or removing centralized CEP systems.

Moreover, there are several approaches handling the problems of distributed CEP systems which are introduced by moving mobile users.

[28] focuses on moving range queries. Range queries are queries which "return data relative to a consumer-specified spatial range" [28]. Thus, answering moving range queries requires a continuous data stream with range query results for changing locations. The authors of [28] argue, that existing CEP systems cannot answer such queries without massive useless computation overhead since these systems have to place a "set of CEP operators for each potential range of interest" [28]. The authors fix this issue by means of "dynamic reconfiguration of CEP operators“ [28] and computing only those range queries which are requested by any consumer.

As stated in [29, 30], CEP operators have to be placed near to the user (i.e., "at the edge of the network" [30]) in order to achieve a good performance (i.e., low latency). MigCEP [29] faces the problem, that those CEP operators have to be migrated to new locations if the user moves. The authors argue, that "each migration comes with a cost beacause operators are associated with local states" [29]. To minimize those costs, the authors propose a migration algorithm which leverages the migration plan concept. That is, the costs are estimated and the best migration target is determined (by means of "predicted mobility patterns"

[29]) beforehand in order to avoid unnecessary migration costs. RECEP [30] face the issue, that "supporting a large number of consumers in a dynamic environment" [30] requires a huge amount of resources. However, the authors state that especially at the "edge of the network" [30] the resources are limited. RECEP solves this problem, by means of "reusing computations and streams between operators" [30] and tolerating little errors.
In its current version, PAN does not solve these mobile user problems. However, we plan to face these issues in our future work. Moreover, we suppose, that the flexibility we gain by using the pull-based instead of the push-based approach is beneficial regarding these problems.

7 Conclusion

In this thesis, we have shown that it is possible to construct a scalable and flexible real-time complex event detection system by distributing the workload onto multiple workers hosted on peers in a P2P network and combining these workers to a workflow by means of a *pull-based* instead of the common *push-based* publish/subscribe approach.

Therefor, we have developed and implemented the PAN approach. PAN is based on the workflow-based architecture idea proposed by Jergler et. al. [6]. Jergler et. al. split the overall workload into subtasks which are performed by different workers (called *task elements*). These workers are connected by means of ring buffers to a workflow. However, all workers are executed on the same machine. Hence, Jergler's architecture is not scalable. In order to change this, we followed Jergler's suggestion and distributed the workflow by means of a publish/subscribe system. Thereby, we transformed Jergler's architecture idea into a scalable solution.

The general idea to distribute a workflow onto several machines by means of a publish/subscribe system is not novel. However, to the best of our knowledge, PAN is the first system which uses a pull-based publish/subscribe approach instead of the common push-based approach to distribute the workload of a CEP system. As a result, in PAN, not the publisher of a certain stream is responsible for disseminating new tuples to all subscribers but each subscriber is responsible for fetching the tuples from the publisher. Hence, the workflow definition direction changes. This enables the dynamic extension of the workflow at runtime, i.e., adding repeaters for load balancing or new clients as sinks, without changing anything in the existing workflow. In consequence, PAN is not only scalable in terms of data but also w.r.t. the number of client requests.

Evaluations with the extended ACM DEBS 2013 Grand Challenge scenario show that PAN is able to analyze the input streams of and generate correct statistical output streams for the captured soccer match in real-time. Moreover, they confirm that PAN is able to eliminate computational bootlenecks by distributing the workflow on more machines and that its load balancing feature enables PAN to scale w.r.t the number of client requests. In addition, the evaluations give some indications about PAN's requirements on the environment, i.e., show that the PAN workers can be geographically distributed in Europe.

Bibliography

[1] Amazon Kinesis. http://aws.amazon.com/de/kinesis/. Last accessed: 07.08.2014.

[2] Mutschler, C., Ziekow, H., and Jerzak, Z. The DEBS 2013 Grand Challenge. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 289–294. ACM, Arlington, Texas, USA (2013).

[3] ACM DEBS 2013 Grand Challenge description. http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails. Last accessed: 07.08.2014.

[4] Jacobsen, H.-A., Mokhtarian, K., Rabl, T., Sadoghi, M., Sherafat Kazemzadeh, R., Yoon, Y., and Zhang, K. Grand Challenge: The Bluebay Soccer Monitoring Engine. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 295–300. ACM, Arlington, Texas, USA (2013).

[5] Wu, Y., Maier, D., and Tan, K.-L. 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. ACM, Arlington, Texas, USA (2013).

[6] Jergler, M., Doblander, C., Najafi, M., and Jacobsen, H.-A. 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. ACM, Arlington, Texas, USA (2013).

[7] Madsen, K. G. S., Su, L., and Zhou, Y. 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. ACM, Arlington, Texas, USA (2013).

[8] Gal, A., Keren, S., Sondak, M., Weidlich, M., Blom, H., and Bockermann, C. Grand Challenge: The TechniBall System. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 319–324. ACM, Arlington, Texas, USA (2013).

[9] Badiozamany, S., Melander, L., Truong, T., Cheng, X., and Risch, T. 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. ACM, Arlington, Texas, USA (2013).

[10] LMAX Disruptor Library. https://github.com/LMAX-Exchange. Last accessed: 07.08.2014.

[11] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565 (1978).

[12] Mattern, F. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 120–131. Chateau de Bonas, France (1988).

[13] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, Hollywood, California, USA (2012).

[14] Christoph Schuler, Roger Weber, Heiko Schuldt, Hans-J. Schek. Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04. IEEE, San Diego, California, USA (2004).

[15] Brettlecker, G. and Schuldt, H. The OSIRIS-SE (stream-enabled) infrastructure for reliable data stream management on mobile devices. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1097–1099. ACM, Beijing, China (2007).

[16] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-tp-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160. ACM, San Diego, California, USA (2001).

[17] Maymounkov, P. and Mazières, D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In Druschel, P., Kaashoek, F., and Rowstron, A., editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin Heidelberg (2002).

[18] Thompson, M., Farley, D., Barker, M., Gee, P., and Steward, A. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical Paper (2011). URL http://disruptor.googlecode.com/files/Disruptor-1.0.pdf.

[19] Jetty - Servlet Engine and Http Server. http://www.eclipse.org/jetty/. Last accessed: 07.08.2014.

[20] Google Gson. https://code.google.com/p/google-gson/. Last accessed: 07.08.2014.

[21] Apache Log4j. http://logging.apache.org/log4j/2.x/. Last accessed: 07.08.2014.

[22] Simple Logging Facade for Java (SLF4J). http://www.slf4j.org/. Last accessed: 07.08.2014.

[23] Jurca, O., Michel, S., Herrmann, A., and Aberer, K. Processing Publish/Subscribe Queries over Distributed Data Streams. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 21:1–21:4. ACM, Nashville, Tennessee, USA (2009).

[24] Kakkad, V., Attar, S., Santosa, A. E., Fekete, A., and Scholz, B. Curracurrong: a stream programming environment for wireless sensor networks. *Software: Practice and Experience*, 44(2):175–199 (2014).

[25] Kakkad, V., Dey, A., Fekete, A., and Scholz, B. Curracurrong cloud: Stream processing in the cloud. In *Data Engineering Workshops (ICDEW), 30th International Conference on Data Engeneering*, ICDE '14, pages 207–214. IEEE, Chicharo, Illinois, USA (2014).

[26] Isoyama, K., Kobayashi, Y., Sato, T., Kida, K., Yoshida, M., and Tagato, H. A scalable complex event processing system and evaluations of its performance. In *Proceedings of the 6th ACM International Conference on Distributed Event-based Systems*, DEBS '12, pages 123–126. ACM, Berlin, Germany (2012).

[27] Schilling, B., Koldehofe, B., Pletat, U., and Rothermel, K. Distributed Heterogeneous Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-based Systems*, DEBS '12, pages 150–159. ACM, Berlin, Germany (2012).

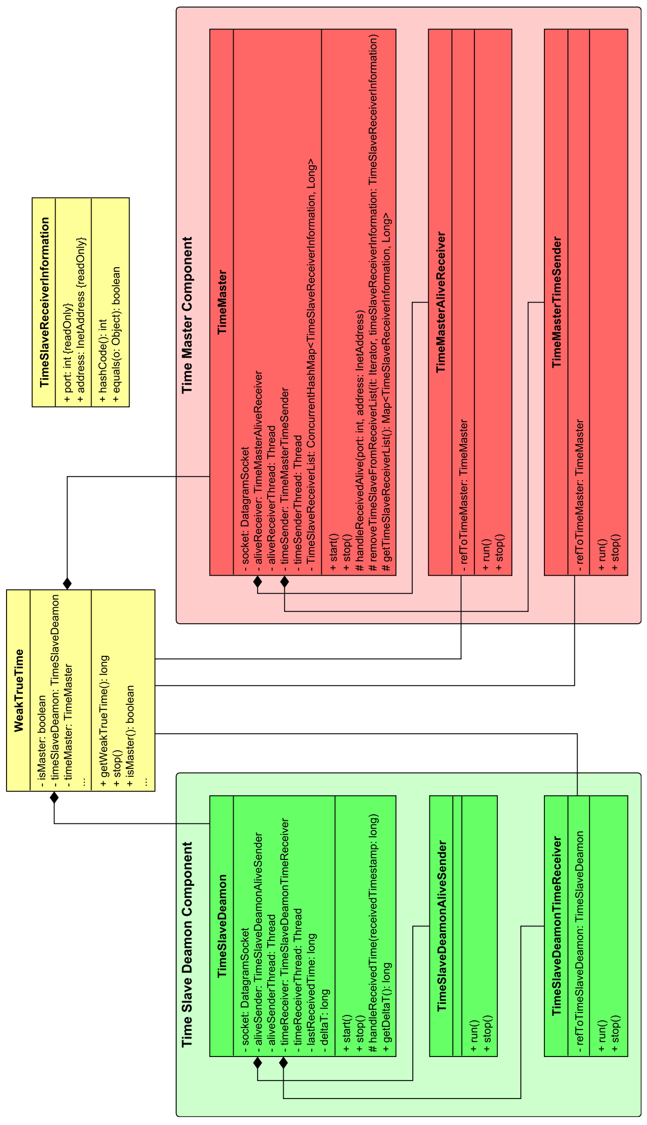
[28] Koldehofe, B., Ottenwälder, B., Rothermel, K., and Ramachandran, U. Moving Range Queries in Distributed Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 201–212. ACM, Berlin, Germany (2012).

[29] Ottenwälder, B., Koldehofe, B., Rothermel, K., and Ramachandran, U. 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. ACM, Arlington, Texas, USA (2013).

[30] Ottenwälder, B., Koldehofe, B., Rothermel, K., Hong, K., and Ramachandran, U. RECEP: Selection-based Reuse for Distributed Complex Event Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 59–70. ACM, Mumbai, India (2014).

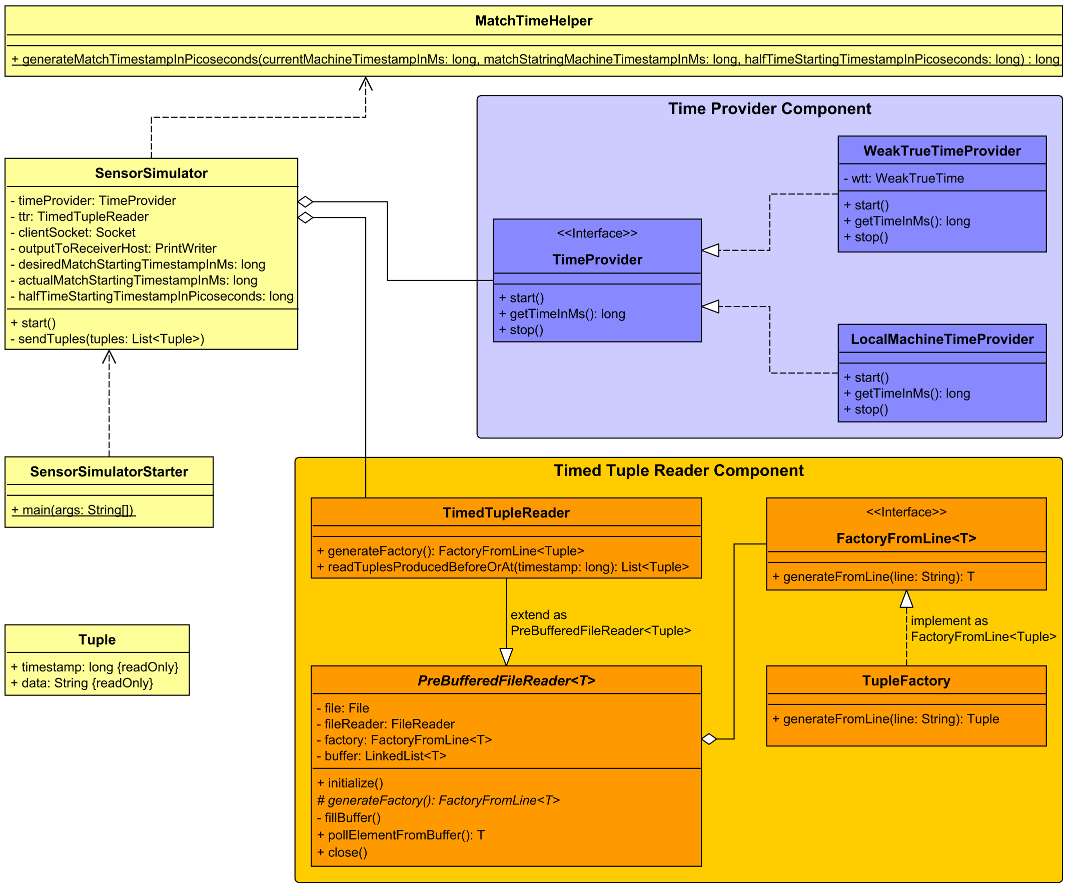
A
Class Diagrams

A.1 WeakTrueTime Class Diagram



TimeSlaveReceiverInformation
+ port: int {readOnly}
+ address: InetAddress {readOnly}
+ hashCode(): int
+ equals(o: Object): boolean
Time Master Component
TimeMaster
- socket: DatagramSocket
- aliveReceiver: TimeMasterAliveReceiver
- aliveReceiverThread: Thread
- timeSender: TimeMasterTimeSender
- timeSenderThread: Thread
- TimeSlaveReceiverList: ConcurrentHashMap<TimeSlaveReceiverInformation, Long>
+ start()
+ stop()
# handleReceivedAlive(port: int, address: InetAddress)
# removeTimeSlaveFromReceiverList(it: Iterator, timeSlaveReceiverInformation: TimeSlaveReceiverInformation)
# getTimeSlaveReceiverList(): Map<TimeSlaveReceiverInformation, Long>
TimeMasterAliveReceiver
- refToTimeMaster: TimeMaster
+ run()
+ stop()
TimeMasterTimeSender
- refToTimeMaster: TimeMaster
+ run()
+ stop()
WeakTrueTime
- isMaster: boolean
- timeSlaveDeamon: TimeSlaveDeamon
- timeMaster: TimeMaster
...
+ getWeakTrueTime(): long
+ stop()
+ isMaster(): boolean
...
Time Slave Deamon Component
TimeSlaveDeamon
- socket: DatagramSocket
- aliveSender: TimeSlaveDeamonAliveSender
- aliveSenderThread: Thread
- timeReceiver: TimeSlaveDeamonTimeReceiver
- timeReceiverThread: Thread
- lastReceivedTime: long
- deltaT: long
+ start()
+ stop()
# handleReceivedTime(receivedTimestamp: long)
+ getDeltaT(): long
TimeSlaveDeamonAliveSender
+ run()
+ stop()
TimeSlaveDeamonTimeReceiver
- refToTimeSlaveDeamon: TimeSlaveDeamon
+ run()
+ stop()

A.2 Sensor Simulator Class Diagram


MatchTimeHelper
+ generateMatchTimestampInPicoseconds(currentMachineTimestampInMs: long, matchStatringMachineTimestampInMs: long, halfTimeStartingTimestampInPicoseconds: long) : long
SensorSimulator
- timeProvider: TimeProvider
- ttr: TimedTupleReader
- clientSocket: Socket
- outputToReceiverHost: PrintWriter
- desiredMatchStartingTimestampInMs: long
- actualMatchStartingTimestampInMs: long
- halfTimeStartingTimestampInPicoseconds: long
+ start()
- sendTuples(tuples: List<Tuple>)
SensorSimulatorStarter
+ main(args: String[])
Tuple
+ timestamp: long {readOnly}
+ data: String {readOnly}
Time Provider Component
<<Interface>>
TimeProvider
+ start()
+ getTimeInMs(): long
+ stop()
WeakTrueTimeProvider
- wtt: WeakTrueTime
+ start()
+ getTimeInMs(): long
+ stop()
LocalMachineTimeProvider
+ start()
+ getTimeInMs(): long
+ stop()
Timed Tuple Reader Component
TimedTupleReader
+ generateFactory(): FactoryFromLine<Tuple>
+ readTuplesProducedBeforeOrAt(timestamp: long): List<Tuple>
extend as
PreBufferedFileReader<Tuple>
PreBufferedFileReader<T>
- file: File
- fileReader: FileReader
- factory: FactoryFromLine<T>
- buffer: LinkedList<T>
+ initialize()
# generateFactory(): FactoryFromLine<T>
- fillBuffer()
+ pollElementFromBuffer(): T
+ close()
<<Interface>>
FactoryFromLine<T>
+ generateFromLine(line: String): T
implement as
FactoryFromLine<Tuple>
TupleFactory
+ generateFromLine(line: String): Tuple

B
Sensor Simulator Parameters

Parameter name	Description
filename	The name/path of the sensor data file of the sensor which has to be simulated.
receiverHostName	The hostname or IP address of the sensor data stream receiver.
receiverHostPort	The port of the sensor data stream receiver.
startingTimestampInMs	The timestamp at which the sensor simulator should start generating the sensor data stream.
isWTTMaster	Defines if the sensor simulator should create a time master or a timeslave deamon WeakTrueTime instance.
myWTTPort	The local WeakTrueTime port.
masterWTTHostName	The time master's WeakTrueTime hostname or IP address.
masterWTTPort	The time master's WeakTrueTime port.

C REST-Interfaces

C.1 Publish/Subscribe Repository

Target	Description	Parameter	Parameter Description
/debug	Get a HTML overview of the current mapping. That is, a list of all published streams and the corresponding publisher(s).	–	–
/publish	Add a new publisher to the repository. Returns a JSON result status object.	?s=<*identifier*>	Identifier of the stream which is published
		?h=<*hostname*>	Hostname (e.g., IP) of the publisher
		?p=<*port*>	Port of the publisher
		?rep=<*isRep*>	True if the publisher is a repeater for this stream. Optional parameter: *isRep* = *false* if not specified.
/subscribe	Returns a publisher for a certain stream in a JSON result object.	?s=<*identifier*>	Identifier of the subscribed stream
		?norep= < *isRepUnallowed* >	True if the retrieved publisher must not be a repeater. Optional parameter: *isRepUnallowed* = *false* if not specified.

C.2 Worker

Target	Description	Parameter	Parameter Description
/debug	Get a HTML overview of all subscribed input as well as all published output data streams of the worker.	–	–
/data	Returns a list of tuples for a certain output stream as a JSON object. The tuple list can be further specified by the parameters.	?s=<*identifier*>	Identifier of the stream
		?i=<*index*>	Tuple list only contains tuples with a larger *allTimeIndex* (i.e., $tuple.allTimeIndex > index$). Optional parameter: $allTimeIndex = -1$ if not specified.
		?l=<*limit*>	Tuple list only contains the latest *limit* tuples. Optional parameter: $limit = \infty$ if not specified.

D Workers and Components

This appendix contains a list of all internal components and PAN workers we implemented in our prototype for evaluating the PAN approach.

D.1 Internal Components

D.1.1 Generic

Name	Input(s)	Output(s)	Description
InterPanStream-ForwarderComponent	List of all input streams to forward	All input streams	Forwards received inter-PAN input streams and thus transforms them into intra-PAN streams.
IntraPanStream-RepeaterComponent	List of all streams to repeat	All input streams	Repeats intra-PAN input streams in order to enable load balancing.

D.1.2 ACM DEBS 2013 Grand Challenge Specific

Name	Input(s)	Output(s)	Description
ActiveBallComponent	All ball sensor data streams (i.e., *SENSOR4*, *SENSOR8*, *SENSOR10* and *SENSOR12*)	*ACTIVEBALL*	Detects the active ball and generates an output stream containing the ID of the active ball as well as its position, acceleration and velocity data.
BallHitDetector-Component	The active ball stream (i.e., *ACTIVEBALL*) as well as all average player position streams (i.e., *A1-A8* and *B1-B8*)	*BALLHITS*	Detects ball hits by means of the ball's acceleration as well as the player who hit the ball.
HeatMapComponent	A single average player position stream (e.g., *B2*)	Heat Map streams for this player	Generates Heat Map Streams for 3 resolutions (16×25, 32×50 and 64×100) and 5 time windows (1, 5, 10, 20 minutes as well as the whole game) as specified in the ACM DEBS 2013 Grand Challenge.
PlayerAverage-Component	All sensor data streams for a specific player (e.g., *SENSOR97* and *SENSOR98* for *B2*)	Average position stream for this player (e.g., *B2*)	Generates a single stream for a specific player by averaging the position, velocity and acceleration data of all sensor data streams.
PlayersBallPossession-Component	The ball hits stream (i.e., *BALLHITS*)	Ball Possession streams for all players	Generates ball possession streams for all players as specified in the ACM DEBS 2013 Grand Challenge.
TeamsBallPossession-Component	The ball possession streams of all players	Ball Possession streams for all teams	Generates ball possession streams for both teams for 5 different time windows (1, 5, 10, 20 minutes as well as the whole game) as specified in the ACM DEBS 2013 Grand Challenge.

D.2 Workers

D.2.1 Generic

Name	Component(s)*	Description
OnlyInterPanStream-ForwardingWorker	–	Forwards all received inter-PAN input streams and thus transforms them into intra-PAN streams.
IntraPanStream-RepeaterWorker	IntraPanStream-RepeaterComponent	Repeats all intra-PAN input streams in order to enable load balancing.

*Please note, that each worker which expects inter-PAN input streams (see JSON config) additionally performs a single InterPanStreamForwarderComponent.

D.2.2 ACM DEBS 2013 Grand Challenge Specific

Name	Component(s)*	Description
ActiveBallWorker	ActiveBallComponent	Detects the active ball and generates an output stream containing the ID of the active ball as well as its position, acceleration and velocity data.
AvgPlayerPosition-Worker	1 PlayerAverageComponent per player	Generates a single stream for each given player (e.g., *B1-B4*) by averaging the position, velocity and acceleration data of their sensor data streams.
BallHitDetectorWorker	BallHitDetector-Component	Detects ball hits by means of the ball's acceleration as well as the player who hit the ball.
HeatMapWorker	1 HeatMapComponent per player	Generates Heat Map Streams for 3 resolutions (16×25, 32×50 and 64×100) and 5 time windows (1, 5, 10, 20 minutes as well as the whole game) as specified in the ACM DEBS 2013 Grand Challenge for each given player.
PlayersBallPossession-Worker	PlayersBallPossession-Component	Generates ball possession streams for all players as specified in the ACM DEBS 2013 Grand Challenge.
TeamsBallPossession-Worker	TeamsBallPossession-Component	Generates ball possession streams for both teams for 5 different time windows (1, 5, 10, 20 minutes as well as the whole game) as specified in the ACM DEBS 2013 Grand Challenge.

E Workflows

This appendix contains graphs illustrating the workflows used in the PAN evaluation (see Section 5.2) as well as an exemplary JSON config. An explanation of the PAN workers used in the following workflows is given in Appendix D. Please note, that for illustration purposes the sensor data streams are abbreviated with their IDs (e.g., *106* instead of *SENSOR106*) in the Full Game graphs.

E.1 Full Game

E.1.1 Full Game on 3 Peers

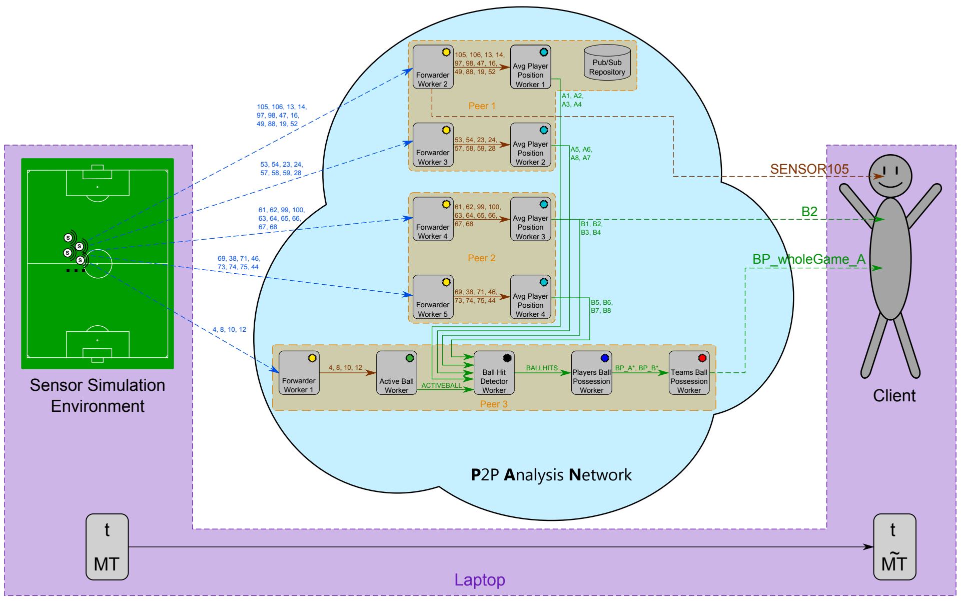


Figure E.1: Full Game Workflow distributed on 3 Peers

E.1.2 Full Game on 6 Peers

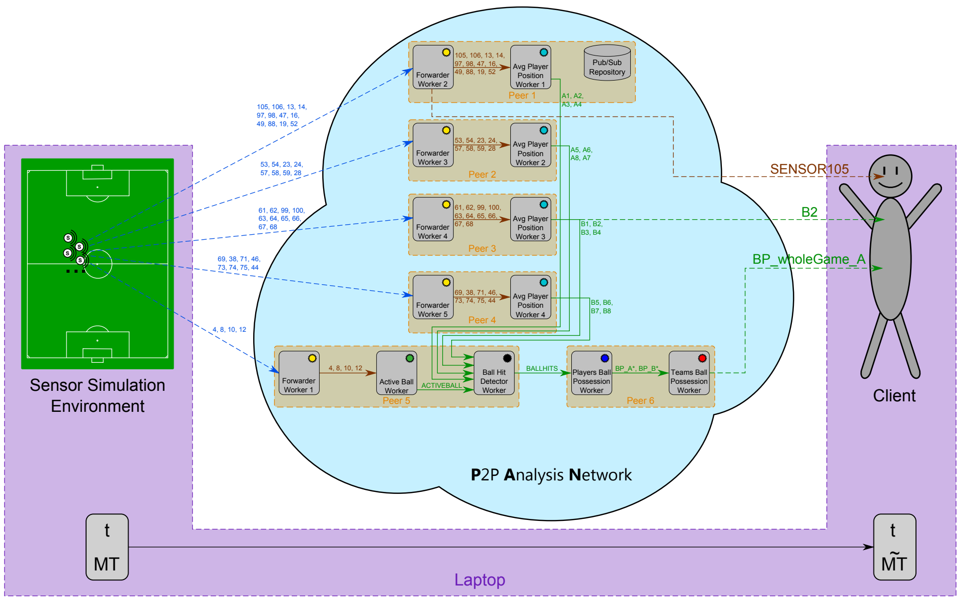


Figure E.2: Full Game Workflow distributed on 6 Peers

E.1.3 Full Game on 8 Peers

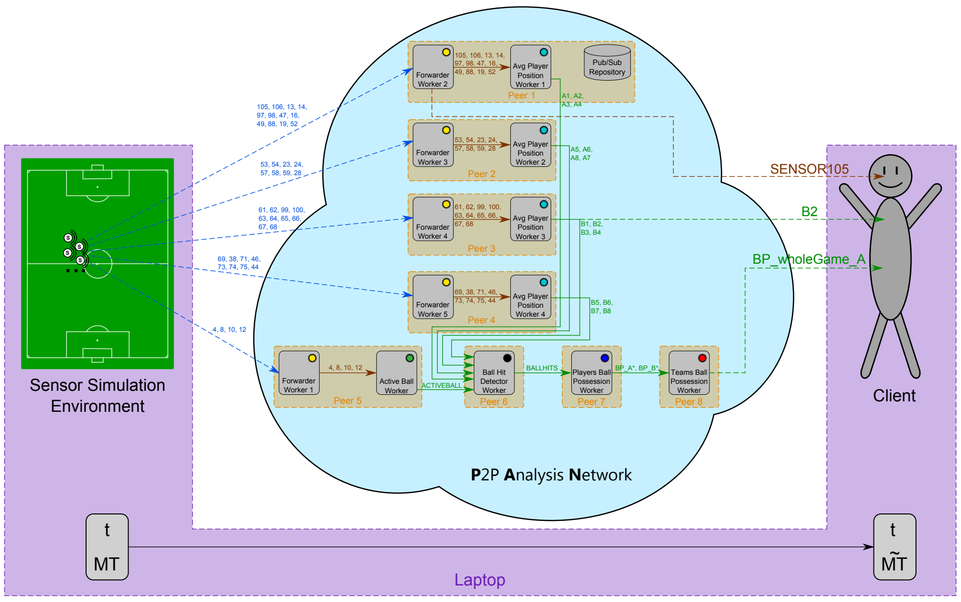


Figure E.3: Full Game Workflow distributed on 8 Peers

E.1.3.1 JSON Config

```
{
        "workflowName": "Full Game Cloud 8 Peers",
        "author": "Lukas Probst",
        "pubSubRepository": {
                "logFileName": "PubSubRepository",
                "hostName": "10.0.0.4",
                "port": "8080",
                "sshPort": "22"
        },
        "workers": [
{
                        "name": "Forwarder Worker 1",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.
                            OnlyInterPanStreamForwardingWorkerStarter",
                        "logFileName": "Forwarder1",
                        "hostName": "10.0.0.8",
                        "sshPort": "22",
                        "outputPort": "51001",
                        "interPanInputStreamsReceiverPort": "50001",
                        "interPanInputStreams": ["SENSOR4", "SENSOR8", "SENSOR10", "SENSOR12
                            "],
                        "additionalParametersString": ""
                },
                {
                        "name": "Forwarder Worker 2",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.
                            OnlyInterPanStreamForwardingWorkerStarter",
                        "logFileName": "Forwarder2",
                        "hostName": "10.0.0.4",
                        "sshPort": "22",
                        "outputPort": "51002",
                        "interPanInputStreamsReceiverPort": "50002",
                        "interPanInputStreams": ["SENSOR105", "SENSOR106", "SENSOR13", "
                            SENSOR14", "SENSOR97", "SENSOR98", "SENSOR47", "SENSOR16", "
                            SENSOR49", "SENSOR88", "SENSOR19", "SENSOR52"],
                        "additionalParametersString": ""
                },
                {
                        "name": "Forwarder Worker 3",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.
                            OnlyInterPanStreamForwardingWorkerStarter",
                        "logFileName": "Forwarder3",
                        "hostName": "10.0.0.5",
                        "sshPort": "22",
                        "outputPort": "51003",
                        "interPanInputStreamsReceiverPort": "50003",
                        "interPanInputStreams": ["SENSOR53", "SENSOR54", "SENSOR23", "SENSOR24
                            ", "SENSOR57", "SENSOR58", "SENSOR59", "SENSOR28"],
                        "additionalParametersString": ""
                },
                {
                        "name": "Forwarder Worker 4",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.
                            OnlyInterPanStreamForwardingWorkerStarter",
                        "logFileName": "Forwarder4",
                        "hostName": "10.0.0.6",
                        "sshPort": "22",
                        "outputPort": "51004",
                        "interPanInputStreamsReceiverPort": "50004",
                        "interPanInputStreams": ["SENSOR61", "SENSOR62", "SENSOR99", "
                            SENSOR100", "SENSOR63", "SENSOR64", "SENSOR65", "SENSOR66", "
                            SENSOR67", "SENSOR68"],
                        "additionalParametersString": ""
                },
                {
                        "name": "Forwarder Worker 5",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.
                            OnlyInterPanStreamForwardingWorkerStarter",
                        "logFileName": "Forwarder5",
                        "hostName": "10.0.0.7",
                        "sshPort": "22",
                        "outputPort": "51005",
                        "interPanInputStreamsReceiverPort": "50005",
                        "interPanInputStreams": ["SENSOR69", "SENSOR38", "SENSOR71", "SENSOR40
                            ", "SENSOR73", "SENSOR74", "SENSOR75", "SENSOR44"],
                        "additionalParametersString": ""
                },
                {
                        "name": "Active Ball Worker",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                            ActiveBallWorkerStarter",
                        "logFileName": "ActiveBall",
                        "hostName": "10.0.0.8",
                        "sshPort": "22",
                        "outputPort": "51006",
                        "interPanInputStreamsReceiverPort": "50006",
                        "interPanInputStreams": [],
                        "additionalParametersString": ""
                },
                {
                        "name": "Average Player Position Worker 1",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                            AvgPlayerPositionWorkerStarter",
                        "logFileName": "PlayerAveragePosition1",
                        "hostName": "10.0.0.4",
                        "sshPort": "22",
                        "outputPort": "51007",
                        "interPanInputStreamsReceiverPort": "50007",
                        "interPanInputStreams": [],
                        "additionalParametersString": "REFEREE:@SENSOR105,SENSOR106@%A1:
                            @SENSOR13,SENSOR14,SENSOR97,SENSOR98@%A2:@SENSOR47,SENSOR16@%A3:
                            @SENSOR49,SENSOR88@%A4:@SENSOR19,SENSOR52@"
                },
                {
                        "name": "Average Player Position Worker 2",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                            AvgPlayerPositionWorkerStarter",
                        "logFileName": "PlayerAveragePosition2",
                        "hostName": "10.0.0.5",
                        "sshPort": "22",
                        "outputPort": "51008",
                        "interPanInputStreamsReceiverPort": "50008",
                        "interPanInputStreams": [],
                        "additionalParametersString": "A5:@SENSOR53,SENSOR54@%A6:@SENSOR23,
                            SENSOR24@%A7:@SENSOR57,SENSOR58@%A8:@SENSOR59,SENSOR28@"
                },
                {
                        "name": "Average Player Position Worker 3",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                            AvgPlayerPositionWorkerStarter",
                        "logFileName": "PlayerAveragePosition3",
                        "hostName": "10.0.0.6",
                        "sshPort": "22",
                        "outputPort": "51009",
                        "interPanInputStreamsReceiverPort": "50009",
                        "interPanInputStreams": [],
                        "additionalParametersString": "B1:@SENSOR61,SENSOR62,SENSOR99,
                             SENSOR100@%B2:@SENSOR63,SENSOR64@%B3:@SENSOR65,SENSOR66@%B4:
                             @SENSOR67,SENSOR68@"
                },
                {
                        "name": "Average Player Position Worker 4",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                             AvgPlayerPositionWorkerStarter",
                        "logFileName": "PlayerAveragePosition4",
                        "hostName": "10.0.0.7",
                        "sshPort": "22",
                        "outputPort": "51010",
                        "interPanInputStreamsReceiverPort": "50010",
                        "interPanInputStreams": [],
                        "additionalParametersString": "B5:@SENSOR69,SENSOR38@%B6:@SENSOR71,
                             SENSOR40@%B7:@SENSOR73,SENSOR74@%B8:@SENSOR75,SENSOR44@"
                },
                {
                        "name": "Ball Hit Detector Worker",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                             BallHitDetectorWorkerStarter",
                        "logFileName": "BallHitDetector",
                        "hostName": "10.0.0.9",
                        "sshPort": "22",
                        "outputPort": "51011",
                        "interPanInputStreamsReceiverPort": "50011",
                        "interPanInputStreams": [],
                        "additionalParametersString": ""
                },
                {
                        "name": "Players Ball Possession Worker",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                             PlayersBallPossessionWorkerStarter",
                        "logFileName": "PlayersBallPossession",
                        "hostName": "10.0.0.10",
                        "sshPort": "22",
                        "outputPort": "51012",
                        "interPanInputStreamsReceiverPort": "50012",
                        "interPanInputStreams": [],
                        "additionalParametersString": ""
                },
                {
                        "name": "Teams Ball Possession Worker",
                        "mainClass": "ch.unibas.cs.dbis.pan.worker.starter.debs.
                             TeamsBallPossessionWorkerStarter",
                        "logFileName": "TeamsBallPossession",
                        "hostName": "10.0.0.11",
                        "sshPort": "22",
                        "outputPort": "51013",
                        "interPanInputStreamsReceiverPort": "50013",
                        "interPanInputStreams": [],
                        "additionalParametersString": "A:@A1,A2,A3,A4,A5,A6,A7,A8@%B:@B1,B2,B3
                             ,B4,B5,B6,B7,B8@"
                }
        ]
}
```

Listing E.1: fullGameCloud8Peers.json

E.1.4 Full Game on 14 Peers

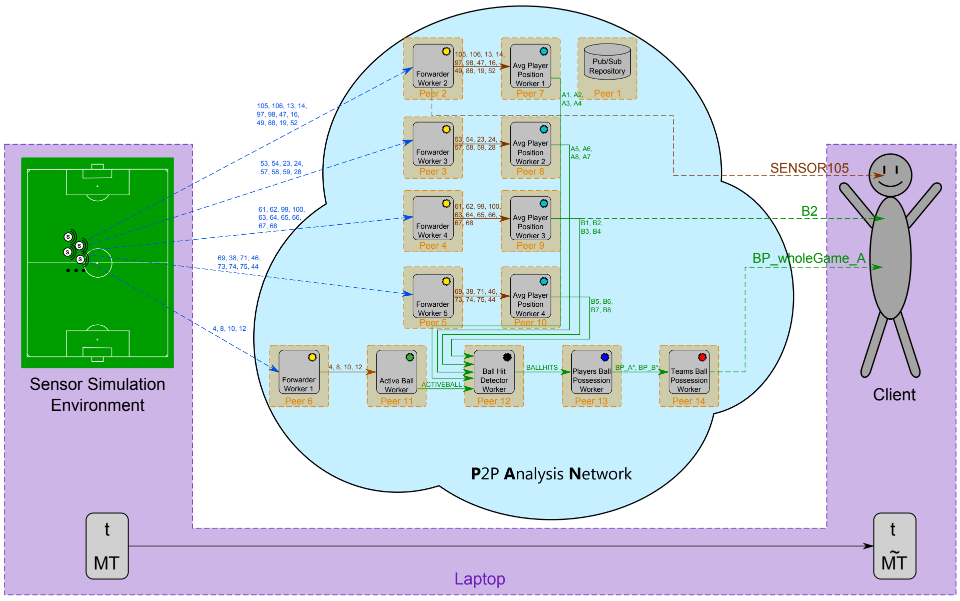


Figure E.4: Full Game Workflow distributed on 14 Peers

E.2 Sensor Forwarding

E.2.1 1 Forwarder

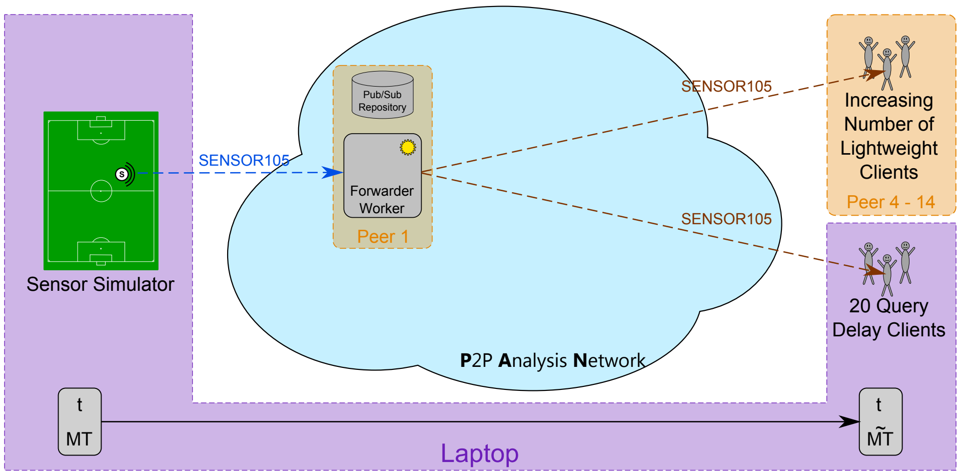


Figure E.5: Single Stream Publisher (1 Forwarder)

E.2.2 1 Forwarder, 1 Repeater

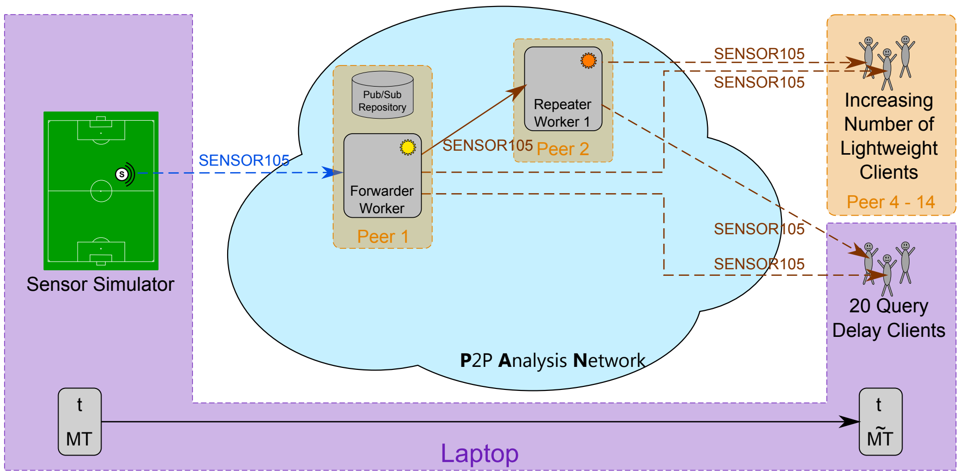


Figure E.6: Two Stream Publishers (1 Forwarder, 1 Repeater)

E.2.3 1 Forwarder, 2 Repeaters

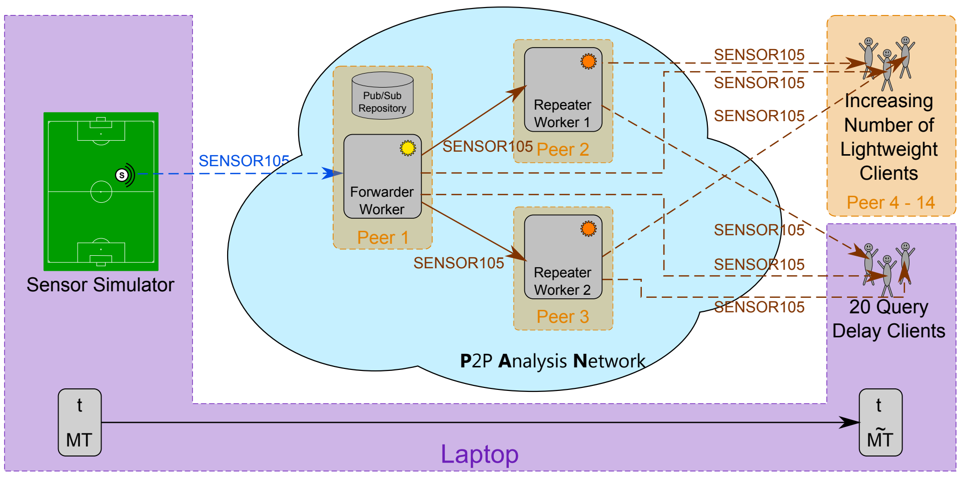


Figure E.7: Three Stream Publishers (1 Forwarder, 2 Repeaters)

E.3 Heat Map

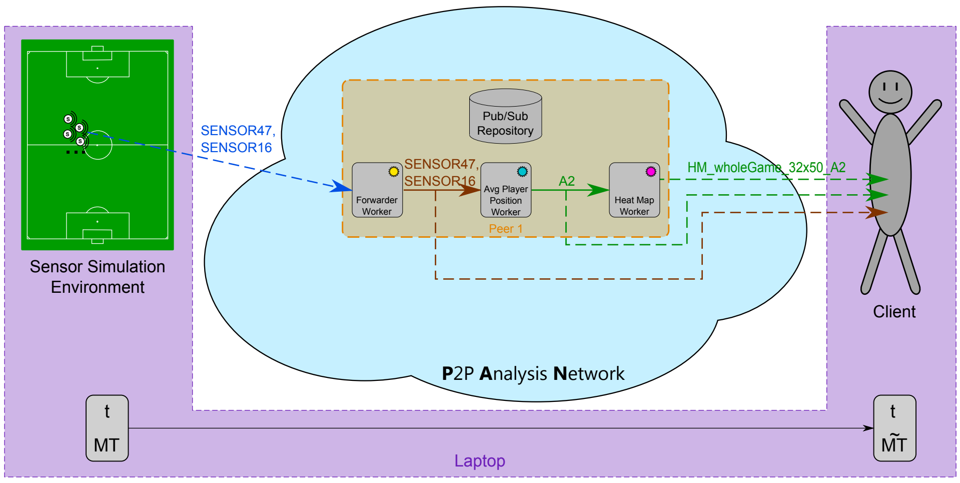


Figure E.8: Heat Map Workflow generating the Heat Maps for Player A2

F
Evaluation Settings

F.1 Sensor Simulator Config

Config file: *global/Constants.java*

Variable name	Value
MIN_BUFFER_SIZE	100
MAX_BUFFER_SIZE	500
MATCH_START_TIMESTAMP_IN_PICOSECONDS	10753295594424116
FIRST_HALF_END_TIMESTAMP_IN_PICOSECONDS	12557295594424116
SECOND_HALF_START_TIMESTAMP_IN_PICOSECONDS	13086639146403497
MATCH_END_TIMESTAMP_IN_PICOSECONDS	14879639146403497
SKIP_HALF_TIME_BREAK_IN_SIMULATION	false
TIME_TO_WAIT_FOR_THE_TIMEPROVIDER_ TO_INITIALIZE_IN_MS	3000
CHECK_PERIOD_BEFORE_START_SIMULATION_IN_MS	10
CHECK_PERIOD_IN_MS	50
STATISTICS_CALCULATION_PERIOD_IN_MS	5000
MATCH_SIMULATION_SPEEDUP	1.0
USE_WEAK_TRUE_TIME	true
WTT_SYNC_PERIOD_IN_MS	1000
WTT_ALIVE_PERIOD_IN_MS	5000
WTT_ALIVE_TIMEOUT_IN_MS	17000

F.2 PAN Config

Config file: *ch/unibas/cs/dbis/pan/helper/Config.java*

Variable name	Value
DEFAULT_WORKER_IO_RING_BUFFER_SIZE	100
HEATMAP_IO_RING_BUFFER_SIZE	50
INTER_PAN_STREAM_FORWARDER_INTERVAL_IN_MS	20
INTRA_PAN_STREAM_REPEATER_INTERVAL_IN_MS	20
WAITINGTIME_BEFORE_SUBSCRIBE_IN_MS	10000
ENABLE_AUTOMATIC_PERIODICALLY_PULL_FROM_OTHER_WORKER	true
PLAYER_AVERAGE_INTERVAL_IN_MS	5
ACTIVE_BALL_INTERVAL_IN_MS	5
BALL_HIT_DETECTOR_INTERVAL_IN_MS	5
PLAYERS_BALL_POSSESSION_INTERVAL_IN_MS	500
TEAM_BALL_POSSESSION_INTERVAL_IN_MS	1000
HEAT_MAP_INTERVAL_IN_MS	1000
GRAB_INTERVAL_FOR_LONG_TERM_STATISTICS_IN_MS	100
HTTP_CLIENT_TIMEOUT	15000

F.3 Client Config

Config file: *ch/unibas/cs/dbis/pan/debsdebugging/helper/Config.java*

Variable name	Value
DATA_GRABBER_INTERVAL_IN_MS	20
PUBSUB_REPOSITORY_HOSTNAME	10.0.0.4
PUBSUB_REPOSITORY_PORT	8080
HTTP_CLIENT_TIMEOUT	15000
DRAW_FIELD_WIDTH	900
DRAW_X_MARGIN	150
DRAW_Y_MARGIN	150
DRAW_VELOCITY_DIVISOR	50000000
HEAT_MAP_PREFIX	HEATMAP_wholeGame_32x50_
MATCH_START_TIMESTAMP_IN_PICOSECONDS	10753295594424116
STOP_MATCH_TIMESTAMP_IN_PICOSECONDS	12253295594424116

G Evaluation Results

G.1 Sensor Simulation Environment

The followings graphs show time difference statistics of incoming sensor data streams produced by the sensor simulator environment in milliseconds. The statistics were measured every 5 seconds at a dedicated Debugging Stream Receiver. All 42 sensors are simulated in real-time. In the distributed setups the sensors are equally distributed onto two machines. The machine clock of the second machine was manually set approximately 20 seconds into the pasts. The sliding window size for the moving averages is 10.

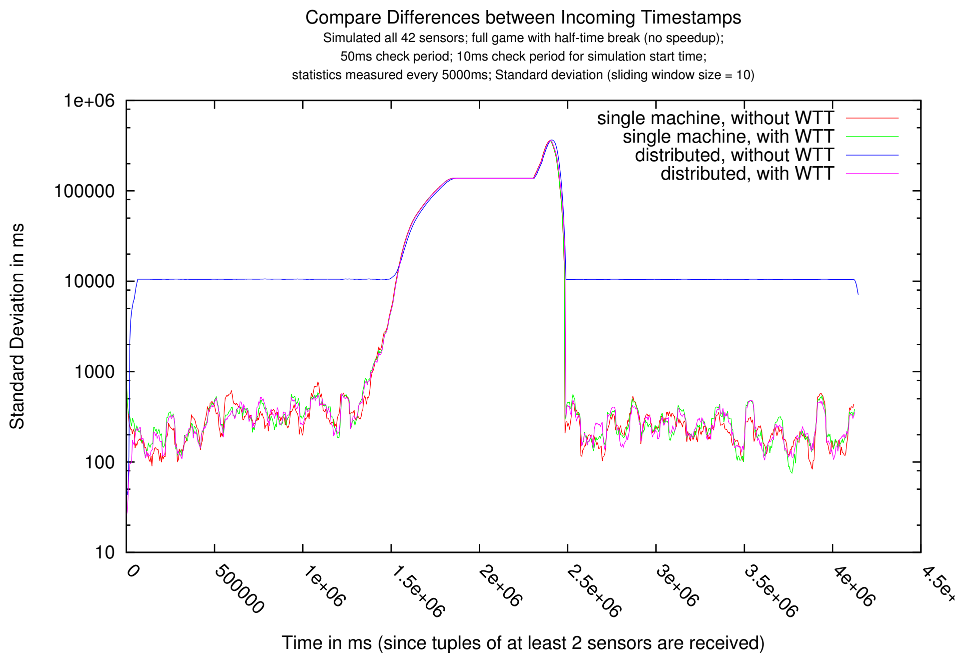


Figure G.1: Time Difference Comparison. Compares the standard deviation (sliding window average) for all four setups.

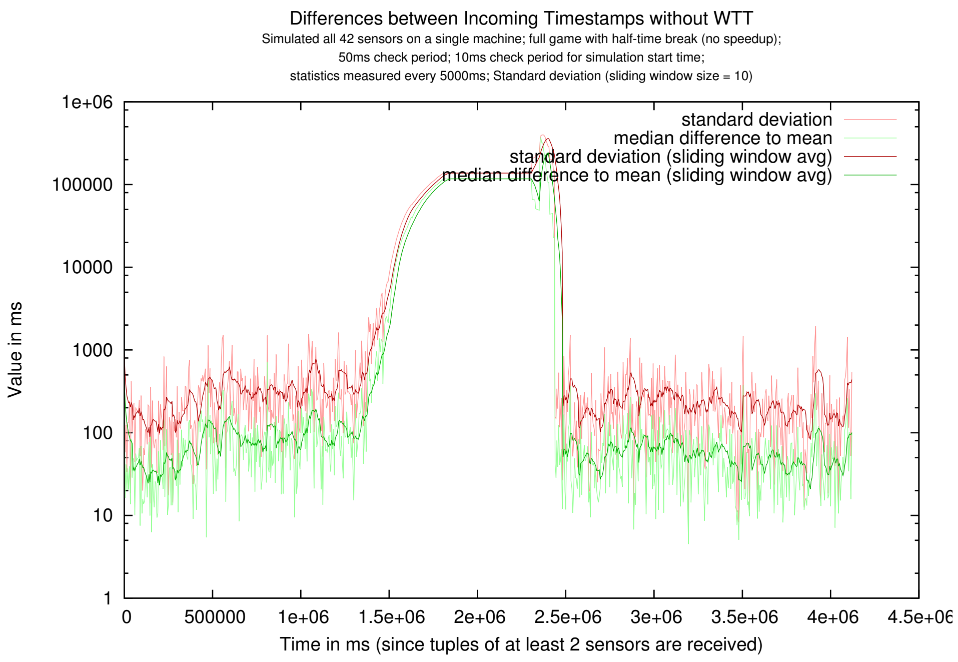


Figure G.2: Time Difference Statistics for Single Machine Setup without WeakTrueTime

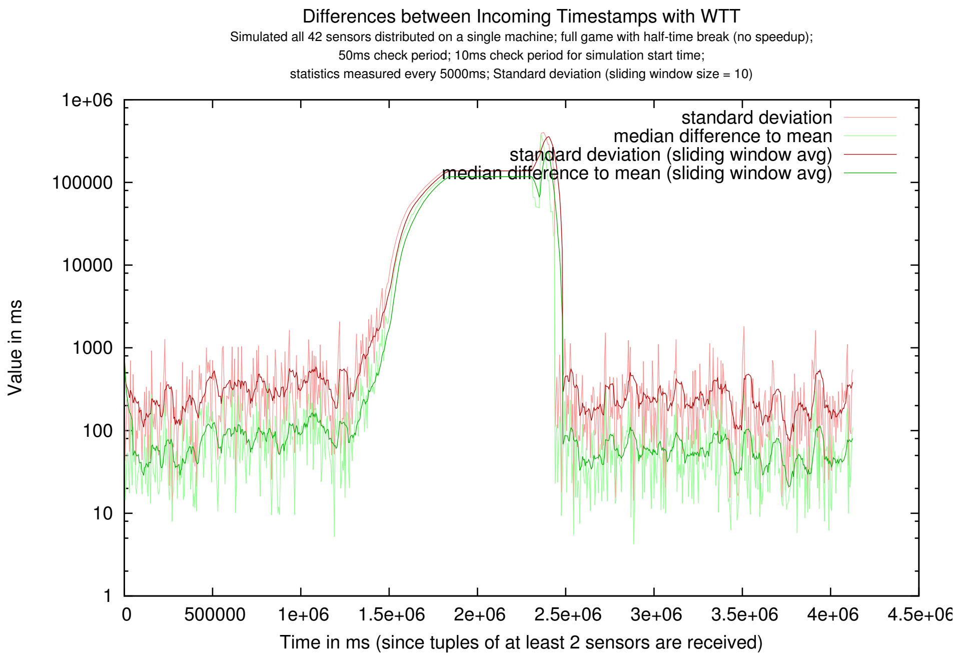


Figure G.3: Time Difference Statistics for Single Machine Setup with WeakTrueTime

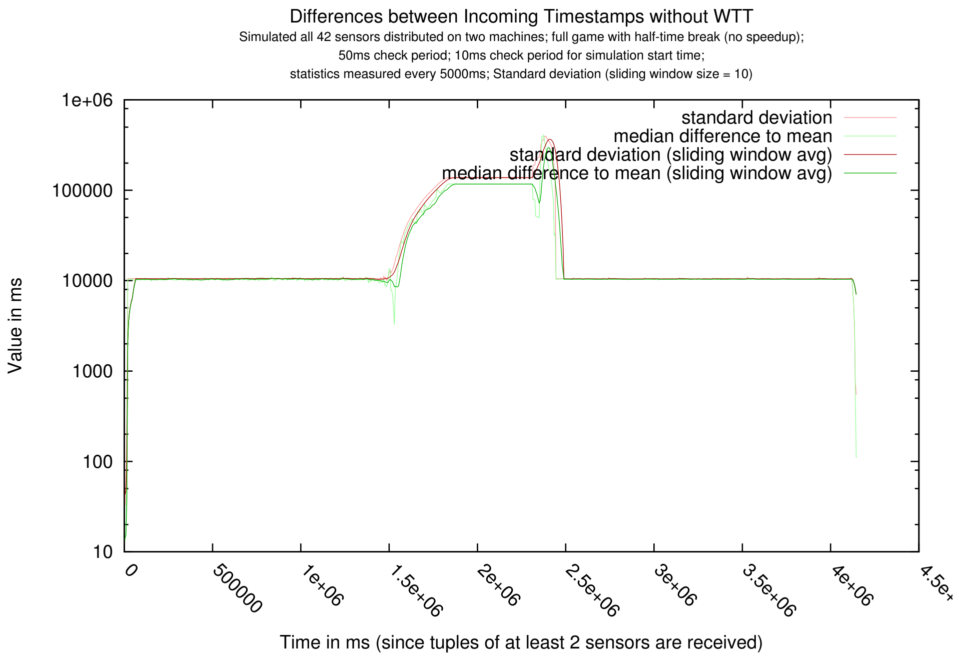


Figure G.4: Time Difference Statistics for Distributed Setup without WeakTrueTime

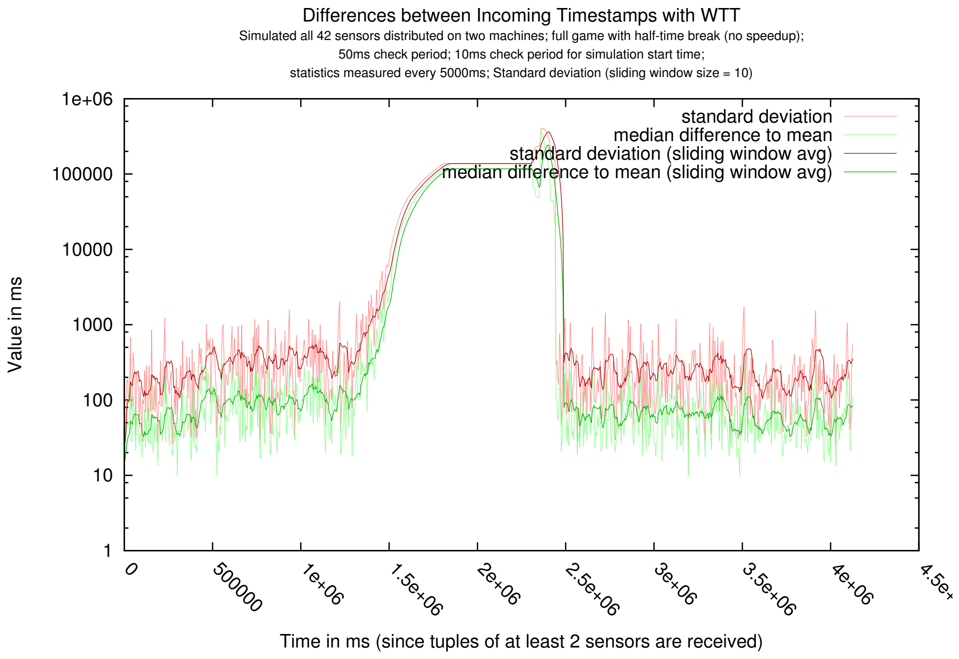


Figure G.5: Time Difference Statistics for Distributed Setup with WeakTrueTime

G.2 PAN

This section contains graphs and tables for all values measured in the course of evaluating PAN. The average (*avg*), variance (*var*) and standard deviation (*std*) are rounded to two decimal places. *#retrievedTuples* denotes the number of different (w.r.t. the tuple timestamp) retrieved tuples during the data fetching at the query delay client.

G.2.1 Degree of Distribution

Evaluation data for Section 5.2.3.2.

G.2.1.1 Table

Stream	#Peers	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
SENSOR105	3	69.76	2820.75	53.11	19	63	94	244	1454	8474
	6	66.61	35076.96	187.29	20	60	86	162	13740	10651
	8	73.79	3831.43	61.90	19	68	104	214	3735	10709
	14	88.96	2157.47	46.45	37	84	115	214	1683	8620
B2	3	2923.73	1.15E8	10727.63	31	127	320	49364	53090	8719
	6	123.99	94864.37	308.00	26	86	137	1205	13819	12488
	8	165.68	300369.00	548.06	29	96	149	3248	8116	12440
	14	114.72	9008.44	94.91	43	101	148	350	2156	10305
BP_wholeGame_A	3	1141.07	446151.12	667.95	215	1014	1684	3951	6602	554
	6	1078.43	469419.62	685.14	242	1025	1603	2900	13199	569
	8	948.44	188761.48	434.47	214	901	1426	1914	5537	688
	14	961.06	138894.02	372.68	214	925	1438	1729	3743	732

Table G.1: Statistics for *SENSOR105*, *B2* and *BP_wholeGame_A* in Increasing Number of Peers Evaluation

G.2.1.2 Graphs

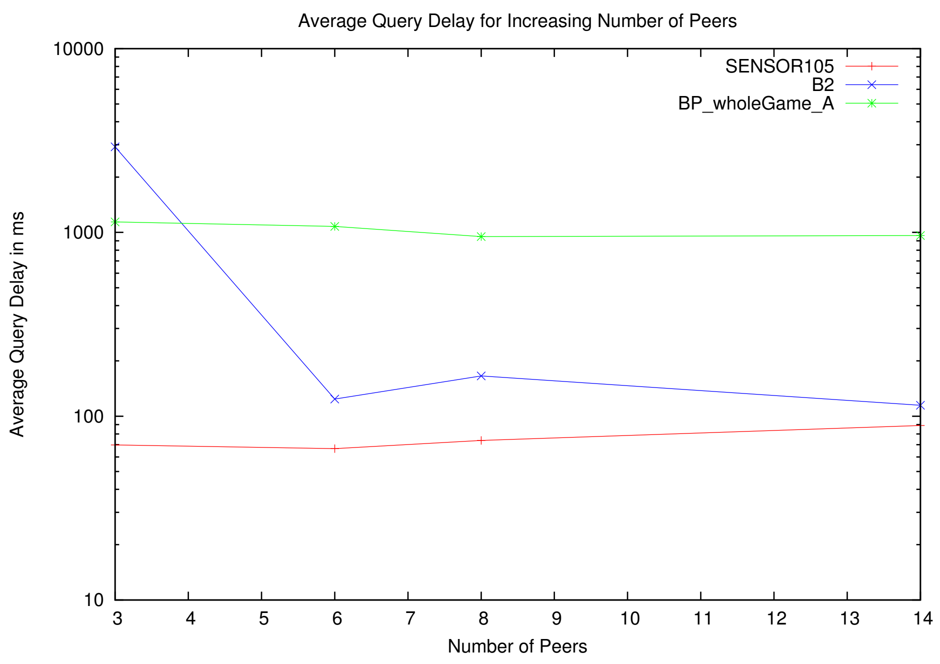


Figure G.6: Average Query Delay for Increasing Number of Peers

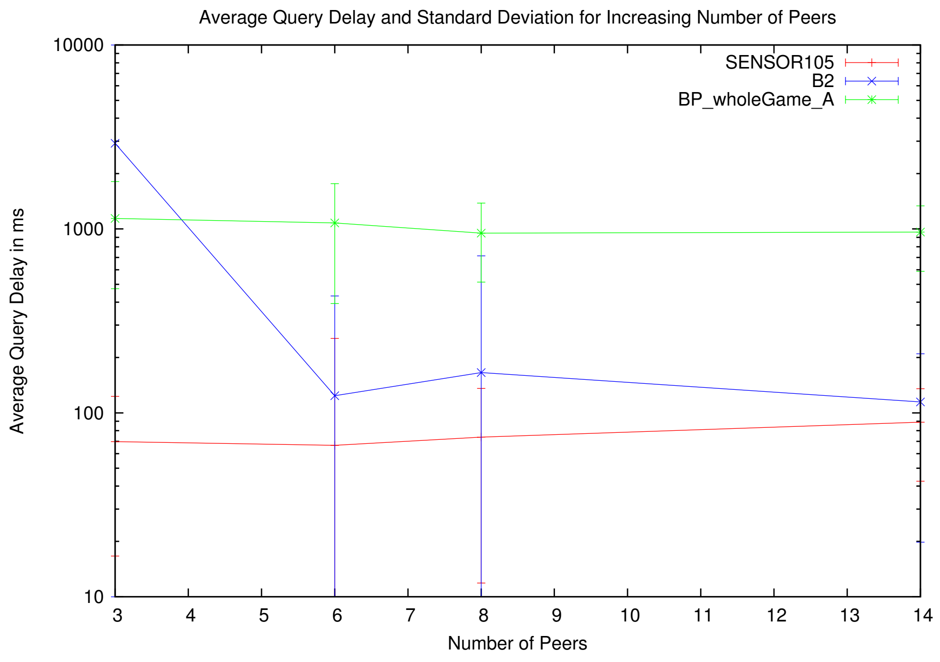


Figure G.7: Average Query Delay and Standard Deviation for Increasing Number of Peers

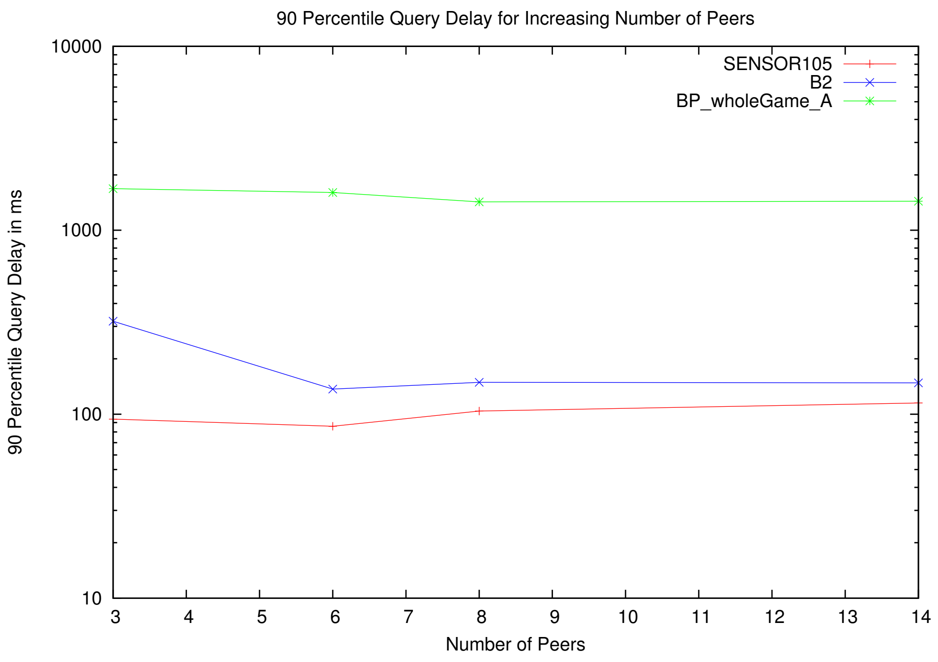


Figure G.8: 90 Percentile Query Delay for Increasing Number of Peers

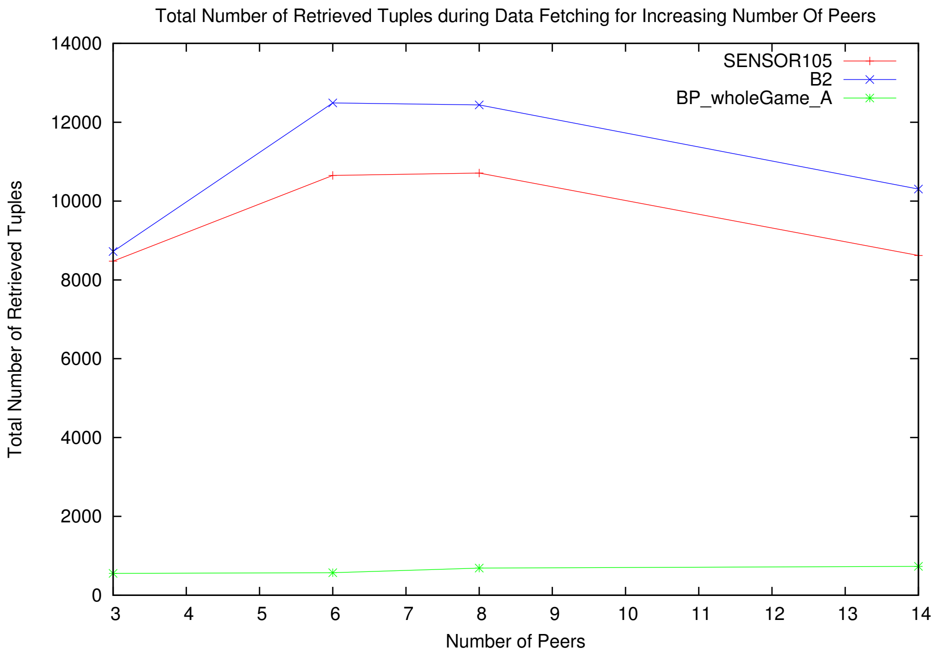


Figure G.9: Total Number of Retrieved Tuples during Data Fetching for Increasing Number of Peers

G.2.2 Latency

Evaluation data for Section 5.2.3.3 (Latency).

G.2.2.1 Table

Stream	Latency in ms	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
SENSOR105	0	60.74	508.55	22.55	20	59	83	135	326	13194
	5	71.89	615.22	24.80	27	69	95	165	308	10791
	10	82.93	1063.69	32.61	33	79	108	200	1545	9495
	15	93.44	1154.41	33.98	36	90	121	229	1216	8573
	20	103.92	1172.36	34.24	43	101	133	235	656	7811
	25	117.17	1404.39	37.48	49	112	149	267	607	7172
	30	119.96	1672.64	40.90	55	113	158	282	678	6556
	35	130.42	1850.33	43.02	59	123	165	294	1505	6124
B2	0	92.47	13730.05	117.18	25	78	115	301	3153	15870
	5	98.50	661.69	25.72	41	96	123	212	434	13167
	10	116.53	1106.37	33.26	52	114	144	232	1783	11500
	15	136.63	1173.78	34.26	65	134	168	268	1601	10290
	20	153.66	1268.20	35.61	77	150	188	283	1421	9302
	25	175.67	1502.08	38.76	76	173	215	310	1107	8456
	30	187.28	2104.74	45.88	99	184	230	327	2113	7754
	35	205.39	2419.01	49.18	108	201	253	354	1897	7203
BP_wholeGame_A	0	906.31	121925.66	349.18	189	891	1347	1645	4048	733
	5	1002.23	137290.13	370.53	272	965	1461	2060	3144	727
	10	1049.32	115296.14	339.55	288	1015	1509	1759	2296	711
	15	1100.87	114840.34	338.88	309	1101	1524	1847	2941	698
	20	1189.23	117260.36	342.43	410	1182	1638	1921	2082	612
	25	1207.77	111491.32	333.90	498	1211	1646	1907	2165	587
	30	1309.25	116811.62	341.78	505	1296	1737	1961	3384	524
	35	1360.17	123405.32	351.29	549	1346	1799	2093	3908	483

Table G.2: Statistics for *SENSOR105*, *B2* and *BP_wholeGame_A* in Increasing Latency Evaluation

G.2.2.2 Graphs

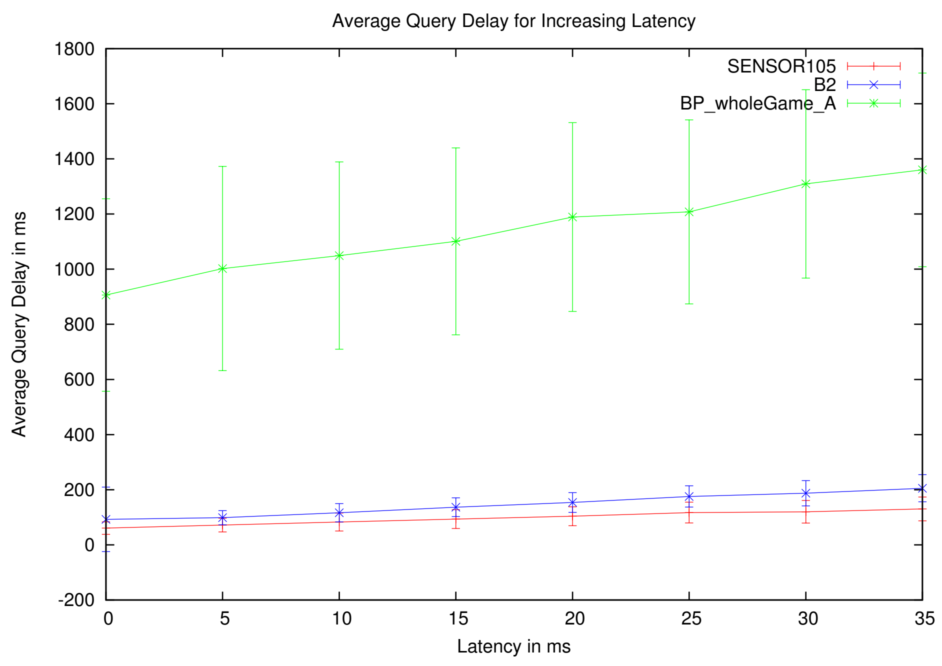


Figure G.10: Average Query Delay and Standard Deviation for Increasing Latency

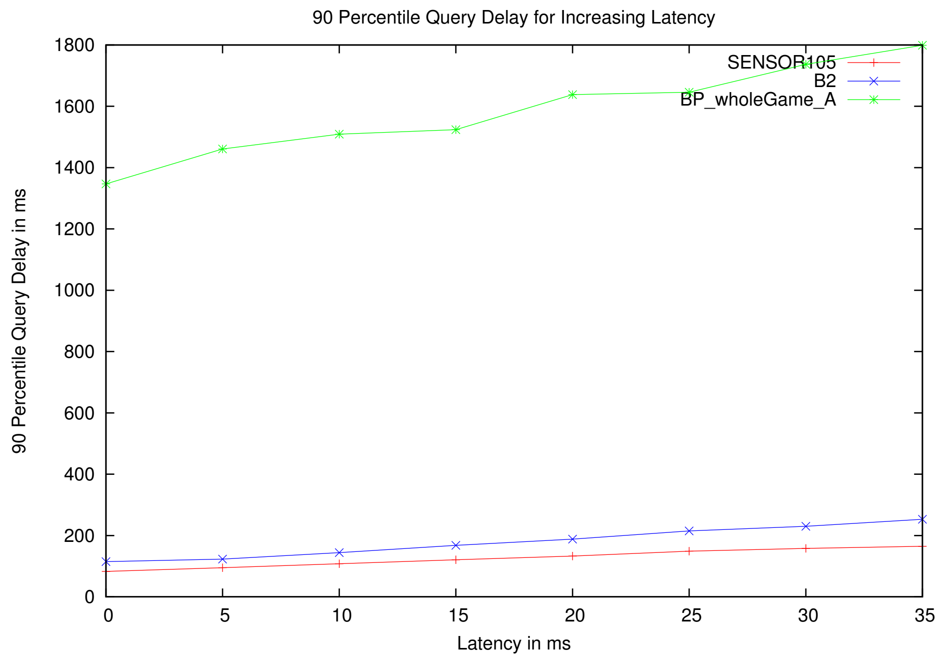


Figure G.11: 90 Percentile Query Delay for Increasing Latency

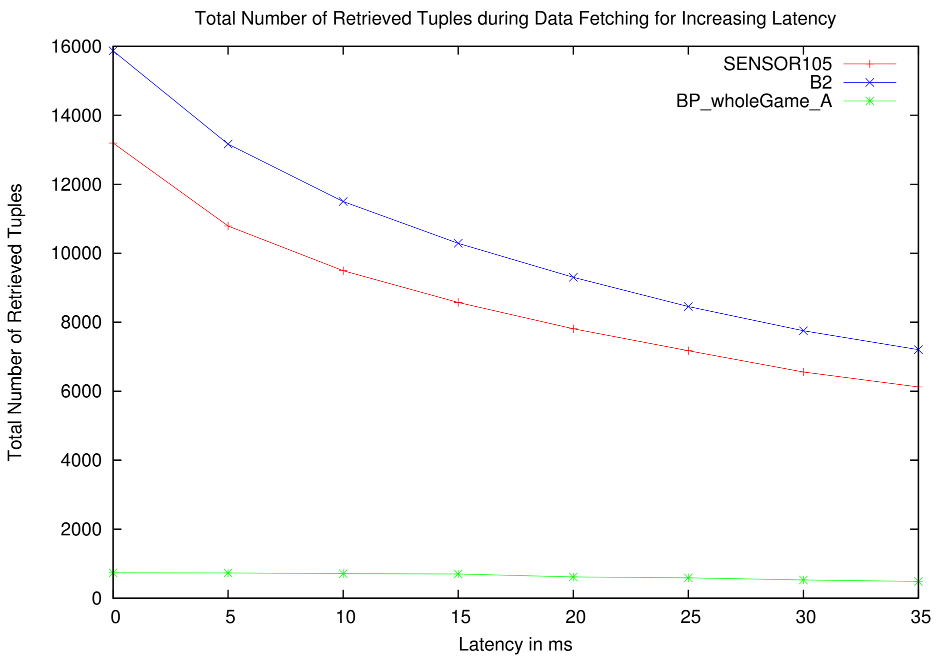


Figure G.12: Total Number of Retrieved Tuples during Data Fetching for Increasing Latency

G.2.3 Bandwidth

Evaluation data for Section 5.2.3.3 (Bandwidth).

G.2.3.1 Table

Stream	Bandwidth in kb/s	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
SENSOR105	50000	61.35	1295.86	36.00	19	59	83	164	1837	13290
	47500	60.95	664.76	25.78	20	59	83	175	784	13232
	45000	61.11	819.47	28.63	19	59	83	166	966	13462
	42500	61.46	1030.92	32.11	19	59	83	187	1407	13410
	40000	60.99	660.97	25.71	19	59	83	166	589	13329
	37500	61.15	938.38	30.63	19	58	83	174	1029	13627
	35000	60.83	870.03	29.50	19	58	82	158	1331	13512
	32500	61.30	852.68	29.20	19	59	83	171	974	13348
	30000	60.77	749.64	27.38	19	59	82	164	933	13532
	27500	61.22	700.86	26.47	19	59	83	196	587	13428
	25000	61.67	1079.07	32.85	18	59	83	167	1103	13303
	22500	62.44	2436.46	49.36	19	59	83	188	1967	13257
	20000	61.67	1163.79	34.11	19	59	83	197	1551	13290
	17500	62.50	866.42	29.44	19	60	85	184	987	13121
	15000	91.79	1625.26	40.31	37	86	118	262	1212	8596
	12500	74.36	23580.56	153.56	20	67	98	224	7718	11222
	10000	62.95	1367.70	36.98	19	60	84	213	1839	13085
	7500	95.25	2113.27	45.97	36	89	122	270	1033	8197
	5000	99.16	3331.83	57.72	38	92	126	283	1601	8229
	2500	157.29	6566.32	81.03	42	144	201	465	1926	6268

Table G.3: Statistics for *SENSOR105* in Decreasing Bandwidth Evaluation

Stream	Bandwidth in kb/s	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
B2	50000	91.68	14593.98	120.80	25	78	111	293	3061	16103
	47500	94.23	18467.85	135.90	26	78	119	307	3585	15892
	45000	88.74	18213.43	134.96	28	76	106	279	3762	16349
	42500	86.37	5237.38	72.37	25	77	108	277	2035	16279
	40000	88.70	7832.82	88.50	25	78	114	271	2626	16125
	37500	87.05	9646.29	98.22	26	77	107	272	2857	16482
	35000	86.66	7855.77	88.63	24	77	107	278	2699	16360
	32500	90.43	19733.21	140.47	26	77	108	291	3868	16194
	30000	86.37	6124.04	78.26	25	77	107	272	1870	16384
	27500	85.96	6135.63	78.33	27	77	107	264	2325	16327
	25000	90.30	10664.71	103.27	26	78	116	281	3149	16065
	22500	92.07	14368.42	119.87	25	78	109	324	3468	16008
	20000	89.70	6534.78	80.84	27	78	116	302	2351	16053
	17500	93.91	12273.73	110.79	24	79	121	324	2775	15773
	15000	116.22	14350.95	119.80	42	102	140	357	3024	10256
	12500	98.10	25867.37	160.83	26	85	122	300	7429	13399
	10000	93.55	22770.19	150.90	24	79	111	285	3825	15886
	7500	126.84	17890.08	133.75	45	104	174	564	3277	9683
	5000	126.75	24527.17	156.61	49	105	163	429	3556	9807
	2500	200.29	47751.59	218.52	69	164	266	756	4522	7279

Table G.4: Statistics for *B2* in Decreasing Bandwidth Evaluation

Stream	Bandwidth in kb/s	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
BP_wholeGame_A	50000	1044.19	176456.22	420.07	233	999	1545	2310	3032	513
	47500	1051.81	176978.31	420.69	221	1018	1556	2297	3354	503
	45000	1114.99	206940.27	454.91	199	1066	1636	2801	3431	495
	42500	1064.83	211567.60	459.96	193	1037	1585	2518	4547	467
	40000	1101.46	201861.41	449.29	219	1087	1560	2367	5327	444
	37500	1087.17	204672.97	452.41	211	1058	1559	2695	4253	413
	35000	1155.56	212143.37	460.59	250	1153	1689	2854	3581	394
	32500	1152.32	261831.49	511.69	212	1107	1728	3212	4540	328
	30000	1466.11	667072.52	816.75	363	1309	2391	4693	5549	254
	27500	1547.03	812825.59	901.57	199	1377	2622	4947	6927	237
	25000	2371.05	1969983.87	1403.56	344	1980	4382	6707	6967	153
	22500	3876.59	2972704.09	1724.15	581	3861	6166	8458	8458	92
	20000	5183.55	1926555.67	1388.00	2449	5002	6885	9311	9311	66
	17500	6394.34	2257519.37	1502.50	2605	6104	8019	10653	10653	61
	15000	8109.09	3845726.99	1961.05	3902	7273	10194	10451	10451	11
	12500	11558.61	6314664.57	2512.90	7446	11334	15262	16522	16522	18
	10000	13930.88	6668665.73	2582.38	10319	14275	17004	19547	19547	16
	7500	2272.67	333383.89	577.39	1293	2254	3115	4154	4154	49
	5000	7837.11	5957408.54	2440.78	3104	7945	12374	12374	12374	9
	2500	10659.50	158802.25	398.50	10261	11058	11058	11058	11058	2

Table G.5: Statistics for *BP_wholeGame_A* in Decreasing Bandwidth Evaluation

G.2.3.2 Graphs

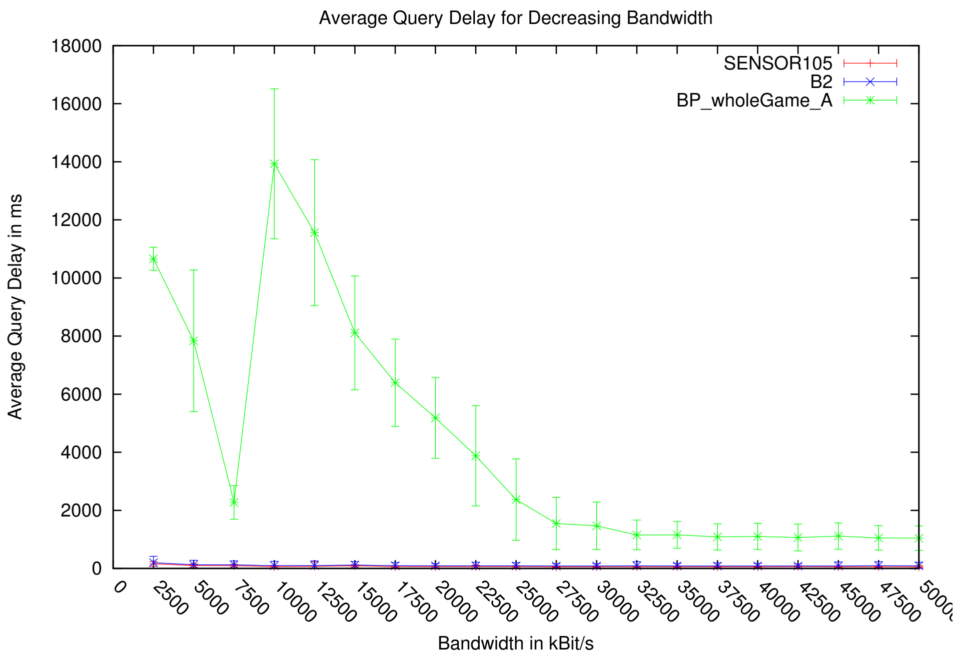


Figure G.13: Average Query Delay and Standard Deviation for Decreasing Bandwidth

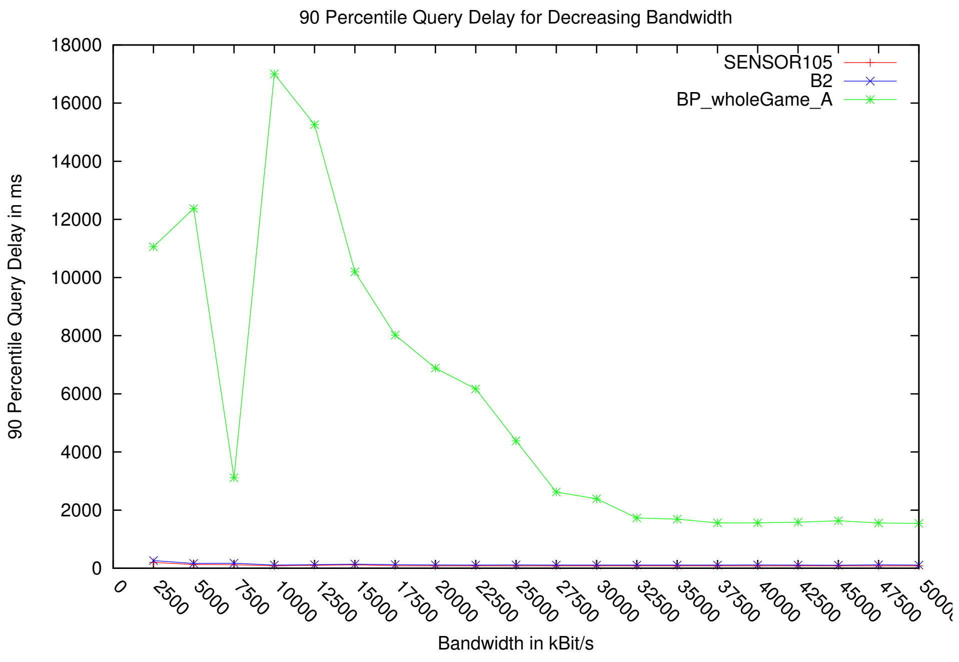


Figure G.14: 90 Percentile Query Delay for Decreasing Bandwidth

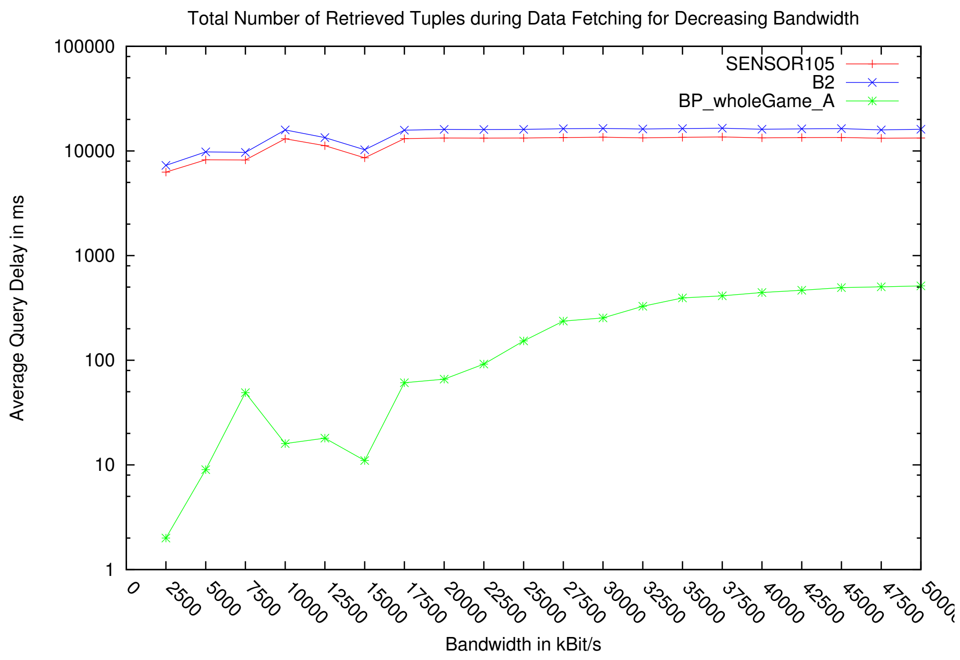


Figure G.15: Total Number of Retrieved Tuples during Data Fetching for Decreasing Bandwidth

G.2.4 Consistency

Evaluation data for Section 5.2.3.4.

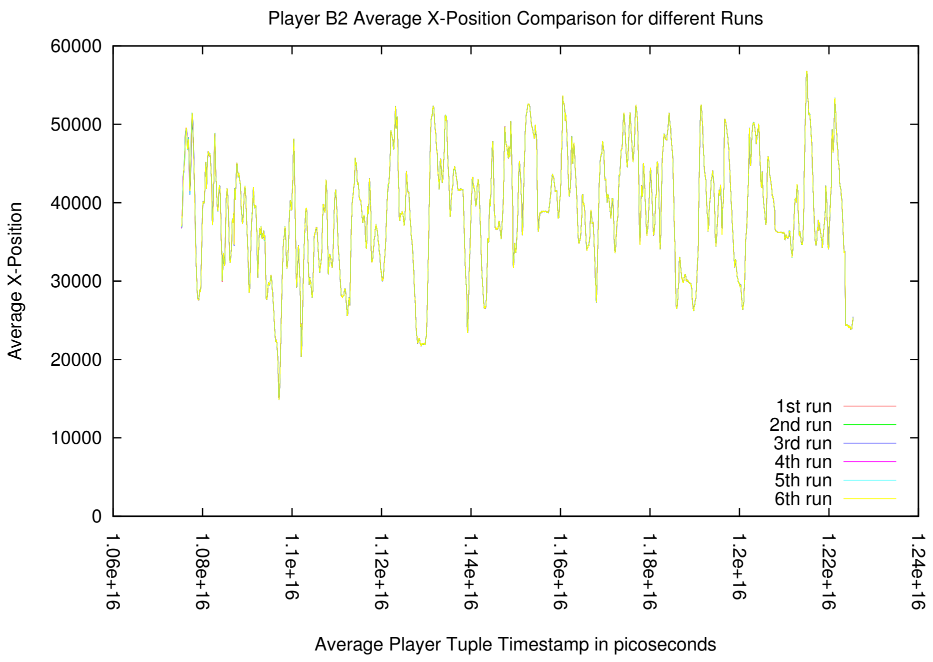


Figure G.16: Player B2 Average X-Position Comparison for different Runs

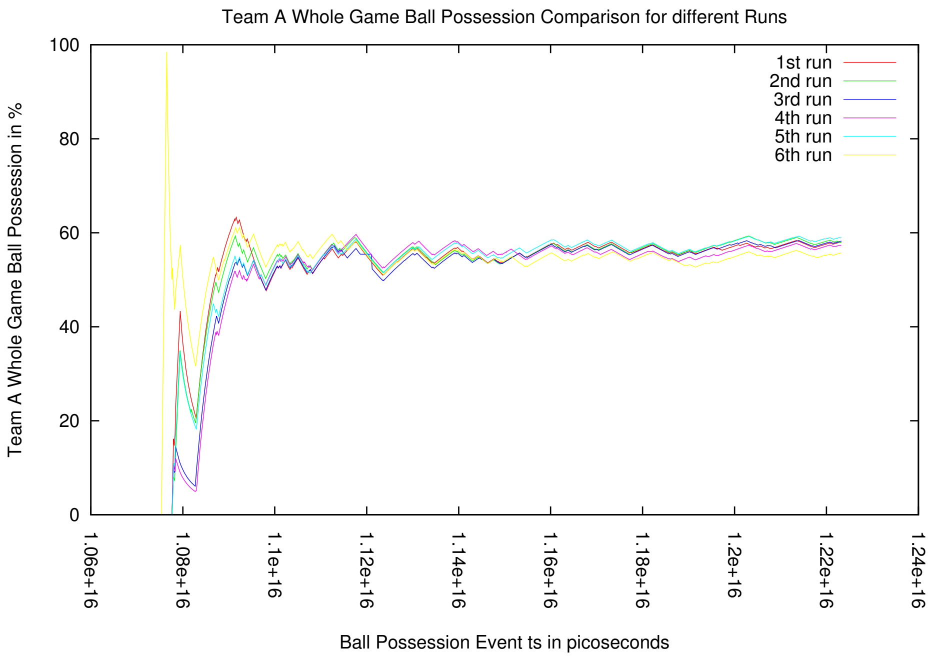


Figure G.17: Team A Whole Game Ball Possession Comparison for different Runs

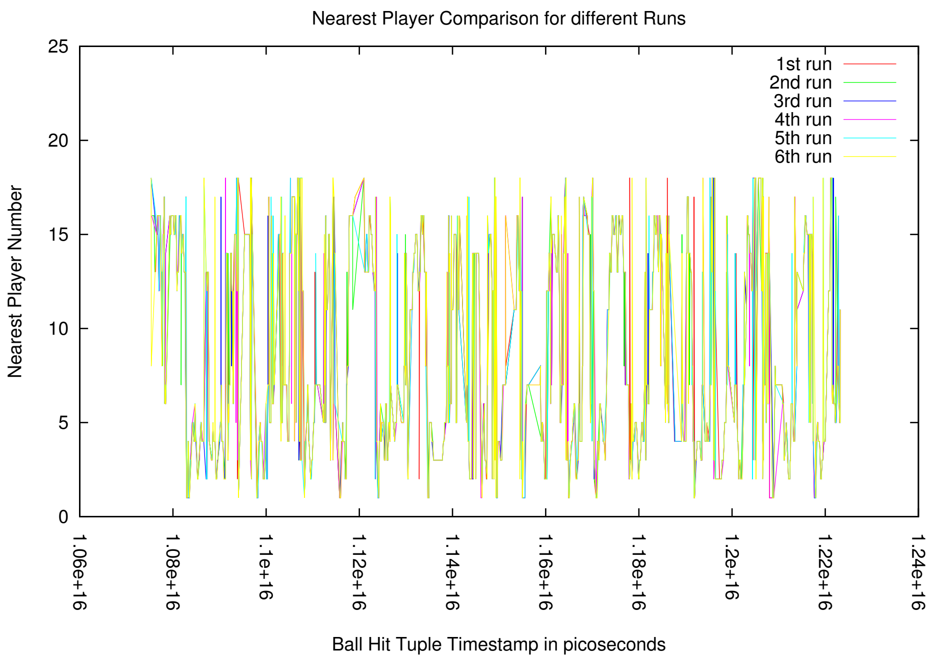


Figure G.18: Nearest Player Comparison for different Runs. 1-8 denote players A1-A8. 11-18 denote players B1-B8.

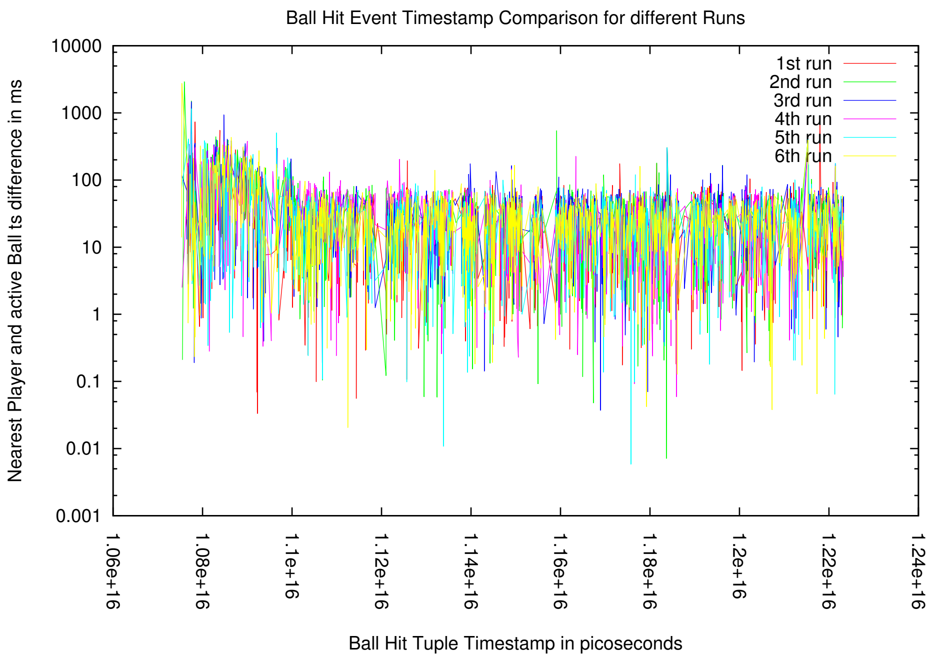


Figure G.19: Ball Hit Event Timestamp Comparison for different Runs

G.2.5 Stream Repeaters and Load Balancing

Evaluation data for Section 5.2.4.

G.2.5.1 Table

#Peers	#Clients	Avg	Var	Std	Min	Median	90 Perc.	99 Perc.	Max	#retrievedTuples
1	40	82.35	2227.59	47.15	19.9	75.9	116.85	225.45	1509.05	18180.25
	50	102.56	3452.91	58.68	20.85	90.75	163.25	283.95	1417.75	13949.00
	60	116.11	5739.48	75.71	21.1	103.6	181.6	320.35	2193.45	12331.45
	70	134.72	7100.49	84.23	22.35	118.85	221.65	379.9	1463.55	10371.65
	80	168.85	13844.52	117.46	22.7	147.7	302.2	451.85	2697.35	7712.00
	90	182.60	14975.37	122.29	22.15	158.6	332.1	526.85	1860.7	7023.50
	100	209.53	21754.16	147.31	22.6	183.7	381.25	597.9	4158.3	5918.00
2	40	72.58	608.61	24.50	22.8	71.4	94.2	138.25	641.3	25570.75
	50	73.47	2298.15	43.64	22.35	70.45	94.8	179.85	1605.8	24050.00
	60	100.24	4382.41	60.25	26.6	94.9	131.4	210.8	2853.5	20540.15
	70	105.76	5558.89	69.21	24.75	97.5	146.3	259.1	2412.05	17216.95
	80	120.16	9274.96	87.54	28.55	110.4	167.1	285.2	4241.55	15843.45
	90	146.35	15084.71	112.47	29.7	130.95	215.4	352.55	3416.55	12727.30
	100	159.74	18507.90	127.19	31.8	144.85	227.55	359.7	4567.9	12096.35
3	40	77.81	467.20	21.59	24.1	77.0	100.35	130.15	349.55	25621.30
	50	74.18	583.81	23.75	22.75	73.1	96.2	134.45	669.45	25739.05
	60	76.18	646.56	24.74	23.25	75.15	98.75	144.85	550.95	25320.05
	70	82.67	1375.52	35.76	23.5	79.5	105.15	205.4	899.6	24042.50
	80	97.16	3162.14	54.03	27.5	91.1	125.7	255.65	1661.9	21254.75
	90	110.23	2188.60	45.19	31.5	104.9	145.8	236.6	1230.05	19067.60
	100	118.80	4717.89	65.60	30.25	111.0	161.25	267.85	2217.2	17123.95

Table G.6: Statistics for *SENSOR105* in Increasing Number of Clients Evaluation. During the evaluation each statistic is measured at each of the 20 fixed query delay clients. The table contains the average over these 20 values.

G.2.5.2 Graphs

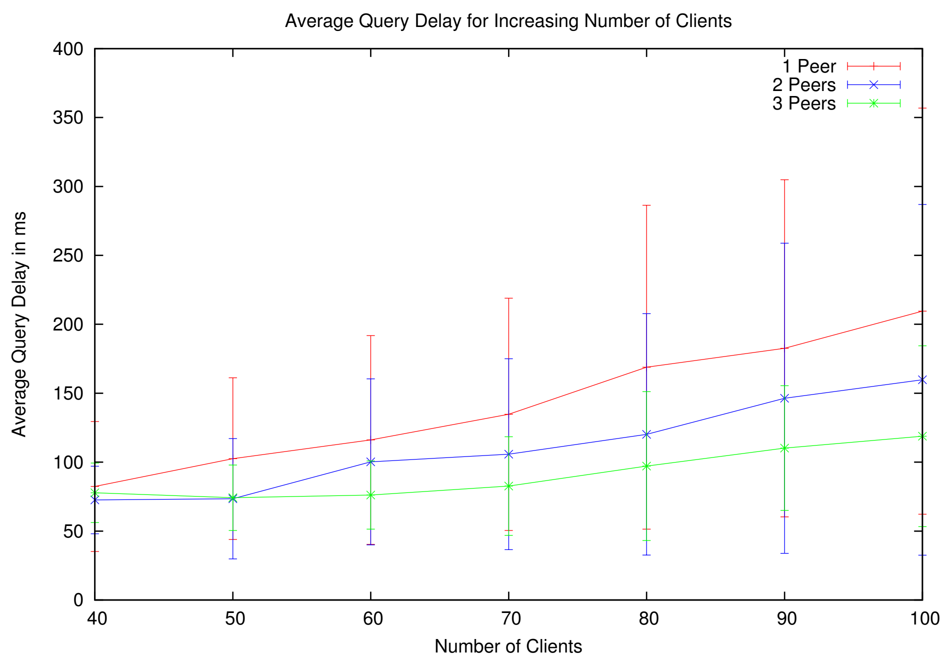


Figure G.20: Average Query Delay and Standard Deviation for Increasing Number of Clients

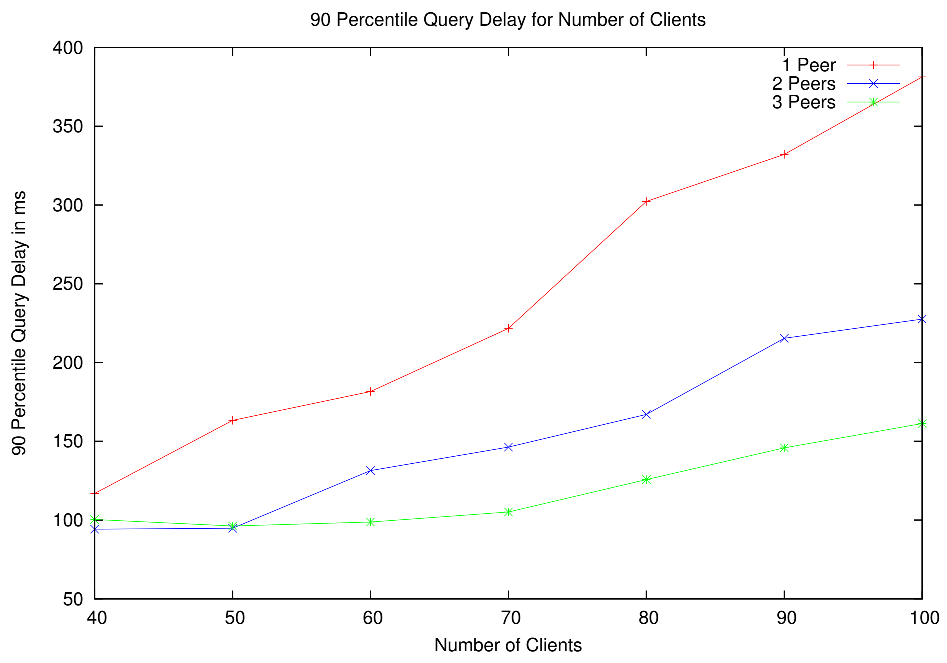


Figure G.21: 90 Percentile Query Delay for Increasing Number of Clients

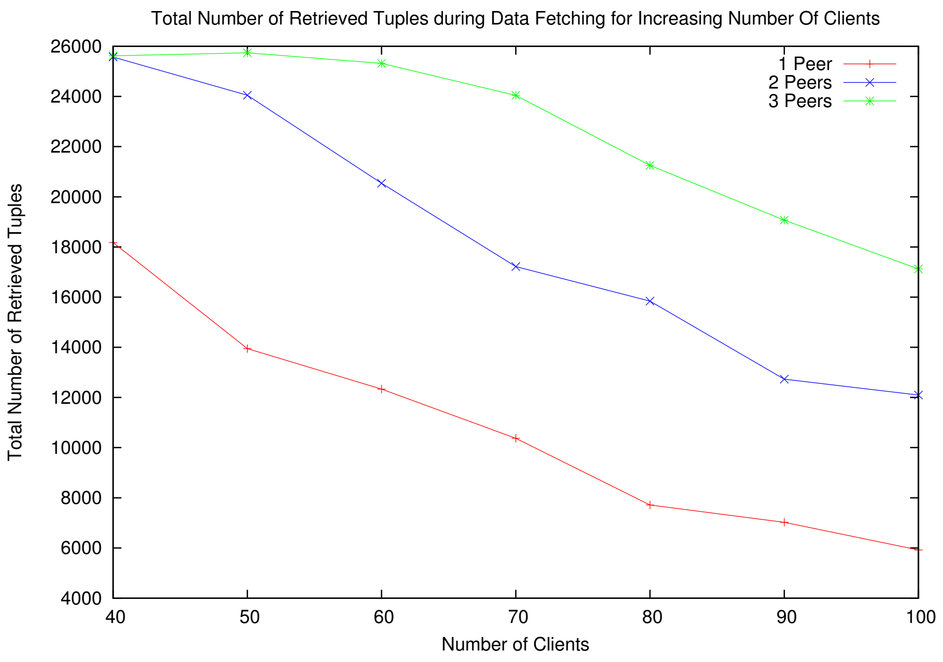
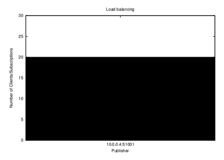
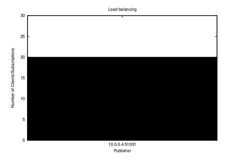
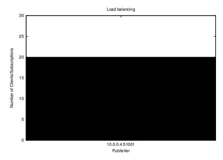
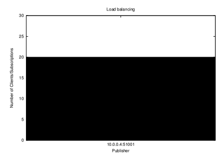


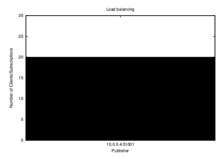
Figure G.22: Total Number of Retrieved Tuples during Data Fetching for Increasing Number of Clients

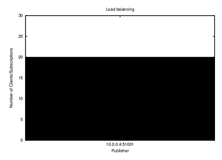

(a) 1 Peer, 40 Clients

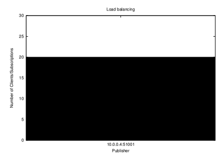

(b) 1 Peer, 50 Clients

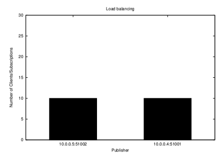

(c) 1 Peer, 60 Clients

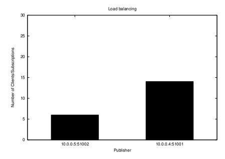

(d) 1 Peer, 70 Clients

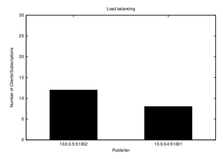

(e) 1 Peer, 80 Clients

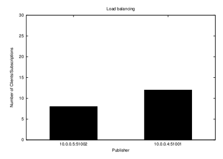

(f) 1 Peer, 90 Clients

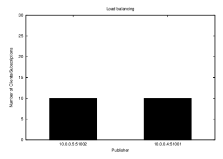

(g) 1 Peer, 100 Clients

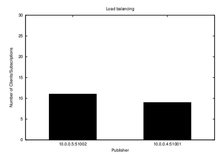

(h) 2 Peers, 40 Clients

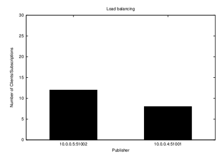

(i) 2 Peers, 50 Clients

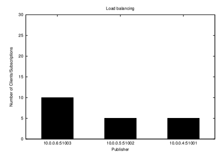

(j) 2 Peers, 60 Clients

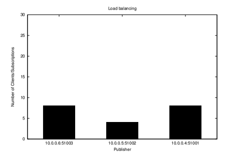

(k) 2 Peers, 70 Clients

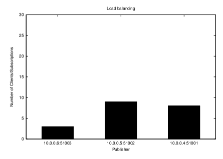

(l) 2 Peers, 80 Clients

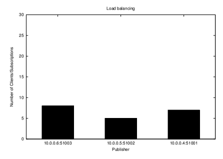

(m) 2 Peers, 90 Clients

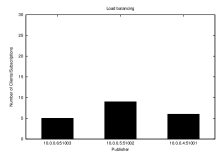

(n) 2 Peers, 100 Clients

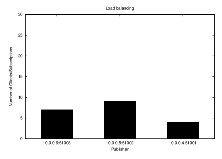

(o) 3 Peers, 40 Clients


(p) 3 Peers, 50 Clients


(q) 3 Peers, 60 Clients


(r) 3 Peers, 70 Clients


(s) 3 Peers, 80 Clients


(t) 3 Peers, 90 Clients

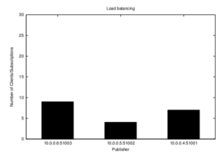

(u) 3 Peers, 100 Clients

Figure G.23: Load Balancing. Graphs show which publishers are used by the 20 fixed query delay clients for retrieving *SENSOR105*, i.e., how the workload of the 20 fixed clients is balanced between the available publishers in each run.

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

~~Bachelor's~~ / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:

PAN - A P2P Approach for Scalable Complex Event Detection in Distributed Data Streams

First Name, Surname *(Please print in capital letters)*: Lukas Probst

Matriculation No.: 09-050-402

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes [✓ No]

Place, Date: Basel, 08.08.2014

Signature: [illegible]

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI
BASEL