Pull-based Real-Time Complex Event Detection in Multiple Data Streams – the PAN Approach

Lukas Probst Ivan Giangreco Heiko Schuldt

Technical Report CS-2016-003
University of Basel, Switzerland
Email: {lukas.probst|ivan.giangreco|heiko.schuldt}@unibas.ch

Abstract

With the proliferation of embedded sensors, the number of data streams and the volume of streamed data has increased enormously. This has strongly influenced both our business and our private life and has brought forward a large variety of monitoring applications in different domains. In all these applications, the analysis of data streams in real-time is essential. One of the main challenges in data stream analysis is the detection of complex events out of the raw streaming data. In this report, we present PAN, a generic middleware for distributed real-time complex event detection (CED) which is able to analyze multiple distributed data streams. In PAN, CED applications are defined as workflows and are executed by dedicated workers in a distributed way in a P2P network. These workers use pull-based publish/subscribe for communication. This allows to dynamically extend analysis workflows at run-time and to balance the load between workers. As a consequence, it makes PAN scalable not only in terms of the number of data streams as well as the number and the complexity of the analyses but also in terms of the number of clients that retrieve the analysis results. Evaluations based on an extended version of the ACM DEBS 2013 Grand Challenge scenario show the effectiveness and efficiency of PAN.

This technical report is an extended version of our previous work [19] that has been published by Springer International Publishing in the Proceedings of the 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2016). It add details on the inner workings of PAN, especially on the pull-based publish/subsribe interaction between workers.

University of Basel

Department of Mathematics and Computer Science
Databases & Information Systems Group
Spiegelgasse 1, CH - 4051 Basel Switzerland. Website: http://dbis.cs.unibas.ch

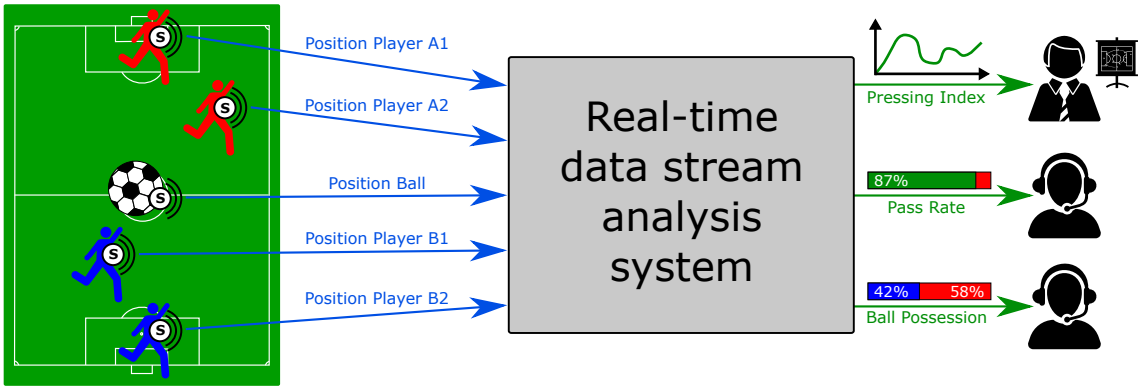


Figure 1: Soccer match scenario

1 Introduction

The last decade has seen a vast proliferation of both software and hardware sensors, embedded into smart phones and wearables. These devices sense their environment and, according to the IoT vision, most of them are connected to the Internet. Therefore, they are able to disseminate the data they measure in form of continuous data streams. Hence, the number of data streams and the volume of streamed data has increased enormously. Nevertheless, the analysis of these Big Data streams in real-time is essential. This applies not only to single streams, but also to multiple streams coming from different modalities. In particular, the detection of complex events out of the raw streaming data in real-time is a major challenge and at the same time an important aspect in a variety of applications, ranging from business to personal and entertainment applications.

As an example, consider a soccer match scenario as illustrated in Figure 1. Assume that all players and the ball are equipped with several wearable, unobtrusive sensors. Each sensor produces a continuous stream of data. These streams need to be analyzed to produce added value on a match for different stakeholders: coaches want to analyze tactical compliance (e.g., pressing index) at run-time, to give immediate feedback to their team, while broadcasters like to see statistics (e.g., pass rate or ball possession) to integrate into their live video. Hence, all users (clients) are interested in different events that need to be detected out of all the incoming streams. This requires an infrastructure that i.) allows to implement, in a modular way, basic components for detecting complex events (e.g., ball hits) and for performing other analysis operations (e.g., calculate average positions), and that ii.) supports the combination of these components into workflows in order to hierarchically combine detected events to more high-level events (e.g., detect passes in a sequence of detected ball hits). Moreover, the infrastructures has to iii.) scale with the number of streams, with the number of clients that retrieve the analysis results and with the number and complexity of the analyses, and iv.) allow for easy re-configuration of workflows when the information need of clients changes.

In this report, we introduce PAN (*P*2P *A*nalysis *N*etwork), a generic distributed real-time complex event detection (CED) middleware which jointly addresses these challenges. PAN is able to analyze multiple distributed input data streams and to concurrently handle several analysis requests of different clients (see Figure 2). In PAN, CED applications are

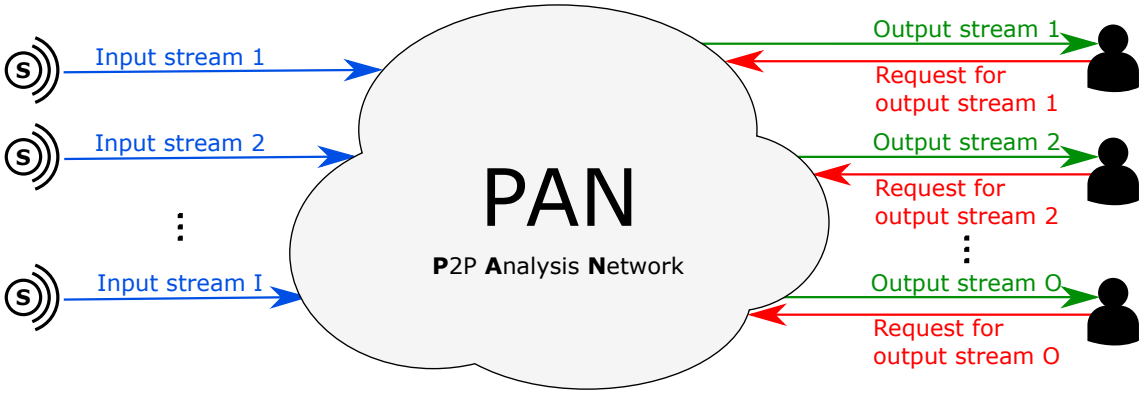


Figure 2: CED with I input streams and O output streams

defined as workflows consisting of workers whose internal components implement complex event detectors or other analysis operators such as aggregators. These workflows are executed in a distributed way in a P2P network based on a pull-based publish/subscribe communication between workers. A main feature of PAN is the high degree of scalability it can achieve (w.r.t. concurrent data streams, client requests, and complex event types) and its dynamics which allows to extend or re-configure CED workflows at run-time.

The first generation of CED systems, also called complex event processing (CEP) systems, has been built with a centralized architecture (e.g., [1, 11, 14]). This significantly limits their scalability, especially if complex events have to be detected in real-time. If the number of sensors (and thus input streams) becomes too high and/or if the algorithms for event detection become too complex, the computational effort for performing all analyses on all incoming streams will eventually impact the performance of the entire system. The same is true if the number of client requests to be served concurrently significantly increases.

More recent approaches to CED such as [4, 5, 9, 23] use a distributed architecture and forward streams in a publish/subscribe style between workers that perform parts of the complex event detection. However, this is mostly done in a push-based style where each worker directs its output stream(s) to the subscribers, i.e., to downstream workers or clients. This, in turn, limits the dynamics that can be achieved as a publisher of a stream has to know all the subscribers in order to send new data stream elements to them. Thus, additional communication is necessary to properly link new workers or clients to existing workflows. The pull-based interaction of PAN, in contrast, significantly facilitates the re-configuration of workflows – such as adding new clients as sinks to existing workflows or even extending existing workflows with new workers that generate new data streams for new clients. Moreover, the pull-based approach also allows to easily integrate new workers to balance the load of the client requests.

The contribution of this report is twofold. First, we present PAN, a novel middleware architecture for distributed, scalable, and flexible CED that seamlessly combines ideas from workflow management (definition of CED workflows), P2P systems (distributed, scalable CED), and pull-based publish/subscribe interactions (load balancing and dynamic re-configuration of CED workflows). Second, we provide the results of an evaluation of PAN's performance and scalability characteristics on the basis of a sports use case (CED in

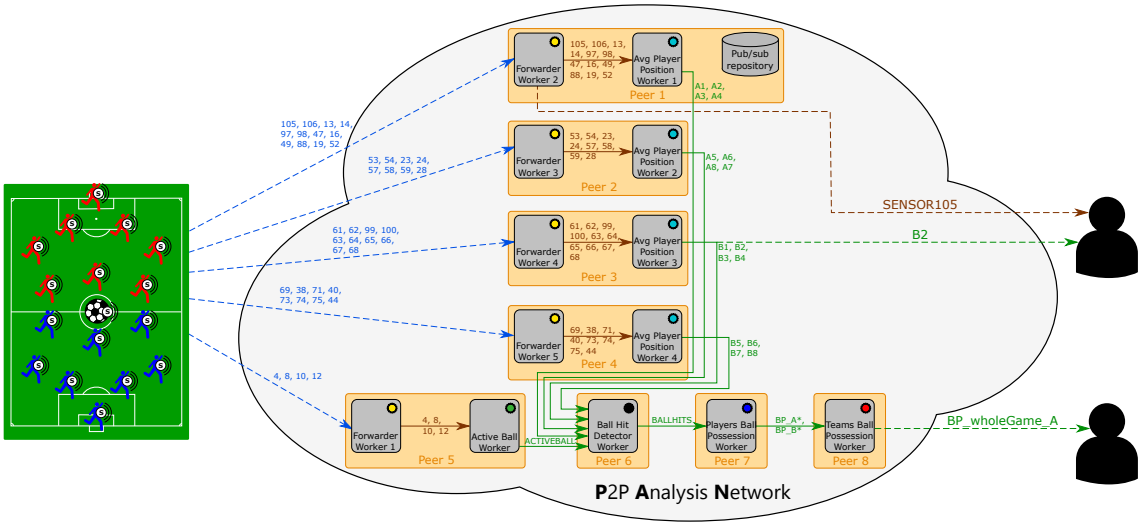


Figure 3: Sample CED workflow in PAN for soccer match analysis. The initial input streams are taken from the ACM DEBS 2013 Grand Challenge [10, 17]. For illustration purposes, multiple streams between a sender and a receiver are depicted with a single arrow.

soccer match analysis) using an extended version of the ACM DEBS 2013 Grand Challenge scenario [10, 17]. The results show the effectiveness and efficiency of the PAN approach and the gain that can be achieved by dynamic load balancing.

This technical report is an extended version of our previous work [19] that has been published by Springer International Publishing in the Proceedings of the 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'16). It add details on the inner workings of PAN, especially on the pull-based publish/subsribe interaction between workers.

The remainder of this report is organized as follows: We introduce PAN in Section 2 and present details on its implementation in Section 3. In Section 4, we report on the evaluation of PAN. Section 5 presents related work and Section 6 concludes.

2 PAN

In this section, we present and discuss the concepts of PAN. The main idea behind PAN is to obtain a high degree of scalability and flexibility for real-time CED applications by distributing the workload across several peers in an unstructured P2P network and communicating via pull-based publish/subscribe.

2.1 CED Workflows in PAN

In PAN, CED applications are defined by means of *workflows*. They consist of so-called *workers* which provide basic functionality for CED and which are combined using a partial order that allows both sequential and parallel execution of workers, depending on the semantics of a concrete CED application. Figure 3 illustrates a sample workflow which

generates the player as well as the team ball possession statistic streams for the soccer use case (details on how the workflow generates these statistics in multiple steps are given in Section 4.1). This workflow includes several intermediate streams, i.e., streams that are generated as output streams by some workers and consumed as input streams at other, subsequent workers. For instance, the ball hits stream (*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. The *sensor devices* producing the initial input streams are sources of a workflow. Moreover, there are devices outside PAN called *clients* that consume the output streams of the workers. The clients are the sinks of a workflow and might join it only for a short time.

Each worker is hosted on a *peer*, which is either a physical machine or a Cloud instance. Each peer, in turn, can host a single or multiple workers. With this design, PAN obtains a high degree of flexibility in terms of workflow distribution and deployment. On the one hand, the entire workflow can be executed on a single machine (when hosting all workers on a single peer) if this machine is powerful enough, or the workflow has a small workload. On the other hand, a workflow can also be fully distributed onto a large number of (computationally weak) peers hosting workers which compute only small subtasks. Such flexibility is fundamental since PAN is designed as a generic CED middleware and thus should be applicable to a large variety of use cases and deployment scenarios.

To standardize the interaction between workers in PAN, they only share data via network communication, independent from their deployment (i.e., even if they are hosted on the same peer).

2.2 Pull-Based Publish/Subscribe in PAN

The workers of a CED workflow have to communicate to deliver data streams along the workflow. Hard-wiring the communication between the workers would lead to a highly inflexible system, earmarked for a specific workflow. In contrast, the publish/subscribe style of interaction allows to decouple the sender of a data stream and its receiver. Each worker simply has to publish all its output data streams, and these streams can be subscribed by other workers and clients. This enables using the same workers for different workflows and facilitates modifying and extending workflows at runtime.

In contrast to most other publish/subscribe-based CED systems which communicate in a push-based style, PAN uses a *pull-based* approach. In this approach, the CED workflow is basically defined from the sink(s) to the source(s). While in the push-based approach a publisher is responsible for pushing newly generated data stream elements to all subscribers, in the pull-based approach a subscriber is responsible for fetching the data from the publisher. As a consequence, a publisher does neither have to keep a list of all subscribers of its output streams nor push new data stream elements to all of them.

The major advantage of pull-based over push-based publish/subscribe is that the former is more dynamic and flexible. First, the pull-based approach enables subscribers to fetch data with different frequencies or even on demand. If a subscriber (i.e., a worker or a client) only needs new data stream elements every second, it does not have any benefit from immediately receiving all data stream elements which are produced with 100 Hz by the publisher. Instead the subscriber can send a new data fetch request every second to retrieve all new data stream elements and thus reduce the computational load at the

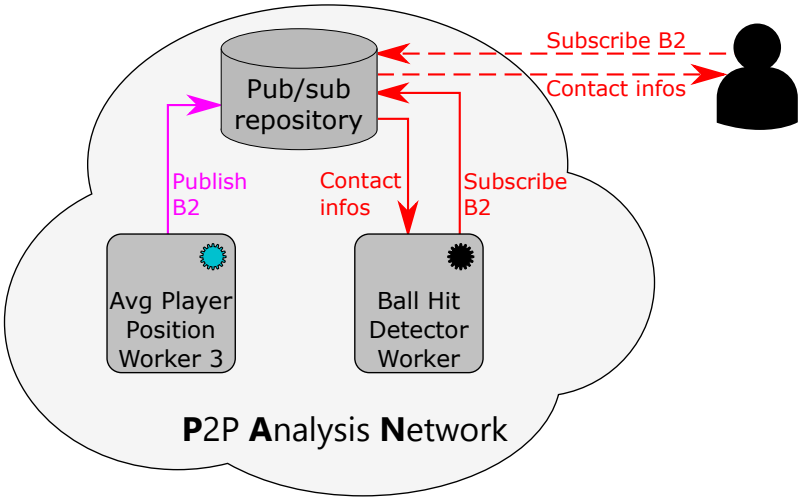


Figure 4: Publish/subscribe interaction in PAN

publisher. If additionally only the latest data stream element is required every second, the subscriber can issue a data fetch request for only the latest data stream element and thus further significantly reduce the network traffic. We have used this feature to implement the client requests in a very efficient way. Second, the pull-based approach facilitates the dynamic adaptation of the CED workflow at run-time as the publishers do not need to know their subscribers. Thus, the workflow can be modified by adding new workers and clients without changing anything at or informing the upstream workers. In consequence, pull-based publish/subscribe further facilitates load balancing by adding new workers, so-called Repeater Workers, during run-time if the number of client request grows and thus promotes PAN's ability to scale w.r.t. the number of clients (see Section 2.4).

However, while in the push-based approach a publisher distributes new data stream elements immediately to the subscribers after having produced them, pull-based interactions come with a delay between the generation of a data stream element at the publisher and the pull initiated by the subscriber. Minimizing this delay would be very costly as it would require the subscribers to frequently issue fetch operations in order to pull new data stream elements from the publisher, leading to high network consumption. In contrast, larger intervals reduce the network consumption and the computational effort, but increase the delay. Finding the optimal fetch interval depends on the application (i.e., on the expected velocity of events). It may also be different for every individual client and worker, even for the same streams. Nevertheless, the flexibility provided in PAN by the pull-based approach outweighs the delay in stream distribution as the latter can be subject to an optimization which jointly considers timeliness and the costs for higher network traffic. Moreover, note that a too large fetch interval does not result in lost data stream elements at the workers since they fetch all new data stream elements instead of only the latest one (unless the number of new stream elements exceeds the capacity of the local ring buffer of the worker).

PAN uses a central publish/subscribe repository which stores a mapping from the stream identifier to a list of publishers. The ability to store multiple publishers per stream is required for load balancing purposes. When a new worker is deployed, it has to publish

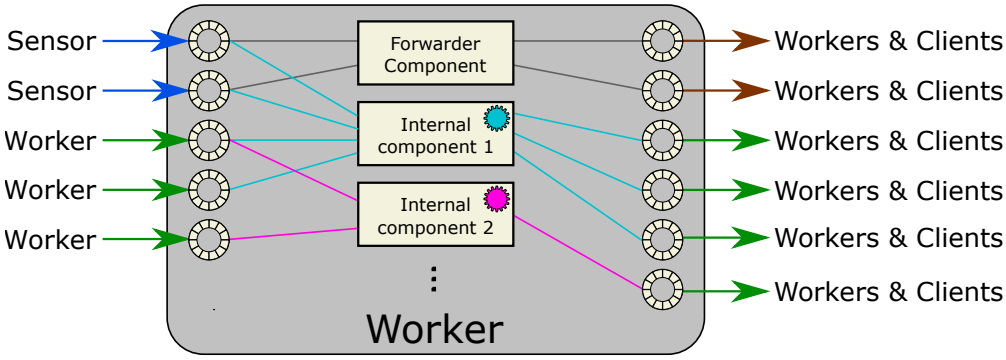


Figure 5: Architecture of a PAN worker

all its output streams. Subsequently, all potential subscribers (clients or other workers) can use this information, as illustrated in Figure 4, to identify the publisher and pull the data stream from there. Note that the repository has to be contacted only once, when the link between subscriber and publisher is established; it does not need to be queried at run-time once the stream between both parties has been established. Hence, the central repository does not become a bottleneck. In contrast to this, a broker approach is used in many push-based publish/subscribe systems and thus also in some CED systems with a push-based communication style to eliminate the need to store a list of all subscribers at the publisher. This approach thus enables some of the flexibility features PAN naturally provides, but introduces a performance bottleneck and thus potential scalability issues since the brokers have to forward every transferred data stream element.

2.3 PAN Workers

A PAN worker is a building block for the CED workflows. At the interface, each worker generates one or several output stream(s) on the basis of one or several input stream(s) it consumes – either directly from a sensor or from other workers. The input streams are processed by one or several components inside the worker. An internal component implements a simple forwarder, a complex event detector, or another analysis operator such as an aggregator. The output streams contain the analysis / complex event detection results of these components.

PAN uses separate ring buffers to handle a worker's input and output streams and thus to connect two workers (see Figure 5). A worker's input ring buffers contain the latest data stream elements of its input streams while the output ring buffers are filled with the data stream elements created by the worker's components, i.e., with its output data stream elements. Each worker runs a server to enable subscribers (i.e., downstream workers and clients) to fetch elements of the output data streams it publishes. More precisely, the server answers data fetch requests with the corresponding output data stream elements that are stored in its output ring buffers. In return, a worker can fetch new data stream elements from a publisher either periodically in a fixed interval or on demand, upon request by an internal component, to fill its input ring buffers. The advantage of the fetch-on-demand variant is that new data stream elements are only retrieved when they are needed and thus there is no unnecessary computational load at the publisher for handling more fetch

requests than necessary. However, if a component within a worker requests a fetch-on-demand, it will be in idle wait until the fetch request is completed.

In contrast to the pull-based communication among workers and between workers and clients, each sensor always pushes all its data to only one of the first workers of a CED workflow which then publishes the sensor data stream for all other workers and clients. For this purpose, every worker that receives input data streams from external devices runs a generic FORWARDER COMPONENT that simply forwards these streams (see Figure 5).

All generic and application-specific components of a worker run in parallel. A component can use all input data streams for its analysis task, i.e., it is able to read elements from all input ring buffers. Moreover, a component is able to generate one or multiple output data stream(s) and hence fill the corresponding output ring buffer(s). However, a worker's components are strictly separated from each other. They neither share state, nor can they directly communicate with each other. If a component needs the result of another component, they have to be deployed on different workers (even when running on the same peer), and to be connected via their output and input buffers, respectively. The benefit of this design choice is that it supports creators of CED applications in implementing workers in a parallel, modular and standardized (w.r.t. communication) way.

2.4 Scalability

Increasing the number of sensor data streams to be analyzed, the number of different analyses that have to be performed, or the complexity of these analyses results in an increased computational effort. However, PAN enables handling overload situations by distributing the workers of a CED workflow on more peers or even by distributing the overall workload across more workers which can then be deployed on peers with free capacities. In consequence, PAN scales w.r.t. the number of data streams and w.r.t. the complexity and the number of analyses.

Moreover, also an increasing number of clients that retrieve the analysis results, i.e., that fetch output data stream elements from the workers, can cause performance problems. For instance, the peer which hosts the worker that generates the team ball possession output stream of team A (*BP_wholeGame_A*) might not be capable of handling all fetch requests for elements of this stream, especially if hundreds or thousands of clients are interested in this stream. However, PAN can solve this problem and thus achieve a high degree of scalability w.r.t. the number of client requests by exploiting several redundant workers, if available.

For this, PAN uses generic REPEATER COMPONENTS. These are components that simply forward a given set of input data streams (published by another worker) without altering them. Using the REPEATER COMPONENT, a worker is able to subscribe to an output data stream of another worker, store the data stream elements in its own output ring buffer, publish the exact same stream and thus add itself as an additional publisher to the publish/subscribe repository. The data fetch request load can then be balanced by distributing the subscribers on the publishers. Due to the flexibility of the pull-based approach, new REPEATER WORKERS that solely contain a REPEATER COMPONENT can be added at run-time whenever needed. Hence, it is possible to monitor the load in the system and dynamically add new REPEATER WORKERS if the existing publishers cannot handle all requests.

3 Implementation

We have implemented PAN fully in Java in order to support hosting workers on different platforms. Moreover, all communication between PAN workers, clients, and PAN's publish/subscribe repository is done via REST interfaces with JSON objects. This includes fetch operations for output data stream elements of workers as well as publications and subscriptions of streams. For this purpose, each worker and the repository run a web server to handle these operations using the Jetty library[1]. Furthermore, the workers use the Jetty library during runtime to issue operations and receive the results via HTTP requests. Java objects are converted to JSON objects with the Google Gson library[2]. The main benefit of using REST interfaces for data transfer is that they are language independent. Hence, it is possible to write clients in different languages as long as they are able to perform HTTP requests. However, the REST interfaces introduce avoidable communication overhead (due to the HTTP header and the JSON syntax) and computational effort for running the web server. Especially in scenarios with high-frequency streams and data-intensive workflows running on resource-constraint peers, this could become a bottleneck. Therefore, we plan to facilitate using binary JSON and a more lightweight web server implementation in our future work.

In order to create a CED application that runs on top of PAN, one solely has to implement the workers (including their internal components) in Java if they do not already exist and to specify the workflow in a JSON config file, similar to TechniBall's XML approach [12]. The config file defines various parameters for each worker, such as its type, the host (i.e., the peer), the expected sensor data streams for the Forwarder Component (that is automatically added if necessary) and optionally some type-specific parameters (e.g., the set of streams forwarded by a Repeater Worker). Moreover, it defines the host of the publish/subscribe repository. Appendix B shows a sample config file for the workflow illustrated in Figure 3. The config file is used to automatically deploy the CED system. The actual connection between the workers is done at start-up time using the publish/subscribe repository.

4 Evaluation

In this section, we present the results of a series of evaluations of PAN with an extended version of the ACM DEBS 2013 Grand Challenge scenario [10, 17].

4.1 Set-Up and Metrics

In our evaluation each peer is deployed as an instance in the Microsoft Azure Cloud platform[3]. Both clients and (simulated) sensors are deployed on a separate physical server[4], outside of the Azure Cloud. The operating system of the physical machine as well as of the Cloud instances is Ubuntu 12.04 (LTS). The ping between two peers within the Cloud

[1] Jetty: `http://www.eclipse.org/jetty/`
[2] Google Gson: `https://code.google.com/p/google-gson/`
[3] Small VM instances (standard A1), 1 core 1.6 GHz CPU, 1.75 GB RAM
[4] Lenovo ThinkPad W530, Intel Core i7-3820QM CPU @ 2.70 GHz, 12 GB RAM

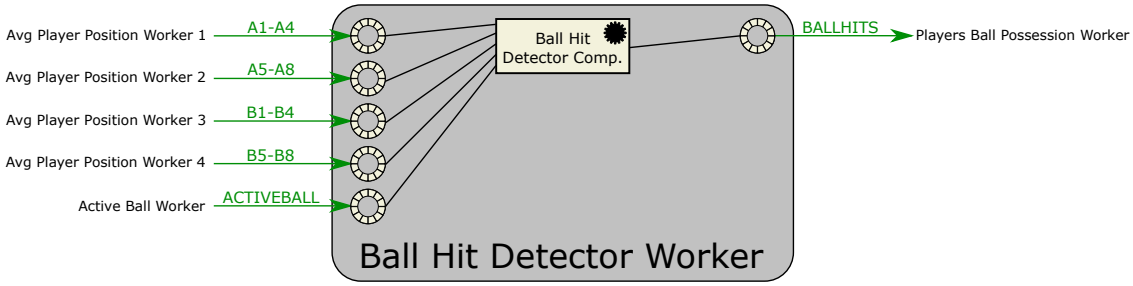


Figure 6: Ball Hit Detector Worker. For illustration purposes we have depicted only a single arrow and ring buffer for all streams emitted by the same sender. In fact, there is a single input ring buffer for each input data stream.

is approximately 0.9 ms and the ping from the physical machine to a peer in the Cloud is around 21 ms.

For evaluation purposes, we have implemented a soccer match analysis application that answers a subset of the queries specified in the ACM DEBS 2013 Grand Challenge [10, 17]. More precisely, we have implemented the required workers as well es their internal components and specified a CED workflow (depicted in Figure 3) that generates ball possession streams for the players and for the teams in multiple steps. The input data stream of the Grand Challenge reflects the measurements of sensors that are attached to the shin guards of the players and inside the ball and include position, timestamp, velocity and acceleration information. We have used the first 25 minutes of the (short, 8 vs. 8) soccer match that has been captured for the Grand Challenge in our evaluation. However, we have extended it by generating a single input data stream for each sensor instead of only one input data stream that contains the measurements of all sensors in order to demonstrate PAN's ability to handle multiple distributed input data streams. As a first step of the CED workflow, a set of Forwarder Workers receives the sensor data streams and publishes them for all workers and clients. Subsequently, a set of Average Player Position Workers generates a single position data stream for each player. More precisely, every Average Player Position Worker contains four Player Average Components that run in parallel.[5] Each of these components aggregates the sensor data streams of a player to a single one (e.g., *B2*) by periodically calculating the average of the values stored in the latest data stream elements of all sensors corresponding to the player and generating a new output data stream element with the average values. Moreover, there is an Active Ball Worker that generates a data stream with the current position of the active ball (*ACTIVEBALL*). The Ball Hit Detector Worker illustrated in Figure 6 consumes the average position data streams of all players (i.e., *A1-A8* and *B1-B8*) and the active ball stream to detect ball hits. A ball hit is detected when the ball acceleration is greater than 55 m/s² and the closest player's distance to the ball is lower than 2.5 m. In this case the Ball Hit Detector Component generates a new data stream element for the *BALLHITS* stream that reflects a ball hit of the closest player. Subsequently, the Players Ball Possession Worker uses this stream to generate a stream for every player (e.g.,

[5] Average Player Position Worker 1 additionally contains a fifth Player Average Component that generates a position data stream for the referee (*REFEREE*). This stream is not shown in Figure 3 as it is not used to generate the ball possession streams.

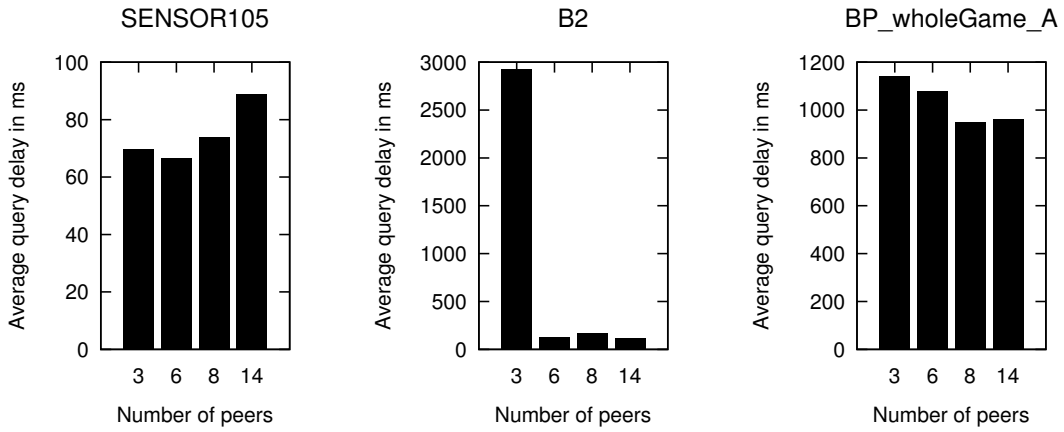


Figure 7: Average query delay for *SENSOR105*, *B2* and *BP_wholeGame_A* with increasing number of peers

BP_B2) whose latest element contains the number of milliseconds the player has been in possession of the ball since the match has started. Finally, the Teams Ball Possession Worker consumes all these player ball possession streams, aggregates the therein contained information and generates output data streams with team ball possession statistics for different time windows (e.g., *BP_wholeGame_A*).

In addition, we have defined simple workflows with a single Forwarder Worker that receives a sensor data stream and 0 to 2 Repeater Workers that act as additional publishers for this sensor data stream (see Appendix A.2).

PAN's performance is measured using the *query delay* that indicates how long the system needs to calculate and generate a certain output data stream (and thus to answer a client query). It is defined as the difference between the machine time when receiving an output data stream element at the client $MT(c)$ and the machine time $MT(s)$ at which the corresponding sensor data stream element has been emitted. Note that the query delay measures not only the time PAN needs for complex event detection but also comprises the time for sending the sensor data stream to the first worker and for fetching the output data stream from the last worker of the CED workflow.

4.2 Effects of PAN Size

The first evaluation series analyzes PAN's performance by varying the number of peers hosting the workers of the soccer match analysis workflow. We use four different deployments with 3, 6, 8, and 14 peers for the workers and the publish/subscribe repository (see Appendix A.1). The Jetty client periodically (every 20 ms) fetches the latest element of three data streams, produced at different positions in the workflow: a forwarded sensor data stream (*SENSOR105*), an intermediate output data stream (*B2*), and a final output data stream (*BP_wholeGame_A*).

Figure 7 illustrates the results of this evaluation. While the average query delays of the *SENSOR105* and the *BP_wholeGame_A* streams are rather constant, the query delay of the *B2* stream is approximately 20 times higher in the three peers setting (2923.73 ms) than

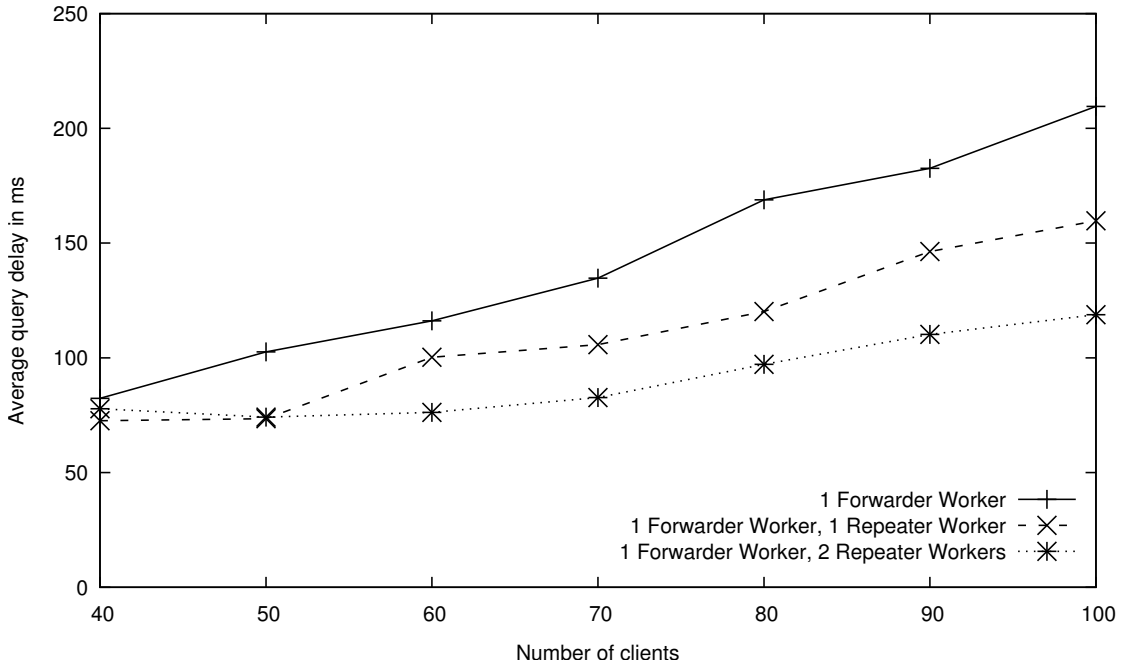


Figure 8: Average query delay for *SENSOR105* with increasing number of clients

in the other settings (123.99 ms, 165.68 ms, and 114.72 ms). The reason for this is that the two peers that are supposed to be responsible for generating the average player position streams in the three peers deployment (see Appendix A.1.1) are not capable of doing so. In contrast, distributing the four Average Player Position Workers on four peers yields a good performance. Thus, the evaluation shows that PAN enables eliminating such computational bottlenecks by distributing the workflow onto more peers. Moreover, the evaluation shows that distributing the workflow does not inflate the query delay. In consequence, our first evaluation series confirms that PAN is able to scale w.r.t. the number of input data streams to be analyzed as well as with the number and complexity of the analyses that have to be performed.

4.3 Load Balancing

In what follows, we evaluate PAN's load balancing capabilities by analyzing how the system scales w.r.t. an increasing number of client requests and how the scalability can be improved by distributing client requests to multiple workers.

In order to reduce the risk of side effects that may corrupt the evaluation results and thus might even conceal the real trends, we use simplified workflows that only receive a single input data stream from a sensor (*SENSOR105*) and publish this stream to clients outside PAN. These workflows are illustrated in Appendix A.2. They differ solely in the number of workers that repeat the stream. Due to these repeaters, the clients' accesses to the stream can be distributed among multiple peers. For the evaluation, we increase the number of clients up to 100. 20 Jetty clients which measure the query delay are constantly running on the physical server and 20 to 80 lightweight Python clients are deployed on ten

Cloud instances that are not used at the same time for PAN workers. Each client fetches the most recent data stream element from its publisher every 20 ms.

Figure 8 shows the evaluation results for the three deployments. For each setting, the query delay increases with an increasing number of clients. Thus, as expected, PAN's performance decreases if the load introduced by the client requests increases. At the same time, it can be seen that the more peers publish a stream, the better is the performance, as the load can be balanced. Hence, the evaluation results confirm that PAN's load balancing feature enables handling large client numbers and thus that PAN can scale w.r.t. the number of clients.

5 Related Work

As Pietzuch et al. have shown in [18], every distributed content-based or even topic-based publish/subscribe middleware can be transformed into a distributed CED middleware by distributing the steps of the detection process to the publish/subscribe brokers. This is equivalent to linking workers that perform parts of a CED process by means of a publish/subscribe system as we do in PAN.

Besides PAN, there are multiple other CED middlewares that distribute their detection components in a P2P network and that link these components using publish/subscribe. However, to the best of our knowledge, PAN is the first CED middleware that does so in a pure pull-based way. In contrast, RACED [5] follows a hybrid approach that supports both pull-based and push-based data transfers between detection components. This is beneficial for optimizing the network traffic and the delay when answering a query of a single client. However, since in RACED the data stream requested by a client has to be generated along its shortest path tree (SPT), clients cannot share a workflow if they do not have the same SPT while in PAN no such duplication is needed (unless this is done on purpose for load balancing). In [9] the authors of RACED propose a single-tree deployment strategy for the T-REX middleware [7] that allows workflows to be shared by clients such that the same output stream does not have to be generated multiple times.

In general, two types of CED middleware systems exist: *language-based* systems and *worker-based* systems. In language-based CED systems, such as, for instance, Amit [1] or Cayuga [11], complex events are defined in an abstract definition language. In worker-based CED systems, complex events are defined by means of implementing workers in a Turing-complete programming language. While PAN is a worker-based CED middleware, RACED uses a language-based CED middleware approach and thus suffers from limitations when it comes to defining arbitrary complex output streams (e.g., RACED cannot generate the team ball possession statistic streams of our soccer match analysis application since it does not support aggregates). Moreover, even though T-REX is based on TESLA [6], a more expressive language for defining complex events than RACED, it still suffers from some limitations as the client can neither define complex calculations nor small programs that have to be performed in order to detect a complex event or to generate the corresponding output data stream element.

In theory, language-based CED middleware systems which support user-defined aggregates enable defining all possible complex events [8, 15]. However, doing so may result in lengthy, complicated and unreadable definitions. Worker-based CED middlewares like

PAN, OSIRIS-SE [4] or Storm [23], in contrast, facilitate the implementation of arbitrary workers in a modular way and thus alleviate such problems as distribution of complex CED applications to more basic workers reduces the inherent complexity.

OSIRIS-SE [4] is a distributed worker-based CED middleware that has been built on top of the distributed workflow engine OSIRIS [21]. OSIRIS leverages global repositories to execute a static (predefined) workflow at several distributed service providers in a P2P style. Besides scalability, OSIRIS-SE's main focus is reliability. Since it leverages a push-based publish/subscribe approach, CED applications based on OSIRIS-SE may achieve a higher throughput than those based on PAN, but are at the same time less flexible.

Storm [23] is another infrastructure for processing data streams in real-time that can be categorized as a distributed worker-based CED middleware. Storm's spouts are equivalent to PAN's Forwarder Workers and Storm's bolts are equivalent to arbitrary PAN workers that perform any analysis. However, the communication in Storm is just the other way round as in PAN. In PAN workers communicate via pull-based publish/subscribe while in Storm bolts and spouts push their output data stream elements to downstream bolts. Moreover, in PAN sensors push their data stream elements to the Forwarder Workers while in Storm the spouts have to pull the input data stream elements. We argue that while pulling the input data stream elements from web APIs (e.g., Twitter streams) is unproblematic real-world sensors might not be able to cache measurements and in consequence cannot support pulling. Another difference between PAN and Storm is that Storm automatically distributes the workload (i.e., the spout and bolt instances) on the peers while the worker-to-peer allocation in PAN has to be specified manually in the workflow config file. Although, an automatic distribution might be desirable in many scenarios this can again be problematic if the input data streams are produced by sensors which are only able to push their streams to a specific predefined location.

The PAN middleware that requires each sensor to push its data stream only to a single worker and that allows the other workers to pull the sensor data stream elements from there is similar to the architecture proposed in [2]. However, in [2] the nodes that receive the data streams from the sources are just proxies that do not perform any analysis and the complex event detection is completely performed on a single node. In consequence, [2] is actually a centralized CED system.

The DHEP approach [20] connects full-fledged centralized CED systems to a distributed CED system with all features of the centralized CED systems. PAN, in contrast, combines small workers to CED workflows. However, both approaches have in common that they are flexible during runtime.

In reply to the ACM DEBS 2013 Grand Challenge [10, 17], six systems have been proposed [3, 12, 13, 14, 16, 24]. While the workers and CED workflows of PAN are based on the requirements of the challenge and thus have some similarity with all these systems, PAN's architecture is mainly influenced by the approach of Jergler et al. [14] that proposes a workflow-based architecture for CED in which different workers are connected with non-blocking ring buffers. While [14] states that the concept can in general be implemented in a distributed way using publish/subscribe, only a centralized implementation is presented. Hence, PAN fills a void as it promotes these concepts to a distributed and thus scalable system. Moreover, since the system presented in [14] has only a single input buffer, it cannot handle and analyze multiple input data streams. Furthermore, [14] is not able to answer multiple and arbitrary requests from different clients in parallel.

6 Conclusion

In this report, we have introduced the distributed real-time CED middleware PAN. It uses workflows to define CED applications and distributes the workload onto multiple workers that are hosted by peers in a P2P network. Using a pull-based communication model enables dynamic re-configuration of the CED workflow at runtime. The worker-based approach allows to implement parts of CED workflows in a modular way, ranging from simple stream forwarding to highly sophisticated analyses. Evaluations have shown that PAN enables eliminating computational bottlenecks by distributing the workflow on more peers and that its load balancing feature enables PAN to scale w.r.t. the number of client requests.

In our future work, we plan to organize the workers in a structured P2P network (i.e., a distributed hash table such as Chord [22]) to store the mapping of streams to publishers in a distributed and reliable way. Moreover, we plan to automate the worker-to-peer allocation that so far has to be specified manually in the workflow config file while still preserving the possibility to manually specify the location of a subset of the workers if necessary. In addition, we plan to further analyze, evaluate and compare different approaches to publish/subscribe-based communication in CED middlewares, in particular a pull-based vs. a push-based communication approach.

Image Credits

Several figures in this report use icons designed by Freepik (`http://www.flaticon.com/packs/humans-3`). The soccer field is a Wikipedia graphic designed by NielsF, available at `https://de.wikipedia.org/wiki/Datei:Offsidelarge.svg`.

References

[1] A. Adi and O. Etzion. Amit - the situation manager. *VLDB Journal*, 13(2):177–203, 2004.

[2] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based Complex Event Detection across Distributed Sources. *Proceedings of the VLDB Endowment (PVLDB 2008)*, 1(1):66–77, 2008.

[3] S. Badiozamany, L. Melander, T. Truong, X. Cheng, and T. Risch. Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-defined Aggregate Functions. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS 2013)*, pages 325–330, Arlington, TX, USA, 2013. ACM.

[4] G. Brettlecker and H. Schuldt. Reliable distributed data stream management in mobile environments. *Information Systems*, 36(3):618–643, 2011.

[5] G. Cugola and A. Margara. Raced: an Adaptive Middleware for Complex Event Detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware (ARM 2009)*, Urbana Champaign, IL, USA, 2009. ACM.

[6] G. Cugola and A. Margara. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010)*, pages 50–61, Cambridge, United Kingdom, 2010. ACM.

[7] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[8] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[9] G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.

[10] ACM DEBS 2013 Grand Challenge Description. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails`.

[11] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, et al. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the 2007 CIDR Conference (CIDR 2007)*, pages 412–422, Asilomar, CA, USA, 2007.

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

[13] H.-A. Jacobsen, K. Mokhtarian, T. Rabl, M. Sadoghi, R. Sherafat Kazemzadeh, Y. Yoon, and K. Zhang. Grand Challenge: The Bluebay Soccer Monitoring Engine. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS 2013)*, pages 295–300, Arlington, TX, USA, 2013. ACM.

[14] M. Jergler, C. Doblander, M. Najafi, and H.-A. Jacobsen. Grand Challenge: Real-time Soccer Analytics Leveraging Low-Latency Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS 2013)*, pages 307–312, Arlington, TX, USA, 2013.

[15] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB 2004)*, pages 492–503, Toronto, Canada, 2004.

[16] K. G. S. Madsen, L. Su, and Y. Zhou. Grand Challenge: MapReduce-Style Processing of Fast Sensor Data. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS 2013)*, pages 313–318, Arlington, TX, USA, 2013. ACM.

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

[18] P. R. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *Network, IEEE*, 18(1):44–55, 2004.

[19] L. Probst, I. Giangreco, and H. Schuldt. PAN – Distributed Real-Time Complex Event Detection in Multiple Data Streams. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2016)*, pages 189–195, Heraklion, Crete, Greece, 2016. Springer.

[20] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel. Distributed Heterogeneous Event Processing. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010)*, pages 150–159, Berlin, Germany, 2012. ACM.

[21] C. Schuler, C. Turker, H.-J. Schek, R. Weber, and H. Schuldt. Scalable peer-to-peer process management. *International Journal of Business Process Integration and Management*, 1(2):129–142, 2006.

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

[23] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014)*, pages 147–156, Snowbird, Utah, USA, 2014. ACM.

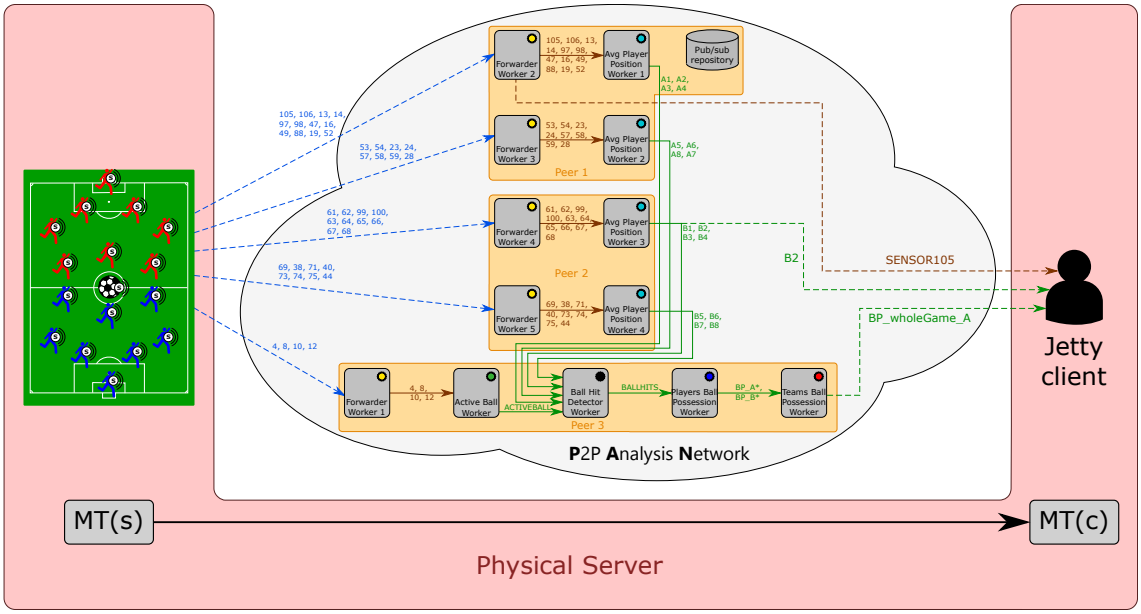
[24] Y. Wu, D. Maier, and K.-L. Tan. Grand Challenge: SPRINT Stream Processing Engine as a Solution. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS 2013)*, pages 301–306, Arlington, TX, USA, 2013. ACM.

Appendices

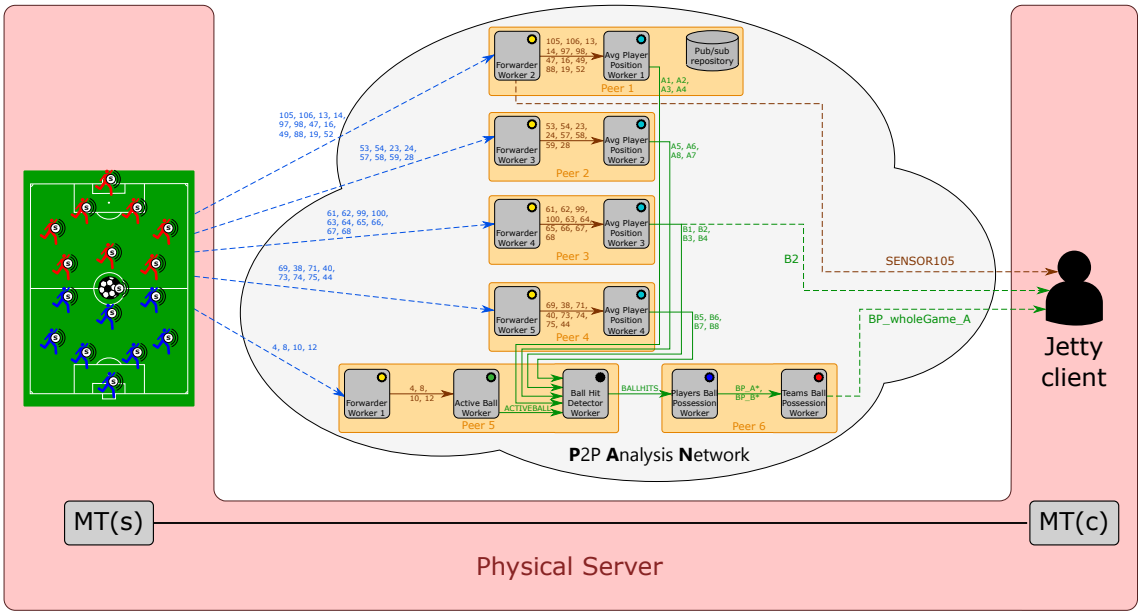
A Evaluation Setups

A.1 Soccer Match Analysis

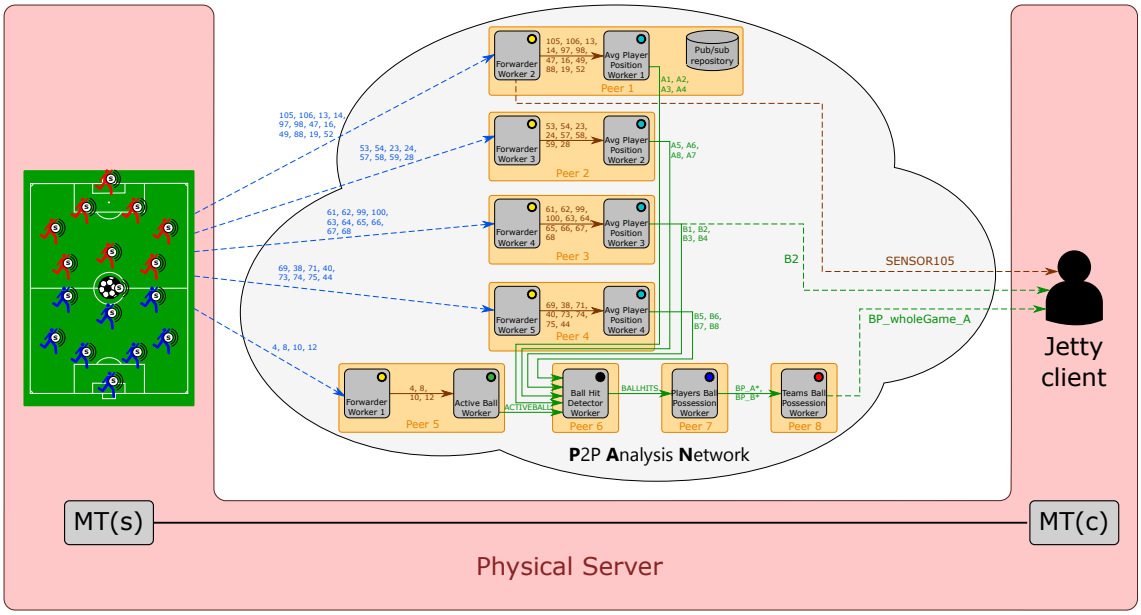
A.1.1 Soccer Match Analysis Workflow Deployed on 3 Peers



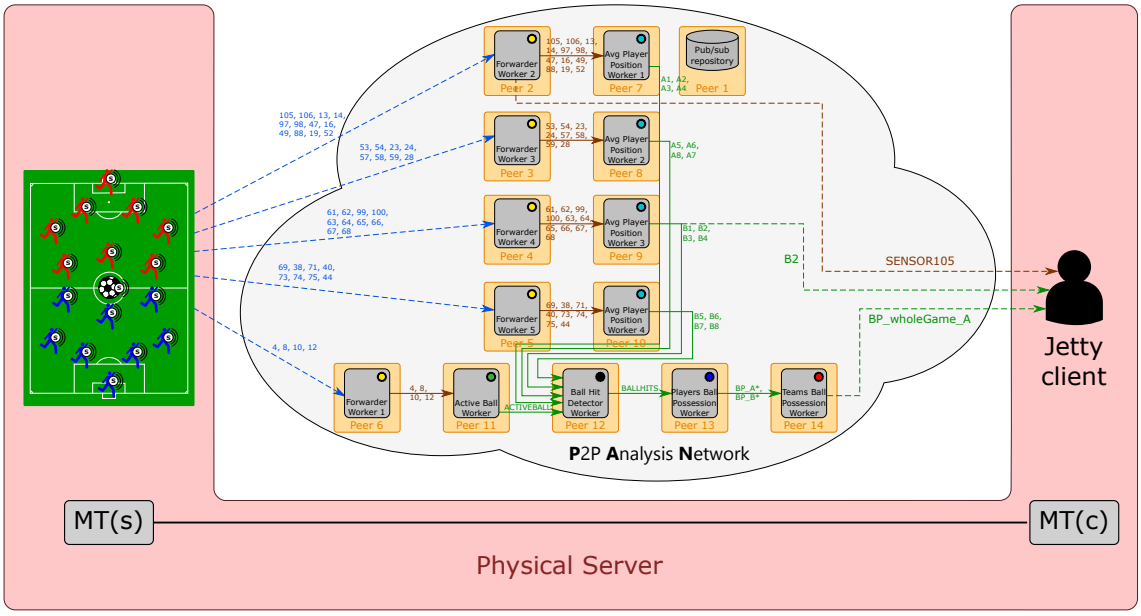
A.1.2 Soccer Match Analysis Workflow Deployed on 6 Peers



A.1.3 Soccer Match Analysis Workflow Deployed on 8 Peers

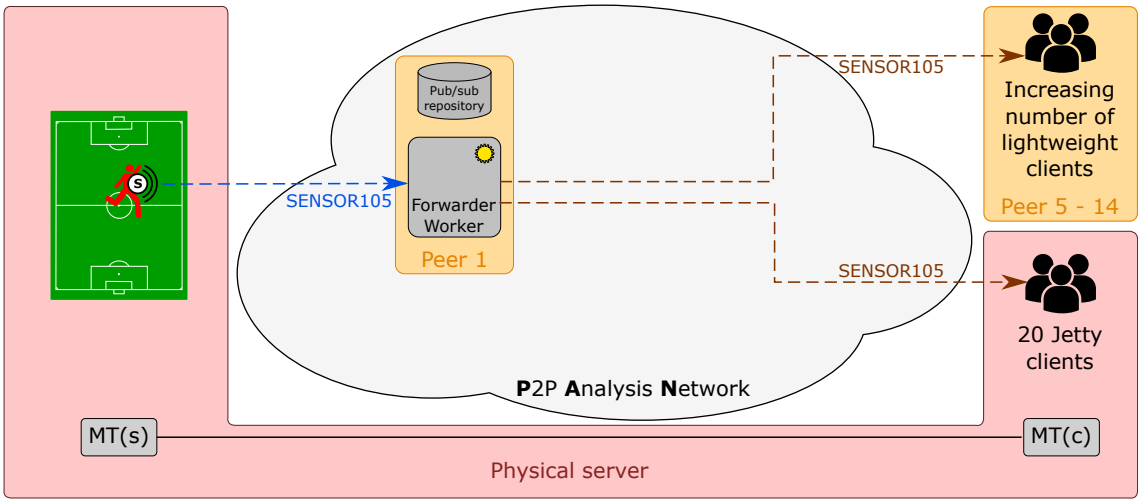


A.1.4 Soccer Match Analysis Workflow Deployed on 14 Peers

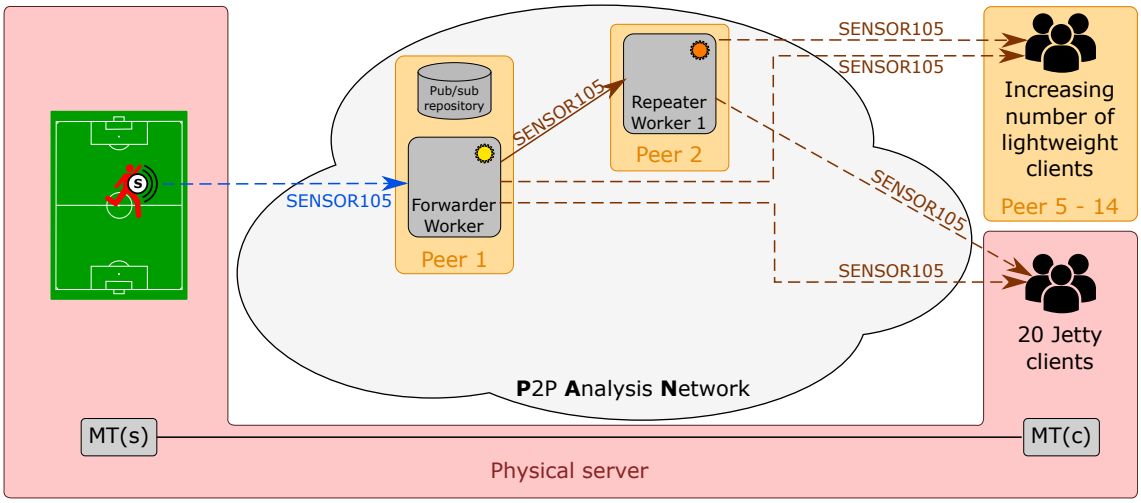


A.2 Load Balancing

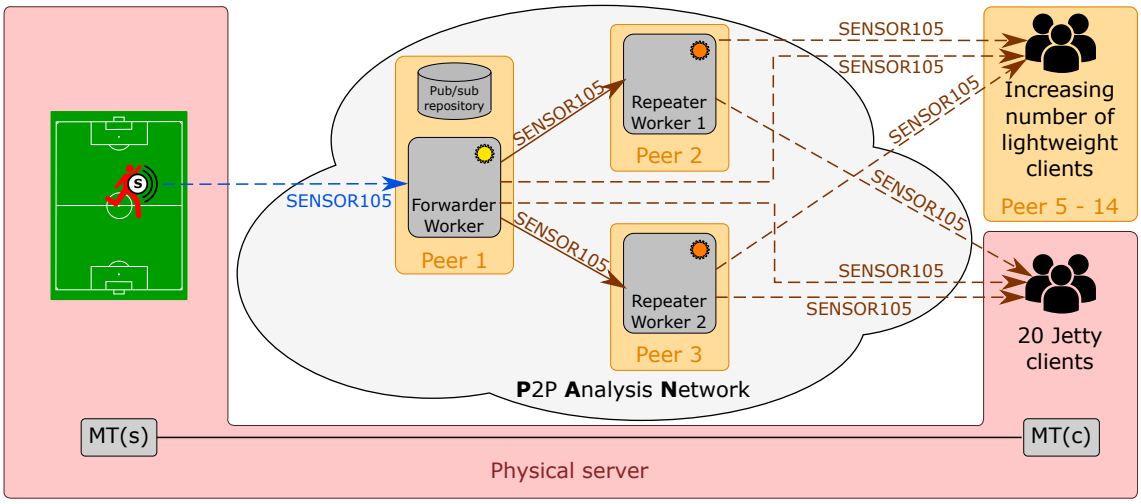
A.2.1 Load Balancing Evaluation Setup with one Forwarder Worker



A.2.2 Load Balancing Evaluation Setup with one Forwarder Worker and one Repeater Worker



A.2.3 Load Balancing Evaluation Setup with one Forwarder Worker and two Repeater Workers



B Sample 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@"
        }
    ]
}
```

C Evaluation Results

All values are given in milliseconds and rounded to two decimal places.

C.1 Effects of PAN Size

	Number of peers			
Stream	3	6	8	14
SENSOR105	69.76	66.61	73.79	88.96
B2	2923.73	123.99	165.68	114.72
BP_wholeGame_A	1141.07	1078.43	948.44	961.06

C.2 Load Balancing

	Number of clients						
Publishers	40	50	60	70	80	90	100
1 Forwarder	82.35	102.56	116.11	134.72	168.85	182.60	209.53
1 Forwarder, 1 Repeater	72.58	73.47	100.24	105.76	120.16	146.35	159.74
1 Forwarder, 2 Repeaters	77.81	74.18	76.18	82.67	97.16	110.23	118.80