Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams

Christopher Mutschler[1,2] and Michael Philippsen[1]
{christopher.mutschler,philippsen}@cs.fau.de

[1]Programming Systems Group, CS Dept., University of Erlangen-Nuremberg, Germany
[2]Sensor Fusion and Event Processing Group, Locating and Comm. Systems Dept.,
Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany

***Abstract*—Event-based Systems (EBS) are used to detect and analyze meaningful events in surveillance, sports, finances and many other areas. With rising data and event rates and with correlations among these events, sequential event processing becomes infeasible and needs to be distributed. Existing approaches cannot deal with the ubiquity of out-of-order event arrival that is introduced by network delays when distributing EBS. Order-less event processing may result in a system failure.**

We present a low-latency approach based on K-slack that achieves ordered event processing on high data rate sensor and event streams without a-priori knowledge. Slack buffers are dynamically adjusted to fit the disorder in the streams without using local or global clocks. The middleware transparently reorders the event input streams so that events can still be aggregated and processed to a granularity that satisfies the demands of the application. On a Realtime Locating System (RTLS) our system performs accurate low-latency event detection under the predominance of out-of-order event arrival and with a close to linear performance scale-up when the system is distributed over several threads and machines.

***Index Terms*—Event-based Processing, Message-oriented Middleware, Scalability and Performance, Publish/Subscribe.**

I. Introduction

High data rate sensor streams occur in many applications, such as surveillance, sports, finances, RFID systems, etc. [1]. As the massive data load is a challenge to the scalability of any system extracting meaningful information, event-based systems (EBS) recently gained much interest. The aim of EBS is to filter and aggregate events (special occurrences of interest) and to iteratively transform them into higher level events until they reach a level of granularity that is appropriate for an end user application.

Since requirements for Event Processing Systems (EPS) can be diverse, systems for event processing are manifold. For instance, consider a warehouse management system. Major focus is on the programming interface of such an EPS so that users can implement detection rules themselves. High update rates are usually not that important [2]. Another application is distributed event processing on sensor nodes. Communication is to be avoided due to high power consumption. Filtering and aggregation are of utmost importance, and memory is only available sparsely [3]. Both applications aim in a similar direction but differ significantly in their workloads and required response times.

In contrast, this paper focuses on applications that demand high-performance, distributed event processing with low latencies. Reliability is important since we usually deal with stateful event detectors. Network failures are unlikely but if they occur, they are all the worse. Although memory is available at high quantity, buffers must be as small as possible to meet the latency requirements. Such low latencies are crucial for performing immediate actions, such as cameras focusing on particular points of interest, financial trading decisions, triggering safety equipment in vehicles, etc.

Assume that we locate players in a soccer game and we apply event-based processing on the position streams. For instance, a blocked shot on goal high-level event is defined by a composition of events from lower levels, see Figure 1. Depending on the position data rate, the detection of proximity on Layer 4 may already consume all the processing power of an entire machine, no matter how efficiently it is implemented. Hence, the detection of events that rely on proximity (in ball sports, there are dozens) becomes infeasible on a single machine.

The solution is to distribute event algorithms (event detectors), across several machines. A middleware manages EPS startup and event dissemination. However, it is not simple to distribute event detectors beyond one machine. Events are generated at different points in the network and are no longer timely synchronized. If the sensor data rate is high, out-of-order events are predominant. As a consequence, naively distributed event detectors process events incorrectly and produce wrong results.

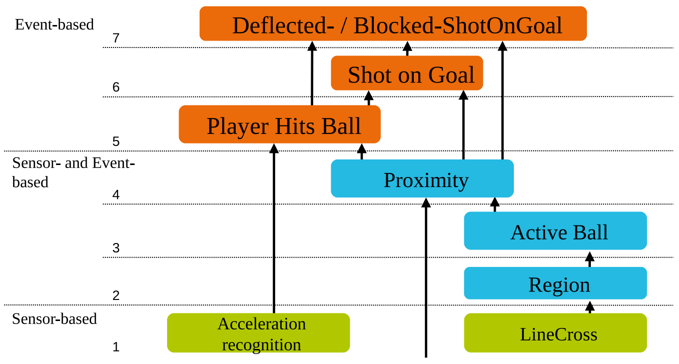


Fig. 1. Event hierarchy levels.

Eliminating these ordering issues by finding an optimal distribution of event detectors over the available nodes is not always possible. Moreover, it would not solve the problem because of application-specific delay types (such as back-setting delays[1]). We therefore focus on the distributed derivation of optimal K-values for the K-slack algorithm. K-slack transparently buffers and reorders events before they are processed by event detectors.

Unfortunately, existing work is insufficient because it either lacks support for distributed processing or for correct event ordering. Established methods also fail for negative patterns since they assume fixed buffer sizes. Timing offsets need to be measured dynamically in relation to dedicated event sets in order to enable an in-order processing of events.

The rest of the paper is organized as follows. We define our time model semantics and provide basic definitions about event streams, (out-of-order) events, and time stamps before we motivate the problem (Section II-A and II-B). Subsequently, we present the main contributions of this paper:

- We formalize the problem and prove that the original K-slack approach does not work for hierarchical event detectors in a distributed system (Section II-C and II-D).
- We show how to make K-slack work by ordering event streams with dynamically adjusting slack buffers under the absence of a global and local clock. Our middleware solution does not need a-priori knowledge, the application developer does not specify delays,[2] and interesting events are still generated with low latency such that, for example cameras, can take immediate action on the live output of the system. Also our solution does not restrict the detector implementation since the middleware reorders the event streams transparently (Section II-E).
- We formally prove the correctness of our dynamically adjusting K-slack approach (Section II-F).
- Since initialization parameters of the algorithm at system startup cannot be defined a-priori, Section III describes two methods to properly estimate them at runtime.

Section IV evaluates our method under real-life conditions when it is used to analyze position stream data from a Realtime Locating System (RTLS) in a sports application. Soccer events suffer from different types of delays, for which we discuss the influence on both the system stability and the correctness of the system output. We also show, that our system can handle massive out-of-order event arrival. Slack buffers for event ordering are dynamically (re-)sized and are optimal in respect to detection latency. Section V discusses related work before Section VI concludes, identifies limitations, and indicates directions for future work.

[1]Back-setting delays occur when events are generated with time stamps that are earlier than the time stamps of the events that cause them. For instance, to detect the shot on goal in Layer 6, the angle of the shot must be right. However, we only know the angle after a few more position events have been received and processed. So the shot-on-goal event ID has a back-setting delay and can only be inserted into the event stream long after it has actually happened.

[2]Most delays depend either on the number of event streams or the object behavior and can hardly be estimated before runtime anyway. If they are estimated, overall detection latency may be too high. For instance, an offside in soccer must be detected as quickly as possible to blow the whistle.

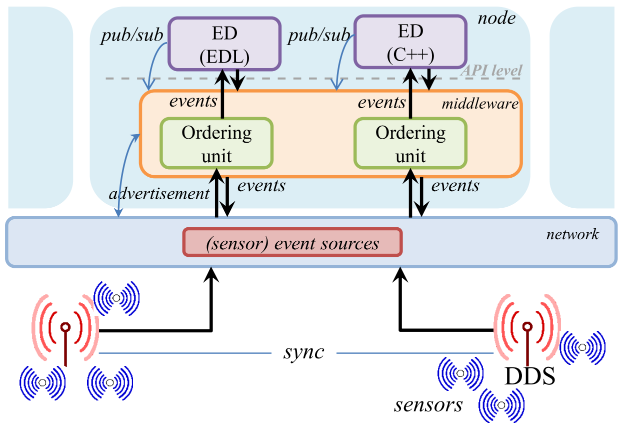


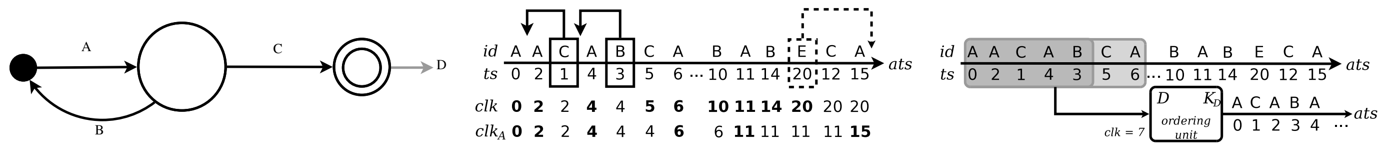
Fig. 2. Distributed publish/subscribe EPS.

II. Self-Adaptive Ordering Units

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

It is difficult and error-prone to implement event detectors that can process out-of-order events. Especially as developers often have no clue about the timing delays their code may face at runtime. Systems that provide a high-level Event Definition Language (EDL) to manage out-of-order events [4] often restrict the expressiveness of event definition. This paper presents a hybrid approach: expressive event detectors can still be implemented in a native programming language with the assumption of ordered events because the middleware transparently reorders out-of-order events under the hood. Hence, the middleware has no knowledge about the patterns that are implemented in the detectors.

Our runtime system is depicted in Figure 2. It consists of several data distribution services (DDS) that collect sensor data (for example an antenna that collects RFID readings), and several nodes in a network that run the same event processing middleware. On top there are event detectors spread over the network. An event detector and the middleware exchange subscriptions, publications, and necessary control information. The middleware has no knowledge about the complex event pattern that is implemented in the detector (that can either be implemented in a native programming language [5], or some EDL [6]), and the detector has no knowledge about the distribution of other event detectors and the runtime configuration. At startup the middleware has no knowledge about event delays but just notifies other middleware instances about event publications and subscriptions (advertisement) [2].


A
C
D
B
id A A C A B C A B A B E C A
ts 0 2 1 4 3 5 6 ··· 10 11 14 20 12 15
ats
clk 0 2 2 4 4 5 6 10 11 14 20 20 20
clk_A 0 2 2 4 4 4 6 6 11 11 11 11 15
id A A C A B C A B A B E C A
ts 0 2 1 4 3 5 6 ··· 10 11 14 20 12 15
ats
D
K_D
ordering unit
clk = 7
A C A B A
0 1 2 3 4 ···
ats

(a) Example for A!BC. (b) Out-of-order event stream. (c) Sorting window over event stream.

Fig. 3. Out-of-order examples.

For each detector the middleware also provides a personal ordering unit for the incoming event stream. The middleware dynamically adapts the buffer size of the ordering unit by means of the method described in Section II-E. The middleware is therefore generic and encapsulated, and does not incorporate the application-specific complex event definition that is implemented in the detectors.

A. Time Model Semantics and Definitions

The time model we assume is that sensor events are time-stamped from the same discrete time source before they are sent to the network for processing. This requires synchronization of all system units that directly communicate with the sensors. However, this is not a great loss of generality because applications that require a low detection latency usually have the means to time-stamp sensor events when they are generated. For instance, in warehouse applications, the RFID readers may synchronize over LAN, time-stamp the sensor readings accordingly, and push the data packets as sensor events to the network. In a locating system the microwave signals of transmitters are extracted by several antenna units that are synchronized over fiber optic cables [7].

We use the following terminology throughout the paper:

Event type, instance and time stamps. An event type defines an interesting occurrence and is identified by a unique ID. An event instance is an instantaneous occurrence of an event type at a point in time. It can be a primitive (sensor) or a composite event. An event has three time stamps: an occurrence, a detection, and an arrival. All time stamps are in the same discrete time domain according to our time model. An event appears at its occurrence time stamp ts, or just time stamp for short. It is detected at its detection time stamp dts. At arrival time stamp ats the event is received by a particular EPS node. The occurrence and the detection time stamp are fixed for an event at any receiving node whereas the arrival time stamp may vary at different nodes in the network.

Out-of-order Event. Consider a stream of events e_1, e_2, $\cdots$, e_n that is sorted ascending to their arrival time stamps: $e_k.ats < e_{k+1}.ats$, $(1 \leq k < n)$. An event e_j is an out-of-order event if there is an event e_i with $1 \leq i < j \leq n$ and $e_i.ts > e_j.ts$.

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

B. Motivation

Without an ordering unit, event processing may fail. Consider the following example. To detect that a player kicked a ball, we wait for the events that a ball is near the player and then, that the ball is kicked, i.e., a peak in acceleration. Between the two events there may not be the event that the ball leaves the player, because in that case the ball would just have dropped to the ground. More formally: if we receive event A (near) and subsequently C (acceleration peak) and not B (not near) in between, we generate event D.[3] Figure 3(a) gives a finite state automaton for event D. To simplify, we leave out the differentiation of transmitter IDs for player identification.

In the first part of the event stream in Figure 3(b) the events C with time stamp 1 (C1 for short) and B3 are received too late. With the occurrence time stamp 1, the in-order placement of C1 would be between A0 and A2, and with the occurrence time stamp 3, the in-order placement of B3 would be between A2 and A4, but the arrival time order is different. A detector that ignores ordering, incorrectly detects D out of A2/C1 and cannot detect D out of A4/C5.[4]

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

In Section IV we show detailed measurements of event delays from the detector in Figure 3(a). A and B are delayed up to 180ms whereas C has only 5ms delay. Without ordering these events, the detector cannot always work correctly.

C. The K-Slack Approach

The well-known K-slack algorithm [8] deals with out-of-order events. It uses a buffer of length K to delay an event e_i for at most K time units (K must be known a-priori). K-slack has shown significant reduction of the run-time state (the number of buffered data elements) when executing queries over streams. Although K-slack has been developed for non-distributed and single-threaded stream applications it can be used in distributed environments. Given some local clock clk, Li et al. [9] buffer an event e_i at least as long as $e_i.ts + K \leq clk$. As there is no global clock in a distributed reactive system, each node synchronizes its local clock according to *the largest*

[3]This is similar to the book shelf reading discussed in RFID-based EPS.

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

time stamp seen so far on any incoming event. See the clk-line in Figure 3(b). Bold numbers indicate an update of clk. Note that the right hand side of the above inequation (also) depends on the stream of incoming events e_i.

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

The ordering unit in Figure 3(c) implements the K-slack approach with K_D=3 for the given event stream. It applies a sliding window to the input stream, delays the events according to their time stamps, and produces an ordered output stream of events. The dark-grey area shows the events that are emitted when updating clk to 7, the brighter-grey area shows the events that remain in the buffer. The size of the sliding buffer is not defined by the number of events but varies with both clk and the event time stamps. With K-slack we can also deal with back-setting delays because it is simply a part of an event's overall delay. The event is ordered as if it would have a real delay.

D. Problem Definition

A single fixed a-priori K does not work for distributed, hierarchical event detectors. As K-slack takes K time units to generate `D` out of `A/C` (the ordering unit has to wait at least until clk=`C`.ts+K before `C` can be emitted to the detector that may set `D`.ts:=`C`.ts[5]) an event detector on a higher layer that waits for `D` and that only buffers for K time units, may miss `D` because `D`.$ats\geq$`D`.$dts\geq$`D`.ts+K. Waiting times (must) add up along the hierarchy.

An individual K-value per event detector solves the problem. Such a K_n must at least be set to a value larger than $max(K_{n-1})$, that is larger than the maximal delay of all the subscribed events (because then each detector buffers its events longer than the lower levels delay their events). If all K_n are then sufficiently large, K-slack works properly and provides sorted event streams. Although this sounds good at first, conservative and overly large K-values result in large buffers and therefore long latencies for hierarchical event processing, and must be avoided for the types of applications that we target.

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

1) **clk grows unexpectedly** $\Rightarrow$ a previously determined K-value is too small. Consider Figure 3(b) again. On receipt of `E20`, clk is set to 20. With K_D calculated from the previous value of clk, the event sequence `A11/B14/C12` does not make the event detector generate `D` because at clk=20 `C12` arrives too late for the current buffer size. `A11` and `B14` are emitted with reception of `E20` and then `C12` is processed out-of-order.
2) **K grows unexpectedly (or is still unknown, 0)** $\Rightarrow$ a previously determined K-value is too small. In Figure 3(b) the maximal delay of `C` is set to 3=4-1 as soon as `A4` arrives. The maximal delay of `B` is 2=5-3. Hence, for `D` the suitable K is the maximum of the worst delays of all its input events, namely $K_D = \max(0, 2, 3) = 3$. Assume now that between `A6` and `B10` another `B` with time stamp 8, and between `B10` and `A11` another `C` with time stamp 7 arrives (that is later than it was expected to arrive). Then we would retrofit the value of K_D to 11-7=4. Although the detector should have fired for `A!BC` (time stamps 6-7), it did not generate the `D` because at that time, the old value of K=3 has still been in place and `A6/B8` have already been processed.

E. Our Solution

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

1) Derivation of clk*:* The problem of unexpected clk changes can be fixed by no longer setting the clock to the largest time stamp seen so far *on any incoming event*, but to only use one or more designated event types for setting it. While the above definition of out-of-order events has only identified events that are late, we now also use the clk-values to postpone events that arrive too early. Consider an event stream e_1, e_2, $\cdots$ e_n as before. Assume that clk is only set by events of type ID. Then event e_j is ***out-of-order*** if there do not exist e_i, e_k, with $e_i.id$=$e_k.id$=ID and $e_i.ats \leq e_j.ats$ so that $e_i.ts$$\leq$$e_j.ts$$\leq$$e_k.ts$ from now on.

In Figure 3(b) an unexpected change of clk can be avoided if the clock is set only on incoming events of, for instance, type `A`. See the clk_A line in Figure 3(b). `C12` arrives while clk_A is still 11 and fits for the current size of K.[6] By avoiding sudden changes of K, early events are postponed until K has been updated and will no longer make the detector fail.

The remaining question is which event type to pick for setting clk. The higher the occurrence frequency of the picked event type is, the fewer events need to be postponed, the

[5]`D` takes the occurrence time stamp of `C`.

[6]It is irrelevant if the picked clk-setting event type is used by the detector because it is only used to adjust the right hand side of the above inequation, that is clk in $e_i.ts$+$K$$\leq$$clk$. `B`, `C` or some other event type that is embedded in the stream work equally well.

smaller are the resulting K-values, and the better are the measured delays. If there is a choice of event types, the one with the more stable and fixed delay is preferable, because it better reflects the real time. If otherwise the events of a certain type vary in their arrival times, clk does not behave smoothly. For better clock update frequency, instead of using just one event type, it is possible to use a set of event types for clk-setting, provided that those event types have the same absolute delays.[7] High data rate sensor events (ideally from one source) with precise time stamps are excellent candidates.

This technique makes K-slack work when there is no global clock in a distributed system. Moreover, it serves as a general solution in stream applications and improves K-slack behavior in general.

2) Derivation of K: With a given stream of incoming events e_i and a sufficiently stable clk, the key idea is to perform the runtime delay measurements by comparing an event's occurrence time stamp with its arrival time stamp, and to use this knowledge about disorder to derive a suitable K. Recall that for an event detector d the proper K_d is calculated as the maximal event delay of all subscribed events: $K_d = max_j\,[\delta(e_j)]$. The maximal delay of an event is $\delta(e_j)$=$e_k.ts$-$e_j.ts$, where e_k is the next event that updates clk.[8]

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

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

[7]It is insufficient if those event types only form an ordered sub-stream. For instance, setting clk to `C5` can be critical because delays may be measured in respect to a clk-setting event with higher delay and may then be emitted too early. If events are sorted by a clock that does not reflect the real time, postponing events by measured delays may not be appropriate.

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

Algorithm 1: Pseudo code for the ordering unit (without λ, and without reduction of K)

```
Data: InputEvent $e$, EventDetector $d$,
      DelayCalculationList $lst$, $K_d$, clkEvents
if $e.id \in d.GetSubscriptions()$ then
    $d.inputBuffer.InsertionSort(e)$;
    $lst.add(e)$;
if $e.id \in \{clkEvents\}$ then
    $clk \longleftarrow e.ts$;
    $d_{max} \longleftarrow 0$;
    // calculate new delays
    for Event $e_{tmp} : lst$ do
        $d_{tmp} \longleftarrow clk - e_{tmp}.ts$ ;  // delay of $d_{tmp}$
        $delays\,[e.id]$ .add($d_{tmp}$);
        $d_{max} \longleftarrow (d_{tmp} > d_{max})\ ?\ d_{tmp} : d_{max}$;
    if $d_{max} > K$ then          // check $K$-increase
        $K \longleftarrow d_{max}$;
        // ts=clk-K
        propagatePseudoEvent($d.id$, $K$, $clk - K$)
    while Event $e_{tmp} \longleftarrow d.inputBuffer.front()$ do
    // event relaying
        if $e_{tmp}.ts + K \leq clk$ then
            if $e_{tmp}.isNoPseudoEvent$ then
                $d$.relay($e_{tmp}$);
            $d$.inputBuffer.popFront();
        else
            return;
```

3) Event Ordering Units: The resulting middleware has both a stable clock clk for the right hand side and a sufficiently good K for the left hand side of our inequation. These values are used to setup the required event ordering unit that turns an out-of-order stream into a sorted input for the detector, see Figure 3(c). The ordering unit is a black box that is mounted between the original event stream and the input of the event detector so that the event detector can assume sorted input. Note that there is usually more than one event detector per middleware and machine, each of which has a specific ordering unit with a suitable and detector-specific K that only picks the subscribed events from the main event stream. Algorithm 1 shows the pseudo code for the implementation of such an ordering unit. Due to limited space we skip the K-overfitting and reduction with λ.

Example. Figure 4 shows such an event ordering unit for the input stream of Figure 3(b). clk is set whenever an `A` is received, see the bold values in the clk_A line. At the beginning when `A0` and `A2` are received, there are no measurements and K is still 0, which means that both events are immediately passed to the output stream and are not delayed (they fulfill $e_i.ts$+K≤clk_A). When `C1` is received, it is pushed to the

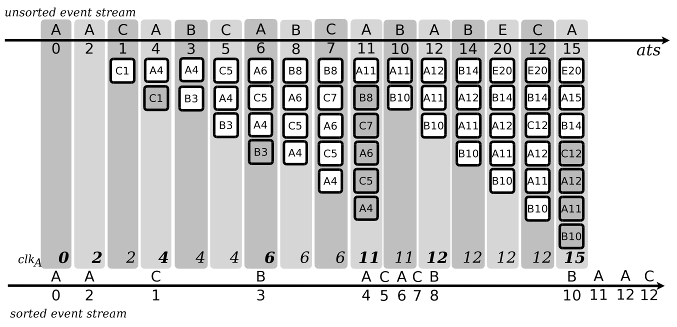


Fig. 4. Event ordering unit for event detector D.

buffer and waits until A4 updates clk_A. As the delay for C1 is 3=4-1, we set K=3 and relay C1 ($e_i.ts$+K=1+3$\leq clk_A$ =4). A4 is buffered at least until clk_A equals 7=4+K. With A6 the maximal delay of B3 is 3=6-3, K holds, and B3 is passed to the output stream.

The output stream is a sorted sequence of events with a minimal delay. Whenever an input event is received (pseudo events are ignored), it is sorted into a buffer according to its occurrence time stamp. If out-of-order events are rare, insertion sort of new events usually is just a simple and fast push to the head of the buffer. Whenever clk is updated and $e_i.ts$+$K\leq clk$ holds for some tail events e_i in the buffer, we emit those e_i to the output stream and process the next input event. In the startup phase of the example, only C1 remains out of order since we did not measure a delay yet. In Section III we describe two techniques to improve startup behavior.

F. Formal Proof

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

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

$$\hat{\delta}(e_j) \leq \hat{\delta}(e_k) + \delta(e_j). \tag{1}$$

$\hat{\delta}(e_j)$ denotes the real delay of e_j according to wall-clock time.[9] In other words, the real delay of e_j must be less than or equal to its maximal delay $\delta(e_j)$.

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

$$e_i.ats \leq e_j.ts + K. \tag{2}$$

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

[9]The wall-clock time is the real time that is not used by our method but only applied here to prove the correctness.

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

Proof of Property (1), Delay Measurement: Recall that the maximal delay of an event e_j is defined as $\delta(e_j|clk \leftarrow e_k) = e_k.ts - e_j.ts$, with the restriction that the wall-clock arrival time wc (which we have no access to) of the events is $wc(e_j)<wc(e_k)$. When we insert our delay calculation into the property of inequation (1) we get

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

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

$$e_{j/k}.ts = wc(e_{j/k}) - \hat{\delta}(e_{j/k}), \tag{3}$$

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

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

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

Proof of Property (2), Event Ordering: In order to prove that no event detector receives out-of-order events when setting K to the maximum of all event delays $\delta(e_n)$ we show that for any pair of events $\langle e_i,e_j\rangle$ we reorder it correctly. As the property then holds for any pair of subscribed events this proves the property for the whole set of events. Formally, we need to prove that $e_i.ats \leq e_j.ts + K$ holds:

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

Since $e_j.ts > e_i.ts$ but $e_j.ats < e_i.ats$ and Property (1), it follows that $\delta(e_i) > \delta(e_j)$. We can simplify the inequation:

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

From Property (3) follows that

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

which we can replace in the inequation:

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

Since $\delta(e_i) \geq \hat{\delta}(e_i)$, which follows from the proof of Property (1), and $e_i.ts < e_j.ts$, which is the precondition, the inequation is always true. □

Hence, based on the proof of Property (1) that the maximal delay $\delta(e_i)$ is larger or equal than the real delay of e_i (which means that waiting for $\delta(e_i)$ time units suffices to receive a late e_i), the proof of Property (2) shows that input event sets are always reordered correctly.

III. Initialization

With knowledge about the expected delay of events, the middleware can order them. However, at startup the middleware does not have this knowledge and stumbles, see `C1` in Figure 4. Here are two ways to correctly initialize the delays.

A. Iterative Delay Calculation

In many applications we either can run and restart a system several times, or we have some pre-recorded sensor streams that can be employed to calibrate the correct configuration. Figure 5 depicts this strategy. The idea is to start from an initial configuration c_0 (= values for all K's) and to derive c_1 (a proper configuration for the event detectors on level 1) by running the system and measuring sensor event delays directly. The application will fail because of stumbling detectors, but at the end of this run all events on the first level have suitable K-values that are used when the system is restarted. Restart i, $i > 1$, hence derives c_i with correctly measured delays for the event detectors on levels $\leq i$. Algorithm 2 converges after at most n iterations, with n being the highest hierarchy level. We assume that there are no cycles in the detection hierarchy.

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

B. Semi-Configured Delay Estimation

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

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

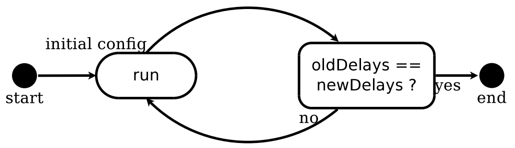


Fig. 5. Iterative delay update.

Algorithm 2: Iterative Delay Calculation

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

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

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

Algorithm 3 iteratively traverses the hierarchy and propagates all pseudo events through the network. After they have reached the highest hierarchy level, K values for all detectors are reasonably set. There is no need for further iterations.

IV. Evaluation

For the evaluation we have analyzed position data streams from a Realtime Locating System (RTLS) installed in the main

Algorithm 3: Pseudo Event Propagation

```
Data: Configuration $c_{old}$, $c_{new}$, Clock $clk$, Jitter $d_j$
begin
    for EventDetector ed : GetFirstLevelDetectors() do
        for Event $e_i$ : GetPublishedEvents(ed) do
            $d_p \leftarrow c_{old}$.GetProcessingDelay($e_i$);
            $d_b \leftarrow c_{old}$.GetBackSettingDelay($e_i$);
            $e_i.ts \leftarrow clk$-$d_j$-$d_p$-$d_b$;
            Send($e_i$);
    while ReceivePseudoEvent(e) do
        $c_{new}$.SetDelay($e.id$, $clk - e.ts$);
        for EventDetector ed : GetDetectorsBySub(e) do
            if $c_{new}$.AllMeasured(ed) then
                for Event $e_i$ : GetPublishedEvents(ed) do
                    $d_o \leftarrow c_{new}$.GetOrderingDelay($ed$);
                    $d_p \leftarrow c_{old}$.GetProcessingDelay($e_i$);
                    $d_b \leftarrow c_{old}$.GetBackSettingDelay($e_i$);
                    $e_i.ts \leftarrow clk$-$d_p$-$d_b$-$d_o$;
                    Send($e_i$);
```

soccer stadium in Nuremberg, Germany. This RTLS tracks 144 transmitters at the same time at 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Each player is equipped with four transmitters, one at each of his limbs. The sensor data consists of absolute positions in millimeters, velocity, acceleration, and Quality of Location (QoL) for any direction [7].

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

We present results from applying our event processing system and our algorithms on position data streams from the stadium. Our platform consists of several 64-bit Linux machines, each equipped with two Intel Xeon E5560 Quad Core CPUs at 2.80 GHz and 64 GB of main memory that communicate over a 1 Gbit fully switched network. For our tests we organized a test game between two amateur league football clubs and processed the incoming position streams from the transmitters.[10]

Section IV-A analyzes delay and timing issues. We discuss how certain delay types affect event detectors from different hierarchy levels. Although out-of-order events are predominant and certain events are significantly delayed our technique nevertheless derives optimal K-values. Section IV-B evaluates

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

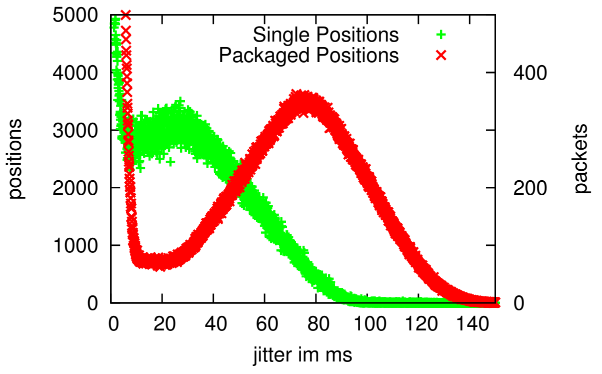


Fig. 6. Position jitter delay.

the system performance and its robust and accurate event detection. We focus on measurements of a specific event detector and show how delays of input events have influence on the self-adapting ordering units of the middleware. We prove that we estimate an optimal K at runtime with the result of smaller buffers and less latency. Overall we achieve a close to linear performance scale-up for distribution over several machines.

A. Delay Discussion

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

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

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

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

We recorded position arrivals in our RTLS at 70% of the maximal system capacity, which is with 36,000 positions per second. Consider the *single positions* plot in Figure 6. Positions from ball transmitters usually have a jitter delay below 5ms, caused by routers and different transmission lane lengths. Positions from other transmitters have jitter delay above 5ms because for them, the position data is extracted from the microwave signal with lower priority. Some position events even have a jitter delay of up to 100ms.

An RTLS usually packages ten positions of a particular transmitter into one packet. Packaging adds delay to all but the last position in the package and hence adds jitter delay. For the low priority positions this adds up to 145ms, see the *packaged positions* plot in Figure 6.

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

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

Ordering delays can vary from a few milliseconds to hundreds of milliseconds. Figure 7 shows the distribution of optimal K-values for the soccer application up to hierarchy level 9. On higher hierarchy levels K-values are much larger but subscribed events occur only seldom. Note that the drop at hierarchy level 2 is due to the fact that our soccer application uses some event detectors that do not rely on low priority transmitters.

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

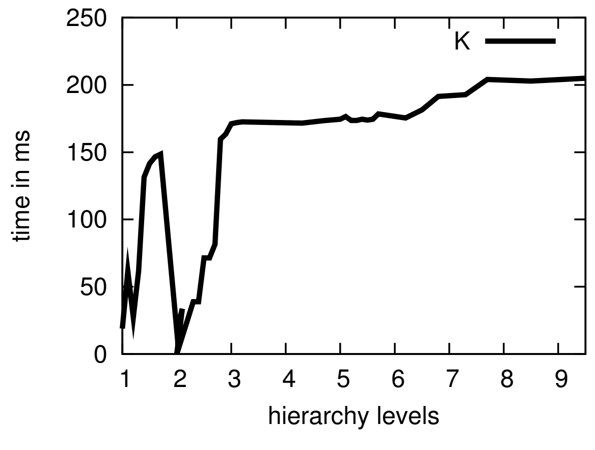


Fig. 7. Distribution of K. K values between i and $i+1$ show event detectors of level i, sorted by growing K.

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

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

From Section II-E we know that event delays are used to select K-values. On lower levels of the detector hierarchy, where position data is processed, delays are dominated by the position jitter delay. On higher hierarchy levels other delay types gain more importance. Position jitter delay is only introduced at the lowest level, whereas other types of delay add up along the hierarchy, see Figure 7.

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

The K-values must be as small as possible to reduce latency. Besides the fact that we may not even be able to predict many of the above delay types at all, manually pre-selecting a fixed (increase of) K is not a viable option. For reasons of reliability any manual approach would need to significantly overfit the K-values. For the K's of the first level (which is the position jitter), K-values around 500ms per detector are reasonable because the system could become fully charged and packets would get lost. For any higher hierarchy level, additional 25ms are needed to compensate for worst-case network, system inconsistencies, and detector runtimes that usually also depend on the event load that we cannot estimate any better. Even without any back-setting delays, the K-values of the highest event detectors are at least around 1 second. In contrast, Figure 7 shows that our dynamic technique finds much smaller K-values that are suitable for the actual system state instead of a worst-case scenario. We achieve latencies that are less than a fifth of the manual ones.

Such a wide spread of delay types as presented here is not unique to position sensor streams. For instance, it is also seen in financial market data where stock markets are timely

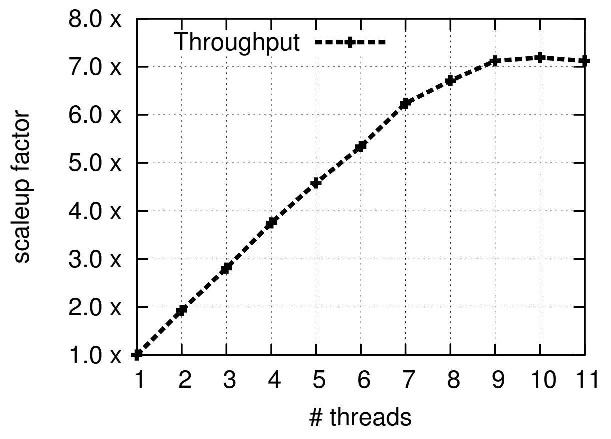


Fig. 8. Event throughput for threads.

synchronized by the backend but the data is distributed over the internet. In RFID systems the situation is similar, as many readers collect data and send them to a centralized server for analysis.

B. Application Performance

Our middleware implementation takes about 8,000 lines of C++ code to implement the basic functionality and event ordering. On top of it, we implemented 482 different event detectors in 15 levels from simple line events up to complex technical scenarios (also in C++). On average the size of an event packet is about 200 Bytes. The largest is about 4 KBytes.

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

1) Event Throughput Scalability.: In contrast to many other EPS such as SASE [6] or Cayuga [10] our system can use threads and can be distributed across several nodes to run in parallel. For a benchmark of our system we performed different geometric calculations and processed position information from our localization system. Figure 8 shows the system performance when distributing the processing over several threads. We achieve a close to linear performance scale-up of the event throughput with respect to the number of available threads. We reach a plateau for 8 threads or more since this is the number of physical cores. The performance scale-up for the distribution over several machines strongly depends on the actual allocation of event detectors and their processing hierarchy. However, even in the worst scenario we achieve a performance scale-up when distributing across several machines.

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

2) Detector Reliability and Latency.: To evaluate the event ordering units we recorded the delays of incoming events for the *Player Hits Ball* event detector, see Figure 9. These events are *Is Near*, *Is Not Near*, both oscillating between 5 and 45 ms delay, and *Ball Acceleration* with a delay of 1-2ms. We took out the single positions jitter for simplification. Other detectors of the soccer application behave similarly. The technique presented in Section II-E to select a suitable K-value at runtime correctly orders over 95% of all events even without any a-priori knowledge, see the straight black line. There are rare points at which K is too small and causes a misdetection, see the crosses, before it is increased. With an overfitted K and an added safety margin of λ=0.5 the detector works considerably better (upper line): not a single K misses the maximal event delay, the event detector is always supplied with ordered events and is fully reliable.

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

C. Discussion

We first zoomed into the sub-delays of the total event delay, discussed their quantities, and argued that out-of-order events are the rule in event stream processing and that efficient low-latency solutions are highly required. We demonstrated quantitatively that a manual K-selection causes considerably higher detection latencies (>8.4×). Our middleware system can work in a network and performs event ordering without any a-priori knowledge. It adaptively finds the best K's that minimize buffer costs and latency at runtime. Event detectors hence see transparently pre-sorted event streams as their inputs and are therefore easy to implement.

Unfortunately, there is no related work that makes use of a

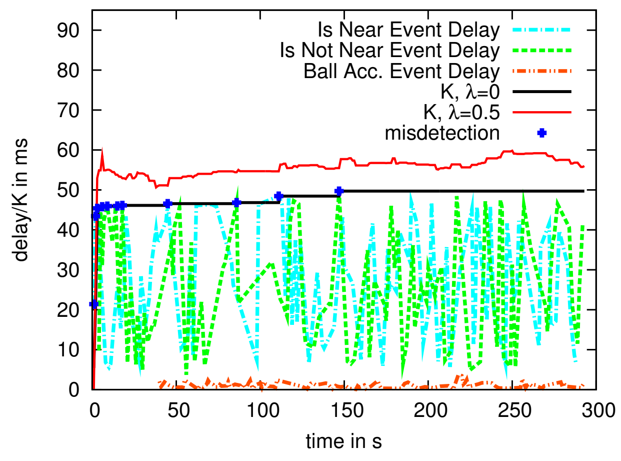


Fig. 9. Delays and K.

similarly large real world setup. Existing work cannot be used for a quantitative comparison because it either lacks support for distributed processing, or for correct event ordering when dealing with negative patterns or back-setting delays.

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

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

V. Related Work

As already discussed, related work in the field of event processing is manifold since requirements are diverse. Section V-A gives an overview of recently emerged EPS. We focus on RFID-based systems because their requirements are most similar to ours when considering different data sources and out-of-order events. Classical work such as Lamport et al. [11] cannot be applied since the total order provided by local clocks is ambiguous on different nodes. Section V-B then focuses on methods and techniques from other contexts.

A. Event Processing Systems

SASE [6] is an event processing engine for RFID readings that is also the foundation of follow-up work [12, 13]. SASE works with Nondeterministic Finite Automata generated from event queries. Although Li et al. [9] solves some problems of out-of-order event arrival, SASE fails when it is distributed over several machines.

Another EPS for RFID readings is Cayuga [10]. Work built on top of Cayuga is demonstrated in [14, 15]. Cayuga uses a timing mechanism built on priority queues and *epochs*. As in SASE, Cayuga assumes that events are delayed for at most K time units. K is defined a-priori. In an epoch it only processes events with a time stamp from that epoch. Further work by Brenna et al. [16] distributes Cayuga over a network. In contrast to our work there is no back-setting and events may not cross epoch boundaries.

The Complex Event Detection and Response system CEDR [17] comes with a query language to express a wide range of event patterns, comprising temporal correlation and negation. The language also has consistency levels for latencies and out-of-order events. CEDR handles out-of-order events by retracting incorrect output and by adding correct, revised output in turn. However, if out-of-order events are predominant, almost all generated events must be retracted, often triggering a cascade of retraction along the detector hierarchy. This poses non-trivial challenges to the memory management and to preclude retractions we have to accept high latencies.

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

B. Methods

O'Keeffe et al. [18] analyze the influence of communication errors in addition to timing uncertainties due to the lack of a global clock in distributed systems. Their complex event language can declare detection policies to specify how to deal with errors. The authors address different tasks than we do: their events have time intervals rather than time stamps due to clock uncertainties, their event load is significantly lower, and they do not consider timing issues introduced by distributed processing as critical.

Brito et al. [19] speculatively process events in parallel. This is achieved by means of an underlying Software Transactional Memory (STM) infrastructure. The basic approach is different from ours in that the authors assume that stateful event detectors need to be executed sequentially. Although the presented STM method achieves good results for parallel execution on multicore systems, it would suffer when it is distributed because buffering and committing of events would no longer be efficient.

Fodor et al. [4] split complex events into a set of binary goals of subpatterns. Goals are chained and a complex event is detected whenever the top goal is reached. They do not cope with additional delays and also assume that events are processed on a single machine. Signalling those goals over a network presumably degrades performance and also introduces delays. Moreover, in order to form those goals, events cannot be arbitrarily defined as they must be implemented in an EDL with limited expressiveness.

GauthierDickey et al. [20, 21] focus on event ordering in peer-to-peer games for massively multiplayer online games (MMOGs). They introduced NEO, a low-latency event ordering protocol that prevents players from cheating in an untrusted environment and also abandons the need for a client/server infrastructure. Time is divided into rounds of fixed length that are an implicit upper bound for the delays. NEO does not fulfill our requirements because it simply discards events that are too old for the current round.

Tucker et al. [22] and Li et al. [23] use special annotations embedded in data streams, called *punctuations*, to specify the end of a subset of data. These serve as window delimiters and time markers, and inform that no event will be generated with a lower time stamp. However, for negative patterns, an event detector must fire a punctuation at each sensor event, otherwise upper level detectors will buffer and hold the processing. The introduced network load is not tolerable. Also timed punctuations do not work in application-specific event detectors due to the lack of knowledge about the runtime configuration and would introduce latency.

Chandramouli et al. [24] permit stream revisions by using punctuations. They give an insertion algorithm for out-of-order events that removes invalidated sequences. Since our system is highly distributed, removing invalidated sequences is not possible. Events that need to be invalidated may already be consumed/processed on other nodes.

Srivastava et al. [25] model stream time differences by wall-clock dependencies, and define clock-skews between data sources. For any data arrival they set heartbeats on each stream. A heartbeat at wall-clock time c is the maximal application time stamp τ such that all tuples arriving from that stream after time c must have a time stamp $> \tau$. By using the local clock and by generating heartbeats, time constraints can be defined with high accuracy. This also makes their method costly (slow) and limits the scalability of the system. With 150 detectors (streams) and 50,000 sensor events per second we would insert 7.5 million heartbeat events per second.

VI. Conclusion

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

The performance (in terms of latency reduction) is limited by the derivation of clk. If clock updates are rare, the detection latency of events increases. Therefore, future work will focus on how to exhaustively exploit the derivation of clk and therefore K by setting the internal clock by several event types that have different delays. A second topic is to loosen the strict and conservative buffering approach and to open up for speculative processing and retraction of out-of-order events.

Acknowledgements

This work is supported by the Fraunhofer Institute for Integrated Circuits IIS whose RTLS called RedFIR we have used. We are grateful to the researchers at the Fraunhofer IIS for sharing their work and system with us.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. 21th ACM Symp. Principles Database Systems*, (Madison, WI), pp. 1–16, 2002.

[2] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer, Berlin, 2006.

[3] G. Wittenburg, N. Dziengel, C. Wartenburger, and J. Schiller, "A system for distributed event detection in wireless sensor networks," in *Proc. Intl. Conf. Information Processing in Sensor Networks*, (Stockholm, Sweden), pp. 94–104, 2010.

[4] P. Fodor, D. Anicic, and S. Rudolph, "Results on out-of-order event processing," in *Proc. 13th Intl. Conf. Practical Aspects of Declarative Languages*, (Austin, TX), pp. 220–234, 2011.

[5] Z. Jerzak and C. Fetzer, "BFSiena: a communication substrate for StreamMine," in *Proc. 2nd Intl. Conf. Distributed Event-Based Systems*, (Rome, Italy), pp. 321–324, 2008.

[6] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proc. ACM Intl. Conf. Management of Data*, (Chicago, IL), pp. 407–418, 2006.

[7] T. v. d. Grün, N. Franke, D. Wolf, N. Witt, and A. Eidloth, "A real-time tracking system for football match and training analysis," in *Microelectronic Systems*, pp. 199–212, Springer Berlin, 2011.

[8] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," *ACM Trans. Database Systems*, vol. 29, no. 3, pp. 545–580, 2004.

[9] M. Li, M. Liu, L. Ding, E. Rundensteiner, and M. Mani, "Event stream processing with out-of-order data arrival," in *Proc. 27th Intl. Conf. Distrib. Comp. Systems Workshops*, (Toronto, CAN), pp. 67–74, 2007.

[10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," in *Proc. 10th Intl. Conf. Extending Database Technology*, (Munich, Germany), pp. 627–644, 2006.

[11] L. Lamport, "Time clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, 1978.

[12] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. ACM Intl. Conf. Management of Data*, (Vancouver, CAN), pp. 147–160, 2008.

[13] M. Liu, M. Li, D. Golovnya, E. Rundensteiner, and K. Claypool, "Sequence pattern query processing over out-of-order event streams," in *Proc. 25th Intl. Conf. Data Eng.*, (Shanghai, China), pp. 784–795, 2009.

[14] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White, "Cayuga: a general purpose event monitoring system," in *Proc. 3rd Biennial Conf. Innovative Data Systems Research*, (Pacific Grove, CA), pp. 412–422, 2007.

[15] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: a high-performance event processing engine," in *Proc. ACM Intl. Conf. Management of Data*, (Beijing, China), pp. 1100–1102, 2007.

[16] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed event stream processing with non-deterministic finite automata," in *Proc. 3rd Intl. Conf. Distributed Event-Based Systems*, (Nashville, TN), pp. 3:1–3:12, 2009.

[17] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: a vision for event stream processing," in *Proc. 3rd Biennial Conf. Innovative Data Systems Research*, (Pacific Grove, CA), pp. 363–374, 2007.

[18] D. O'Keeffe and J. Bacon, "Reliable complex event detection for pervasive computing," in *Proc. 4th Intl. Conf. Distributed Event-Based Systems*, (Cambridge, UK), pp. 73–84, 2010.

[19] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *Proc. 2nd Intl. Conf. Distributed Event-Based Systems*, (Rome, Italy), pp. 265–275, 2008.

[20] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr, "Low latency and cheat-proof event ordering for peer-to-peer games," in *Proc. 14th Intl. Workshop Network and Operating Systems Support for Digital Audio and Video*, (Cork, Ireland), pp. 134–139, 2004.

[21] C. GauthierDickey, V. Lo, and D. Zappala, "Using n-trees for scalable event ordering in peer-to-peer games," in *Proc. 15th Intl. Workshop Network and Operating Systems Support for Digital Audio and Video*, (Washington, DC), pp. 87–92, 2005.

[22] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[23] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," in *Proc. VLDB Endow.*, vol. 1, (Auckland, NZ), pp. 274–288, 2008.

[24] B. Chandramouli, J. Goldstein, and D. Maier, "High-performance dynamic pattern matching over disordered streams," in *Proc. VLDB Endow.*, vol. 3, (Singapore), pp. 220–231, 2010.

[25] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proc. 23rd ACM Symp. Principles Database Systems*, (Paris, France), pp. 263–274, 2004.