Runtime Migration of Stateful Event Detectors with Low-Latency Ordering Constraints

Christopher Mutschler[1,2] and Michael Philippsen[1]
{christopher.mutschler,michael.philippsen}@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*—Runtime migration has been widely adopted to achieve several tasks such as load balancing, performance optimization, and fault-tolerance. However, existing migration techniques do not work for event detectors in distributed publish/subscribe systems that are used to analyze sensor data. Since low-latency time-constraints are no longer valid they reorder streams incorrectly and cause erroneous event detector states.**

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

I. Introduction

Runtime migration of system components is the method of choice for load balancing, performance optimizations, fault tolerance etc. [1]. But existing solutions do not work well for distributed event processing systems (EPS) that are used to analyze high data rate sensor streams with low latency. In such systems input data streams usually have a data rate of several thousand events per second, sources may be arbitrarily spread, and events arrive massively out-of-order.

Consider the EPS in Fig. 1. An event detector (ED) that runs on host N_3 subscribes to four events (e.g. sensor readings), namely `A`, `B`, `C`, and `D`, each generated at a different point in the distributed environment. For simplification, these events are detected with zero delay. However, for further processing on host N_3 they will be received with different delays. For instance, `A` (5ms+20ms+30ms+5ms=60ms) and `D` (5ms+30ms+5ms=40ms) have a much higher delay than `C` (5ms+10ms+5ms=20ms). `B` has the lowest delay (0ms) since it is detected a sensor reading device that is directly connected to the same host and does not travel through the network. The ED on N_3 measures these delays at runtime and reorders the events into a totally ordered event input stream. The size of the reordering buffer is selected as low as possible to guarantee event detection with lowest latency. The generated event is detected with at least 60ms delay and may be subscribed by another ED on another host for further processing.

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

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

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

The rest of the paper is organized as follows. Section II reviews related work. Section III provides basic definitions

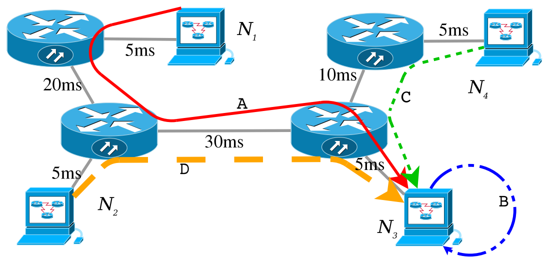


Fig. 1. Latencies in a distributed EPS.

about the applied time model, and sketches the key ideas of the event ordering. Section IV then presents the details of our migration algorithm. We carefully consider delay differences and guarantee an in-order event processing at all times. We evaluate our method under real-life conditions in Section V and discuss the runtime migration when processing position sensor data from a Realtime Locating System (RTLS) in a sports application.

Runtime migration is needed for this use case. Assume that we locate players in a soccer game and we apply event-based processing on the position streams. Events such as ball hits, goals, or fouls are automatically detected by the EPS and used to assist referees of control camera control systems.

Soccer rule violations such as handball, fouls, etc. are punished more severely if players are inside the penalty area. There we not only need to process the players' position events in more detail but the EDs are computationally intense as, for example, they (try to) derive the players' intentions. Hence, CPU loads get unbalanced and the system fails if EDs are not migrated soon enough, see. Sec. V-B.

II. Related Work

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

CR/TR-Motion [1] uses checkpoint/recovery and trace/replay to achieve a fast migration of VMs. Checkpoints from the source VM are recovered at the destination, and call traces from the source are replayed so that both machines are consistent. However, the authors do not forward data and do therefore not consider that the order of incoming commands may be different at the new host.

Bradford et al. [2] deal with the transfer of a local persistent VM state. After migration, network connections are redirected to the new host and commands from old connections are forwarded. The old VM is shut down as soon as all the old clients are gone. However, in contrast to our approach, both machines not only have to run in parallel while commands are being forwarded, but the order is ignored in which commands are received over the network.

MOSIX [3] is a cluster management system that supports interactive processes and resource discovery for workload distribution. As MOSIX migrates processes and redirects system calls it has the same disadvantages as the previous approaches.

Pipelined State Partitioning (PSP) [4] time-slices stateful operators (multi-way window-based join operations) and then distributes the fine-grained states over a cluster to form a virtual computation ring. The states are relocated if CPU loads are unbalanced. However, although the operator itself is stateful, the states are not. Moreover, the authors assume that input streams are equally ordered on each host.

Liu et al. [5] combine *state spill* and *state relocation*, and use decision making to use one of them. However, both do not work if events need to be reordered.

Endler et al. [6] provide a comparison of various handover techniques for mobile devices from which *new/old domain service* ensures a total order. However, both methods forward

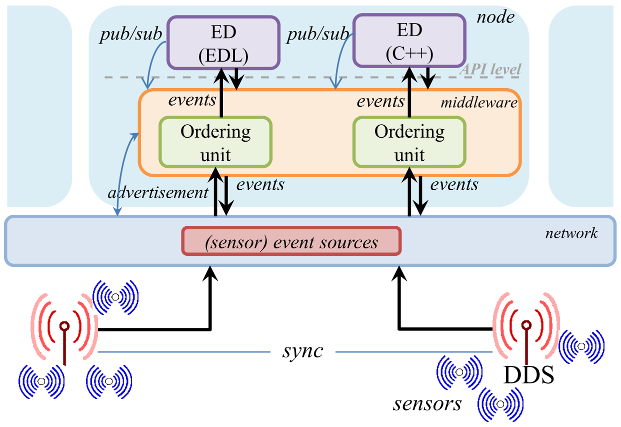


Fig. 2. Distributed publish/subscribe EPS.

the complete event stream until the new service takes over.

Most of the previous approaches considers the order of incoming commands and/or data explicitly. That is because usually the source of the commands, i.e., the user's workstation, is static and commands are still received in correct order. However, if we deal with multi-user VMs, problems may occur if two users try to modify the same file. At the original VM, user A's command may be received first, whereas at the migrated VM, user B's command will be first. The VMs are then out-of-sync. Approaches like CR/TR-Motion would repeat the recovery and replay process in such unlikely situations. However, for event detection such situations are very likely and migration would probably result in endless recovery and replay cycles.

III. Preliminaries

Fig. 2 depicts our distributed publish/subscribe-based event processing system. It is a network of several machines that run the same middleware to process sensor readings that are collected by a number of data distribution services (DDS), e.g., antennas that collect RFID readings. EDs are spread across the machines. An ED communicates subscriptions, publications and control information with the middleware that does not know the ED's event pattern; the ED is unaware of both the distribution of other EDs and the runtime configuration. The middleware implements a push-system with unknown subscribers. At system startup the middleware has no clue about event delays on other hosts but just notifies other middleware instances about event publications.

As it is difficult to manually implement EDs that process out-of-order events and developers often do not know the delays that their code may face at runtime, the middleware provides a personal event ordering unit per ED. For that, it extracts a local clock out of the event stream, see Sec. III-B. The middleware is thus generic and encapsulated, and does not use the application-specific event definition of the EDs.

A. Time Model Semantics and Definitions

The time model we assume is that sensor events are timestamped 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 is identified by a unique ID. An event instance is an instantaneous occurrence of an event type at a point in time. An event has two time-stamps: an occurrence and an arrival, both are in the same discrete time domain. An event appears at its occurrence time-stamp ts, or just time-stamp for short. At arrival time-stamp ats the event is received by a particular EPS host. The occurrence time-stamp is fixed for an event at any receiver whereas the arrival time-stamps may vary.

Out-of-order event. Consider an event stream $e_1, \cdots e_n$. Events of type $\mathtt{ID}$ are used to set the local clock. Then e_j is out-of-order if there do not exist e_i, e_k, with $e_i.id{=}e_k.id{=}\mathtt{ID}$ and $e_i.ats{\leq}e_j.ats$ so that $e_i.ts{\leq}e_j.ts{\leq}e_k.ts$, i.e., e_j.ats does not fit between the two consecutive clock updates.

B. Self-Adaptive Ordering Units

K-slack [8] assumes that an event e_i can be delayed for at most K time units. Hence, the ordering unit of a particular ED that takes a stream with potential out-of-order events and produces a sorted event stream needs K as the maximal delay of all subscribed events and a K-sized event odering buffer.

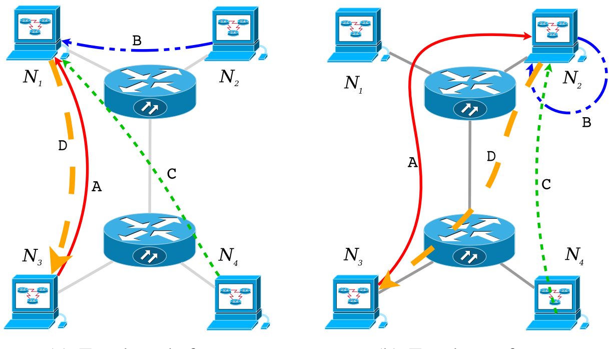
The dynamically generated ordering unit is mounted between the event input stream and the ED, extracts a local clock clk out of the event stream, and delays both late and early events as long as necessary to avoid out-of-order events.

While there are EPS that use programmer-configured K-values, it is better to measure event delays at runtime and to configure optimal K-values dynamically [9]. One reason is that EDs in practice often form an event processing hierarchy to detect the events of interest. For instance, to trigger smooth camera movements by events, all the K's must be as small as possible but as large as necessary. Overly large K's result in large buffers and high latencies for EDs along the upper processing hierarchy.

Notice that the migration of an ED not only affects the input delays of the migrated ED but may also affect the input delays of upper level EDs that subscribe events generated by the migrated ED although the subscribers do not partipate in the migration. Sudden increases of K may lead to out-of-order processing on higher level EDs. Our migration algorithm also addresses this problem.

IV. Runtime Migration

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



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

Fig. 3. Event detector migration.

Consider the network topology depicted in Fig. 3. Before migration (Fig. 3(a)) an ED runs on host N_1 and subscribes three events $\mathtt{A}$ (published by N_3), $\mathtt{B}$ (published by N_2), and $\mathtt{C}$ (published by N_4). It generates event $\mathtt{D}$ (subscribed by N_3). When we migrate this ED from N_1 to N_2 (Fig. 3(b)), the sources of the subscribed events remain the same, but their delays most definitely change. The delay of $\mathtt{B}$ at the new host N_2 shrinks, because after migration $\mathtt{B}$ is a local event. The delays of $\mathtt{A}$ and $\mathtt{C}$ may or may not shrink. Note that the delay of $\mathtt{D}$ at N_3 may also change even though the subscriber at N_3 does not participate in the migration.

Unless delays shrink for all involved events, a naive migration is likely to fail because the migrated ED (or any ED on a higher level of the hierarchy) no longer sees the subscribed events in correct order as its K-sized ordering buffer that worked well on the old host is too small for the new host.

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

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

A. Cooperative Handover

The key migration idea is a cooperative handover in which the new host not only subscribes the necessary input events from their sources, but the old host also forwards those events to the new host. The new host can then derive the correct order by combining delay information from events that arrive along two paths. Our algorithm consists of two different, interleaved steps: (1) migration and (2) delay adaption and echo cancellation. Fig. 4 depicts a sequence diagram of the

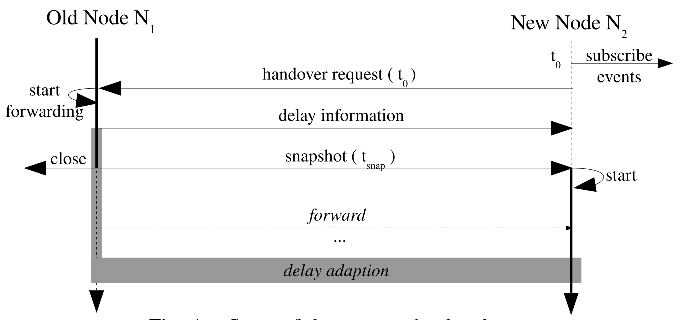


Fig. 4. Steps of the cooperative handover.

cooperative handover. Below we explain the two steps in detail by taking up the example from Fig. 3.

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

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

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

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

Step 2: Delay adaption and echo cancellation. With the above migration step we can correctly move a running ED from N_1 to N_2. Nevertheless, at N_2 event delays and therefore

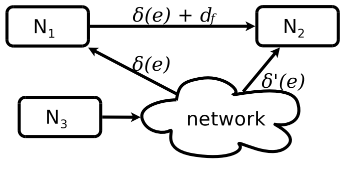


Fig. 5. Delay $\delta(e)$ of event e before migration, $\delta'(e)$ after migration; d_f is the forwarding sub-delay.

Algorithm 1: Delay Adaption and Echo Cancellation.

```
Data: $ED_{new}$, $d_f$, $DelayList\ delays$, $EventList\ echo$
begin
    for $delay\ d : delays$ do
        $d \leftarrow d + d_f$;
    $ED_{new}.K \leftarrow K \leftarrow \max(delays)$;        // set $K$
    $Event$ e;
    while $e.receive()$ do
        if $e.id \notin ED_{new}$.GetSubscriptions() then
            continue;
        if $e$.isDirect() then
            $delays.at(e.id) \leftarrow clk\text{-}e.ts$;
            $K_n \leftarrow \max(delays)$;
            if $K_n \neq K$ then
                ED$_{new}.K \leftarrow K \leftarrow K_n$; // r.-fit $K$
                $NotifyDelayChange()$;
            $StopForward(e.id)$;
        if $echo.contains(e)$ then
            $echo.erase(e)$;            // duplicate.
        else
            $echo.add(e)$;
            $ED_{new}.pushToOrderingUnit(e)$;
```

suitable K-values may be different, as the delays of the subscribed events may have changed. Instead of starting with a fresh K'_D=0, it is initialized according to the delay information received from N_1 in the first packet, i.e., K'_D = max(δ(A)+d_f, δ(B)+d_f, δ(C)+d_f) = max(30+5, 10+5, 20+5) = 35ms.

As shown in Fig. 5 the new host receives all events twice. An event e reaches N_2 directly with a delay $\delta'(e)$ and it also reaches N_2 with a delay $\delta(e) + d_f$ because it is forwarded by N_1. As soon as N_2 receives an event along the direct route, it can update its K by using $\delta'(e)$ instead of $\delta(e)$+d_f. For instance, a delay of δ'(A)=25ms for the first directly received A reduces K_D to max(**25**, 15, 25)=25ms.

While receiving events twice is advantageous for initializing K and for setting up the reordering unit, echoed events would pose problems for the ED. To make sure that the event ordering unit only sees an event once, one of them needs to be dropped so that only one event is in the ordered input stream of the ED. Our echo cancellation works as follows: for each event type there is a first time when the new host sees both an event and its echo. Before that moment, events with lower delay are passed along the ordering unit (and late events are dropped). Afterwards, as soon as the direct event is received, to get rid of the echos at the end of the transition phase, N_1 is notified that this event type does not need to be forwarded any longer. Algorithm 1 gives the (abbreviated) pseudo code of the delay adaption and echo cancellation.

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

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

V. Evaluation

We have analyzed position data streams from a Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. The RTLS tracks 144 transmitters at 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Players are equipped with four transmitters, one at each of their limbs. The sensor data consists of a time-stamp, 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. Since soccer events like pass, double pass, shot on goal, etc., happen within a fraction of a second, event processing must ensure that events are detected in time so that a hierarchy of EDs can for instance control a system for smooth camera movement or help a reporter to work with the live output.

A. Comparison with classic migration

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

In this setup, we migrate an ED for detecting a *pass*. It subscribes to four different event types and emits the pass event. Other EDs behave similarily.

Approaches that ignore the order of events [2], [10] only copy the snapshot and inevitably fail. Approaches that run in parallel [2], [5] until all delays are correctly measured are less efficient than those that use stream forwarding [6]. We denote the latter as *classic* approaches and compare our migration with them.

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

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

Fig. 6(a) shows the K-values of the ED migrated in the classic way (first replay, dotted line, K starts at 40ms). This ED fails 5 times within the first 17 seconds after migration. Whenever it fails, K is increased to prevent future misdetections (as dynamic K-slack buffering would do). One event type first shows up after 30s but did not increase K.

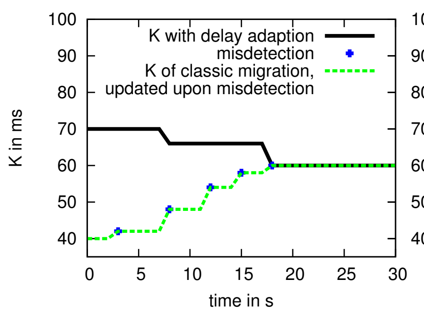
In comparison, in the second replay our cooperatively migrated ED subscribes to the events twice. It receives the events on the direct path (as the naively migrated ED does) and a forwarded copy from the old site which includes the forwarding sub-delay of 30ms. Hence, our technique makes the ED start with a higher K-value (40+30=70ms) at the beginning. And whenever direct events show up early, the K-value is lowered. As shown in Fig. 6(a) our novel technique avoids misdetections as the K-value is always large enough. Moreover, K of the migrated *pass* ED is just 17% too large at the beginning (70ms instead of 60ms) and melts down quickly. This is a small price tag for perfect detection.

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

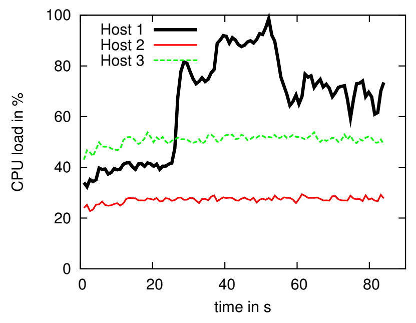
In the example of Fig. 6(a), the classic migration with event forwarding sorts the events on the original host for 30 seconds with a delay of K=40ms before they are forwarded to the target host with an additional forwarding delay of d_f=30s. The total delay on the target host is 70ms. This goes on for 30s, before the migrated ED can switch to the newly configured reordering unit (K=60ms). This results in the same correct detection that our cooperatively migrated ED achieves, but without the early meltdown of K, i.e., with a higher accumulated latency.

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

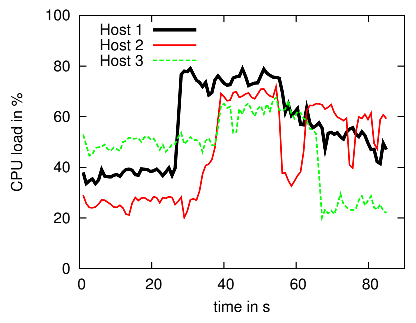
The overhead of the correct but inefficient classic migration with event forwarding is prohibitive since migration is triggered often because of bandwidth or load bottlenecks.



(a) Delays and K after state transition.



(b) CPU loads without migration.



(c) CPU loads with migration.

Fig. 6. Evaluation results when applying migration.

B. Use Case

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

For the evaluation we pick up the use-case described in Sec. I. Fig. 6(b) shows a timeline of the CPU load on three hosts. At the beginning, EDs are somehow distributed over the machines with a rather balanced load. After about 25 seconds, the load on host 1 rises rapidly because after a corner kick many players run into the penalty area. This triggers many EDs and increases CPU consumption. Without migration, CPU 1 gets fully loaded and the system fails as it can no longer perform event detection although the other two machines run at a moderate load. Note that in Fig. 6(b) the system fails at after about 53 seconds when it hits 100% load. The drop of the load afterwards is an artifact of the use case. Due to the misdetections of some EDs, other EDs misdetect as well and reach simpler and invalid states. The system keeps misbehaving for a long time after the peak. With migration, there is no peak and no such misbehavior. Migrating EDs to hosts 2 and 3 smoothes load distribution, see Fig. 6(c).

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

If EDs are migrated with techniques that run EDs in parallel [2], [5] the loads on nodes 2 and 3 increase as in Fig. 6(c), but node 1 behaves as in Fig. 6(b). With classic migration based on stream forwarding [6] we see similar CPU loads as in Fig. 6(c). However, the difference is the immense networking overhead that lasts for a long time.

VI. Conclusion

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

Future work will optimizes the ED distribution by heuristics and try to further reduce the duration of forwarding.

Acknowledgements

This work is supported by the Fraunhofer Institute for Integrated Ciruits 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] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proc. 18th ACM Intl. Symp. High Perf. Distrib. Computing*, (Garching, Germany), pp. 101–110, 2009.

[2] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proc. 3rd Intl. Conf. Virt. Exec. Env.*, (San Diego, CA), pp. 169–179, 2007.

[3] A. Barak, A. Shiloh, and L. Amar, "An organizational grid of federated MOSIX clusters," in *Proc. 5th IEEE Intl. Symp. Cluster Computing and the Grid*, (Cardiff, UK), pp. 350–357, 2005.

[4] S. Wang and E. Rundensteiner, "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing," in *Proc. 12th Intl. Conf. Extending Database Technology Adv. in Database Technology*, (Saint Petersburg, Russia), pp. 299–310, 2009.

[5] B. Liu, M. Jbantova, and E. A. Rundensteiner, "Optimizing state-intensive non-blocking queries using run-time adaptation," in *Proc. 23rd Intl. Conf. Data Eng. Workshop*, (Istanbul, Turkey), pp. 614–623, 2007.

[6] M. Endler and V. Nagamuta, "General approaches for implementing seamless handover," in *Proc. 2nd Intl Workshop Principles of Mobile Computing*, (Toulouse, France), pp. 17–24, 2002.

[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] 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, Canada), pp. 67–74, 2007.

[9] C. Mutschler and M. Philippsen, "Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams," in *27th Intl. Conf. Par. & Distrib. Processing Symp.*, (Boston, MA), May 2013.

[10] 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.