Towards a Distributed Self-Optimizing Event Processing System for Realtime Locating Systems (RTLS)

Doctoral and PhD Workshop Paper

Christopher Mutschler, Advisor: Michael Philippsen
University of Erlangen-Nuremberg
Department of Computer Science (Programming Systems Group)
Martenstr. 3, 91058 Erlangen, Germany
{christopher.mutschler,philippsen}@cs.fau.de

ABSTRACT
From the ubiquity of sensor data follows a need for automatic processing to extract valuable information. Realtime Locating Systems (RTLS) provide many parallel position data streams for interacting objects. Event-based processing has turned out to be the method of choice for the reactive analysis of such position data streams.

Due to insufficient knowledge about the object and system behavior, and thus the event load at runtime, we create a distributed event processing system (EPS) that adapts to the variation in the observed environment. For that, events are ordered with respect to their delays, event detectors are migrated between nodes, and the system is scalable as the number of trackable objects and sensors grows.

Related work turns out to be insufficient because it either lacks capabilities for distributed processing, cannot order events, or does not handle negative patterns. Here we present a PhD thesis that addresses the challenges of massively distributed, self-optimizing, and fault-tolerant event processing of high data rate sensor streams. We line out the essential techniques that are already developed and sketch future steps towards the solution.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed Applications; D.1.3 [Programming Techniques]: Concurrent Programming—Distributed Programming

General Terms
Algorithms, Design, Reliability

Keywords
distributed event processing, publish/subscribe, middleware, fault-tolerance, mobility, self-management, sensor streams

1. INTRODUCTION
With wireless localization techniques we can track moving objects in almost arbitrary environments. Realtime Locating Systems (RTLS) can track many objects simultaneously and provide accurate positions at high data rates. By automatically processing this position information, we ease analytical processes in many fields of application. Consider the quantitative analysis of sports games like soccer.

To detect meaningful data about players, teams and games, and to display information (e.g. percentual ball possession) on television, we currently require several human observers that count certain events. Further scenarios arise in training. To simulate authentic game situations, cones are placed at specific points on the field, time keeping is assured by light barriers, and velocities are either calculated as a result of previous measurements or are directly determined with lasers. Such setups require both a lot of time and effort for construction and execution, and are also highly error-prone. Qualitative results are thus highly doubtful.

Analyzing the position streams provided by RTLS aids in the implementation of both scenarios. The aim is to automatically derive events such as passes, ball possessions, or shots from the position data streams. In training, the system measures times and sends them to, for instance, a tablet PC on the field where the trainer can immediately use this information to improve the training or to compare the performance to previous exercises.

Event-based processing of position data streams has turned out to be a valuable method of choice. Since many interesting incidents, i.e. events, depend on common lower-level events, it is obvious to calculate these basic events only once, and to form an event hierarchy from those basic events. Figure 1 shows such an event hierarchy to detect a blocked shot on goal. By splitting a pattern into various sub-events, we can reuse those sub-events for the detection of other high-level events. Lower software complexity, no coupling be-

![Figure 1: Hierarchy for blocked shot on goal [1].](image-url)
between event implementations, no memory synchronization, implicit parallelism, and easier maintenance are further advantages that facilitate the development of such event-based systems.

Due to the mobility of RTLS, it is desirable to minimize both costs and setup times. Therefore, we make use of smartphones to form a wireless ad-hoc network in order to reduce the need for a sophisticated server infrastructure that is for example built into a stadium. The aim of the presented thesis is to develop essential methods for this distributed EPS to achieve an automatic adaption to the environment, self-optimization, and reliability and fault-tolerance to encounter temporal unavailability of nodes. Event detectors can be implemented without caring about runtime configurations or machine/link breakdowns, and the underlying framework assures a correct event dissemination, and an optimized CPU and network usage.

2. CHALLENGES

To reach the aforementioned goals, the EPS has to meet some requirements. We identify and characterize them below.

Data rate. The system must process high data rate sensor streams. For the above-mentioned applications, we receive 50,000 positions per second, where each player is equipped with four transmitters, one at each of his limbs. Each position packet carries not only the coordinates but also velocities and accelerations for each direction as well as measurement qualities (Quality of Location, QoL). Therefore, buffers must be used with care and events must be processed with low latency.

Event ordering. Individual position events mostly arrive out of order but need to be processed in order if they are related to each other. Our experience shows, that naive event detectors often cause misdetection and system failures when they process out-of-order events. Since event delays are not known a-priori and add up along the event processing hierarchy, naive buffering is no solution. Event detectors themselves may also introduce delays on top of certain event type delays. Buffers for sorting must be as small as possible to avoid overhead but as large as necessary to properly order event streams. But overly large buffers result in high memory demands and, again, in higher output delays.

Scalability. A single node cannot process all events either due to its computing power or its battery capacity. Scalability must be achieved by distributing event detectors over available nodes in a network. This amplifies the problem of buffering and sorting of events.

Migration. To enhance the load balancing of event detectors, we need a technique to migrate them at runtime. The challenge is not to miss events and to always achieve a correctly ordered event input. Since event delays vary between nodes, we cannot just copy and start an event detector on another machine. Migration decisions besides runtime have also to consider the cost of sending data over wireless network and battery consumption.

Reliability. The advantage of our event-based approach is to reuse calculations and to form a hierarchy of events. But at the same time we need to take care that those events are safely delivered because otherwise event detectors may miss important information and get suspended in invalid states. Link breakdowns are highly probable in wireless communication.

Runtime optimization. An arbitrary distribution of event detectors is rarely the best solution. But an offline optimization to find the best configuration is not possible since the event load is only predictable and changes with time. Therefore the configuration must be optimized continuously. Moreover, there is not always the possibility to establish a centralized coordinator to optimize the global state of the EPS. Especially for a mobile setup, we need cost models to optimize the distribution of detectors by factors like battery consumption, communication usage, and latency. In addition to that, a set of event detectors may temporarily be shut down because there are no subscribers. For that we have to ensure that the correct state of the detectors can be restored when starting them again.

Mobility. When using mobile RTLS, we want to reduce the need for servers to minimize both system costs and setup times. Our primary goal is to use a combination of a stationary server and an ad-hoc network of smartphones for the distributed processing, see Figure 2. The components can either communicate over wireless LAN or broadband communication such as UMTS. It is important to optimally distribute battery (and communication) intensive processing over the mobile and stationary devices. Moreover, mobile devices may join and leave the distributed EPS dynamically. The system must ensure that event processing is continuously distributed over the temporary available units.

3. RELATED WORK

Current systems and methods only partly fulfill the above mentioned requirements. Although several EPS have been presented for processing out-of-order event streams, they do not meet the plurality of demands. The ordering techniques implemented in SASE [2, 3, 4, 5] fail if event processing is distributed over several machines. Whereas Cayuga [6, 7, 8, 9] may be distributed, it fails if event delays cross epoch boundaries. Such boundaries are inevitably crossed when events are generated with timestamps from the past (backsetting delays\(^1\)). Moreover, both systems

\(^1\)Backsetting delays occur when events need some prospective events for estimation, and can thus only be inserted in the event stream long after they have actually happened.
assume that maximal event delays are known a-priori. In most cases, we do not have this knowledge since delays depend on the runtime configuration. A maximal fixed border is often not tolerable or even possible due to the aforementioned reasons, and a dynamic initialization of those delays is not part of any of the systems. Complex Event Detection and Response [10] handles out-of-order events by retracting incorrect output and by adding correct, revised output in turn. But this triggers a cascade of retractions for almost every output event since out-of-order events are predominant. Approaches that require event detectors to be implemented in a complex event language [11, 12] overly restrict the flexibility to express patterns and situations. Hence they are no valuable options. As it is better to not restrict the developer of event detectors at all, the underlying system must manage and completely hide the ordering semantics from the event detectors. The migration of event detectors that require a local persistent state at runtime has not yet drawn any interest. Existing systems for distributed event processing generally provide the ability to move event detectors but do not address persistent states and varying event delays. To achieve a balanced CPU and/or network load, or to shutdown nodes for maintenance, we need a technique to safely migrate an event detector at runtime while processing continues. Similar work is found in the area of live migration of virtual machines (VM) [13, 14, 15]. However, the requirements are different and the algorithms cannot be applied or adapted for EPS, because ordering of input commands from different sources is not a critical issue in the VM context as in such unlikely cases the VM migration fails.

4. ACHIEVEMENTS & ONGOING WORK

In this section we draft our steps to achieve correct event ordering (Section 4.1) and save runtime migration of stateful event detectors (Section 4.2). Both methods build the basis for a completely self-optimizing event processing system. There is more work to be done to solve the fault-tolerance and reliability issues (Section 4.3), and to address mobility (Section 4.4). Both sections sketch the roadmap of ongoing work.

4.1 Event Ordering

Naive event detectors often assume timely ordered input events. But in practice, the order in which events are received does not reflect the timestamp order. Our experience shows that it is difficult and error-prone to implement event detectors that can process out-of-order events, since event delays are not known before runtime.

To provide an ordered input event stream for each event detector, we need to derive knowledge about the individual event delays in the runtime system configuration, i.e., the distribution of event detectors over the available nodes in the network and the object behavior. For example, the velocity of moving objects affects the delay. In [16] we have presented a way to order events if there is no global clock available. The central idea is that each node identifies event types with equal delays and sets its local clock to the timestamp of these events whenever such an event is received. The maximum delay of an event \( e_i \) at node \( s_k \) can then be calculated as \( \delta_i(s_k) = e_i.ts - clk \), where \( clk \) is the next update of the local clock. With these event delays for a given configuration, we implement a specific event ordering unit that generates an ordered input event stream with many events for each particular event detector. Consider an event detector that processes events A, B, and C, see Fig. 3. When we derived event delays and a local clock \( clk \), we can setup an ordering unit that produces a correctly ordered event input stream for the event detector.

We have also solved the problem with sudden increases in certain event delays. When event delays increase or the events that update the local clock no longer reflect the real time, i.e., the delay of those events is no longer equal, without our solution the ordering units stumble over higher event delays and the event detectors may receive out-of-order events. The key idea is to keep track of the recently measured delays and use their standard deviation to better estimate the expected delay of this event type. The ordering units then absorb sudden delays and adapt to the changes rather than to disseminate out-of-order events.

In addition to event ordering and to deal with sudden changes in the delays we also found a way to solve the problem that initially (when the system starts) there are no measurements available to properly calibrate the ordering units. For that we presented two different solutions, called Iterative Delay Calculation and Semi-Configured Delay Estimation. The former method repeatedly runs the EPS and iteratively sets the delays in the ordering units. The latter methods work considerably faster by identifying the total delay of an event by a sum of sub-delays. These sub-delays are used to propagate pseudo events through the network. We measure the delay on these pseudo events as described before and use these delays to parameterize the ordering units.

The result of our methods is, that when we start an EPS with an arbitrary configuration, it initializes fast and correctly, and adapts the delays due to changes in the environment without propagating out-of-order events to the event detectors. Buffers are kept at minimal uses to avoid overhead but are large enough to guarantee a total order. We cope with the challenges of high data rate and the event ordering. Furthermore, we have built the basis for scalability since event detectors can be arbitrarily distributed over several machines. There may arise a need to modify the existing methods in order to fulfill the requirements of runtime optimization and mobility since devices may temporarily shut down. We also may need to consciously withhold events in the backend in order to process them in bursts on the mobile devices. Moreover, we currently use the position data stream to set our local clocks. Due to the high data rate and communication efforts in future we may need to set the clock from a combination of event timestamps and local system clocks.
4.2 Runtime Migration

Due to system failure or maintenance, or because machines get overloaded or even exhausted, we need to move an event detector from one node to another at runtime.

To achieve self management and/or optimization in distributed EPS we presented a solution to migrate stateful event detectors at runtime in [17]. The key idea is to take a time-stamped snapshot of an event detector and to use this snapshot for the initialization of the detector at the new host. The new host uses the timestamp to identify the events that still need to be processed after initializing the detector with the snapshot. Therefore, the event detector is restored to the valid state and can continue with the processing at the new node.

We also found a solution to adapt the event delays on the new node so that correct ordering is guaranteed even after migration, and described it in [18]. By forwarding subscribed events to the new host we can set an upper bound of the maximal event delays in our new ordering units. Forwarding of an event type can stop as soon as the new node measured its own delay for it. Thus the new node provides a totally ordered event input stream to the migrated event detector by comparing the delay information from both nodes. As a result, we safely migrate an event detector at runtime by simultaneously ensuring total order for the new event detector. Event detectors are copied and immediately shut down, no event is processed twice, and network overhead is kept at minimum.

We cope with the challenges of runtime migration. Moreover, we have built the basis for the runtime optimization, reliability, and mobility issues. However, for good mobility, the runtime migration needs to complete considerably faster. Currently, migration is completed once all involved events have been received at least once. But since nodes may disappear quickly, we need to improve our current method to achieve a faster delay adaption.

4.3 Fault Tolerance

With stateful event detectors, it is not tolerable that certain events are not delivered or event detectors stop working. If communication links temporarily break down or machines are suspended, event processing fails. As before, implementing fault-tolerant event detectors is no valuable solution since the programmer has no clue about the runtime configuration. For instance, an event detector whose subscriptions are local events does not need to care about communication errors since there is no communication link. And as before, fault-tolerant event detectors are error-prone since the complexity increases severely. Therefore, reliability and fault-tolerance must be implemented in the EPS.

For that, we are exploring the following options.

Reliability by redundancy. To ensure a continuous processing, an event detector is duplicated in the network to ensure, that in cases of errors, the redundant detector still disseminates valid output. Our current method for event ordering must be modified to take the maximum delay of both paths the original event detector and its duplicate. Echos are discarded. The result is a larger buffer for event ordering and additional CPU load even if no errors occur. However, errors are immediately handled and no switching times are introduced.

Reliability by snapshot recovery. We can continuously take snapshots of event detectors and distribute the snapshots over the available nodes. When a node crashes or communication breaks down, the node(s) that host(s) the snapshot(s) can activate the snapshot and process the buffered events. This does not introduce relevant CPU load. However, we need time to recognize an error and to replay the snapshot. It is to evaluate in which cases those switching times are tolerable. Maybe the runtime migration algorithm has to be modified because it does not guarantee an upper bound for completion.

For both techniques we further need to determine the appropriate node that hosts the reliability for a detector. At best, this node subscribes and already processes a subset of the subscribed events so that the networking overhead is kept low.

Algorithms for the selection of those nodes are being planned to investigate in the ongoing PhD thesis. As part of the ongoing PhD thesis we plan to solve the issues of reliability and will build the basis for mobility.

4.4 Mobility

As stated in Section 1, RTLS should be mobile so that they can be installed in short time. To reduce both installation time and cost, we want to reduce the need for server infrastructures for the distributed processing of events. Consider the example in Figure 2 again. Our target infrastructure consists of smartphones in the immediate vicinity of an access point and (optionally) a stationary server.

To achieve this kind of mobility we have to address several technical problems. First, wireless devices that are part of the distributed EPS may quickly leave the range of the data dissemination and can no longer participate in processing. On the contrary, other wireless devices may get in range and participate in the processing. Since the distribution of event detectors is highly non-stationary we need to migrate event detectors much faster than in non-mobile infrastructures. The delay adaption may not complete before devices get unavailable.

Second, some of the event detectors may not be available on some devices. Before we migrate an event detector to a device, we need to send the executable code to this event detector and load it dynamically into the runtime environment.

Third, we need to investigate a combined cost model that optimizes CPU use, battery consumption, latency, and communication cost. Without such a cost model, the system would probably completely exhaust particular devices so that a correct operation of the whole system cannot be guaranteed any longer. Ongoing work of the PhD project will solve the mobility and runtime optimization issues to achieve a highly mobile distributed event processing network on which event detectors can be dynamically allocated.
in communication links and cutoffs of some devices. That way, we can also observe how event detectors migrate between the simulated devices. We can also demonstrate the efficiency of the optimization according to the developed cost model. For the real world test we use Android smartphones. The current implementation is written in C/C++ that can be natively executed when it is properly compiled. The trainer can configure training exercises on a tablet PC that generates event detectors for an event hierarchy. These detectors can be distributed over the available smartphones in the range of the access point. We evaluate the fault-tolerance and reliability by switching particular phones off and on or putting them out of range. The optimization is verified by examining the battery statuses and communication bandwidth, and the time for which the system is fully functional as more and more phones will go out of battery and switch off.

6. CONCLUSION
We motivated the problem of distributed event processing over high data rate sensor streams and outlined the challenges and requirements. We also outlined open topics when dealing with the mobility of computing nodes. Related work turns out to be insufficient because it either lacks the ability for distributed processing, correct event ordering, or back-setting delays.

We sketched the basic ideas of our method, which already achieves adaptable delay handling and runtime migration of event detectors. Furthermore, we presented ongoing work packages of the PhD project that will achieve fault-tolerance, mobility, and self-optimization.

REFERENCES