Abstract—Distributed computing systems often strive to decouple their communicating components (threads on a single node, or nodes in a network) from each other with respect to time, space/location, or flow/synchronization, so that they can operate as independently of each other as possible. One approach to let components in a distributed system communicate in such an independent, decoupled fashion is to follow a Publish/Subscribe paradigm.

It is, however, not at all obvious that such distributed systems then remain capable of real-time operation. We are particularly interested in applying such features to Wireless Sensor Networks (WSNs), where a small footprint is required and traditional middleware approaches are far too heavyweight.

We show that it is possible to combine the Publish/Subscribe paradigm with real-time features. We demonstrate this by a blackboard-based Publish/Subscribe system with real-time extensions that we have implemented on MAN-TIS OS [1]. We show that timeliness of the system’s actions can be achieved without re-introducing tight couplings.

Keywords—real-time; publish/subscribe; wireless sensor networks; blackboard

I. INTRODUCTION

Distributed systems are often designed in a way that its components (usually computers or sensor nodes) can act independently of each other. This is done, e.g. to maximize scalability or robustness of the distributed system.

One crucial aspect of distributed systems is how their components communicate with each other. Several concepts are conceivable, such as tuple spaces, message passing, Remote Procedure Calls (RPCs), or a publish/subscribe concept. The latter is interesting because it allows the components to be decoupled from each other with respect to time, space, and flow.

The question to be addressed in this paper is how a publish/subscribe concept relates to real-time concepts. These two concepts seem to clash at first glance, because the publish/subscribe system decouples communication in time and flow, i.e. makes it asynchronous; real-time execution on the other hand suggests implicit synchronization or time coupling. We show that it is possible to combine the publish/subscribe system with real-time constraints without sacrificing either the decoupling or the real-time guarantees.

A common problem is how long communication between the computers in a distributed system takes and how reliable it is. We will assume that events can be transferred over a reliable communication channel that operates with minimal delay, i.e. we assume that messages are available at the receiver once they have been completely sent, and we will disregard packet generation times. Based on these assumptions, we can simplify our demonstration to inter-process communication on a single computer. For communication channels with bounded delivery times, and bounded-diameter networks with constant overhead, it is straightforward to generalize our concepts to a distributed case as well.

Finally, we will investigate this problem under the additional constraint that the solution needs to be suitable for WSNs, where there is only little computing power and little memory available.

We will continue this paper with an introduction to the two underlying concepts (publish/subscribe and real-time; the remainder of this Section), then present our concept (Section II), and outline our prototypical implementation and results from test runs (Section III). We will show how our proposal relates to other work (Section IV) and finally conclude (Section V).

A. Publish/Subscribe model

The Publish/Subscribe model is a versatile approach to manage information sharing and communication. Eugster et al. [2] describe the basic Publish/Subscribe model components (Figure 1) as follows:

- **Events.** These are either messages or remote procedure calls, and they bear an event identifier or event type to classify them. For instance, an event type might represent observations such as a stock quotation or a room temperature; events might convey information on the internal state of the communication system, such as link state, current queue depth, remaining battery charge, or free memory.

- A central **EventService** that serves three purposes. It allows publishers to register themselves and publish events. It also allows subscribers to subscribe or unsubscribe to one or more event types; subscriptions can contain patterns or filters that are applied to published events in order to select which events the subscriber wants to see. Finally, the EventService actually notifies subscribers, that is, it dispatches events from publishers to subscribers according to their current subscriptions.

- There are publishers that provide data to the EventService, by triggering events.
• **Subscribers** will be notified of the published events if the latter match the respective subscribers’ filters.

This approach enables three important decouplings that are useful for scalability and independence.

First, there is a decoupling in space or naming. Publishers and subscribers need not know each other; the EventService acts on their behalf to relay the events.

Second, Publish/Subscribe (P/S) decouples communication partners in time. The EventService can buffer events so that the publisher need not be active at the same time as the subscriber.

Third, there is a decoupling in flow or synchronization. This property differs from the previous one in that the publication or notification actions are asynchronous, thus subscribers need not constantly poll for events, nor need publishers wait for notification or delivery to complete.

### B. Real Time

Real-time capability means that a timely response to time-critical events is assured. Practical examples might be alarms of certain observed environmental events or periodic sampling of physical variables.

To simplify matters, we assume hard real-time requirements, meaning that deadlines must not be missed. Consequently, we also assume that the whole distributed system will not be overloaded. Otherwise, deadline violations would be unavoidable.

For definitions of real-time related terms, we refer to Buttazzo’s text book [3]. Specifically, we will use the term “lateness”, a time value which describes how much too soon (negative) or how much too late (positive) the desired action completes, and “tardiness”, which is the lateness if positive, otherwise 0.

### II. Concept

It seems that real-time requirements contradict the decoupling in time and flow in publish/subscribe systems. This is not the case, however: A decoupling in time does not imply that there is an unbounded delay between an event being published and notified to the subscribers. Instead, we can modify the EventService such that it dispatches the notifications within the specified deadlines.\(^1\)

Also, a coupling of the flow from publisher to subscriber need not be reintroduced for a timely notification to occur if the EventService assumes responsibility for a timely delivery. We can thus maintain the decoupling in time, space, and flow, and additionally also give real-time guarantees.

To achieve real-time operation in a publish/subscribe system, we will need two extensions:

1. The subscribers can specify a deadline during the subscription that specifies the maximum delay until the subscriber is notified of a new event. Further extensions beyond the deadline are of course conceivable to support other purposes. In particular, it may be useful to apply cost or weight or priority information to improve scheduling if overload shall be allowed.

2. Of course it is also possible to let publishers specify the deadlines, or to make the deadlines a trait of the individual event type. Regardless of the way that the deadlines are specified, the blackboard is aware of them and can cause appropriate action. Thus, the choice of who will specify the deadline can be left to the application using our concept.

On the assumptions (1) that the system is not overloaded, meaning that it is actually possible to fulfill all deadlines, and (2) the schedulability is known and given, it will suffice that the EventService reorders the dispatch of notifications appropriately to attain the required real-time operation.

If the system is overloaded, two options are conceivable: either the system rejects new publishes that would make an existing schedule non-feasible, or it accepts the job and degrades performance to a best-effort, but with missed deadlines. Here, additional cost information can allow the scheduler to make better choices as to which deadlines may be missed and which should be kept.

For strict real-time fulfillment, we also need to make sure that event matching effort does not jeopardize the deadlines. We can achieve that by imposing limits on the maximum number of subscribers, and then subtract the worst-case matching times from the deadlines internally.

Since we do not need to add further information about possible senders or recipients, we are not reintroducing couplings that P/S systems have allowed us to remove.

### III. Prototype

In order to prove the concept, we have implemented a prototype of the proposed concept. It comprises a whole blackboard-based P/S system on MANTIS OS [1]. It supports subscription, publishing, and notification functionality. The blackboard is programmed as a thread within the OS. We have added two test applications that can be run without and with subscriber-specified deadlines. The description will start with the test environment, then continue with the data structure, then detail the applications. We will finally present results from test runs.

\(^1\)When communicating across a bounded-delay channel, or in a bounded-diameter network with constant overhead, we can also easily compensate for its delay: we subtract the upper delay bound from all deadlines.
A. Operating System/Scheduler Extension

MANTIS OS is a multithreaded operating system optimized for WSNs; it has a priority-enabled, time-slicing, round-robin scheduler. The default time slice is 20 ms long. Threads with elevated priority operate in a round-robin fashion until no more runnable threads are available at the same or higher priority level.

The second extension proposed is Section II is achieved by adding a real-time scheduler to MANTIS OS. In our implementation, we chose – without loss of generality – the Earliest Deadline First (EDF) scheduler because it is simple to implement and it provably minimizes the maximum tardiness [3].

We have modified the scheduler such that it exhaustively searches real-time threads for the one with the earliest deadline. This is then scheduled. If no threads with deadline are found, MANTIS OS’s regular scheduler takes control. We chose this implementation because the number of elements in the list is very low (MANTIS OS supports only 10 user threads by default, one of them is our blackboard thread), so that the overhead for more sophisticated administration would likely not pay off.

B. Data Structure Extension

The blackboard keeps several data structures for administrative purposes. One of them keeps subscriber data, and it records a thread pointer, a callback function to check if a published event is interesting to the subscriber, and a mutex. The mutex is initialized in “locked” state and gets released when a subscriber is notified. The subscriber uses a blocking call to lock the mutex, which it will obtain when a notification arrives.

The first extension described in Section II is implemented by adding a deadline field to the data structure that describes the subscription in the EventService.

C. Test Application Morse

This test application was written to demonstrate concept and implementation. It sends a Morse-encoded string from a sender thread to a receiver thread. The latter visualises the Morse signal by flashing a light-emitting diode (LED). In between, we have hooked up a proxy thread that takes real-time clock readings and derives latency for message delivery from those real-time readings. The proxy is not necessary for real deployments, but it allows us to more easily observe the system behaviour in our test. On the other hand, it increases the latency, because the event has to be published and received twice before it reaches its actual destination.

We run the application several times for 120 seconds each time, with a varying number (from 0 to 4) of background threads that never have real-time priority and that simulate load by running an endless while(1) { } loop. We run this whole set with real-time enabled and disabled for subscriptions, and compare the results.

In detail, our implementation creates two event types, one called “event_proxy” and one called “event_sender”.

D. Test Application Busy Subscribers

The second test application does busy-waiting after each of the blackboard-related actions, to see how the system behaves if the involved publishing and subscriber threads are busy doing other things beyond communication.

We start one publisher thread and two threads that subscribe to one separate event each. One subscriber uses a deadline of 30 ms, the other of 15 ms.

The subscriber threads wait for a notification to arrive. After its receipt, the notified subscriber thread busy-waits for 15 ms, to simulate data processing, then blocks, waiting for the next event notification.

The publisher thread publishes the first event, busy-waits for 10 ms, publishes the second event, busy-waits for another 10 ms, and then sleeps for 2 s, and repeats.

A sample for an application run with real-time features disabled is shown in Figure 2.

We run the experiment for one minute, with deadlines enabled and disabled, and each time with a background load of zero, one, or two non-real-time threads that are always busy. Again, we determine the latencies of the published event until the subscriber has received the event notification.

E. Results

The “busy subscribers” test application (Section III-D) has been run on a Moteiv Tmote Sky [4] sensor node. This
node type is based on a Texas Instruments MSP430F1611 ultra-low-power 16-bit RISC microcontroller with 48 kB Flash-ROM, and 10 kB RAM and very similar to the well-known Telos.B [5] node. MANTIS OS adjusts the microcontroller’s master clock (MCLK) to nearly 3.7 MHz.

Both test applications have been run on the WSim [6] software simulating this node type.

For the first application, we have analyzed how accurate the timings of the Morse output signal have been (Table I); we show the absolute maximum latencies observed during the whole 120 s that the test application was running in each run.

As we can see, the real-time enabled system provides accurate timing of the edges of the Morse signal, where the non-real-time system is up to 158 ms late (in real deployment, this could be halved, because the proxy is an artifact of our test application and not inherent to the system).

Results for the second application are shown in Table II (from WSim simulation) and Table III (from actual hardware test runs). We can see that without real-time scheduling, events arrive too late (non-zero tardiness), particularly the second event arrives often late, and prediction is difficult: the standard deviation for the first event is sometimes of the same order of magnitude as the actual figures.

With real-time enabled in the second application, we get very timely and predictable delivery: not a single event arrives late, we have low standard deviation of lateness, and events arrive very soon.

We thus obtain hard real-time operation with our extensions.

IV. RELATED WORK

Surprisingly, few results are known that relate to real-time P/S in WSNs or similar distributed systems; while documents on P/S by itself, real-time, or data distribution are readily found. The lack of such specific papers has already been mentioned by Sharifi, Taleghan, and Taherkordi [7], and Deng et al. [8]. Many sources address real-time aspects, but apply this term to describe a mere reduction of latency, without considering hard real-time behaviour, often even without regard to best-effort real-time scheduling.

Lalanda, Charpillet and Haton present “a real-time blackboard-based architecture” [9] that improves the response time of knowledge-based (expert) systems, in the domain of Artificial Intelligence. This is a specific kind of blackboard, and their approach is to approximate the result in order to reach the deadline by working to refine its result as long as the deadline permits; additionally the actual real-time demonstration appears pending. Our work does not require approximation.

De Boer, Gabbielli, and Meo present a “timed Linda language” [10]. Linda, however, assumes a global shared memory that it calls “tuple space”, which requires communication in its implementation. Our proposal does not require a global shared memory.

The Object Management Group (OMG) Data Distribution Service [11] specification describes a Quality of Service (QoS)-enabled Data-Centric Publish-Subscribe (DCPS) layer. DDS’s notion of QoS also comprises hard deadlines, however the complexity of interface and layering make DDS/DCPS appear too heavyweight to be applicable to WSNs today.

Similar considerations apply to other middleware approaches, they often require Java 2 Enterprise Edition or are based on CORBA, or are otherwise tied to the Internet architecture [8] and hence not applicable to WSNs.

Li et al. [12] describe a rule-based publish-subscribe mechanism for real-time applications, however it is a soft real-time system and assumes computers with hard disks, which are typically not given in our scenario.

Stankovic, Son and Liebeherr propose “BeeHive” [13], a real-time, fault-tolerant, secure and quality-of-service aware “global virtual database”, for applications in internet services, defense applications, and to cache data from sensornets. It appears oversized for WSNs though.

Sharifi, Taleghan and Taherkordi present a “Publish-Subscribe Middleware for Real-Time Wireless Sensor Networks” [7], their work however is based on a semi-probabilistic approach proposed by Costa, Picco and Rossetto in 2005 [14]. This middleware reduces the average delay from generation of an event to its receipt, but without respect to any deadlines. Our proposal is deterministic.

V. CONCLUSION

We have shown that it is indeed possible to combine real-time constraints with a Publish/Subscribe paradigm, and leverage the advantages of both approaches. We have demonstrated that such a system can practically improve the timely behaviour of a system; where a non-real-time enabled system would be unpredictable, ours is predictable and meets all deadlines.

ACKNOWLEDGMENT

This work is partially financed by the European Commission under the Framework Programme 6 Information Society Technologies (IST) Project “Wirelessly Accessible Sensor Populations (WASP)”, contract no. IST-034963.

REFERENCES


### Table I
Maximum event latencies or max. tardiness [ms] between (S)ender, (T)ime-tracking (P)roxy and (R)ceiver, relative deadline 2 ms for “real-time enabled” mode. WSIM [6] simulation results. → * means that the message was relayed through the time-tracking proxy, rather than delivered directly.

<table>
<thead>
<tr>
<th>load threads</th>
<th>real-time disabled</th>
<th>real-time enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S → P</td>
<td>P → R</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

### Table II
Minimum, average/standard deviation, maximum latency, and maximum tardiness in ms for both events arrivals at subscriber. WSIM simulation.

<table>
<thead>
<tr>
<th>load threads</th>
<th>real-time event 1</th>
<th>real-time event 2</th>
<th>real-time event 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>latency</td>
<td>tardiness</td>
<td>latency</td>
</tr>
<tr>
<td></td>
<td>min. avg. sdev. max.</td>
<td>min. avg. sdev. max.</td>
<td>min. avg. sdev. max.</td>
</tr>
<tr>
<td>0</td>
<td>-19 -9.9 10.2 2</td>
<td>2 -10 -3.3 1.9 3</td>
<td>0 -15 -14.1 0.3 14</td>
</tr>
<tr>
<td>1</td>
<td>-18 -5.8 8.9 2</td>
<td>2 25 25.0 0.0 25</td>
<td>0 -15 -15.0 0.0 15</td>
</tr>
<tr>
<td>2</td>
<td>+9 19.6 2.1 20</td>
<td>20 45 45.0 0.0 45</td>
<td>0 -15 -14.9 0.3 14</td>
</tr>
<tr>
<td>0</td>
<td>-29 -29.0 0.0 -29</td>
<td>0 -15 -14.1 0.3 14</td>
<td>0 -15 -14.9 0.3 14</td>
</tr>
<tr>
<td>1</td>
<td>-29 -29.0 0.0 -29</td>
<td>0 -15 -15.0 0.0 15</td>
<td>0 -15 -14.9 0.3 14</td>
</tr>
<tr>
<td>2</td>
<td>-29 -29.0 0.0 -29</td>
<td>0 -15 -15.0 0.0 15</td>
<td>0 -15 -14.9 0.3 14</td>
</tr>
</tbody>
</table>

### Table III
Minimum, average/standard deviation, maximum latency, and maximum tardiness in ms for both events arrivals at subscriber. Test run on Telos.B.

<table>
<thead>
<tr>
<th>load threads</th>
<th>real-time event 1</th>
<th>real-time event 2</th>
<th>real-time event 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>latency</td>
<td>tardiness</td>
<td>latency</td>
</tr>
<tr>
<td></td>
<td>min. avg. sdev. max.</td>
<td>min. avg. sdev. max.</td>
<td>min. avg. sdev. max.</td>
</tr>
<tr>
<td>0</td>
<td>-20 -19.6 0.5 -19</td>
<td>0 2 2.1 0.4 4 4</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
<tr>
<td>1</td>
<td>-18 -3.5 0.8 0</td>
<td>25 25.0 0.0 25</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
<tr>
<td>2</td>
<td>20 20.0 0.0 20</td>
<td>45 45.0 0.0 45</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
<tr>
<td>0</td>
<td>-30 -29.3 0.5 -29</td>
<td>0 -15 -15.0 0.0 -15</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
<tr>
<td>1</td>
<td>-30 -29.0 0.2 -29</td>
<td>0 -15 -15.0 0.0 -15</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
<tr>
<td>2</td>
<td>-30 -30.0 0.0 -30</td>
<td>0 -15 -15.0 0.0 -15</td>
<td>0 -15 -15.0 0.0 -15</td>
</tr>
</tbody>
</table>


