Wireless Sensor Network, **2010, 2, 612-628**
doi:10.4236/wsn.2010.28073 Published Online August 2010 (http://www.SciRP.org/journal/wsn)

A Real-Time-Enabled, Blackboard-Based, Publish/Subscribe Architecture for Wireless Sensor Nodes

Björn Stelte
Faculty of Computer Science, Universität der Bundeswehr München, München, Germany
E-mail: *bjoern.stelte@unibw.de*
Received May 6, 2010; *revised May* 18, 2010; *accepted May* 25, 2010

Abstract

Wireless sensor network nodes have only limited resources concerning memory and battery life-time. Memory can be efficiently used by sharing data, and the life-time of a battery can be extended, when the node has long power saving sleep-phases. We propose a publish/subscribe architecture that achieves these two aims. The results of our work are of great interest for sensor application developers, giving them now the opportunity to use our architecture for sharing data among different applications on the node as well as the different layers of the operating system. We introduce a blackboard which is used for centrally storing published values, like measured data from a monitored sensor. This makes it possible to share stored data without monitoring the sensors once again, which is advantageously concerning power consumption, memory space, and reaction time. Beside the proposed publish/subscribe method for sensor nodes with its notification possibilities, our architecture fulfills also real-time requirements. We show how the well-known sensor operating system MANTIS OS can be extended by a real-time enabled, blackboard-based publish/subscribe architectture. This architecture and first of all its implementation is of special interest for cross layer optimization of sensor applications. Cross-layer approaches benefit from our architecture because the available implementation can be used as an efficient framework for central storing and managing of shared values.

Keywords: Blackboard, Real-Time, Publish/Subscribe Architecture, Wireless Sensor Network

1. Introduction

Today, sensor nodes are on the one hand small in size and economically cheap to produce, but on the other hand the applications on the nodes get more and more complex. So an application developer has to cope with limited resources, like memory space, power consumption, and CPU performance. Resource-sharing can moderate the resource limitations. In this paper the sharing of information between program parts is considered, not the communication between devices.

A resource-sharing system enables parts of the system to share information. A system that uses continuous queries for the resource-sharing is called publish/subscribe system. The continuous queries are the subscriptions and the submitter of the queries are the subscribers. If a new value is published by a publisher, this event triggers an inquest whether at least one of the queries is matching. So the system handles all publications and subscriptions as well as the notification process.

An good example for a publish/subscribe system can be found at the stock exchange. Because the price of a share is fluctuating, a profit is possible as long as a prospective customer buys the share when the price is low and sells it later when the price has risen. Assumed that the customer is willing to maximize his total profit, he will observe more than only one share at the same time. But the more shares the customer has in its portfolio, the more complex gets his situation.

At the stock exchange a publish/subscribe notification system is often used to moderate the complexity. Customers submit subscriptions with the name of the share and a threshold value for the price of the monitored share to a notification system at the market. This allows to have more than one subscriber for the same subscription. So instead of having several customers monitoring a share, the system does the job for them. If the given price is reached, then the system tries to immediately inform all relevant subscribers.

So at the stock exchange a publish/subscribe notification system is very useful. The system provides advantages for both the customer and the stock exchange. The

customer displaces the monitoring process to the system and saves time, because he gets a notification of his subscribed event as requested. The stock exchange benefits from the publish/subscribe system, because the number of customer queries is reduced dramatically. Secondly, the same continuous queries from different submitters can be summarised to one query for all interested subscribers. To sum up, resources are spared, if a publish/subscribe system is used for sharing stock information.

2. The Publish/Subscribe Paradigm

For a dynamic interaction between components, asynchronous communication models are used, like message passing, broadcast, and publish/subscribe. Eugster *et al.* Describe in [1] that only the publish/subscriber communication fully assists all three decoupling dimensions, which are the time, space and synchronization decoupling.

Karl and Willig explain in [2] as well as Köpke *et al.* in [3] that also a WSN application can benefit from the publish/ subscribe paradigm. So before analyzing the suitability of publish/subscribe for sensor node applications, we should have a deeper look at the publish/ subscribe basics first. A subscriber is not interested in all events a publish/subscribe service may support. So the subscriber specifies the events it is interested in and cuts the amount of all possible events towards its interest. There are three different types of subscriptions possible, namely.

1) Topic-based
2) Content-based
3) Type-based

At a topic-based publish/subscribe scheme, participants subscribe to a topic or subject, which is identified, e.g., by keywords. This is like joining a group, where pub- lished data for the group is forwarded to all group members. A mailing list is a good example for this type of publish/subscribe, where every topic is like an event ser- vice of its own. In a content-based publish/subscribe scheme, the decision if a subscriber gets a notification or not, is based on present keywords in the content of the event. A subscriber defines filters or constraints, which identify valid events. If a published value fulfills the subscription filter, the notification service informs the active subscriber. The other way round, if it is not valid the subscriber gets no notification message. The type- based publish/subscribe variant enables the subscriber to specify in which kind of values it is interested in. An event can be of different subtypes, e.g., a number could be of the type's integer or float. Type safeness may be an argument to use this type of publish/subscribe scheme. The content-based scheme is an extended form of the topic based one. At the topic-based publish/subscribe, every publication is forwarded to every subscriber of the event. The content-based scheme uses an addition filter, allowing to notify only all subscribers of the event with a matching subscription.

2.1. Publish/Subscribe Architecture for Sensor Nodes

Especially for distributed system publish/subscribe architectures are widely available. So if such an architecture is fitting for a sensor node, we can use the architecture for an implementation. But unfortunately, these classical publish/subscribe architectures are not suitable to sensor nodes, because they pay no attention to the before mentioned resource limits of sensor nodes.

As Köpke *et al.* describe in [3], a blackboard can be used for storing shared values at a central point. Within a data-centric control interface, setting and accessing data is possible. In their implementation for TinyOS, they use a publish/subscribe based control interface. The interesting part is how they solved the space problem which leads to the fact that concepts like the well-known Elvin concept is being unimplementable for WSNs. Köpke *et al.* use a blackboard for storing the data. In their implementation, each published information is defined by a unique number, a so called notification event number. A publisher writes its data on the blackboard and then informs the blackboard about the publication. Because of the unique event number, the publisher can tell the blackboard the corresponding event for the published value. The blackboard itself consists of three elements: the index table, the subscriber table and the subscriber flags. The tables are of constant size, which have to be calculated at compile-time. The index table consists of two entries for each event, namely the event offset and the event count. The offset value states the relative start position of the event at the subscriber table and subscriber flags. The count value gives the amount of subscribers for the event. So the index table has a size of twice of the number of events. The subscriber table stores the handlers for each subscriber. The position of the handler can be calculated from the offset value in the index table. An entry in the subscriber array has two flags for each subscriber. The first flag is the published flag and the second one is the subscribed flag. The published flag is set if an event is published for the subscriber. The subscriber flag is set only if the subscriber is currently subscribed to the event, otherwise the flag is clear. Subscriptions and unsubscriptions can be done at run-time, but the storage for the subscriber is reserved at compile time. The subscriber flags are stored in RAM, all other tables are transferred into the program structure at compile time. So the concept uses a minimal amount of RAM storage. This is effective and safes memory because of a central storage. But the concept has the disadvantage of being not flexible. The number of events and subscribers, and also the handler addresses have to be known

at compile time. A modification at run-time is not possible, because the storage is in ROM. So a subscriber can not declare it is interest in an event at runtime, this is done at compile time. The unsubscribe functionality is more like a pause function, where the subscriber declares its not willing to get more notifications at this point in time. The concept uses the publish/subscribe paradigm for decoupling in space and time. In Köpkes concept subscribers can only make a topic-based subscription for an event. On every event change they get notified if their subscription is active. The subscribers are notified by calling the function whose address is stored at the handler field.

2.2. Static Data Storage

Köpke's concept allows only topic-based and static subscriptions. Our idea is to extend the given concept to a content-based system, so that subscribers can set and modify a subscription filter, like type and limit, as well as the stored handler address. Instead of storing index and subscriber table in the structure, these now have to be stored in RAM too, in order to modify the values at run-time. Because a WSN node has only limited resources, especially RAM, the maximum number of events and subscribers is limited. At compile time, the needed storage size has to be calculated and reserved, since most operating systems for WSN nodes have no malloc() functionality available (MANTIS has non).

A developer should know the number of events which are published by a publisher thread, and subscribers for the events. Subscriber and events are in a relationship to one another. Each subscriber has to know the name or id of the event it wants to subscribe to. So the storage capacity required by the concept is calculated at compile time. A test should show at compile time, if enough additional space is available to handle the storage of the blackboard.

Köpke *et al.* use one published flag per subscriber in the subscriber flags array. In this concept, we use one publish flag for each event. There can be several distinct subscribers for one event, with different subscription filters. The subscriber flags array here is similar to Köpkes architecture, within the difference that one publish flag followed by n subscriber flags and one block flag are used for one event. The block flag is needed for safeguarding reasons. To calculate the bit position of the flags, we need no additional field in the index table. The offset value used to calculate the handler reference can also be used here for the subscriber flags. If a publisher sends a new value for an event to the blackboard, the published flag for this event is set and in the notification state entered later, only the events with an activated published flag have to be processed. In contrast to Köpke's concept, this publish/subscribe system is able to handle content-based notifications. On subscription, the subscriber commits the event identifier and the subscription filter. This filter consists of two fields, namely the subscription type and subscription limit. When a new value is published and the subscriber is subscripted for this event, the blackboard has to prove, if the subscription filter is fulfilled or not. On a positive decision, the subscriber gets a notification. As in Köpke's concept, this could be done by invoking a handler. The advantage of this concept over Köpke's is that it is more flexible. A subscriber is able to unsubscribe, if a subscription is not needed anymore, and a second subscriber can later subscribe and can reuse the reserved storage from the first subscriber. In order to reduce the amount of storage, the index table can be skipped if the number of subscribers is fixed, e.g. having fixed the number of subscriber for each event to 6, the offset in the subscriber flags is 8. So for each event, only 1 Byte has to be reserved for the flags per event statically. The problem here is fragmentation.

In the worst case, assuming at least one subscriber is available, space for 5 additional subscribers are unused, but the storage for them is still allocated. If more than 6 subscribers are interested in the event, a dummy subscriber can be created, which publishes to new event on notification of this dummy subscriber. Additionally subscribers can subscribe to this event, instead of the fully subscribed root event.

2.3. Dynamic Data Storage

In a static storage, storage reservation is necessary at compile time. Space for events and subscribers is calculated and each gets its own part of storage space. Another concept is to reserve a given size of storage for the notification system at compile time and reserve space for events dynamically at run time. On creation of an event, a data container is created, which holds all relevant data of the event. The published values as well as the subscriber filters are outsourced and only accessible by a pointer. Therefore, the event container holds the pointer addresses to these data. Beside the pointers, information like the number of subscribers and the number of publishable values is also stored in the container. The number values are needed to calculate the last position of either the published value array or the subscriber arrays for subscriber limits and types. The subscriber flags are stored in the event container. Unlike the static variant, the subscriber flags are not limited and can have different sizes for each event. The number of subscribers can be found in the container, so the size of the subscriber flags in bits is two plus number of subscribers. Each event container has a pointer to the next container, so each container is reachable through its predecessor. A direct access on a container could be done by using a index table, which holds all next pointers.

Storing pointers to the values instead of the values in

the container has the advantage that sharing is possible. In a situation where two different events have the same subscribers, the pointers for subscriber limits and types are the same. The individual subscribers of the two events share the subscriber array and so reduce the amount of needed space. A modification of the filters will influence both events at the same time, this means that cycle counts are reduced for modifying the filters. Secondly, not only the last published value can be stored, now several values can been stored externally with the pointer to the first value in the container. Each event container can handle a different number of published values. The containers are linked by a next pointer, so a list of containers arises. This list can represent a priority order, where a container in the front has a higher priority as its successor.

The priority order can be changed at run time by swapping the next pointers. At the notification phase, the blackboard starts with highest priority containers. As in the static concept, only event container, whom published flag in the subscriber flags array is set, need further processing. If at least one of the subscriber flags is set as well and the corresponding subscriber filter is true, the blackboard marks the subscriber to be notifiable. After all containers are processed, the notify list with all marked subscribers can be processed. A method like first-come-first-serve should be adequate here, with the subscribers of higher prioritized events first. Reordering should be used for containers which are used often. Reordering can be done at run time, depending on a situation or event. The event container is easily expandable for the developer. Additional values can be stored in the structure by adding new variables in the event container. The main problem with the dynamic storage concept is the needed storage capacity and the needed computational time for modifycations. Specially on publishing a value for an event, we need a index table to get the current pointer address to the container for the event quickly. The index table is only needed for publishing. Without the table, we would have to search for the correct container by hopping from one container to the next, until the correct container is found. The index table, holding the id of the event at its pointer to the container, has also the problem of being static. Since sensor operating systems like MANTIS OS do not support malloc() functionality, the storage for the index table has to be reserved at compile time. So even when the dynamic storage concept is used, the amount of accepted events is limited at compile time.

2.4. Combination of Static and Dynamic Storage

Since neither the static nor the dynamic concept (explained in the sections before) seems to really suit to WSN nodes, a combination of static and dynamical should be discussed. The idea is to reduce used space while keeping flexibility.

The static concept limits the subscriber to define better filter constraints for their subscription. A subscriber can only post one filter for each event. It should be possible for the subscriber to define several rules, which have to be valid for notification. In Elvin [4], a special subscription language was used for defining subscription filters. Defining a special subscription language is expensive regarding storage and computational time. Another idea is the creation of special subscriber type, used for dummy subscriptions. When the first subscription filter is valid, a sub-event is notified by a dummy subscriber with the second filter rule. The second stage subscription rule is validated after the notification and the real subscriber can be informed if this second subscription filter is also valid. This idea leads to a tree-like subscription filter net. So a nesting of subscriptions covers the same complexity as a subscription language, like the one in Elvin.

At the dynamic concept, an index table has to be used for finding the requested event container or the list of container has to be gone through to find the correct container. In the static variant, no index table is needed, because the number of subscribers are fixed. The idea now is to get a combination of the two concepts, but the question is how to do this? First, an observation may help. Subscriptions are for an event and the event is well known to the developer. The application developer uses a subscriber for an event with a fixed identifier. So the developer knows the total number of subscribers and events in the system. As a result, the developer is able to reserve as much space as necessary for the notification system at compile time.

Because the dynamic concept uses storage space wastefully, a more static concept is needed. **Figure 1** shows the combination of static and dynamic storage for one event. A management frame is used instead of a container. This frame consists of six parts, which have a total size of 7 Bytes. The first part contains the well known subscriber flags. The number of subscribers is limited in this example to six, so the first management frame part has exactly a size of 8bit. This is profitable, because having a 8 bit CPU, flags can be set and cleared rapidly without being vigilant about byte bounds. The second part of the management frame consists of eight 1bit flags for describing type and status of the event. We will discuss the meaning of this field later in this section. The remaining parts of the management frame are the four pointers to the published data, the subscriber limit, type, and handler. A pointer itself has normally a size of 32 bit, but with the observation in mind that every data or subscriber value has a size of 32 bit, only storing the position instead of the real pointer address is enough. In the management frame, each of the positions, which can be stored, has a size of 10 bits. So for all four positions, we need 40bits, and with adding the 16bits from part one and two, we have 56 bits, which are 7 Bytes, in total for a management frame. As in the static variant, an index table

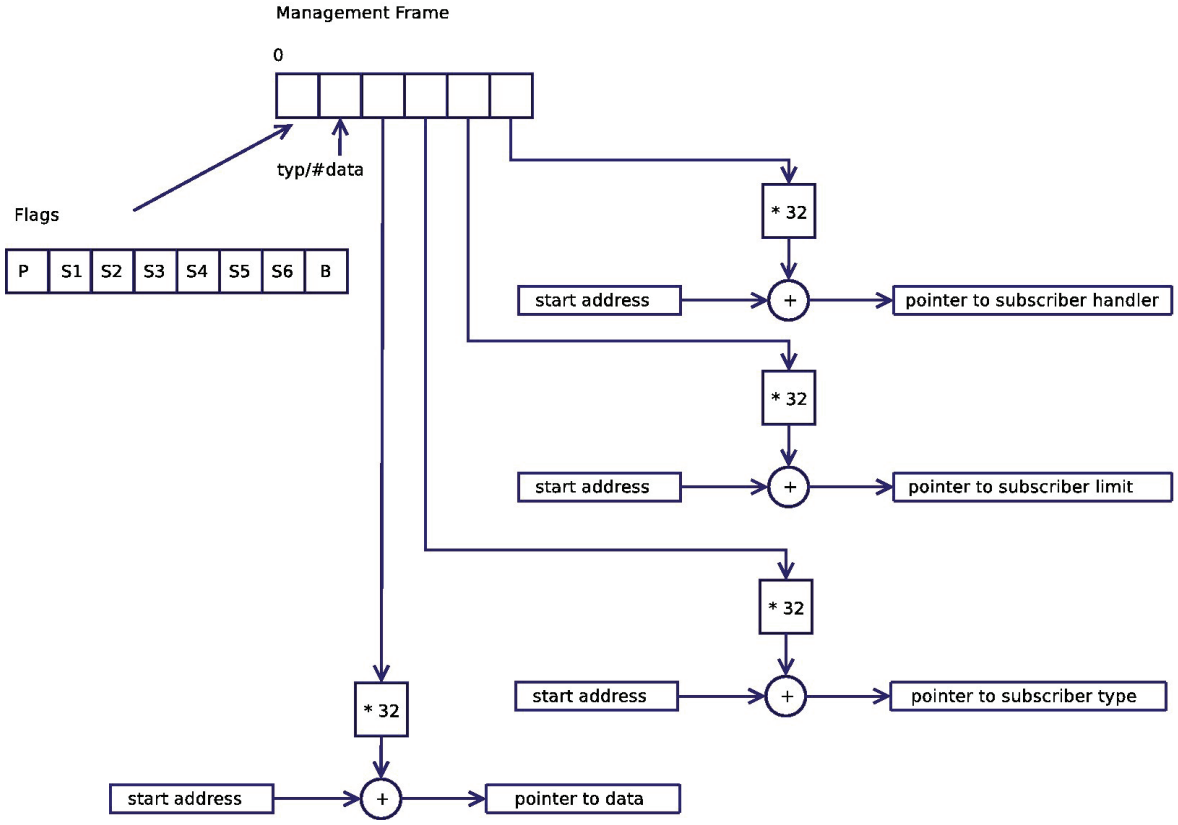

Figure 1. Effective data storage.

is not needed, because each management frame has the same size. So calculating the start position of event x is easy.

We have reserved 10 bits for storing the position, instead of the pointer. The question is know how to use this position for calculating the real pointer and is the size of 10 bits huge enough? First we will show that 10bits are sufficient by taking into consideration that data is stored on a sensor node. Secondly the pointer calculation with position value as an offset value is shown.

With 10 bits representing the position of a value, there are 210 possibilities. So 10 bits are sufficient to number at least 1024 values. Each value has a size of 32 bits and so in total 32 bits × 1024 = 32768 bits or 4096 Bytes are addressable.

A sensor node like the Mica2dot has only 4 kByte RAM, the considered 10bits are adequate for addressing 32bit values. With a start address and the 10bits for the position, the pointer is calculateable. The position multiplied with 32 gives the offset, which has to be added to the start position. The result is the searched position address of the value.

When the total size of RAM would be reserved for the notification system, realistically at most 35 events can be created with 35 × 6 = 210 possible subscribers. When all of them store an individual subscription filter and all events store one published value, than 35 × 7 Bytes + 35 × 96 Bytes = 3605 Bytes are used. Not surprisingly, the amount of needed space can be cut, if subscription sharing is used, like in the dynamic variant. Additionally, unlike in the dynamic variant, the subscription blocks for all 6 subscribers can be partly shared. As mentioned, no index table is needed, because of a static size of the management frame. Therefore, the number of possible subscriptions to one event is fixed to a number like six. If there are more than six subscribers for an event, the idea is to create a new event and make one special subscription at the first event, which is always true at examination. The new event is called a sub-event of the first one. There may be at most six sub-events for each event and also each sub-event could have its own sub-events. So the number of subscribers for an event is only limited by the available storage space and not by the architecture. The special subscription used here needs a significant value in the subscription type and the position of the sub-event in the subscription limit field. As seen in the dynamic concept, a priority based scheme is possible. The subscribers in the root event have a higher priority than its sub-events. Next to prioritization, it is possible to create a sequence of subscriptions, which represents multiple equations of a subscription.

For example, a subscriber who would like to get a no-

tification, provided that the published value is larger than a limit x and smaller than a limit y, defines two subscription filters. The first filter, from the type forwarding and greater with the subscriber limit value of x, only notifies a sub-event if the greater equation is valid. The subscriber handler field of this first subscriber filter holds the address of the sub-event. The second subscriber filter is the filter of a subscriber of the sub-event. This filter is a normal subscription filter with a smaller type of subscription equation and the limit y. If the first equation is valid, the sub-event is activated by a notification. If also the second subscription equation is valid, the real subscriber gets a notification. The positive effect is, that the published value is stored only once.

Both management frames have the same position stored for the published value. It is also possible to extend this strategy for n equations, but than reducing the offcuts is more and more important. When sub-events are created, a tree of events is built. In this scenario, it is possible to use the quenching strategy from Elvin. If no subscriber of a sub-event is active, all subscriber flags are cleared, than the subscriber flag for the sub-event should also be cleared in the subscriber flags array of the root event. This has the consequence that the blackboard cuts the sub-event branch of the event tree. Quenching for sub-trees reduces computational time and should be used as often as possible.

As mentioned before two bits in the management frame are reserved for the event type. So far there is no sensible necessity for using the event type. A later possible usage may be the decision if one of the bits should be used as a storage flag. This flag than indicates if the blackboard information is stored in RAM or in the ROM part. A set storage flag lets the blackboard use for calculating the correct storage pointers the start address of the RAM part as a cleared flag means the start address of the ROM part has to be taken. If more than one publication should be stored, the difference between the number of the publish and the subscriber type pointer has to be greater than one. Than there is enough space reserved for storing more than one published value. For simplification the front value, which is in the position of the pointer address should always been the latest value. So on each publication all published value has to be pushed 32 bits to the right within their space reservation. Than the first entry is overwritten by the new published value.

2.5. Evaluation

In this section, we will compare the memory usage of the three discussed storage concepts. Since a WSN node has only limited amount of available memory, one goal is to use as less as possible memory for storing event and subscriber values plus minimizing the management overhead. On the other hand, the storage concept should be flexible. The flexibility of the dynamic concepts need more memory than the static one, but the question is how big this gap is and how is it justifiable? For a comparison, we need the memory usage of all three concepts. First we will calculate the number of bits the static concept needs for storing one event. With the values of **Table 1**, definition 1 shows the amount of needed bits for one event with n subscribers and one published value.

DEFINITION 1. bitsstatic = 97 bits * #subscribers + 2 bits + 32 bits * #value

For the dynamic concept, **Table 2** shows the additional needed space the dynamical concept needs comparing to the static one. This additional amount of memory is statically, so for each event the dynamic concept needs at least 160 bits more memory than the static concept. Definition 2 shows the total amount of memory in bits for one event with n subscribers and i published values.

DEFINITION 2. bitsdynamic = 97 bits * #subscribers + 162 bits + 32 bits * #values

The static-dynamically mixed concept needs 56 bits for the management frame and 32 bits per published value. For each event six subscribers are possible and space is reserved for this six subscribers, which is 6 * 96bits per event. When more than six subscribers wants to subscribe, additional events, so called sub-events are created. Each of them has its own management frame and the additional space for the subscribers. That published values are referenced by the value pointer, which is for all subevents the same. Definition 3 shows the needed space mathematically.

DEFINITION 3. bits concept = 32 bits * #values + (56 bits + (6 * 96 bits)) * (ceil(#subscriber/6)))

On a node, like on any other computer board, the available memory for the notification system is fixed. In the following figures a capacity of 4 kByte is assumed. As **Figure 2** shows, the extreme points, where a huge amount

Table 1. Storage for the static event handling.

published value	32 bit
status flags	2 bit + n bit * subscribers
subscriber handlers	32 bit * subscribers
subscriber limits	32 bit * subscribers
subscriber types	32 bit * subscribers

Table 2. Additional Storage for the dynamic event handling.

pointers	96 bits
numbers	16 bit
event type	16 bit
next pointer	32 bit

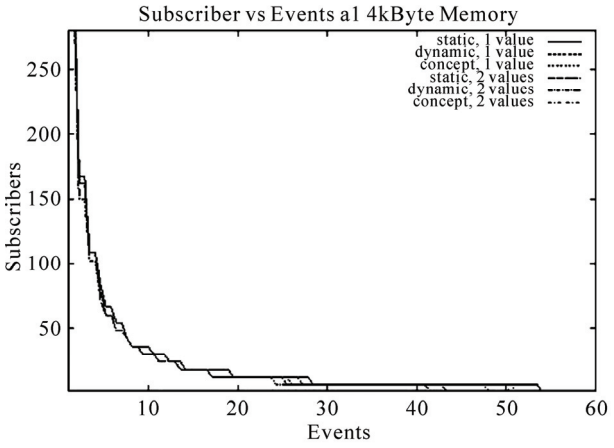

Figure 2. Number of possible events vs subscribers for 4 kByte.

of subscriber or events are capable, are unusable. In this example a range of 3 to 20 events seems to be a good working area. On the one hand a huge amount of subscribers but a less number of events are supported and on the other hand a huge amount of events but a less number of subscribers are capable. It is a trade-off between this two factors. A developer has to decide, for which of them he wants to optimize his system. When a subscription on many events should be possible, like up to 20 events in the example of 4 kByte of memory, the question is how many subscribers are supported in average. Assuming a uniform distribution of the subscribers on the events, all events have the same number of subscribers, all events get the same quota of memory allocated. All calculations in this section are for events with six subscribers. In this scenario the static concept dominates its competitor concepts, because its overhead per event is small. The dynamic concept needs only a static additional amount of 160 bits per event, assuming that all stored values, the published value and the subscription values, can be stored compact. This assumption leads to an efficient storage but additional computational time for shifting the values on adding and removing new subscribers or events. The presented architecture does not need this additional computational time but pays its dynamically with additional needed memory

3. Real-Time-Enabled Publish/Subscribe

A value is published on the blackboard when the publish function is executed within a thread. The function needs as parameters the event id, which is unique, and the new value, which should be published. When the blackboard thread is activated later, the blackboard works off the publishing procedure. First it tests if the event is blocked by another process. If the blocked flag in the subscriber flags array is set, the publisher has to wait for unblocking or skip the publishing process for this event. When the block flag is not set, the blackboard set the block flag and the publish flag. Now no other process can disturb the publishing process for this event. The publish flag is set and so leads to later verification of the subscription. If all subscriber flag are cleared the published flag should also be cleared, because when there is no subscription, no subscriber can get a notification. Also it could be possible to inform the producer about this situation. This process is called quenching and instructs the producer to skip further publishments for the event. When at least one subscription is made for the event, the producer has to be informed once again, now to stop the skipping.

In our concept, not the complete management frame has to be modified, only the first byte needs a modification. The first byte consist of the subscriber flags, which should be notified as described. In order to store the published value in the right position, the publish block of the management frame is read and the value is stored at the real address. After finalization of the publishing process the block flag in the subscriber flags array is cleared. A consumer thread calls the subscribe function of the blackboard API for creating a subscription for an event. The event id and the subscription filter values are given to the function by parameters. The subscriber gets a unique subscription number, which is the next free number of the all subscriber flags in the event, back as a result of the function call. Which this number a later pausing or unsubscribing can be done. Like in the publishing process, the block flag is tested and set. If no other subscriber flag is set before and quenching is used, the producer has to be informed, that there is at least one subscriber, who is interested in its published value. The first two bytes of the management frame have to be modified.

In the first byte, the subscriber flags array, the corresponding subscriber flags has to be set. In the second byte, the type field, also the corresponding flag is set, standing for a reservation of the subscription block for the subscriber. Since a structure optimization for reducing offcuts is not done so far, the data flags block should be a copy of the subscriber flags. The needed next free subscriber number, is the first position of a subscriber flag, which is cleared and which representing flag in the data block is not set. If no space is available in the management frame for an additional subscriber, a new event is created, called sub-event. One of the subscribers is copied to the new event, meaning coping subscription filter values and setting the subscriber and data flags in the new sub-event. In the gap of the old subscriber in the root event, a new subscriber is created with a subscription filter of a special forwarding type. This subscription is always true and on notification processing, the new sub-event gets activated by notification. Because the root event holds the position to the published value, the position can be copied to the position stored in the sub-event.

The subscription filter values are subscription type, limit and handler. These values are stored at the position, which stand in the three position blocks of the manage-

ment frame. The first subscriber takes the first 32bit beginning at the given position, the second subscriber the next 32bits and so on. This procedure is done for all three values. An unsubscription has to be divided in a pausing and a real unsubscription request. Pausing means that the subscriber wants to skip notifications for a short time and wants to resume later. Unsubscribing means that the subscriber gives its subscription up and that the reserved memory space can be used be another subscriber. The subscriber executes the unsubscribe routine with the parameter of the event id, the subscriber number and if its willing to pause or not.

Like in the other processes, the block flag is tested and set first. If the corresponding subscriber flag is set, the flag will be cleared. If not pausing, but unsubscribing is chosen, also the corresponding data flag is cleared. After all, the block flag is cleared and the unsubscribe process has finished. If a new value is published and a subscription inequality of at least one subscriber is fulfilled, the subscriber needs a notification. But how to notify the subscriber? Before possible solutions can been discussed, the goals should be clear. A notification method should be

1) effective concerning space and time
2) be implementable (Mantis prototype)
3) fulfill the three publish/subscribe constraints

Subscribers can be splitted into two parties. Subscribers, which are interested only in the notification itself, not in the published value, are in the first party. These subscribers are namely the alert subscribers. The second party of subscribers need the value for their further proceeding, like sending a message with the published value to another node. These subscribers react on a notification like the alert ones, but also need the value at notification. Each of these subscribers could be build as an alert subscriber, which requests the value at notification or the value is passed to the subscriber with the notification.

Subscribers who are only interested in getting a notification as an alert have three choices in getting a notifycation. First, the subscribers could poll for a notification. The blackboard needs a notification queue, in which all subscriber ids are stored which have to be to notified. On a poll request, the blackboard will give a true back as response, when there is an entry for the subscriber in the queue and false otherwise.

After every valid polling request, the found subscriber entry is removed form the queue. The problem with the polling method is, that when its used too often, too much computational time is wasted, having an active consumer thread. So the thread with the waiting and polling subscriber disables the operating system to switch to lower electric consumption state. The time the subscriber waits for the notification may be used to let the micro-controller sleep, but with an active polling process this not possible. In a situation, where the producer of the event is not in the same thread of the subscriber, it is possible to reduce the polling to a minimum.

On a reactivation of the thread, only one polling is allowed, because a second request would always return false. But even in this situation, the polling method inhibits an efficient power management. The only positive fact is, that the subscriber defines the point in time, when the notification should be processed.

A second idea is the usage of a function call. In some operating systems a alert function is available, which starts a function, when a given point in time is reached. For execution of the function, a reference to the function is given to the alert process on creation of the alert. The same could be done here. The function would be executed in the context and time of the blackboard. So the subscription thread is definitely not blocked and does not wait or pull for getting a notification. The problem is, that the function may use some variables from the subscribing thread. This is not allowed, better not possible, when the function is executed at the blackboard thread. Only global variables are reachable for the function in order to publish the computed new value. Another idea may be, that the function uses the publish function to make the value known. But then the whole process chain has to be built as nested functions. This complicates the process unnecessary in context of debugging and implementation interested.

The second point is that this function call method breaks the multi-thread concept. The process chain would be computed at the blackboard thread. The main scheduler would be useless in this situation. So a new additional, inline scheduler must be implemented in the blackboard. This scheduler schedules all function calls for the blackboard. As we can see, the function call method is tricky and so arduously to proper implement. A third idea, how to implement a notification, is the idea of a locked subscriber block in the subscribing thread. At the first sight, blocking the subscriber is not wished, concerning the decoupling rules. But here, not the whole subscriber is blocked, only a part of it, the part which handles the things to do when a notification arrives, is locked. Instead of out-sourcing the code in a function, which has to be executed on notification, here the code is still in the thread, but locked with a global available semaphore, which the subscriber has defined. The previous discussed notification model with function calls can been transformed in this model. The code of the notification functions for the subscribers build subscription blocks in a special subscription thread. Each thread is locked by a semaphore, which is unlocked if the blackboard thread calls the notification for the thread. Instead of executing functions at notification, the semaphore is unlocked and the subscription thread is activated once.

The only thing the blackboard has to do, is to unlock the semaphore. In the subscriber block of the thread, it has than to be decided, if the semaphore should be relocked again or not. The positive effect on this method is that first the subscriber decides the point in time of the notification

in the thread and secondly thread variables are still available. The problem is, that there could be a problem with the scheduler, when the subscribing thread is not activated contemporary. When the blackboard clears a semaphore for a subscriber, the subscriber could execute the now unlocked code. In a situation, where a producer publishes a new value for this event before the subscriber thread was activated, the blackboard once again tries to notify the subscriber by unlocking the already unlocked semaphore and the subscriber only gets the notification for the last published value. So when a notification guarantee should be given, than the scheduler has to prior the thread with an outstanding notification. The needed modification for the scheduler should be minimal. An additional queue is needed, where the blackboard writes all subscriber ids to notify down. The scheduler works off this list and gives the corresponding subscriber thread the highest priority. In the next step, the subscriber id is removed form the list, so that on the next scheduler round, the thread get its old priority back. The problem here, is that the scheduler must know, which subscriber id belongs to which thread id. The blackboard can store this information at subscription. Each thread knows its id, so this id could be given as a parameter to the blackboard on subscription and stored in a extra table. Either the scheduler reads this table to decide which thread to prefer or the blackboard does this and gives the scheduler the thread id instead of the scheduler id. The interference in the scheduler process has effects on the thread order. When many notifications should be worked off, it could be that some threads will become stunted, because all notified threads have a higher priority. Aging is a solution to avoid this scenario. A second problem is the question how in which order the high priority threads should be activated. A normal first-come-first-serve scheduling strategy should do that work in a normal scenario, but in a real-time enabled notification system the whole scheduler must been modified.

4. Concept of a Blackboard for WSN

So far, we have shown how to store events and subscriptions on the blackboard and how producer and consumer can use the blackboard API. Now we will discuss the blackboard process, which is settled in an own thread, the blackboard thread. The assignment of the blackboard thread is to process all publish and subscribe requests as well as verifying the subscriptions and start the notification process if necessary. The blackboard thread can join five different states. These states are

1) sleep state
2) modify state
3) prove state
4) notify state
5) optimize state

The sleep state is always the first and last state in the blackboard process. First, being in this state, it is proved if at least one producer has published a value. This can be done, by looking in the publisher queue. If this list has at least one entry, the blackboard thread switches to the modify state, else the thread is let sleep and the scheduler can switch to another waiting thread. The publishing queue is needed, because the publish/subscribe paradigm demands a decoupling in time. When a publisher would access the storage management directly, than one part of the blackboard assignment would be done by the producer.

But than the producer would be blocked, while it writes its data to the blackboard storage. To prevent this, the producer only pushes its data to the blackboard, which queues this in the publisher queue. Not surprisingly, the same is done for the subscriber so the queue is extended to also store subscription requests. On activation of the blackboard thread, the queues are worked off by switching to the modify state. In the modify state, each entry in the queue is worked off in a first-come-first-serve manner. This is done because of the fact that a subscriber should only get a notification for an event, which is published after the subscription. For the unsubscribe request its vice versa, if a consumer unsubscribes no further notification should arrive after the unsubscription. If all subscriptions in the queue would have be done before publishing in the blackboard, than the resulting process order may be incorrect. After the processing of each queue entry, the item can be purged from the queue. When the queue is empty and at least one publish entry was processed, than the thread switches to the prove state. Else the blackboard has finished and switches to the sleep state.

In the prove state, the blackboard process goes over all management frames, in order to find an activated publish flag in the subscriber flags array of the frame. If one is found, it is proved if one or more subscriptions are fulfilled. For each of them an entry with the subscriber id and the subscription handler address is stored in the notification queue. Next, the publish flag is cleared and the next management frames are tested. When all frames are worked off and at least one notification has to be done, the blackboard thread switches to the notify state. Else, no notification has to be done and so the sleep state is activated.

In the notify state, all notification assignments, which are stored in notification queue, have to be worked off in a first-come-first-serve order. Therefore, all notifications semaphores, which reference is the subscription handler address, are cleared. Following, the value of the thread id, which belongs to the subscriber, is pushed to the scheduler.

As described in the section before, the scheduler decides, when a thread is activated, When the queue is processed, the blackboard finishes its process by switching to the sleep state.

Optional, a optimize state can be insert after the publish state. The assignment of the optimize state is to reduce offcuts in the blackboard storage. This state increases the needed computational time of the blackboard, so it should only be used, if the fixed available memory for the blackboard is nearly reached.

5. Integration of Publish/Subscribe into MANTIS OS

The interaction style describes how parties communicate which each other. As Rozanski and Woods describe in Chapter 11 of [6] the interaction style for a blackboard depends on referential and non-temporal coupling. The referential coupling means that the sender has a reference to the receiver's address. Concerning the blackboard the sender is the publisher which publishes an event. The publisher knows the event identifier, so it has an indirect reference to all subscribers of the event by the event itself. An asynchronous data exchange between sender and receiver is a non-temporal coupling. Exactly this decoupling in time is one of the three main conditions explained in Section 2. The point in time when the receiver gets the notification of the publication is not predictable. It could happen at once after the publication or some time later. Therefore, setting a deadline is conceivable in order to isolate the notification point in time. This directly leads to the idea of an real-time enabled publish/subscribe system. The presence of deadlines requires the modification of the operating system scheduler. The scheduling of threads, which contain deadline afflicted subscribers, leads to the question if all deadlines could be met. First and foremost, the used real-time scheduler algorithm determines, if all deadline expections will be fulfilled. For simplifying the choice for the correct scheduler algorithm, we will make two important assumptions in this paper. We will assume that the system is always schedule-able and that the system is never in an overloaded condition. This assumptions can be made, because the task is to create a real-time enabled publish/subscribe system and not a real-time enabled OS.

This makes things different, because in this scenario a central component, the blackboard, decides the point in time when the deadline of a thread should be modified and how the value of the deadline should be modified.

5.1. Overview MANTIS OS

MANTIS OS [1] is built in the classical multi-thread design. The application threads are separated from the underlying operating system, which is split into two layers. The upper layer consists of the system API. The threads are using the system API, when a radio communication is started or a sensor is triggered. The system API is based on the lower layer which consists of the kernel, the communication (COMM) and the devices (DEV) part. The COMM part handles asynchronous I/O, like radio, serial, etc. and the DEV part all synchronous I/O e.g. reading data from the sensor. The COMM part is interrupt-driven and so no polling is used. For example a received data packet from the antenna triggers a hardware interrupt, which pushes a transfer function for storing the packet into a receive queue. The main part of the base layer is the part of the operating system kernel. Besides the existence of mutex, semaphore and alarm functionality, the scheduler is the main part of the kernel.

The scheduler decides which of the available and ready threads to activate. Threads can call a sleep function to sleep for a defined time. Then the system API gives the scheduler the opportunity to determine a new schedule omitting the sleeping threads. If no thread is ready for activation, a idle thread is activated, which lets the micro-controller sleep for the rest of the time-slice.

The number of threads in MANTIS OS is limited to at most 8. This was a design decision with the total available memory in mind. The problem is that each thread gets a reserved memory block of its own. These blocks are reserved statically with the best-fit method. So a dynamical allocation inside the thread is not possible. MANTIS OS is built in the classical multi-thread design. The application threads are separated from the underlying operating system, which is split into two layers. The upper layer consists of the system API. The threads are using the system API, when a radio communication is started or a sensor is triggered. The system API is based on the lower layer which consists of the kernel, the COMM and the DEV part. The COMM part handles asynchronous I/O, like radio, serial, etc. and the DEV one all synchronous I/O e.g. reading data from the sensor. The COMM part is interrupt driven and so no polling is used. For example a received data packet from the antenna triggers a hardware interrupt, which pushes a transfer function for storing the packet into a receive queue.

The main part of the base layer is the part of the operating system kernel. Besides the existence of mutex, semaphore and alarm functionality, the scheduler is the main part of the kernel.

The scheduler decides which of the available and ready threads to activate. Threads can call a sleep function to sleep for a defined time. Then the system API gives the scheduler the opportunity to determine a new schedule omitting the sleeping threads. If no thread is ready for activation, a idle thread is activated, which lets the micro-controller sleep for the rest of the time-slice.

5.2. Scheduler

The scheduler is preemptive designed, allowing MANTIS OS to switch active threads even if they have not finished their execution. The scheduler is built as a priority-based

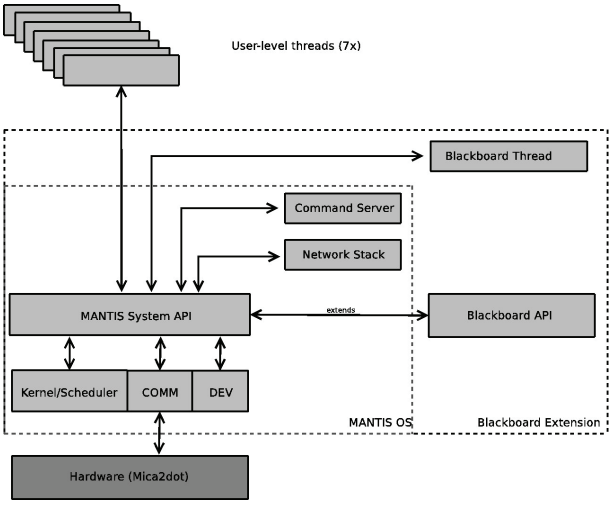

Figure 3. Blackboard extension of MANTIS OS.

Round-Robin scheduler with five defined priority levels. The time slices are 20 msec long and controlled by a hardwarebased timer interrupt. At the end of a time slice or if a thread calls an interruption command (like mos_thread_sleep()), the running thread is halted and appended to the end of the ready queue. The dispatcher function of the scheduler grabs the first thread from the ready queue and executes a context switch. The ready queue itself is split into five separated queues. Each priority level has its own queue, so in total we have five queues for the ready queue. The implemented Round-Robin scheduler first looks at the existence of a head element in the highest priority queue, then in the next lower prioritized queues. The queues are build as a chained list with a head and a tail element. The cost for the search of a defined element in the list has a complexity of O(n). Beginning the search at the head of the list, at most all n elements have to be visited once. The round-robin scheduler always takes the first element, the head of the list, which it found in O(1). The extended scheduler needs to purge the ready queue from the thread the EDF scheduler has chosen for activation. So if the EDF part is active, the dispatcher function has to find the thread in the ready queue and to overwrite the predecessor threads next pointer with the next pointer of the chosen thread in the list.

Fortunately, MANTIS OS already has such a function, which is called mos tlist get thread(). Unfortunately, this function has a bug not finding the correct thread nor any thread in the list. The problem here was a incorrectly defined if clause, not correctly testing the thread id against the search id. We have bug-fixed this function and use it in the EDF part of the dispatcher.

As seen in the section before, the implemented combined scheduler always tries to find a EDF schedule-able thread first and only falls back to the round-robin method if none is found. To find at least one thread with a deadline, all entries for the threads have to be visited once. The MANTIS OS scheduler uses a thread table for storing all thread relevant information. Here, we have extend the thread table by the deadline variable. To set the deadline for a variable, I have written a new function called set deadline (thread, deadline). This function can be used to assign a given deadline to a certain thread. The scheduler will go through the all thread table entries beginning at the first and memorizes the thread that is in the ready status and has the lowest deadline of all ready threads. This search has a complexity of O(n), with n as the number of all threads.

If no thread is found, the round-robin part of the dispatcher determines a thread to activate. Going through the different priority queues of the ready queue, this has a complexity of O(k) with k as the number of queues.

If a thread is found, the EDF part has to remove the thread from the ready queue. In contrast to the round-robin part, the priority of the chosen thread is known, so the search function mos tlist get thread() is used here. This function needs at least one iteration to find the correct thread in the queue and in worst case n iterations, which is the number of elements in the queue. Because only 10 threads are possible in MANTIS OS, the time needed for searching and removing the element of the list is short. My extension of the dispatcher function has the same complexity class of O(n) as the original scheduler of MANTIS OS.

As well as the real-time ability, activating threads only when they are needed is also a design goal of the scheduler. To do so, a new thread state, called blackboard state, was created.

A thread in this state cannot be activated by the Round-Robin scheduler, because the thread is not in the needed ready state and so it's not listed in the ready queue. The EDF scheduler part is able to schedule this kind of thread. It handles the blackboard state thread like a ready thread. So if the thread is in the blackboard state and has a deadline lower than the maximum value for a deadline, it is schedule-able by the EDF scheduler part. The advantage over the normal scheduling is that the thread is only activated if it has a deadline. Because only the blackboard thread sets the deadline for a thread with at least one active and valid subscription, a thread in blackboard state will only be woken up when it is required.

The wake-up process is similar to the sleep method. The difference between the sleeping and the blackboard state is that a sleeping thread cannot been awoken when it sleeps and that the sleeping thread sleeps for a defined time. After the sleep time, the thread has to compete for a time slice, processes its commands and sleeps again. A thread in blackboard state is only activated if the correct published value is available and is suspended again after its processing. This new method allows longer microprocessor sleep and suspend times, because the idle thread will be activated more often as when only sleeping thread

would be used.

In order to set the blackboard state for a thread, we have implemented a new function, called mos give up(). This function changes the current thread state to the blackboard state by overwriting the state parameter of the thread entry in the thread table. The Round-Robin scheduler does not know this new state type and so it does not interfere. The blackboard thread uses this blackboard state. If for example a value is published, the blackboard thread gets a very low deadline and is activated after the another thread publishes a value.

When the blackboard thread has finished the processing, it sets the deadline to infinity and the state type back from running to the blackboard value. So the blackboard thread is only active when it is required. No unneeded processing time is wasted, because the thread is not activated when it has nothing useful to do.

5.3. Blackboard API

The publish and subscribe process can be done at once by directly executing the corresponding function or by using a buffer which the blackboard processes afterward. By buffering all publish/subscribe commands the sender of these commands has not to hand over process time for the publish or subscribe process. When the blackboard thread is the next activated thread after the buffer is filled, we can guarantee that the all buffer entries are processed directly after the thread switching. The blackboard thread processes all buffer entries the notification process has started. We have implemented a queue which is filled with blackboard commands that the API makes available to the application threads. On filling the buffer or executing the publish command, the deadline of the blackboard thread is modified.

The value of the deadline is set to the lowest possible value for the deadline (the number one). This will lead to an activation of the blackboard thread instead of any other one on the next scheduler run. The blackboard is preferred against the other threads. So when a thread executes a buffering command or pushes at least one new publication, the blackboard thread is always executed directly after the publishing thread.

Each entry in the buffer consists of several components. The most important one is the identifier, which has the following values for the different blackboard commands.

1) subscribe
2) modify subscription
3) unsubscribe
4) pause subscription
5) publish

Other stored components are always the number of the event for that the command is and the current time-stamp. If a subscription command is called, the buffer has also to store the subscription parameters subscriber id, type, value, deadline, and the address of the semaphore of the notification process. The buffered communication between the application threads and the blackboard thread is asynchronous. All commands are buffered first and processed later. The problem is that it could happen that an unsubscription is temporally before a publication. So this unsubscribe command has to be processed before the publish process to avoid side-effects.

The time when the command is stored in the buffer is important. All buffered commands have to be processed in the order of their entering in the buffer. So the implementation of filling and removing buffer elements is used in the first-in first-out manner.

Besides using the buffered commands subscribe_ request() and publish_request(), the direct commands subscribe() and publish()—without using the buffer—are also possible. The difference is the point in time when the command is processed. The direct commands are executed at calling and use up processing time of the current thread. On the other hand, the buffered commands are stored temporary and executed when the blackboard thread is activated. The processing of the calling thread is not inhibited, because the command execution is after the thread was suspended from the scheduler. The second difference can be found at the error handling. Because the buffered commands communicate asynchronously with the blackboard, the blackboard ignores errors. The calling thread does not wait on the command execution, so no value is returned as status information. It is different for the direct command procedure. Because the thread executes the command and waits on a result afterward, an error return is possible.

5.4. Blackboard Thread

As described before the blackboard thread is only awaken if at least one element is in the blackboard buffer. The blackboard thread will process the buffer in the modify state beginning with the head element of the buffer. Afterwards the kind of the element is identified by its id number. Each kind of element has its own function that is called for storing the parameters on the blackboard storage and the management frames. This first modify state is finished when no element is left over. In the first instance of the second phase it is proven if at least one command processed in the modify state was a publish command. Therefore, a flag is set on occurrence of a publication.

In the prove state the test flag can be easily tested on activation. If no value was published before, the blackboard is finished and resets its deadline afterward. Because no value was published, no subscriber needs a notification.

There is no new value for which a subscriber may wait. The next thread will be chosen by the scheduler and this one is not the blackboard thread, because its deadline has been set back to infinity and its thread state has been set to

the blackboard value. But if at least one publish command was processed in the modify state, there is the possibility that one or more subscription filters are fulfilled and so a notification process has to be launched.

First the events with possible notifiable subscribers have to be found. Therefore, the search algorithm goes through all management frames and stop if it found one with a set publish flag. This flag is cleared and for all active subscribers of the event it has to be tested if their subscription filter matches to the stored publication. If the test result is positive, the notification of the subscriber is interpolated. In this notify state, the stored subscription data is used to clear the semaphore at the given address and to set a new deadline for the thread with the stored thread id.

The deadline is the latest point in time when the notification arrives at the subscriber. For setting the new deadline, the deadline has to be calculated. The formula for computing this point in time is t_deadline = t_ published + Dt_deadline where Dt_deadline is the value for the deadline given at the subscription and t_published is the point in time when the value was pushed from the publisher thread in the blackboard queue.

In the implementation, the calculation is started as early as possible or more exactly directly after the positive subscription test. The calculated deadline time is passed to the scheduler by setting the new deadline time for the subscriber thread in the thread table. If this thread has already a deadline, which is lower or equal than the new calculated value, the old will not be overwritten. Then the old deadline is tighter as the new one and so the old value dominates the new one. When all active subscriptions are proven for the event, the next event with an enabled publish flag is searched. The blackboard process has finished when all management frames have been visited.

The elapsed time of the notification process is O(n) with n events or in worst case 6 * n with n events in total and all with a set publish flag. Normally a blackboard thread should be finished within a time slice. If not and the dispatcher is launched while the thread is active, the blackboard thread is chosen again by the EDF scheduler part and because current running thread and new thread are the same, no context switch is done. So the blackboard thread always finishes its task before another thread is chosen for execution.

6. Validation of the Implementation

First, we will show that if a deadline-enabled notification for at least one subscriber is available, the deadline for the subscription thread is set and the thread is activated within the deadline. We assume that all deadline threads are schedule-able concerning their deadlines, since this is our prerequisite as mention before. So the publisher willing to publish a new value starts the publish request() function. Inside this function call, the deadline for the blackboard thread is set to an initial value. This leads to an activation of the blackboard thread after the currently running thread is interrupted, because of the initial deadline value. No other thread could have a lower deadline than the blackboard thread.

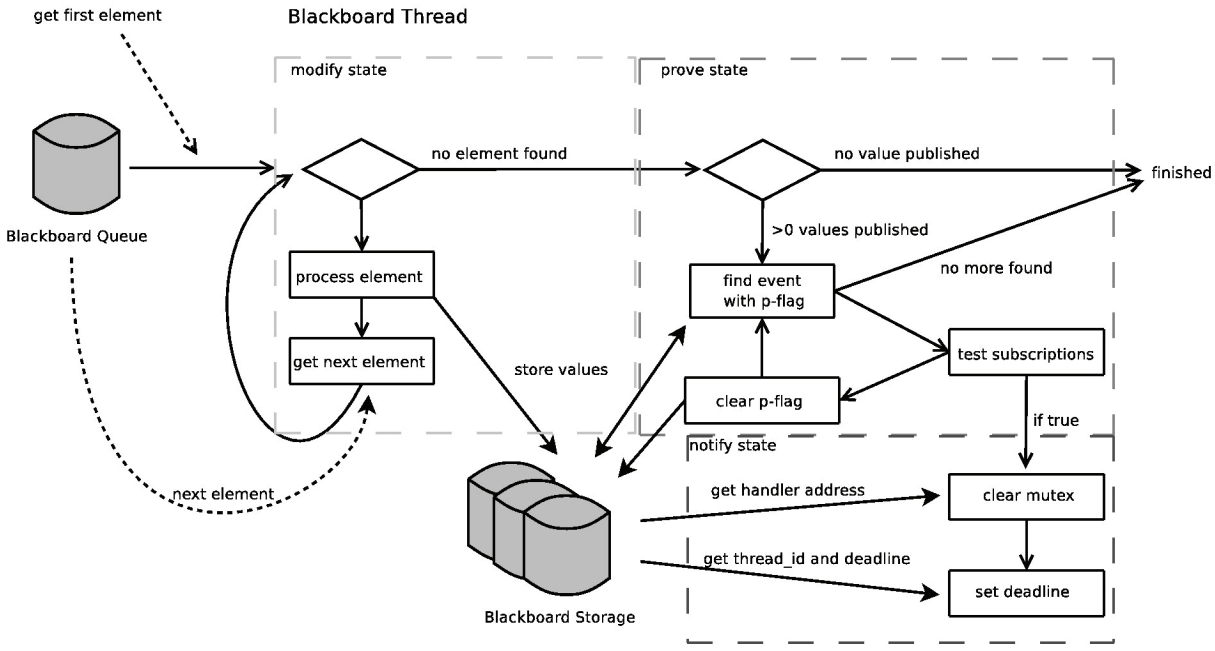

Figure 4. Blackboard thread process.

The largest period of time that a thread could be activated is the duration of one time slice, here 20 msec. At the latest of this time period the blackboard thread is running, processes the publish request and starts the notification process. Because the publish/subscribe paradigm forbids to block the publisher, starting the blackboard thread directly as the next thread is the earliest permitted point in time. Beside the activation time of the blackboard thread, this thread has to be nonpreemptive. This is guaranteed by the fact that the blackboard thread keeps its initial deadline value until the blackboard task has finished.

If the scheduler tries to switch the threads, the EDF part of the scheduler will find the blackboard thread as the one with the highest priority and will keep running the blackboard thread. So if the blackboard thread has finished its task, every mutex for the subscription notification has been cleared and all subscription threads have their deadline. After the blackboard task execution the scheduler is ready to activate the deadline enabled threads in their priority or better deadline defined order. Secondly, we will discuss the problem of mutual exclusions. Shared resources are normally saved by using the mutual exclusion concept. But by using a real-time scheduler and mutual exclusion, a so called priority inversion can occur. This means that if a low priority thread allocates a resource by setting a mutex, a later try to allocate a high priority thread is blocked and has to wait until the mutex is cleared. So there is the possibility that the blocked thread misses its deadline, because it has to wait for the lower priority thread to release the mutex.

Therefore, it is important to activate a high priority thread early to prevent priority inversion. The first condition, the early activation of the highest priority thread, is fulfilled by the EDF scheduler, because the highest priority means that the thread has the lowest deadline of all ready threads. The second condition needs some modifycation of the mutex processing. For our implementation we have chosen the concept of priority inheritance as described in [5] for preventing the priority inversion problem. Therefore a thread that tries to set a mutex another thread with a lower priority has blocked, helps the mutex owner by passing on its lower deadline to the thread. The original deadline is preserved within the mutex structure and reverted when the mutex is released.

The needed modifications in the MANTIS source code are only marginal. A blocked thread is in the so called blocked thread state and its deadline is not influenced. So after releasing the mutex the blocked thread gets back in the ready state and is - because of its lower deadline - the first thread that will be activated. Its waiting time abetted its priority class, now it is even more important to schedule this thread.

The following shows parts of the mutex code. On each mutex set attempt it is proven if the mutex is free or not. If not it is tested if a priority inversion situation has occurred or not. This could only happen if the current thread deadline is lower than the deadline of the mutex owner thread. Than the original deadline of the owner thread is stored in the mutex structure and afterwards its deadline is overwritten with a copy of the current thread's deadline. Since the current thread is blocked, the thread which has set the mutex is activated afterwards. Only the blackboard thread could have a sooner deadline as the owner thread, all other threads have lower priority.

In the code that unlocks the mutex, only one additional line was added for testing on existence of a stored old deadline and reverting it. On every successful mutex unlock, this line is executed before the scheduler function is called. Because MANTIS OS differs between mutex and semaphore, mutex and semaphore code is separated. So also the semaphore code is modified in the same manner as described for the mutex code.

6.1. A Simple Test Case

This section discusses the use of the implemented publish/subscribe architecture in MANTIS OS by an introducing example application. This application consists of three autonomous threads with two of them periodically sleeping and one being only activated on notification of the blackboard. The application is spilt into tasks and each task has its own thread. One of the threads takes over the publisher part and the two others are processing the published data.

The publisher process, which is in a sensor application the part which enquires a sensor, publishes a number and sleeps afterwards for 30 milliseconds by executing the mos sleep() function of MANTIS OS. The thread is rescheduleable after the expiry of the sleeping time. The only task of this thread is to regularly push data to the blackboard, which has to react afterwards.

The two threads with the subscribers for the publisher event are split in a thread, which is only activated on notification (subscriber1) and one which is periodically and on notification activated (subscriber2). The second subscriberthread sleeps for 10 msec after each execution period.

In each running state of the publisher thread, one value is published before the state is switched to sleeping. The blackboard thread is executed afterwards and modifies the management frame and stores the value. Afterwards, the blackboard tests all subscriptions of the subscribers of the published event. If the subscription rule is valid, the blackboard notifies the thread with the subscriber by setting a deadline and clearing the correct mutex. Because subscriber1 has a shorter deadline defined as subscriber 2, we have included debug code so that we could measure the execution times. The runtime diagram (6.1) shows the execution times of the MANTIS OS threads

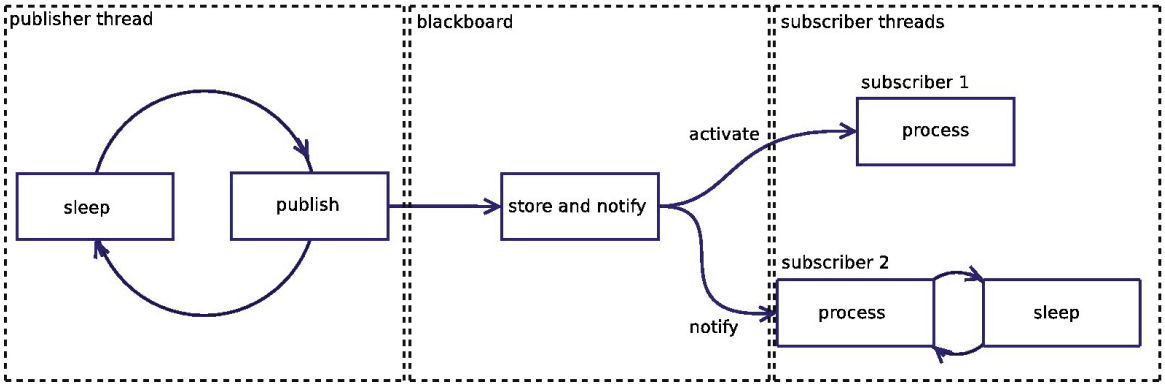

Figure 5. Test case scenario.

for this example. The example implementation was executed on a Mica2Dot sensor node. The node was connected to the host PC via JTAG interface. On the host, the avr-gdm application in combination with the averice JTAG communication software was used.

Each round-robin time-slice has 20 milliseconds, but as the diagram (6.1) shows, no thread needs a complete time-slice of its own. The publisher thread is activated every 30 msec as supposed and executes within 1 millisecond. The blackboard thread is launched directly after the sleeping phase of the publisher thread begins. Then subscriber1 is activated before subscriber2, because of its sooner deadline. The diagram also shows long periods of the idle thread, which means that the micro-controller has relatively long sleep periods. Two threads, the blackboard and the subscriber1 thread, are only activated when its necessary. So a long active idle thread is possible, which allows a low overall power consumption. Also, less context switches are the consequence. So besides a better response time, the multi-threading overhead is reduced by using the blackboard thread state intensely. Blackboard stated threads are executed in a just-in-time manner.

In this example, the publisher thread issues new values and than goes to sleep at once. In the next scenario, we have added a simple waiting routine directly after the publication command. So it is possible to simulate a working task, since in a normal application the publishing thread needs some time for monitoring the sensors.

The problem is that the publish/subscribe system should not block the running task. So it is possible that the current thread is not interrupted until the time slice is over, originate in that the developer has not used a interrupt function like the sleep command before the time slice interrupt occurs. If a defined notification deadline is sooner as the remaining time of the time slice, then the deadline is not grantable. In a test series we will show that a defined deadline is met when possible. In the other condition it is not predictable how long a subscriber has to wait on the notification additionally. **Figure 6** shows the results of the test series. The results show the time the notification of the subscriber needs with reference to the time the publisher thread processes the waiting task after the publication. The deadline in this example is set to 10 msec for the subscriber. As long as the additional processing time of the publisher is under 10 msec, the deadline is met. All notifications are transferred within10 msec. But in the scenario when the additional processing time is over 10 msec, the values fluctuate. As the results show, the deadline of the notification is fulfilled as long as the publisher thread is interrupted in good time. If the developer does not let the thread interrupt after a publiccation and processed s time-consuming task instead, the deadline is not met. But the results also show that if the total additional processing time is over the time slice value of 20 msec, the time needed to notify the subscriber level off at about 11 msec. In this condition, the elapsed time slice is responsible for a thread switch. Because then the blackboard thread has the highest priority, the blackboard processes the publication and notifies the subscriber before the interrupted thread is rescheduled again. This example shows that a developer has to take care, if the deadline is set to a value under 20 msec, and the publisher threads task is not interrupted soon enough after the publication.

If the developer chooses for the subscriber (in this example) a deadline above 11 msec, the deadline would be met in the over-load scenario. A deadline of 20 msec and above would be always met. The problem is that the developer not always knows how long the blackboard thread has to wait for the interruption of the publisher thread. Therefore, the developer should execute a dispatch thread() or mos sleep() directly after the publication. If this is not done, then the ime needed between publication and interruption has to be stimated and added to the deadline value. The test results how that it is important to choose an acceptable value for he deadline. A solution may be to increase the deadline alue by one unit every time the deadline is not met. Further esearch should show if this method is feasible and useful. The

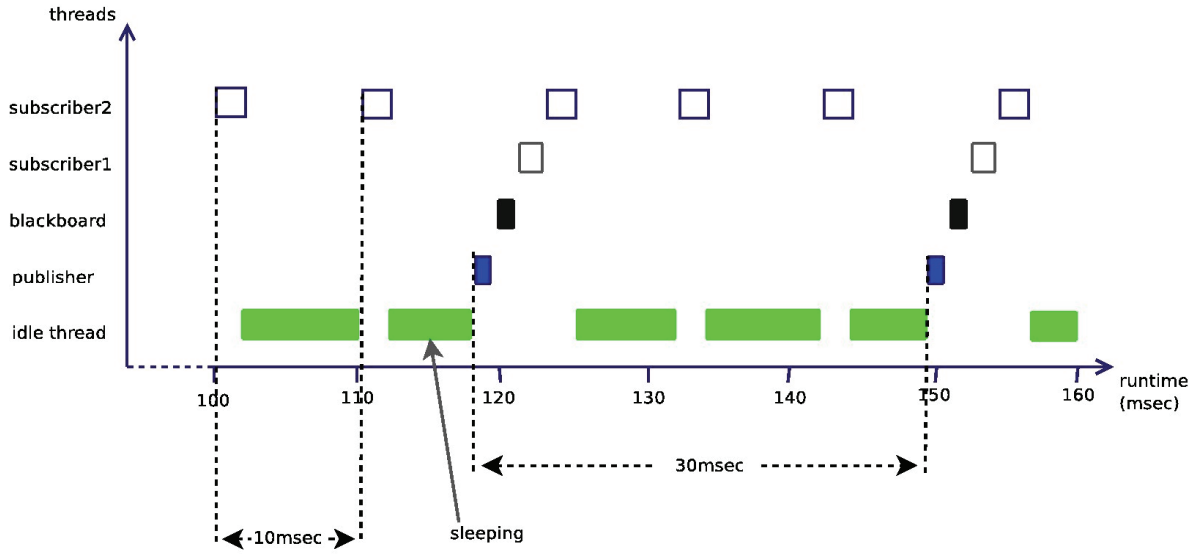

Figure 6. Run-time diagram of the test case example.

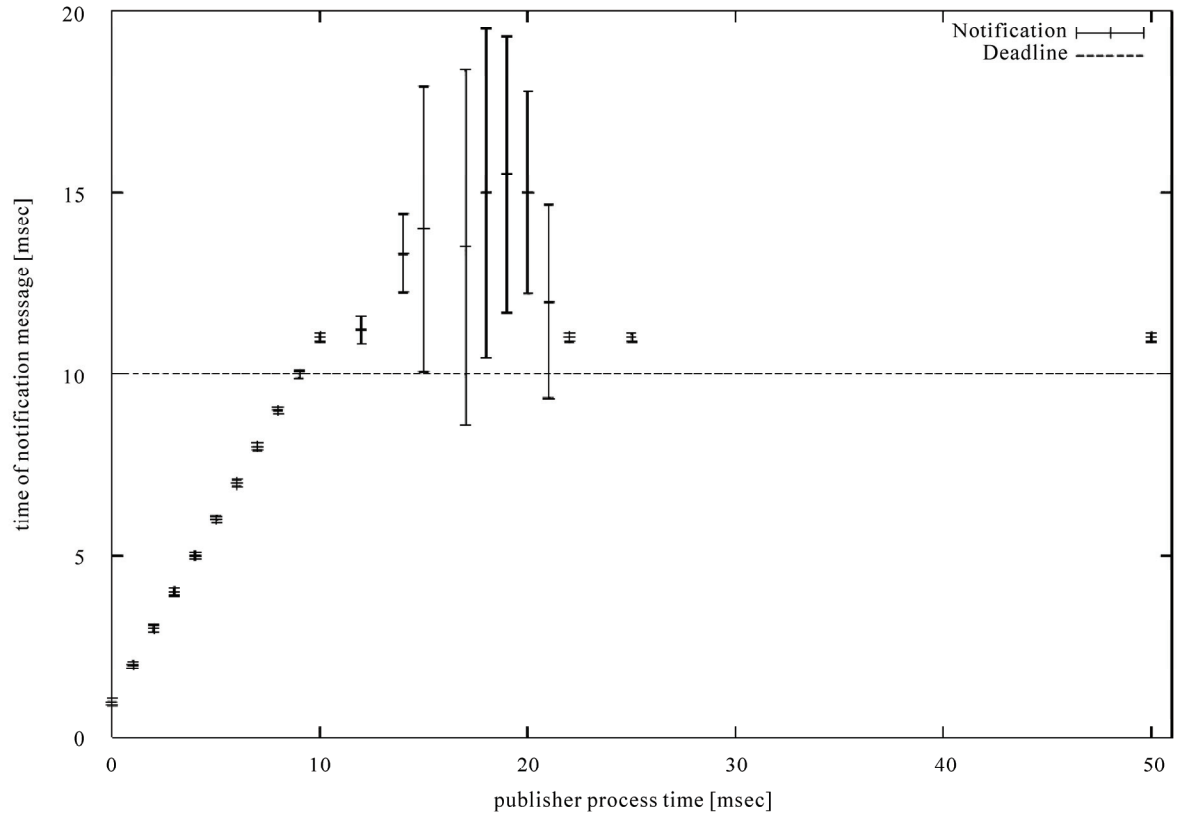

Figure 7. Results of the test case measurement.

concept is similar to the deadline aging concept, used y several real-time scheduler implementations. We will not urther discuss this problem, because one conceptional assumption s that the thread is interrupted early enough when ubscribers have to be notified.

The example shows that the deadline of the subscriber is met f possible. In the condition above 10mesc, the point in time hen the subscriber is notified is not pre-

dictable anymore as expected.

7. Conclusions

We have shown in this paper that it is possible to develop and implement a blackboard-based publish/subscribe architecture for sensor nodes which is also able to fulfill real-time requirements. Next to the real-time ability the introduced architecture flexible concerning the amount of subscribers, subscriptions and events but on the other hand memory space is economically used. We have shown this by comparing our concept against a memory sparing but static architecture and against a total flexible but memory wasting dynamic storage concept. Our blackboard-based approach shows a good tradeoff between memory usage and flexibility. Also, it is shown that the three publish/subscribe constraints, namley time, space and synchronization decoupling, are redeemed. Therefore, sensor application designer have now the possibility to use our available framework for easily sharing values between different program parts at least for MANTIS OS based applications. The development of crosslayer conceptions benefits from our architecture since the central storing of the published values is managed through our presented blackboard. All publishing and notifying processes are hidden for the application developer. The needed operating system modifications especially of the scheduler was shown exemplary for the MANTIS OS. The postulated real-time attribute of the publish/subscribe architecture made it necessary to convert the existing scheduler implementation towards an EDF scheduler. Not only it was shown that it is possible to make a sensor operating system real-time enabled but also the shown architecture benefits from the real-time ability. Subscribe are able to define deadlines concerning the notification which enables us to keep track of the system even in an overload situation where deadlines can be fulfilled anymore.

8. References

[1] P. Eugster, P. Felber, R. Guerraoui and A. Kermarrec, The Many Faces of Publish/Subscribe, 2003.

[2] H. Karl and A. Willig, "Protocols and Architectures for Wireless Sensor Networks," John Wiley & Sons, Hoboken, 2005.

[3] A. Köpke, V. Handziski, J.-H. Hauer and H. Karl, "Structuring the Information Flow in Component-Based Protocol Implementations for Wireless Sensor Nodes," *Proceedings of Work-in-Progress Session of the* 1*st European Workshop on Wireless Sensor Networks* (*EWSN*), Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, Berlin, January 2004, pp. 41-45.

[4] B. Segall and D. Arnold, "Elvin has Left the Building: A Publish/Subscribe Notification Service with Quenching," 1997.

[5] M. Hohmuth and H. Härtig, "Pragmatic Nonblocking Synchronization for Real-Time Systems," *Appears in the Proceedings of the* 2001 *USENIX Annual Technical Conference*, Boston, 2001, pp. 217-230.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson and R. Han, "ANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *Mobile Networks & Applications*, Vol. 10, No. 4, 2005, pp. 63-79.

[7] N. Rozanski and E. Woods, "Software Systems Architecture: Working With Stakeholders Using View-Points and Perspectives," Addison-Wesley Professional, Reading, 2005.

