In Proc. of 5th European Conference on Wireless Sensor Networks (EWSN), Bologna, Italy, January 2008 Springer, accepted for publication.

A Component Framework for Content-based Publish/Subscribe in Sensor Networks

Jan-Hinrich Hauer, Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz

Telecommunication Networks Group
Technische Universität Berlin, Germany
{hauer,handzisk,koepke,willig,wolisz}@tkn.tu-berlin.de

Abstract. Component-based architectures are the traditional approach to reconcile application specific optimization with reusable abstractions in sensor networks. However, they frequently overwhelm the application designer with the range of choices in component selection and composition. We introduce a component framework that reduces this complexity. It provides a well-defined content-based publish/subscribe service, but allows the application designer to adapt the service by making orthogonal choices about: (1) the communication protocol components for subscription and notification delivery, (2) the supported data attributes and (3) a set of service extension components. We present *TinyCOPS*, our implementation of the framework in TinyOS 2.0, and demonstrate its advantages by showing experimental results for different application configurations on two sensor node platforms in a large-scale indoor testbed.

1 Introduction

The publish/subscribe interaction scheme is a high-level service abstraction that is well adjusted to the needs of large-scale distributed applications [1]. The scalability and robustness of the scheme stem from the indirect and asynchronous type of interaction and make it particularly suitable for creating data-centric sensor network applications. In the content-based publish/subscribe variant, subscribers express their interest in events by injecting subscriptions into the system that contain constraints on the properties of the events. Publishers that generate events post notification messages to the system, and when a notification matches the constraints of a registered subscription, it is delivered to the corresponding subscriber (Fig. 1).

Any design for sensor networks is subject to tight constraints in terms of energy, processing power and memory. These constraints frequently drive developers to pursue vertically integrated solutions that are highly-optimized for specific scenarios but lack flexibility [2, 3]. Breaking up the design into fine, self-contained and richly interacting *components*, has proven to be a viable approach for resolving the tension between the need for reusability and the efficiency costs of abstractions [4]. The flexibility provided by the component modularization,

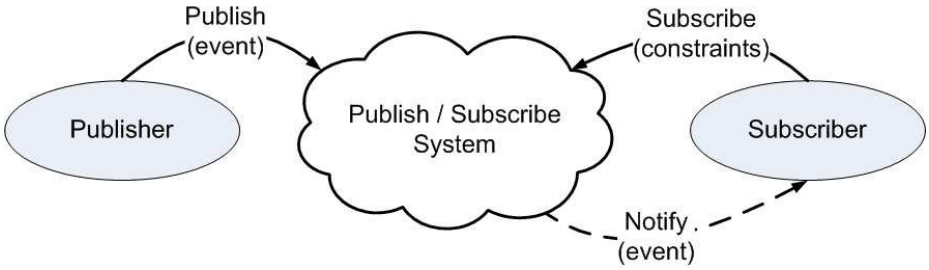


Fig. 1: A content-based publish/subscribe system.

however, also carries the risk of overwhelming the application developer with the range of options for component selection and composition.

Component frameworks can reduce this complexity by imposing structure on top of the component model in the form of composability restrictions and by offering well-defined, service-specific interfaces to the rest of the system. Designing a component framework is a fine balancing act of *fixing* the service interface at a level of abstraction that will maximize the gains in productivity, while keeping those parts of the architecture with significant impact on the performance *flexible* enough to be able to benefit from domain-specific optimization [5].

In this paper we present the design, implementation and evaluation of a flexible component framework that provides a well-defined content-based publish/subscribe service, but allows the application designer to adapt the service by making orthogonal choices about the communication components for subscription and notification delivery, the supported data attributes, and a set of service extension components. The framework uses an attribute-based naming scheme augmented with metadata containing soft requirements for the publishers and run-time control information for the service extension components. It supports different addressing schemes and interaction patterns.

In the next section, Sect. 2, we present the general architecture and design rationale behind our concept. Section 3 contains a more in-depth discussion of the implications from the decoupling between the publish/subscribe core and the communication protocols. We evaluate the generality and flexibility of our design in Sect. 4, using experimental results from large-scale deployments of *TinyCOPS*, the implementation of our framework in TinyOS 2.0, on two sensor node platforms, under scenarios involving different types of applications, network protocols and extension components. In Sect. 5 we discuss the related work and Sect. 6 summarizes and concludes the paper.

2 Architecture

In this section we present the main features of the architecture in a top-down fashion, covering the naming scheme and API as well as the internal decomposition and extension facilities of the framework. The discussion on the implications

Basic Publish/Subscribe API	Extended Publish/Subscribe API
Subscriber:	
`Subscribe`(C)	`Subscribe`(C | M)
`Unsubscribe`()	`Unsubscribe`()
`Notify`(A)	`Notify`(A | M)
Publisher:	
`Publish`(A)	`Publish`(A | M , `push`)
	`Listener`(C | M)
Matching:	
	`Matching`(C , A)

Fig. 2: The basic publish/subscribe API and the extended version that is provided by our framework. A square represents a set of constraints (C), metadata (M) or attribute-value pairs (A). The extended `Publish` primitive takes an additional `push` parameter which influences the matching point and is explained in Sect. 3.2.

of the decoupling between the publish/subscribe core and the communication protocols is deferred to Sect. 3.

2.1 Naming Scheme and API

To represent subscription and notification content, our framework adopts the attribute-based naming scheme presented in [6]: a subscriber expresses its interest in data through a conjunction of constraints over attribute values. Disjunctive constraints need to be expressed as separate subscriptions. A constraint is a *(attribute, operator, value)* tuple and represents a filter on attribute data, for example (*Temperature,* $\geq$*, 30*). Publishers publish data in form of notifications containing *(attribute, value)* tuples, for example (*Temperature, 32*). A notification matches a subscription if every constraint in the subscription is satisfied by a *(attribute, value)* tuple in the notification.

If a subscription consisted only of constraints over attribute values a subscriber would not be able to explicitly influence the properties of the communication or sensing process like, for example, the sampling rate. Such control properties are conceptually different from the data constraints and can usually not be matched by corresponding *(attribute, value)* tuples in the notification. We extended the basic naming scheme by allowing subscribers to include *metadata* in subscriptions. Metadata is either exchanged between publisher/subscriber components or plays a key role in controlling service extensions (Sect. 2.3). It represents control information with soft semantics and is excluded from the matching process.

Metadata is represented by one or more *(attribute,value)* pairs, for example (*SamplingRate, 10*). Conceptually, it represents a notification that the subscriber attaches to the subscription. Metadata is specified per subscription and multiple active subscriptions may have different values for the same metadata attribute.

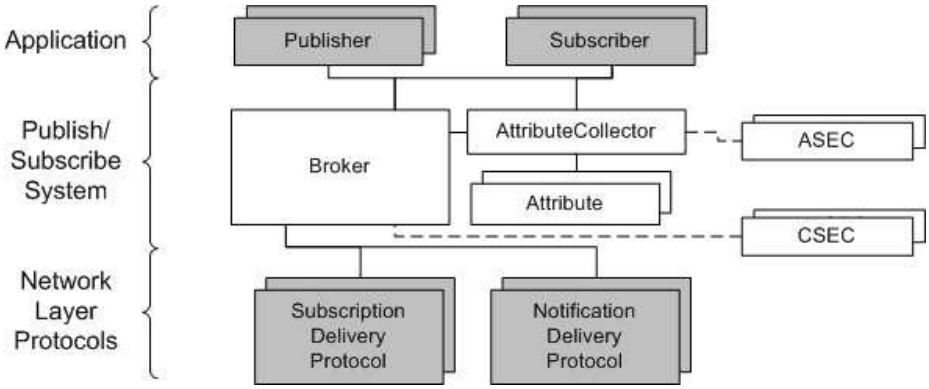


Fig. 3: The high-level decomposition of the framework.

Since metadata is non-binding a publisher may apply local optimization techniques: for example, in order to reduce sampling overhead the publisher may decide to combine two subscriptions that address the same attribute by sampling only once with an average sampling rate when the rates are similar, or using the maximal sampling rate when not.

The modified naming scheme is supported by two extensions of the basic publish/subscribe service: a "listener" service and a "matching" service. The "listener" service can be used to inform the application about newly arrived subscriptions, which it then can inspect to decide whether to start or stop publishing notifications. The "matching" service may be used by the publisher to check whether a set of attributes disqualifies it from matching a registered subscription. If, for example, the first collected attribute violates a constraint, collecting further data is pointless. When used, these primitives may result in a tighter coupling between publishers and subscribers than in the traditional model, but they have the potential to increase the efficiency of the data collection process, resulting in overall application performance gain. Fig. 2 compares our extended API with the basic publish/subscribe service.

2.2 Core Decomposition

Figure 3 shows the decomposition of the framework. The Publish/Subscribe service is distributed and the figure represents an instance of the framework on one sensor node. A publish/subscribe application is divided into a variable number of *Publisher* and *Subscriber* components. A Publisher component can listen for subscriptions, collect data and publish notifications and Subscriber components can issue subscriptions and receive matching notifications. The *Broker* component provides the publish/subscribe service to the application, it manages the subscription table and it can apply the matching algorithm to filter out notifications that do not match a registered subscription.

The data ("events") that subscribers can subscribe to and publishers can publish are encapsulated in *Attribute* components. In addition to a data collection interface, an Attribute component must provide a matching interface that compares two of its data items based on an attribute-specific operator. The motivation is twofold: first, an Attribute component represents functionality that Publisher components should be able to reuse and access independent of the specific attribute properties (data type, metric, etc.). Secondly, matching operators are usually attribute dependent: for example, when sensor readings are affected by hardware-related jitter, the operator "=" should not be interpreted as the exact equality of two values. To increase modularity and keep the core matching algorithm decoupled, this information should be provided by the particular Attribute component.

Within the network, all attributes and operators are represented by integral identifiers. Attribute identifiers are globally unique, while operator identifiers are unique within the scope of a particular attribute. On the edge of the network a translation between identifiers and attribute semantics is performed using XML maps.[1] The *AttributeCollector* component structures access to the attributes: it maps a request based on the attribute/operator identifier to an actual Attribute component that is registered *at compile time* (but could even be added at runtime by dynamic over the air code updates).

2.3 Service Extensions

At the beginning of this section, we introduced the notion of metadata: by including metadata in a subscription a subscriber can influence the communication and sensing process. Often, such control functionality can be isolated in self-contained components for reuse in different applications. For example, a *caching* component could decrease sampling overhead by buffering frequently accessed attribute data when the considered data attribute has high direct sampling costs or is computationally intensive, like feature extraction from acoustic signals. We call such components Service Extension Components (SEC). A SEC represents reusable functionality that can be plugged into the framework without modification of existing code. A SEC can realize an additional service (as in the caching example) or extend the communication path with additional control information (timestamps, message sequence numbers, etc.). A SEC is associated with one or more dedicated metadata attributes, for example the maximum allowed caching duration, and made available by the application designer *at compile time* (it could even be added at runtime by dynamic over the air code updates). A SEC can be activated dynamically by a subscriber on a per-subscription basis by including an appropriate metadata attribute in the subscription.

The framework supports two different types of extension components, *Communication* SEC (CSEC) and *Attribute* SEC (ASEC). A CSEC can intercept incoming packets before (and outbound packets after) they are processed by the

[1] For lack of space we refer the reader to our XML specification made available as part of our implementation described in Sect. 4

Broker in order to scan the included metadata attributes and, if applicable, perform a specific operation. It can, for example, be used to aggregate notification messages in order to reduce overall network traffic. Since CSECs can also be used to add control information (timestamps, etc.) a subscriber can use the CSEC(s) located on the publisher nodes to (conceptually) assemble its own message header by adding appropriate metadata attributes. ASECs are used analogous for attribute access: they can intercept the requests for attribute data and instead return buffered or processed data dependent on the metadata included by the particular subscriber.

In combination, metadata and SECs realize a soft "control path" in parallel to the basic publish/subscribe "data path". Since SECs are self-contained components and can usually be designed agnostic to data attribute semantics they are easily reusable in different applications and on different platforms. However, when multiple SECs are in use, their ordering must be defined by the application designer because it may influence their overall semantics.

3 Communication Decoupling

The classical content-based publish/subscribe systems have tightly integrated filtering, routing and forwarding mechanisms [7, 2, 8] resulting in more optimized, but less flexible solutions. Our framework departs from this tradition and decouples the communication mechanisms from the publish/subscribe core (Fig. 3). The core broker component has clean interfaces towards the external protocol components, thus *trading some of the optimization potential for increased flexibility* in selecting the subscription and notification protocols.

By exposing the choice of the protocols to the application designer, our framework allows the adaptation of the publish/subscribe service to the specific needs of the application. The type of the communication protocols as well as their energy consumption are likely to have a huge impact on the overall performance, and the application designer should be aware of these implications [9] to make an optimal selection for the particular application. In the following we concentrate on three important aspects of this decoupling and on the architectural features of the framework that address them.

3.1 Different Addressing Models

In contrast to the integrated solutions that rely on a pure content-based routing and forwarding mechanisms, the flexibility of our framework raises the challenge of interfacing with communication protocols that support different dissemination patterns like broadcast, multicast, convergecast, point-to-point, etc., using various addressing models like address-free, id-centric or geographic addressing.

To support this wide range of communication mechanisms we rely on three architectural features. First, the core of the framework is agnostic to the underlying addressing model, and all information relevant for operation of the service is encapsulated in the form of metadata, subscription filters or notification data.

Secondly, the interfaces towards the subscription and notification delivery components are kept address-free. Finally, all the addressing information for the communication protocols is provided/consumed by their respective components or wrappers, while the framework provides hooks that facilitate its encapsulation and tunneling when so required. To illustrate this process, we examine the handling of the address information on the subscription and notification path separately.

Subscription Path. On the subscription path, a common delivery pattern is one-to-all (broadcast): a subscriber wants to receive notifications from any publisher with matching data in the network. This pattern is naturally supported by the address-free interface. In the case of one-to-many (multicast), the subscriber application defines the scope of the subscription delivery expressed as metadata attribute (hop-count, geographic scope, etc.) inserted in the subscription. The metadata is transparent to the publish/subscribe core and after registration of the subscription in the subscription table its content is passed onto the respective subscription delivery protocol component. The protocol component (or a thin wrapper) extracts the scoping attributes from the subscription content (via suitable accessor functions provided by the core) so that they can be used or translated into corresponding protocol parameters. Depending on the nature of the scoping parameters, this mechanism might increase the coupling between the subscriber and the subscription delivery protocol, but the publish/subscribe core does not require any adaption. An id-centric, point-to-point subscription delivery, although very atypical communication pattern for a publish/subscribe application, can also be supported with the this mechanism.

Notification Path. On the notification path, the message delivery patterns are potentially more diverse. To abstract from address information and decouple the application from the particular addressing scheme of the notification delivery protocol, we employ the mechanism visualized in Fig. 4: after a subscription has been issued by the application (1), the notification delivery protocol component (on the subscriber node) can use a hook provided by the core to add the local address of the subscriber as metadata information in the subscription, just before it is disseminated in the network (2).

The addressing information may be expressed using any naming/addressing scheme because the metadata value is transparent to the publish/subscribe core. After the subscription has been disseminated (3) and registered in the subscription tables of potential publishers (4), whenever a notification is published (5, 6) the notification delivery protocol instance on the publisher node can extract the particular source address of the subscriber and use it as address parameter (7).

Thus, the core provides two hooks to the notification delivery protocol: one for attaching the local address to a subscription on the subscriber node and one for reading it out on the publisher node. Both hooks are used optionally – if the notification delivery is, for example, based on flooding or uses data-centric addressing, it will neither add nor read any metadata to/from a subscription.

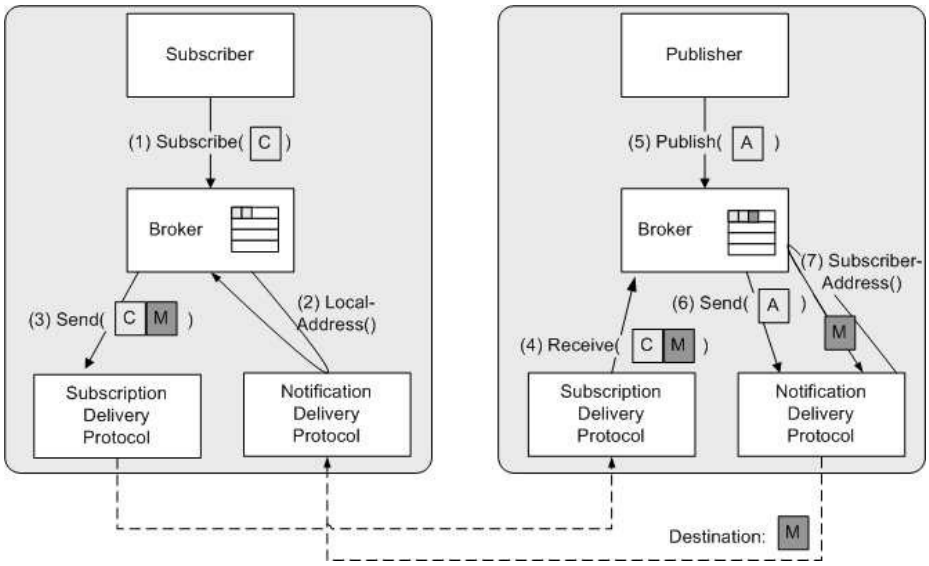


Fig. 4: Enabling different addressing schemes by tunneling address information as metadata between a subscriber and a publisher. Squares represent constraints (C), metadata (M) or attribute-value pairs (A).

In this way, the Broker, Publisher and Subscriber components remain shielded from the addressing models used by the communication substrate. Even more, the addressing on the subscription and the notification path is decoupled so different addressing models can be used with respect to each other.

At the cost of increased coupling between the core and the communication substrate, the same architecture can even be used to support a "classical" integrated content-based routing protocol. Through single-hop subscription scoping, the core can relinquish complete control over subscription injection and forwarding to the underlying integrated protocol, allowing complex schemes like subscription coverage or merging to be implemented. The resulting duplication of state (subscription table entries, etc.) can be reduced to a certain degree using hooks exported by the core facilitating buffer space sharing. The design of this support is one focus of our future work.

3.2 Control of the Matching Point

The departure from the integrated content-based routing and forwarding approach, brings to the surface the question of the "matching point" in the network, i.e. the point where the published notifications are matched against the content filters in the subscriptions. Since the subscription and the notification messages are delivered by potentially separate protocols that do not explicitly share common state, a conscious decision has to be made about where in the

network this information would confluence so that it can be passed to the core for matching.

A misplacement of the matching point with respect to the application requirements and the selected communication protocols can result in significant performance penalties as notifications or subscriptions needlessly consume precious networking resources. In general, the optimal location of the matching point depends on many factors like network topology, ratio of publisher to subscriber nodes, frequency of subscription/unsubscription and publication, selectivity and locality of filters, etc.

Our framework supports two major scenarios by default: the filter matching is either applied on the publisher or on the subscriber nodes. Our decision is motivated by several observations. In many sensor network applications, we are faced with either a "pull" or a "push" interaction pattern, i.e. either a small set of subscribers is interested in notifications generated by a much larger set of publishers, or vice-versa, many subscribers are interested in the notifications from a smaller number of publishers. This means that the optimal approach involves either a network wide subscription dissemination with filter matching performed on the publisher nodes or network wide notification dissemination with matching performed at the subscriber nodes [9].

For the cases in between these two extremes, the framework can be extended with a CSEC that determines the optimal points using an integrated content-based routing and forwarding protocol, or from a dedicated "matchmaker" service [10]. The broker component provides a hook that CSECs can then use to execute the matching algorithm, without introducing tight coupling between the underlying protocols and the publish/subscribe core.

3.3 Protocol Impact on the Service Semantics

The selection of the subscription and notification delivery protocols is also influenced by the non-functional requirements of the particular application. For example, the application designer may be faced with a scenario where subscriptions need to be updated frequently and not reaching exactly all of the available publishers is acceptable. In this case a protocol for probabilistic best-effort subscription dissemination may be sufficient. On the other hand, an application may require more reliable dissemination of subscriptions and is willing to accept continuous control traffic in the background. In this case a reliable dissemination algorithm would be more suitable. If sufficient resources are available, the application designer might even choose multiple subscription or notification protocols in parallel.

Our framework does not impose any limits on the quality of service provided by the underlying communication protocols, effectively treating them as black box components. Whenever a subscription is issued or a notification is published, the framework will eventually convert the subscription/notification content into payload of the selected protocol component. The choice of protocols therefore has direct impact on the delivery semantics of the publish/subscribe messages, and with that on the semantics of the provided service.

The core itself is not influencing the quality guarantees of the underlying protocols, but SECs can be used to this aim, for example, by periodically retransmitting subscription messages, temporarily storing notification messages, etc. We contrast the performance and the semantic effects of different types of subscription dissemination protocols in the evaluation Sect. 4.

4 Evaluation

Assessing the full impact of a component framework is a difficult task. As with any other software architecture, the most reliable feedback ultimately comes from surveying users after extended periods of day-to-day use. The development of a reference implementation and its evaluation, however, can be considered as an important first step towards this goal. A real prototype demonstrates that the general design can be implemented under the specific constraints of the target domain. Furthermore, through careful micro-benchmarking executed in controlled, yet realistic setting of modern sensor network testbeds, it provides an opportunity for gaining deeper insight into the specific feature set and the involved design tradeoffs.

To this end, we have developed a reference implementation of the framework, called *TinyCOPS*, using the TinyOS 2.0 [11] execution environment. TinyOS 2.0 is a second-generation component-based operating system for sensor networks that keeps many of the basic ideas of its predecessor while pushing the design in key areas like portability, robustness and reliability.

For the evaluation, we have opted against head-to-head comparison of *TinyCOPS* with other monolithic publish/subscribe frameworks because the overall performance of the frameworks is dominated by the underlying protocols and not the architectural features, there is currently no TinyOS 2.0 implementation of a monolithic publish/subscribe framework that would facilitate direct comparison, and even if such an implementation was available, the comparison results would be vulnerable to differences in the invested optimization effort.

Instead, the evaluation scenarios in this section are focused on demonstrating the flexibility and versatility of the design. We present results corroborating our claims that: (1) the framework exports significant performance tradeoffs to the application in an easy-to-use fashion, (2) the framework is general and flexible enough to support different interaction patterns and (3) the code, memory and execution time overhead is acceptable.

The presented data were obtained using TWIST [12], our multi-platform testbed for indoor experimentation with wireless sensor networks. TWIST provides basic services like node configuration, network-wide programming, out-of-band extraction of debug data and gathering of application data. It also allows to control the power supply for individual nodes, a feature we use to introduce node failures in the experiment described in Sect. 4.1. TWIST spans three floors of our office building and is populated with eyesIFX and Tmote Sky nodes in an approximate $3m \times 3m$ grid.

Starting with a simple data collection application scenario we present experimental results which show that the choice of dissemination protocols can exhibit considerable performance tradeoffs (Sect. 4.1). We then gradually increase the complexity of the application. Section 4.2 describes the integration of a send-on-delta service extension component and the effects on application performance and in Sect. 4.3 we show how *TinyCOPS* is used to extend the application with an alarm notification service realizing both "pull" and "push" interaction pattern at the same time. We then report on the code size requirements and processing time overhead in a typical *TinyCOPS* application (Sect. 4.4).

4.1 Tradeoffs in Protocol Selection

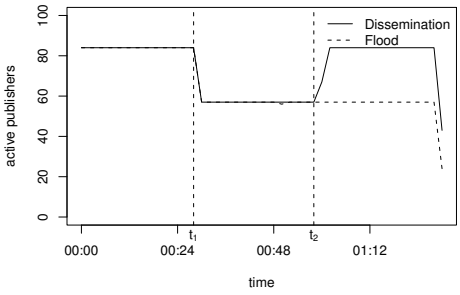
To demonstrate the tradeoffs that *TinyCOPS* exposes to the application designer through protocol selection we contrast two subscription delivery protocols: a plain flooding protocol (every node that hears a subscription broadcasts it to all its neighbours once) and an epidemic broadcast protocol. The latter is part of the TinyOS 2.0 core and based on the Trickle algorithm [13]: it lets nodes *continuously* broadcast status information about the subscriptions they have received. Whenever a node hears an older subscription than its own, it broadcasts an update to its neighbours. In contrast to the flooding protocol, which ends its operation after a short time, the epidemic protocol (called "TinyOS 2.0 Dissemination") remains active.

We created a simple *TinyCOPS* application with one subscriber and the rest of the nodes used as publishers. In our first measurement we disseminated the subscription via plain flooding. In the second, we used the TinyOS 2.0 Dissemination protocol. The modification is done by changing a single line of the *TinyCOPS* application configuration. For notification delivery in both measurements we use the TinyOS 2.0 Collection Tree Protocol (CTP) performing best-effort, multihop delivery of notifications to the sink of the tree (subscriber).

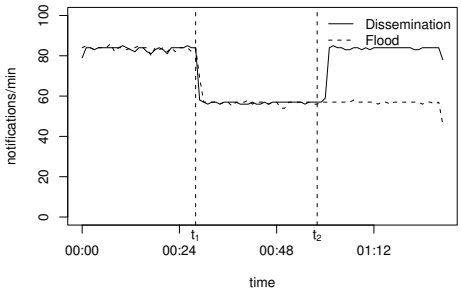
Both measurements lasted 90 minutes and were made with 86 Tmote Sky nodes, 85 publisher nodes and one subscriber (used as basestation, bridging to/from a PC). At time t_0 a subscription was injected asking for notifications to be published with a rate of one notification per minute by each publisher. After 30 minutes, at time t_1, one third of the publisher nodes (randomly chosen) were shut down and 30 minutes later, at time t_2, they were powered up again. Nodes that were shut down lost all state including subscription table entries.

Figure 5a shows the percentage of active publishers over time. We define active publisher as a node that has registered a subscription and published at least one notification. At time t_1 the number of active publishers decreases by about 30% due to our active power management. The difference between the protocols becomes visible at time t_2 when these nodes are powered up again: the epidemic Dissemination protocol quickly manages to spread the subscription to the recovered nodes, while the flooding protocol cannot (the subscription was injected only once at time t_0).

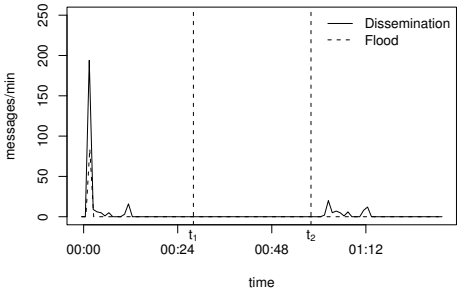
Figure 5b shows the changes in notification goodput perceived by the subscriber. We define notification goodput as the number of distinct notifications



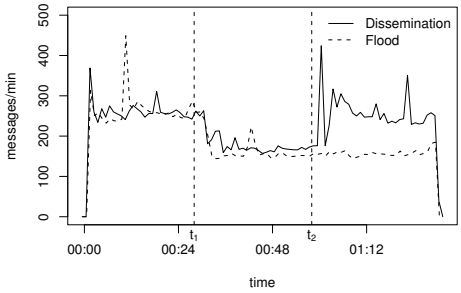
(a) Number of active publisher nodes.



(b) Notification goodput using TinyOS 2.0 Dissemination and flooding for subscription delivery.



(c) Subscription protocol traffic.



(d) Notification data traffic.

Fig. 5: Tradeoffs in Protocol Selection.

that arrive at the subscriber in a fixed time window of one minute. The curves almost match the number of active publishers and indicate a very good delivery ratio of CTP.

We used the serial backchannel of the testbed to let all nodes periodically output status information about the number of different messages they had sent over the wireless channel. This information allowed us to derive the traffic for subscription delivery as depicted in Fig. 5c. The figure visualizes the tradeoff between the protocols: the flooding protocol generates one message for each node in the network at the time the subscription is injected. The Dissemination protocol generates more messages, but is able to update the rebooted publishers at time t_2. Finally, our setup allowed us to determine the number of notification messages sent in the network by all nodes over a time window of one minute (Fig. 5d) – on average 3 messages were sent per notification, however our setup did not allow us to differentiate between retransmission and forwarded messages.

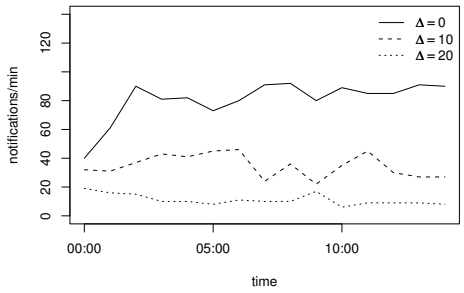


Fig. 6: Effects of a varying delta on notification goodput.

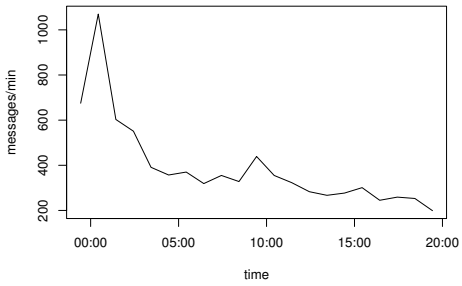


Fig. 7: Example of a "pull and push" interaction.

4.2 Adding a Service Extension Component

To decrease notification traffic and effective energy consumption, we modified the baseline application described in the previous section to realize a "send-on-delta" approach: notifications should be published only if the attribute values deviate by more than Δ from the previously published notification. Δ is defined by the subscriber and specified as the metadata of the subscription. To make the functionality resuable we implemented it as a CSEC *SendOnDeltaC* that intercepts outgoing notifications. It maintains a buffer for the last published notification, calculates the difference between the attribute values and suppresses the publishing if the difference is smaller than specified in the corresponding subscription. It is agnostic to attribute semantics and can be used for an attribute with integral data type.

We performed three measurements with 86 eyesIFX nodes and varying Δ and observed the effects on notification goodput as perceived by the single subscriber. The subscription asked for light sensor data to be published with a rate of one minute by each publisher and Δ was chosen 0, 10 and 20, where 0 means that all notifications are published and 10 and 20 represent the Δ of luminosity in absolute values of the raw eyesIFX light sensor reading. Figure 6 shows the effect on notification goodput: with a higher Δ, more notifications are suppressed by the CSEC, giving to the subscriber application a powerful runtime control over the tradeoff between data resolution and communication overhead.

4.3 Creating a Combined Push and Pull Application

Previous work [9] has shown that the interaction pattern between publishers and subscribers ("pull" vs. "push") can significantly affect application performance and should be carefully aligned with the ratio of publishers to subscribers. We created an application that included two Publisher components, one for periodic temperature data collection and one for generating fire alarm messages. We wanted the fire alarm event to quickly propagate to all rooms of the office building, but periodic measurements to be collected only by a single subscriber.

We therefore selected a single node to disseminate a subscription which notifications from the first Publisher component had to match (locally, based on the "pull" model). Fire alarms, however, were "pushed": whenever the second Publisher component detected a fire alarm regardless of any registered subscription, it immediately distributed the notification to all nodes in the network. The first Publisher component was "wiring" the subscription delivery protocol to the core and using CTP for notification delivery. The second Publisher component "wired" the flooding protocol for notification delivery.

Figure 7 shows a trace of the communication rates collected over 20 minutes on 85 Tmote Sky nodes. It represents the total number of packets sent by all nodes for a fixed time window of one minute. One subscription for periodic data collection is issued at the start of the measurement using the TinyOS Dissemination protocol, 10 minutes later we simulate a fire alarm, by sending a serial packet to one of the publisher nodes (randomly chosen). This node the started a flood of notification messages. The increase in traffic is visible by a small spike, however it is almost masked by the high level of CTP "pull" traffic.

4.4 Code Complexity and Execution Time Evaluation

We use the number of "Physical Source Lines Of Code" (PSLOC [21]), as well as the flash and RAM size, to evaluate the relative complexity of the major *TinyCOPS* components. The results in Fig. 8 show that the broker component is by far the most complex one in terms of code size. However, the framework allows composing lean applications, like the *StdPublisherP* component, which is a generic Publisher component included in *TinyCOPS* for convenience. It listens for a subscription and publishes corresponding notifications by querying the AttributeCollector for attribute data. The *Send-On-Delta CSEC* was introduced in Sect. 4.2 and can handle attributes of different integer sizes, a flexibility that is paid in increased flash consumption.

Building a *TinyCOPS* application involves making decisions about the Attribute as well as service extension components, the number of Publisher / Subscriber components and the respective communication protocols. As a result a typical TinyCOPS application configuration (defining the set of components that are linked together) can be composed with about 30 PSLOC – for comparison, the Dissemination protocol wrapper consists of 97 PSLOC.

To get an insight in the processing overhead introduced by *TinyCOPS* we measured the code execution time for the subscribe and publish operations on a Tmote Sky node. We used an application that subscribes to one attribute and measured the time it takes for a subscription/notification message to pass through the *TinyCOPS* core and protocol wrapper components (CTP / Dissemination wrappers). Under this scenario, the main tasks of the core were management of the subscription table and performing the matching algorithm. With a CPU operating frequency of 4 MHz, the subscription send-path took 144 μs and the subscription receive-path 281 μs. For notifications, the execution time

Description	Component Name	PSLOC	Flash (B)	RAM (B)
Broker	`BrokerImplP`	671	2838	19
Send-On-Delta CSEC	`SendOnDeltaP`	103	1126	12
Publisher	`StdPublisherP`	180	738	70
Subscriber (+ gateway)	`SubscriberGWImplP`	138	554	129
AttributeCollector	`AttributeCollectorP`	87	154	4
CSEC "Glue"	`CSECDispatcherImplP`	115	98	2
Temperature Attribute	`Msp430InternalTemperatureP`	20	2	-

Fig. 8: Code Size and memory footprint of an example *TinyCOPS* application: one Publisher, one Subscriber, one CSEC and one Attribute component.

for the send-path was 127 μs and 88 μs on the receive-path.[2] While the results are dependent on the time spent for matching an attribute-value pair with a constraint (we used the *TinyCOPS Ping* attribute), there are no additional "deferred" costs involved (for example, posting tasks or setting timers for later execution).

5 Related Work

The problem of providing an effective abstraction representing the sensor network services has been the focus of several prior works. The proposed solutions have ranged from database-like abstractions [3], application-specific virtual machines [14] to mobile agent systems [15] and abstract regions [16].

In [17], the SPIN family of protocols is presented, that use metadata-based negotiation phase to protect the network resources from unnecessary data exchanges. The content filtering capability in our framework has the same goal. In our case, the metadata part of the subscription message is used to convey a set of "non-binding" requirements from subscribers to publishers while the constraints express the imperative filtering. In SPIN, the metadata format is considered to be application dependent. We believe that the attribute-based naming scheme is flexible enough to support the majority of data-driven applications. Having a fixed naming scheme helps in optimization of the matching components and improves the portability of the application code.

Our naming scheme is much closer to the one used in the Directed Diffusion family of protocols [2], but we make clear distinction between the metadata and constraint and support attribute-specific operators. Conceptually, however, more important is the difference in the level of decoupling between the middleware service implementation and the communication protocols. Our framework not only delineates cleanly at this interface, it also allows for individual customization of the subscription and the notification delivery protocols and provides infrastructure for address information tunneling and matching point control.

MiLAN [18] is a flexible sensor networks middleware that continually tracks the application needs and performs run-time optimizations of the network and

[2] For comparison: the time between posting a task and executing its first line of code takes 48 μs (assuming an otherwise idle system)

sensor stacks to balance the application QoS and the energy efficiency. It is positioned as a general framework that can also be used with resource rich wireless technologies like IEEE 802.11 and Bluetooth. Our framework is concentrated on the class of relatively resource limited sensor network hardware [19], where compile-time optimization has comparably large impact, and where the run-time modifications are mostly limited to parameter tuning (like the selection of the concatenation timeout).

Like *TinyCOPS*, the Mires middleware [20] provides a publish/subscribe service, but uses the component architecture of TinyOS 1.x. It uses a topic-based naming scheme that lacks the expressiveness of the content-based filtering. While Mires envisions the possibility of introducing new services (like aggregation) using extension components, the choice of the communication protocols is fixed.

6 Conclusion

A major design goal of the presented content-based publish/subscribe framework is to separate out those service sub-tasks which are expected to have large impact on the resource usage. This decomposition strives to give an application designer a simple and flexible means to select protocol components and data attributes according to his needs, and to give him more fine-grained control over the publish/subscribe service through the concept of extension components.

TinyCOPS is the implementation of our component framework aligned with the design philosophy of TinyOS 2.0. The flexibility of *TinyCOPS* to support different sensor node platforms, communication protocols and interaction patterns has been demonstrated experimentally. On the example of a "send-on-delta" service extension component, we have illustrated how the framework can be augmented in order to give the application designers additional control knobs for trading-off different performance objectives. Our experiences with *TinyCOPS* suggest that by careful component decomposition and interface design, it is indeed possible to achieve a good balance between efficient resource usage and reusable software design.

References

1. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2) (2003)
2. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Transactions on Networking (TON) **11**(1) (2003)
3. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1) (2005)
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. of the ninth international conference on Architectural support for programming languages and operating systems (ASPL 2000). (2000)

5. Fayad, M., Schmidt, D.C.: Object-oriented application frameworks. Commun. ACM **40**(10) (1997)
6. Carzaniga, A., Wolf, A.L.: Forwarding in a content-based network. In: Proc. of ACM SIGCOMM 2003, Karlsruhe, Germany (August 2003)
7. Mühl, G., Fiege, L., Buchmann, A.P.: Filter similarities in content-based publish/subscribe systems. In Schmeck, H., Ungerer, T., Wolf, L., eds.: International Conference on Architecture of Computing Systems (ARCS). Volume 2299 of LNCS., Karlsruhe, Germany (apr 2002)
8. Hall, C.P., Carzaniga, A., Rose, J., Wolf, A.L.: A content-based networking protocol for sensor networks. Technical Report CU-CS-979-04, Department of Computer Science, University of Colorado (August 2004)
9. Heidemann, J., Silva, F., Estrin, D.: Matching data dissemination algorithms to application requirements. In: SenSys '03: Proc. of the 1st international conference on Embedded networked sensor systems, New York, NY, USA (2003)
10. Ge, Z., Ji, P., Kurose, J., Towsley, D.: Matchmaker: Signaling for dynamic publish/subscribe applications. In: ICNP '03: Proc. of the 11th IEEE International Conference on Network Protocols, Washington, DC, USA (2003)
11. Levis, P., Gay, D., Handziski, V., J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, A.Wolisz: T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin (November 2005)
12. Handziski, V., Köpke, A., Willig, A., Wolisz, A.: Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor network. In: Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006), Florence, Italy (May 2006)
13. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In: First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI). (2004)
14. Levis, P., Gay, D., Culler, D.: Active sensor networks. Proc. of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005) (May 2005)
15. Fok, C.L., Roman, G.C., Lu, C.: Mobile agent middleware for sensor networks: An application case study. In: Proc. of the 4th Int. Conf. on Information Processing in Sensor Networks (IPSN'05), IEEE (April 2005)
16. Welsh, M.: Exposing resource tradeoffs in region-based communication abstractions for sensor networks. SIGCOMM Comput. Commun. Rev. **34**(1) (2004)
17. Kulik, J., Heinzelman, W., Balakrishnan, H.: Negotiation-based protocols for disseminating information in wireless sensor networks. Wirel. Netw. **8**(2/3) (2002)
18. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to support sensor network applications. IEEE Network **18**(1) (2004)
19. Hill, J., Horton, M., Kling, R., Krishnamurthy, L.: The platforms enabling wireless sensor networks. Commun. ACM **47**(6) (2004)
20. Souto, E., Guimares, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: a publish/subscribe middleware for sensor networks. Personal Ubiquitous Comput. **10**(1) (2005)
21. Wheeler, D.A.: Counting source lines of code (SLOC). `http://www.dwheeler.com/sloc`