Types in Their Prime: Sub-typing of Data in Resource Constrained Environments

Klaas Thoelen(✉), Davy Preuveneers, Sam Michiels, Wouter Joosen, and Danny Hughes

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
klaas.thoelen@cs.kuleuven.be

Abstract. Sub-typing of data improves reuse and allows for reasoning at different levels of abstraction; however, it is seldom applied in resource constrained environments. The key reason behind this is the increase in overhead that is caused by including hierarchical information in data types as compared to a flat list. Where hierarchical data typing is used, it is often represented using verbose textual identifiers or numerical encodings that are suboptimal with regards to space. In this paper, we present an encoding function for hierarchically typed information, based on the properties of prime numbers. It provides a compact representation of types, fast subsumption testing even on resource constrained platforms and support for the evolution of the data type hierarchy. We demonstrate the feasibility of our approach on two representative communication models in constrained environments; a publish/subscribe event bus and a RESTful application protocol. We evaluate the performance of our encoding function and show that it has limited overhead compared to a flat list of data types and that this overhead is outweighed by reduced memory and communication overhead once applied.

Keywords: Sub-typing · Constrained environments · Prime numbers

1 Introduction

Resource constrained networked systems that operate in dynamic environments often require frequent discovery and updating of information flows. Consider an environmental comfort level app on a smart phone that integrates with a smart office environment. For every change of room, the app needs to discover and connect to locally available sensor data sources. In current systems, these data sources are typically typed using a *flat-list* ordering (e.g. temperature, humidity, CO_2), which requires the individual discovery and use of each data source. An alternative practice is to arrange these types into a *hierarchy* that specifies *is-a* relationships (i.e. subsumption) between its constituent elements. This allows for reasoning over groups of elements, called *sub-types*, which are collectively represented by a more abstract element, called a *super-type*. Discovery and data retrieval can consequentially be simplified; e.g. by specifying the more abstract

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2014
I. Stojmenovic et al. (Eds.): MOBIQUITOUS 2013, LNICST 131, pp. 250–261, 2014.
DOI: 10.1007/978-3-319-11569-6_20

sensor data type during discovery instead of the aforementioned specific data types. As such, this results in a reduction of required configuration actions as they can be grouped on a more abstract level. Additionally, it relieves the developer from the full details of the often complex data flows and more clearly reveals the general principles and goals of the application.

The nature of resource constrained environments however poses some requirements as to how sub-typing is provided. First, resource constraints require a compact and efficient solution: (i) storage of sub-typing information should be limited to only locally relevant information, (ii) exchange of that information should occur in a compact manner, and (iii) sub-type testing should require limited computation. Secondly, in order to accomodate changes to applications running on a long-term infrastructure, it is necessary that changes to the hierarchy can be made with as little overhead and disruption as possible. Specifically, adding new types should not cause changes to the encoding of existing types already in use as this requires that type system updates be sent to all nodes in the network, an expensive and highly disruptive process.

The current state-of-the-art in typing of messages in constrained environments either provides no support for sub-typing [1–4] or sub-optimal support for sub-typing based upon textual identifiers [5,6] or simple numerical encoding techniques [7–9]. The verbosity of current approaches holds little advantage in constrained machine-to-machine environments.

Inspired by the encoding proposed in [8], we present an hierarchical type encoding function that is optimized for Class 1 [10] constrained devices (∼10 kB RAM, ∼100 kB Flash). The encoding function exploits the properties of prime numbers and has been specifically re-designed to work within the mentioned constraints. Specifically, we (i) increase the compactness of type encodings, (ii) simplify subsumption testing and (iii) reduce the amount of in-network information needed to perform those tests. We demonstrate the general applicability of sub-typing in both a distributed publish/subscribe based event bus and on top of a RESTful application protocol. Evaluation of the encoding function and both applications show that the encoding has limited overhead compared to a flat list of data types and that this overhead is outweighed by reduced memory and communication overhead once applied.

The remainder of this paper is structured as follows. We describe the encoding function in Sect. 2 and evaluate it in Sect. 3. Two exemplary applications are described and evaluated in Sect. 4. Section 5 discusses related work and we conclude in Sect. 6.

2 Arranging Data in a Hierarchy

The structure of data hierarchies and the encoding function need to be well adapted to the dynamics and constraints of the environments under scope. E.g. in smart offices, the deployed infrastructure often hosts multiple concurrent applications which are subject to changing requirements. This drives evolution of the data hierarchy which therefore needs to provide meaningful abstraction levels

with future extensions of the data set in mind. Driven by both our experience in building sensor network applications [11,12] and restrictions imposed by underlying systems [5], we restrict the data hierarchies we support to *single-inheritance* structures only. As in practice this has shown to be sufficiently expressive, we trade-off its support and expressiveness with increased compaction.

Once a hierarchy is in place, an encoding function is used to represent the hierarchical information to allow efficient subsumption testing. With resource constraints in mind, we highlight the following requirements for such a function:

- **Compact representation.** A compact representation means that a data identifier must contain all necessary sub-typing information in an encoding that uses the least amount of bytes possible.
- **Efficient subsumption testing.** Subsumption testing should require (i) minimal computation and (ii) minimal storage of hierarchical information.
- **Conflict-free incremental encoding.** Incremental encoding allows the addition of new data into the hierarchy at any time without requiring the recomputation of existing data identifiers.

To meet these requirements, we adapt our previous encoding function presented in [8]. By restricting the hierarchies to *single-inheritance* structures, we drastically increase the compaction. This results in reduced memory and communication overhead and simplifies subsumption testing; as our encodings fit standard supported integer types, we can use standard operations and no longer require additional logic for subsumption testing as in our prior work.

2.1 Prime Number Assignment and Encoding

Our encoding function is based on the multiplication of prime numbers. As shown in Fig. 1, a prime number is assigned to each data item or vertex in the tree. Once this is done, the vertex's identifier is set to the multiplication of its own prime with the primes of its ancestors; or consequentially, the multiplication of its own prime with the identifier of its parent. Following from the definition of prime numbers, this causes the identifiers to be divisible (i.e. without remainder) only by the identifiers of their ancestors. The subsumption test that we apply to test whether vertex A subsumes vertex B ($B <: A$) is thus to check whether the modulo operation of their two identifiers is equal to zero. More formally:

$$B <: A \Leftrightarrow id_B \bmod id_A = 0$$

By restricting the hierarchy to a single-inheritance tree, we can introduce a number of optimizations to the original prime number assignment in [8]. Primarily, we reuse prime numbers in the various sub-trees of the hierarchy. This means that more often identifiers are factorizations of the lower-value prime numbers and consequentially have a lower value themselves. Secondly, as there is only a single root, we can assign its *prime* the value of *1*. Although not a prime number by definition, for our purposes this is of no concern whilst again increasing the compactness of the encoding.

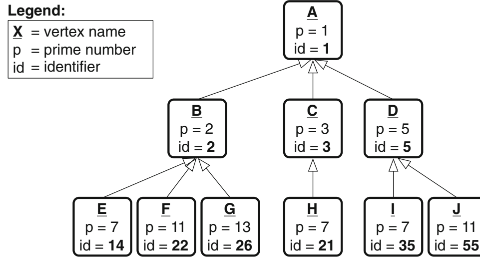


Fig. 1. Hierarchy encoding with reuse of prime numbers.

The prime number assignment algorithm works as follows. Starting at the root of the tree, we assign each vertex a prime number in a top-down and breadth-first manner. The children of each vertex are assigned prime numbers that follow the largest prime number used by that vertex or its siblings. In general, primes are thus reused across disjunct sub-trees, like the ones rooted by B, C and D in Fig. 1. As a result will each subtree contain the only identifiers which are divisible by its root's identifier. In Fig. 1 for instance, only the descendants of node B have identifiers which yield zero for modulo two.

We can prove that our prime number assignment algorithm conserves the subsumption relationships as follows. Assume a set of vertices in a hierarchy $\chi = \{C_1, C_2, ..., C_n\}$. We define $\Gamma(C_i)$ as the union of C_i's assigned prime and the set of primes it inherits from its ancestors. The encoding function that determines a vertex' identifier can then be written as:

$$\gamma(C_i) = \prod_j p_j \text{ with } p_j \in \Gamma(C_i) \tag{1}$$

As proven in [8], the subsumption relation between two vertices C_1 and C_2 can then be defined as:

$$C_1 \text{ subsumes } C_2 \Leftrightarrow \gamma(C_2) \bmod \gamma(C_1) = 0 \tag{2}$$

Now, by definition of subsumption; C_1 can only subsume C_2 if there is a subtree rooted at C_1 which contains C_2 (possibly as root). Given our prime number assignment algorithm, this means that:

$$\Gamma(C_1) \subset \Gamma(C_2) \tag{3}$$

By definition of the encoding function (1), each vertex's identifier is a multiple of each element in the set of primes it inherits. Under single inheritance, reusing primes thus does not influence the correctness of the subsumption test, as it will succeed only in case the set of primes of the more abstract vertex C_1 is a subset of the set of primes of the other vertex C_2, as stated in (3). However, under multiple inheritance and reuse of prime numbers, this equation would not hold. In that case the set of primes of a multiple inheriting vertex can also

be a superset of a vertex that only inherits from one of its ancestors, hereby breaking the correctness of the subsumption test under multiple inheritance. Consequentially, our adapted encoding function can be safely applied, yet to the intended single inheritance hierarchies only.

2.2 Incremental Encoding

Once a hierarchy is encoded, new vertices can be added without the need to fully re-encode the hierarchy. To support evolution, we apply an *append-only* strategy in which new leafs can always be added to the hierarchy, yet existing vertices are never removed from it. The primary reason for this strategy, is that removing deprecated vertices also releases their primes. Reusing those primes for new vertices could lead to false positives during subsumption testing when running applications still refer to the deprecated vertices. Therefore, we abandon the deprecated vertices by no longer using them, but retain them as a placeholder for previously used primes. By exception, a hierarchy can be fully re-encoded to eliminate deprecated vertices. This should however not be done frequently as it requires an expensive update of all software referring to the hierarchy.

When adding a new leaf, some guidelines as to prime number assignment have to be respected to guarantee the subsumption relations. Two cases can be distinguished:

1. **As a sibling to siblings without offspring.** This is the easiest case as we only need to take the siblings into account. The prime to be assigned is the prime that follows the largest prime of the siblings. E.g. in Fig. 1, adding a new leaf K as a child of D will be assigned *13* as prime.
2. **As a sibling to siblings with offspring.** In this case, we need to take the offspring of the siblings into account. E.g. in Fig. 1, a new vertex K as a child of A, and thus a sibling to B, C and D, will be assigned *17* as prime.

Such an addition might thus involve a tree-traversal phase to determine the set of primes used in a certain subtree. However, as this is performed outside of the resource constrained environment, it is of a lesser concern. Adding a leave thus does not require a recomputation of other identifiers and the new hierarchy is fully backwards compatible with the old one.

2.3 Knowledge Distribution of the Hierarchical Structure

During operational use, the full hierarchy and its encoding remain off-line representations that are created, maintained and referenced to in the more resource-rich back-end. Inside the constrained network, each node only requires hierarchical information about the data that it uses and thus does not require full knowledge of the hierarchy.

We furthermore reduced the information required for subsumption testing to only the identifiers. The original encoding function still results in large identifiers

(> 8 bytes), requiring custom support for efficient modulo calculation. Therefore, it uses a more elaborate subsumption test, which uses both the identifiers and primes to first rule out subsumption based on a number of theorems before actually performing the modulo operation. The increased compaction however does allow the use of the standard modulo operation on most platforms and thus eliminates the primes from subsumption testing. Besides reducing processing overhead, this also reduces memory and communication overhead as only identifiers and not primes need to be stored and exchanged.

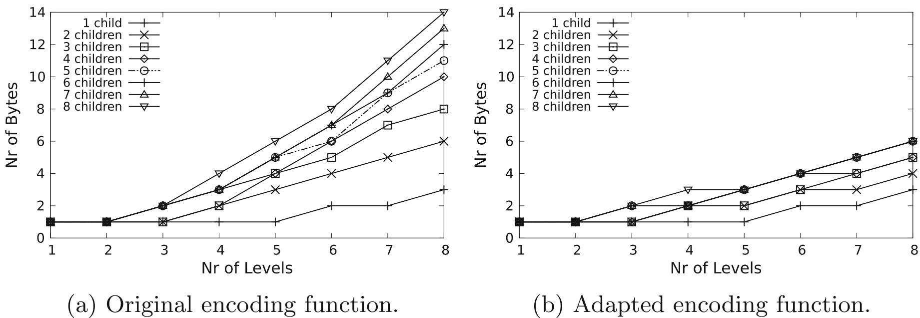
3 Evaluation of the Encoding Function

We evaluate our encoding function by (i) showing its improved compaction, and (ii) the limited processing overhead of the subsumption test.

We evaluate the compaction of our encoding function by comparing the number of bytes it requires to encode identifiers with those required by the original encoding function. We generated artificial hierarchies of 0 up to 8 levels, with each vertex having 0 up to 8 children. Figure 2 shows the number of bytes that were needed to encode each tree's largest identifier. The lower inclination of the graphs in Fig. 2b clearly show that by reusing prime numbers, identifiers can be encoded more compact. While the original encoding requires up to 14 bytes, our version only needs 6 bytes at the most. On average, the identifiers of all tested hierarchies require 5 bytes for the original encoding and only 3 bytes in the adapted version; a substantial benefit in constrained environments with small payload sizes and limited memory.

The increased compaction is confirmed by the encoding of the more realistic hierarchy used in the *smart-office* scenario discussed in Sect. 4.1. Table 1 shows the reduction in primes used, compared to the original encoding. This ultimately results in more compact encoded identifiers. Also the density, i.e. the percentage of integers used as identifiers up to the largest identifier, is higher.

We evaluated the performance of subsumption testing on a Zigduino-r1 sensor node (16 MHz, 128 kB Flash, 16 kB RAM) [13]. Pair-wise subsumption



(a) Original encoding function.

(b) Adapted encoding function.

Fig. 2. The adapted encoding function substantially reduces the number of bytes required to encode identifiers.

Table 1. Comparison of the encoding functions for the smart office hierarchy.

	Nr of vertices	Nr of primes	Largest id	Avg nr of bytes per id	Density
Original	112	112	4481481	3	0,002 %
Adapted	112	24	30914	2	0,36 %

testing among all identifiers in the smart-office hierarchy resulted in an average of 22,3 µs per test. This included deserialization from a variable-length byte-array representation. Pair-wise equality testing, as would be performed using a flat-list ordering, resulted in an average of 4,1 µs per test. This increase in processing time is considered acceptable as it is less than an order of magnitude and is accompanied with the benefits of sub-typing as discussed in the following section.

4 Sub-typing in Exemplary Applications

We applied the proposed encoding function on two representative communication models used in constrained environments. The benefits are distinct in each case. On top of a publish/subscribe event bus, sub-typing results in decreased configuration overhead; while a RESTful application protocol primarily benefits from the compact representation achieved by encoding.

4.1 Sub-typing in a Publish/Subscribe Event Bus

In the *Loosely-coupled Component Infrastructure* (LooCI) [4] sub-typing can be applied to identify events exchanged over a publish/subscribe event bus. LooCI's runtime reconfigurable event bus allows to establish bindings between the interfaces of components. Those bindings are persisted as entries in binding tables which guide the dispatching of published events as shown in Fig. 3. Distributed bindings between components on different nodes consist of two sub-bindings; one on the publisher's side and one on the subscriber's side. As LooCI uses a flat list of event types, each type of event that needs to be exchanged between components requires a separate binding entry.

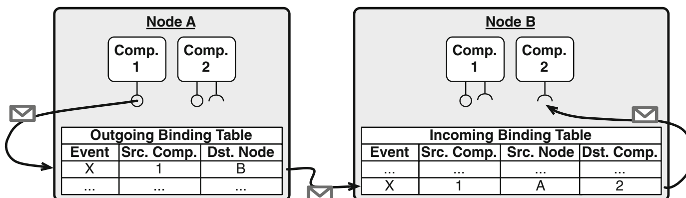


Fig. 3. Conceptual drawing of LooCI's event dispatching based on distributed binding tables.

The benefit of applying event sub-typing in LooCI is in the reduction of binding actions. Using sub-typing, bindings can be created on a more abstract level, allowing groups of similar events to be dispatched based on a single binding. This is important as every binding action results in messages that are disseminated in the network to update the binding tables of the respective nodes; one message for a local binding, two messages in case of a distributed binding.

Evaluation. Specifically, we applied sub-typing to a *smart-office* application. This features an office in which desks, doors and windows are equipped with sensor nodes that monitor the environment and presence of people. Data is collected and pre-processed on sensor nodes and forwarded to a back-end for further processing. PC and smartphone clients subscribe to the back-end for updates about the working comfort and security conditions (air quality, unauthorized access, etc.) in the office. Due to space limitations we refer to [14] for more details about the application's composition and event type hierarchy.

As shown in Table 1, the smart-office event type hierarchy includes 112 event types, with the largest identifier being 30914. On average this requires 2 bytes for encoding, which we precede by a one byte length indicator to support variable-length encoding. The resulting average of 3 bytes per identifier is close to the fixed 2 bytes LooCI uses by default to represent flat-listed even types. This small increase is more than outweighed by the resulting decrease in binding actions that are needed to compose the smart-office application. Using default LooCI, 55 binding commands were needed, which by applying sub-typing is reduced to only 35. This is a 36 % reduction in messages disseminated into the network to configure event dispatching. This reduction is substantial and subject to a multiplication effect due to frequent reconfiguration in dynamic networks.

The described results are of course application specific and bigger applications might lead to larger event hierarchies and thus larger identifiers. On the other hand, we expect larger applications to benefit more from sub-typing. In Sect. 3 we already showed that the accompanying overhead of large hierarchies remains practical. While the reduction of binding commands depends on the hierarchy's structure, overall, sub-typing will never increase the number of binding commands required and when applied sensibly can reduce the amount of commands extensively.

4.2 Sub-typing on Top of a RESTful Application Protocol

In RESTful protocols, the proposed encoding function can be used to more compactly identify hierarchically organized sources of information, called *resources*. Resources are typically identified using a Uniform Resource Identifier (URI) that contains a hierarchically structured node-local path to the resource, formatted as a human-readable string. We propose to more compactly encode these resource paths using our prime-based encoding.

Designed for constrained environments, the Constrained Application Protocol (CoAP) [5] aims at realizing a REST architecture for the most constrained

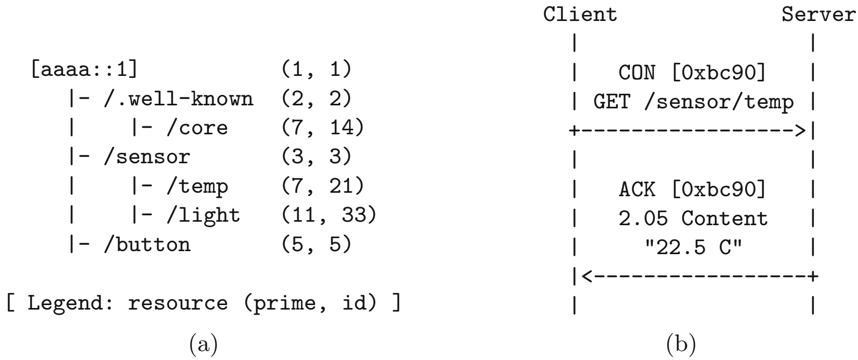


Fig. 4. A hierarchy of resources on a CoAP server with address `aaaa::1` (a), and a GET request for the `/sensor/temp` resource with matching response (b).

devices in machine-to-machine (M2M) applications. Figure 4a presents an example set of resources deployed on a sensor node. To interact with these, COAP supports the four traditional REST methods: GET, POST, PUT, DELETE. E.g. a GET method with URI `coap://[aaaa::1]/sensor/temp` can be issued to request temperature data from a CoAP server (see Fig. 4b).

We can make two important observations with regards to resource paths:

1. The string representations of resource paths form excessive identifiers that cause considerable memory and communication overhead. While practically beneficial, human-readable strings are not vitally important in a M2M environment where direct human interaction is not a primary concern.
2. Hierarchically organizing CoAP resources brings structure and thus allows *super*-resources to delegate requests to their *sub*-resources. The hierarchical information contained in the resource path is thus of practical importance.

Both observations motivate the application of our encoding function on CoAP resource paths. While retaining hierarchical information, resource paths can be represented much more compactly. This reduces memory overhead since each resource no longer involves storage of a lengthy human-readable resource path. Communication overhead is primarily reduced during intensive interactions such as CoAP resource discovery. This is executed using block-wise transfer of the description of all primary resources on a node. In case the description does not fit in a single block, subsequent blocks are requested individually. By reducing the representation of resource paths, the number of blocks can be reduced, thus eliminating not only their transmission, but also those of their respective requests.

Once resource paths are encoded, as in Fig. 4a, the numerical identifiers can be used in two ways. In the first one, a *string*-formatted representation of the identifier replaces the original human-readable path. E.g. the URI `coap://[aaaa::1]/sensor/temp` becomes `coap://[aaaa::1]/21`. The benefit of this variant is that no adaptations to CoAP are required. In the second variant,

Table 2. Comparison of the original and encoded representation sizes of the CoAP resource paths defined in the IPSO Application Framework.

Path representation	Total nr of bytes	Avg nr of bytes per path
Original	499	9,24
Encoded (string)	158	2,93
Encoded (uint)	84	1,53

more compact *uint*-formatted identifiers are used. As this is not compatible with string-formatted URI Paths, a new Option is needed in the CoAP specification that indicates the use of uint-formatted resource path identifiers.

Evaluation. We encoded the set of resource paths specified by the IPSO Application Framework [6], which defines the interfaces of 47 REST resources. Table 2 compares the original human-readable resource paths with our two encoded variants. We evaluate the total number of bytes required to represent all resources and the average number of bytes per resource this results in. While the resource paths in [6] are already fairly compact and only partially human-readable, encoding them still results in an extensive reduction in representation size; 68 % for the string-formatted representation and 83 % for the uint variant.

The added benefit of this reduction becomes apparent during CoAP resource discovery. E.g. human-readable versions of all resource paths consume 49 bytes in a discovery reply by the server in Fig. 4a. In contrast, using the averages in Table 2, only 16 bytes and 12 bytes are needed when using string- and uint-formatted encodings, respectively. This optimization can lead to considerable reduction in communication in dynamic resource constrained environments.

5 Related Work

We discuss related work in terms of alternative encoding functions and the application of sub-typing in publish/subscribe systems and RESTful protocols for constrained environments.

Sub-typing and related encoding functions have been studied in many areas like programming languages, database management, knowledge representation, policy enforcement, etc. For a detailed discussion, we refer to a survey of existing techniques in our prior work [15]. The solutions presented there support multi-inheritance hierarchies and often lack (efficient) support for incremental encoding. Based on our experience, we traded-off multi-inheritance support for increased compactness, as discussed in Sect. 2.

The publish/subscribe messaging pattern is often used in distributed computing in both constrained and traditional environments. While in traditional environments sub-typing of messages is a regular feature [16–18], it is typically implemented by reuse of language-provided inheritance (e.g. Java), ill-suited for constrained environments. Most related work concerning constrained environments does not discuss message typing [2,3,19]; suggesting a flat list ordering.

DSWare [20] does apply the notion of event hierarchies, but in a different manner; higher-level compound events (e.g. explosion event) are inferred from the detections of a set of lower-level atomic events (e.g. sensor observations). The event hierarchy is thus purely functional and does not represent typing of events. In our prior work [21], we used sub-typing to reduce the energy consumption of event routing based on semantic event information. This solution uses our original encoding function, which is significantly optimized in this paper.

Like the IPSO Alliance [6], the Open Mobile Alliance (OMA) [9] has specified templates of CoAP resources and their organization in a hierarchy. While IPSO uses compact human-readable resource paths, OMA makes use of numerical resource paths that identify resources registered with the OMA Naming Authority (OMNA). While the latter shares our application of non-human-readable resource identifiers, both solutions use less compact resource identifiers than can be realized with our encoding function.

6 Conclusion

As resource constrained environments continue to integrate with the Web and become first-class citizens in mobile and ubiquitous applications, they will be subject to higher degrees of interaction. To facilitate discovery and use of their data sources, sub-typing can be applied to allow reasoning on higher levels of abstraction. Yet, in contrast to the resource constraints at hand, sub-typing implies overhead to represent the hierarchical relations between types of data.

In this paper, we presented an encoding function for hierarchically typed data that is designed with resource constraints in mind. This encoding function features low overhead both in representation of hierarchical information and subsumption testing. However, this overhead is outweighed by the benefits that are achieved once sub-typing is applied. While on the one hand sub-typing results in reduced reconfiguration overhead in LooCI's publish/subscribe event bus. On the other hand, our encoding allows for more compact representations than the human-readable resource identifiers commonly used in CoAP's RESTful interactions. We can thus conclude that when realized efficiently, the added benefits of sub-typing can be extremely useful in more resource constrained environments.

Acknowledgement. This research is partially supported by the Research Fund, KU Leuven and iMinds (a research institute founded by the Flemish government). The research is conducted in the context of the COMACOD and ADDIS projects.

References

1. Buonadonna, P., Hill, J., Culler, D.: Active message communication for tiny networked sensors. In: Proceedings of the IEEE Conference Infocom 2001 (2001)
2. Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: a publish/subscribe middleware for sensor networks. Pers. Ubiquitous Comput. **10**(1), 37–44 (2005)

3. Hauer, J.-H., Handziski, V., Köpke, A., Willig, A., Wolisz, A.: A component framework for content-based publish/subscribe in sensor networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 369–385. Springer, Heidelberg (2008)
4. Hughes, D., Thoelen, K., Maerien, J., Matthys, N., del Cid Garcia, P.J., Horré, W., Huygens, C., Michiels, S., Joosen, W.: Looci: the loosely-coupled component infrastructure. In: 11th IEEE International Symposium on Network Computing and Applications (NCA), pp. 236–243, August 2012
5. Shelby, Z., Hartke, K., Bormann, C.: Constrained Application Protocol (CoAP). http://tools.ietf.org/html/draft-ietf-core-coap-18
6. Shelby, Z., Chauvenet, C.: The IPSO Application Framework. http://www.ipso-alliance.org/wp-content/media/draft-ipso-app-framework-04.pdf
7. Kovacevic, A., Ansari, J., Mähönen, P.: Nanosd: a flexible service discovery protocol for dynamic and heterogeneous wireless sensor networks, pp. 14–19. IEEE Computer Society, Los Alamitos (2010)
8. Preuveneers, D., Berbers, Y.: Encoding semantic awareness in resource-constrained devices. IEEE Intell. Syst. **23**(2), 26–33 (2008)
9. OMA LWM2M. http://technical.openmobilealliance.org/Technical/release_program/lightweightM2M_v1_0.aspx
10. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained Node Networks. http://tools.ietf.org/html/draft-ietf-lwig-terminology-05
11. Hughes, D., Thoelen, K., Horré, W., Matthys, N., del Cid Garcia, P.J., Michiels, S., Huygens, C., Joosen, W., Ueyama, J.: Building wireless sensor network applications with looci. Int. J. Mobile Comput. Multimedia Commun. **2**(4), 38–64 (2010)
12. Thoelen, K., Hughes, D., Matthys, N., Fang, L., Dobson, S., Qiang, Y., Bai, W., Man, K.L., Guan, S.-U., Preuveneers, D., Michiels, S., Huygens, C., Joosen, W.: A reconfigurable component model with semantic type system for dynamic wsn applications. J. Internet Serv. Appl. **3**(3), 277–290 (2012)
13. Zigduino-r1. http://logos-electro.com/zigduino-r1/
14. http://people.cs.kuleuven.be/~klaas.thoelen/mob2013
15. Preuveneers, D., Berbers, Y.: Prime numbers considered useful: ontology encoding for efficient subsumption testing, Department of Computer Science, K.U.Leuven, Leuven, Belgium, CW Reports CW464, October 2006
16. Esper. http://esper.codehaus.org/index.html
17. Java Messaging Service. http://www.oracle.com/technetwork/java/index-jsp-142945.html
18. Corba Notification Service. http://www.omg.org/spec/
19. Russello, G., Mostarda, L., Dulay, N.: A policy-based publish/subscribe middleware for sense-and-react applications. J. Syst. Softw. **84**(4), 638–654 (2011)
20. Li, S., Son, S.H., Stankovic, J.A.: Event detection services using data service middleware in distributed sensor networks. In: Zhao, F., Guibas, L.J. (eds.) IPSN 2003. LNCS, vol. 2634, pp. 502–517. Springer, Heidelberg (2003)
21. Preuveneers, D., Berbers, Y.: μc-semps: energy-efficient semantic publish/subscribe for battery-powered systems. In: Proceedings of the 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, pp. 1–12, December 2010