# A comparison of static and dynamic component models for Wireless Sensor Networks[*]

Barry Porter, Utz Roedig, François Taïani, and Geoff Coulson

Computing Department, Lancaster University
{barry.porter / roedig / francois.taiani / geoff}@comp.lancs.ac.uk

**Abstract.** In this paper we provide a detailed discussion and evaluation of the theoretical and practical differences between static and dynamic component models as the foundations of programming wireless sensor nodes. As the static benchmark we examine the nesC component model underpinning TinyOS; and as the dynamic benchmark we examine the OpenCom component model underpinning the Lorien operating system. Both models are well established in their respective domains and have at least 2nd generation implementations available. We identify 4 key mechanisms required by the dynamic approach beyond those needed by the static approach, and using the TelosB implementations of both models we demonstrate the performance differences involved in the support of each of these mechanisms. We conclude that while the static approach has inevitably better performance, the overhead of the dynamic approach is sufficiently low that it is a promising foundation in support of future WSN research in dynamic and adaptive systems.

## 1 Introduction

Component-based programming provides a powerful programming abstraction by enforcing interface-based interaction between system modules and thus avoiding any hidden interaction via direct variable access or inheritance models. This in turn offers the potential for truly black-box integration of system modules to ease the composition and configurability of complex systems.

Both static and dynamic component models provide the developer with an identical programming model. The static variant synthesises systems at compile-time and 'flattens' the component model into a monolithic, static image for deployment which no longer bears any component-model features. This carries the benefit of whole-system compiler optimisation. The dynamic variant instead maintains component separation and boundaries to carry the component model over to runtime for synthesis. This carries the overhead of representing the component model at runtime, and only permits per-component compiler optimisation, to the benefit of allowing later modifications to the system architecture in a very lightweight, online manner.

[*] This work was partially supported by the ICT Programme of the European Union under contract number ICT-2008-224460 (WISEBED)

In light of rising recent research interest in dynamic and adaptive systems for wireless sensor networks beyond the more classical static systems, we use this opportunity to examine the theoretical and practical differences involved in the two approaches. Notable indicators of this trend towards dynamic and adaptive systems can presently be seen in two main domains of WSN research.

The first is *protocol* adaptivity research, in which a single protocol is designed to be adaptive to its environment or other context. Examples are traffic- or energy-tunable / self-tuning MAC protocols [1, 2], adaptive sampling rate protocols to reduce unnecessary data [3], or even a general framework for managing the parametric adaptation of multiple such protocols and services [4]. Some adaptive protocols in this class additionally employ selective sub-component activation and deactivation over time to model different protocol modalities, such as a moving rendezvous point [5].

The second adaptivity research domain takes the next level up and explores *compositional* adaptivity where the system selects at runtime from a pool of available components depending on observed system state or context. This latter domain of adaptivity tends to emerge when stability in the collection of useful service/protocol variants is reached, and is also evidenced in systems which can execute a number of optional protocols or services depending on their current requirements. Examples of research leveraging both kinds of compositional adaptation are adaptive home monitoring applications with diverse services activated and modified according to context [6], periodic or context-based activation of system maintenance protocols in large-scale outdoor deployments [7], and selecting between different routing systems and radio interfaces based on perceived environmental risk conditions [8].

While a static model sufficiently supports simple parameter-based *protocol* adaptation or tuning, a dynamic model represents a good fit both for adaptive protocols using different modalities by dynamically selecting sub-components, and for both kinds of *compositional* adaptation. We believe that both research topics are likely to continue to increase as the field matures and more generalised infrastructural sensor networks pervade the environment, particularly in large networks of heterogeneous cooperating objects which adapt and organise towards their current task as the network's roles and environmental conditions evolve over time. In this paper we use the static and dynamic component model variants to explore the implications of this in terms of general static vs dynamic programming approaches.

As our static benchmark we use the nesC component model [9] which supports the TinyOS [10] WSN operating system, and as our dynamic counterpoint we use the OpenCom component model [11] which supports the Lorien [12] WSN OS. Both are mature models with at least 2nd generation implementations. Both models offer a near-identical programming model to the developer, save that nesC is entirely static such that systems are composed from components at compile-time after which system composition cannot change, whereas OpenCom is entirely dynamic such that systems are composed from components at run-time and can evolve the system composition arbitrarily thereafter.

Note that we do not here consider system update by means of downloading alternative code; while both models have approaches to this, and it is an important aspect to expore in future, we focus here solely on the theoretical and performance differences of the programming models in themselves to establish a baseline comparison of the quiescent states of the static and dynamic models.

In the remainder of this paper we first in Section 2 present a theoretical comparison of the two models, and in Section 3 we provide an evaluation of the two models in practice based on the nesC and OpenCom implementations. In Section 4 we discuss related work, and we conclude in Section 5.

## 2 Theoretical comparison

A component model [13] is defined in general terms as a programming paradigm supporting distinct modules which have *provided* and *required* interfaces and must interact solely through these. For a component to function correctly its required interfaces must be connected to the type-compatible provided interfaces of other components.

As both static and dynamic component models provide this same fundamental programming model to the developer, the differences between the two are observable entirely in the mechanisms that the dynamic model adds to the static model in support of runtime compositional dynamics. In other words, the static component model is a strict subset of the dynamic component model in terms of required support mechanisms.

We define in a generalised way the mechanisms needed by a dynamic component model beyond those needed by a static model as follows, based on the most common features of dynamic component models (e.g. [14, 15, 11]):

- **M1:** A runtime with meta-data describing the current system composition
- **M2:** A boot-time configuration mechanism
- **M3:** Per-component instantiation support
- **M4:** A mechanism supporting interchangeable component connections

We now discuss each of these in general terms, including different possible implementations, and note the specific OpenCom implementation for WSNs as used for evaluation purposes in the following section.

### 2.1 Runtime and meta-data

To support the representation of the system's current architecture, and later changes to that architecture, a meta-representation is required with a runtime to maintain that representation and permit (safe) changes to it. This representation holds the component instances, the currently advertised provided and required interfaces of those individual instances, and the current interconnections of those interfaces between component instances. In a static component model this data is encoded statically in configuration scripts and is discarded after compilation as it will never change. In a dynamic model this data is retained at runtime to enable later modifications. The data can be encoded in varying levels of detail depending on the provided meta-data API and system configuration approach. The number of operations provided by the runtime is implementation-specific, as is the amount of runtime validation and verification provided to prevent dangerous operations.

*OpenCom* In terms of meta-data the implementation models the system architecture as the available component types in the system image; the instances of those types; the provided/required interfaces of those instances; and the interconnections of those interfaces. All of this data is stored in RAM for fast access and modification. The runtime implementation provides complete validation of the correctness of operations to maintain system integrity, such as disallowing the destruction of a component which still has incoming connections to its provided interfaces.

### 2.2 Boot-time configuration

The system architecture of a dynamic component model is typically initialised to an empty set, and the programmer must employ an approach to configure the desired components and connections when the system boots. This may range from fully automated system configuration approaches based on XML file parsing to manual hard-coded configuration based on simple rules.

*OpenCom* This aspect of the implementation is by nature pluggable; for evaluation purposes here we employ a hard-coded configuration / reconfiguration system based on simple rules.

### 2.3 Instantiation support

A useful feature of component models is the ability to employ multiple instances of certain components connected in to different parts of the overall component system to re-use common functionality. A simple example would be to connect one instance of a data processing component to each sensory input component to perform duplicate suppression. In a static component model instantiation can be provided at compile-time but the cardinality post-compilation is fixed. A dynamic component model permits components to be instantiated and destroyed at runtime as desired, requiring each component to have a form of constructor and destructor to initialise its instance-specific state.

*OpenCom* The implementation here employs a standard construct and destruct method in which components register / destroy their initial provided/required interfaces and any additional state required.

### 2.4 Interchangeable connection support

Finally, a dynamic component model naturally requires the ability to alter the connections between components at runtime, for example when swapping one component for another. In a static model inter-component connections are fixed at compile-time and can therefore simply be direct-addressed.

*OpenCom* The implementation uses function pointers whose addresses are known to the meta-data API; when a connection or disconnection is made these addresses are modified. When a component invokes an operation on one of its required interfaces, this operation takes place via the function pointer to be invoked on the currently-connected provided interface.

These four mechanisms represent everything that is required to implement dynamic system support beyond what is required in a static component model, able to instantiate and destroy components at runtime and change the way that they are interconnected.

## 3 Performance in practice

In this section we evaluate the real-world implementation of a static component model and a dynamic one, presenting the results in terms of the four mechanisms discussed in Section 2. Our evaluation is based on a system which performs typical WSN functionality, involving independent tasks which: i) periodically read a sensor value, ii) perform some processing on the data from this sensor, and iii) send the results via the radio towards a remote sink. The evaluation is performed on the TelosB [16] platform, a sensor platform comprising an $8MHz$ MSP430 CPU, $48KB$ program memory (ROM) and $10KB$ RAM.

We implement the example system both under the static component model of nesC in TinyOS (v2.1), which we refer to as $S$, and under the dynamic component model of OpenCom in Lorien (v1.6), which we refer to as $D$. $S$ is subject to full-system optimisation as a monolithic program during synthesis/compilation, and contains none of the dynamic model mechanisms of Section 2; $D$ on the other hand is subject only to per-component optimisation as components maintain their strong separation post-synthesis, and of course contains all of the mechanisms of Section 2. The component graph for $D$ is shown in Figure 1.

We note at this point that Lorien was not developed as a direct dynamic translation of TinyOS, but rather was developed independently with its own specific aims [12]. Nevertheless the two operating systems are similar enough that evaluating them in the way we do here focuses to as large an extent as possible only on their respective static and dynamic component model foundations and the resulting differences in resource usage. Both are open-source projects and our experimental setup is easy to replicate.

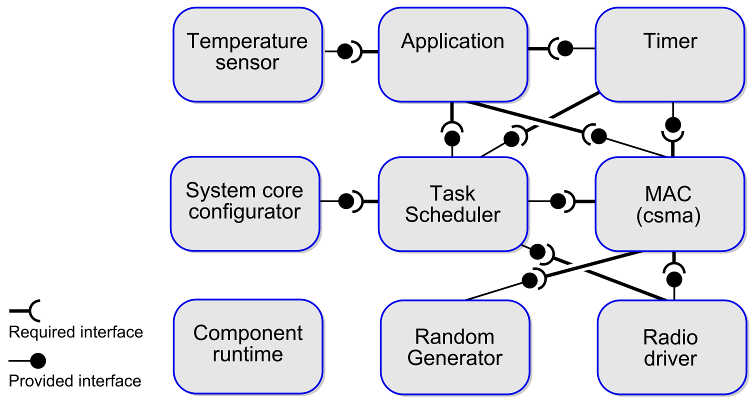

**Fig. 1.** Component graph (with connections simplified) of our evaluation system $D$

### 3.1 Program memory (ROM)

The complete system image for $S$ takes 17616 bytes of program memory; while the image for $D$ takes 27434 bytes, an increase of 55%. Other than slight anticipated variances in component implementations, the additional program memory in $D$ is caused by mechanisms M1, M2 and M3 of the underlying dynamic component model, shown in Table 1. Here we see 5.5KB of fixed-cost for mechanisms M1 and M2; this cost is the same in the majority of Lorien systems and represents 31% of the ROM overhead over $S$. The other 24% comes from the system-specific cost for $D$ of 3KB of instantiation support (mechanism M3) in $D$'s components, with an average cost of 350 bytes per component in $D$.

| Element | ROM overhead in $D$ |
|---|---|
| M1 | 4KB |
| M2 | 1.5KB |
| M3 | 3KB |

**Table 1.** Breakdown of ROM overheads of $D$

### 3.2 RAM

The total RAM usage in $S$ is 912 bytes representing the component-specific state (such as radio buffers etc) of all used components; by comparison $D$ uses 2749 bytes in total for both component-specific state and dynamic component model meta-data (M1), an increase of 200% in RAM. The breakdown of meta-data costs is shown in Table 2, listing per-instance costs of each kind of meta-data – from which general RAM overheads can be estimated for different systems – and the totals specifically for $D$. 1842 bytes in total are used by $D$ for component meta-data in the dynamic component model, representing the dynamic model's overhead, with the remaining 907 bytes used for component-specific state. The latter statistic's slight difference to $S$ is simply due to a different default memory allocation strategy of Lorien's components to those of TinyOS. We also note that all of the RAM footprint of $S$ is static memory, whereas that of $D$ is generally dynamic, shrinking and growing as needed. Note that a 'receptacle' in Table 2 is a required interface, and an 'interface' a provided interface.

| Element | Per-item generic cost | Total generic cost in $D$ (+ specialisations) |
|---|---|---|
| Component instance | 44 bytes | 396 bytes (+ 88 bytes) |
| Interface | 20 bytes | 460 bytes (+ 160 bytes) |
| Receptacle | 24 bytes | 432 bytes (N/A) |
| Connection | 18 bytes | 306 bytes (N/A) |

**Table 2.** Dynamic model RAM overheads (M1)

### 3.3 CPU usage

In terms of processing cycles, the overhead of $D$ is manifested in its boot-time configuration program (M2), and in its use of indirection to support component interconnection (M4).

In terms of system startup time, in comparison with $S$, $D$ takes an additional $23ms$ to boot. This is a one-off cost and not very significant.

OpenCom's indirect interconnection implementation uses simple function pointers for inter-component calls. The overhead of such calls over a plain function call, as used in $S$, is $170ns$ (taken as an average over many calls in order to be measurable). Given that most WSN operations (e.g. sensing, processing and sending data) take on the order of milliseconds, we do not believe that this is a significant overhead either.

### 3.4 Energy

The energy consumption of a sensor node is related to the time its components are active; idle components can be put into a power efficient sleep state to conserve energy. As already discussed, $D$ uses function pointers to perform inter-component calls which add an overhead of $170ns$ to each call compared to a fixed monolithic system. Depending on how often inter-component calls are performed within an application, a different total processing overhead will be accumulated (where the total processing overhead is defined as the sum of all inter-component call overheads over a given execution period).

To estimate typical overheads in terms of energy consumption in our evaluation system, we stimulate both $S$ and $D$ with external events according to a realistic usage scenario. We assume a sensor network that forms a binary tree topology, and that sensor nodes perform a sensing event every $4sec$. We further assume that the sensing component requires $1ms$ to acquire data, the application component requires $5ms$ to process data, and the network components require $1ms$ to forward the data towards the remote sink. These values represent typical processing durations as observed in real deployments [17]. Depending on the number of hops $h$ a node is away from the sink, and the depth $(H)$ of the binary tree, a node must perform not only its sensing operation but also forwarding of data from its child nodes. For evaluation we measure the total processing overhead in dependence of $n = H - h$.

A sample of the results are shown in Table 3, demonstrating an energy profile difference of 0.49% on average between $D$ and $S$ over a $180sec$ sampling period. $S$ generally outperforms $D$, though not always; our results rather show that function pointer overhead of $D$ is not the dominant source of performance difference between the two approaches, and that the actual implementations of elements like the network stack have a far higher impact on performance.

| Node position $n$ | Idle Time $(D)$ | Idle Time $(S)$ | Difference of $D$ |
|---|---|---|---|
| 2 | 98.75 | 99.37 | (+)0.62% |
| 3 | 98.50 | 99.03 | (+)0.53% |
| 4 | 98.30 | 98.76 | (+)0.46% |
| 5 | 97.37 | 97.11 | (−)0.26% |
| 6 | 95.19 | 94.59 | (−)0.59% |

**Table 3.** Total processing overheads (M2 + M4)

## 3.5 Summary

In this section we have examined the quantitative performance differences between two mature static and dynamic component models. While we have examined only one specific system we have also where possible provided generalised figures from which much of the overall performance picture can be estimated for other systems which have different numbers of components and interconnections in the dynamic model.

From our specific example, the resource overhead of the dynamic model is 1837 bytes extra RAM of the 10KB available on the TelosB platform used for evaluation, an increase of 200% over $S$; 10KB extra ROM of the 48KB available, an increase of 55% over $S$ (of which 31% is a one-time fixed cost); and an average difference of 0.49% in energy usage, where the top performer between $D$ and $S$ actually varies demonstrating that in this case implementation is more important than the system-building technology. We believe that overall these statistics represent a good case for the gain in system flexibility towards adaptive and dynamic systems offered by a dynamic component model.

## 4 Related work

While we are not aware of any other work on direct static-dynamic component model comparisons as the foundations of WSN systems, there are number of operating system and middleware endeavours based on dynamic system support both in and beyond the WSN field.

SOS [18] and LiteOS [19] propose a static kernel atop which dynamically loaded modules can be downloaded. While neither support a component model as defined here, and do not therefore provide the same kind of comparison, the results reported in terms of energy overhead caused by indirect addressing of inter-module access are similar to ours.

Dynamic or semi-dynamic component models have also been proposed to operate on top of otherwise static operating systems as a middleware layer [20, 21]; again they lack the baseline comparison with a static model we provide here, and additionally do not provide the full-system comparison we have explored where the respective component models permeate the entire system rather than just a part of it. Again however results relating to additional energy consumption caused by the use of a dynamic model are comparable to our findings.

Finally, notable efforts outside the WSN field include THINK [22] and Pebble [23]; both employ dynamic component models in support of workstation-class operating systems. While positive results are again reported, the focus on higher-resourced platforms makes them difficult to compare specifically on WSN-class systems, and specific static-dynamic component model comparisons are not made.

## 5 Conclusion

In this paper we have presented a detailed discussion and evaluation of the theoretical and practical differences between static and dynamic component models as the foundations of programming wireless sensor nodes.

We have defined a set of 4 necessary supporting mechanisms to enable dynamic component models over and above what is needed to support static component models, and we have evaluated a nesC-based static system and an OpenCom-based dynamic system on common WSN hardware, finding that the resource overheads of this support are modest.

With increasing interest in dynamic and adaptive systems research in the WSN community, and particularly compositional dynamics and adaptation, we believe that our findings point to a useful and practical enabling technology in support of this research into the future – notably in more generalised, infrastructural networks of cooperating objects which adapt to new roles and conditions over time, but also broadly in any networks of cooperating devices which need to adapt as the fabric of the network or its functionality spectrum evolves.

The support of such systems with software that is fundamentally dynamic throughout, with the strong component separation and architectural safety guarantees of a dynamic component model, is entirely feasible on current-generation hardware across all domains of resource consumption.

## References

1. Borms, J., Steenhaut, K., Lemmens, B.: Low-overhead dynamic multi-channel mac for wireless sensor networks. In: EWSN. (2010) 81–96
2. Hurni, P., Braun, T.: Maxmac: A maximally traffic-adaptive mac protocol for wireless sensor networks. In: EWSN. (2010) 289–305
3. Chatterjea, S., Havinga, P.: An adaptive and autonomous sensor sampling frequency control scheme for energy-efficient data acquisition in wireless sensor networks. In: DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems, Berlin, Heidelberg, Springer-Verlag (2008) 60–78
4. Lorincz, K., Chen, B.r., Waterman, J., Werner-Allen, G., Welsh, M.: Resource aware programming in the pixie os. In: SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems, New York, NY, USA, ACM (2008) 211–224
5. Joung, Y.J., Huang, S.H.: Tug-of-war: An adaptive and cost-optimal data storage and query mechanism in wireless sensor networks. In: DCOSS. (2008) 237–251
6. Taherkordi, A., Rouvoy, R., Le-Trung, Q., Eliassen, F.: A self-adaptive context processing framework for wireless sensor networks. In: MidSens '08: Proceedings of the 3rd international workshop on Middleware for sensor networks, New York, NY, USA, ACM (2008) 7–12
7. Cao, Q., Stankovic, J.A.: An in-field-maintenance framework for wireless sensor networks. In: DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems, Berlin, Heidelberg, Springer-Verlag (2008) 457–468
8. Hughes, D., Greenwood, P., Blair, G., Coulson, G., Grace, P., Pappenberger, F., Smith, P., Beven, K.: An experiment with reflective middleware to support grid-based flood monitoring. Concurrency and Computation: Practice and Experience **20**(11) (2008) 1303–1316
9. Gay, D., Levis, P., von Behren, R.R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. SIGPLAN Notices **38**(5) (2003) 1–11
10. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGOPS Operating Systems Review **34**(5) (2000) 93–104
11. Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Transactions on Computer Systems **26**(1) (February 2008) 1–42
12. Porter, B., Coulson, G.: Lorien: A pure dynamic component-based operating system for wireless sensor networks. In: MidSens '09: Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks, New York, NY, USA, ACM (2009) 7–12
13. Szyperski, C.: Component Software: Beyond Object Oriented Programming. Addison-Wesley (2003)
14. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain (June 2002)
15. Dowling, J., Cahill, V.: The k-component architecture meta-model for self-adaptive software. In: REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Springer-Verlag (2001) 81–88

16. Moteiv: Tmote Sky Datasheet http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf. (2006)
17. Duffy, C., Roedig, U., Herbert, J., Sreenan, C.J.: A Comprehensive Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems. Journal of Networks (JNW) **3**(3) (2008) ISSN 1796-2056.
18. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services, Seattle, Washington, USA (June 2005) 163–176
19. Cao, Q., Abdelzaher, T., Stankovic, J., He, T.: The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In: IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks, IEEE Computer Society (2008) 233–244
20. Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society (2007) 69–78
21. Mottola, L., Picco, G.P., Sheikh, A.A.: FiGaRo: Fine-grained software reconfiguration for wireless sensor networks. In: Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN08), Springer Verlag (January 2008) 286–304
22. Fassino, J.P., Stefani, J.B., Lawall, J.L., Muller, G.: Think: A software framework for component-based operating system kernels. In: ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2002) 73–86
23. Gabber, E., Small, C., Bruno, J., Brustoloni, J., Silberschatz, A.: The pebble component-based operating system. In: ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (1999) 20–20