

PERVASIVE GRIDS: INTEGRATING SENSOR NETWORKS INTO THE FIXED GRID

Geoff Coulson

*Computing Department, InfoLab 21, South Drive,
Lancaster University, Lancaster LA1 4WA, England*

geoff@comp.lancs.ac.uk

Abstract Sensor networking has emerged as a key research challenge on the road to fully pervasive distributed systems. Much current work on sensor networking, however, focuses on sensor networks in relative isolation, and de-emphasises their integration into the wider networked environment. We are focusing on such integration in the context of so-called ‘pervasive grids’ in which sensor networks are integrated with computationally-intensive grid-based computations in the fixed network. In this paper, we provide a brief overview of our middleware approach for pervasive grid environments and discuss work in progress on application scenarios (i.e. detecting floods in river valleys), and on basic technology (i.e. deploying our component based middleware approach on scarcely-resourced sensor nodes).

Keywords: Sensor networks, middleware

1. Introduction

Sensor networking has emerged as a key research challenge on the road to fully pervasive distributed systems. Much current work on sensor networking, however, focuses on sensor networks in relative isolation, and de-emphasises their integration into the wider networked environment. Typically, it assumes that a dumb ‘gateway’ node will provide a proxy-like interface between a ‘simple’ sensor network and a ‘complex’ hosting network (e.g. the fixed internet).

However, we believe that a *closer integration* between the sensor and internet domains will become increasingly important as sensor networks become more widely deployed. Such integration is required at both the network layer and at the distributed processing layer. At the *network layer*, integration is needed to accommodate *dynamic gateways*. Relying on a single, fixed, gateway node introduces an obvious bottleneck and

single point of failure. It should therefore be possible to dynamically change the numbers and configurations of gateways as the sensor network grows, or as traffic patterns between the sensor network and the fixed network change, or as gateways crash, etc. Network layer integration is also useful to support *seamless addressing*, to allow sensor devices to be individually addressed from arbitrary locations. This is *not* to say that it is always necessary or desirable to address sensor nodes as individual entities. However, it is clearly a requirement in some sensor network environments (e.g. where it is necessary to take a reading from a specific location; or to send a command to a specific actuator).

Integration at the *distributed processing layer* is primarily needed to support *multiple interaction paradigms*. Depending on the application and the underlying infrastructure, there are many possible ways in which the various nodes in the sensor and fixed network domains might interact. Typical examples are database-inspired data collection, publish-subscribe, fan-in, multicast, streaming, request-reply, and flooding. Ideally, these various interaction paradigms will run seamlessly across the heterogeneous sensor/fixed network, and it will be possible to dynamically instantiate them depending on application needs, and to have new nodes (in both the sensor and fixed domains) join and leave as required. Distributed processing layer integration is also needed to support *local computation*. Although many current sensor nodes are ‘dumb’, in some cases they are capable of supporting a significant amount of computation (see below). In such circumstances, it becomes feasible to perform ‘local’ processing on sets of sensor nodes—especially where there is a favourable trade-off between this and paying a high cost to ship a large amount of data to the fixed network for remote processing. Furthermore, the optimal location for such processing—local or remote or mixed—may well vary as environmental conditions change (e.g. as batteries run down, sensor nodes die, the size of the computation changes, etc.). Finally, integration at the distributed processing level in terms of *programming model commonality* is also highly desirable. The challenge here is as far as possible to provide the same programming APIs on primitive sensor devices as is found in traditional PC-based programming environments. We focus especially on this issue in this paper.

In our current research we are exploring a range of such integration issues. We are particularly interested in what we call *pervasive grid scenarios* in which sensor networks are integrated with computationally-intensive grid-based computations in the fixed network. An example of such a scenario is given in section 3. Such scenarios typically involve close coordination and integration between the sensor network and the ‘fixed grid’ to achieve, for example, computational steering of a fixed-grid

simulation by current sensor data, or dynamic tuning of sensor behaviour to best accommodate the changing information needs of a simulation.

The rest of this paper is organised as follows. Section 2 provides a brief overview of our approach to the provision of middleware for pervasive grids. Next, section 3 discusses in more detail an example of what we call a ‘pervasive grid’. Then, section 4 discusses work-in-progress on applying our middleware approach on resource constrained sensor devices. Finally, we survey related work in section 5, and offer our conclusions in section 6.

2. Middleware Approach

2.1 Software Component Model

Our approach to middleware provision generally is to construct middleware-level software in terms of a low-level language-independent *software component model* that is supported by a minimal runtime API that enables dynamic loading, instantiation and binding of fine-grained components. Crucially, this component model, called OpenCOM [1], is a *local* model—distribution aspects are assumed to be built on top of and in terms of the component model (i.e. by encapsulating the necessary communications protocols etc. inside components). The Gridkit system discussed below is an example of such a distribution system. By adopting this approach, we essentially abandon the traditional notion of a ‘middleware layer’ in favour of a world-view in which each node is provided with a suitable set of components that work together to offer the specific middleware-related (and indeed application-related) functionality required by that particular node. The rationale for our approach is that middleware-level software must increasingly provide strong support for flexible *configuration* (e.g. for different applications and classes of device—including primitive sensor devices) and *reconfiguration* (e.g. to support dynamic adaptation in the face of environmental change). We believe that a fine-grained, cross-layer component approach is a highly-promising way of achieving this.

The component model itself is complemented by the notion of *component frameworks*. These are runtime entities that provide an intermediate structuring abstraction midway between individual components and complete systems. Component frameworks support dynamically ‘plugged in’ components and embody integrity rules that constrain attempted plug-in operations. Component frameworks are independently and optionally deployable (using the runtime API), and their precise configuration is tailored to the needs of the target system being implemented. They may be low-level or high-level (e.g. addressing different

levels of functionality such as network, middleware or application functionality); and large or small; and ‘local’ or distributed (i.e. like OpenCOM components themselves or like the distributed Gridkit framework discussed below).

More details on OpenCOM and the component framework approach are available in the literature [1].

2.2 Gridkit

Our basic approach to achieving sensor net/ fixed net integration is i) to employ an extensible range of *overlay networks* that run seamlessly over the two (or more) networks and mask their underlying heterogeneity; and ii) to provide on top of this a flexible distributed processing layer that is overlay-aware and can drive cross-layer adaptation. The approach enables us to employ fit-for-purpose overlays that address in an deployment/application-specific way the network layer issues identified above (i.e. automatic gateway configuration and addressing) and provide suitable support for an extensible range of interaction paradigms at the distributed processing layer.

Gridkit [2] is an OpenCOM-based middleware framework that is an embodiment of this approach and attempts to provide a generic support environment for pervasive grid environments. Crucially, by taking advantage of OpenCOM’s flexible configuration properties, Gridkit can be profiled to run on both primitive sensor nodes and fixed network nodes. Gridkit consists of two component frameworks: the Overlay CF and the Interaction CF. The Overlay CF supports pluggable overlay networks, while the Interaction CF then supports ‘pluggable’ interaction paradigms. Over the last year or so we have populated the two Gridkit CFs with a range of plug-ins. In the Overlay CF, for example, we have implemented Chord [4], Scribe [5], Application Level Multicast (i.e. TBCP [3]), Gossip and probabilistic multicast. In the Interaction CF we have implemented publish-subscribe, streaming and groups. More detail on Gridkit can be found in [2].

3. Application Scenarios

We are currently exploring a number of pervasive grid scenarios at the application level. In this section we focus specifically on a project in which we are instrumenting a river valley in the Yorkshire Dales with a sensor network designed to provide early warning of flash floods and to monitor the extent of floods that do occur. The deployment consists of some 10’s of sensors placed at intervals along the river. The basic sensor node platform is based on small XScale-equipped boards called

Gumstix [6]. These are the size of a pack of chewing gum and can be equipped with a range of on-board sensors such as temperature and water pressure sensors. We also employ sensor nodes equipped with video cameras which we use to monitor river flow rates (see below). The nodes are powered using a combination of batteries and solar panels. For communication, they employ both 802.11 and Bluetooth networks. A small number of nodes additionally have the capability to act as gateways to the fixed network; GPRS is used to realise the sensor net to fixed net link. All the nodes support an OpenCOM runtime and run the Gridkit middleware. As the Gumstix devices run Linux and have a reasonable amount of memory it was straightforward to port OpenCOM/Gridkit to them.

In operation, the sensors gather and collate information and ship it back to computationally-intensive simulation software running in the fixed grid. The publish-subscribe interaction paradigm is employed to provide a consistent programming platform across the sensor and fixed domains. This works closely with a supporting overlay that organises the gathering of sensor data and also accommodates ‘intermediate’ nodes that perform collation and combination of the sensor data.

The flooding scenario is of especial interest in its scope for *local computation*. This takes two forms: First, nodes run simple flood models on site to alert local stakeholders of pending floods via SMS and local signage. Second, and more interestingly, the sensor network executes a parallelised image-processing algorithm to measure the flow rate of the river. The data that drives this algorithm comes from the above-mentioned nodes that are equipped with video cameras. These cameras point at the river surface and pass the resultant images to this parallel algorithm. The latter locates identifiable pieces of floating debris on the river surface and, by comparing the times at which a given piece of debris appears on images produced by successive cameras, computes the flow rate of the river. Interestingly, performing this computation locally, within the sensor network itself, *is the only viable option*—because of its size, it would not be viable to ship the raw data to the fixed grid over the GPRS link.

The scenario is also interesting in terms of its scope for *adaptation*. In particular, the overlay network is capable of adapting in terms of the physical network it uses and in terms of the choice and number of gateways it uses for communication with the fixed network. For example, in the quiescent condition, when the river is low, the overlay uses the Bluetooth network for inter-node communication. Thanks to its low power requirement, this network is the best choice where nodes are close together and the required data rate is low. However, on the

first indication of the river rising, the sample rate employed by the sensors increases to the point where the resultant data rate is too much for Bluetooth. At this point, the distributed processing layer (which notices the bottleneck) asks the supporting overlay to adapt. As a result, the overlay switches from Bluetooth to 802.11, which supports a higher data rate, although at a correspondingly higher rate of power consumption. In addition, a switch to 802.11 can occur when sensors die (either from battery failure or physical damage) and the network therefore becomes partitioned; here, the greater range of 802.11 can overcome breaks in the upstream-to-downstream chain of Bluetooth links.

More detail on this application scenario can be found in the literature [7]. We also have a number of other projects that are investigating the pervasive grid concept—e.g. the Open Overlays project which is exploring collaborative fire-fighting in forests [8]; and the RUNES project which is focusing on safety and rescue in road tunnels [9].

4. Technology Issues

As mentioned, we believe that an important element of sensor-fixed integration is the provision of a common programming model across the full range of devices involved in a pervasive grid. To this end, we discuss in this section work in progress on applying the OpenCOM approach on resource-constrained sensor devices such as Telos Motes [10]).

4.1 OpenCOM on Motes

In the short term, we are looking to implement OpenCOM’s runtime API (load, instantiate, bind etc.) on top of an existing sensor node OS. Our starting points are a C/Unix version of the OpenCOM runtime, and a sensor node OS called Contiki [11]. The main issue here is to replace the runtime’s original Unix-specific component representation and loading services (based on Linux shared objects and *dlopen()*) with equivalent Contiki functionality. This is relatively straightforward as Contiki (unlike other popular sensor node OSs such as TinyOS [14]) already provides the ability to load code dynamically.

The resulting OpenCOM implementation will enhance sensor node operating systems like Contiki with the following value added features:

- Familiar abstractions. The programmer is presented with a consistent component-based programming model that is identical to that available on large computers in the fixed grid. In addition, standard services and protocols that support dynamic deployment and reconfiguration in OpenCOM can naturally be added to motes as and when this is deemed useful (and memory constraints permit).

- Fine-grained loading. Although it supports dynamic loading, Contiki only allows one loaded image at a time. The OpenCOM implementation will add *incremental* loading so that new functionality can be added without disturbing ongoing computations.
- Multiple component instantiation. OpenCOM adds support for multiple *instances* of software components ('services' in Contiki terminology). This is highly useful for dynamic reconfiguration scenarios, and is likely to become increasingly central as more sophisticated software gets deployed on mote class devices.
- Dependency tracking. Keeping track at runtime of multiple dynamically loaded/ unloaded software elements ('services') is currently challenging for Contiki programmers. The OpenCOM notion of 'receptacles' (i.e. required interfaces) eases this task by making such dependencies explicit.

This all adds a modest memory footprint overhead to Contiki. Our current estimate is around 5 Kbyte (not including threads; see below), although the work is still in progress.

In the longer term, we plan to eliminate the dependency on Contiki and to create an implementation of OpenCOM that runs *natively* on the bare mote hardware. This should result in improvements in both footprint and performance. To achieve this, we will progressively discard Contiki code that is not directly needed to support the OpenCOM API, and wrap general OS services such as device drivers and protocol stacks as OpenCOM components.

4.2 Achieving Lightweight Concurrency

Our original C/Unix version of OpenCOM includes an (optional) non pre-emptive user-level threads package. However, it is infeasible to port this 'traditional' user-thread implementation directly to a mote-class device with limited memory and no virtual memory support. This is because of the need to provide stack space for each thread, which takes a considerable amount of memory and also leads to significant internal and external memory fragmentation which wastes memory still further.¹

¹Internal fragmentation arises from the need to ensure that each stack has sufficient spare space that it will not overflow; external fragmentation arises from the fact that the multiple stack areas are scattered at random throughout the heap depending on when the owning thread was created. Note that a traditional unithreaded system in a non-virtual memory environment incurs neither internal or external fragmentation: its single stack grows from one end of the memory while the heap grows from the other end.

We are therefore pursuing an alternative concurrency design that mitigates these problems. We will focus on this issue in some detail in this section as concurrency support is an area in which current mote-based programming environments are arguably somewhat lacking. Contiki, for example, employs dedicated concurrency mechanisms which, while lightweight, are (arguably) too constraining and unfamiliar to be acceptable to programmers; see section 5 for more detail on these.

A naive ‘straw-man’ approach to the avoidance of these problems would be to use a *single* (contiguously-allocated) stack area to underpin *all* threads. Only the currently-running thread would use the stack, and non-contiguous pieces of heap memory would be dynamically allocated to store the individual stack frames of threads that are not currently running. On each context switch, the stack frames of the currently-running thread would be saved to heap memory, and the frames of the to-be-scheduled thread would be copied from this memory to the single contiguous stack area. This design would neatly avoid both internal and external fragmentation *but* at an enormous cost: the whole stack must be copied every time a thread is blocked and restored.

However, we believe that the basic idea of stack copying does indeed form the basis of a viable approach. To achieve viability, we impose the following (minimal) constraints on OpenCOM programmers:

- 1 The C functions that directly realise an OpenCOM component’s externally-visible entry points may only be called by *other* component entry point functions, via an OpenCOM binding component², and *not* by other arbitrary C functions.
- 2 Operations that may result in a thread context switch (e.g. blocking on a semaphore, creating a new thread, or calling *yield()*) may only be issued from within an externally-visible entry point function (as defined above).

We believe that these constraints are tolerable to OpenCOM programmers, who may still freely create threads, block on semaphores, use recursion, etc. In fact, OpenCOM programmers already hold an implicit mental model of component entry point functions being somehow ‘primary’, and the C functions that are called from them as ‘secondary’ or auxiliary. Furthermore, the design *does not fail* if the constraints are violated—violations simply lead to a (possibly drastic) drop in performance (see below).

²‘Binding components’ encapsulate the binding between a receptacle and an interface [1]. They provide a level of indirection that is crucial to the design outlined here. Incidentally, we can also exploit this indirection to insert code that periodically de-fragments heap memory.

We now outline the design of our proposed scheme. It essentially operates like the ‘straw man’ proposal above, but assumes the above constraints. It also adds the following mechanism: whenever a call is issued on a component’s externally-visible entry point, before pushing a new stack frame on the stack, the thread runtime, executed from within the intervening binding component, saves the current stack frame into a (per-thread) linked list of extant ‘entry point’ stack frames; and then ‘pops’ the stack pointer to point to below the current stack frame (i.e. to the bottom of the stack area). Then, when such a call returns, the runtime (again, from within the binding component) copies the previously-saved stack frame from the linked list back again to the bottom of the stack area.

Essentially, this scheme solves the memory fragmentation problem along the lines of the ‘straw man’ approach while only incurring an overhead of $n+m$ stack frame copies per context switch, where n is the number of stack frames currently on the stack and m is the number to be restored for the to-be-scheduled thread. Crucially, in cases where the constraints have been met, $n=m=1$ so that only *two* stack frames need to be copied per context switch. This is because entry point functions, which according to the constraints are the only functions in which context switches should take place, are always called on a stack with only a single stack frame (i.e. the rest of the thread’s context is saved on the above-mentioned linked list in heap storage).

We can summarise the benefits of the scheme as follows:

- The internal and external memory fragmentation problems mentioned above have gone away. Memory can be organised in a similar way to a traditional unithreaded architecture: the (single) stack area can grow from one end of memory, and the heap can grow from the other end.
- Even in the worst case, the scheme does not demand any more memory than a traditional multi-stack implementation. This is because the memory for any given stack frame is *either* on the contiguously-allocated stack, *or* in an area of heap memory, but never in both at the same time. The scheme does not even require memory for linked list pointers, as the stack frame’s framepointer location can be used for this.
- In actuality, the scheme is likely to use *significantly less* memory overall than a traditional implementation. This is because, assuming the constraints are being honoured, blocked threads will only ever need to store stack frames for the OpenCOM entry point calls

they currently have outstanding; any ordinary C function calls they make will have returned before the thread blocked.

On the negative side of the trade-off, the overheads are as follows:

- On each call of a component entry point there is an overhead of saving and restoring a number of stack frames (just *one* stack frame in each case if constraint 1 has been honoured).
- On each context switch there is the overhead of saving a number of stack frames and restoring others (just *one* in each case if constraint 2 has been honoured).

Given the typical granularity of components, we would expect relatively few cross-component calls, and thus the first overhead would be expected to be incurred relatively infrequently. Similarly, we would expect relatively few context switches and relatively small stack frames as compared to standard systems. Implementation of the scheme is now underway and we will soon be in a position to carry out a detailed empirical evaluation of its performance and resource-usage properties.

5. Related Work

Most environmental sensing projects reported in the literature ([17] is a good example) have taken a ‘dumb gateway’ approach (see section 1.1) to linking the sensor domain and the fixed network. The wider middleware community has considered in a generic way the need for integration in the face of network heterogeneity. However, this work is largely geared to specific cases of integration, and/or has been tightly coupled to particular interaction paradigms. For example, the ALICE [13] project has investigated how CORBA bridges can operate between wireless and fixed infrastructures, although they focus on a single interaction paradigm (i.e. RPC) in a single deployment case (two specific types of network). In a similar vein, the publish-subscribe interaction paradigm has been extended (e.g. see [15]) to support events being sent and received between endpoints in both wireless and fixed networks. However, as far as we are aware no middleware supports a general solution to the problem of selectively supporting a range of interaction types over a range of network types.

In the specific area of lightweight concurrency on sensor nodes, most OS-based solutions employ an event-based concurrency model rather than true threads. While this is simple and has a low resource overhead (no need for stacks), it is notoriously difficult to write concurrent application code as non-blocking event handlers. This observation has led some

researchers to look for a compromise between events and true threads. One notable attempt has been made by the designers of the Contiki OS in the form of so-called *proto-threads* [12]. These have the benefit that (unlike event handlers) it is possible to block within a proto-thread. However, compared to our design, the Contiki design is extremely constraining for the programmer: a proto-thread may only run within a single function; that function does not have its automatic variables preserved across blocking; and the programmer may not freely use C case statements within proto-threads. On the other hand, the memory overhead of proto-threads is much lower than that of our design. In terms of static overhead, the additional complexity of our design presumably requires more memory for the thread code itself; and in terms of per-thread overhead, while the authors of the Contiki design report an overhead of only 2 bytes per proto-thread (which presumably corresponds to the space needed to store the program counter), our approach requires 10's of bytes for the struct used to represent each thread, plus an elastic overhead in terms of stack frame storage (which depends on the depth of stacked calls made by each thread when it blocks). Nevertheless, we believe that the higher level of abstraction provided by our design is potentially worth this overhead.

6. Conclusions

This paper has argued for closer integration between the sensor network domain and the fixed network, and has discussed some of our work in progress in this area. The general approach advocated is to address the need for integration at both the (overlay-based) networking layer and the distributed processing layer. We have also discussed an application scenario that clearly requires the integrated approach (i.e. in terms of local/remote computation and network adaptivity). True integration also involves the provision of a common programming model across both sensor nodes and large machines on the fixed network. We have discussed this issue in terms of providing a common OpenCOM-based API across all device classes, and have focused especially on the provision of a thread service that is, on the one hand, familiar to programmers, and, on the other hand, promises to be lightweight and well-adapted to resource-constrained nodes with no virtual memory.

7. Acknowledgements

Thanks to Gordon Blair and Francois Taiani who are equal partners in this work. Thanks are also due to the following researchers: Paul Grace, Phil Greenwood, Danny Hughes, Barry Porter, Rajiv Ramdhany,

Siva Thirunavukkarasu and Nirmal Weerasinghe. We also gratefully acknowledge our colleagues on the EU-funded RUNES project who have who have adopted the OpenCOM approach and contributed significantly to its development and application. Finally, we wish to acknowledge our funders—the EPSRC, the NWDA and the EC.

References

- [1] Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J., “A Component Model for Building Systems Software”, Proc. IASTED Software Engineering and Applications (SEA04), Cambridge, MA, USA, Nov 2004.
- [2] Grace, P., Coulson, G., Blair, G.S., Porter, B., “Deep Middleware for the Divergent Grid”, Proc. IFIP/ACM/USENIX Middleware 2005, France, Nov 2005.
- [3] Mathy, L., Canonico, R., D. Hutchison, D., “An Overlay Tree Building Control Protocol”, Proc. 3rd International COST264 Workshop on Networked Group Communication, pages 7687, London, UK, Nov 2001.
- [4] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., Balakarishnan, H., “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, Proc. ACM SIGCOMM, San Diego, 2001.
- [5] Castro, M., Druschel, P., Kermarrec, A-M., Rowstron, A., “SCRIBE: A Large-Scale and Decentralised Application-Level Multicast Infrastructure”, IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications), 2002.
- [6] Waysmall Computers, “Gumstix Embedded Computing Platform Specifications”, <http://gumstix.com/spexboards.html>, 2006.
- [7] Hughes, D., Greenwood, P., Coulson, G., Blair, G., Pappenberger, F., Smith, P., Beven, K., “GridStix: Supporting Flood Prediction using Embedded Hardware and Next Generation Grid Middleware”, Proc. 4th IEEE International Workshop on Mobile Distributed Computing (MDC 2006), co-located with WoWMoM, 2006.
- [8] Grace, P., Coulson, G., Blair, G., Porter, P., “Addressing Network Heterogeneity in Pervasive Application Environments”, Proc. InterSense (First International Conference on Integrated Internet Ad hoc and Sensor Networks, Nice, May 2006.
- [9] Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S., “The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems”, Proc. 16th Annual International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05), Berlin, Germany, 2005.
- [10] Telos Motes, www.moteiv.com/pr/2006-02-08-tmoteinvent.php, Feb 2006.
- [11] Dunkels, A., Grnvall, B., Voigt, T., “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”, Proc. First IEEE Workshop on Embedded Networked Sensors 2004, Tampa, Florida, USA, Nov 2004.
- [12] Dunkels, A., Schmidt, O., Voigt, T., “Using Protothreads for Sensor Node Programming”, Proc. REALWSN 2005 Workshop on Real-World Wireless Sensor Networks, Stockholm, Sweden, June 2005.
- [13] Haahr, M., Cunningham R., V. Cahill, V., “Towards a Generic Architecture for Mobile Object-Oriented Applications”, Proc. 2000 IEEE Workshop on Service Portability and Virtual Customer Environments, San Francisco, Dec 2000.

- [14] Gay, D., Levis, P., Culler, D., “Software Design Patterns for TinyOS”, Proc. ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’05), Chicago, June 2005.
- [15] Mhl, G., Fiege, L., Grtner, F., Buchmann, A., “Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems”, Proc. 10th IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunications Systems, pp.167-176, Texas, USA, Oct 2002.
- [16] Sivaharan, T., Blair, G.S., Coulson, G., “GREEN: A Configurable and Re-Configurable Publish-Subscribe Middleware for Pervasive Computing”, Proc. Distributed Objects and Applications 2005 (DOA05), Agia Napa, Cyprus, 2005.
- [17] Tateson, J., Roadknight, C., Gonzalez, A., Khan, T., Fitz, S., Henning, I., Boyd, N., Vincent, C., Marshall, I., “Real World Issues in Deploying a Wireless Sensor Network”, Proc. Workshop on Real-World Wireless Sensor Networks REAL-WSN’05, Stockholm, Sweden, June 2005.