Dynamic Reconfiguration in Sensor Middleware

Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, Danny Hughes
Computing Department
Lancaster University
Lancaster, UK
(gracep, geoff, gordon, porterbf, hughesdr)@comp.lancs.ac.uk

ABSTRACT

Middleware solutions for sensor networks have so far mainly focused on communication abstractions, ad-hoc message routing protocols, and power conservation techniques. We argue that customisation and dynamic reconfiguration of sensor network middleware are additional important dimensions to consider. This paper describes a sensor middleware that can be customised to suit different sensor application types, and provides a reflective approach for co-ordinated network-wide dynamic reconfiguration of sensor behaviour. To evaluate our approach we illustrate customisation and dynamic reconfiguration of the Gridkit sensor middleware in a flood-monitoring scenario.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns (Reflection).

General Terms

Management, Design, Experimentation.

Keywords

Sensor middleware, reflection, dynamic reconfiguration.

1. INTRODUCTION

Wireless sensor networks are used in a wide range of application domains including, for example, environmental monitoring and disaster management. The role of sensor middleware is to support the application developer, and shield her from: i) the complexity of developing applications on heterogeneous low-level hardware such as MICA Motes, Gumstix, or bespoke sensor technology, and ii) routing messages across heterogeneous ad-hoc networks e.g. Bluetooth, 802.11b, Zigbee, and others. Hence, the initial sensor middleware solutions have concentrated on common communication abstractions (e.g. event subscription [11], database-style [2, 13], or tuple-spaces [14]) over a variety of ad-hoc message routing strategies [3,9]. Some proposals have additionally considered the management of resources within sensor networks e.g. power consumption [15].

However, existing sensor middleware proposals have yet to approach the problems caused by the following two important

characteristics of sensor applications:

- *Environmental diversity;* sensor networks in different application fields require different hardware, different networks, different styles of communication. Hence, a fixed middleware solution is not applicable to all applications; rather a customisable middleware is potentially better equipped for use across many application types.
- *Dynamic change;* sensor networks are deployed in environments that are inherently subject to change, and sensor network middleware must therefore be inherently adaptive. For example, sensor network middleware used in disaster scenarios will typically need to react to increasingly hazardous conditions—e.g. flooding in a monitored location may cause the network to perform poorly or fail.

We therefore suggest that sensor middleware must consider these additional dimensions. In this paper, we examine three aspects of our Gridkit middleware [7], which address these dimensions:

- *Pluggable communication abstractions.* Using these, the application developer is able to select the appropriate communication abstraction for their application. For example, temperature monitoring might be best done using event-subscription, whereas a streaming-based abstraction may be better suited to video monitoring.
- *Pluggable routing protocols, and overlay networks.* In Gridkit, sensor networks are created as virtual network topologies (i.e. overlay networks), which can be selected to support different application requirements. For example, an overlay that conserves power may be most appropriate for one application type, whereas an overlay that focuses on robustness might be better in a different setting. Different overlays optimise for different characteristics such as these primarily by differing in terms of their topology and in how messages are routed between nodes.
- *Co-ordinated Dynamic Reconfiguration.* Sensors need to adapt their behaviour to cope with changing environmental conditions—e.g. sensors could increase the frequency at which they send messages, or change their routing protocol to cope with increasing failure rates in the sensor network. To support such cases, co-ordinated adaptation of the per-host middleware across all nodes in the sensor network is required—e.g. the routing protocol on every node must be updated. Gridkit introduces the concept of *distributed frameworks* for this purpose; these consist of a set of sensor nodes and a set of distributed reflective meta-protocols that allow i) the inspection of the current network-wide middleware configuration, and ii) the coordinated reconfiguration of software elements across nodes.

To evaluate our approach to sensor middleware we document in this paper how Gridkit is customizable to support a real-world case study involving the management of flooding in a river valley

in the North-West of England. We concentrate in particular on how our middleware is able to perform system wide adaptations of the overlay networks to react to changing conditions such as increasing water flow, and sensor failure due to flooding.

The remainder of the paper is structured as follows. Section 2 examines the Gridkit framework and its support for pluggable communication abstractions and overlay networks. Section 3 then describes the distributed frameworks and reflective meta-protocols used for dynamic reconfiguration. Section 4 evaluates the Gridkit middleware in its support of the case study. Section 5 investigates areas of related work, and section 6 draws conclusions and proposes important areas of future research.

2. THE GRIDKIT SENSOR MIDDLEWARE

Gridkit [7] is a generalized middleware framework that can be specialized to operate in diverse application domains; e.g. Grid computing, pervasive computing, and mobile computing. In this paper, we focus on tailoring Gridkit to the domain of sensor middleware. Fundamentally, Gridkit is built in terms of a component model called OpenCOM v2 [4]. This employs a minimal runtime that supports the loading and binding of lightweight software components at run-time. OpenCOM is used in the construction of all higher-level middleware.

In Gridkit, middleware is built upon a core distributed framework known as the *overlays framework*. This hosts, in a set of distributed overlay framework instances, a set of per-overlay plug-in components, each of which embodies *i*) a *control* element that cooperates with its peers on other hosts to build and maintain some virtual network topology, and *ii*) a *forwarding* element that routes messages over its virtual topology.
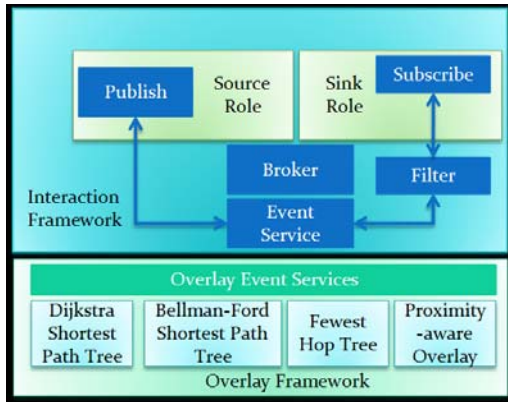

Figure 1: Example Gridkit Sensor Middleware

Above the overlays framework is a set of further customisable frameworks that provide functionality in various orthogonal areas, and can optionally be included or not included on different devices; these frameworks are discussed in more detail in [7]. Using this framework approach, we have recently developed the sensor middleware that is illustrated in figure 1. In this, the *interaction framework* contains a customizable event service. A node can be just a publisher, just a subscriber, or, optionally, it can act as a broker in the event service. Underpinning this framework is a set of overlays for the distribution of events towards sinks. This is also customizable from i) a centralized spanning tree using Dijkstra's shortest path

algorithm, ii) Bellman-Ford's decentralized shortest path tree overlay, iii) a fewest hop tree, or iv) an overlay that is aware of proximity. Hence, it is possible to customize both the interaction framework level and the overlay level to suit individual application requirements.

3. DYNAMIC RECONFIGURATION

There are two important dimensions in the dynamic reconfiguration of sensor networks: *local* and *distributed*. Local adaptation is the dynamic reconfiguration of software elements on an individual node; whereas distributed adaptation is the adaptation of the behaviour across a sensor network. Therefore, a distributed adaptation consists of a series of local adaptations. In Gridkit, dynamic reconfiguration is based around software architecture elements known as *component frameworks*. There are two styles of component framework supporting reconfiguration, as discussed in the following two sub-sections.

3.1 Local Component Frameworks

The local component framework model (illustrated in figure 2) is based on the concept of composite components as proposed in the OpenORB project [1]. Each framework has a reflective meta-interface (*ICFMetaInterface* in figure 2) that enables inspection and dynamic adaptation of the local 'architecture' of the composite component in terms of its local components and connections. Additionally, the integrity of each framework is maintained in the face of dynamic change, using developer specified architectural rules plugged into the component framework (through the *IAccept* interface).
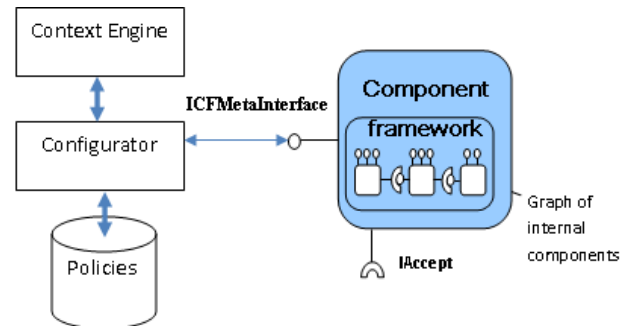

Figure 2: Local Component Frameworks in Gridkit

The second aspect of the local component framework model is the use of the configurator pattern [12] as illustrated in figure 2. A configurator is assigned to each framework instance, and acts as a unit of autonomy for making decisions about when and how to change the framework. Each configurator maintains a set of local policies for its framework. It is connected with the Gridkit context engine [7] to receive relevant environmental events; and communicates with its host framework through the meta-interface. This separation of the configurator allows different configurators and policies to be used for different framework types; for example, a protocol framework may employ a policy that restricts plug-in components to be composed only in 'stacks'. Typical configurator policies in Gridkit use the Event-Condition-Action pattern. When an event is detected, it triggers the corresponding action, which is a reconfiguration script of component inserts, deletes, disconnects, connects, or replaces.

3.2 Distributed Component Frameworks

3.2.1 Overview

A distributed component framework is a set of local component framework instances of the same type located across a set of co-ordinated devices, typically providing co-ordinated middleware functionality—e.g. an overlay network, a shared middleware communication channel (group binding), etc. The design of the distributed framework model follows the same basic themes as for local frameworks—i.e. the use of reflection to support inspection and adaptation of software, and configurators to enforce autonomic actions. There are a number of important elements concerned with the creation of distributed frameworks, which are now discussed in turn.

3.2.2 Meta-Object Protocol & Reification Strategies

Each distributed framework (figure 3) maintains a basic meta-object protocol (MOP) that reifies the information about the contents of the framework in terms of the node members only. The operations allow the insertion and deletion of local framework elements into/from a given distributed framework, and the inspection of information about the set of individual participants. This interface is made available from the core middleware (IDCFMetaInterface in figure 3).
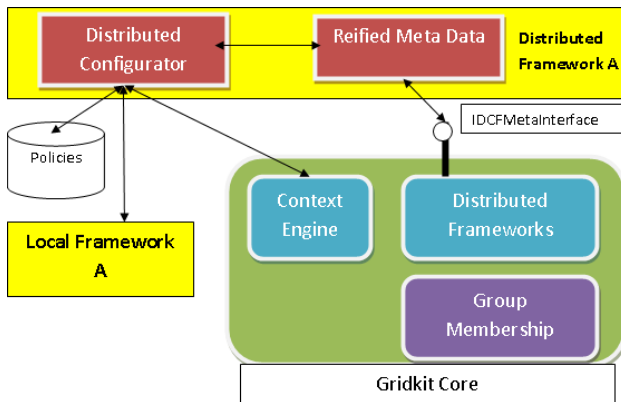

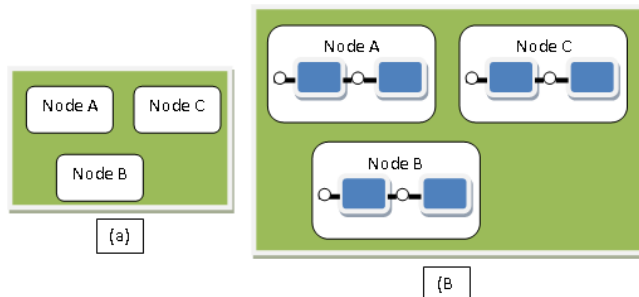**Figure 3: Distributed Frameworks (per host components)**

Figure 4: Alternative reification of meta-data

Meta-information can be reified to various locations; it does not necessarily need to be stored at every node (especially in resource-constrained sensor networks). Therefore, depending on the precise requirements, the information could be stored at a central node, or at a subset of the nodes, or at all of them. This depends on the reification strategy employed by the meta-protocol; i.e. whether the host contains the "Reified Meta Data" component (see figure 3) that collects the published information. For when there is more than one instance of this component in the framework, consistency protocols must be utilized to ensure the same view is maintained across nodes.

For the implementation of this meta-object protocol we use a lightweight group membership service as the base mechanism for distributing meta-data; this data then builds the view of the system wide architecture. This is customizable in its implementation: typically different group membership overlays will suit different sensor models—e.g. a sensor network with high node mobility will require a different group membership overlay from a more static network topology. So far, we have used the scalable membership protocol SCAMP [6] to maintain meta-data between members of a distributed framework.

The basic meta-information maintained for a each framework is shown in figure 4(a); this is essentially just the local framework members of the distributed framework. However, this is enough to find out all subsequent information about the framework architecture; the meta-data contains references to the local frameworks' reflective interfaces (*ICFMetaInterface*), which can then be used to reflect all the component information in the distributed framework. However, a 'push'-based reification model with richer meta-data may also be employed. This will utilize more resources, but may potentially reduce the network traffic and time to make reconfiguration decisions, as the data can be stored locally with the configurators. The richer meta-data is shown in figure 4(b); this adds individual component and connection information in the same format as provided in local frameworks. To distribute the data in a 'push' fashion the local meta-data is gossiped to all other members using the lightweight group membership service. The selected storage points use the information (component, connections, etc.) to build a distributed view of the network wide framework.

3.2.3 Configurators

Distributed configurators (as seen in figure 3) follow the same pattern as for local frameworks. They receive events about changing environmental conditions, select policies and then perform distributed reconfigurations. However, individual frameworks may have more than one configurator (e.g. there could be one on every node). Therefore, consensus protocols are used to ensure that all members of the framework agree on the action to perform. Our evaluation has so far focused on single configurators (see section 4), however, we are also investigating the introduction of selectable and replaceable consensus algorithms into our distributed frameworks.

3.2.4 Quiescence

For safe dynamic reconfiguration it is important to ensure that updates complete atomically and do not impact the integrity of the network. There are two parts to this: i) making the framework safe to adapt, i.e. placing it in a quiescent state, and ii) ensuring that the reconfiguration is complete and correct.

In our current implementation, local framework instances maintain a readers/writers lock to place it into a quiescent state: any standard interface call or meta-inspect is a reader, any meta-write is a writer. Hence, no reconfiguration can take place locally while a thread is executing in the framework. Currently our distributed frameworks use this capability; each local framework is placed into a quiescent state through a command propagated via

the meta-group service. Once locally quiescent a notification is returned to the configurator. Upon the condition that all members are in a quiescent state then the reconfiguration continues. After reconfiguration, like with local frameworks (*IAccept* plug-in), the update can be checked through inspection of the meta-data to validate the integrity of component updates across multiple nodes. An invalid reconfiguration can be detected and repaired.

The disadvantage of the above approach is that it may be too resource intensive, and may not scale suitably for large numbers of hosts. Therefore, we are investigating replaceable decentralised strategies for safely updating components. Like the reification and configuration approaches, we advocate that the quiescence strategy should also be selectable for the needs of an application.

3.2.5 Constraints

The distributed component framework model must be inherently more flexible than the local model, as there are many more constraints that disallow a single fixed model being utilised. These are described as follows:

Resources. All nodes may not be equal; some nodes may have more resources than others, e.g. a controller or gateway node. Hence, some nodes may have the resources to make reconfiguration decisions and enforce them, and others may not. Additionally, participating in expensive reconfiguration protocols may drain the resources of some sensors (see 4.2).

Adaptation styles. Frameworks should provide selectable styles of adaptation: e.g. centralised versus decentralised. One reconfiguration may be better suited to a centralised approach, e.g. to ensure consistency; minimize resource usage; whereas another may be suited by a decentralised approach.

4. EVALUATION

To evaluate our approach we demonstrate the effectiveness of Gridkit in supporting a flood-monitoring sensor network,; exploring how adaptations are performed in the face of changing environmental conditions. We also measure the costs introduced by increased flexibility and the ability to dynamically adapt.

4.1 Reconfiguration Case Study

The scenario examines how to predict flooding in a river valley in the North-West of England. Hydrologists need to deploy sensors (such as depth and flow-rate sensors); and the data from these is fed into off-site flood prediction models. Figure 5 illustrates the deployed sensor network. The sensors are placed at locations along the river; depending on the distances between sensors, different networks are used to communicate e.g. Bluetooth and 802.11b for shorter distances, GPRS for longer communication. The sensor nodes employ the Gumstix hardware platform, configured with multiple network interfaces (i.e. Bluetooth, GPRS, and 802.11b); they have an Intel XScale 400MHz CPU, 64 Mb of RAM and 16Mb of flash memory. These devices support a Linux kernel and run the Java 1.4.1 virtual machine. More detail about the scenario is in [8].

Off-site data dissemination is supported by the use of *spanning tree-based overlays*. These are commonly used in wireless sensor networks to disseminate data from a large number of sensors to a small number of logging or bridging nodes that form the 'root' of the tree. Prime examples of spanning trees are Shortest Path (SP) and Fewest Hop (FH) trees. FH trees are optimized to maintain a minimum number of hops between each node and the root. They minimise the data loss that occurs due to node failure, but are suboptimal with respect to power

consumption. SP trees, on the other hand, are optimised to maintain a minimum distance in edge weights from each node to the root. As a result, they tend to consume less power than FH trees, but are more vulnerable to node failure. Examples of the two trees are shown in figure 6.
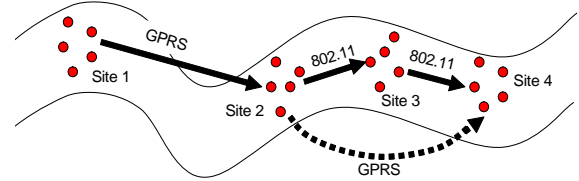

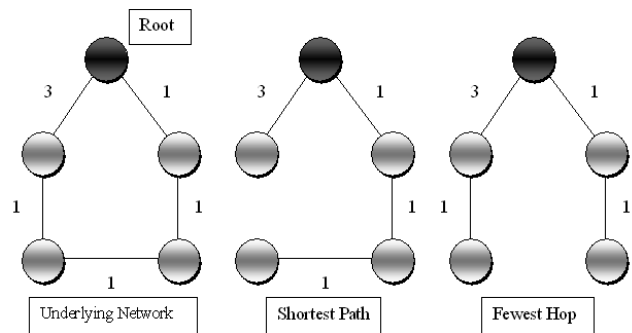**Figure 5: Sensor network to monitor river valley flooding**

Figure 6: Spanning tree overlays for sensor networks

Our middleware is customized as described in section 2. The tree root (gateway) node's interaction type is configured as the subscriber role to receive events of interest (i.e. those describing flooding information). The remaining nodes configure the interaction type to publish event information towards the subscriber. These nodes are all underpinned by a spanning tree overlay framework, which ensures that events are successfully routed to the sink of the network. The middleware also reacts to environmental changes. We now describe in detail the steps involved in one example reconfiguration scenario.

State 1 – Normal Conditions
When the sensor network is operating under normal conditions, the middleware is customized to be underpinned by an overlay providing a shortest path tree connected using a Bluetooth network. Hence, it is customized to save power.

State 2 – Flooding predicted
The scenario uses (locally computed) flood prediction models to determine when the valley will flood. These models generate an event stating that flooding is about to happen. Hence, the network adapts itself to become less vulnerable to node failure. The nodes configure themselves into a new topology – a FH tree.

Reconfiguration from state 1 to state 2
Reconfiguration is limited to the overlay framework only; hence, we deploy a distributed overlay framework consisting of local overlay framework instances. We illustrate the reconfiguration of the distributed framework in figure 7. The implementation of a spanning tree consists of three components in the local component

framework instance. The *control* component manages the topology i.e. how this node is connected to other nodes in the tree. The *forwarder* component routes messages towards the root of the tree. The *state* component stores local data such as neighbour nodes. The implementation of the shortest-path tree and fewest hop tree differ only in the control component—i.e. nodes are connected to neighbours based on different metrics. Therefore, to reconfigure the overlay from a shortest path tree to a fewest hop tree the control component must be reconfigured on each node.

The case study is small scale, with limited node mobility; therefore, the distributed overlay framework consists of a single configurator that stores the meta-data for the framework, reacts to the flooding event produced by the prediction models, and enacts the reconfiguration on each of the nodes to ensure that they are safely and accurately updated.
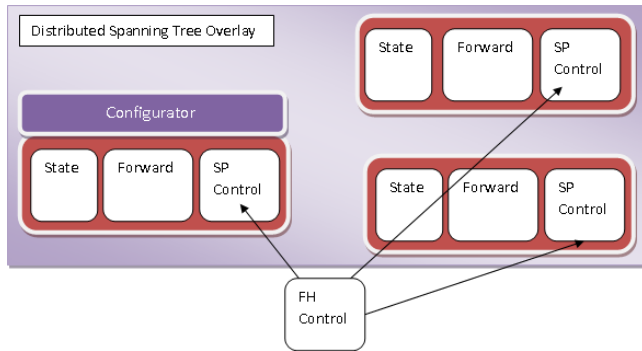

Figure 7: Reconfiguring the control component across 3 hosts

For space reasons, we illustrate only a single reconfiguration example. However, the distributed framework approach can be applied more generally. We believe it will support reconfigurations for alternative co-operating middleware functionality. For example, alternative overlay reconfigurations: changing the routing strategy by replacing the forwarder components, or increasing the dependability of the overlay by inserting new nodes into the framework, or replacing the repair algorithms. Similarly, it can be applied in different application domains such as multimedia and mobile e.g. the communication binding can switch across all hosts from streaming to text messaging when there is a reduction in available bandwidth.

4.2   Quantitative Costs

Our approach introduces complexity and autonomous behaviour into the realm of sensor middleware. Here we examine some of the costs that these bring. Gridkit is currently implemented using Java, and hence requires a virtual machine to be available on each sensor node (that said, Gridkit itself is language independent, and the approach could easily be replicated using different language implementations). In this section, we examine the memory costs—i.e. how much memory footprint is required to run the Gridkit middleware. This is a function of the Java class size. We next examine the dynamic memory footprint of the cost of creating dynamic frameworks: i.e. how much run-time memory.

4.2.1 Memory Footprint Cost of Middleware

Table 1 describes the static memory footprint cost (i.e. size on disk) of the jar files that make up the core elements of the sensor middleware. These show that a reasonable amount of memory space is required for each customised personality e.g. the

publisher role in a distributed framework totals 236 Kbytes; this easily fits onto the Gumstix, however is too large for minimal sensors e.g. Motes. This is constrained by the use of Java, and we are investigating C based approaches to minimise these values.

4.2.2   Dynamic Memory Cost of Meta-Data

Figure 8 illustrates the expense of maintaining meta-data in the network. This is a measure of the size of data stored in a node's system memory during the operation of the distributed framework that manages the spanning tree overlay (figure 7). The graph shows that meta-data held about local frameworks (3 connected components) remains constant irrespective of the number of nodes in the network. The basic distributed meta-data (figure 4a) increases a small constant amount for each new node in the network (approx. 144 bytes). However, the richer meta-data (figure 4b) sees a greater constant increase in system memory use due to the storage of component meta-information. However, in the scenario, only the root nodes maintain basic and richer meta-data; the cost is not spread across the network.

Table 1: Static Memory Footprints of sensor middleware

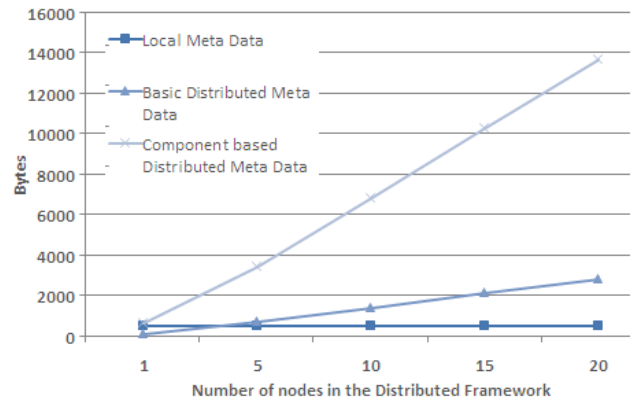| Component	Static Memory (Kbytes)
OpenCOM runtime	76
Gridkit Core Middleware	40
Group-based Meta-Architecture	52
Publisher Role	24
Subscriber Role	36
Spanning Tree Overlay	44

Figure 8: Dynamic Memory Costs of meta-data

5.   Related Work

There are a number of existing sensor middlewares, some of which were described in the introduction. These are important solutions in identifying the key characteristics required by sensor middleware, namely easy to use abstractions, suitable routing strategies, and resource management policies. However, none of these pieces of work considers dynamic reconfiguration and customisability to the same degree as Gridkit.

Outside the domain of sensor networks there are a set of technologies related to distributed reconfiguration. For example, k-Components [5] offers a decentralised agent-based approach;

i.e. each node makes local decisions about adaptation, until a global consensus is reached. Gridkit supports both centralised and decentralised approaches, and in that manner can be tailored to resource-constrained domains such as sensor networks. NecoMan [10] offers an alternative approach to dynamic reconfiguration. It supports safe, co-ordinated updates of distributed services, typically related to network protocols. It has not yet been applied to multiple reconfiguration domains to illustrate its flexibility; however, it presents many interesting ideas that could be applied within our frameworks. Ensemble [16] is a micro-protocol stack framework that is able to adapt its configuration dynamically; however, the reconfiguration mechanism is closely coupled to the micro-protocol implementation, and isn't as re-usable as Gridkit.

6. Conclusions and Future Work

In this paper, we have proposed the consideration of new dimensions in sensor middleware i.e. it should be customizable for the application environment, and also dynamically reconfigurable to change the behaviour of the sensor network. We have demonstrated how the flexible Gridkit middleware platform can be tailored specifically to the sensor domain. We have provided a component-based programming model for the development of sensor middleware and applications. In addition, we have introduced the concept of reflective distributed frameworks to manage the reconfiguration of software components across the sensor network. The approach was evaluated in a real-world sensor application, and shown that although the added complexity adds a cost to the performance this is not confining. Finally, we identify that this work is highly complementary to existing sensor middleware research e.g. ad-hoc message routing and power management.

There are a number of interesting future areas of research inspired by this work. Firstly, the creation of higher-level declarative languages that can be used by both middleware and application developers to describe reconfiguration actions on the sensor network, and also deal with potential conflicts that may arise from multiple policies. Secondly, the introduction of security measures to the distributed framework to ensure only authentic nodes can join a framework, and only members of the framework can make reconfigurations. Finally, the investigation of resource management policies (local and global); these will be used to adapt the middleware behaviour to conserve resources such as network bandwidth and battery power.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Blair, G., et al. "The design and implementation of Open ORB 2". *IEEE Distributed Systems Online*, 2(6), Sept 2001.

[2] Bonnet, P., Gehrke, J. and Seshadri, P. Querying the Physical World. *IEEE Personal Communications,* 7, 5 (October 2000), 10–15.

[3] Braginsky, D., and Estrin, D. Rumor routing algorthim for sensor networks. *In Proc. of the 1st ACM Int. workshop on Wireless sensor networks and applications* (2002), ACM Press, pp. 22--31.

[4] Coulson, G., et al. A Component Model for Building Systems Software. In Proc. of the Software Engineering and Applications (SEA'04), Cambridge, MA, USA, Nov 04.

[5] Dowling, J. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD Thesis. Trinity College, Dublin, 2004.

[6] Ganesh, A., Kermarrec, A., Massoulie, L. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. *In Proc. of the 3rd Int.Workshop on Networked Group Communication*, London, UK (2001).

[7] Grace, P., et al. Deep Middleware for the Divergent Grid. *In Proc. of ACM/IFIP International Middleware Conference*, Grenoble, France, December 2005.

[8] Hughes D., et al.. An Intelligent and Adaptable Flood Monitoring and Warning System. *In Proc. 5th UK E-Science All Hands Meeting (AHM'06),* Nottingham, UK, 2006.

[9] Intanagonwiwat, C., Govindan, R. and Estrin, D. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. *In Proc. of the 6th MOBICOM Conference,* Boston, MA, USA. August, 2000.

[10] Janssens, N., Michiels, S., Holvoet, T. and Verbaeten, P. NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks. In Proc. of Middleware 2004, Toronto, Canada, 2004.

[11] Jiao, B., Son, S. and Stankovic, J. GEM: Generic Event Service Middleware for Wireless Sensor Networks. *In Proc. of the 2nd International Workshop on Networked Sensing Systems (INSS)*. San Diego, California, USA, June 2005.

[12] Kon. F. *Automatic Configuration of Component-Based Distributed Systems*. PhD Thesis. University of Illinois at Urbana-Champaign, May, 2000.

[13] Madden, S., Franklin, M., Hellerstein, J. and Hong, W. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. *In OSDI 2002*, Boston, USA, December 2002.

[14] Murphy, A. and Picco, G. Transiently Shared Tuple Spaces in Sensor Networks. *In Proc. Workshop on Middleware for Sensor Networks*, San Francisco (CA, USA), June 2006.

[15] Slijepcevic, S., and Potkonjak, M. Power efficient organization of wireless sensor networks. *In Proc. IEEE International Conference on Communications*, vol. 2, pp 472-476, Helsinki, Finland, June 2001.

[16] van Renesse, R, Birman, K, Hayden, M, Vaysburd, A. and Karr, D. Adaptive Systems Using Ensemble. Software Practice and Experience 28:9 (August 1998), 963-979