Semantic Web Meets Autonomic Ubicomp

Roxana Belecheanu†, Gawesh Jawaheer*, Asher Hoskins*, Julie A. McCann* and Terry Payne†

†School of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ
{rab2,trp}@ecs.soton.ac.uk

*Dept. of Computing, Imperial College London
180, Queen's Gate, South Kensington Campus
London SW7 2AZ UK
{gawesh,asher,jamm}@doc.ic.ac.uk

Abstract: The placement of autonomic systems' management functionality into a ubiquitous computing environment is a difficult task due to the lack of systems' resources and the need for 'intelligence' to ensure that the system is self-healing/organising or configuring. For such systems to adapt to changes to their current environment they need to be able to (re) configure the workflow of their services. In this paper, we propose the ANS, an autonomic middleware for ubicomp devices. We briefly describe the architecture of ANS and its autonomic behaviour. The ANS follows a service centric design and therefore lends itself to semantic service description. This paper introduces the concept of services with ancillary behaviour and illustrates the use of OWL-S to semantically describe them. Commitment is discussed as an example of ancillary behaviour which can be used to achieve self-healing at service level. We illustrate our approach with examples from the homecare scenario of a cardiac patient.

1 Introduction

In recent years we have witnessed substantial development towards realising the vision of Ubiquitous Computing (Ubicomp). Described by Mark Weiser in a now famous article [11], ubicomp is about a world where computing devices are everywhere and yet disappear in their environment. This has lead to the emergence of fields such as sensor networking where researchers are taking up the challenge of building small battery powered computing devices which integrate with (as well as monitor) their environment. Furthermore, the ability of such devices (which may also be mobile) to communicate using low bit radio has opened a vast array of potential applications.

An open question in the field of ubicomp is how to manage devices and their applications. This is not a trivial problem and the characteristic properties of such devices such as size and processing constraints limit portable or deployable solutions that have been proposed within the field of traditional distributed systems. For example, the small size and relatively low costs of these devices imply that they are highly likely to be deployed in large numbers. Once deployed, communication with

the devices are done either through low bit radio or wireless LAN technologies, both of which are highly expensive in terms of power consumption. Thus, message passing, which is one of the traditional means of building distributed applications, cannot be used lightly. Another important factor is that of a device being aware of its peers (with respect to a given task) and of both their status and its own. This awareness is necessary for coalitions of devices and services to react to, or pre-empt failure, both at the individual (service or device) level and at the coalition level.

Our proposed solution to the problem of managing a distributed, open, ubiquitous environment is the Autonomic Networked System (ANS), an architecture that can be used to manage and develop applications for ubicomp. The ANS architecture includes autonomic middleware (which we are developing) that simplifies the management and development of ubicomp applications. For example, managing an evolving set of devices and services that may grow dynamically through the incremental addition of new devices, or managing transient devices that enter the ANS environment. One of the challenges of handling such devices is that it is difficult to assume a-priori knowledge of the services that may be available. Likewise, one cannot simply assume that all service descriptions will adhere to a single ontology (especially if different devices originate from different vendors, etc). To address this, we are leveraging research on interoperable service descriptions (i.e. Semantic Web Services) for tasks such as service discovery, management, invocation, and monitoring.

The Semantic Web effort is developing standards and technologies to facilitate machine understanding of content on the Web, in a way that allows for richer discovery, integration and navigation of data and automation of tasks [1]. To this end, the Semantic Web extends the data existing on the Web through conceptual annotations, by defining ontologies for each domain of knowledge and by specifying logical rules for processing this knowledge [17]. In the context of a ubiquitous environment like the ANS, the Semantic Web provides an essential infrastructure for representing and linking devices with data sources and describing and reasoning about their functionalities and capabilities. OWL-S [2] is a service framework consisting of a set of Semantic Web ontologies that support the description of, and reasoning about services, thus providing a uniform view to service functionality [16].

Furthermore, due to the high variability of Quality of Service (QoS) of wireless data communications (e.g. line rate, delay, throughput, error rate), there is also a higher need for adaptability of the services running on the connected devices. Abstracting device functionality (i.e. physical functionality) as services (i.e. logical functionality) and adding semantic information to the syntactic definition of these services can help achieve the higher level flexibility of the ubiquitous system, discovering and re-establishing links between devices and their services dynamically.

In this paper, we describe how ANS services benefit from semantic annotations within the context of the ANS project [12]. In Section 2 we outline the application domain chosen for the project and present autonomic computing and how it relates to the ANS. We also describe the various autonomic behaviours exhibited by ANS and illustrate these with an example. In Section 3, we present the building blocks of the ANS and we describe the ANS middleware. Section 4 describes the semantic service description using OWL-S. And finally, we examine related work in Section 5 and present our conclusions.

2 Autonomic Ubicomp in homecare

To build a proof of concept, we have chosen a challenging application domain for the ANS, namely the provision of care in the home for coronary heart disease (CHD) patients. For the purpose of testing our implementation of ANS, we have been building the associated hardware and preparing a laboratory setup to act as a testbed for our applications (a discussion of the hardware aspects of ANS is out of the scope of this paper but can be found in [13]). One can imagine a CHD patient living in an intelligent home equipped with sensors, which are running the ANS. This is a dynamic environment where sensors may fail but recover from that failure (self-heal) and friendly alien devices such as a visiting nurse's PDA may interact with sensors in the home (self-configure).

Borrowing from biology, an *autonomic system* is one that provides background support without requiring external intervention (e.g. the human nervous system). Likewise, in computing, an autonomic system is one that has the following properties:

- *Self-healing* - devices can leave the system without applications breaking.
- *Self-configuration* - The system can set itself up and discover other devices automatically.
- *Self-optimisation* - At all times the system is in the "best" state. Note that different devices have different ideas about what constitutes "best", i.e. given a certain situation the services provided are as best they can be at that time.

Autonomic behaviour is implemented systemically across all layers of the ANS architecture (as opposed to having an "autonomic network" or "autonomic software" layer). For example, in the network layer packets that are not acknowledged can be retransmitted. A device that fails to acknowledge a packet after a certain number of tries is assumed to have left the network (either by moving or failing). Control then moves up to the application layer, which must send commands back down to the network layer to search for alternate, equivalent services. If one is found then the application may have to reconfigure to cope with any differences between the failed and the newly discovered service. If no such service can be found, then the application should handle this gracefully, either by identifying an alternate way of achieving the overall task, or through graceful degradation. Self-configuration is similar, whereby the application starts with no services, but searches for them on the network, essentially achieving the overall task by building it from components that best fit the requirements. For example, a nurse with a PDA might enter the intelligent home, but for the PDA to communicate with the home's ANS, it might require a service that supports secure communication (such as secure drivers, etc).

For a system to be self-optimising the service discovery phase must include some information about how to pick the optimal service. The ANS does this by requesting services that meet a set of parameters (representing QoS requirements for example) and picking those devices that come closest to those parameters. By regularly seeking alternate services, the system can refine its performance by identifying and utilising better services as they appear.

In order to illustrate an example of self-healing behaviour, assume that two location services are available in a room: one based on video and running on device ***V*** that can return an accurate location, and one based on simple light sensors, running on device

L, which can only return a vague location. An application running on device ***A*** wishes to know the location of a person in the room to the greatest possible accuracy. Hence:

1. ***A*** sends a service request for a location service that will deliver location data with a high degree of accuracy
2. ***V*** responds stating that it can provide that service and can meet the degree of accuracy required
3. ***L*** responds stating that it can provide that service but not at that level of accuracy
4. ***A*** chooses ***V***as a location service
5. At some point later, ***V*** stops working
6. ***A*** now sends out another service request
7. This time, only ***L*** responds.
8. ***A*** chooses ***L***. The location information it now receives is not as good as that from ***V*** but will enable the application to keep running.

In the future, if device ***V*** becomes available again, the system will self-optimise through re-tendering of its services.

3 ANS building blocks

The main building blocks of the ANS architecture are shown in **Figure 1**. The ANS middleware is described in this section whilst the semantic service description is explained in Section 4. The ANS comprises of the middleware that is decentralised among the ubicomp devices in the intelligent home and includes the semantic service description of all the functions that the devices can carry out. The application as such is decoupled from the ANS but communicates its autonomic logic to the ANS in terms of trigger action rules. Typical ubicomp devices in the ANS environment would be microcontroller based wireless sensor nodes, PDAs and embedded PCs.

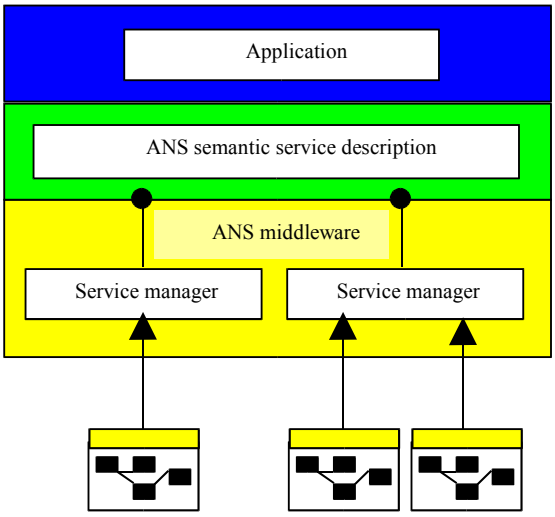

Figure 1 ANS components

At the lowest level, the ANS middleware protocol provides a common language that ANS devices can use to communicate. The protocol runs on top of a packet-based network layer such as IP. This packet-based layer need not provide lossless transmission since the ANS contains mechanisms for dealing with lost packets. The ANS protocol is deliberately very simple so that all ANS devices can support it fully. All complex and device specific oper-

ations are implemented as services. Along with basic low level networking commands such as `acknowledge` and `ping`, and service discovery commands, the ANS provides two commands that are used to implement services; these are “Labelled Command” and “Labelled Data” commands respectively.

Firstly, a “Labelled Command” is used for all operations on a service that cause an action of some kind. The “Labelled Data” command is used for the transfer of data to and from a service. Labelled data packets may be cached by devices on the network whether they understand the service that the data relates to or not, and then used at a later date to reconstruct data streams in the event of a network or device failure.

All labelled data and command packets contain an ID number to identify the service that they refer to. This ID number is unique only to the device providing the service and is made available during the service discovery phase. Some services may also implement a sequence number so that out of sequence packets or repeated packets may be discovered and dealt with. Not all services require this functionality however. For example, a service that controls a light does not care if it is mistakenly sent two "turn light on" commands in sequence, the net result (the light being lit) is the same.

4 Semantic Web Services

One of the problems with deploying devices within a home-care environment is that as different vendors may describe similar devices in different ways, one cannot make any assumptions about standardized or shared vocabularies. This leads to problems both in terms of describing and discovering services (as there is no guarantee that the service description will use the same terms as those found in the service request), and interpreting the service description of a newly introduced service. Such scenarios may be resolved by either enforcing a policy of using devices that conform to a given vocabulary, or though the use of mediators that translate between different ontologies. Semantic Web Services utilise formal concepts logically defined within the Semantic Web to describe their capabilities and to define usage protocols. By using such concepts, logical entailments can be made to determine if two services or devices are similar (such as a “photo-quality printer” is a printer that produces very high resolution images on, for example, glossy paper).

4.1 OWL-S

OWL-S [2] is an OWL [3] based ontology language for describing Semantic Web Services. It consists of three top-level ontologies that contain constructs for describing various aspects of a service. These ontologies are the *Profile*, the *Process Model* and the *Grounding*:

- The *Profile* contains a high level declarative description of the service: e.g. the service capability (i.e. what it does), its inputs, outputs, requirements for invocation (i.e. preconditions) and the effects of its execution. The *Profile* may also provide information about non-functional aspects of the service, such as QoS measures, cost, availability, etc., as well as information about

the service provider. The description of service properties and capabilities are useful for automatic service discovery; an agent can use it to determine the applicability of a particular service for a particular task [4].

- The *Process Model* contains a declarative specification of the operations provided by the service, i.e. how the service works. Each of the operations, or processes, are defined in terms of initial resources (i.e. inputs and preconditions), and resulting outcomes (outputs and knowledge-effects). A control and dataflow model is represented as a hierarchical workflow, using one of several workflow constructs (e.g. sequence, choice, split-join operators, etc). This approach facilitates the construction of new services in terms of a composition of existing services; hence a meta-goal could be described using different compositions of different services.
- The *Grounding* is a mapping of the abstract information of the inputs and outputs described in the *Process Model*, onto the actual messages exchanged by the provider and the requester (described using the Web Services Description Language – WSDL). Typically, a grounding specifies a communication protocol, message formats, port numbers and other service-specific details necessary to access the service.

Attempting to resolve semantic mismatch through logical reasoning, however, does not guarantee any form of autonomic behaviour; it merely assists with composing different services within an environment of heterogeneous services. One important issue within a home-care environment is that there may be a limited number of devices, each providing services that take a finite time to execute, and that may have to be shared by several contexts. Given this, no guarantees can be made that any single service will be available for invocation at any given time, due to the fact that the service may be busy responding to another service request. Such problems also occur within Agent and Grid-based environments - for example, it is possible to make a "reservation" on an OGSA grid service for access at a given time. Likewise, agent teams can form through agreements (such as "commitments") made ahead of invocation time. To facilitate such agreements between services, it is necessary to support and apply additional (or ancillary) services to the main (i.e. core) service descriptions.

4.2 Using OWL-S to map Ancillary services in ANS

Ancillary services within ANS are those that provide additional functionality to the core service as described within an OWL-S description, and which augment the service's capability, or enhance the QoS achieved by the core service. Typically, a service description presents the necessary information for executing that core capability. However, services are frequently accompanied by supporting functionality, which is relevant only to the context of the core service, but may not play a direct role in the services invocation (such as managing accounts, monitoring behaviour, or making reservations for deferred execution).

One of the risks in a ubiquitous environment is the high failure rate of message transmission due to packet loss, which results in failure of communication between devices and/or sensors. Due to this risk, and because of high power consumption that characterises message transmission, it becomes necessary to reduce the number of service requests, or to support a mechanism whereby several activities may be requested

via a single call. Examples of ancillary behaviour include operations like commitment, authentication, encryption/decryption, costing, etc. These can be either mandatory (i.e. operations that must be executed in addition to the service's core functionality, such as authentication, or the service will not work otherwise), or optional; they can contribute to the delivery of a service either by providing a supporting role (i.e. enable the core service) or an enhancing role (increase the value of the core service) [5].

Autonomic services in ANS manifest ancillary behaviour; commitment is an example of particular importance to ANS and to a ubiquitous system in general, as explained in the next section.

4.3 Commitment based services in ANS

The concept of commitment and its use in teamwork models has been widely researched, both theoretically [6, 7,8] and as application to different practical situations of cooperative problem solving [9]. Commitment is a core concept in the theory of local and social agent behaviour and is defined by the notion of achieving a persistent, joint goal. A persistent goal is a commitment to an action and has the following implications on the beliefs and behaviour of an agent: once a persistent goal is adopted, the agent believes that the goal is currently false and it wants the goal to be true. The agent will then hold these two beliefs until either the goal becomes true, or the agent comes to believe that it will never be true, or the goal becomes irrelevant with respect to some other higher-level goals.

Within ANS, commitment helps achieve two objectives regarding ANS services:

1. *persistency*: by defining that, once a service agent adopts a goal it will not be able to drop it unless certain conditions arise and it will have to keep trying to attain that goal, until it succeeds.
2. *self-awareness*: through commitment, services can be aware of other services they are dependent on.

To illustrate how commitment may be used to achieve self-healing behaviour, imagine that the ANS provides a location service, which is a composition of a floor sensor service and a vision service. The floor sensor service provides approximate details of the patient's location to the vision service, which uses this to target the camera and identify the actual physical position and orientation of the patient. If the floor sensor service fails, then an acoustic sensor service is used instead.

By implementing the location service as a commitment-based service, its invocation would implicitly require that the location service becomes committed to the action described by its workflow, and which has as an implicit goal to serve the patient's accurate location to the application. As soon as it detects that the floor sensor service has failed, the location service will attempt to achieve the goal to which it has committed, by discovering an alternative service for determining the rough position of the person (e.g. through using the acoustic service).

4.4 Describing ancillary services in OWL-S

Ancillary service descriptions should be loosely linked to ANS service specifications, such that the resulting workflow for both the core and ancillary services will be realized (through entailment) when an agent reasons about the service (with respect to a usage context). Ancillary ANS service behaviour has two salient characteristics:

1. It is *dynamic*, i.e. it involves communication with other services of which instances can only be discovered at runtime. Thus it cannot be included in the static description of the service's process.
2. It is *core function independent*, i.e. it is common to a range of services with different core functions. It can therefore be annotated in a separate ontology, in order to be referenced by several higher-order services and thus to eliminate redundancy.

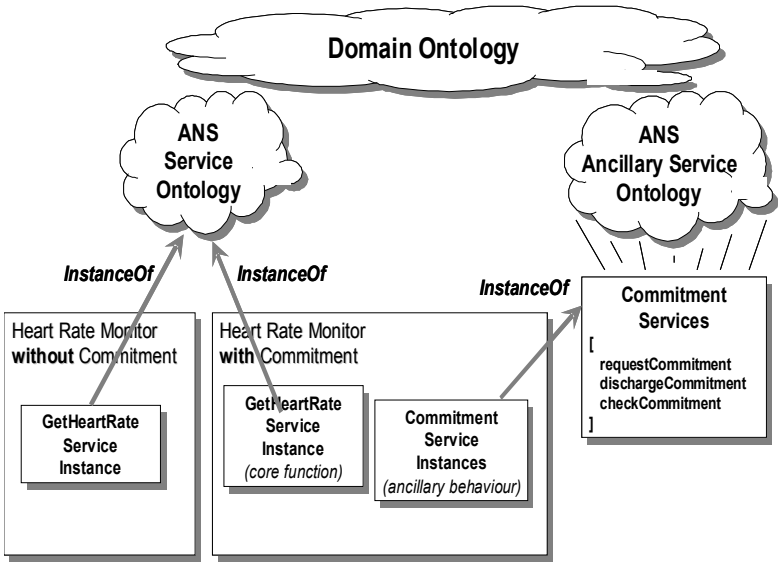

Figure 2: ANS ontologies and the derivation of core and ancillary services

For these reasons, ancillary behaviour is abstracted and described separately from a given service's core functionality, thus facilitating several different services sharing the same ancillary behaviours. For this reason, the semantic description layer of ANS consists of the following ontologies (Figure 2):

1. The *Domain ontology* – this ontology represents an OWL-based representation of the domain in which the ANS is being applied. For example, with the healthcare scenario, it would include concepts such as:
   - ANS roles and users (patient, nurse, doctor, etc.)
   - Devices (e.g. PDA, lamp, TV set)
   - Sensors (e.g. temperature, heart rate, cabinet sensor, light, acoustic, pressure)
   - Knowledge/Information concepts that are used or consumed by the services (e.g. heart rate, weight, light intensity, temperature, location, time).

2. The *ANS services ontology* – describes core services without ancillary behaviour, like *GetTemperature*, *GetHeartRate*, *SetLightIntensity*, *etc*. Many of these may be offered by a sensor or device within the environment. For example, an intelligent lamp may offer two different light services: *SetLightIntensity* and *GetLightIntensity*. One way to classify these services is by the function they perform (Figure 3).

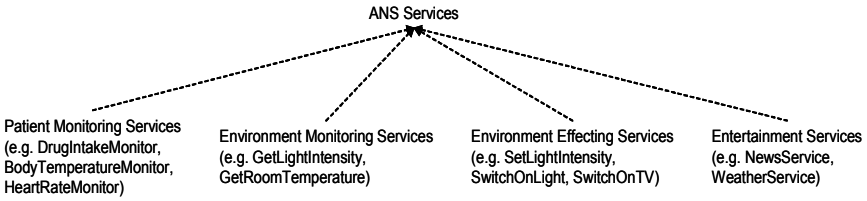

Figure 3: A classification of ANS services

3. *ANS ancillary services ontology* – describes services like commitment services, encryption services, etc. From the point of view of OWL-S annotation, this ontology can be viewed as a repository of partially bound workflows (see below).

Services within the healthcare environment may be software-only services (i.e. they are independent of any specific hardware), or they may represent, or be offered by a physical device (such as a sensor or actuator). The ontologies define a set of semantic concepts that can be used to describe these services; the concepts themselves may be class definitions (such as defining a role – e.g. *Nurse*), or they could define a specific instance (e.g. an instance of a Nurse might be "*Ms Nightingale*"). Service definitions consist of an OWL-S service instance, with other instances defined by their concept classes (defined, for example, in one of the ontologies described above). Traditional OWL-S services assume one or more core service instances (as in *GetHeartRate* in the "*Heart Rate Monitor without Commitment*" service in Figure 3 above). The following describes how such services can be augmented by: i) defining ancillary services (as in the "Commitment Service" instance, defined by the ANS Ancillary Service Ontology); and ii) linking core service descriptions with ancillary services.

To illustrate how core services can be augmented by ancillary services, consider the following example: A Heart Rate monitoring service has been defined that will support commitment (i.e. *HRMonitor_with_Commitment*). The resultant workflow for this service is a sequence of three processes: *RequestCommitment*, *GetHeartRate* and *DischargeCommitment*. The first and last processes in this sequence (*RequestCommitment* and *DischargeCommitment)* represent subprocesses of the ancillary behaviour of the *HRMonitor_with_Commitment*, and are provided by the ANS Ancilary Service Ontology. The *GetHeartRate* process represents the core function of this service, which is defined here as an atomic process:

```
<process:AtomicProcess rdf:ID="GetHeartRate">
 <hasOutput rdf:resource="ansConcepts#HeartRate"/>
</process:AtomicProcess>
```

To combine the core and ancillary workflows, we use the OWL-S *Simple Process* class as an unbound service abstraction that can be realised at runtime by another service instance. The commitment workflow can be written as follows:

```
<process:CompositeProcess rdf:ID="Commitment_Process">
 <process:composedOf>
  <process:Sequence>
   <process:Components>
    <process:AtomicProcess rdf:resource="#RequestCommitment"/>
    <process:SimpleProcess rdf:resource="#Core_Function"/>
    <process:AtomicProcess rdf:resource="#DischargeCommitment"/>
   </process:Components>
  </process:Sequence>
 </process:composedOf>
</process:CompositeProcess>
```

Additional annotations are added to the core *GetHeartRate* service definition to link in with the ancillary commitment service definition. Two sets of annotations are required to achieve this: i) by stating that the *GetHeartRate* process ***realizes*** the abstract *Simple* process included in the ancillary service Process Model; and ii) by referencing the ancillary service Profile, via the ***serviceParameter*** property:

```
<process:AtomicProcess rdf:ID="GetHeartRate">
 <process:realizes rdf:resource="ansAncillary.owl#Core_Function"/>
 </process:AtomicProcess>
...
<ANSServiceProfile rdf:ID="HRMonitor_with_Commitment_Profile">
 <service:presentedBy rdf:resource="#HRMonitor_with_Commitment"/>
 <profile:has_process rdf:resource="#GetHeartRate"/>
 <ansProfile:ANSAncillaryFunctionality>
  <ansProfile:ANSAncillary>
   <profile:sParameter rdf:resource="ansAncillary#Commitment_Profile"/>
  </ansProfile:ANSAncillary>
 </ansProfile:ANSAncillaryFunctionality>
</ANSServiceProfile>
```

The ***realize*** statement provides the expansion of the *Core_Function* definition within the ancillary workflow. By asserting this statement in *GetHeartRate*, the ancillary service definition is effectively unbound, until reasoned about in context with the *GetHeartRate* service. In addition, this service would also include a Grounding definition for the *GetHeartRate* atomic process, thus providing an invocable binding to the otherwise abstract Simple process definition.

An important consideration is how a service requester can distinguish between a committed and non-committed service during discovery, as typically a service is requested according to the functional parameters of the core service. To achieve this, *ANSAncillaryFunctionality* is defined as a *serviceParameter* type of property, in the profile of any ANS service. Thus, service discovery mechanisms can seek services with respect to core functionality, but then inspect non-functional properties to determine what ancillary services are also provided as part of the service description.

By separating the core function from the ancillary behaviour, the main research task becomes the runtime composition of the workflows describing the core and ancillary sub-processes into one executable workflow that can be enacted. Also, the implementation of commitment needs to be carefully thought, due to the challenges this raises in ubiquitous environments (e.g. [10]).

5 Related Work

There have been other approaches to designing a middleware for sensor systems. For instance, [14] describes the Impala middleware for developing sensor applications. Impala is designed to work on microcontroller based wireless sensors and provides the functionalities of an operating system, resource manager and event filter. The authors elaborate on the need for software adaptation in wireless sensor networks. The sole application they focus on is a routing protocol for transmitting messages. They propose a technique for enabling adaptation of the routing application by loading different routing applications on each sensor node. Each application is characterised by a set of parameters and the routing application with the best parameters is chosen to run. Rules define the conditions under which a switch of applications should take place. In contrast to ANS, the granularity of adaptation in their work is coarse grain. Another limitation of their work is their narrow focus on adapting a routing application from which it is difficult to make general conclusions. In contrast, as illustrated by the examples in the previous sections, the autonomic behaviours in ANS are not specific to a particular application. Furthermore, by incorporating the concept of services in the ANS, the latter lends itself to semantic service descriptions which aids discovery and integration. Although the Impala system was designed for microcontroller based wireless sensors, the implementation described in [14] was carried out on PDAs (HP IPAQ). Once again, this is a limitation, which prevents any generalisation of their results. On the other hand, ANS will be implemented on real microcontroller based wireless sensors, which is being built as part of the project [13]

[15, 18] describe another middleware for wireless sensor networks, concentrating on the structural aspects of such a system. For example, although they identified the ability to adapt as an important characteristic of the middleware, their work is mainly conceptual and no implementation details are given. By comparison, autonomic behaviours form the crux of the ANS. We also note that the work presented in [15, 18].

6 Conclusions

This paper presents ANS, an autonomic middleware system for ubicomp devices. The paper discusses the types of autonomic behaviour of ANS (self-healing/optimization/configuration) and how these can be achieved via a service-based design and through the application of Semantic Web to annotate ANS services. We introduce the concept of services with ancillary behaviour (also called higher-order services). We argue that ancillary behaviour can be used to realize the self-healing nature of the ubicomp environment, taking as an example the concept of commitment developed in agent based research. We then show how OWL-S can be used to separate and abstractize the ancillary functionality from the core functionality of the higher-order service, to reflect the dynamic and core-function independent nature of ancillary behaviour. Future work will address the runtime composition of core and ancillary sub-processes into one executable workflow. The model will then be applied within a real sensor network and its performance will be measured in terms of the ability to reconfigure dynamically on devices with highly limited resources.

References

1. Koivunen, M. and Millerm E. – W3C Semantic Web Activity, in Semantic Web Kick-Off in Finland – Vision, Technologies, Research and Applications, May 2002, pp. 27-44.
2. The OWL Services Coalition: Semantic Markup for Web Services (OWL-S): http://www.daml.org/services/owl-s/1.0/
3. Dean, M., Schreiber, G., Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuiness, D.L., Patel-Schneider P.F., Stein, L.A.. "OWL Web Ontology Language Reference", W3C Candidate Recommendation, 18 August 2003, http://www.w3.org/TR/owl-ref/
4. McIlraith, S.A., Son, T.C. and Zeng, H. Mobilizing the Semantic Web with DAML Enabled Web Services. In Semantic Web Workshop, 2001.
5. Baida, Z., Akkermans, H. and Bernaras, A., 2003., The configurable nature of real-world services: analysis and demonstration, ICEC-Workshop.
6. P. R. Cohen and H. J. Levesque. Teamwork. Handbook of MultiAgent Systems, 25(4):487–512, 1991.
7. Levesque, H. J. and Cohen, P. R. On acting together. In Proceedings of AAAI-90, 1990.
8. P. R. Cohen and H. J. Levesque, Intention is choice with commitment, Artificial Intelligence, 42 (1990), pp. 213-261
9. Tambe, M., Towards flexible teamwork, Journal of Artificial Intelligence Research, 7 (1997), 83-124.
10. Chen, H. and Finin, T.– Beyond distributed AI, Agent Teamwork in Ubiquitous Computing, In: Workshop on Ubiquitous agents on embedded, wearable and mobile devices, AAMAS Conference, Bologna, July 2002
11. Weiser, M, "The Computer for the Twenty-First Century," Scientific American, pp. 94-10, September 1991
12. http://www.ubicare.org/projects-ans.shtml
13. McCann J, Hoskins A and Jawaheer G, ANS (Autonomic Networked Systems) for Ubiquitous Computing, Adjunct Proceedings of UbiComp 2004, Nottingham, UK, September 2004
14. Liu T, Martonosi M, Impala: a middleware system for managing autonomic parallel sensor systems, Proceedings of the 9th ACM SIGPLAN symposium on Principles and practice of parallel programming, Jun 2003
15. Blumenthal J, Handy M, Golatowski F, Haase M, Timmermann D, Wireless sensor networks – new challenges for software engineering http://www.vs.inf.ethz.ch/publ/se/sensornw-etfa-uni-ro.pdf
16. Lassila, O. and Adler, M., Semantic Gadgets: Device and Information Interoperability, April 2003.
17. Laamanen, H., Helin H., Laukkanen, M., Semantic Web and Software Agents meet Wireless World. In: Semantic Web Kick-Off in Finland – Vision, Technologies, Research and Applications, May 2002, pp. 241-250
18. Römer K, Kasten O, Mattern F, Middleware Challenges for Wireless Sensor Networks.ACM Mobile Computing and Communication Review, pp. 59-61, October 2002