Context as Autonomic Intelligence in a Ubiquitous Computing Environment

Markus Huebscher, Julie A McCann and Asher Hoskins

Department of Computing,
Imperial College London, UK, SW7 2BZ,
P: +44 (0) 207 594 8375, F: +44 (0) 207 581 8024
E-mail: {mch1, jamm, asher}@doc.ic.ac.uk

Abstract: This paper presents the ANS architecture that uses ubiquitous computing to monitor medical patients in the home. Since there is no notion of the patient carrying out maintenance of such a system, it must be self-managing or *autonomic*. In the ANS sensors, such as temperature, location, etc., use a form of logic to abduce a context, i.e. the state/quality of a given device or its function. Our contribution lies in the emergent autonomicity of the architecture driven by its ability to derive the most appropriate source for a particular application. This is done by allowing the application to define mathematically its own notion of what level of service provides the best satisfaction and is based on Quality of Context attributes that describe each alternative. These application-specific definitions allow the ANS to optimally adapt to alternative sources when context providers fail or their quality changes. Further, we uniquely evaluate the trustworthiness of the different sources of context, thus allowing the applications to choose how much risk they are willing to take in the hope of receiving satisfaction. The ANS framework is lightweight and can provide real-time adaptation, which is necessary in resource-starved ubiquitous computing environments that support medical applications.

Keywords: Autonomic frameworks, ubiquitous computing, utility functions.

Reference: to this paper should be made as follows: Huebscher M and McCann J.A., Hoskins A., (200x) 'Context as Autonomic Intelligence in a Ubiquitous Computing Environment', *Int. J. of Internet Protocol Technology (IJIPT) – Special Issue on: 'Context in Autonomic Communication and Computing'*, Vol. n, No. n, pp.n-n.

Biographical Notes: **Markus C. Huebscher** received his Diploma degree at the Swiss Federal Institute of Technology Zurich (ETH) in 2002 and is now working on his PhD thesis at Imperial College London. His research interests include self-adaptiveness in context-aware applications, wireless sensor networks and mobile hand-held devices.

Dr Julie A. McCann received her PhD doctorate in 1992 University of Ulster, N. Ireland, and then joined City University, London, where she became the leader of the Systems Architecture Research Centre, researching adaptive Operating System, Database and Distributed systems projects. She is currently at Imperial College lecturing Operating Systems courses and has published over 40 refereed papers. Her current research lies in self-management systems architectures for improved quality of service and performance. Julie is a Chartered Engineer and a Member of the British Computer Society.

Asher Hoskins graduated with a degree in Computer Systems Engineering at the University of Kent at Canterbury in 1994. He then joined Philips Research Laboratories in Redhill. In 1997 he started the wearable electronics projects with two recent graduates from the Royal College of Art. During its five-year lifespan this project produced many electronic and conceptual designs for wearable systems, including the successful Philips/Levi ICD+ jacket. At the beginning of 2004 he moved to Imperial College to work on the ANS project.

1. The Road to Calm

Weiser's vision of ubiquitous computing [14] involves the notion of *calm* whereby the system, which may be composed of many very differing forms of computing elements, tailors its operation to best fit the user, application or goals required of it. Consequently, *calm* requires a paradigm shift in computing. Most of the work addressing this in the field of ubiquitous computing (ubicomp) has centred around human computer interaction and, due to the big brother nature of a constantly monitoring sensor network, security and trust. However, more recently there

Copyright © 200x Inderscience Enterprises Ltd.

has been the beginnings of a move towards pushing the *calm* down into the very infrastructure of the ubicomp architecture. These architectures can consist of heterogeneous sets of computing devices such as PCs, PDAs, and (combinations of) sensors, which monitor ambient temperature or the location of a person in a given environment, for example. This equipment may use a combination of wired (e.g. traditional IP or power based X10) and wireless networking (e.g. 802.11), drawing power from normal electricity supplies, ambient power (the lighting infrastructure) or batteries. Applications of ubicomp range from environment monitoring to home entertainment to medical patient monitoring. For such a ubicomp architecture to provide a *calm* computing environment, the user will not be interested in how this heterogeneous architecture is installed, interacts, or is maintained. Consequently, a new body of research has focused on autonomic computing as a way to build *calm* ubicomp systems.

Current autonomic research has focused on the self-management of larger server systems running on PCs and more powerful equipment, and on the whole has shown that the extra intelligence required to self-manage does in fact cost [8]. The realm of ubicomp brings new challenges to autonomic computing, that is we do not have the luxury of the resources available to current autonomic architectures such as [8, 9 and 4 etc.]. These tend to have a centralised autonomic engine holding the intelligence while governing the systems' self-adaptation. Due to resources being highly restricted in a network of sensors, autonomic intelligence must be distributed amongst its nodes and its footprint must be extremely lightweight (e.g. our sensor nodes hold 1K memory).

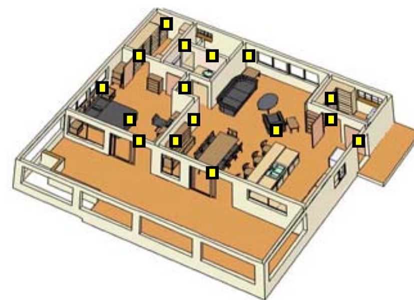
In this paper we introduce a ubicomp framework named ANS (Autonomic Networked Services). In this architecture we wish to provide emergent-like autonomic intelligence that adapts to many differing activities carried out by the system and users, and can depend on temporal and locational aspects. We must be able to seamlessly plug and unplug units into the ubicomp system. The functionality of these nodes would be determined by the context into which they are plugged or located. For example a *webpad* unit can be the remote control for your television when in the sitting room, the interface to your fridge in the kitchen or just idly hung on a wall when not needed; possibly doing backups of data to the network or ordering groceries to stock a depleted refrigerator. When a unit breaks or upgrades are required, new units can also be added seamlessly.

Like the Autonomic Nervous System of living creatures our ANS operates without the need for control by the user, functioning in an involuntary, reflexive manner in the background without their interference or knowledge of its mechanics. This is exactly what is needed to support ubicomp environments, especially in the "intelligent home" providing medical applications where constant technical support is impossible. A ubiquitous network must have a high degree of automatic adaptability and reconfigurability since it is inconceivable that we should require users to perform explicit systems management and maintenance. Therefore the ANS is self-configuring, self-optimising, and self-repairing.

2. The ANS and its Application

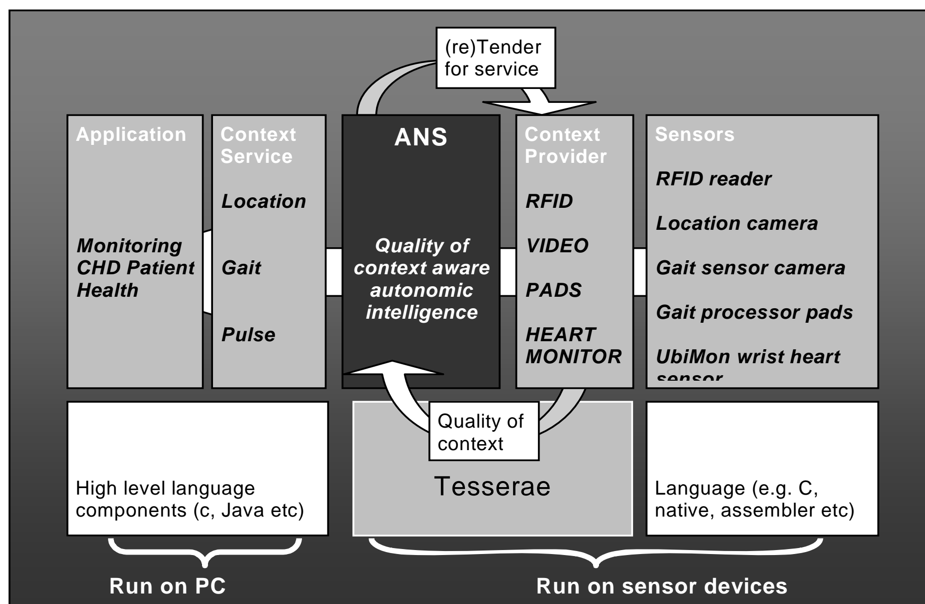
The provision of care in the home for coronary heart disease (CHD) patients is the challenging application domain we have chosen for the ANS. For the purpose of testing our implementation of ANS, we have been preparing a laboratory mock-up of a home to act as a testbed. One can imagine a CHD patient living in an intelligent home equipped with sensors, which are running the ANS (see Figure 1). This is a dynamic environment where sensors may fail or their reading may no longer be trusted, but must recover from these problems (self-heal). Also, *friendly alien devices* such as a visiting nurse's PDA may suddenly join the home system and interact therein (self-configure). Furthermore, in an emergency situation, more resource may be required in a given room to relay e.g. detailed patient vital signs to an approaching ambulance, furthermore other less used sensors must be willing to reconfigure their primary function and donate their processing or communications capacity to the emergency application, and to do this without delay (re-configure).

Figure 1 Home plan with sensor positions



To achieve this we have the ANS software executing on each of the sensor nodes in the home. Effectively we wish the autonomic intelligence to be emergent within the system and to avoid any central management components for both reliability and performance reasons. To increase reliability each given node in the network can have many functions e.g. video sensor might be relaying were the patient is within the home (*location*) as its primary function, but also may be capable of analysing the gait of the patient, should that be required. This redundancy is key to the function of the ANS; based on the well-established idea that sensing is cheap while actuation is expensive. In the ANS architecture the redundant functions are programmed as components in whatever language is best suited to the task (e.g. C or Java etc). The autonomic logic that prescribes when to switch functionality (re-configure) is described using our lightweight language, Tesserae, see Figure 2. L*ocation* and *gait analysi*s are known as *context services* in ANS. If a node requires a location service from the best device it can request that and when a given device is no longer providing

Figure 2 ANS architecture for Heart monitoring



the 'best' service (in terms of say quality or accuracy) ANS automatically reconfigures the next or better still a new 'best' service joining the system.

Effectively all services are described in terms of the context they provide in the ANS and this essentially drives the emergent autonomic intelligence throughout the architecture. Figure 2 shows the specific ANS architecture. The application level represents the code that runs typically on a PC in the home that is looking after the patient, perhaps relaying heart or other monitoring data to a hospital etc. To do this the application requests a service such as 'where is the patient?', 'how are they standing?', and 'what is their current heart rate?', see Figure 2. Any language can be used to write the components that make up the application code as well as the context service processing. The core of the system is the ANS protocol. An ANS node using a given context service is kept aware of its quality/accuracy though the use of a *feedback* mechanism in the form of given context provider's current quality of context. So for example, patient location can be achieved through the use of RFID or a video sensor tracking the patient's movement. When the application wishes to use location data it requests location and the ANS dynamically maps this to the context provider that best suits the quality required e.g. RFID. Further if that sensor should fail during this binding, the ANS will re-tender for another location service and the backup e.g. VIDEO can be used. The dynamism required for this situation is programmed in Tesserae, a language that allows the autonomic rules to be defined and the dynamic binding/rebinding at run time of the sensor component software (which again can be written in any language including native code). Tesserae and the sensor code components will typically run on the sensor processors themselves. This sensitivity to context and quality is what uniquely drives the autonomicity in the ANS architecture; through this simple mechanism we can achieve the lightweightness we require to provide the self-management in sensor networks aiming to achieve ubicomp *calm*.

The main focus of this paper is to illustrate how the ANS uses context and context provision in its autonomic intelligence. The next section, 3, discusses context and context awareness. It shows how Quality of Context (QoC) drives the self-management aspects of the ANS and its adaptation to change using utility functions. Section 4 describes how we can trust the QoC being reported; i.e. measure dependability in the architecture. Section 5 discusses Tesserae and our proof-of-concept architecture and devices. Comparing our work to related research can be found in section 6. The final section, 7, presents our conclusions and future directions.

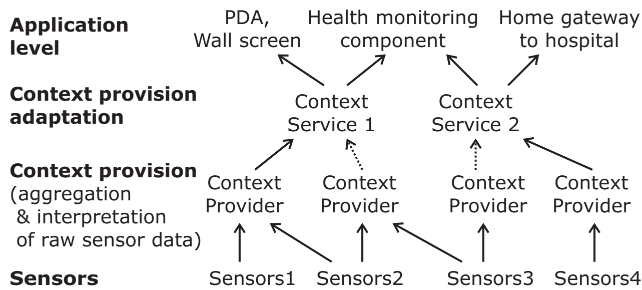
3. Autonomicity in ANS

3.1 Context Awareness

The perception of context and its quality essentially drives the autonomicity in ANS. Traditionally context-awareness is the ability of an application to adapt itself to the context of its user(s), whereby a user's context can be defined broadly as the circumstances or situations in which a computing task takes place [10]. One of the most common contexts is the location of the user (or of objects of interest). In smart-homes, location can be obtained using a variety of different alternative sensor types, including ultrasonic badges, RFID-tags, video cameras and even pressure sensors in the floor [7]. The quality (which is quantitatively application-specific) of the location information acquired by different sensors will however be different. For instance, ultrasonic badges can determine location with a precision of up to 3 cm, while RF lateration is limited to 1–3 m. Thus, we can define properties, which we call Quality of Context (QoC) attributes, that characterise the quality of the context data received. QoC is essential to the ANS in choosing the

best alternative among those available when delivering a specific type of context to an application.

Figure 3 Layers of ANS context provision



While different types of contexts will have QoC attributes specific to them, there are certain attributes that will be common to most contexts. Based on [2], we identify the following common attributes:

- *Precision* measures how accurately the context information describes reality, e.g. location precision.
- *Probability of correctness* (*poc*) measures the probability that a piece of context information is correct. For instance, determining the current posture of a person (sitting, standing, lying on the floor in distress) using a video camera has a different probability of correctness than using pressure sensors in the furniture. This metric differs from precision, as precision measures accuracy when a measurement is correct, whereas this metric describes the probability that the measurement is correct in the first place.
- *Resolution* denotes the granularity of information. This can for example mean spacial coverage. For instance, in a kitchen there can be hot spots with high temperature (oven, cooker, toaster), which may not be picked up by temperature sensors in the room if spacial coverage is low. Increasing the number of sensors in the room and optimising how they are spread can achieve finer spacial granularity.
- *Up-to-dateness* specifies the age of context information. Also of interest is how likely the measurement is still accurately describing the present.
- *Refresh rate* is related to up-to-dateness, and describes how often it is possible or desired to receive a new measurement. Different applications will have different refresh rate requirements. However, it is preferable to keep the refresh rate as low as possible whilst still delivering adequate context information, so as to minimise resources such as wireless network bandwidth whilst aiming to maximize the lifetime of battery-powered sensors.

QoC differs from the typical notion of Quality of Service (QoS), because context information has quality metrics even when it is not provided as a service to any clients. In the ANS, context providers need to specify QoC attributes for the context information they deliver. These attributes may vary over time and therefore must be updated regularly.

3.2 ANS Context structure

Figure 3 shows the layers in context provision in the general ANS architecture. At the bottom *sensors* deliver raw sensor data. These could be wireless sensor networks, ultrasonic badges for location, RFID tags for identification, video cameras for tracking, or others. These produce raw sensor data, often preprocessed to save communication cost as much as the (often power constrained) sensor devices allow. These data are passed up one level in the framework to the *context providers*. Context providers (CPs) are components (software or hardware) that aggregate and interpret the sensor data to produce some higher-level context, e.g. location, identity, type of activity, health condition of a person. As illustrated in Figure 3, it is possible for one CP to use sensor data from a set of different sensors, as data redundancy often improves the total reliability of the derived context[1].

Moving up one level, we have *context services* (CSs). These connect to different context providers that provide the same type of context, e.g. location, but implemented using different underlying sensors, and connect to the applications above. They allow an application to use a type of context whilst abstracting from the actual instance of a context provider.

Certain CPs and applications may use context information from a CS to provide higher-level context. For instance, a CS may provide location in the form of 2D or 3D coordinates in the home. A CP may then use this information to determine the room in the home this location maps to. Again, this higher-level context would be delivered through a context service.

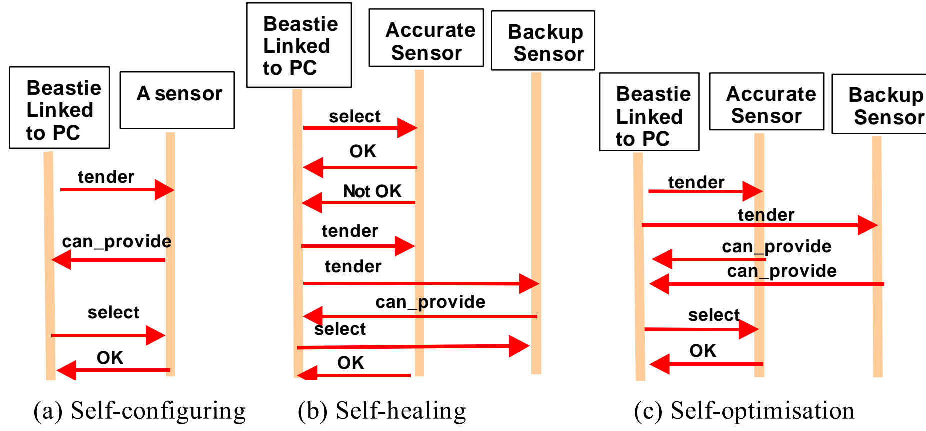
CSs execute adaptation in the system. This means switching between available CPs, e.g. when the QoC of the currently used provider deteriorates excessively or the QoC of an unused but available provider improves greatly (*self-optimising*). It also handles automatic failure recovery (*self-healing*) when the currently used provider is unable to deliver context, e.g. because of excessive sensor failure, and automatic upgrade to newly installed sensors and context providers (*self-configuring*). Thus, our framework implements adaptation through the basic use of a *virtual* abstraction layer. Evidently, QoC attributes are crucial metrics in adaptation decisions. In the next section, we describe how this adaptation process is decided and executed.

3.3 Adaptation

While service discovery is necessary in any service-oriented distributed environment, we tailor the service discovery protocol to provide efficient monitoring of Context Providers (CPs) for the purpose of adaptation. This includes

[1] This is more general than the approach in the Context Toolkit [12], where one sensor maps exactly to one context widget, and separate components called aggregators and interpreters collect and differentiate the sensor data.

Figure 4 Message Processing chart illustrating ANS Autonomic Protocol



(a) Self-configuring (b) Self-healing (c) Self-optimisation

rapidly discovering failure of CPs and updating the QoC of each CP. Section 3.4 mentions how adaptation is triggered, while Sections 3.5 and 3.6 explain how we determine the new CP to switch to.

3.4 Triggering adaptation

Many autonomic architectures contain an event-triggered adaptation engine that decides upon the adaptation externally by monitoring the status of CPs and notify events such as CP failure, a new CP has appeared, or of changes to QoC [8,6]. However this again is too centralised for our architecture, therefore each context service node in ANS decides whether an adaptation should occur in a passive way. That is, when it requests a retender for services, because either it has detected a lowering of the QoC of the currently bound CP or retendering is set to happen at regular intervals, the context service receives the up-to-date QoC for all the services that answer the request automatically. Therefore the concept of a heartbeat being sent around the system is also not necessary [13,6], thus saving bandwidth, radio and other resource heavy activities. The next section shows how the context service chooses which new CP to switch to.

3.5 Choosing the best Context Provider

To recap, an application makes use of a number of context services that in turn use one or more context providers. The application defines the minimum QoC required for correct functioning. When a CP, currently bound to a particular context service is no longer able to fulfil that QoC, the context service requests a retender, Figure 4 shows an example message diagram for this protocol. The retender is broadcast (or multicast to a region within an area using routing by attribute[2]) with the QoC it wishes to receive. Those that can provide the service to at least the same quality or close enough are bound to that service.

We define an application's satisfaction for a particular CP as a utility function that maps the CP's QoC attributes to a value that quantifies the application's satisfaction (where values ≥ 0 mean the application is satisfied with this CP and values < 0 mean the application is not satisfied, e.g. the service the application provides to the user suffers from the bad QoC of this CP). We chose to use utility functions over alternatives because this has been proven to provide enough knowledge to allow the system to discern between alternatives in a rich yet lightweight way. Further, they give us the flexibility to be able to fine-tune parameters and to adjust thresholds dynamically. Given each application's idea of the CP utility it requires in the form of its utility function, the adaptation engine on each node can then apply each application's utility function to each CP and select for each application the best CP.

For example, the utility function could define a reference point of the application's QoC expectations and then use linear distance (1-norm), Euclidean distance (2-norm), or return the maximum distance (maxnorm) between a CPs provided QoC and an application's QoC expectation. However, as we are combining numerical distance values of different properties, e.g. location precision in metres with refresh rate in Hz, we need to define a standard scale for all dimensions. The simplest way to do this is to introduce a scaling factor for each QoC attribute. That is, if a particular context type has n QoC attributes and let the application's wishes (i.e. reference point) for this context be $\vec{a} = (a_1, \ldots, a_n)$ (where we use $a_i = \otimes$ to indicate an application's indifference to the i-th QoC attribute), a CPs QoC $\vec{c} = (c_1, \ldots, c_n)$ (where we use $c_i = \otimes$ to indicate a CPs inability to provide a quantitative value for the i-th QoC attribute) and the scaling factors for the QoC attributes $\vec{s} = (s_1, \ldots, s_n)$, then we have:

Using $\vec{d} = (\vec{c} - \vec{a}) \cdot \vec{s}$

$$\left|\vec{d}\right| = |d_1| + |d_2| + \ldots + |d_n| \qquad \text{(1-norm)}$$

$$\left\|\vec{d}\right\|_2 = \sqrt{|d_1|^2 + |d_2|^2 + \ldots + |d_n|^2} \qquad \text{(2-norm)}$$

$$\left\|\vec{d}\right\|_\infty = \max\{|d_1|, |d_2|, \ldots, |d_n|\} \qquad \text{(max-norm)}$$

where $c_i - a_i = 0$ for $a_i = \otimes$

and $c_i - a_i = o(a_i)$ for $c_i = \otimes$.

$o(.)$ determines an application's satisfaction (or dissatisfaction) when the CP is unable to provide an estimate of a QoC attribute, given the value wished for by the application.

Actually, the various norms will not be applied directly as described above, the sign (satisfaction (+) vs. dissatisfaction (−)) of each dimension should be considered

[2] Routing by attribute is a mechanism, which allows us to direct packets to a given room or an individual's devices. This is not described in this paper and is a subject of a future paper.

as well as the final distance. For instance, it may only makes sense to add positive terms in the distance if there are no negative ones, i.e. extra quality in one dimension does not counteract the lack of quality in another dimension. In general, this definition implies that an application prefers it when it gets better QoC than it asked for, i.e. $c_i - a_i > 0$ potentially contributes to an application's satisfaction. For instance, an application that finds objects for a forgetful user will work better as precision of location information improves. However, there are cases where an application does not need and cannot use better QoC than it asked for, e.g. location precision in the case of a light control system that turns on and off lights when an individuals enter and leave rooms. In that case, d_i should be redefined as

$$d_i = \begin{cases} (c_i - a_i) \cdot s_i & \text{if } c_i - a_i < 0 \\ 0 & \text{otherwise} \end{cases} \quad \forall i = 1, \ldots, n$$

In other words, an application is satisfied (i.e. $d_i = 0$) when the i-th QoC attribute is at least as good as it wishes it to be and the distance of this attribute can only affect the total distance when there is dissatisfaction (i.e. $d_i < 0$).

Furthermore, up to now we have assumed that each QoC attribute is equally important. Since this is not generally the case, e.g. precision may be more important than refresh rate, applications may set weights $\vec{w} = (w_1, \ldots, w_n)$ to the n QoC attributes. Thus, we perform $\vec{d} \leftarrow \vec{d} \cdot \vec{w}$ before computing the final p-norm distance (or any form of distance).

One issue remaining is how to choose the scaling factor. In principle, we could determine the scaling factor by choosing a sensible scaling factor for each QoC attribute, based on an interval of likely values. However, this is tedious and subjective. We believe that a better way is to use a special case of the *Mahalanobis* distance, where we measure for every QoC attribute the standard deviation over all available values from CPs, and then express application distances, i.e. $\vec{c} - \vec{a}$, as multiples of the standard deviation for that attribute. In other words, given the standard deviation vector $\vec{\sigma}$, where σ_i is the standard deviation of the i-th QoC attribute over all CPs, the scaling factor $\vec{s} = 1/\vec{\sigma}$. This distance has the advantage of being independent of the scale used in the dimensions, i.e. using centimetres instead of metres to measure precision will not affect the numerical value of the distance. However, the range of values of the available CPs will affect the scaling factor of each QoC attribute.

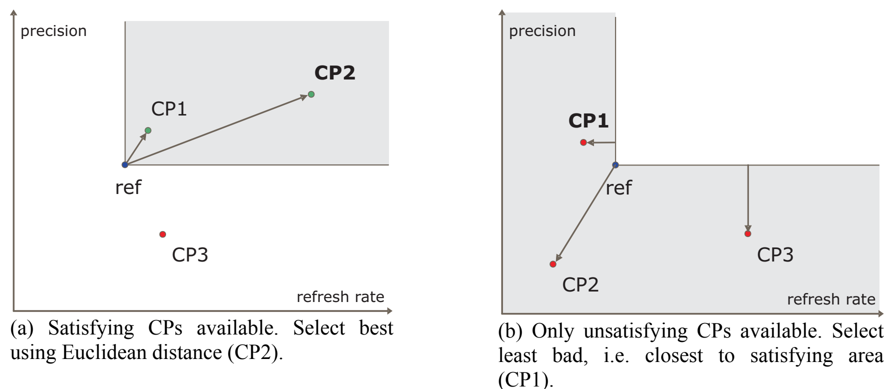
Simple utility functions could set independent thresholds on a certain number of QoC attributes, e.g. a certain amount of precision with a certain refresh rate is desired. However, more complex utility functions could use dependence between QoC attributes. For instance, for location, an application may be interested in a certain amount of precision. However, if refresh rate is good, then lower precision is also acceptable. This can be modelled as a decision tree. Also, applications can send updated utility functions at run-time, e.g. in response to context change or user input.

3.6 An example showing how a utility function can be used

In this section, we describe an example utility function that uses Euclidean distance in the QoC space for rating the satisfaction of an application with providers of location information. Much of the approach has already been described more abstractly in the previous section.

For simplicity in the description of the example, let us assume that the QoC space is defined by just two dimensions; the precision and refresh rate of a location service. Without loss of generality, we can also define that for every dimension a bigger number is better. Thus, we do not measure precision in metres, as a precision of 5 metres is not as good as a precision of 1 metre, but instead take the inverse of precision. Now, our example utility function defines a reference point in the QoC space that represents desirable values for the QoC attributes in a CP. Consequently, this point defines an area in the QoC space where every point represents a CP whose QoC attributes are at least as good as the reference point (see shaded area in Figure 5(a)). In other words, any point in this area (or hyperspace for the general n-dimensional case) will satisfy the application and will be preferred to points elsewhere in the QoC space. If the application does not need better QoC than the reference point, it can choose any CP in this area at random. This can be achieved by setting the same

Figure 5 Graphical representation of a utility function based on Euclidean distance.



(a) Satisfying CPs available. Select best using Euclidean distance (CP2).

(b) Only unsatisfying CPs available. Select least bad, i.e. closest to satisfying area (CP1).

satisfaction value for all CPs in the shaded area, whereby the middleware will then pick one at random. However, if the application wants the best available CP, then the utility function can return the Euclidean distance between the reference point and each CPs point in the shaded area. As a result, the middleware will pick the best alternative with respect to the reference point (CP2 in Figure 5(a)).

Actually, an application may value one dimension more than another. For example, a *frequently-lost-objects finder* application for a smart-home will value precision more than refresh rate, as determining the location of the lost object with best possible precision is the top priority, while refresh rate is of relatively little importance, as a lost object in the home is unlikely to move much. Instead, an automatic light control system will value refresh rate more than precision, since the lights should turn on in a room the moment a user approaches an entrance to that room, and not seconds after the user has entered the dark room. Yet, the precise location of the user at the time of measurement is not that important. Applications may therefore introduce a weight for each dimension before computing the Euclidean norm. This example also illustrates why utility functions and the notion of CP quality are application specific.

There may be a case where we only have CPs that do not entirely satisfy the QoC attributes of the reference point (Figure 5(b)). Again, there are various sensible approaches, but a possible option is to choose, from the unsatisfying CPs, the one that minimises the distance from the hyperspace of satisfying CPs. This is equivalent to saying that we ignore how good the good QoC attributes are (mathematically, we set their distance in the distance vector to 0), and look only at the attributes that don't satisfy the reference point; we then minimise the Euclidean distance of only these attributes from the reference point. For instance, in Figure 5(b), the distance of CP1 is determined only by refresh rate (since precision satisfies the reference point), for CP2 the distance is given by both attributes, and in the case of CP3 the distance is determined by precision. The final utility values are made negative, so when the middleware chooses the alternative with highest utility, it will pick the CP with minimal negative distance (CP1 in Figure 5(b)).

3.7 Dependability

We believe that among the descriptive attributes of a CP that are used as input to an application's utility function, there should also be a measure of the CP's dependability. In the ANS, we define the dependability of a CP as *the probability that, when it delivers context information, the quality of this information will match the descriptive attributes advertised*. Therefore, if a location precision of 10cm is advertised, but the actual location is 50cm from what is delivered by the CP, then the CP is being unreliable. However, a CP advertising a location precision of 100cm but delivering information within 50cm of the actual location can be considered dependable. Including dependability in the input of the utility function allows an application to choose how much risk it is willing to take in the hope of receiving good quality information. Dependability is different from all other descriptive attributes in that it cannot be determined by the CP itself (as it would choose maximum dependability of dp=1), therefore must be determined externally. We use a learning model that takes as input binary positive/negative feedback from context consumers and cross-validation with other CPs and feeds this feedback into a parameterised probability density function that is used to predict the CP's current dependability. The model allows for dynamic trust by keeping a window of recent feedbacks that affect the learning model. Thus, should the ratio of positive/negative feedbacks change over time so, too, will the predicted dp of the CP. This is described in more detail in [6].

4. Implementation

As mentioned in Section 2, we are building a proof-of-concept *intelligent home* in our laboratory to test the ANS under more realistic saturations. This will consist of a number of sensor devices, monitoring the patient's movements, gait, heartbeat etc. We present our hardware and software architecture here to give an indication of the sorts of devices this system will run on. Therefore the reader can immediately see that the use of traditional centralised autonomicity or more heavyweight AI techniques beyond that of utility functions would be much too heavyweight and prone to failure in this environment.

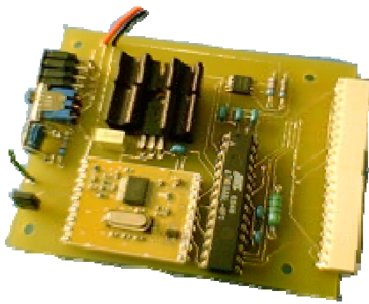
4.1 Hardware

Our sensor nodes are named Beasties (see Figure 6), and are connected via a wireless network to the other hardware devices such as screens and a PC. Beasties are small battery powered microcontroller devices with a wireless data link. They have a standard bus connector so that one or more peripherals can be easily added. Each Beastie has 8kBytes Flash program memory with 1k SRAM, 32kBytes of non-volatile FRAM, and 512bytes EEPROM - running at a clock speed of 4MHz giving approximately 4 MIPS (which can be upgraded to 8 MHz at the expense of increased battery usage). The radio is switchable between 433.93MHz and 434.33MHz FM, running at 20kbps maximum data rate with a maximum range of 200m. Ease of hardware modification and programming are key to the Beastie nature. The components that make up the Beasties are through-hole devices so that test probes can be easily attached and modifications made easily. Therefore sensors are then easily fitted to the Beastie e.g. video camera for gait detection where the detection and analysis component code runs on the Beastie microcontroller.

4.2 Components and Adaptation Architecture

The ANS architecture expects that the application code down to the sensor code be componentised. One of the great advantages of component-based software is its ability to adapt and change at runtime without increasing the complexity of the software or requiring standard routines to be rewritten. Furthermore, as only the components required

Figure 6 A Beastie Node



by the system are loaded at run-time, the footprint on the devices are as low as possible.

The intelligence that carries out the adaptation between components based on the perceived QoC is written in Tesseræ. The Tesseræ compiler is structured as shown in Figure 7. The Tesseræ compiler reads a "top-level configuration" file written in the Tesseræ language, loads component descriptions and implementations as necessary and runs implementation language compilers as sub-processes. The whole system is designed to be highly modular with standard interfaces between modules. This allows the system to be updated and improved one part at a time without affecting the rest of the system. The output of the Tesseræ compiler is a single monolithic program in the Aisle language.

The Aisle code passes through a code optimiser that does as much static evaluation as possible and removes unused code (components may include a large number of options and functions, only a few of which will be used at any one time). The optimiser also emits Aisle code. It is possible to embed system-specific native code into Aisle code; components that provide an abstract interface to hardware will use this feature to control the hardware.

The code generator is the only part of the compiler system that has to be changed when it is targeted to new hardware. This program takes Aisle code and produces native code ready to be fed into an assembler or compiler on the target system. It may also do further optimisation.

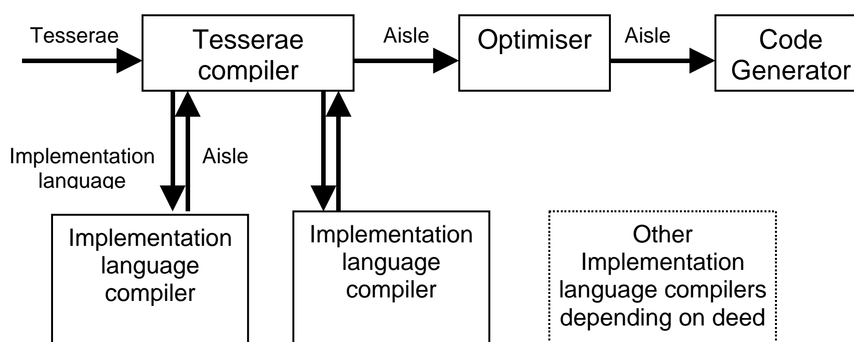
An important part of the Tesseræ design is that component implementations can be written in multiple languages (see figure 2). When the Tesseræ compiler comes across the start of an implementation function in a source file, it will connect to an appropriate compiler sub-process and pass it the source code of that function along with any metadata necessary to steer the compilation. The compiler will reply with a block of Aisle code representing the component. The compilers that may be used are specified in a configuration file and so new languages may be added to the system without any recompilation of the rest of the Tesseræ component toolset.

5. Related Work

There have been a number of attempts to build an 'intelligent home' in recent research. This is defined as "the integration of technologies and services, applied to homes, flats, apartments, houses and small buildings with the purpose of automating them and obtaining and increasing safety and security, comfort, communication, and technical management" [18]. Most of these have been using standard sensing components to obtain data from the inhabitants where the main focus of the research was to study the human-home interaction. Examples of these are; The Aware Home project, The Philips HomeLab and MIT's PlaceLab [15,16,17]. Possibly the closest project in terms of application is the Smart Medical Home hosted by the Centre for Future Health at the University of Rochester [19], which combines the use of a Medication advisor, Gait Analyser, Object locator and a mechanism for Sensing harmful bacteria. This work is complementary to the work we present here as their focus has been on the sensing components and the algorithms to process the sensor data, but not the placement of autonomic intelligence into the architecture, which we are doing. Therefore, as far as we are aware the ANS is the first specifically autonomic architecture for the 'intelligent home'.

However, the ANS can be used for more than the 'intelligent home' as its structure is a general autonomic architecture. Autonomic research activities can broadly be categorised into four areas: monitoring of components, interpretation of monitored data, creation of a repair plan (i.e. an adaptation of the system), and execution of a repair plan. Based on this, two categories of autonomic system architecture emerge: multi-agent systems and architecture design-based autonomic systems. The former consist of

Figure 7 Tesseræ compiler structure



standard agent-based technologies and do not have a centralised concept of autonomic intelligence (i.e. each agent has its own goals, social ability etc) – examples of such systems are [20, 21]. Alternatively, in the architecture design-based approach, the individual components are not per se autonomic. Instead, the infrastructure that handles the autonomic behaviour of the system and typically uses an architectural description model of the running system (which is not necessarily autonomic in itself) to monitor the running system, reason about it and determine appropriate adaptive actions. The adaptivity infrastructure is typically clearly separated from the running system and examples are [22,23, 8, 24] – the final one representing many of the architectures described in the IBM literature. Due to the ANS being highly constrained in terms of computing resources (i.e. sensor networks) we were unable to use either of these approaches. ANS is closer to the agent-based approach in that the intelligence is more 'emergent' within the system, distributed, but is not formally goal oriented as such.

In terms of the hardware and runtime nature of the ANS using Beasties and our Tesseræ language, the nearest work is that of the Berkeley Motes using the *nesC* language running in the *TinyOS* runtime environment [25]. Initially we considered using this architecture but soon found that the Mote environment, having been designed more for 'smart dust' style projects (where the nodes run homogenous sensing applications e.g. environmental monitoring), were unsuitable for our heterogeneous sensing applications requiring redundancy at each node. Dynamic reconfiguration is key to the ANS' ability to adapt on demand in a high-speed and in a lightweight way. Whilst *nesC* is similar to Tesserae in that it allows componentisation, it does not provide for dynamic reconfiguration at runtime.

The Context Toolkit inspired our use of context [3], however we extend their work by using the context to provide more general autonomic intelligence. That is, Context Toolkit is a framework aimed at facilitating the development and deployment of context-aware applications only [12]. It abstracts context services, e.g. a location service, from the sensors that acquire the necessary data to deliver the service. Thus, in a different deployment environment, the provider of a context type can be exchanged with another provider that may use a fundamentally different type of sensors, yet the application does not need to be modified to access this different source of the same context. However, there is no mechanism that allows context services to adapt and react to failure or degradation of the underlying sensor infrastructure, e.g. by switching to an alternative means of acquiring the same type of context. In fact, the situation where multiple means of acquiring the same context may be dynamically present in the system is not considered. We started with this assumption and then addressed the issues of self-management and adaptation. There have also been other middleware approaches to contextawareness middleware. For instance, [11 and 5] describe an infrastructure that provides context-awareness support to applications. It uses first order logic predicates to model contexts and allows deduction of higher-level contexts using rule-based approaches. Effectively, the application can delegate context-awareness to the infrastructure, which takes care of obtaining the context, evaluating the rules (provided by the application) and calling the actions. In contrast, in our solution applications delegate to the middleware the choice of most appropriate context provider. Bhatti and Knight [1] have proposed a QoS middleware for multimedia distribution in the Internet that also uses a utility function for determining the best alternative for delivering the media to the application, for instance choosing the most appropriate codec for audio delivery considering the current properties of the network, i.e. bandwidth and latency. However, the adaptation context of their middleware is quite different. QoS middleware typically address the issue of adapting the delivery of data across the network in response to changes in network properties. In our case, we are choosing the source of the data given various alternatives with varying quality (where quality is quantitatively an application-specific concept). Furthermore, our alternatives are not fixed, but are dynamically determined by the currently available context providers in the network and their (possibly dynamic) QoC attributes. This requires a service discovery protocol that reacts quickly to changes in the providers, so that the adaptation engine has an accurate view of the providers. Moreover, while Bhatti and Knight describe a fixed utility function for deciding adaptation, we use application-specific utility functions. An interesting design aspect in their middleware is how they engineer their utility function to prevent state-flapping, i.e. the situation where the network QoS state is near a utility function's threshold, causing the middleware to continuously flap between two adaptation states; a subject we are also examining.

6. Conclusions

Our goal was to introduce the notion of autonomic computing into a ubiquitous computing infrastructure, which we call ANS. While there has been much research on autonomic computing, it has not been specifically applied to ubicomp applications. Yet, particularly in a smart-home environment, self-management of the system is important because we cannot assume that there will be a technical administrator present round the clock to solve problems that may arise and fine-tune the system. Extending the notion of context-awareness, our proposed solution has assumed that services are described as context providers, which are able to feedback or estimate their Quality of Context. The self-management in ANS is governed by the maintenance of an overall satisfactory QoC. To do this ANS collects a fusion of discrete context data from multiple sources using a Bayesian network approach that takes into account the probability of correctness of the different sources and their dependability. We have proposed a trust model that allows the dependability (*dp*) of context information providers (CPs) to be determined through feedback from context

information consumers. This *dp* can then be used to evaluate, given multiple alternatives, which CP a consumer wishes to use. We believe this to be a better alternative to the traditional approach of using a trusted authority to certify the *dp* of CPs, as our *dp* approach reflects the recent behaviour of the provider and allows for dynamic *dp*. It remains to be seen whether these assumptions are reasonable in real pervasive computing environments and applications. To this end, we briefly described our Testbed environment in which we will deploy the ANS.

Although the ANS was designed in the context of smart home application scenarios and context-awareness, it can be generalised to the case where there are multiple service providers for the same service, e.g. printing services, and based on the applications' wishes we choose the optimal service provider for each application, e.g. nearest printer, best-quality printer. In general, we believe utility functions to be better suited to adaptive service provision than query-based service discovery, as they rank all available alternatives by their utility and allow the ANS to autonomically determine the best alternative on current service failure.

7. References

[1] S. N. Bhatti, G. Knight. Enabling QoS adaptation decisions for Internet applications. Computer Networks, 31(7):669–692, 1999.

[2] T. Buchholz, A. Kupper, M. Schiffers. Quality of context: What it is and why we need it. In Proceedings of theWorkshop of the HP OpenView University Association 2003 (HPOVUA 2003). Geneva, 2003

[3] K. Dey, G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction, 16:97–166, 2001.

[4] Garlan D., Monroe R. T., Wile D., 'Acme: An Architecture Description Interchange Language', In *Proc. of CASCON '97*, November 1997

[5] K. Henricksen, J. Indulska, A. Rakotonirainy. Modeling context information in pervasive computing systems. In Proceedings of the First International Conference on Pervasive Computing, pp. 167–180. Springer-Verlag, 2002. ISBN 3-540-44060-7

[6] M. C. Huebscher, J. A. McCann. Adaptive middleware for context-aware applications in smart-homes. In Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC). October 2004.

[7] J. Hightower, G. Borriello. Location systems for ubiquitous computing. Computer, IEEE, 34(8):57–66,August 2001.

[8] McCann J .A., Jawaheer G., Sun L., 'Patia: Adaptive Distributed Webserver (a Position Paper)' International Symposium on Autonomous Decentralized Systems (ISADS). April 2003

[9] McCann J .A. 'The Database Machine: Old Story, New Slant?' Proceedings of the first Biennial Conference on Innovative Data Systems Research, VLDB, January 5-8 2003

[10] S. Meyer, A. Rakotonirainy. A survey of research on context-aware homes. In Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003, pp. 159–168. Australian Computer Society, Inc., 2003. ISBN 1-920682-00-7.

[11] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, K. Nahrstedt. A middleware infrastructure for active spaces. Pervasive Computing, IEEE, 1(4):74–83, 2002.

[12] D. Salber, A. K. Dey, G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 434–441. ACM Press, 1999. ISBN 0-201-48559-1.

[13] Sterritt R., Bustard D. *Towards an Autonomic Computing Environment*. University of Ulster, Northern Ireland.

[14] M. Weiser. The computer for the 21st century. Scientific American, 265(3):94–104, September 1991.

[15] Kidd C. D, Orr R., Abowd G. D, Atkeson C. G, Essa . MacIntyre A, B., Mynatt E., Starner T. E, Newstetter W., The Aware Home: a living laboratory for Ubiquitous Computing Research. Proceedings of the Second International Workshop on Cooperative Buildings, Oct 1999

[16] Philips Research, Ambient intelligence in HomeLab, 2002, http://www.research.philips.com/Assets/Downloadablefile/ambientintelligence-2456.pdf,

[17] MIT PlaceLab, 2003, http://architecture.mit.edu/house_n/web/projects/PlaceLabNov11-2003.pdf

[18] Poulson D, Nicolle C, Galley M, Review of the current status of research on 'Smart Homes' and other domestic assistive technologies in support of TAHI, Loughborough University, Oct 2002, http://www.lboro.ac.uk/research/esri/smarthomes/

[19] Smart Medical Home, http://www.futurehealth.rochester.edu/smart_home/

[20] Kuo-Ming C., James A., Norman P.. *A Framework for Intelligent Agents within Effective Concurrent Design*. The Sixth International Conference on Computer Supported Cooperative Work in Design, 12-14 July 2001, Pages 338–343

[21] Wise A. et al. Using Little-JIL to Coordinate Agents in Software Engineering. In Automated Software Engineering Conference (ASE 2000), September 2000

[22] Garlan D., B. Schmerl. *Exploiting Architectural Design Knowledge to Support Self-Repairing Systems*. Proceedings of the 14th international conference on Software engineering and knowledge engineering. July 2002

[23] Valetto G., Kaiser G.. *A Case Study in Software Adaptation*. ACM, Proceedings of the first workshop on Self-healing systems. November 2002.

[24] Kephart J. O., Chess D.M.. *The Vision of Autonomic Computing*. Computer, IEEE, Volume 36, Issue 1, January 2003, Pages 41-50

[25] Gay D., Levis P., von Behren R., Welsh M., Brewer E., and Culler D., *The nesC Language: A Holistic Approach to Networked Embedded Systems*, Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.