MACE - Adaptive Component Management Middleware for Ubiquitous Systems

Mohamed Ahmed*
Department of Computer Science, UCL
Malet Place
London, WC1E 6BT, UK
m.ahmed@cs.ucl.ac.uk

Robert Ghanea-Hercock
BT Labs, Adastral Park
Martlesham Heath
Ipswich IP5 3RE, UK
robert.ghanea-hercock@bt.com

Stephen Hailes
Department of Computer Science, UCL
Malet Place
London, WC1E 6BT, UK
s.hailes@cs.ucl.ac.uk

ABSTRACT

If the hype is to be believed, we have come very close to the realisation of a ubiquitous computing environment. There are already a wide variety of devices, networking technologies and bespoke services; and yet the vision of anywhere anytime computing is proving somewhat elusive. Software abstractions and metaphors that were developed for desktop applications do not extend to ubiquitous computing. Because of the frequency of contextual changes and the paucity of resources, new distributed applications require much more flexible support for controlled reconfiguration, self-adaptation, and recovery of components.

We present a lightweight component management Middleware that provides flexibility by allowing design, deployment, and run-time reconfigurability. At design and deployment time, the developer can design a system by structuring software components according to a specific scenario. Then, at run-time, she can dynamically reconfigure the system, adjust to new environments, or dynamically add mechanisms that enables self-adaptation.

1. INTRODUCTION

In many ways, we have come very close to the realisation of ubiquitous computing. In developed societies, individuals often own several ubiquitous devices from PDAs and laptops, to mobile phones and GPS systems. There are a growing number of ubiquitous networks: (W)LAN/MAN (Ethernet & IEEE 802.11), GSM/GPRS/3G WANs, and, more recently, a growing set of PANs (Bluetooth, Zigbee, AudioNet etc.). A few ubiquitous services have been already been developed and deployed (for example, location-based services). In spite of this, the reality of development and deployment of ubiquitous computing applications is not in such good shape. Abstractions and metaphors of multiple-users/single-machine and single-user/single-machine scenarios do not extend to multi-user/multi-machine situations with a much stronger emphasis on autonomic machine-to-machine interaction. It is expected that, within the next ten years, the number of devices per person will increase by anything up to two orders of magnitude, but that the majority of such devices will be embedded and devoted to single purpose applications. However, such approaches miss out on a major revenue stream, in which a number of devices working together may provide services that cannot easily be provided by individual devices.

From the era of "complex devices and simple networking" we are moving to an era of "simple devices and complex networking". For us, ubiquitous computing environments imply the following:

- Heterogeneity within (i) the capability of systems and devices, (ii) the structure of systems with regard to their composition. Proposed systems are typically a loose coupling of traditional centralised networks providing the backbone infrastructure to support numerous small, lightweight, and often mobile components that provide sensory input, perform actuation and may interact in an ad-hoc manner.
- Behavioural heterogeneity in the operation of components and systems. This includes differences in their task, scope, autonomy, policies of interaction and the utilities gained from interacting.
- A variety of systems/devices with different scales and capabilities that must interact to support the required ubiquity. This raises two issues: (i) to realise the invisibility criteria, devices must necessarily become smaller and more tightly embedded in their environments. While current trends continue to forecast ever-increasing capacity in ever-shrinking devices, we can still safely assume that size will limit capacity [6]. (ii) The increase in the number and heterogeneity of components in an environment leads to a dramatic increase in the complexity of managing the environment. Centralised methods do not scale well and hinder the dynamism of systems [9].

  Further, constraints such as battery, computation power, memory, and sensing capabilities limit the amount of

*Contact Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MPAC '06, November 27-December 1, 2006 Melbourne, Australia
Copyright 2006 ACM 1-59593-421-9/06/11 ...$5.00.

useful work that a device is able to perform locally, thereby severely affecting the scope of solutions enabling software subsystems such as security [14]. With respect to this, Estrin et. al. remark: “Fidelity and availability will come from the quantity of partially redundant measurements and their correlation, not the individual component's quality and precision” [6].

- The embodiment of components in their environment is among the main requirements to achieve ubiquity. Embodiment means that components are embedded in their operational environment, and therefore physical access is limited. Issues such as the update of functionality or management policies cannot be performed easily, further highlighting the need for autonomic and self-regulating capabilities.

In the following sections we outline MACE (Middleware for Adaptive Component managEment). MACE is aimed at addressing the challenges listed, through a hybrid agent-component approach. This approach is taken in due recognition of the inherent trade-offs between flexibility and fault-tolerance. For example, though increased flexibility is reflected in the capacity to reconfigure a system, it makes it more difficult to reason about its performance and control, particularly when the likelihood of failure is non-trivial. Therefore, our aims for MACE are to support the design, deployment, and run-time (re)reconfigurability of systems, while still maintaining fault-tolerance.

MACE implements a distributed component model in which components may be local or remote and supports synchronous and asynchronous communication. This is used to provide a high-level of modularity and flexibility that enables fine-grained control over the behaviour of MACE-based system at the interface level. For example, components can easily monitor how their functionality is being used and who is using it, paving the way for fine-grained policy-managed context adaptation and security.

The remainder of the paper is organised as follows: section 2 presents an overview of the system architecture. Section 3 discusses the usage of the system, provides additional details of its functionality and discusses potential security issues. Section 4 reviews related work and finally, section 5 concludes this paper.

2. SYSTEM ARCHITECTURE

MACE is a component framework and methodology aimed at supporting flexibility and fault tolerance. At the most abstract level, MACE simply enforces that components provide interfaces and receptacles that are respectively used to specify their functionalities and dependencies, and that are chained to compose systems.

Given the requirements listed for ubiquitous environments, software must be capable of being both robust and extremely flexible in order to realise the functionality envisioned. In support of this, MACE inherently modularises software design, allowing software developers to construct application-level software from distinct components that each provide a subset of the functionality required by an application. To do so, MACE employs four *component profiles*, in which higher-level profiles extend the functionality provided by of lower-level profiles, as depicted in Figure 1.

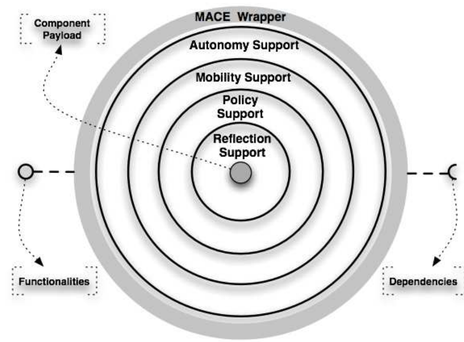


Figure 1: The hierarchical arrangement of MACE component profiles.

The component profiles that MACE implements are as follows:

1. **The Basic Component** profile consists of the *component payload*, which provides the real functionality and the *MACE wrapper*. This profile simply requires that components supply their dependencies and functionalities (for the resolution and execution of services), and incurs a minimal overhead.

   The profile is primarily used to enable components to participate within an application and it is catered towards the lowest level devices. As such, within a component system, this profile's function is simply to process services and make use of higher profiles to support extended functionality.

2. **The Policy-Managed Component** profile adds support for adaptive/autonomous behaviour through a policy enforcement module. With regard to security considerations, policies dictate the runtime behaviour of a component by dynamically hiding and revealing the functionalities (and associated dependencies) of a component, based on the context or condition, or by imposing constraints on execution flows.

   Policies are also used to perform context adaptation. In this role, the rules in the policy dictate the conditions for adaptation; for example, when to migrate a component from one host to another or initiate service discovery.

   Policies are defined outside of the component and statically compiled to check that they are well-formed. In our case we focus on detecting cycles and violations of non-monotonic constraints. A detailed discussion of this part of the work can be found in [1].

3. **The Mobile Component** profile adds the support for physical mobility to the basic component profile. Because there is no specific runtime environment for MACE, mobile components also provide the hosting environment for MACE components to support the boot strapping of systems.

4. **The Autonomous Component** profile adds significant capabilities to the *basic component* profile by introducing high-level reflection in a white pages listing, publish-subscribe support, and application consistency

checking in the form of dependency cycle detection. This profile essentially adds the features required for a component to manage itself independently.

High-level reflection is used to permit a component to be aware of other components with which it interacts. In conjunction with a service discovery component, this profile is used to support autonomous service discovery and component management functionality. For example, the Middleware we have developed using MACE [13] uses a Component Manager (CM) derived from the autonomous component profile, to provide data routing, service discovery/resolution and high-level reflection support. In effect, this profile transforms the component into an autonomous agent, capable of perceiving its environment, and state and acting independently.

5. **The Mobile Autonomous Component** profile provides mobility and hosting support for the autonomous component profile, and is essentially a template for a mobile agent.

2.1 Requirements Analysis

MACE implements a weak logical mobility model and aims to provide a highly modular, hybrid agent-component based abstraction to designing and implementing software. MACE simplifies the "componentisation" process by aiming to act simply as a wrapper - requiring only knowledge of the dependencies and functionalities of a component. This information is also used explicitly to support component reflection and to resolve the dependencies of components.

To see the properties of MACE, we expand on how it addresses each of the requirements mentioned for component models in [4, 18]:

Modularity is at the heart of the MACE model, with the aim of enabling and maintaining independence between the components that are used to build a service. Modularity enables: (i) The separation of concerns and functionality between modules that make up an application, thereby easing the software development and maintenance cycle [15]. (ii) Support for fine-grained management of policy and context management.

Wide applicability of the framework is achieved by fully decoupling the components that are used to make up an application. MACE components are referred to as network objects inhabiting a given location (local or remote) and all interaction between components at all levels is handled through well-defined RPC interfaces. We have adapted the XMLRPC [17] and, as a consequence, the payload of all method calls is plain or binary XML.

Through the use of the RPC we are able to abstract from any specific requirements on runtime environments, because only basic strings are transferred and object representation is constructed locally. Secondly, XML provides the capacity to use schemas to translate between heterogeneous environments and local component representations. This precaution eases the deployment of components in heterogeneous environments, by removing language and platform interdependence between components.

Separation of application and policy is achieved through the use of local configurations. These perform two functions. (i) They state the runtime requirements of components such as library dependencies or networks locations. (ii) They are used to generate the component run-time logic based on policy requirements.

Support for runtime reconfiguration is achieved mainly through the modularisation of functionality. MACE provides a clear separation between system building and system management. For example, MACE components may be removed from or added to running system.

Because of the lack of a heavyweight binding procedures such as class loading, components always remain independent of each other and can be modified easily, without requiring any further modification to their connections - of course any reconfiguration requires that existing interfaces be respected.

Component Managers (based on the autonomous component profile) are used to provide high-level management (see Figure 3.1*b* and *c*). These act as autonomous agents utilising a publish-subscribe methodology to provide a global vantage point of the systems they manage and can be used to support a holistic application view and control - for example, as is required to manage an applications life-cycle.

Selective transparency to the deployment environment is achieved through the loose component binding that results from the use of an RPC based mechanism. This means that components are free to address the advantages or constraints of their target environment, provided they still respect the calling conventions.

High performance of systems is always a necessity; in MACE we have completely decoupled the application payload and the component mechanism; therefore MACE works as an "application wrapper". Some performance cost has been incurred by the extra marshaling procedures required by an RPC. However, since the systems that are likely to suffer from this overhead will tend be those involved in heavily interactive behaviour; it is not expected that these components will be required on the lowest capacity devices.

Security and adaptability for MACE is addressed through a policy-driven behaviour module. This is a layered approach designed to be easily extended - to support application-specific requirements.

In our current implementation, we use shared keys to support secured interaction between components and policy enforcement may be built into the application through use of the policy-managed profile. The policy module allows for the application's control logic to be auto-generated from a high-level description at the point of deployment.

3. SYSTEM FUNCTIONALITY

MACE has been used to develop the components for the ADAM Middleware [13], as well as a light weight VPN [12]. In this section, we demonstrate how to use MACE in development of component-based systems as well as providing

additional insights into its operation. Standard software engineering techniques are used to provide a component structure [15]. Once the application payload has been designed, there are two ways to proceed: (i) by extending the appropriate component profile and defining the functionalities/dependencies of the component, or (ii) defining a policy for the component and using our policy module to auto-generate the appropriate component wrapper.

3.1 Initialisation

During system initialisation, it is assumed that each component is able to find the initial location of the component manager (CM), as defined in its configuration file. In the case of the autonomous component profile, the component comes with its own CM. The initial configuration may also include public keys that are used to verify components.

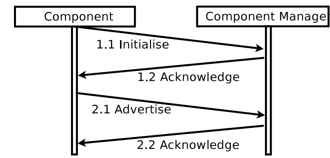
When a component is initialised (see Figure 3.1*a*) it contacts the CM (1.1). If the component manager is on-line it returns an acknowledgement that it is ready (1.2). If the acknowledgement is not received, within a certain time interval (specified in the component's policy definition), or an error message is received, the component launches the recovery behaviour defined for the event, e.g. try to reconnect later.

If the CM is, however, on-line and ready, the component sends its description (2.1) including its physical location, functionalities and dependencies (in XML). This information is used by the CM to build a service description list that is then used to advertise the functionalities of components and resolve dependencies between components. Each record in the description list contains a service name, the interface to access this service, and physical address of the service (or the component that provides this service). If all goes well again an acknowledgement message is returned to the initiating component (2.2), otherwise an event is raised and the component is free to initiate its own recovery behaviour, for example subscribing to be notified when the functionality it requires becomes available or attempting to reconnect at given intervals.

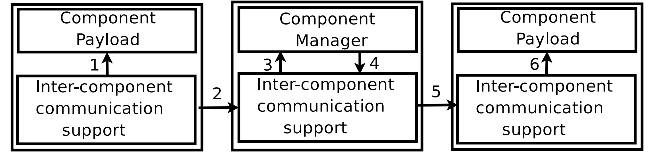
Figure 3.1*b* demonstrates the basic task execution procedure in MACE. The CM always acts as a proxy, providing a redirection service and enabling a global view of the system. Consider the following example, during the execution of task T_1 "Component 1" needs to execute task T_2. Being aware of the interface for task T_2 "Component 1" creates a request to execute task T_2 (1). Using the API provided, this request is encapsulated in an XML message and submitted to the CM (2). The CM uses the request name and its parameters to identify the appropriate task provider/interface (using its service description list) (3).

Once the destination interface has been located, the CM re-writes the the destination of the request (4) and forwards it (5). Once the request reaches its final destination, it is decoded and executed (6). When the execution finishes, the results are returned to "Component 1" in the same way.

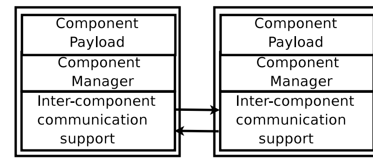
To address scalability problems introduced by have a centralised CM, MACE-based systems are also able to function in an decentralised mode using the Autonomous Component profile, as shown in Figure 3.1*c*. The profile enables components to embed the CM functionality, thereby enabling them to manage their behaviour independently of the rest of the system. The process of interaction here is no different to the basic component model, except that the CM functionality



(a) Component registration



(b) Basic component interaction



(c) Autonomous component interaction

Figure 2: Component initialisation and task execution behaviours

resides within the component.

Though the autonomous component model offers the capacity for an application to be constructed of components that reside at or are controlled by autonomous domains (without direct access to each others code), its cost is a loss in the capacity for a holistic management of the overall system; since each component is responsible for managing its own functionality, only local views are available to it. This problem can be addressed by using the publish-subscribe capacity of MACE, to enable components to cooperate in sharing a top-level view of the whole system.

3.2 Local Configuration

The behaviour of components is controlled through local configurations that dictate the policies that apply to a component. The policy provides three important services. First it lists how events raised by a component are handled; this includes requests for services or notifications from the environment. Second, it provides a limited key service, to support the bootstrap process. Last, it provides support for the policy independence clause, through listing the functional requirements of a component.

MACE uses policies as reactive strategies to events that occur in the system or environment. Therefore all calls to the exposed functionality of a component are treated as an event. The policy module works by creating MACE wrappers for the component payload based on the policy specification. For example, to control the behaviour of the hosting capability of a component, we may place restrictions (as shown in Listing 1) that determine when to permit a com-

ponent to be hosted.

Listing 1: "A simplified high-level policy specification"

```
on <boolean> LoadComponent(<CompStruct> component):
    preconds:
        authorised(component) == True
        canLoad() == True
    action:
        $loadComponent(component)
    postcond:
        $addComponent(component)
```

The advantage of this method is that it simplifies the component behaviour logic by (i) auto-generating it and (ii) separating it from the functional specification of the component. It also makes it possible to statically check the policy definition for cycles and contradictions or against predefined constraints before deploying a component. Components are ready for deployment once their policies are defined and the component wrapper is generated.

If, however, the policy definition or the functionality of a component changes during runtime, the component needs to be updated - this involves recompiling and redeploying the component. This is not a functional requirement of our approach, but a limitation of using a compiled rather than a purely interpreted language.

3.3 Security Considerations

Security of communications and fault tolerance are vitally important for distributed systems. MACE employs symmetric cryptography for authentication and for securing internal communications. The lengths of keys and their usage may be chosen statically by application developers or dynamically by applications themselves, depending on available resources and current application requirements.

For distributed systems such as MACE, the issue of denial-of-service (DoS) is not an unreasonable concern; nodes may be easily compromised, especially if they are constrained devices. To mitigate such risks, we have designed a simple DoS detection mechanism that monitors the rate of failed calls. Once a DoS attack has been detected, its source may be blacklisted or the Middleware may physically relocate itself by moving to another node. Naturally, the attacker may forward the attack to the new location, but this location need not be obvious, giving sufficient time for the component manager to report the attack and continue normal operation from its new location until attacked again. This "chasing game" scenario would decrease performance of a distributed system, but not render it useless. We are currently involved in work that addresses this issue [16].

3.4 Implementation and analysis

To ensure that MACE remains lightweight, it has been prototyped in the J2ME CLDC profile using a customised XMLRPC. The *autonomous mobile component* profile (our most heavyweight profile) plus all its support libraries occupies less than 40K. The payload of all method calls is "text/xml" tuples in the form *¡methodName, params¿* (responses are simply parameter lists, see [17] for more details) and simple response of `''Hello, world''` incurs a cost of 137 Bytes.

With regard to performance efficiency, the majority of our cost is incurred at the RPC. This is reflected in both the size of the data transfered when making calls (due to the verbosity of XML), and the required marshalling of the data. Though this makes our approach computationally slower compared to local binding approaches [3, 18], it offers both language and platform independence as well the extra fault tolerance afforded by the complete decoupling of the components.

In essence, the methodology of MACE aims to address the issues raised by the heterogeneity envisioned in Ubicomp environments through simplicity, robustness and flexibility. MACE heavily emphasises component modularity, opting for an RPC based approach rather than an Object Oriented one, so that components and processes are always independent of each other. This means that failure in one process cannot cascade and affect the running of its dependents or dependencies. Further more, MACE-based systems can easily recover by invoking their policy based routines. Controlled (re)configuration is supported through the use of local configurations that separate system building from management, whilst policy-based management allows for rule-based self adaptation.

4. RELATED WORK

Amongst others, K-Components [5], OpenCom [4] and SATIN [18] are examples of modular platforms for building context-adaptive applications. K-Components emphasises the role of connectors as "first class" entities in support of reflection and provides a description language for specifying component adaptation rules, but not the mechanisms to control or reason about the satisfiability of the rules. OpenCom and SATIN provide local component models and, respectively, focus on the specification and primitives for interfaces, receptacles and logical mobility. The RUNES [3] model builds heavily on this work, generalising the approaches for a finer grained component framework.

The OSGi [11] framework and MACE share similar aims, but the standalone OSGi specification is more catered towards deploying services in centralised environments, and supporting the vendors "application life-cycle" management in a transparent manner. As such, OSGi components (referred to as bundles) are self contained (there is no support for distributed components) and this is reflected in OSGi's management of dependencies, whereby management and resolution are left to the component programmer.

These approaches differ from ours in their use of local component models and their Object Orientated viewpoint. The focus is on creating generic APIs that provide the functionality for interoperability and reuse, as opposed to our focus on differentiated capacity and commonality in data and information exchange. The advantage of this is most clearly demonstrated in MACE's tolerance to failure. For example, though failures along the dependency chain affect the operational functionality of components, the components are themselves untouched and can easily detect the failure of their calls. In contrast, local binding means that failures in the process threads of coupled components can lead to cascade effects - unless explicit measures are taken to separate component threads and explicitly detect and handle runtime errors.

CALMA [2] addresses limitations for Ubicomp systems such as perception, planning and mobility, by introducing an intensional stance mechanism to help reason about the state of the environment. However, the work is very much centred

on the client server model, whereby agents refer to more capable servers to do the work on their behalf. This is also the case for [10] where the FIPA approach provides flexibility but the platform is tightly coupled and heavyweight.

Aspect Orientated approaches such as [8] and [7] present an approach to constructing components from aspects that represent the various functionality of the code-base. However, this abstraction does not fully simplify the problem - much like the "componentisation" issue, designers are still left to decide what constitutes an aspect and how to decouple application functionality to produce them.

5. CONCLUSIONS AND FUTURE WORK

MACE is aimed at simplifying the development of distributed applications for ubiquitous environments. It allows the organisation of a distributed system in a number of sufficiently small components to enable its operation in networks that include resource-constrained devices. MACE is not tied to particular network type or topology and the RPC protocol may be modified without loss of generality. Components are allowed to migrate dynamically between network nodes. By supporting asynchronous communications between components MACE can easily tolerate its components going offline without completely losing its functionality.

If a component fails, or a node on which it resides is disconnected, the system administrator may easily instantiate the component on another node. If permitted, the Middleware can autonomously adapt to such situations, thereby supporting graceful recovery from failure. MACE has been designed to be small and easy to use while still providing implicit support for flexibility and fault-tolerance.

To extend this work, we are looking into (i) addressing the performance deficiencies of the RPC mechanism, (ii) investigating the policy issues (with regard to security) raised by the extra granularity and autonomy introduced by the component system, and (iii) finally providing full implementations of the API in a variety of languages.

6. ACKNOWLEDGEMENTS

The authors would like to thank BT for funding under the MARS project and the EC for funding under RUNES.

7. ADDITIONAL AUTHORS

Rae Harbird (`r.harbird@cs.ucl.ac.uk`) and Alexndr Seleznyov (`alexandr.seleznyov@nokia.com`).

8. REFERENCES

[1] M. Ahmed and S. Hailes. A policy-based management framework for networked embedded systems: The runes approach. Technical report, Department of Computer Science, University College London, 2006.

[2] S. H. Chuah, S. W. Loke, S. Krishnaswamy, and A. Sumartono. Calma: Context-aware lightweight mobile bdi agents for ubiquitous computing. In *Workshop on Agents for Ubiquitous Computing, in conjunction with AAMS 2004.*, July 2004.

[3] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany), Sept. 2005.

[4] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. Opencom v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, Cambridge, MA, USA, November 2004.

[5] J. Dowling. *The Decentralised Systems Coordination of Self-Adaptive Components for Autonomic Computing Systems.* PhD thesis, Department of Computer Science, Trinity College Dublin, 2004.

[6] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[7] P. Falcarin and G. Alonso. Software architecture evolution through dynamic aop. In *EWSA*, volume 3047 of *Lecture Notes in Computer Science*, pages 57–73. Springer, 2004.

[8] L. Fuentes and D. Jimènez. An Ascpect-Orientated Ambient Intelligence Middleware Platform. In *MPAC'05*, 2005.

[9] T. Grandison. Trust specification and analysis for internet applications. Technical report, 2001.

[10] M. W. Michael Pirker, Michael Berger. An approach for fipa agent service discovery in mobile ad hoc environments. In *Workshop on Agents for Ubiquitous Computing, in conjunction with AAMS 2004.*, July 2004.

[11] OSGi Alliance. About the osgi service platform, revision 4.1. Technical report, 2005.

[12] Z. Sarmadi, M. Ahmed, and S. Hailes. Light weight component-based VPN . Technical report, Dept. of Computer Science, University College London, London, UK, 2005.

[13] A. Seleznyov, M. Ahmed, and S. Hailes. *Intelligent Spaces - an Application of Pervasive ICT*, chapter Co-operation in the Digital Age - Engendering Trust in Electronic Environments. Kluwer, 2005.

[14] F. Stajano and J. Crowcroft. The butt of the iceberg: hidden security problems of ubiquitous systems. Technical report, Norwell, MA, USA, 2003.

[15] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Boston, MA, USA, 1999.

[16] C. Wallenta. Detecting malicious activities in directed diffusion based sensor networks. Master Thesis, System Architecture Group, University of Karlsruhe, Germany and Dept. of Computer Science, University College London, UK, Sept. 2006.

[17] D. Winer. Xml-rpc specification. Technical report, UserLand Frontier, http://www.xmlrpc.com/spec, 1999.

[18] S. Zachariadis. *Adapting Mobile Systems Using Logical Mobility Primitives.* PhD thesis, Department of Computer Science, University College London, University of London, May 2005.

APPENDIX

A. EXAMPLE LOCAL CONFIGURATION

Listing 2: "A sample config file for a component"

```
component_name = SumComponent
component_host_name = localhost
component_host_port = 2006
component_key = 1024 01:c1:ba:34:51:ac:7c ... RSA
component_res_path = dist/repository
component_jar_depends = comSys-1.0.jar, ...
component_exec = java -jar
component_manager_name = comSysManager
component_host_name = localhost
component_manager_host_port = 2004
...
```