

---

## **Context-driven migration and diffusion of pervasive services on the OSGi framework**

---

Davy Preuveneers\* and Yolande Berbers

Department of Computer Science,  
Katholieke Universiteit Leuven,  
Celestijnenlaan 200A, Leuven B-3001, Belgium

Fax: +32 16 327996

E-mail: [davy.preuveneers@cs.kuleuven.be](mailto:davy.preuveneers@cs.kuleuven.be)

E-mail: [yolande.berbers@cs.kuleuven.be](mailto:yolande.berbers@cs.kuleuven.be)

\*Corresponding author

**Abstract:** The ubiquity of wireless *ad hoc* networks and the benefits of loosely coupled services have fostered a growing interest in service-oriented architectures for pervasive computing. Context-aware services and runtime adaptation are key to expedite human interaction with dynamic pervasive computing environments. We explore two types of live service mobility to avoid service disruptions in mobile *ad hoc* networks. Service migration moves a service at runtime from one host to another including its state, while service diffusion replicates the service and the state on multiple hosts. We analyse the basic requirements for service mobility and evaluate our OSGi-based implementation for service mobility with real life experiments. Our results show that the overhead of state transfer and synchronisation is limited for relatively small applications and that the delay for handing over to a replicated service in a small scale network with enforced network failures remains minimal.

**Keywords:** adaptation; context; mobile and pervasive services.

**Reference** to this paper should be made as follows: Preuveneers, D. and Berbers, Y. (2010) 'Context-driven migration and diffusion of pervasive services on the OSGi framework', *Int. J. Autonomous and Adaptive Communications Systems*, Vol. 3, No. 1, pp.3–22.

**Biographical notes:** Davy Preuveneers received an MSc in Computer Science in 2002 and an MSc in Artificial Intelligence in 2003 from the Katholieke Universiteit Leuven in Belgium. Since 2003, he is a PhD Student and Research Assistant in the DistriNet Research Group at the Department of Computer Science in the Katholieke Universiteit Leuven. His research is in the field of middleware and service-oriented architectures for context-aware service interaction and adaptation, in particular in mobile and ubiquitous computing environments.

Yolande Berbers received her MSc and PhD in Computer Science from the Katholieke Universiteit Leuven in 1982 and 1987, respectively. Since 1990, she has been an Associate Professor in the Department of Computer Science at the Katholieke Universiteit Leuven and a Member of the DistriNet Research Group. Her research interests include software engineering for embedded software, ubiquitous computing, service architectures, middleware, real-time systems, component-oriented software development, distributed systems, environments for distributed and parallel applications and mobile agents.

She runs several projects in cooperation with other academic partners and/or industry. She teaches various courses on programming real-time and embedded systems, and on computer architecture.

---

## 1 Introduction

The late Mark Weiser (1991) envisioned ubiquitous computing as the next logical step to the era of the traditional desktop as the mainstream computing paradigm. Smart objects that are invisibly embedded in the environment around us will interact with one another and adapt to the situation at hand without any human involvement. Service-oriented computing (SOC) (Papazoglou and Georgakopoulos, 2003) is a key enabler of Weiser's vision. It represents the current state-of-the-art in software architecture (Shaw and Garlan, 1996) that utilises services as fundamental building blocks for the rapid development and deployment of applications. It relies on a service-oriented architecture (SOA) to organise loosely coupled services, to bind them and manage their life cycle. By breaking up an application into an orchestration of independent reusable services with well-defined interfaces, SOA facilitates the creation of new intelligent applications. It provides the required flexibility to customise applications to highly dynamic environments by adapting the service deployment and composition to the new context.

The focus of this paper is on how we can improve the availability and accessibility of services in a highly dynamic *ad hoc* network where intermittent network connectivity can disrupt an application when multiple services on mobile and stationary devices are temporarily coupled to behave as one single application. We propose to replicate the same service and its state at runtime (i.e. live mobility of services) on multiple devices near to the user that can provide a guaranteed quality of service (QoS) with support for state synchronisation between the replicated services. This way, we increase the number of opportunities a user can interact with a service, and we can better deal with volatile network connections by handing over to a replica if the connection between two services breaks down. As such, we say that the same service has *diffused* to multiple hosts. We will illustrate the applicability of smart service diffusion in a distributed setting on top of the OSGi (Open Services Gateway Initiative, 2007) framework.

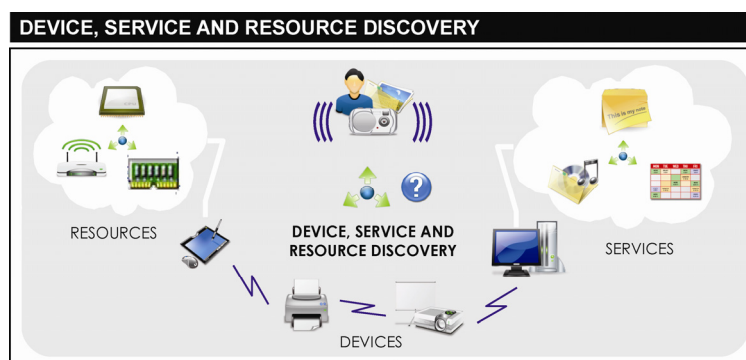
Service mobility presumes that services are distributed in a network on which they can be discovered and in which they can interoperate seamlessly. OSGi initially did not have any extensions for transparent distributed services. However, transparent remote service invocation on the OSGi framework is an issue that has been addressed by several researchers and projects. The Eclipse Communication Framework (The Eclipse Foundation, 2007) provides a flexible approach to deal with remote services and service discovery on top of the Equinox OSGi implementation. Remote OSGi services can be called synchronously and asynchronously on top of various communication protocols. The R-OSGI (Rellermeier et al., 2007) is a similar framework to deal with distributed services.

The previous two projects address network transparency for OSGi services, but did not deal with live service mobility. We aim to replicate a service at runtime on one or more hosts to increase accessibility and expedite human interaction with the service. This means that we need to redeploy the service and transfer the state (Kramer and Magee, 1990) of the service, and keep the state synchronised when diffusing a service to

multiple hosts. The Fluid Computing (Bourges-Waldegg et al., 2005) and flowSGI (Rellermeyer, 2006) projects explicitly deal with state transfer and discuss state synchronisation mechanisms and ways to resolve state conflicts. Optimistic replication is a technique to enable a higher availability and performance in distributed data sharing systems. The advantage of optimistic replication is that it allows replicas to diverge. Although users can observe this divergence, the copies will eventually converge during a reconciliation phase. In Saito and Shapiro (2005), the authors provide a comprehensive survey of techniques developed to address divergence. Because conflicting states break transparent handover to a replicated service, our approach to service replication and state synchronisation follows the more traditional pessimistic replication and aims to prevent divergence.

In order to keep the service diffusion approach scalable, the targets selected for diffusion need to be well-chosen. That is why we use context-awareness (Dey, 2001) as a way to improve the target selection for service migration and diffusion. Information about the characteristics of the services and the devices in the vicinity helps to select appropriate targets for service mobility. By keeping track of the service context, we also learn for each service the devices and the user will most likely interact with. As a result, compared to the Fluid Computing and flowSGI initiatives, our method enhances the peer selection for service replication by using context information of the service and the device. This approach provides better guarantees that the application will actually work and be used on the device. Implementing context-awareness with OSGi has been investigated in the past. In Gu et al. (2004), the OSGi platform was used to perform context discovery, acquisition and interpretation and relied on an OWL ontology-based context model that leverages semantic web technology. In Yu et al. (2006), the authors present an OSGi infrastructure for context-aware multimedia services in a smart home environment. Although the application domain our target goes beyond multimedia and smart home environments, there are other context-awareness systems (Kim et al., 2001; Lee et al., 2006) leveraging the benefits of the OSGi framework to implement context-aware behaviour (Figure 1).

**Figure 1** Smart diffusion of services requires awareness on the availability of devices, services and resources that surround the user (see online version for colours)



Note: In this illustration, a camera is looking for a device with adequate resources that can show its pictures on a better display.

The paper is structured as follows. In Section 2, we list the requirements to enable seamless service migration and diffusion. Section 3 provides details on how we realised these requirements on top of the OSGi framework. In Section 4, we conduct experiments that illustrate the effectiveness of service diffusion in a real life scenario. We measure the overhead of state transfer and synchronisation as well as any benefit in improved availability and accessibility. We end with conclusions and future work in Section 5.

## **2 Requirements for service migration and diffusion**

Pervasive services offer a certain functionality to nearby users. They are accessed in an *anytime-anywhere* fashion and deployed on various kinds of mobile and stationary devices. When users or devices become mobile, live service mobility can provide a solution to the increased risk of disconnecting remote services. In this section, we review several non-functional concerns and requirements that need to be fulfilled to ensure that the migration and diffusion of a service in a mobile and pervasive setting can be accomplished.

### *2.1 Device, service and resource awareness*

In a pervasive services environment, the multiuser and multicomputer interaction causes a competition for resources on shared devices. Therefore, knowledge about the presence, type and context of the devices (including resource-awareness about the maximum availability and current usage of processing power, memory, network bandwidth, battery lifetime, etc.) are prerequisite to guarantee a minimum usability and QoS. Before relocating a service to another host, a service discovery protocol (SDP) can be used to verify if the service is not already available (Helal, 2002).

### *2.2 Explicit service state representation and safe check-pointing*

Stateless services can be replaced with similar ones or redeployed on another host at runtime without further ado as long as the syntax and semantics of the interfaces remain the same such that the binding can be reestablished. Stateful services require a state transfer after redeployment before they can continue their operations on a different host. Furthermore, the service must be able to resume from the state it acquired. Therefore, services should model the properties that characterise their state and make sure that check-points of their state are consistent (Kramer and Magee, 1990; Vandewoude et al., 2006).

### *2.3 State synchronisation and support for disconnected operation*

Due to possible wireless network disruptions in the mobile setting of the user, complete network transparency of remote service invocations can have a detrimental effect on service availability and accessibility. The service platform should provide support to deal with intermittent network connectivity in order to recover from temporary failures in network connectivity between two services. With handover to replicated services whose states are synchronised, the effect of network disruptions can be reduced. Moreover, with discrete state synchronisation, the requirement for continuous network connectivity can be mitigated.

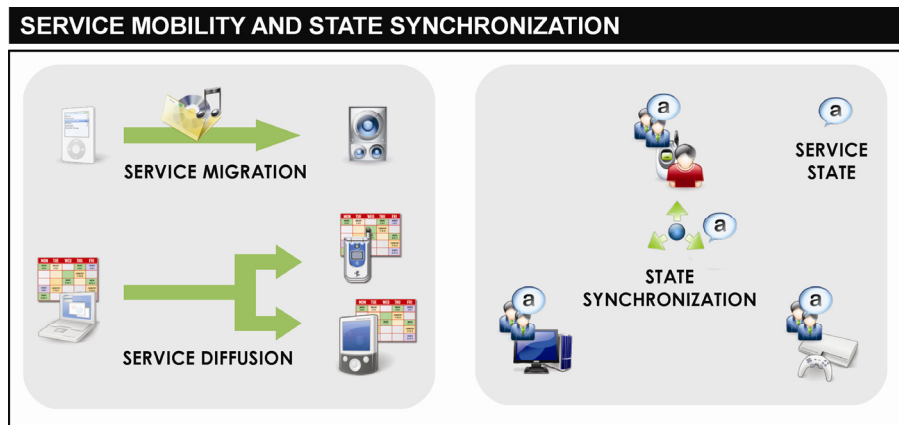
## 2.4 Enhanced service life cycle management

The service platform must provide functions to manage the life cycle of services, such as deployment, withdrawal, starting, stopping and updating. If a service cannot be run locally because of the lack of resources, the service needs to be moved to a remote and more powerful device in the vicinity of the user. The service life cycle management should move services to new hosts and replicate the state if necessary and support two kinds of service mobility (see Figure 2):

- *Service migration*: once the replicated service has moved to a new host, the original one is stopped and uninstalled. This is the typical behaviour of ‘follow-me’ applications that move along with the user.
- *Service diffusion*: in this adaptation, the original service becomes active again once the state has been extracted. New replicas acquire the service state and get activated. After activation, service states are synchronised.

Of course, a combination where a service is replicated on multiple hosts, but where the original is uninstalled should also be possible. Also note that plain service migration does not require any state synchronisation at all, but handing over to a service replica is no longer possible when trying to recover from a network failure. As the effect of network disruptions cannot be mitigated completely, we must also clean up stale replicated services. This problem is similar to distributed garbage collection where heartbeats or lease timeouts are used to detect disconnections and recycle memory. Similar algorithms can be reused to clean up these services.

**Figure 2** Live migration and diffusion of services (see online version for colours)



Note: Service migration completely relocates the service to another host, while service diffusion redeploys the same service on other hosts and ensures service state replication and synchronisation.

### **3 Context-aware service mobility on the OSGi framework**

In this section, we will discuss how the previous requirements have been implemented on top of the OSGi framework (Open Services Gateway Initiative, 2007). This lightweight service-oriented platform has been chosen for its flexibility to build applications as services in a modularised form with Java technology. OSGi already offers certain functionalities that are needed to implement the service migration and diffusion requirements. OSGi is known for its service dependencies management facilities and the ability to deal with a dynamic availability of services. Moreover, OSGi can be deployed on a wide range of devices, from sensor nodes, home appliances, vehicles to high-end servers and allows the collaboration of many small components, called bundles, on local or remote computers. As such, OSGi is a viable platform for service orientation in a ubiquitous computing environment.

#### *3.1 Extending service descriptions with deployment constraints*

Services in the frame of OSGi are published as a Java class or interface along with a set of service properties. These service properties are key-value pairs that help service requesters to differentiate between service providers that offer services with the same service interface. Service providers and requesters are packaged into a OSGi bundle, that is, a JAR file with a manifest file that contains information about the bundle, such as version numbers and service dependencies. Service descriptions are stored in the service registry of OSGi. Service requesters can discover and bind to a service implementation by actively querying for the service or by subscribing to a notification mechanism in order to receive events when changes in the service registry occur.

A recent addition to the OSGi R4 framework that simplifies the registering of plain old Java objects (POJOs) as services and the handling of service dependencies are the declarative services (DS) specification (Open Services Gateway Initiative, 2007). Dependency information that is currently not mentioned in the service descriptor deals with non-functional properties such as hardware, software and resource constraints. For example, one OSGi bundle could be successfully deployed on a J2ME CDC Foundation Profile, while another could need at least a J2ME CDC Personal Profile or a J2SE virtual machine because it uses AWT for a graphical user interface. Moreover, resource (memory, processing power, storage) and hardware (screen, audio, keyboard) dependencies need to be specified as well. The current key-value pair format for service properties is too limited to describe these complex constraints. As discussed in our previous work (Mokhtar et al., 2008; Preuveneers and Berbers, 2008; Preuveneers et al., 2004), ontologies provide a convenient richer specification format to describe and discover pervasive services with support for QoS and context-awareness. We therefore opted to add a ‘deployment’ entry into the declarative service descriptor in which we refer to an ontology that is included in the JAR file of the OSGi bundle:

---

```

<?xml version="1.0"?>
<component name="jabber">
  <implementation class="communication.impl.JabberChatClient" />
  <service>
    <provide interface="communication.ChatClient" />
  </service>
  <deployment>
    <require name="resources1" class="descriptor.owl#MemoryDependency" />
    <require name="software1" class="descriptor.owl#JavaVMDependency" />
    <require name="hardware1" class="descriptor.owl#DisplayDependency" />
    <require name="hardware2" class="descriptor.owl#KeyboardDependency" />
  </deployment>
</component>

```

---

It models the above dependencies as class restrictions on concepts defined in our context ontologies.<sup>1</sup> A device should comply with these constraints if the service is to be deployed successfully. For example:

---

```

<owl:Class rdf:about="#MemoryDependency">
  <rdfs:subClassOf rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
Hardware.owl#RAM" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
Hardware.owl#currAvailable" />
      <owl:someValuesFrom rdf:resource=">= 98304" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

---

The previous constraint declares that a device should have at least one memory resource instance (of class *RAM* or one of its subclasses) with a property *currAvailable* that has a value larger than or equal to 98,304. The type of the property is specified in the *Hardware.owl* ontology, and for this value it is *bytes*.

---

```

<owl:Class rdf:about="#JavaVMDependency">
  <rdfs:subClassOf rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
Software.owl#VirtualMachine" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
Software.owl#hasRenderingEngine" />
      <owl:someValuesFrom rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/
01/java.owl#JavaAWT" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

---

The above constraint declares that a device should have at least one Java virtual machine instance with a GUI rendering engine instance that belongs to the JavaAWT class. The main advantage of our context ontologies (Preuveneers et al., 2004) is that complex semantic relationships only need to be specified once and can be reused by every service descriptor. Moreover, each device profile can specify their characteristics with the same ontologies. If a device then would specify that it runs OSGi on top of JDK 1.6, then an AWT rendering engine dependency would semantically match, but a Java MIDP 2.0 LCDUI rendering engine would not. Resource and hardware constraints can be expressed in a similar way. The matching itself to verify that a service can move to a particular host is carried out by a context enabling service that is described in Section 3.2.

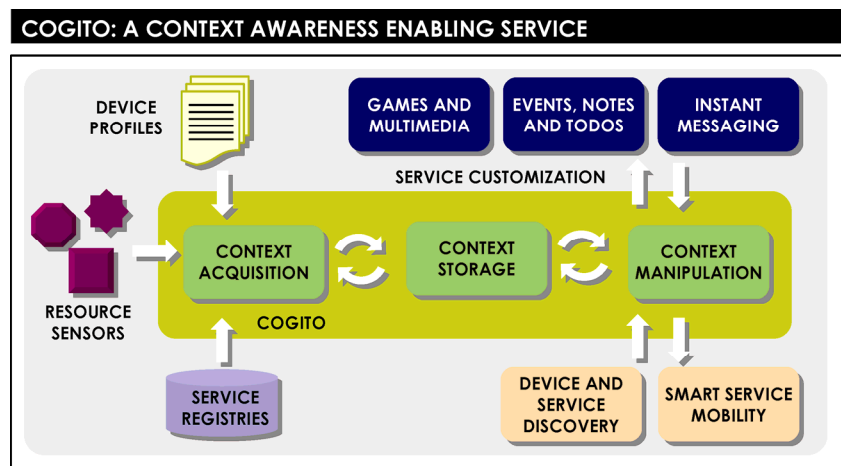
### 3.2 Context-awareness as an OSGi distributed declarative service

In order to diffuse OSGi services in a way that takes into account the heterogeneity and dynamism of a ubiquitous computing environment, we need to be aware of what the characteristics and capabilities of these devices are, where they are located, what kind of services they offer and what resources are currently available. The context gathering, inspecting and transforming objects (CoGITO) framework will provide this information on each host. It gathers and utilises context information to positively affect the provisioning of services, such as the personalisation and redeployment of services tailored to the customer's needs. The core functions of the enabling service are provided as a set of OSGi bundles that can be subdivided into the following categories:

- *Context acquisition*: these bundles monitor for context that is changing and gather information from resource sensors, device profiles and other repositories within the system itself or on the network.
- *Context storage*: a context repository ensures persistence of context information. It collects relevant local information and context that remote entities have published in a way that processing can be handled efficiently without losing the semantics of the data.
- *Context manipulation*: these bundles reason on the context information to verify that context constraints are met. Besides a rule and matching engine that exploits semantic relationships between concepts (Preuveneers and Berbers, 2008), it also provides adapters that transform context information into more suitable formats.

A high-level overview of the building blocks of the context-awareness enabling service is given in Figure 3. We will now discuss some of the design patterns and architectural styles that were used to decompose the CoGITO framework into multiple OSGi services. This modular approach saves resources when certain context managing services are not required and allows for service migration and diffusion of parts of the context framework itself as well.

**Figure 3** Building blocks of the CoGITO enabling service (see online version for colours)



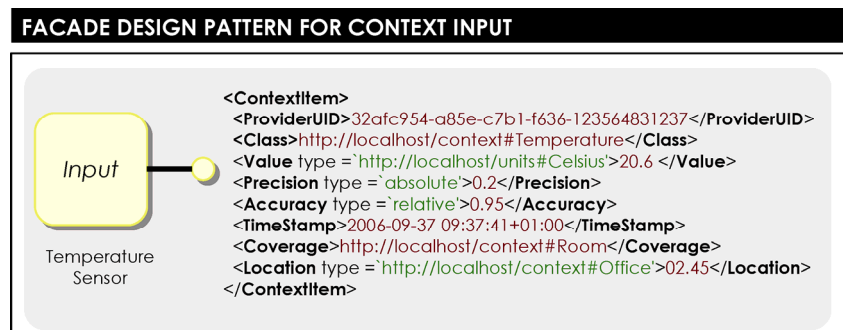
### 3.2.1 Perception of the physical world

Sensors are typically used to capture data from the physical world. A large variety of sensors exists that sense time, location, temperature, motion, touch, light, acceleration, etc. However, there are other ways to perceive the current context. For example, a scanner that detects the presence of certain RFID-enabled items, web services providing access to information, data profiles stored on a device or a user that manually provides information can serve as input. In order to verify the relevance of the detected information, sensed values can be typed with a set of quality attributes:

- 1 *Accuracy*: this refers to the degree of veracity. It describes the closeness of the measured value to the actual true value, or the extent to which the provided information is correct (e.g. 36.8°C versus 37.1°C, or 99%).
- 2 *Precision*: this is closely related to the accuracy parameter, but describes how detailed a measurement is stated (e.g. 36.9 ± 0.1°C). Measurements that are precise are not necessarily accurate and vice versa.
- 3 *Spatial coverage*: this parameter defines the geographic scope of the information. For example, the temperature provided by an outside weather station is unusable for controlling the central heating in a building.
- 4 *Timeliness*: timeliness refers to how current the provided information is at the time of delivery. If not sufficiently up-to-date, the temperature information may not be useful either for the task at hand.
- 5 *Semantic interpretability*: some sensors may express data in semantically related types, for example, 'Celsius' and 'Fahrenheit'. If these relationships are not understood, relevant information could be neglected.

We encapsulate context acquisition into an input component with a unified interface implemented according to the *facade* design pattern (Gamma et al., 1995). The input component hides the inner workings of the sensor and adds the relevant quality properties to the sensed values (as illustrated in Figure 4). As the other context managing components will use the unified interface, we reduce dependencies with this design pattern and hence increase flexibility. The encapsulation into an abstract representation also simplifies the comparison of input components that return similar context information.

**Figure 4** An abstract representation of a sensor typed with quality attributes (see online version for colours)

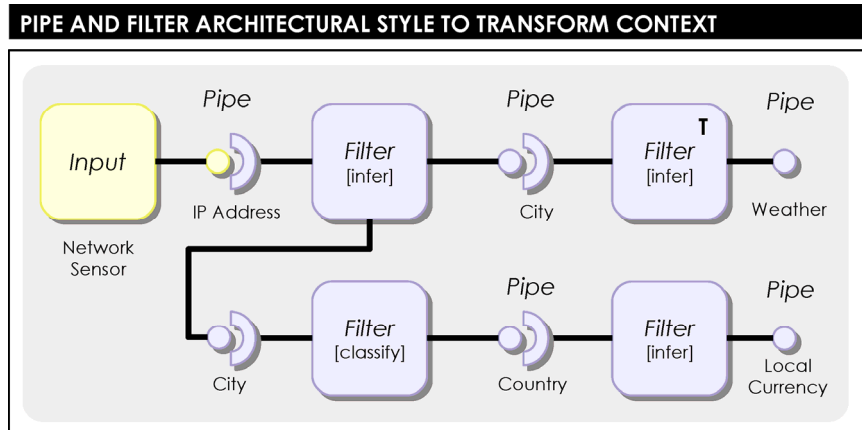


### 3.2.2 Filtering and translation of the sensed context

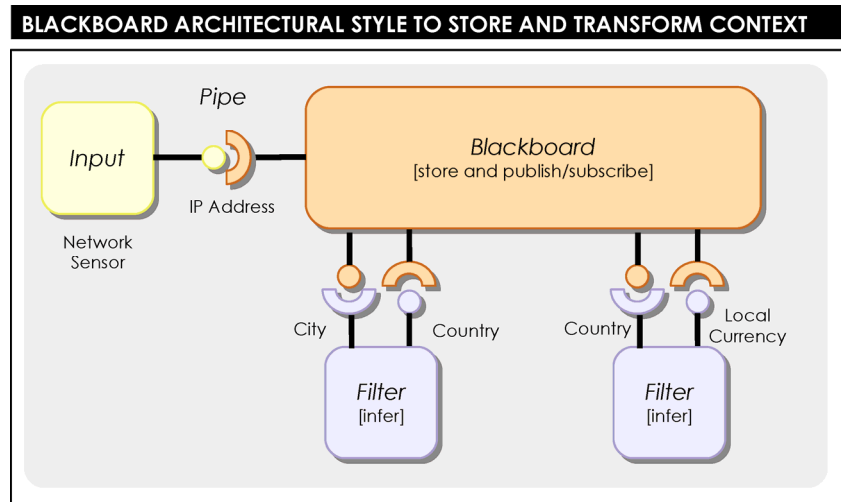
While most context-aware systems have sensors in place to capture information, not all of them perform extensive processing or reasoning on the collected data before determining which action to carry out. Once the context information has been acquired, we can inspect and reflect upon the collected data by processing it with context specific translation components or with general purpose reasoning algorithms. For both techniques, historic context values can be preserved to aggregate multiple sensor values or to detect new patterns.

Translating the representation of sensed context, using it as input to gather related context, or creating a filtered view to select relevant context are all common context transformation operations. Figure 5 shows how the classical *pipe and filter* architectural style is used to filter or transform sensed data. Here, we use it to convert a sensed value (e.g. an IP address) to the name of a city by using of a geomapping database, which in turn can be used to derive the current weather or the name of the country to which it belongs and its currency. It is clear that this architectural style is very flexible to collect data related to the sensed information. However, one must pay special attention to when this translation chain is carried out: when the information is sensed or when the information is requested by the application. For example, inferring the weather is time-dependent while inferring the local currency is not. As such, context transformation not only requires support for orchestration, but also coordination between the context managing components.

**Figure 5** A chain of time-independent and time-sensitive context translations (see online version for colours)



**Figure 6** Decoupled transformation of context with support for persistence (see online version for colours)

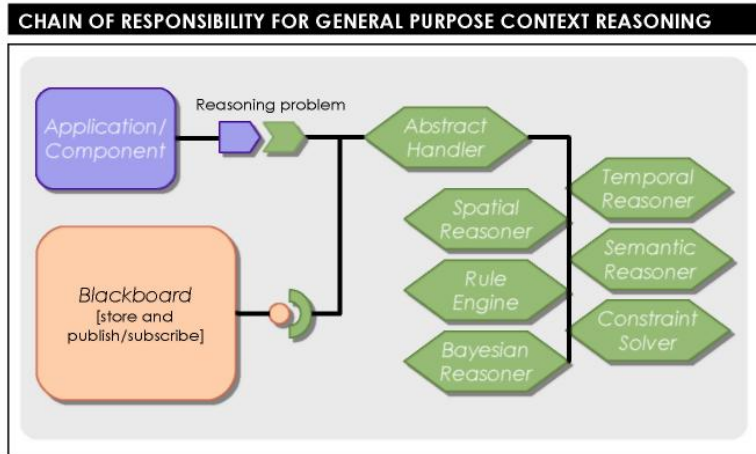


### 3.2.3 Sharing context with the blackboard pattern

The *blackboard* architectural pattern, as illustrated in Figure 6, decouples context translating components. Instead of directly communicating with one another, they interact through the mediation of a shared repository. This pattern also allows them to subscribe for context changes in their area of interest. The main advantage over the pipe and the filter pattern is that there is no predetermined sequence on how these components process the context data. Since there is no direct communication among the components, the architecture topology makes replacing components very easy. Implementing persistence of context data is another feature that can be easily achieved with this pattern. The downside is that in a distributed environment, the node that hosts the repository will become a bottleneck. If this node goes down, the whole system will be affected.

### 3.2.4 Integrating reasoners with the chain of responsibility pattern

The various context transformations in the previous sections were carried out by domain specific components, and now we integrate general purpose reasoners to achieve hybrid reasoning. We use the *chain of responsibility* design pattern to encapsulate these reasoning algorithms. Each reasoner is typically suited to solve a defined class of problems. The chain of responsibility design pattern avoids coupling of the sender of a request to the receiver (i.e. a particular reasoning engine) by giving more than one object the chance to handle the request. Reasoning problems are passed along a chain (see Figure 7), until one context reasoner accepts the problem and solves it.

**Figure 7** Decoupling between the sender and the receiver for solving a context reasoning problem (see online version for colours)

CoGITO is implemented as a distributed declarative service. As the declarative service specification does not cover the registration of remote services, we have each device announcing its presence and that of its context enabling service with a SDP like UPnP, SLP or ZeroConf. Upon joining a network, each other node in the network creates a remote proxy to the enabling service of the joining node, and the DS bundle will ensure lazy registration and activation of the proxy whenever remote context is acquired. When a node leaves the network, the proxies on the other nodes are deactivated. This approach for distributed DS simplifies the collection of context information on the network, but more importantly it also enables transparent sharing of intensive context processing services (such as a rule or inference engine bundle) on resource-constrained devices.

### 3.3 *Service state representation, extraction and synchronisation*

In order to replicate stateful services on multiple hosts with support for state synchronisation, we need to explicitly model the state variables of the service, extract them at runtime and send them to the replicated peers. In order to keep state representation and exchange straightforward and lightweight, we encapsulate the state of a service in a JavaBean. JavaBeans have the benefit of

- 1 being able to encapsulate several objects in one bean that can be passed around to other nodes
- 2 being serialisable
- 3 providing get and set methods to automate the inspection and updating of the service state.

The approach is quite similar to the stateful session beans in the Enterprise JavaBeans specification (Burke and Monson-Haefel, 2006). Stateful session beans maintain the internal state of web applications and ensure that during a session, a client will invoke method calls against the same bean in the remote container. In our approach, however,

a JavaBean is situated locally within each replicated service and does not involve remote method calls. Instead, the contents of the JavaBean is synchronised with those of the replicated services.

### 3.3.1 *Incremental state synchronisation*

The most straightforward way to synchronise the state between the replicated services is to capture and share the state as a whole. For services with a small and stable state, this approach works rather well. However, when a large volatile service state needs to be synchronised with multiple diffused services, network delays and throughput become an issue. Imagine a presentation service that holds a series of images and an internal clock as part of its state. While the clock changes each second, the images do not. Nonetheless, a large state of more than 100 kB that needs to be synchronised each second with each replicated presentation service would not be unimaginable in the case of full state synchronisation. It is clear that the size of the state may become a bottleneck for full state transfer.

Therefore, when modelling the state variables of the services, we identify whether they are volatile or stable. State synchronisation will start with a full state synchronisation and continue with incremental state transfers of the volatile state variables. To ensure that, we have a consistent state on each diffused service, we track if and when each state variable has changed. If a state variable has changed, it will be included in the following incremental state transfer (even if it is a stable state variable). The reason why we do not incrementally transfer per changed state variable is because this fine-grained state synchronisation requires each replicated service to never miss a single state update. With a more coarse-grained mechanism grouping several small volatile state variables in the state transfer (even if they have not changed since the last update), a diffused service can recover more easily from a missed service state update. Because it is not easy to identify in advance which state variables are stable or volatile, we use the time of the last state variable change and the change frequency to modify the type of the state variable at runtime. In our current implementation, we mark state variables as stable if their state transfer rate is at least an order of magnitude smaller than the transfer rate of volatile state variables. Basically, our incremental state synchronisation scheme comes down to a dynamic partitioning of the service state into a fast and slow changing part. In theory, however, we could dynamically partition the service state into more than two parts with each part its own state transfer frequency, but due to management complexity and bookkeeping overhead per state variable we have chosen not to implement this feature.

### 3.3.2 *State conflict resolution*

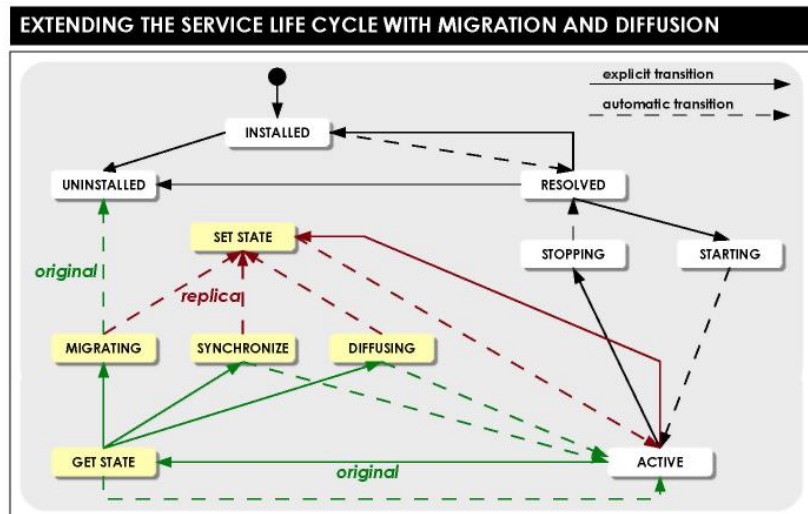
Our current implementation synchronises as soon as the state of an application has been changed. When network failures arise, the state updates are put in a queue and another attempt is carried out later on. The queue holds the revision number of the last state that was successfully exchanged. If, however, two or more replicated services independently continue without state synchronisation, a state transfer conflict will occur when the involved hosts connect again. In that case, a warning will be shown and the user can choose to treat the services as separate applications, or have one service push its state to the others. This approach is rather crude, but acceptable as long as we are mainly dealing with single-user applications.

### 3.4 Extending the life cycle management to relocate services

In a ubiquitous computing environment where pervasive services are replicated, a user may switch from one replicated service to another. Therefore, we add extra information to the service state that helps to coordinate the synchronisation of the state changes. For example, we add a revision number that is increased after each write operation on the bean and use Vector Clocks (Mattern, 1989) among all peers to order the state updating events.

Figure 8 shows the extended life cycle of a service. The original life cycle states of an OSGi bundle include *Installed*, *Resolved*, *Starting*, *Active*, *Stopping* and *Uninstalled*. The dashed arrows define an automatic transition (e.g. a bundle goes automatically from the *Starting* to the *Active* state), while full arrows denote transitions initiated by the application or the user. We have extended the life cycle with new states. With the new transitions, an active service cannot only be stopped, but also have its state extracted and synchronised. At that point the service can continue to migrate to a new host, during which the original service is uninstalled and the replicated service is inserted its new state before becoming active. Or the original service can diffuse to a new host, after which the original and replicated service become active again. We use a lease timeout algorithm to garbage collect stale replicated services in order to recycle resources. Whether or not the original service will be cleaned up depends on the current context and the interaction pattern with the user. The timeouts are application dependent, but can be reconfigured by the user.

**Figure 8** New service states in the life cycle of a migrating or diffusing service (see online version for colours)



Note: Some of the transitions of the original service are shown in green, the ones of the replicated service(s) in red.

## 4 Experimental evaluation

The usefulness and feasibility of dealing with context-aware service migration and diffusion will be illustrated with three applications:

- 1 a grocery list application
- 2 a Jabber instant messaging client
- 3 a Sudoku game.

The devices used in the experiment include two PDAs, a laptop and a desktop, all connected to a local WiFi network. This setup will simulate the real life scenario in Section 4.1.

### 4.1 Scenario

The scenario goes as follows:

It is Saturday morning and mom and dad go shopping. Everybody at home uses the grocery list application to make up a shared grocery list. In the past, mom sometimes forgot her grocery list at home, but now she only needs to take along her smartphone. The application just diffuses along. Dad also has his PDA with him and runs the same replicated application. They decided to split up the work and go shopping separately. At the mall, each time mom or dad finds a product that they need, it is removed from the shared sticky note application. They get each others updates when the state of the application is synchronised.

Mom is still checking out while dad is already at the car. As he is waiting, he has another go at the Sudoku puzzle that he was trying to solve on the home computer that morning. His daughter is online and starts a conversation. She wants to send him a picture, but the display of his PDA is too small. He takes his briefcase, turns on his laptop, the application migrates and he continues the conversation there. Unfortunately, he forgot to charge his battery so after a while his laptop gives up on him. "Back to the PDA but then without the picture", dad decides. In the meantime, mom has arrived and dad tells her about the picture. She will have a look at it at home once the application synchronises with the desktop computer.

### 4.2 Experiments

In the experiment, all the applications are launched on the desktop PC. The Jabber instant messaging client and Sudoku puzzle diffuse to one handheld, while the grocery list application diffuses to both PDAs. In this experiment, we simplify process the decision of whether to migrate or to diffuse a service to multiple hosts if the opportunity arises. In a real deployment, the CoGITO framework keeps track of when and where a service is used and which of the available devices in the vicinity are matching candidates capable of running the service. As such, the usage pattern of a service in a given context will decide what type of service mobility will be initiated by the framework.

The application states are continuously synchronised between all hosts. Later on, the instant messaging application migrates from one PDA to the laptop and back

(while synchronising with the desktop PC). The mobility of the grocery list is illustrated in Figure 9. Additionally, these three applications all make use of two extra general-purpose services

- 1 a stateless audio-based notification service
- 2 a stateful logging service.

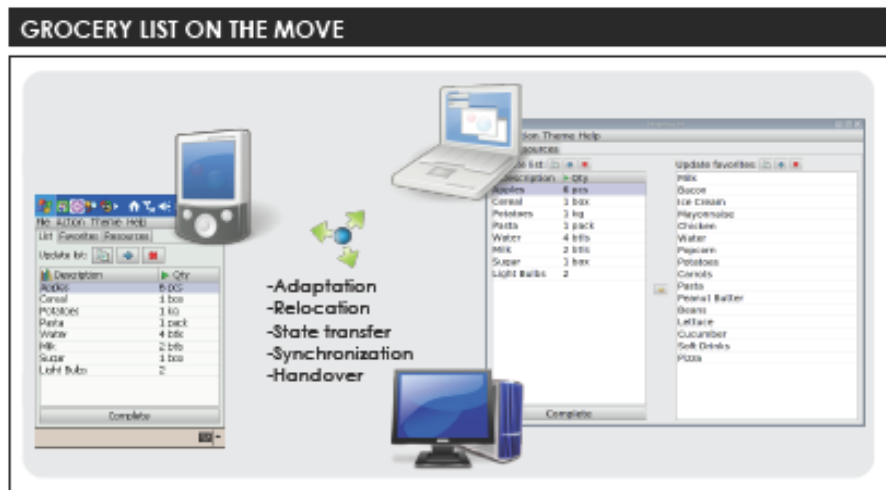
By only deploying them on the desktop computer and synchronising them on the laptop, we enforce that these two services can only be reached from a PDA through a remote connection. The applications will invoke them when

- 1 the grocery list changes
- 2 the online status of somebody changes or
- 3 when the puzzle is solved.

The bindings to the remote services are used to test handover to a replicated service after network disruptions. Let the setup run for 30 min, in which the following actions are simulated:

- Each 30 sec the grocery list is modified and synchronised. This event also triggers an invocation to both the remote services.
- Each 5 sec the state of the Jabber client is synchronised. Each minute the online status of somebody in the contact list changes, and this triggers an invocation to the remote services.
- The Sudoku puzzle changes each 15 sec and the 81 digits of the puzzle are synchronised. A puzzle is solved after 10 min, and this initiates an invocation to the remote services.

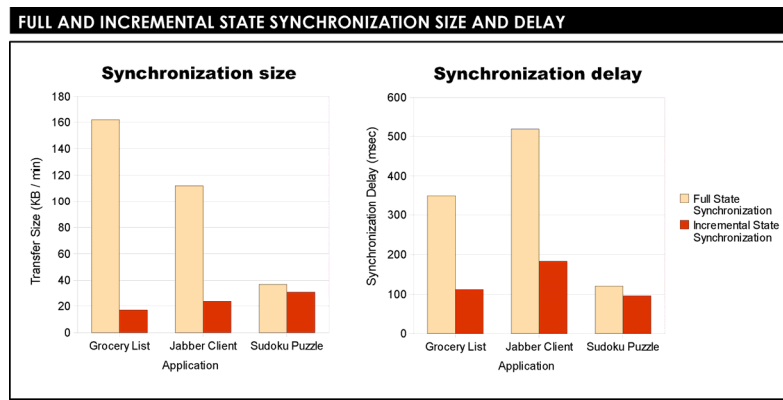
**Figure 9** The grocery list is automatically adapted to fit the new screen size if needed (see online version for colours)



Note: After redeployment the states of all the replicated grocery lists are synchronised.

**Table 1** Overhead results of state transfer, synchronisation and handover delay

	<i>Grocery list</i>	<i>Jabber client</i>	<i>Sudoku puzzle</i>
Bundle size	87,693 bytes	288,235 bytes	54,326 bytes
State size	64,328 bytes	18,235 bytes	1,856 bytes
Relocation time	11 sec	13 sec	7 sec
Handover delay (audio)	51 ms	61 ms	54 ms
Handover delay (log)	157 ms	213 ms	78 ms

**Figure 10** Average state synchronisation sizes and synchronisation delays for the three applications (the lower the size or delay, the better) (see online version for colours)

Each 3 min, we shut down a random network connection for 1 min to verify if the replicated applications can recover from the missed state updates and that handover to one of the remote replicated services. We measured the overhead of state transfer and synchronisation and compare that to the resources that the applications required. We also measured the service availability and responsiveness by measuring the delay of handing over to another remote replicated service after a network disconnection.

In a first variation of the experiment, we always performed full state transfer, while in a second variation, we also allow incremental state synchronisation as discussed in the previous section. Both versions of the experiment were repeated 3 times. The averaged results of the state sizes and the handover delays are shown in Table 1 and are more or less the same for both variations of the experiment. The average transfer throughput and synchronisation delay are shown in Figure 10 for the full and incremental state transfer variation of the experiment separately.

### 4.3 Discussion of the results

The test results show there are only a small overhead for the state transfer. This is mainly a consequence of the size of the applications. They are rather small as they would otherwise not run on mobile devices. For the Sudoku Puzzle, practically all state variables except for the player's name (i.e. the board, the total playing time, etc.) were marked as volatile. So there was no significant difference between full and incremental state synchronisation.

For the Grocery List and Jabber Client applications, we see a big improvement with the incremental state synchronisation because these applications had rather large stable state variables (list of all grocery items, thumbnail pictures of friends, etc.) that were transferred less frequently. For other applications with large chunks of data that need to be replicated only once as part of the state of the service, we expect a similar improvement with incremental state synchronisation.

Delays in state synchronisation are low because the experiments were carried out on a local WiFi network. Other tests in multihop networks showed longer but still acceptable state synchronisation delays (worst case at most 5 times longer). More interesting though is the difference in service relocation time and state exchange time. By proactively moving the service in advance, the time to get an up-to-date replica is a lot smaller for state synchronisation compared to service relocation. As such, service diffusion with state synchronisation provides a much better usability to the user compared to plain service migration. Of course, this assumes the required bandwidth is available.

The handover to the remote services worked in all cases because we avoided the case where all network connections to the remote replicated services are disrupted. In theory, our framework can handle this particular case if the remote service calls can be buffered and invoked asynchronously. However, our current implementation does not support this. Interesting to note for the handover to the replicated services (in our experiment, the stateless audio service and stateful log service), is that handover to another log service takes a bit longer on average. This result is due to the fact that for the audio service, we do not need to wait until the replicated service has acquired the latest revision of the service state. If state synchronisation is taking place during handover, the handover delay can become higher. In our experiment, the remote services were only available on two hosts (the laptop and the desktop). If more devices would host a replicated service, the decision to handover to a particular device could depend on which replica is already completely synchronised.

## 5 Conclusions and future work

This paper presents a context-driven approach to live service mobility in pervasive computing environments. It focuses on service migration and diffusion to multiple hosts to increase accessibility and expedite human interaction with the service. We summarised the basic requirements for service mobility, including enhanced service descriptions, context-awareness to select appropriate targets for service migration and life cycle management support to carry out service relocation with state transfer and state synchronisation. We have discussed how we implemented these requirements on top of the OSGi framework and validated our prototype by means of several applications that are replicated in a small-scale network with enforced network failures.

We studied the effects of service migration and service diffusion. Service migration moves an application from one host to another including its state, while service diffusion replicates the service and the state on multiple hosts. State synchronisation ensures that each service can be used interchangeably. Experiments have shown that the overhead of state transfer and synchronisation is limited for relatively small applications. The results illustrate that, if the available network bandwidth permits, the time to keep the state in sync is a lot smaller than to migrate a service from one host to another. This means that if

a user wants to use another device because it provides a better QoS (e.g. the bigger screen in the scenario), that proactive service diffusion provides a much better usability (i.e. shorter delays).

Future work will focus on a better handling of state synchronisation conflicts, investigating the possibility to automatically partition the service states into more than two parts each with their own optimised state transfer frequency. However, our main concern will always be to keep the supporting infrastructure flexible and lightweight enough for low-end devices with room left to actually run applications. The outcome of this future work could result in a better policy with respect to on how many remote devices a service should be replicated and under which circumstances service migration would be a better approach compared to service diffusion.

## References

- Bourges-Waldegg, D., Duponchel, Y., Graf, M. and Moser, M. (2005) 'The fluid computing middleware: bringing application fluidity to the mobile internet', *SAINT '05: Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05)*, Washington, DC, USA: IEEE Computer Society, pp.54–63.
- Burke, B. and Monson-Haefel, R. (2006) *Enterprise JavaBeans 3.0* (5th ed.), Sebastopol, CA: O'reilly Media, Inc.
- Dey, A.K. (2001) 'Understanding and using context', *Personal Ubiquitous Computer*, Vol. 5, No. 1, pp.4–7.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley Professional, January 15, ISBN-10: 0201633612, ISBN-13: 9780201633610.
- Gu, T., Pung, H.K. and Zhang, D.Q. (2004) 'Toward an OSGi-based infrastructure for context-aware applications', *Pervasive Computing, IEEE*, Vol. 3, No. 4, pp.66–74.
- Helal, S. (2002) 'Standards for service discovery and delivery', *Pervasive Computing, IEEE*, Vol. 1, No. 3, pp.95–100.
- Kim, J-H., Yae, S-S. and Ramakrishna, R.S. (2001) 'Context-aware application framework based on open service gateway', *International Conference on Info-tech and Info-net 2001*, Vol. 3, pp.209–213.
- Kramer, J. and Magee, J. (1990) 'The evolving philosophers problem: dynamic change management', *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, pp.1293–1306.
- Lee, H., Park, J., Ko, E. and Lee, J. (2006) 'An agent-based context-aware system on handheld computers', *International Conference on Consumer Electronics, 2006, USA*, pp.229–230.
- Mattern, F. (1989) 'Virtual time and global states of distributed systems', *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*.
- Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V. and Berbers, Y. (2008) 'EASY: efficient semantic service discovery in pervasive computing environments with QoS and context support', *Journal of System and Software*, Vol. 81, pp.785–808.
- Open Services Gateway Initiative (2007) OSGi Service Gateway Specification, Release 4.1.
- Papazoglou, M.P. and Georgakopoulos, D. (2003) 'Service oriented computing', *Communications of the ACM*, Vol. 46, No. 10, pp.24–28.
- Preuveneers, D. and Berbers, Y. (2008) 'Encoding semantic awareness in resource-constrained devices', *IEEE Intelligent Systems*, Vol. 23, No. 2, pp.26–33.

- Preuveneers, D., den Bergh, J.V., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V. and Bosschere, K.D. (2004) 'Towards an extensible context ontology for ambient intelligence', In P. Markopoulos, B. Eggen, E. Aarts and J.L. Crowley (Eds.), *Second European Symposium on Ambient Intelligence*, Vol. 3295 of *LNCIS*, Eindhoven, The Netherlands: Springer, pp.148–159.
- Rellermeyer, J.S. (2006) 'flowSGI: a framework for dynamic fluid applications', Master's Thesis, ETH Zurich.
- Rellermeyer, J.S., Alonso, G. and Roscoe, T. (2007) 'Building, deploying, and monitoring distributed applications with eclipse and R-OSGi', *eclipse '07: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, New York, NY: ACM, pp.50–54.
- Saito, Y. and Shapiro, M. (2005) 'Optimistic replication', *ACM Computing Surveys*, Vol. 37, No. 1, pp.42–81.
- Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice-Hall, Inc.
- The Eclipse Foundation (2007) *Eclipse Communication Framework*.
- Vandewoude, Y., Ebraert, P., Berbers, Y. and D'Hondt, T. (2006) 'An alternative to quiescence: tranquility', *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Washington, DC, USA: IEEE Computer Society, pp.73–82.
- Weiser, M. (1991) 'The computer for the 21st century', *Scientific American*, Vol. 265, No. 3, pp.94–104.
- Yu, Z., Zhou, X., Yu, Z., Zhang, D. and Chin, C-Y. (2006) 'An OSGi-based infrastructure for context-aware multimedia services', *Communications Magazine, IEEE*, Vol. 44, No. 10, pp.136–142.

## Note

<sup>1</sup>See <http://www.cs.kuleuven.be/~davy/ontologies/2008/01/> for the latest revision of our context ontologies.