PerCraft: Towards Live Deployment of Pervasive Applications

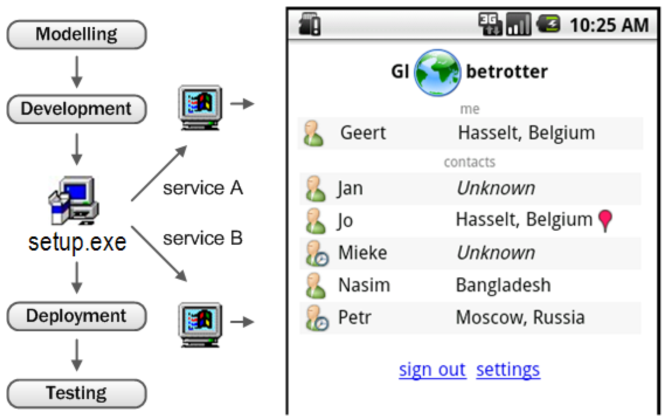
Geert Vanderhulst Kris Luyten Karin Coninx
Hasselt University – transnationale Universiteit Limburg
Expertise Centre for Digital Media
Wetenschapspark 2, 3590 Diepenbeek, Belgium
Email: {geert.vanderhulst,kris.luyten,karin.coninx}@uhasselt.be

***Abstract*—Pervasive applications are typically realized through ad-hoc service and user interface compositions. While many tools focus on the development of pervasive services by masking the complex technical side of a pervasive computing environment, the deployment of an application as a whole – i.e. a set of services and user interfaces – is often forgotten. We present an alternative design strategy and tool for pervasive applications in which pervasiveness is not considered a handicap, but rather as a situation that draws extra attention to the deployment of applications. By crafting pervasive applications and their services as independent context consumers and producers, we illustrate how the behaviour of a pervasive application deployed using our approach can be observed while it executes.**

I. Introduction

The vision of Weiser on ubiquitous computing [13] is being approached increasingly well nowadays: services are embedded in mobile phones and everyday appliances. Next to manufacturers of heterogeneous devices and embedded services, pervasive applications can leverage their input and output capabilities and local resources by deploying their own services on these devices. We define a pervasive application as an orchestration of services (i.e. application logic) running on distributed devices (i.e. computing nodes) which can be interacted with by means of user interfaces that are migrated to end-user devices. Many frameworks that target the development of pervasive applications try to hide the heterogeneity and distributed nature of a pervasive computing environment, e.g. by abstracting the network layer or the deployment process of an application's services [3]. Unfortunately, masking the complexity of a pervasive environment introduces caveats at different stages in a pervasive application's development process and lifecycle which are often neglected:

- *Performance* : hiding essential aspects of a pervasive computing system should be avoided when it constrains developers in taking care of the efficiency of their applications. For example, uncontrolled event subscriptions could disrupt an efficient regulation of event streams and actions that are performed upon receiving an event [6].
- *Deployment* : when services are installed beforehand they give rise to pervasive applications after discovery when brought together [14]. However, installing a new pervasive application while the system is in use demands for live deployment of services over available devices.



Geert	Hasselt, Belgium
Jan	Unknown
Jo	Hasselt, Belgium
Mieke	Unknown
Nasim	Bangladesh
Petr	Moscow, Russia

Fig. 1. PerCraft bundles a wizard with pervasive applications so that they can be installed similar to desktop applications. The services of a pervasive application such as Globetrotter (screenshot) are deployed over several devices and might need to be rewired at runtime to maintain an acceptable performance.

- *Debugging and verification* : if an application does not behave as expected, it is often difficult to attribute a cause to this behaviour without inspecting services individually. Moreover, it can be hard to verify whether a pervasive application behaves as expected in the first place [10].

With the PerCraft approach presented in this paper, we tackle exactly the above mentioned issues. Instead of hiding a pervasive application by invisibly embedding it in the environment, we focus on observable applications that can be inspected at runtime. We present a design strategy that contributes to an increased visibility of pervasive applications in order to gain better control over migration, performance and error handling. Our approach is based on *sensors* and *aggregators* that respectively produce and collect context information, which might give rise to a new application state. Since context information is key to build intelligent systems [2], we ensure the state of resources can be easily retrieved at runtime, even when distributed over an heterogeneous environment (section III). Next, we introduce the Globetrotter application (see figure 1) as running example in this paper which we modelled, developed, deployed and tested using our PerCraft design strategy and tools (section IV). After a short discussion of our approach (section V), we finally draw conclusions.

II. Related Work

Previous research has pointed out that end-users benefit from a do-it-yourself approach to deploy applications in their smart homes [5]. In [5], the authors present a software architecture and tangible deployment tool that allows users to actively participate in the incremental deployment of a ubicomp environment. Other tools focus on the runtime migration and adaptation of user interfaces to interact with pervasive services using end-user devices [11], [4], [9], which can be typically situated after the initial deployment of the pervasive services. For example, in Huddle [9] end-users can specify the data flow between resources in a pervasive environment by means of a generated user interface. Such tools can be considered part of a meta-user interface: an interactive system with its own user interface that provides end-users with means for controlling interactive spaces and evaluating their state, as defined by Coutaz et al. in [1]. With PerCraft, we provide a design approach to create pervasive applications that can be (re)deployed at runtime, interacted with and evaluated using tools integrated in such a meta-user interface. PerCraft's own deployment and monitoring tool also fits in a meta-user interface although it is targeted at (amateur) developers. A related tool, called PerViz, eases the debugging process of pervasive applications [10]. Developers interact with PerViz through a Web browser which provides a centralized location to study and filter aggregated application debugging logs and state. By passing context produced by aggregators and sensors in PerCraft as debugging information to PerViz, these frameworks could be combined to enhance the visibility of a pervasive application and its installation.

PerCraft leverages the ReWiRe framework [12] to deploy services and user interfaces in a pervasive environment. In ReWiRe, software components are deployed as OSGi bundles on available devices. Chen et al. also illustrated that OSGi technology is very suited for dynamic deployment of software components in a pervasive environment in [8]. By adopting an OSGi-based pervasive framework, we can focus on higher-level application deployment and refer to [12] for more details on lower-level service deployment.

III. Distributed context management

Pervasive applications rely on the ability to collect information about resources at runtime [2]. To accommodate this requirement, a software architecture for realizing such applications must support distributed context retrieval on demand and on change. Since the flow of context data between resources has a major impact on the runtime behaviour of an application and its deployment, we suggest a reference architecture for dealing with context as part of the PerCraft framework. In this architecture, an environment model that captures the current context of use encompasses a set of heterogeneous computing nodes and a central repository, the context store, as shown in figure 2. The context store stores information about the *semantics of resources* such as the properties and relations a resource supports which are defined in ontologies. Domain ontologies are imported as *static* documents in the context store and make up a knowledge base that is shared amongst applications. Besides the semantics of the environment and

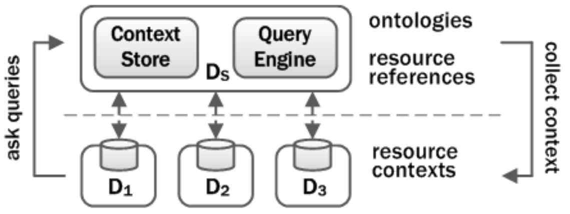


Fig. 2. Ontologies describing the environment topology are imported in the context store which runs on a dedicated server device (D_S). Resources are published on client devices (D_1, D_2, D_3) and advertised with a reference in the context store. Queries are directed to a query engine at the server device which collects resource-specific context data from client devices when needed in order to evaluate a query.

its applications, the context store also includes a registry of references to *instances of resources* whose execution context resides on distributed computing nodes. Each networked device can fulfill the role of computing node, provided it runs a platform on which software can be deployed and executed. In this case, resources such as the device itself, services running on the device or a resource attached to the device are dynamically advertised with a reference in the registry that points to the resource instance (e.g. a data object) on the device. Note that a resource's context is not duplicated in the context store, but returned in a semantic data format upon request. The context retrieval is based on the content negotiation mechanism defined in the HTTP specification[1]: a context requester (e.g. a service) asks for a resource's context represented as e.g. RDF triples and this format is then supplied by the server (i.e. the device that holds the resource). This ensures the state of a resource can be queried against an (aggregated) ontology, regardless of where and how it is stored internally, e.g. on a chip or in a database on the Web.

To simplify the task of acquiring context and notifying context changes, we introduce *aggregators* and *sensors* which are further discussed in the next sections.

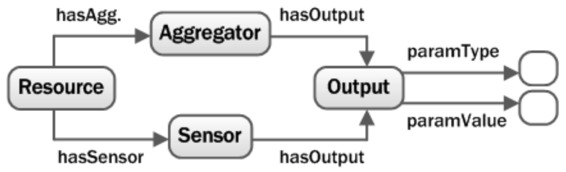


Fig. 3. Resources can request context on demand through aggregators and publish context changes through sensors.

A. Context on demand: aggregators

An aggregator collects information about a resource and delivers this data in a set of output parameters. For example, an OnOffAggregator attached to a light resource could return the state of a light as an integer value (e.g. 1 = On, 0 = Off). Queries are a powerful instrument to realize aggregators

[1]http://www.w3.org/Protocols/rfc2616/rfc2616.html

because they can be executed from any device in the network. Since the entire environment context is described by means of (instances of) ontologies, semantic query languages such as SPARQL and OWL-QL are suitable candidates to interrogate the model. However, these query languages assume that all context data is centralized in a semantic database while we envision distributed storage as illustrated in figure 2. In order to query a distributed context of use using conventional query languages, we propose a query engine layered on top of the CS that processes a query Q in three steps:

1) *Resource selection* : a subquery Q_r, derived from the final query Q or specified manually, selects references to resources whose context should be resolved in order to answer Q.
2) *Context aggregation* : a temporary model is prepared that shares domain knowledge with the CS and which includes the current context of selected resources, fetched from distributed nodes.
3) *Query evaluation* : with all relevant context data aggregated in a temporary model the query Q can be evaluated against this model.

Aggregators hide underlying queries and are executable components that deliver context on demand. They can be compared with "getters" in an object-oriented programming language, returning multiple output values.

B. Context on change: sensors

Resources can publish context information through sensors to which other resources can subscribe. When a context change occurs in a resource, a corresponding sensor related to the resource is triggered and resources subscribed to the sensor will be notified of the change. Sensors are modelled likewise to aggregators and can also exchange context information in output parameters, but instead of being invoked on demand, sensors are triggered upon change. To filter the amount of context updates published to resources, we suggest additional *convenience sensors* which enable a more detailled selection of context changes at the subscription stage. For instance, an ONOFFSENSOR, linked with a LIGHT resource that is triggered when a light's state is updated, can be extended with an ONSENSOR and an OFFSENSOR. In this case, the ONSENSOR is fired as a special variant of an ONOFFSENSOR with an output value corresponding to an ON state. All resources that are subscribed to either the ONSENSOR or the ONOFFSENSOR will then get notified of the context update.

IV. CRAFTING PERVASIVE APPLICATIONS

To illustrate how pervasive applications are created using the PerCraft framework, we will outline the different steps involved in our approach using a prototype application called Globetrotter. The application provides a messenger-like user interface listing friends along with information about their current location. To protect the privacy of its users, the granularity of spatial detail that is published about a user can be configured at a friend-per-friend base. For each friend, the user assigns a profile that determines the level of spatial detail that friend has access to: None (default), Country, City or Spot. The Country and City profiles respectively return the name of the country or the city the user currently resides in, whilst the Spot option grants a detailled read-out of the user's whereabouts (e.g. provided as GPS coordinates). Two plugins extend the Globetrotter application:

1) The TripIt plugin adds support for travel itineraries planned by a user. When a foreign travel destination is reached, it provides access to practical local information such as important phone numbers and cultural habits.
2) The FriendSpotter plugin notifies a user if one of her friends is in the vicinity, e.g. in the same city or within a predefined distance.

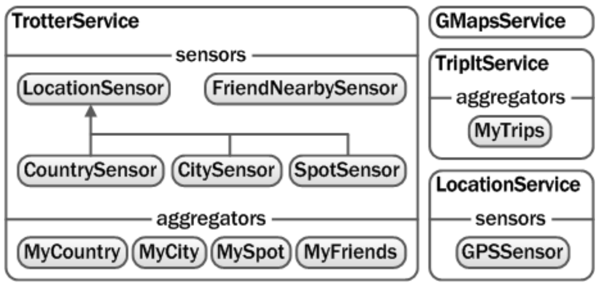
The next sections elaborate on the modelling, development, deployment and testing of this application using PerCraft.

A. Step 1: Modelling

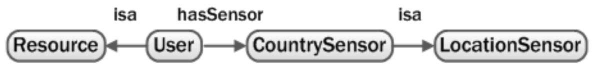
In the modeling step, we break down an application in a set of services for which we identify sensors and aggregators. For example, figure 4 shows an application model for the Globetrotter application. The application is realized using four services of which the TROTTERSERVICE is the most complex one. As shown in figure 4(a), this service defines a number of sensors and aggregators that produce and collect information about a user's location. Basically, the TROTTERSERVICE will filter data it receives from a GPSSENSOR provided by a LOCATIONSERVICE and e.g. transform GPS coordinates into the name of a country using the GMAPSSERVICE and then trigger a COUNTRYSENSOR. Figure 4(b) shows the COUNTRYSENSOR in more detail. Just like the CITYSENSOR and SPOTSENSOR, the COUNTRYSENSOR is linked with the USER concept (which is also a RESOURCE) and inherited from a more generic LOCATIONSENSOR. Hence one can simply subscribe to a $\{User_1, LocationSensor\}$ event to get notified about changes in the user's location. Depending on the granularity of spatial detail the subscriber has access to, the LOCATIONSENSOR will be substituted by e.g. a COUNTRYSENSOR when a location event takes place. Globetrotter's TripIt plugin is supported through the TRIPITSERVICE whilst we provide the FriendSpotter feature using an additional sensor in the TROTTERSERVICE, namely the FRIENDNEARBYSENSOR.

B. Step 2: Development

In the development step, we implement the services and in particular their sensors and aggregators that were identified in the previous step. For this purpose, we rely on the ReWiRe framework, but other frameworks that implement a distributed context store similar to the one discussed in section III could be used as well. Sensors and aggregators are integrated in a service by implementing a `Sensor` and `Aggregator` interface respectively. Aggregators use queries in their backend such as the query in listing 1 (MYCITY aggregator). The query Q_r first aggregates the location context of a user in a temporary model and next, the query Q selects the user's current city from this temporary model. Note that the CITY property is only available if the query requester (i.e. the



(a) Services, sensors and aggregators.



(b) Relation between resources and sensors.

Fig. 4. Application model for Globetrotter.

executor of the aggregator) has sufficient privileges to access this information.

```
Qr : SELECT ?lc WHERE { ?lc :location :TrotterService1 }
Q  : SELECT ?c WHERE { :User1 :city ?c }
```

Listing 1. To select the city a user currently is in, the user's location context is first retrieved.

Sensors, on the other hand, are passed updated context information and are emitted as events (i.e. resource/sensor pairs with output values such as $\{User_1, CitySensor, Tokyo\}$) over the network to interested parties. Additional application logic is added to realize a service to interface with a GPS chip (GPSSERVICE) or functions to configure a user's privacy (TROTTERSERVICE). The GMAPSSERVICE acts as a wrapper for the Google Maps API[2]. Likewise, the TRIPITSERVICE uses the TripIt API[3] to retrieve trips managed in an online TripIt account. Since services are deployed as OSGi bundles in ReWiRe, we encapsulate each service in an OSGi bundle.

C. Step 3: Deployment

In the deployment step, we first connect model and implementation. Each service is mapped on an OSGi bundle, and each sensor and aggregator on a class in this bundle. This process is semi-automated in our tool: using reflection, sensors and aggregator classes are resolved and matched with their representations in the model. For example, the TROTTERSERVICE is linked with the bundle `globetrotter.jar` and its CITYSENSOR with a Java class inside this bundle: `package.globetrotter.CitySensor`. To manage the deployment of an application's services in PerCraft, we classify services as *zombie*, *peasant* or *wizard* service:

- *Zombie service* : The service is maintained and served by a third party. Although its deployment is beyond our control, we can write aggregators and sensors for the service, packed in a peasant service.

[2]http://code.google.com/apis/maps/
[3]http://www.tripit.com/developer/

- *Peasant service* : The service runs on a specific device where it processes either local data (e.g. user input or sensor readings from built-in hardware) or remote data (e.g. to update the state of a local resource).
- *Wizard service* : This service coordinates peasant services and/or includes a user interface (i.e. wizard) to (re)deploy a pervasive application.

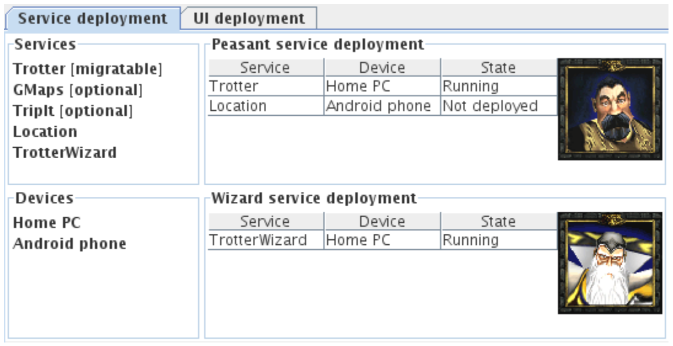
Furthermore, we consider two properties – *migratable* and *optional* – that can be attributed to any peasant service; a wizard service is nor migratable, nor optional by design:

- *Migratable peasant service* : It does not matter where this peasant service runs to make the application work. However, the device it runs on could have an impact on the application's performance.
- *Optional peasant service* : The service is part of the application but not really needed to make it work. When available, it extends the application with extra features.

Each application built using PerCraft consists of a wizard service and a number of peasant services. So far, we only created peasant services for the Globetrotter application of which the TROTTERSERVICE is migratable since all it needs is a CPU and some memory to process incoming sensor events and publish outgoing sensor events. Note also that the GMAPSSERVICE and TRIPITSERVICE are optional peasant services which rely on zombie services (i.e. Web services) in their back-end. The PerCraft tool depicted in figure 5 eases the development of a wizard service. It allows users to drag services on target devices and indicate a master device that will run the application's wizard service. When pressing the deploy button, a deployment descriptor is generated and bundled with a wizard service which is sent to the selected master device. A default wizard service interprets the deployment descriptor and migrates peasant services over selected devices. However, further orchestration is often needed to connect and configure peasant services. This is achieved through a custom wizard service, tailored to the needs of a particular pervasive application. We distinguish between two (re)deployment strategies that can be provided by a wizard service:

- *User-driven* : a deployment wizard (user interface) asks the user about application-specific details. The wizard transforms user input into configuration properties which are sent to appropriate peasant services.
- *Application-driven* : the wizard service automatically ensures that an application runs in optimal conditions even when the configuration of the environment evolves (e.g. devices (dis)appear) or gracefully suspends the application's services upon failures.

In a custom wizard service for the Globetrotter application, we combined both strategies. A built-in wizard lets a user manually rewire a TROTTERSERVICE to another suitable device while the application is in use. Furthermore, the application's wizard service is programmed to gracefully shutdown the application's services upon failure. For example, if a TROTTERSERVICE fails (i.e. becomes unresponsive), the wizard will suspend its related GPSSERVICE as well. As such, a wizard service basically acts as the brain of an application. While a migratable peasant service can suggest a rewiring action, e.g. because the battery of the mobile device it runs on is almost dead, it is the wizard service that will take final decisions regarding the (re)deployment of an application's peasant services and their orchestration.



Services
Trotter [migratable]
GMaps [optional]
TripIt [optional]
Location
TrotterWizard

Peasant service deployment

Service	Device	State
Trotter	Home PC	Running
Location	Android phone	Not deployed

Devices
Home PC
Android phone

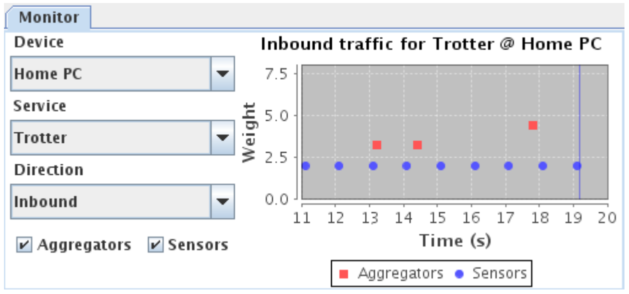
Wizard service deployment

Service	Device	State
TrotterWizard	Home PC	Running

Fig. 5. Generating a deployment descriptor to configure the distribution of peasant and wizard services.

D. *Step 4: Testing*

In the testing step, we focus on monitoring and benchmarking the flow of context information within an application. While an application executes, the PerCraft tool visualizes context produced and collected by sensors and aggregators in its peasant services on a real-time chart, shown in figure 6. Each sensor and aggregator is attributed an estimated weight that indicates the cost for processing it. Factors that influence this cost are for example its network load and required processing power. Note that when selecting a sensor or aggregator on the chart, its outputs and the service and device from where it originates are displayed. As context updates have a major impact on an application's behaviour, they help to explain and thus debug a pervasive application. For example, we can trace back a FRIENDNEARBYSENSOR to a LOCATIONSENSOR which was triggered as a reaction to a GPSSENSOR, analyze their outputs and inspect from which services and devices they originate. Moreover, we can write unit tests as a special type of aggregator (e.g. with a boolean output) to assert that an application behaves as expected at strategic places in its lifecycle. For instance, a unit test aggregator that verifies the correctness of a FRIENDNEARBYSENSOR can reuse and execute the MYCITY or MYSPOT aggregator on your and your friend's TROTTERSERVICE and assert that the collected location data indeed matches the 'nearby' definition. The context flow in an application is also a measure for its performance. For example, deploying a TROTTERSERVICE at the same (mobile) device as a LOCATIONSERVICE is the most efficient when no real-time location coordinates are to be sent out (i.e. all friends are assigned a City privacy level or less). However, when an online friend has access to Spot level spatial data, it could be more efficient to migrate the TROTTERSERVICE to a more performant device; location coordinates need to be sent over the network anyway. Using PerCraft, we can



Device: Home PC
Service: Trotter
Direction: Inbound
☑ Aggregators ☑ Sensors

Fig. 6. Analyzing the flow of context information in a pervasive application. The estimated cost of processing sensors and aggregators in a service is shown in a chart; output parameters can be further inspected at execution time.

evaluate at runtime how devices and the services they run can be optimally orchestrated to deliver a desired quality of service such as a minimal network load. Moreover, wizard services can use quality of service measurements to automatically re(deploy) migratable peasant services at runtime and optimize an application's performance.

By analyzing the context flow in the Globetrotter application, it can be seen that the FRIENDNEARBYSENSOR leads to inefficient behaviour when two users are far away, i.e. if they have Spot level access to each other's location. In this particular case, the location details of a user residing in Tokyo are continuously exchanged with the TROTTERSERVICE of a user in New York and their relative distance is calculated to check whether they are not yet 'nearby'. A more efficient approach would be to poll for location updates instead of listening to LOCATIONSENSOR events as soon as two users are separated more than a certain distance from each other.

V. DISCUSSION

To evaluate our approach, a prototype of the Globetrotter application was created as outlined in section IV. To accommodate different software platforms, we encapsulated Globetrotter's services into OSGi bundles and Android compatible packages, the former to enable usage as a ReWiRe services, the latter for optimal deployment on Android devices. Although the Android platform does support OSGi[4], we opted for a separate light-weight deployment package. It is our believe that we should support tailored components (closely integrated with the platform) as well as loosely coupled components for multiple platforms. This provides a better user experience in situations when a tailored component is available, and minimal access to functionality when a multi-platform component is available. The minimal requirements for a device to be compatible with PerCraft is to implement *install*, *uninstall*, *update*, *start* and *stop* operations to manage the deployment and lifecycle of individual services. Optional operations can be implemented to support distributed reporting about the context flow in services published on a device to enable the usage of PerCraft's monitoring tool. The light-weight PervCraft component for the Android platform that

[4]EZDroid (http://www.ezdroid.com/): OSGi for Android.

implements service management operations includes basic reporting routines. As such, a pervasive application created using PerCraft can embed service implementations for different platforms of which a suitable one is selected when a (re)deployment action is triggered by a wizard service. It is left to a PerCraft client (e.g. a ReWiRe client or Android client) to decide upon a suitable service from the context store in which an application's services are advertised (see section III).

Next to services, user interfaces are required to interact with an application. Several pervasive frameworks treat user interfaces as a special type of services, in which case they can be considered as (optional) migratable peasant services in PerCraft. Opposed to this, infrastructures such as ICrafter [11] and the Personal Universal Controller (PUC) [9] handle user interfaces as separate components that can be (semi-)automatically generated. In our prototype implementation, we opted for a separate Web interface served by a small centralized Web application that manages user accounts and connects with a.o. TROTTERSERVICE instances in its back-end. Deploying a user interface for Globetrotter on an end-user device initiates a Web browser at the target device which is pointed to the URL of Globetrotter's user interface. We opted for a Web interface to support many platforms at once, but richer platform-specific user interfaces that better exploit a device's input capabilities can be embedded in a PerCraft application as well and deployed similar to services.

Using the PerCraft tool depicted in figure 5, we deployed an instance of Globetrotter over an Android phone and a desktop PC. Other instances of the application were deployed on some of our colleagues workstations as to populate the environment with a number of users. To simulate movement of these users, we slightly modified their application instance with a custom location service that produces random location events.

PerCraft supports service compositions that are managed by an application's wizard service or manually orchestrated by an end-user. Besides, dynamic service compositions give rise to collaborative applications: when wizard services are discoverable, applications can synchronize their goals and collaborate. In fact, interaction between wizard services might be essential in an application-rich pervasive environment in order to avoid that applications start competing for resources. Rather than optimizing the deployment of services on an application per application base, a group of wizard services could agree on the best compromise to deliver an acceptable performance for each application in the environment.

VI. CONCLUSIONS

With PerCraft we suggest pervasive applications become end-user products which can be (re)deployed over several available devices, either automatically or assisted by the end-user. Similar to desktop applications, a context-aware wizard bundled with a PerCraft application guides the user through the setup process which is re-invoked whenever the configuration of an environment changes in such a way that the execution of an application nor its performance is threatened. By modelling the exchange of context information as aggregators (on demand) and sensors (on change), we have illustrated using a prototype application that tools can be used to inspect the flow of context information within an application at runtime and hence get insight in the best distribution of an application's services. The major merit of our approach is a simplified deployment of pervasive applications as a whole (i.e. services are not installed individually) whilst still allowing for dynamic service compositions. We acknowledge that developers still need to provide different versions of a service when multiple target platforms are considered. We believe that this situation can only be resolved when a standardized service platform becomes available that runs on recent set-top boxes, Android devices, Windows Mobile phones, etc. Live deployment of services over end-users also brings up privacy and security issues. Directions for future work include an implementation of trust mechanisms in PerCraft such as presented in [7].

REFERENCES

[1] Joëlle Coutaz. Meta-User Interfaces for Ambient Spaces. In *Task Models and Diagrams for Users Interface Design (TAMODIA'06)*, pages 1–15, 2006.

[2] Anind K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[3] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[4] T. Heider and T. Kirste. Supporting Goal-Based Interaction with Dynamic Intelligent Environments. In *European Symposium on Ambient Intelligence (EUSAI'02)*, pages 596–602, 2002.

[5] Fahim Kawsar, Tatsuo Nakajima, and Kaori Fujinami. Deploy Spontaneously: Supporting End-users in Building and Enhancing a Smart Home. In *UbiComp*, pages 282–291, 2008.

[6] Prashant S. Kumar, Qiang Zeng, and Gurdip Singh. Constraining Event Flow for Regulation in Pervasive Systems. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PERCOM'05)*, pages 314–318. IEEE Computer Society, 2005.

[7] Brent Lagesse, Mohan Kumar, Justin Mazzola Paluska, and Matthew Wright. DTT: A Distributed Trust Toolkit for Pervasive Systems. In *Proceedings of the 7th IEEE International Conference on Pervasive Computing and Communications (PERCOM'09)*, pages 1–8. IEEE Computer Society, 2009.

[8] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling Smart Spaces with OSGi. *IEEE Pervasive Computing*, 2(3):89–94, 2003.

[9] Jeffrey Nichols, Brandon Rothrock, Duen Horng Chau, and Brad A. Myers. Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST'06)*, pages 279–288, 2006.

[10] Hubert Pham and Justin Mazzola Paluska. PerViz: Painkillers for Pervasive Application Debugging. In *Proceedings of the 8th IEEE International Conference on Pervasive Computing and Communications (PERCOM'10)*. IEEE Computer Society, 2010. to appear.

[11] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceedings of the 3rd international Conference on Ubiquitous Computing (UbiComp'01)*, pages 56–75. Springer-Verlag, 2001.

[12] Geert Vanderhulst, Kris Luyten, and Karin Coninx. ReWiRe: Creating Interactive Pervasive Systems that cope with Changing Environments by Rewiring. In *Proceedings of the 4th IET International Conference on Intelligent Environments (IE'08)*, pages 1–8, 2008.

[13] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

[14] Fen Zhu, Matt W. Mutka, and Lionel M. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4:81–90, 2005.