Adaptation Support for Stateful Components in PCOM

Marcus Handte[1], Gregor Schiele, Stephan Urbanski, Christian Becker

Institute of Parallel and Distributed Systems
Universität Stuttgart, Germany
firstname.lastname@informatik.uni-stuttgart.de

Abstract. In ever-changing environments as they are envisioned in Pervasive Computing, applications have to adapt to changes in their execution environment. The automated composition of applications from components that are distributed across different devices is an adaptation technique that has been proposed by a number of researchers. While many approaches support the dynamic reselection of stateless components in order to adapt running applications, they often fall short of providing programmable solutions to automatically reselect components that carry application-specific state. In this paper, we discuss the requirements towards this support and we propose a framework that enables the (semi-)automatic reselection of PCOM components.

1 Introduction

Pervasive Computing envisions seamless support for complex user tasks by leveraging the capabilities of the environment. Over time the capabilities change due to user mobility or device failures. As a result, applications have to adapt to the capabilities of their ever-changing execution environment. To support developers in creating such adaptive applications, a number of researchers are focusing on the development of software infrastructures that automate the task of adaptation. One possible approach is the automated composition of applications from components that are distributed across different devices. Infrastructures such as PCOM [1] and P2PComp [4] can effectively determine a suitable composition at application startup and support adaptation by recomposing applications at runtime. However, these approaches are typically restricted to reselecting stateless components and do not provide special support for stateful components. In this paper, we discuss the requirements towards an extended adaptation support that facilitates the development of applications with stateful components. Based on these requirements, we present a framework that enables the automated recomposition of applications containing such components.

The remainder of this paper is structured as follows. The next section presents the underlying system model. Thereafter, we discuss the requirements towards adaptation support for stateful components. Section 4 outlines the framework that we have developed to enable such adaptations. Section 5 describes the framework integration in

[1] This work is funded by DFG Priority Programme 1140 – Middleware for Self-organizing Infrastructures in Networked Mobile Systems.

PCOM and Section 6 provides a short evaluation. Finally, Section 7 concludes the paper.

2 System Model

Our work focuses on peer-based environments, where mobile devices cooperate spontaneously as equal peers without the need of any external infrastructure support. Applications are assembled of software components that are provided by devices in the vicinity. Due to mobility, the availability of devices and their components is continuously fluctuating. New devices can arrive at any time, existing ones can be lost unpredictably and possibly permanently. To dynamically compose an application out of available components, we assume a suitable infrastructure such as RCSM [12], P2PComp [4] or PCOM [1]. For the sake of brevity, the discussion in this paper focuses mainly on our component system PCOM. However, the presented approach can be adjusted to other systems as well. In the following, we briefly sketch the relevant concepts of PCOM. A detailed description can be found in [1].

In PCOM, each device executes a component container, which manages all local components. Components are atomic with respect to distribution. The offered and required functionality of a component is described within a so-called contract. The attributes contained in contracts can be dynamically manipulated at runtime. PCOM's application model guarantees that each component instance is always used by at most one other component instance. Adaptation is supported by enabling instances to switch their used component instances at runtime. Both, container and components use our object-oriented middleware BASE [2] as communication platform. Therefore, all communication is performed through local proxy objects.

3 Requirements

From the presented system model, we can derive the following requirements towards adaptation support for applications with stateful components:

Decentralized operation: Peer-based systems cannot rely on the permanent presence of devices. Thus, the mechanisms used to support the adaptation of stateful components should not rely on the permanent availability of a central coordinating device. Instead, the mechanisms must support peer-based coordination.

Proactive state maintenance: While some communication technologies might be used to approximate the relative mobility of devices, e.g. by measuring the signal-to-noise ratio, a precise and reliable prediction of future disconnections is hard to achieve without additional capabilities, e.g., positioning systems, maps, movement profiles, etc. In order to be suitable for arbitrary environments, the adaptation support must be able to recover from unforeseeable disconnections.

Efficiency and minimalism: Ideally, adaptation support for stateful components should not introduce a performance overhead on component usage. Additionally, to support a broad range of devices it should be minimal regarding its resource requirements.

Tailorable automation: With respect to usability, it would be desirable to offer completely automated state maintenance. However, this often conflicts with requirements like efficiency and minimalism. Thus, a framework should be able to provide various degrees of automation that can be controlled by the component developer. Ideally, a component developer should be able to tune the automation of distinct tasks in order to achieve the best performance. Fully automation – despite its possible costs – should be offered in order to ease the programming of adaptive applications.

4 Approach

In order to support automatic adaptation of stateful components, the component system must be capable of automatically restoring the state of a replaced component at runtime. If the component system can predict the future unavailability of a component, it can proactively store the state of the component before it is replaced. To do this, the system interrupts the execution of the component, creates a consistent checkpoint, restores the checkpoint at the target component and continues the execution using the replaced component. In smart environments this technique is often used to support users that are roaming between different environments [9], [11]. Apart from smart environments, there are a number of other systems that utilize checkpoints to facilitate adaptation. The Condor system for example uses checkpoints to enable the migration of UNIX processes [8] and the one.world system uses checkpoints to support the mobility of so-called environments [5].

In cases where the future unavailability of a component cannot be predicted in a sufficiently accurate manner, the component system must be capable of restoring a component's state at any point in time. A brute-force way to enable this is the creation of a checkpoint whenever the state of the component changes. However, since the device that creates the checkpoint can become unavailable, the checkpoint must be transferred to some other device upon every creation. Thus, this approach clearly introduces massive communication overhead. While there is no general solution to this problem, there is a solution that can handle an important class of components, namely components that behave like state-machines [10]. Since the state of such components depends solely on the sequence of invocations that have been invoked on them, their state can be restored by replaying the same sequence. This concept of invocation histories is also used in mobile computing [6] and database systems [3], where fault tolerance and the preservation of state is realized through transactional message logs that are stored on a reliable server.

Our framework for (semi-)automatic adaptation is based on these concepts as it combines a transparent checkpointing facility with an automatically generated invocation history. The major difference to other approaches results from the fact that peer-based systems cannot rely on the permanent presence of a reliable server. Thus, checkpoint and invocation history must be stored on the device that uses the stateful component. This ensures that the state of the used component can be restored even if the device that executed the component has become unavailable. The component system stores invocations that are invoked on a component and occasionally creates checkpoints to prune the invocation history. Whenever the component is replaced, the

system restores the latest checkpoint and replays any outstanding invocations before the usage continues.

To create an invocation history, the component system stores a serialized copy of all outgoing invocations. This is done on the caller-side at the time when the invocation is marshaled by the middleware. To determine the execution order between different application threads, the component system on the callee-side assigns sequence numbers whenever an invocation is executed. This sequence number is later on transparently transferred to the caller-side together with the result of the invocation. Upon arrival, the system uses the sequence numbers to create a causally ordered history.

To support checkpointing, we first interrupt the component execution by transparently blocking all incoming invocations. Thereafter, we access the internal state of the component and transfer the checkpoint to the using component. There, the checkpoint is stored for later usage and the history is automatically pruned. While we could have implemented the access to the internal state in a transparent manner, e.g. by using Java object serialization, we did not want to break the encapsulation of components. Especially, we wanted to enable component developers to create different component implementations that use interchangeable checkpoint representations. Thus, we added an interface that provides methods to read and write state. This enables developers to manually map to different representations.

Since the component system automatically determines valid component replacements, it needs to be able to determine whether a component can make use of a certain type of checkpoint. As this checkpoint type is determined by the used component, it can vary at runtime depending on the latest binding. We reflect this fact by introducing a dynamic type attribute in the contractual descriptions of components. The maintenance of this attribute is automatically performed by the system.

Finally, to fully automate the creation of checkpoints, the component system must determine a suitable point in time to create them. Some possible parameters for the automatic creation of checkpoints are the size of the invocation history, the execution time of the invocations in the history or the cost creating the current component state. While we have performed a number of experiments with automatically created checkpoints, we currently do not have a solid metric that would be worthwhile discussing. Thus, we leave this as a subject for our future research.

The mechanisms described above are sufficient to fulfill the requirements towards automation, decentralized operation, proactive state maintenance and minimalism. However, in order to support efficiency, we added three access methods that enable component developers to interface with these mechanisms. First, we enable component developers to manually manipulate the history. Thus, they can decide whether a certain invocation should be stored. This is especially useful for idempotent methods, e.g. getters. Second, we enable developers of stateful components to signal points in time, when the creation of a checkpoint would be beneficial. This enables component developers to balance the tradeoff between histories and checkpoints towards arbitrary optimization goals. Finally, we enable component developers to manually force the creation of a checkpoint. Thus, if a component developer can foresee future adaptations, this can be used to speed up the state restoration.

5 Integration

In order to integrate the framework into our component system PCOM, we did not have to make any changes to the original component model besides the additional methods to support fine-tuning. Instead, most of the functionality is integrated into interceptors that are hooked into the proxy objects of our middleware as shown in Figure 1.

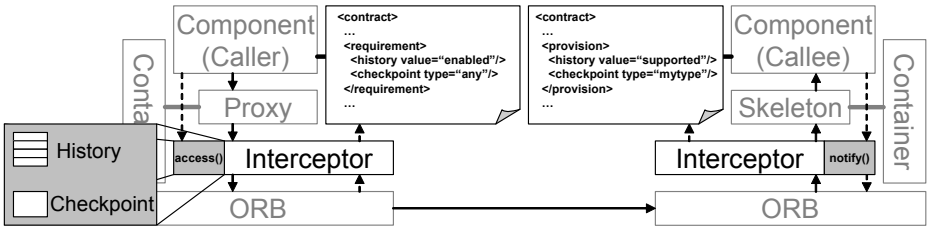

Figure 1. Framework Integration

The calling component can access the interceptor of the proxy in order to manipulate the history and to create checkpoints. Additionally, the interceptor allows the registration of listeners that deliver checkpoint hints provided by the callee. This enables the callee to signal points in time when the creation of a checkpoint would be beneficial. The callee can access the interceptor of the skeleton in order to issue these hints. Both interceptors have access to the contracts of the corresponding components which enables them to retrieve and set the type information of the checkpoints.

Whenever a checkpoint is created, the interceptor of the proxy retrieves its type and ensures that as long as a checkpoint is stored, the type of the checkpoint is reflected as a requirement towards components that can be bound to the proxy. Whenever the bound component is reselected, the interceptor will transfer and restore the state before the proxy can continue using the replacement.

6 Evaluation

To evaluate our approach, we performed experiments and developed some exemplary applications. In the following, we present measurements of the resource requirements and the additional overhead introduced by our framework. After that, we discuss experiences that we made during application development.

To evaluate the resource consumption of the invocation history we measured the time and space requirements of remote calls with varying size. The presented numbers are average results of performing 100 remote calls on two P3 / 700MHz over a 10 MBit LAN. The total number of runs was 30 per measurement. The resulting deviations can be neglected. The results presented in Figure 2 show that the overhead for creating sequence numbers and serializing the message is small compared to the overall time of the remote invocation (~5%). Regarding the space requirements, the introduced overhead is linear with respect to the stored invocation size.

During application development we found that our framework is able to considerably lessen the development effort for reselectable stateful components. As an example, we developed a small application that is able to present Microsoft PowerPoint presentations on a remote system. As the user moves, the application reselects the currently used output component to display the presentation in the user's direct vicinity. The state of an output component consists of the currently presented slide. The framework was able to fully automate the reselection of output components without additional development overhead, allowing faster and less error prone development.

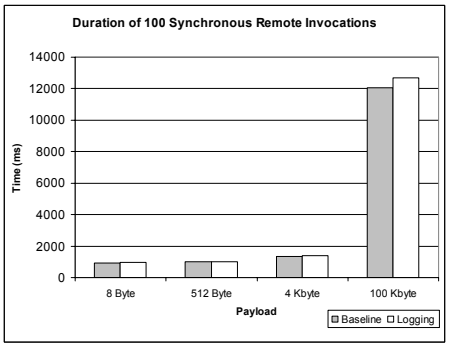

Figure 2. Overhead for Invocation Logging

After the initial development was finished, we used the framework's ability to customize its behavior in order to decrease the resource consumptions of the application. In our example, the framework stored all invocations send to the output component, leading to a history of presented slides. In this application, however, the state of the output component solely depends on the last shown slide. Therefore, we tuned the framework to only log the last invocation that changed the presented slide and deactivated checkpointing altogether. This not only reduces the memory requirements of the history but also minimizes the number of remote calls.

7 Conclusion

In this paper we have presented an integrated framework that enables the (semi-) automated reselection of stateful components in peer-based systems. This framework combines the concepts of checkpoints and invocation histories. To support the dynamics of peer-based environments, the framework is capable of restoring the state of a component automatically, even if the (possibly permanent) unavailability of a device cannot be predicted. Additionally, the presented framework enables application developers to manually fine-tune its internal mechanisms. While manual fine-tuning can increase efficiency in many cases, our experiments indicate that the automatic solution is especially helpful during rapid prototyping. In the future, we plan to analyze different metrics to increase the efficiency of our fully automated adaptation support.

References

[1] Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM – A Component System for Pervasive Computing, 2nd Intl' Conference on Pervasive Computing and Communication, 2004

[2] Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: BASE – A Micro-broker-based Middleware for Pervasive Computing, 1st Intl' Conference on Pervasive Computing and Communication, 2003

[3] Date, C. J.: An Introduction to Database Systems, 7th Intl' Edition. Addison Wesley Longman (2000) 443-451

[4] Ferscha A., Hechinger M., Mayrhofer, R.: A Light-Weight Component Model for Peer-to-Peer Applications, 24th Intl' Conference on Distributed Systems Workshop (ICDCSW'04) (2004) 520-527

[5] Grimm, R.: System support for pervasive applications. PhD Thesis, University of Washington (2002)

[6] Joseph, A.D., Tauber, J.A., Kaashoek, M.F.: Building reliable mobile-aware applications using the Rover toolkit, 2nd ACM Intl' Conference on Mobile Computing and Networking. (1996)

[7] Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management, IEEE Transactions on Software Engineering, Vol. 16(11). (1990) 1293-1306

[8] Litzkow, M., Tannenbaum, T., Basney, J., Livny, M.: Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison. (1997)

[9] Roman, M., Ho, H., Campbell, R.: Application Mobility in Active Spaces. 1st Intl' Conference on Mobile and Ubiquitous Multimedia. (2002)

[10] Schneider, F. B.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Computing Surveys, Vol. 22(4). (1990) 299-319

[11] Sousa, J. P., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. 3rd IEEE/IFIP Conference on Software Architecture. (2002)

[12] Yau, S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S.: Reconfigurable Context-Sensitive Middleware for Pervasive Computing, IEEE Pervasive Computing, Vol. 1(3). (2002) 33-40