# Towards the Description and Execution of Transitions in Networked Systems

Alexander Frömmgen[1(✉)], Björn Richerzhagen[2], Julius Rückert[3], David Hausheer[3], Ralf Steinmetz[2], and Alejandro Buchmann[1]

[1] Databases and Distributed Systems, TU Darmstadt, Darmstadt, Germany
{froemmge,buchmann}@dvs.tu-darmstadt.de
[2] Multimedia Communications Lab, TU Darmstadt, Darmstadt, Germany
{bjoern.richerzhagen,ralf.steinmetz}@kom.tu-darmstadt.de
[3] Peer-to-Peer Systems Engineering Lab, TU Darmstadt, Darmstadt, Germany
{rueckert,hausheer}@ps.tu-darmstadt.de

**Abstract.** Today's distributed systems have to work in changing environments and under different working conditions. To provide high performance under these changing conditions, many distributed systems implement adaptive behavior. While simple adaptation through parameter tuning can only react to a limited range of conditions, a switch between different mechanisms at runtime enables broader adaptivity. However, distributed systems that switch mechanisms at runtime lack a clear abstraction for the adaptive behavior and, thus, usually interleave the adaptation and actual application logic. This leads to complex and error-prone systems that are hard to maintain and not easy to extend.

In this paper, we analyze the adaptations of two distributed systems from different application domains. We identify recurring requirements as well as life cycles of transitions between mechanisms. Based on this, we present a framework that provides a clear abstraction of the underlying transition logic and manages the transitions at runtime. The concept is applied to the two example systems to practically evaluate its benefits. We show that our approach leads to less complex realizations of the adaptive behavior and allows new mechanisms to be integrated easily.

## 1 Introduction

Today's distributed systems operate in challenging environments with rapidly changing working conditions. In order to provide high performance in such dynamic environments, systems need to be highly adaptive. However, simple adaptations by means of configuration parameter adjustments can only react to a limited range of conditions. As distributed systems can be described as a combination of multiple functional blocks, in the following called *Mechanisms*, the ability to switch between mechanisms at runtime and the possibility to extend the system by novel mechanism, proved to provide greater flexibility and enables the system to adapt to a wider range of environmental conditions [14,18]. In the following, such switches between mechanisms are referred to as *Transitions*.


S. Latré et al. (Eds.): AIMS 2015, LNCS 9122, pp. 17–29, 2015.
DOI: 10.1007/978-3-319-20034-7_2

While this idea is appealing, so far, there exists neither a systematic approach of how to generically integrate multiple mechanisms in a complex distributed system, nor how to systematically execute and coordinate transitions between mechanisms at runtime. The latter is especially challenging as it requires a deep understanding of mechanism life cycles and a generic approach for state transfers between mechanisms in transition. As adaptivity has complex implications for the overall system, an explicit design of transition-enabled adaptive behavior is necessary and essential. So far, existing adaptive distributed systems lack a clear *separation of concerns* between the logic realizing adaptivity (the transition logic) and the actual application logic. A general concept to design and implement systems that allow for adaptivity in all key aspects is missing so far.

In this work, we propose a formal description of adaptivity through transitions and provide system developers with a sophisticated framework to describe and realize mechanism transitions in distributed systems. The approach allows mechanisms to be easily added over time to address new scenarios. Furthermore, the framework enables a clear separation of concerns between the transition and application logic. To identify the relevant functionality for the framework, two state-of-the-art systems (Bypass [14] and Transit [18]) targeting different application domains are analyzed. Even though both systems proved already that they highly benefit from adaptive behavior, a systematic approach to describe this behavior is missing so far. Based on the results, a first general design for the execution of transitions is derived and applied to the analyzed systems. By following the presented design, the mechanisms still define their application-specific interfaces. Additionally, transition-enabled mechanisms follow a common life cycle approach, managed by the framework. This allows us to (i) define *Elementary Transitions* between mechanism instances that can be performed during runtime and (ii) easily add new (specialized) mechanisms and transitions to an existing system. First proof-of-concept applications and initial evaluation results show that the approach is well-suited to describe and enable generic adaptivity in distributed systems by supporting mechanism transitions. As this work concentrates on the description and execution of transitions, the highly domain-specific adaptation decisions (based on performance models such as queuing models) are not discussed. However, our work poses a first step towards a generalization of the decision process by providing a unified description and handling of transitions in distributed systems.

The remainder of the paper is structured as follows: Section 2 presents the analysis of two existing adaptive systems and the derivation of requirements for the transition execution framework. The developed methodology, the derived life cycles and the framework is presented in Section 3 and discussed in Section 4. Subsequently, Section 5 presents related work and Section 6 concludes the paper.

## 2 Requirements Derivation

In this section, we analyze two state-of-the-art distributed systems and derive requirements for a transition framework.

### 2.1 Analysis of Bypass

Bypass [14] provides low-latency event dissemination for interactive mobile augmented reality games. In the targeted application scenario most events generated by mobile players are only relevant within a given area of interest around the player's current position. Therefore, once groups of nearby players are detected, the system adapts to local event delivery via ad hoc dissemination protocols. Depending on relevant environmental conditions (e.g. the group size), the local dissemination protocol is reconfigured or replaced. The original paper showed the benefit of adaptations to two local dissemination protocols: (i) a range-limited broadcast scheme and (ii) a probabilistic, gossip-like scheme. The authors further mention the possibility to switch between physical layer protocols, such as Wi-Fi ad hoc and Bluetooth. This leads to an architecture with a set of potential compositions as shown in Figure 1.

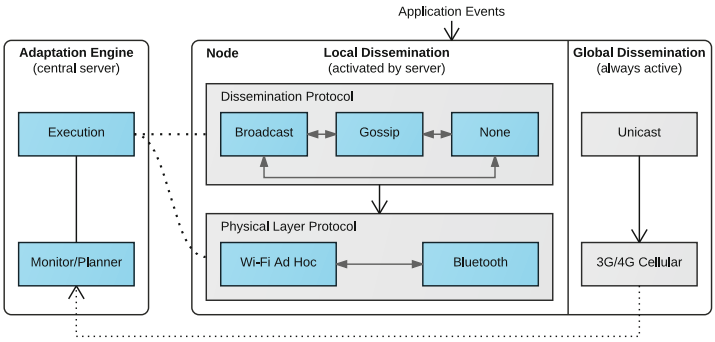


**Fig. 1.** Overview of Bypass and the possible transitions

The reconfigurations, i.e. the transitions between the local dissemination protocols, are not explicitly modeled in [14]. Instead, they are interweaved with the application logic and part of a predefined adaptation strategy at the cloud-based controller. Thus, no existing implementation for the recurring handling of transitions is used. This raises concerns as to how maintainable the adaptation implementation is, in particular how feasible it is to add new dissemination protocols, and how to implement the complex switch between these at runtime. We argue that a systematic methodology and a framework which supports this with well-defined interfaces for transition-enabled mechanisms significantly reduces the complexity of implementing and extending an adaptive distributed system.

Furthermore, the authors show that some nodes utilize different dissemination protocols. This causes approximately 40% of the overall message loss. We therefore propose an explicit life cycle management for the mechanisms involved in a transition. The coordinated life cycle has to enable a smooth handoff between two mechanisms, thereby reducing the amount of lost events caused by the transition itself.

The additionally possible transitions between Wi-Fi ad hoc and Bluetooth impose challenges to the existing transition handling. First of all, the establishment of the Bluetooth connection requires at least a short period of time, which causes an increased latency. Second, the establishment of the connections might fail for multiple reasons, which has to be compensated by the application. Switching between Wi-Fi and Bluetooth also affects the currently utilized dissemination protocol. Furthermore, sophisticated protocols require a specific physical layer protocol as they rely on assumptions regarding communication characteristics such as range or reliability. Therefore, we propose to build compositions of transitions to reflect dependencies, e.g. a transition between physical layer protocols might depend on a transition of the dissemination protocol.

As the aforementioned challenges are recurring for transitions in adaptive distributed systems, they should be handled by a transition framework to reduce the complexity. Depending on the mechanisms, state such as routes or neighbor tables are maintained that could benefit the target mechanism. Transferring state requires domain-specific knowledge, while conceptually being part of the transition logic. A transition framework needs to provide ways to utilize such knowledge in the transition handling to benefit the overall system.

### 2.2 Analysis of Transit

The streaming system Transit [18] realizes an overlay-based multicast service that adapts to a wide range of working conditions. For this, a main feature of the system is to flexibly adapt its overlay topology, which is used to distribute the video streams among participants in the system. The adaptivity is realized by different sets of *topology optimizations* (cf. Figure 2). These optimizations are usually executed in a distributed manner at the participants, also called peers in the following. While Transit includes several other mechanisms that realize adaptivity, including an extension [13] for network-layer multicast, in the following, we focus on overlay topology optimizations as they promise to have great potential for the adaptation considerations in this work.

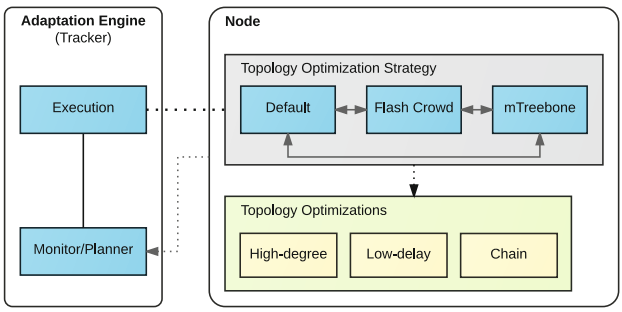


**Fig. 2.** Exemplary mechanisms and transitions in Transit

A challenging use case for topology adaptations are flash crowds, which are common to large-scale (video) streaming events. A large number of users needs to be quickly connected to the system. While in non-flash crowd times complex optimizations are reasonable to gradually improve the delivery process, here it is essential to quickly attach new peers to the topology and serve them with a low startup delay. One way to achieve this is to build topologies where many peers have free upload slots. Complex topology optimizations that could interfere with the attachment process should be disabled during the massive join phases of a flash crowd. Both can be achieved by defining topology optimizations strategies that define which optimizations are run. Therefore, as the topology adaptation logic is already modularized, this part of the system could greatly benefit from well-defined interfaces for transition-enabled components. Even though transitions between topology optimizations seem less complex as transitions in Bypass, a unified transition model which covers recurring challenges would improve the current systems and allow to build more complex systems easily.

## 3 Design and Description Methodology

### 3.1 Analysis of Transition Life Cycles

The systematic development of adaptive distributed systems requires abstractions. In the following, the *Elementary Transition* is introduced as the basic building block. Therefore, the *Life Cycle* of an elementary transition is defined based on the life cycle of the involved components. As the execution of a mechanism (e.g. Bypass's dissemination protocol) can be distributed over multiple nodes, we use *Components* as the smallest unit which is exchanged through a transition. A component implements all mechanism logic that is executed on a single node. Thus, the *Broadcast* components at multiple nodes together represent the *Broadcast Dissemination* mechanism in the distributed system.

To allow the application developer to abstract from the transition logic, a proxy [8] component is used instead of the actual component. This proxy intercepts all method invocations and forwards them to the currently active component instance. This enables a clear separation of concerns, as it strictly separates the application logic from the transition logic. The application only interacts with the proxy. A transition affects the inner workings of the proxy while leaving the reference to the proxy and the interface for the application unchanged. The proxy hides the exchange of the *Source* component with the *Target* component[1]. Additionally, the proxy instance ensures a thread-safe transition in a multithreaded environment. A transition between two components, for example, might not be executed while one thread is executing the source component (more precisely: while at least one method of the source component instance is part of a current execution stack). Similar problems might occur for application layer protocols, e.g. finishing a communication sequence before executing the transition. This follows the idea of *quiescence* as discussed by Pissias et al. [12].

[1] In the following, *component* is used instead of *component instance*.

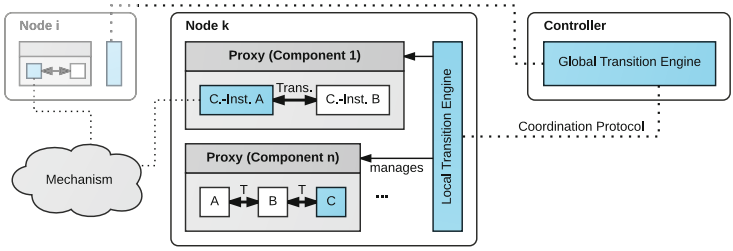


**Fig. 3.** Mechanisms comprise concurrently executed components on multiple nodes, each managed by a proxy instance

Figure 3 illustrates the overall architecture with multiple nodes, each executing components managed by proxies. The proxies are managed by the *Local Transition Engine*, which in turn is coordinated by the *Global Transition Engine*. Components on different nodes interact, thereby forming a mechanism.

Our analysis of the existing adaptive systems and their requirements lead to two kinds of elementary transitions, the *runrun* and the *flip transition*. Their most important difference is the life cycle and the parallel usage of the involved components. The flip transition does not execute both components in parallel and, thus, executes a hard switch, whereas the runrun transition smoothes the transition and executes both components in parallel for a short period of time, enabling a handover of, e.g., protocol state. Even though the runrun transition seems to be favorable, the flip transition is sometimes beneficial or even required due to resource constraints. For example, most mobile devices do not support the parallel usage of infrastructure Wi-Fi and Wi-Fi ad hoc, the parallel usage of Wi-Fi and LTE causes a higher energy consumption, and exclusive used software handles, e.g. established socket connections, cannot be shared. In the following, we discuss both transition types in detail.

**Flip Transitions.** The flip transition executes an abrupt switch between the source and target component. Figure 4 shows the life cycle of a component for such a transition. We model the life cycle as a mealy state machine which exchanges messages between the coordinating instance (the transition engine) and the component. Therefore, the state transitions of the state machine are annotated with the triggering messages (the input) which are sent to the component (above the bar) and the expected answer when the new state is reached (below the bar). For example, the transition engine signals an *init* to a newly created component. Once the component followed its internal initialization steps, it signals *finished* back to the transition engine leading to the *Initialized* state. During the initialization, the component executes preparation tasks such as allocating memory. The proxy ensures that methods of the component are not yet invoked. After the component has started successfully, and therefore can use the shared resources, it reaches the *Running* state and is used by the application.

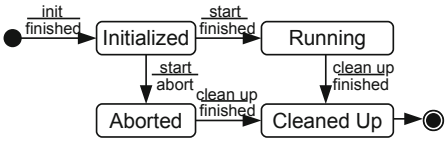


**Fig. 4.** Single component life cycle of a *flip* transition

In case the component start fails, e.g. due to a network connection failure, it reaches the *Aborted* state. From the software engineering point of view, the actual *constructor* logic is distributed among the *init* and the *start* state transition.

Based on the component life cycle, we specify the life cycle of the flip transition (Figure 5). We denote messages exchanged with the components with an according prefix (*src:*, *trg:*). The *parent:*-prefix is used for message exchanges with the local transition engine as discussed later in this section. As the start of the target component may fail, the *Initialized* state has two outgoing state transitions. In case of a success, the flip transition finishes. Otherwise, the local transition engine decides, based on the specification of the application developer, if a new *rec*overy component instance should be initialized or the transition fails. Please note that the details of these steps are skipped in the figure due to clarity.

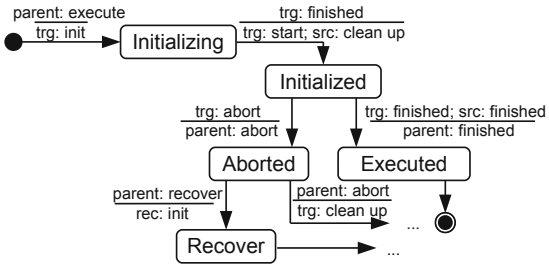


**Fig. 5.** Overall life cycle of a *flip* transition

**Runrun Transitions.** To reduce the performance degradation during the transition and allow a smooth transition, the runrun transition deactivates the source component when the target component is already running. This interweaving requires both components to be executed in parallel. The component life cycle reflects this with additional states (cf. Figure 6). The main difference is the state transition to the *Active* state and, thus, to the internal *Running* state. The *Active*-state additionally contains an *In Shutdown* state. Methods of the component can be executed during both internal active states. As the state transition to the *Running* state may fail, there is a state transition to the *Aborted* state as well. Additionally, the source component can be triggered to return to the *Running* state. In the *Active* state, the component can execute a state transition to the *Cleaned Up* state and clean up.

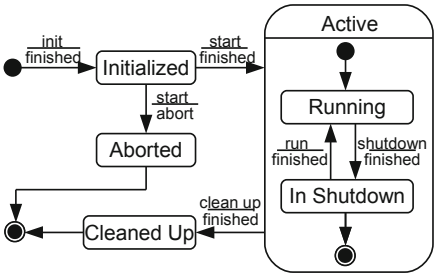


**Fig. 6.** Single component life cycle of a *runrun* transition

Based on this component life cycle, we define the life cycle of an elementary runrun transition (Figure 7). As the source component can be used in the *In Shutdown* state, the proxy can always forward method invocations, and therefore never blocks the application. This holds even for the time during the state transitions to the *Parallel Active* and the *Aborted* state. In case the start of the target component fails, the transition can always return to the source component. As the local transition engine coordinates the execution of a transition, it triggers the state transition to the *Clean up* and the *Rollback* state. Even though the component life cycle introduces a strict contract for the components, it allows easily integrating new mechanisms.

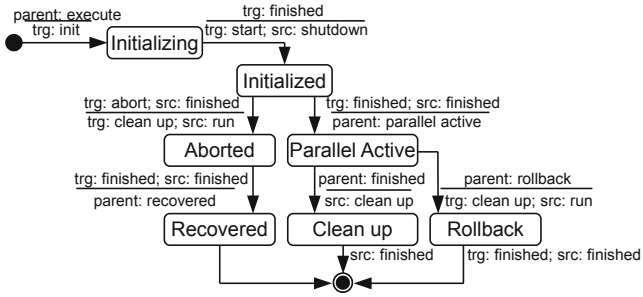


**Fig. 7.** Overall life cycle of a *runrun* transition

### 3.2 A Framework for Transition Description and Execution

Based on the two identified transition types and their life cycles, a systematic description and framework support for transitions is possible. In the following, we propose a transition description language, possible framework support, and suitable visualizations.

The transition description language allows to specify all available transitions and their affected components, as illustrated in Listing 1.1. As typical use cases have only a limited number of possible transitions, the exhaustive description of all transitions is without difficulty. Based on the described transitions, a *Global*

*Transition Engine* coordinates *Local Transition Engines* to execute distributed transitions. As both analyzed distributed systems have a centralized instance (the Cloud Server for Bypass and the Tracker for Transit), the assumption of a global transition engine is reasonable.

The coordination and synchronization of the proxies can be implemented efficiently. As in the two example applications multiple nodes execute the same transitions, the global transition engine does not require a huge and complex state machine (e.g. a Cartesian product of the involved life cycle states). Instead, a single state at the global transition engine can represent multiple states of the nodes. For scalability reasons, the global transition engine could use sub coordinators to handle the coordination messages. As this work focuses on the description and execution of transitions, a central coordinator is assumed. However, for future work, the methodology could be extended to support applications with a decentralized transition control.

The transition engines use the additional information from the transition description to control the transition execution. For example, the time the transition stays in the parallel active state (line 4), and the timeout which leads to a rollback of the transition (line 7) can be specified. In case a 'best effort' approach is sufficient, it is also possible to partly deactivate the transition coordination.

During both kinds of transition, state from the source component can be transferred to the target component. This state is represented by the references of the component and class instances which the source component transfers to the target component. As the transitions in the two analyzed systems both transfer this state in the life cycle state transition to the *Running* state, the framework makes this state transfer explicit and supports it in the transition description language (Listing 1.1 line 3). Components which only implement the stateless strategy pattern [8] do not require state transfer.

```
1  define elementary transition elTransition
2      from ComponentA to ComponentB type runrun
3      transfer state myStateVariable
4      parallel active for 1 minutes;
5  define parallel transition paraTransition
6    foreach A execute elemTransition
7      at least 90% timeout 2 minutes;
```

**Listing 1.1.** Example of the transition description language

Additional domain specific transition logic can be added in the host language and is invoked by the transition engine during the life cycle events. Such additional transition logic can, for example, be used to perform more complex state transfers involving the transformation of data. Even though both evaluated systems are based on Java, the presented approach is agnostic of the utilized programming language. The explicit notation of the transitions allows a clear separation between components and transitions and, this way, enables a systematic development of distributed adaptive systems. A framework which explicitly

supports transitions between components and manages their execution provides multiple additional benefits. Besides development and debugging support, it can, for example, monitor and ensure that only one transition per proxy instance is executed at the same time, and started transitions finally terminate.

Traditionally, there are two viewpoints on systems, the structural and the behavioral [9]. The composition of components describes structural properties, whereas the execution of a transition is behavior which changes the structure. We propose a combined view as a class diagram (Figure 8). The ternary association connects the source and the target component with the specific transition logic. The stereotypes *TransitionEnabled* and *TransitionLogic* are resolved to implementations of the corresponding interfaces. The transferred state (more precisely, the transferred references to other objects) is visualized as well.

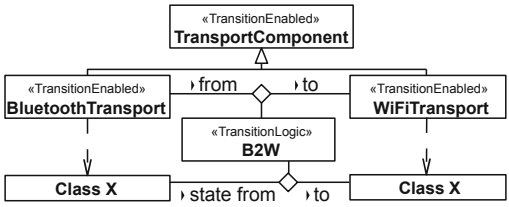


**Fig. 8.** UML visualization of transitions and the composition variability of components

## 4 Discussion

To assess the impact of the proposed design methodology, the presented concepts were applied on both systems. For the existing transitions between local dissemination protocols in Bypass, the new version which leverages the framework reduces the transition specific lines of code to roughly 20% of the original code. This greatly reduces the complexity of the implementation. Even more important, the transition-specific source code is no longer part of the application logic but instead encapsulated by the transition engine, providing separation of concerns. Additionally, this reduces the error probability, as proved transition handling code is reused. In addition to porting Bypass to our framework, we added new component that encapsulate the physical layer protocols as proposed by the authors. Therefore, the existing interaction via Wi-Fi ad hoc was realized as a component instance as shown in Figure 1. Additionally, we added a component instance that enables local communication via Bluetooth and defined a runrun transition between both instances (cf. Listing 1.2).

Our implementations show that Bypass benefits from the concept of parallel and sequential transitions. Switching the local dissemination protocol after the physical protocol switched successfully, is defined as a sequential transition (Listing 1.2), composed of two elementary transitions. In case the connection setup using Bluetooth fails, the overall transition returns to a well-defined system state. Considering the life cycle of a *runrun* transition (Figure 7), the transition enters the *Aborted* state and consequently recovers by just continuing to use Wi-Fi as

active component instance. Composing transitions out of elementary transitions is straight forward with the presented framework, as they can easily be described in the transition description language. Thereby, common pitfalls such as illegal combinations or undefined transitions are detected early. The abstraction of the underlying life cycle enables more complex combinations. For future work, it is interesting to combine constraints on the possible component compositions and properties of the compositions, e.g. as proposed by [7], to further support the developer, e.g. verify legal transitions.

```
1  define elementary transition WiFi2BT
2      from WiFiAdHoc to Bluetooth type runrun;
3  define elementary transition Gossip2BC
4      from Gossip to Broadcast type runrun;
5  define sequential transition SeqTrans (WiFi2BT, Gossip2BC);
```

**Listing 1.2.** An elementary transition from Wi-Fi to Bluetooth composed with a subsequent dissemination protocol transition

Applying the concept to transitions between topology optimization strategies in Transit was possible in a straightforward manner with very localized changes at the respective components. As an extension, the concept of so called *Optimization Providers* was introduced to bundle sets of currently active topology optimizations. The transition framework was then used to allow for *flip* transitions between different provider implementations. As topology optimizations and their execution are decoupled by design, there was no need to consider optimizations and providers that are active in a parallel manner. Providers that reuse topology optimization instances (in case two providers include similar optimizations) benefit from the automatic state transfer of the framework. As a side effect, the approach showed to be very helpful to rapidly define and evaluate new optimization combinations for new scenarios. With the legacy implementation, this requires large changes to various parts of the optimization implementations, thus heavily mixing adaptation and application behavior.

## 5 Related Work

Compositional adaptation [11] is an established concept for adaptive systems. Many systems [5, 15, 16] leverage proxies to enable transparent dynamic composition for the application or extend the host language to enable adaptive behavior [10]. Sadjadi et al. [15] use this concept for communication systems and adapt between different forward error correction schemes in the network. However, these systems lack a model for the actual transition life cycle and state transfer. Additionally, they do not provide abstractions for multiple (parallel or sequential) transitions, even though their proxies can be distributed.

Many component-based systems benefit from an explicit component life cycle. In OSGi [2], for example, bundles and their components follow a clear life cycle. This enables OSGi to install, update, remove, start, and stop components at

runtime without stopping the system. The different components are loosely coupled using service bindings. Even though our presented component life cycle is inspired by the bundle life cycle in OSGi, we consequently enhance this concept to model the life cycle of *transitions between components* in distributed environments. This reduces the switching costs and allows transfer state and composed transitions. The handling of service bindings is hidden by the proxies.

IBM proposes the MAPE-K concept for autonomic control loops [1]. In this model, the *Autonomic Manager* manages the tasks *monitor*, *analyze*, *plan*, and *execute* as well as the globally shared *knowledge*. *Effectors* perform changes on *Managed Elements*, which represent any adaptive software/hardware component. Our proxies represent a special of managed element with clearly modeled transition capabilities. The transition description is globally shared knowledge.

Ferreira et al. [6] developed A-OSGi to support the construction of autonomic OSGi-based applications. Therefore, they integrate the MAPE-K approach into OSGi. Concentrating on the monitoring, the actual adaptations are simple service binding changes, bundle starts, or changes of service properties. A detailed concept for more complex component exchanges and state transfer is missing.

The knowledge about valid configurations and component compositions is important to deal with reconfiguration consistency. Batista et al. [3], for example, model this and use it to check reconfigurations, but do not investigate the actual transitions which lead to these configurations. As our current solution concentrates on the description and the life cycle of transitions, integration of both approaches might allow to reason about consistent and legal transitions.

Coulson et al. [4] propose a middleware approach for reconfigurable distributed systems. The middleware is utilized in the publish/subscribe system GREEN [17] to enable reconfiguration of multiple publish/subscribe-specific components. However, they do not provide a clear description of the reconfiguration possibilities. Transitions in between configurations and implications such as state transfer and life cycle management are not detailed.

## 6 Conclusion and Future Work

In this paper, we identified recurring requirements to handle adaptive behavior in distributed systems. Based on this, we proposed a framework which supports two transition life cycles and provides a clear abstraction of the underlying complex transition logic. The concept was applied to two example systems to practically evaluate its benefits. In both cases, the approach leads to less complex realizations of adaptivity and allows new mechanisms to be integrated easily.

For future work, we consider to leverage the proposed concept and methodology to design and study a new class of highly adaptive distributed systems that were not feasible so far. While the proposed approach builds the foundation for this, additional support for more dependencies between transitions and according coordination strategies, e.g. a decentralized coordination, need to be developed as next steps.

**Acknowledgments.** This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 – MAKI.

## References

1. An Architectural Blueprint for Autonomic Computing. Tech. rep. IBM (2003)
2. OSGi Service Platform Core Specification (2007)
3. Batista, T.V., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
4. Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N.: The Design of a Configurable and Reconfigurable Middleware Platform. Distributed Computing 15(2) (2002)
5. Felber, P., Garbinato, B., Guerraoui, R.: Towards reliable CORBA: Integration vs. service approach. Tech. rep. dpunkt-Verlag (1997)
6. Ferreira, J., Leitão, J., Rodrigues, L.: A-OSGi: A Framework to Support the Construction of Autonomic OSGi-Based Applications. In: Vasilakos, A.V., Beraldi, R., Friedman, R., Mamei, M. (eds.) Autonomics 2009. LNICST, vol. 23, pp. 1–16. Springer, Heidelberg (2010)
7. Frömmgen, A., Lehn, M., Buchmann, A.: A property description framework for composable software. In: Avgeriou, P., Zdun, U. (eds.) ECSA 2014. LNCS, vol. 8627, pp. 267–282. Springer, Heidelberg (2014)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Pearson (1994)
9. Hilliard, R.: Recommended Practice for Architectural Description of Software-intensive Systems. IEEE Std. 1471-2000 (2000)
10. Kasten, E., McKinley, P., Sadjadi, S., Stirewalt, R.: Separating Introspection and Intercession to Support Metamorphic Distributed Systems. In: Distributed Computing Systems Workshops (2002)
11. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. Computer 37(7) (July 2004)
12. Pissias, P., Coulson, G.: Framework for Quiescence Management in Support of Reconfigurable Multi-threaded Component-based Systems. IET Software 2(4) (2008)
13. Rückert, J., Blendin, J., Hausheer, D.: Software-Defined Multicast for Over-the-Top and Overlay-based Live Streaming in ISP Networks. Springer JNSM, Special Issue on Management of Software-defined Networks (2014)
14. Richerzhagen, B., Stingl, D., Hans, R., Groß, C., Steinmetz, R.: Bypassing the Cloud: Peer-assisted Event Dissemination for Augmented Reality Games. In: Proc. P2P. IEEE (2014)
15. Sadjadi, S.M., McKinley, P.K., Kasten, E.P., Zhou, Z.: MetaSockets: Design and Operation of Runtime Reconfigurable Communication Services: Experiences with Auto-adaptive and Reconfigurable Systems. Softw. Pract. Exper. 36(11-12) (2006)
16. Sadjadi, S.M., McKinley, P.K.: ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation. In: Distributed Computing Systems. IEEE (2004)
17. Sivaharan, T., Blair, G.S., Coulson, G.: GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. In: Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, pp. 732–749. Springer, Heidelberg (2005)
18. Wichtlhuber, M., Richerzhagen, B., Rückert, J., Hausheer, D.: TRANSIT: Supporting Transitions in Peer-to-Peer Live Video Streaming. In: IFIP NETWORKING (2014)