Adaptive and Fault-Tolerant Service Composition in Peer-to-Peer Systems

Vivian Prinz[1], Florian Fuchs[2], Peter Ruppel[2], Christoph Gerdes[3], and Alan Southall[3]

[1] Group of Applied Informatics - Cooperative Systems, Institute for Informatics, Technische Universität München, Germany
[2] Mobile and Distributed Systems Group, Institute for Informatics, Ludwig-Maximilians-Universität München, Germany
[3] Siemens AG, Corporate Technology, Information and Communications, Intelligent Autonomous Systems

Abstract. Service-orientation enables dynamic interoperation of distributed services and facilitates seamless service provision or runtime creation of new applications. This dynamic service composition is particularly powerful in peer-to-peer (P2P) systems which offer scalability through self-management and autonomy. However, P2P service composition is nontrivial due to permanent peer churn and lack of central control. Existing approaches reduce composite service initialization to an NP-hard path finding problem. Thus, peer failure adaptation is costly and runtime consideration of peer logons or load changes is not practicable. This paper introduces logical peer groups for service composition. They enable runtime composite service reconfiguration including the migration of services to other peers. A prototype implementation is presented and the algorithms are evaluated through both formal and empirical analysis. The evaluation shows that the approach results in significant reduction of computational complexity, improves fault-tolerance and enables adaptation of logons and load changes which has not been possible so far.

Key words: adaptive, reconfigurable, self-managing, quality of service-aware applications, autonomic applications and systems, peer-to-peer computing, service composition, service-oriented applications

1 Introduction

In recent years the composition of services has been one of the major enablers for many IT companies. Different companies automate order and payment procedures. Portals, online maps or logistic applications offer information aggregated from different providers. In [1], composability is even called the *reason to be* for services because it allows them to be used for multiple purposes. In general, services can be composed statically or dynamically. They are either selected and put together once during composite service implementation, or selected and put together at runtime, i.e. on demand. The latter approach enables runtime integration of intermediate services and permits, for example, the adaptation of

current service usage context or the dynamic creation of new services out of existing ones. In large-scale networks, single services are furthermore offered by different providers. Dynamic service composition facilitates the selection of subservice providers with respect to parameters like availability, performance, load, monetary costs or quality of service (QoS). In this paper, we will call these parameters execution properties. Finally, subservices of a composite service can be exchanged during runtime. Consider a composite service comprising three subservices: one is an RSS reader that delivers text messages to a second subservice that translates the text into another language. The third subservice converts the translated text into speech the user can listen to. The user might want to hear the latest news using a PDA while driving a car. If, for example, the text-to-speech subservice gets overloaded meanwhile, the composite service can switch to a better performing entity which also offers a text-to-speech service.

Our focus is this dynamic service composition in P2P systems. Service providers as well as users are regarded as peers of a fully decentralized distributed system. That is, we assume that the network is not able to, or shall not, provide a central controlling component but is self-managed. Thus composite services can be provided without broker infrastructure and associated administration costs. Moreover, there is no central component that can become a bottleneck or even fail – essential when considering the evolution of distributed systems towards large-scale networks and accompanied scalability requirements. However, inter-peer service composition is nontrivial due to dynamic peer arrivals and departures (churn) implying high failure probabilities and the required decentralization.

Regarding related work, a lot of research has been done on optimized service selection and execution, for example in the fields of load balancing and context-aware computing. Concerning service composition, many central approaches exist, for example for grid environments or web services. Also for P2P systems some solutions have been proposed: *PCOM* [2], *A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids* [3] and *SpiderNet* [4] [5]. These approaches solve composite service initialization by regarding all possible service paths between all service providing peers. They show that the corresponding class of computational problems is NP-hard. This is basically due to the multitude of possible paths that have to be computed using distributed graph or tree algorithms. In addition, these paths have to be compared considering multiple constraints like QoS parameters. The only system that realizes fault-tolerant P2P service composition is *SpiderNet*. It utilizes a Distributed Hash Table (DHT) for decentralized information management and allows dynamic composition and proactive error detection for stateless services. Failures are compensated by migrating the composite service, i.e. the service path selected before, to a backup path. Backup paths are computed during initialization as well and are monitored at runtime using messages along the paths. However, migrating the whole service path on a single subservice failure is costly. None of the existing approaches is able to adapt peer arrivals or variations of execution properties because this requires expensive runtime re-initialization.

In this paper we describe a concept that supports adaptive and fault-tolerant dynamic service composition. Peer churn and changes of execution properties are detected at all times and may cause the migration of single services to other peers. The solution is realized by interacting and self-organizing peer groups and the underlying algorithms are based on nothing but local peer decisions. The remainder of this paper is structured as follows: Section 2 introduces our concept and explains the associated algorithms. Section 3 describes the prototype implementation of the system. Section 4 provides a formal analysis of the algorithms' computational complexity as well as an empirical evaluation of our approach. Section 5 concludes the paper and suggests future work.

2 Service Composition based on Interaction between Logical Service Groups

In this section, we describe our concept for adaptive and fault-tolerant service composition in P2P systems. Thereby, we assume the possibility to store, modify, delete and search for information within the P2P network. Example solutions for this discovery functionality are Pastry [6], Chord [7], CAN [8], Tapestry [10] or Freenet [11]. We refer to information as *resources* and to distributed storage as *resource publishing*. Furthermore, we assume that the P2P network is realized using DHTs and thus enables the implementation of a publish-subscribe mechanism. Publish-subscribe mechanisms are, for example, proclaimed in [12, 13]. They enable single peers to subscribe to a certain kind of resource. Hence, they are notified, if a resource of that type is published, modified or deleted.

2.1 Basic Idea: Logical Service Groups

As explained in Section 1, existing solutions for service composition in P2P systems are very static. Service and backup paths are computed only once during initial service composition and cannot be adapted at runtime because their recomputation is too costly. The idea for a more dynamic service composition solution is not to analyze all possible paths between the peers, but to regard all peers providing a dedicated subservice as a group. Such a *Logical Service Group (LSG)* is defined as the set of available peers that locally provide a dedicated subservice. During subservice execution, one peer of the LSG executes the subservice and n other group members monitor its *heartbeat*. The monitoring peers are called *watchdogs*. Heartbeats are continuous messages being sent to the watchdogs. They enable the watchdogs to detect if the executing peer has failed. Figure 1 illustrates the members of a LSG and their roles. All peers providing a subservice are group members and one of them is executing the subservice. The group is part of a composite service comprising 4 subservices. Data to process, for example RSS feeds, are forwarded between the groups. In general, every peer can be a subservice providing peer in multiple groups. It might be able to execute different subservices of a composite service and LSGs are a logical construct.

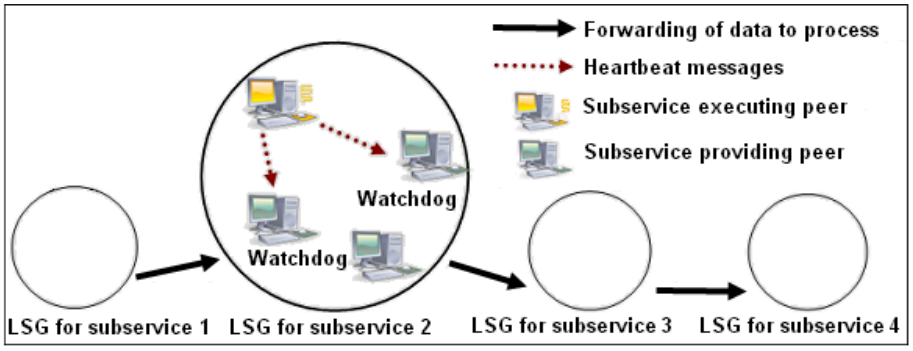

Fig. 1. Service composition based on LSGs.

The formation of a LSG is carried out as follows: On a subservice request, the requesting peer publishes a *SubtaskResource*. This resource specifies the subservice to be executed including information about the composite service it is part of. Above others, it contains input parameters of the subservice and requirements concerning peers' execution properties. Every peer that participates in the group-based execution of subservices subscribes to SubtaskResources it can perform. As soon as a peer is notified about a corresponding subservice request, it publishes a *CandidateResource*. The CandidateResource names it as a candidate for the subservices it can provide. Furthermore, it describes its current execution properties. The group formation is carried out by an initial coordinator. It collects information about all peers providing the subservice by searching appropriate CandidateResources. Afterwards, it compares these information and assigns the role of the executing peer to the best performing peer. The selection criterion is given by the requirements specified in the SubtaskResource. Likewise, it assigns the following n highest ranked peers the role of the watchdogs.

The best performing peer then takes over the subservice execution and the group coordinator role. Because this peer is being monitored, fault-tolerant service execution is guaranteed and the group coordinator is always existent. During subservice execution, the state of the subservice is periodically saved within the *ServiceStateResource*. In case the executing peer fails, the best performing watchdog continues service execution. Thereby, it obtains the current subservice state from the ServiceStateResource. Moreover, the watchdogs are completed by adding another peer. Consequently, failure adaptation takes place with no need to search for a qualified peer first. Besides by peer failure, a subservice take-over can also be initiated if a better performing peer becomes available. This is when the group coordinator comes into operation. As soon as a candidate peer logs on to the network, it is notified about the existing subservice request due to its subscription to the SubtaskResources. It publishes its CandidateResource. By subscribing to these CandidateResources, the coordinator knows if a new CandidateResource is published. It integrates the new candidate into the current peer ratings. Apart from that, other peers' execution properties can get better

or the properties of the executing peer can get worse because of dynamic load changes for example. To handle this, the peers of the group keep their execution properties in the CandidateResources up-to-date. If these resources are modified, the group coordinator is also notified. Thus it is able to recognize if an existing group member should take over subservice execution. In all three cases, the coordinator triggers appropriate role take-overs within the group if necessary.

2.2 Initial Service Composition

An environment that supports service composition has to provide two components – a design component and a runtime environment. Our research focus is the latter. We assume applications with graphical user interfaces exist that support the generation of valid specifications of composite services and their requirements and that translate them into a composite service description the runtime environment can interpret. Given such a composite service request, the runtime environment has to perform the initial service composition. That means it has to select an executing peer for each requested subservice and the selected peers have to know each other to be able to forward data between the subservices.

Regarding the LSG approach, the requesting peer first publishes the SubtaskResource for the preceding subservice in the service chain. In the example given in Section 1 the PDA publishes the one of the text-to-speech service. The subservices and their order are read out of the interpretable composite service description. Afterwards, the requesting peer carries out the formation of the first LSG. As soon as the executing peer of the new LSG is identified, it publishes the SubtaskResource of its preceding LSG, in our example for the translation, and again performs its formation. The procedure is continued until the LSG of the last subservice has been formed.

During these step-by-step LSG formations, the interpretable composite service description is recorded in every newly published SubtaskResource so that peers know which group to form next. Additionaly, every newly selected group coordinator stores the structure of its LSG within a *ServiceGroupResource* and publishes it. The identifiers of these resources are passed stepwise within the SubtaskResources, too. Thus, all executing peers are able to subscribe to the ServiceGroupResource of their successive LSG. The subscriptions facilitate the permanent addressability of a LSG, which becomes relevant during service execution. The same way, the identifier of a resource containing the state of the entire composite service is passed. If a group's executing peer changes, the new one knows if the composite service is still in the initial service composition phase – it must only send heartbeats – or if subservice execution has to be continued. As soon as all LSGs have been formed, the initial service composition is complete. Figure 2 shows the core algorithm of our initial service composition approach. ICD represents the interpretable composite service description. SR, SGR, CR and $CSSR$ stand for SubtaskResources, ServiceGroupResources, CandidateResources and the state resource of the composite service.

An important advantage of our initial service composition approach is that the load for the service group formations is distributed amongst different peers.

```
INITIAL-SERVICEGROUP-COORDINATION()
    ▷ Peer p_{t-1} forms logical service group for SR_t
 1  Determine ICD ∈ SR_{t-1}
 2  Determine SR_t ∈ ICD
    ▷ Additional information passed using SR_t:
 3  Set published SGR-Ids, CSSR-Id and the
    service requesting peer's id/port in SR_t
 4  Set ICD as attribute of SR_t
 5  Publish SR_t
 6  Search next subservice's CRs c_{t1}, .., c_{tk}
 7  Determine selection criteria ∈ SR_t
 8  Sort c_{t1}, .., c_{tk}
    ▷ Ranking is forwarded within SGR:
 9  Create SGR_t
10  Set candidates according to their ranking in SGR_t
11  Determine best performing peer's id/port ∈ CR_best
12  Inform best performing peer to continue group
    coordination passing SGR_t
```

```
SERVICEGROUP-COORDINATION(SGR_t)
    ▷ Peer p_t continues group coordination
 1  Subscribe to own logical service group's CRs
 2  Publish SGR_t
    ▷ Service providing peers know SR_t due to subscription:
 3  Determine CSSR − Id ∈ SR_t
 4  Subscribe to CSSR
 5  Determine best performing peers' ids/ports ∈ SGR_t
 6  Tell them their roles
    ▷ Subscription to SGR for addressability:
 7  if SGR_{t-1} − Id ∈ SR_t
 8     then Subscribe to SGR_{t-1}
    ▷ Form preceding logical service group if existent:
 9  Determine ICD ∈ SR_t
10  if SR_{t+1} ∈ ICD
11     then INITIAL-SERVICEGROUP-COORDINATION()
12     else  Determine id/port of service requesting peer ∈ SR_t
13           Tell it successful initial composition passing SGR_t
```

Fig. 2. Pseudocode notation of the initial service composition algorithm.

These peers are best performing peers of their service groups at least in terms of their subservice's requirements. Moreover, the initial service group formations are executed fault-tolerantly. If the requesting peer fails, the composite service is not required anymore. All further initial coordinators are executing peers of their own service group. If they fail, a member of their LSG continues their role. Hence, initial service composition is performed fault-tolerantly.

Enhancements Our approach includes further processes carried out during initial service composition: If a peer is selected for a dedicated subservice, it is taken into account that its execution properties change and that it might not be able to execute other subservices it could provide. In addition, single subservices of a composite service can be marked optional. This way, one can determine that a composite service is executed even though no peer is available that provides that subservice and fulfills its requirements. In our example, the translation might be helpful for the user but not necessary to get the RSS feeds' messages. Initial service composition is also successful if single optional subservices are not available. Finally, we described linear service chains to simplify matters. Our concept also allows nonlinear subservice arrangements through nestings in the (XML-based) interpretable composite service description and appropriate conceptual extensions. However, a detailed explanation of these mechanisms would go beyond the scope of this paper and is not needed to depict the core features decisive to realize adaptive and fault-tolerant service composition.

2.3 Composite Service Execution

Besides the initial service composition a runtime environment supporting composite services in P2P systems is responsible for the stable and fault-tolerant execution of the subservices and for the data exchange between them. To start composite service execution in our approach, the executing peer last determined informs the requesting peer about the successful initial service composition. Thereby it passes its ServiceGroupResource identifier which the requesting peer

then subscribes to. Afterwards, the requesting peer modifies the state of the composite service so that it is now declaring its execution and triggers a ring message indicating subservice instantiation. Every executing peer reads its succeeding executing peer out of the associated ServiceGroupResource to be able to forward that message. As soon as the requesting peer receives the ring message again, the composite service has been successfully instantiated.

During execution, peers may fail, their execution properties may change or better performing peers may arrive. To adapt the resulting subservice take-overs, the coordinator of every LSG saves the current structure of the group within the ServiceGroupResource, i.e. who is the executing peer and which peers are in the role of the watchdogs. On a subservice take-over, the new coordinator updates these information. As a consequence, the preceding executing peer is notified due to its subscription to that ServiceGroupResource. It reads the new processing peer out of the modified ServiceGroupResource and updates the data link. Every new executing peer first subscribes to that resource. Afterwards, it fetches the current subservice execution state using the ServiceStateResource and continues service processing. Thereby, adequate rollback mechanisms have to be established to guarantee that no intermediate results get lost and no calculations that were already saved in the current subservice state are repeated (see Section 3).

Enhancements Our approach also integrates the adaptation of failures of entire LSGs. They are monitored using detection messages along the ring of executing peers. A LSG fails as soon as the last peer that was able to perform the subservice can not fulfill the service requirements any longer or leaves the network. The preceding executing peer establishes a connection to the next but one LSG. Therefore, all ServiceGroupResources are stepwise forwarded during initial service composition. If the subservice was optional, the execution of the composite service can be continued without adverse effects. Otherwise, the preceding executing peer triggers a ring-message that indicates that the composite service has to be aborted. If no necessary group failed, the completion of a composite service is signalled by the requesting peer or by an executing peer that finalized service execution. Figure 3 shows the pseudocode notation of the algorithms applied during entire, exception-free composite service execution.

```
On-Composition-Finished-Received(SGR_last)
    ▷ Requesting peer receives "INITIAL
    SERVICE COMPOSITION COMPLETE"-message
1   Subscribe to SGR_last
2   Set CSSR state to executing
3   Modify CSSR
4   Post-Ringmessage("START PROCESSING")

On-Start-Service-Processing-Received()
    ▷ Peer p_t receives
    "START PROCESSING"-message
1   Publish SSR
2   Start service execution
3   Post-Ringmessage("START PROCESSING")

Post-Ringmessage(Message)
    ▷ Peer p_t forwards ring message
1   if SGR_{t-1} − Id ∈ SR_t
    ▷ Knows SGR_{t-1} due to subscription:
2     then Determine executing peer's id/port ∈ SGR_{t-1}
3     else Determine requesting peer's id/port ∈ SR_t
4          Send Message to (id, port)

Stop-Composite-Service-Processing()
    ▷ Peer triggers completion of the composite service
1   Set CSSR state to finished
2   Modify CSSR
3   Post-Ringmessage("STOP PROCESSING")
```

Fig. 3. Pseudocode notation of the composite service execution algorithm.

3 Implementation

To show the applicability of our concept, we have implemented the *Service Composition Framework (SCF)*. Because the project comprises 73 java classes and more than 10000 lines of code, we only describe selected elements in this section.

3.1 The Service Composition Framework

For the implementation of the SCF we utilized the *Siemens Resource Management Framework (RMF)* [9]. The RMF realizes basic features of a P2P network. Above others, it provides the two functionalities our concept relies on: a fully decentralized discovery and a publish-subscribe mechanism. Thus the SCF is based upon the RMF. The SCF itself provides an interface for developers. Every service that implements this interface can be executed as a subservice of a composite service. The interface could be kept quite simple. A subservice has to implement four methods that are used by the SCF to control its execution. One that executes the service on passing parameters, a second that stops it, a third that enforces a checkpoint of the current subservice state and a fourth that returns the state. Every result is passed to the framework using a given method. The framework then performs the result's faultless forwarding in case a successive subservice exists. That is, it repeats the forwarding if a results' receipt is not acknowledged and does not enforce a checkpoint until successful transmission. If the local peer fails and a result has not been forwarded, the related input is processed again. Of course it can not be assumed that every subservice generates elementary results but may, for example, work on continuous data streams. In cases like this, developers have to forward their results themselves. Therefore, another interface is provided. It declares a method through which the SCF passes the identifier and the port of the successive executing peer both during initial service composition and on executing peer changes.

By starting a dedicated class of the SCF, a host joins the RMF-network and subscribes to SubtaskResources it can perform. The new peer is then able to participate in composite service execution. To this end, the SCF implements the roles of watchdogs, of executing peers and of (initial) coordinators and allows to request a composite service. Also, it provides different simple comparator classes like a CPU-comparator. Requesting peers refer to them when specifying required execution properties within the (XML-based) composite service description. Coordinators use them accordingly for peer comparisons. Comparator classes can consider an arbitrary number of attributes and may prioritize them differently.

3.2 Test Environment and Example Services

To test the implemented mechanisms we developed a diversified test environment. The environment starts a variable number of peers and triggers a composite service request. During initial service composition, it checks whether all LSGs have been formed, whether the best performing peers send heartbeats

and whether the determined number of watchdogs is active. When the execution phase sets off, it tests if all groups have started execution. Additionally, the environment integrates tests that stepwise change the execution properties of the group members in such a way that a subservice take-over has to take place. Furthermore, we have implemented tests that force peer failures in all LSGs. All these further tests are executed during initial service composition as well as during execution phase. They check whether the new best performing peer resumed the role of the executing peer and sends heartbeats and whether the determined number of watchdogs is active again. In addition, they test if the new peer knows its successor and if its predecessor was notified about the change. During execution, the resume of the subservice's execution is checked as well. Finally, the environment waits for finalization of the composite service and verifies if all results have been forwarded completely and not redundantly.

For test purposes and to visualise the framework's possible fields of application, we have implemented different simple services, i.e. performing certain calculations, and the mentioned example subservices – an RSS reader, a translator and a text-to-speech subservice. All composite example applications were successfully executed using the test environment. Even if executing peers change frequently, all results are forwarded and processed correctly. The system changes the executing peers as soon as their execution characteristics degrade, better performing peers become available or peers fail. The initial service composition and the execution of composite services are performed correctly and fault-tolerantly and all subservices are always executed by the best performing peers.

4 Evaluation

Having shown the applicability of our concept, we now focus on the efficiency of the integrated mechanisms. Therefore, this section provides both an analytical evaluation of the presented service composition approach and an experimental evaluation to substantiate the formal results.

4.1 Formal Analysis

The analytical evaluation investigates the computational complexity of the proposed approach. It aims to quantify the amount of computational resources needed in relation to the problem instance specified by the number of peers, services and so on. The complexity result will subsequently be compared with the complexity of the corresponding *SpiderNet* algorithms (see Section 1).

The use of computational resources is modelled as cost. As our algorithms are not tailored towards a particular DHT implementation, but only assume a DHT with publish-subscribe functionality, we base the cost model on primitive DHT operations. This way, the analytical results are independent from the characteristics of a particular DHT implementation. We distinguish three cost types for DHT operations: $c_{publish}$ is the cost required for making a piece of information available in the DHT for retrieval; c_{search} is the cost incurred when

discovering and retrieving a dedicated piece of information; c_{send} is the cost for directly sending a message to a particular peer. We will quantify the DHT operation calls performed in our algorithms as well as those executed in *SpiderNet* to determine cost functions. To simplify these functions, we make the following two assumptions: The number of candidate peers is the same for each subservice and the number of watchdogs is the same for each LSG. So we will use three different variables: n denotes the number of subservices in the composite service, k is the number of candidate peers for each subservice (which is equivalent to the size of the LSG), and w denotes the number of watchdogs for each LSG.

Initial Service Composition With respect to the previously introduced cost model and variables, the initial service composition algorithm (see Section 2.2) has the following cost function:

$$T_{initial}(n,k,w) = (kn + 5n + 1)c_{publish} + (n + wn + kn + 1)c_{send} + nc_{search}.$$

One can argue that choosing two watchdogs for each LSG ($w = 2$) results in sufficiently low probability for overall failure. Then $T_{initial}(n,k,\ w)$ can be written as $T_{initial}(n,k) = O(kn)$. Thus, the complexity of the initial service composition algorithm is linear in the total number of candidates for the composite service.

Composite Service Execution We distinguish between the exception-free execution and monitoring of the composite service and the handling of different exceptions. The exact cost function for the execution and monitoring algorithm (see Section 2.3) has to include the total duration D of the composite service execution. This is because it influences the number of ring messages and heartbeat messages (interval I_{ring} and interval $I_{heartbeat}$) used for failure detection:

$$T_{exec}(n,w) = (n+2)c_{publish}+(D/I_{ring}(n+1)+D/I_{heartbeat}wn+wn+2n+2)c_{send}.$$

The intervals for heartbeat and ring messages are QoS parameters to be chosen, for example, according to real-time requirements of a composite service. With neglection of D, I_{ring} and $I_{heartbeat}$ and with $w = 2$ the cost function can be written as $T_{exec}(n) = O(n)$. The complexity of composite service execution and monitoring is linear in the number of subservices of the composite service.

Handling exceptions during execution triggers different further steps (see Section 2.3). When execution properties of a peer change or a new peer becomes available, adaptation incurs the following cost:

$$T_{adapt}(w) = 5c_{publish} + (2w + 2)c_{send} + c_{search}.$$

If we again assume $w = 2$, this adaptation requires only constant cost: $T_{adapt} = O(1)$. Handling the failure of a service executing peer has the following cost, which are again constant:

$$T_{failure} = 4c_{publish} + c_{send} + c_{search}, \quad T_{failure} = O(1).$$

Comparison to SpiderNet The previously obtained complexity results are now compared to the complexities of the corresponding *SpiderNet* algorithms [4]. We chose *SpiderNet* because it is the only existing approach realizing fault-tolerant service composition in P2P systems.

It can be shown that *SpiderNet*, which is also DHT-based, produces cost $T^S_{initial}(n,k) = O(k^2(n-1))$ during initial service composition. This is in contrast to $T_{initial}(n,k) = O(kn)$ in our approach. For example, assume a scenario where the composite service is composed of 5 subservices and there are 10 candidate peers for each subservice. Then *SpiderNet* is in the order of 400 calls to primitive DHT operations, while our approach only requires 50 calls.

Analyzing the *SpiderNet* algorithms for service execution yields the cost function $T^S_{execution}(n) = O(n)$. This is in the same order as our approach ($T_{execution}(n) = O(n)$). However, adaptation to failures requires more resources than our approach. This is because *SpiderNet* does not use watchdogs, but monitors multiple backup paths. As a result, exception handling requires to change not a single subservice but the whole service path and more monitoring messages are required in order to achieve the same level of fault-tolerance.

In conclusion, analyzing the complexities of the proposed algorithms showed that the introduction of LSGs results in significant reduction of complexity during initial service composition. Furthermore, the use of watchdogs for detecting failures results in higher fault-tolerance without increasing complexity. These are additional achievements to the newly introduced ability to adapt performance variations and peer arrivals.

4.2 Empirical Analysis

In this section we describe the results of our empirical analysis and evaluate if the results confirm our formal findings. To this end, we have carried out over 8700 measurements of our approach's core procedures with the aid of the SCF.

Initial Service Composition In Section 4.1 we concluded that the complexity of the initial service composition algorithm is linear in the total number of candidates for the composite service. To verify this, we have measured the duration of initial service composition for two, six and ten candidates per LSG with up to twelve subservices. We raised the number of subservices stepwise recording 100 measurements each time which results in 3600 measurements. Figure 4 (a) charts initial service composition duration for a varying total number of candidates of the composite service (kn). It can be seen that it makes a difference if the total number of 20 candidates arises from 2 subservices with 10 candidates each or from 10 subservices with 2 candidates which is due to the stepwise group formations. Thus the graphs grow linear, which verifies our formal findings.

Composite Service Execution The computational complexity of exception-free composite service execution depends on the duration of the composite service. Hence, we focus on the complexity of mechanisms realizing adaptive and fault-tolerant execution here.

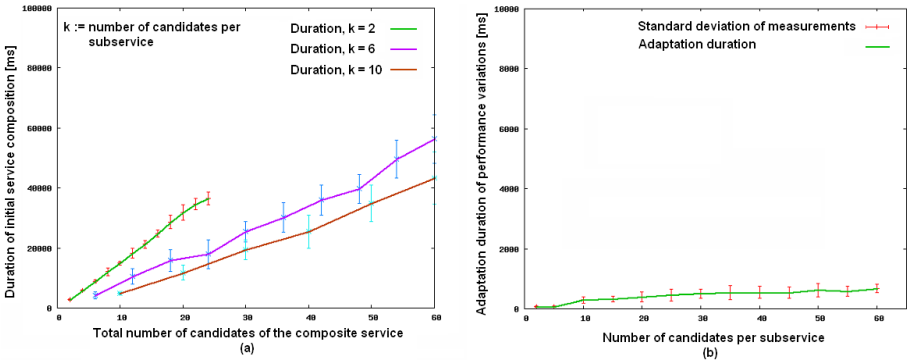

Fig. 4. Experimental results for initial service composition (a) and adaptation of changes of execution properties (b).

We have formally shown that adaptation requires only constant cost when peers' execution properties change or new peers become available. Figure 4 (b) illustrates the duration measurements for this adaptation based on 1200 measurements taken for LSGs with up to 60 members. During formal evaluation, we neglected the costs for integrating a candidate into the current peer ranking because they depend on the subservice's selection criteria. Regarding the experimental results one can see that duration for adaptation behaves almost constant. The slight raise is caused by the growing number of peers to be compared with the candidate. Thus, our formal results have been confirmed.

Concerning the handling of failures of service executing peers, we arrived at the conclusion that this again requires constant cost. Figure 5 charts our results retained from 3900 measurements of failure adaptation duration using varying heartbeat intervals. Figure 5 (a) shows that the number of watchdogs does not influence the adaptation duration. In figure 5 (b) we contrast the duration for failure adaptation with the current heartbeat threshold. This threshold is directly derived from the heartbeat interval. It represents the time the watchdogs shall wait for the next heartbeat of the executing peer, whereby a little time interval is added to compensate delays in message passing. It becomes apparent that average adaptation duration is partially even below the current threshold. In this context one has to be aware that an executing peer can fail shortly before sending its next heartbeat. Hence, the next heartbeat is overdue soon. Nevertheless, one can state that the average adaptation duration is even below the current heartbeat threshold constant which is in agreement with our formal results, too.

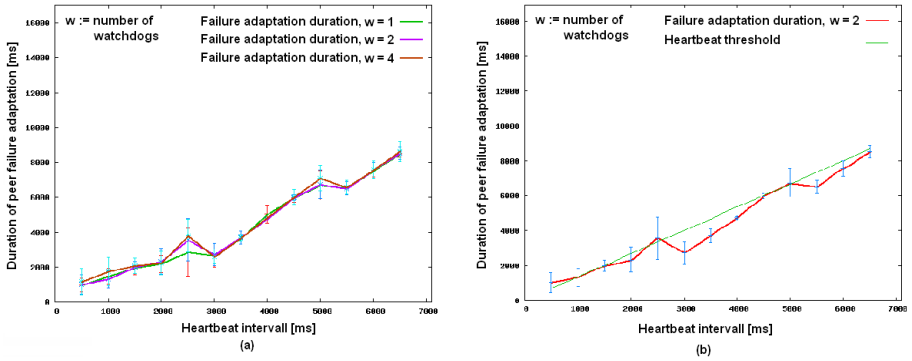

Fig. 5. Experimental results concerning peer failure adaptation.

5 Conclusion

We have presented an approach for adaptive and fault-tolerant dynamic service composition in P2P systems. Composite services are provided by interacting and self-organizing peer groups. Within these groups, watchdog peers monitor subservice executing peers and coordinators detect peer arrivals and variations of peers' execution properties. If necessary, they cause the migration of subservices to other peers. The underlying algorithms are fully decentralized, i.e. they are only based on local peer decisions. The concept has been implemented and formally and empirically evaluated. Amongst others, we were able to show that the introduction of service groups results in significant reduction of computational complexity. The use of watchdogs for detecting failures results in higher fault-tolerance without increasing complexity. Moreover, all data are transferred and processed correctly during composite service execution even if single subservices are often migrated. Our approach is the first one that enables the adaptation of peer arrivals and changes of their execution properties.

A future field of interest is the consideration of deviation from service requirements if no peer is able to fulfill them. Another topic is the integration of a further selection level. One can think of the composition of services that integrate advertisement into content like audio or video and services that provide that kind of media content. Thus, users might pay less for the content. Now it can be left open which advertisement service to integrate. The decision is made during runtime and can depend on information about the user's end device or the user's profile. Users provide these information due to cheaper content. This way, advertisers can be offered a dedicated target group. Because service composition is P2P-based, advertisers can furthermore be arbitrary users that publish the existence of the advertisement service – for example a student giving classes in math to pupils living in his neighbourhood. Our concept therefore has to integrate a further selection level: before selecting a peer for service execution it must be determined which service of a specified kind, i.e. which kind of advertisement service, to integrate.

References

1. Singh, M., Huhns, M.: Service-Oriented Computing. John Wiley & Sons Inc. (2005)
2. Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A Component System for Pervasive Computing. In: Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 04). IEEE Press, Orlando, USA (2004)
3. Gu, X., Nahrstedt, K.: A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids. In: Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC 2002). IEEE Press, Edinburgh, Scotland (2002)
4. Gu, X., Nahrstedt, K., Yu, B.: SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. Technical report, Department of Computer Science. University of Illinois at Urbana-Champaign (2003)
5. Gu, X.: A Quality-Aware Service Composition Middleware. PhD thesis, Department of Computer Science. University of Illinois at Urbana-Champaign (2004)
6. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In: Rachid Guerraoui (Ed.): Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
7. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Liben-Nowell, D., Dabek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. IEEE/ACM Trans. Netw. 11, 17–32 (2003)
8. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A Scalable Content-Addressable Network. In: Applications, Technologies, Architectures,and Protocols for Computer Communication. ACM Press, San Diego, California, United States (2001)
9. Rusitschka, S., Southall, A.: The Resource Management Framework: A System for Managing Metadata in Decentralized Networks Using Peer-to-Peer Technology. In: Gianluca Moro, Manolis Koubarakis (Eds.): Agents and Peer-to-Peer Computing, First International Workshop. LNCS, vol. 2530, pp. 144–149. Springer, Heidelberg (2003)
10. Zhao, B., Kubiatowicz, J., Joseph, A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, Computer Science Division, U. C. Berkeley (2001)
11. Clarke, I., Sandberg, O., Wiley, B., Hong, T. W.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. In: Hannes Federrath (Ed.): Designing Privacy Enhancing Technologies. LNCS, vol. 2009, pp. 46–66. Springer, Heidelberg (2001)
12. Terpstra, W., Behnel, S., Fiege, L., Zeidler, A., Buchmann, A.P.: A peer-to-peer approach to content-based publish/subscribe. In: Proceedings of the 2nd international workshop on Distributed event-based systems. ACM Press, San Diego, California, United States (2003)
13. Castro, M., Druschel, P., Kermarrec, A.-M., Rowstron, A.: Scribe: A large-scale and decentralized publish-subscribe infrastructure. In: Jon Crowcroft, Markus Hofmann (Eds.): Networked Group Communication, Third International COST264 Workshop. LNCS, vol. 2233, pp. 30–43. Springer, Heidelberg (2001)