Modeling Movable Components for Disruption Tolerant Mobile Service Execution

Rene Gabner[1], Karin Anna Hummel[2], and Hans-Peter Schwefel[1,3]

[1] Forschungszentrum Telekommunikation Wien, A-1220 Vienna, Austria
{gabner,schwefel}@ftw.at
[2] University of Vienna, A-1080 Vienna, Austria
karin.hummel@univie.ac.at
[3] Aalborg University, DK-9220 Aalborg, Denmark

Abstract. Software as a Service relies on ubiquitous network access which cannot be assured in mobile scenarios, where varying link quality and user movement impair the always connected property. We approach this challenge by utilizing movable service components between a remote cluster, cloud, or server and the client device using the service. To overcome connection disruptions, service components are moved to the client prior to connection loss and executed locally. Although the basic concept is a brute force approach, challenges arise due to best fitting service decomposition, accurate estimation of connection losses, and best trade-off between moving service components and the overhead caused by this proactive fault tolerance mechanism.
This paper contributes to the general approach by presenting a system architecture based on an extended client/server model which allows to move components. Additionally, an analytical model is introduced for analyzing where to place service components best and extended to investigate failure rates and average execution time in different system configurations, i.e., different placement of service components either on the server cloud or client side. The models presented are based on Markov chains and allow to analytically evaluate the proposed system. Applied to a specific use case, we demonstrate and discuss the positive impact of placing components temporarily at the client in terms of failure rate and mean service execution time.

Key words: Mobile Computing, Software as a Service, Service Decomposition, Markov Model, Disruption Tolerance

1 Introduction

Software as a Service (SaaS) [1] is a field in particular of interest for mobile computing scenarios, like support for mobile workers or mobile business in general. Instead of pre-installed software packages, software is hosted and maintained at a service provider and can be accessed by the user. In this vision, the burden of troublesome installing, updating, and maintaining is taken from the user. In mobile contexts, it is even more beneficial to access the software as a service to

fulfill tasks without having pre-installed too many applications. Computing *cloud infrastructures* are enabling system architectures for supporting the envisioned SaaS solution.

In contrast to stationary scenarios, mobile networked systems are impaired by varying link conditions due to fading effects and environmental disturbances on the wireless medium, other devices competing for access to the wireless link, and moving in and out of the range of a wireless network. As a consequence, intermittent connectivity is likely to happen and has to be addressed to make *mobile SaaS* feasible.

Our approach addresses intermittent connectivity by considering different locations for service execution, i.e., at the (remote) server cloud or the mobile client. In case of stable connectivity, service parts may remain at the server and classical client/server communication will be efficient to assure fastest service execution. In situations of weak connectivity and frequent disconnections, service parts have to be moved to the client to remain operational which will lead to increased service execution times at the low performance mobile device. We see four major challenges of the approach: First, the best fitting granularity of service decomposition and dependencies between service components have to be found. Second, detecting best time periods for placing service components have to be detected, e.g., predicting disconnects in advance. Third, determining optimized allocations of service components for a certain predicted network behavior. Fourth, moving software service parts causes overhead and the trade-off between availability and networking overhead has to be considered.

In this paper, we approach the third research question, as it is a motivating prerequisite for the other challenges, by modeling a service as a composition of parts, i.e., service components, and analyzing how the allocation of these components to client or server side influences certain performance or reliability metrics. Successful service execution means that the components can be accessed and used. Intermittent connectivity now leads either to completely failed services or delayed service execution. We consider both cases and present *(i)* an *analytical model for service failure/success* evaluating the failure rate of services and *(ii)* an *analytical model for service execution time analysis* for different component placement configurations. Hereby, our fault model consists of network disconnection failures only.

The paper is structured as follows: After presenting a survey on related concepts for disconnected service operation in Section 2, we describe the system architecture for movable service components in Section 3. In Section 4 we introduce the analytical model based on Markov chains. Service invocations are modeled as transitions which may succeed or fail due to network failures. In Section 5, we introduce the editor use case and present results for this particular service to demonstrate the potential of both the general concept of meaningful placement of service components for tolerating disconnections and the insights gained by using the analytical models introduced. Section 6 summarizes the work and presents an outlook on future work planned.

2 Related Work

Allowing services to be allocated and executed at different distributed locations was a hot topic in the past years. Fuggetta et al. [4] address the increased size and performance of networks as a motivator for mobile code technologies. Different mobility mechanisms like migration, remote cloning, code shipping, and code fetching are utilized to meet a diversity of requirements. We conceive temporary proactive code migration to support our architecture best. However the main focus is the analysis of impacts of code migration and optimization of component location to achieve best service execution with a minimum of interruption and delay.

When mobile communications became popular, the research area expanded and mobile computing introduced challenges different from traditional distributed computing. These challenges are related to mobile data management, seamless mobile computing, and adaptations due to limited mobile device capabilities. Imielinski et al. [5] describe the implications and challenges of mobile computing from a data management perspective. Important aspects are *(i) management of location dependent data*, *(ii) disconnections*, *(iii) adaptations of distributed algorithms for mobile hosts*, *(iv) broadcasting over a wireless network*, and *(v) energy efficient data access*. While mobile networks grew rapidly, a diversity of different mobile devices were pushed to the market, running different operating systems and execution environments. Because of many different mobile platforms, service development becomes complex and costly, as each platform needs its own implementation of a service.

The SaaS approach can help to overcome multi implementations of services. Instead it is possible to run a service on an execution platform within the network. Every mobile client with access to the network's application server can use such services. Our architecture benefits from the SaaS approach as it overcomes complicated installations on the client and keeps the solution flexible to reconfiguration and component migration at runtime. To execute such SaaS services which support movable components, special execution environments at the client are required. One possible solution is presented by Chou and Li [2]. They adapted an Android based mobile platform for distributed services, and show one way to execute SOA based applications. This architecture supports also access to services deployed in a SaaS environment. Because such SaaS models depend on reliable network connectivity, disruption tolerant networks are also of particular importance for mobile scenarios.

There are various researchers investigating in disruption tolerance. For example, Chuah et al. [3] investigate network coding schemes for disruption tolerant mobile networks. They compare the performance of different schemes and message expiration times to enhance network connections between mobile nodes suffering from intermittent connectivity. Another approach introduced by Ott and Xiaojun [9] is based on the application layer and introduces end-to-end disconnection detection and recovery schemes for mobile wireless communication services. Such end-to-end solutions take advantage of the fact, that the observation of the network is not based on information from the underlying transport

and physical layers, which are not available in all cases. The network prediction function proposed by our architecture could benefit from such end-to-end network state detection solutions.

An approach to deal with interrupted connections is discussed by Su et al. [10]. They propose an architecture for seamless networking utilizing specialized application proxies at the client. Those proxies are tuned to serve a special service like SMTP. In our proposed execution environment, proxies will only be used to support the migration of service components.

3 System Description

We propose an architecture which supports mobile, wireless service execution on thin-clients, based on the Software as a Service (SaaS) paradigm [1]. One major constraint of SaaS is the availability of a stable, *always-on* network connection to the host running the service. Applied in a mobile context, intermittent connectivity caused by disrupted transmissions at the air interface is a major challenge. To overcome this issue we propose to split the service into several service parts (service components) applying service decomposition techniques. Selected service components are moved proactively from the service execution platform to the thin-client in case of estimated bad network quality. The service execution platform is expected to run on a server cloud, in this paper also simply referred to as *server*.

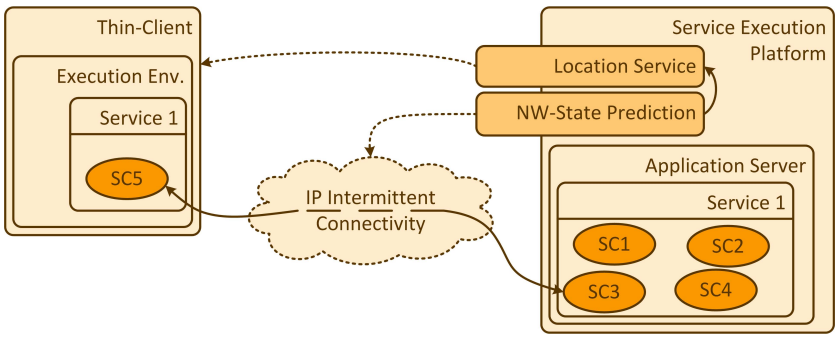

Fig. 1. Overall system architecture.

Figure 1 shows a service *Service 1* which has been decomposed into five *service components* ($SC1$, $SC2$, $SC3$, $SC4$, and $SC5$). Each component is responsible for a well defined task. After it has finished, the execution flow is passed to another service component. This concept is sometimes termed *component chaining model*. The subsequently executed component may however depend on the result of the previous computation, which is modeled probabilistically for the component chaining description in Section 4.1.

The *Network State Prediction* (NSP) function collects and holds information about the current state of the network connection between the server and the thin-client. Additionally, it interfaces a couple of different data sources to predict the network state condition. For instance, the observation of the network state over a longer time period combined with additional geo-location information can be evaluated in this component. The location data can be requested directly from the thin-client if a GPS receiver is available or, otherwise, from a mobile operator. Of course there are other possible data sources which can be integrated by expanding the interface of the NSP. In case we expect network connection degradation, the NSP triggers the application server to move components which are essential for the execution within the next time periods to the client. If the service components have been moved successfully to the client, it is possible to continue service execution even if the connection is lost. In case a component is unreachable caused by a suddenly broken network link there are two possibilities to handle the situation. As described later in section 4.2 the execution fails in case of an unreachable service component. The other approach modeled in section 4.3 has an additional *network down* state to delay the whole service execution. After reconnecting to the service execution platform, the application server might decide to fetch back any of the service components to take over execution again.

In order to support the decision which components should be migrated in a specific network environment, the remainder of the paper focuses on component placement and analyze the impact of different *static* component placement configurations for an example service.

4 Service Component Model

In Section 3 we discussed the system architecture including the view of a service being decomposed into components some of which can be migrated between client and server. In order to make substantiated choices on which configuration to apply in a given setting, this section comes up with different Markov models that allow to analyze the consequence of a certain static placement of service components on client and server side.

4.1 Markov Model for Service Component Flow

An application consists of service components which may reside on the cloud (here referred to as a single application server) or on the (thin) client. The sequence of service components that is invoked in the course of a service execution is modeled as deterministic Markov chain. The service components are thereby assumed to be completely autonomous and are executed sequentially; as a consequence the only interaction between service components occurs when passing the execution flow from component i to component j, where $i, j = 1, ...N$. The transition probabilities between states in the Markov chain model (which correspond to service components) depend on the service type, usage patterns, and

input objects. Those transition probabilities are collected in the stochastic matrix $\mathbf{P}$.[1] The Markov chain model contains exactly one absorbing state, whose meaning is a successful service completion. Without loss of generality, we order the states in this paper in a way that state N is always the absorbing success state. The initial state, i.e., first service component called, can be probabilistically described by an 'entrance vector' $\mathbf{p_0}$. The examples discussed later in this paper always assume state 1 as the single entrance state, hence $\mathbf{p_0} = [1, 0, ..., 0]$.

As the application model described by the transition probability matrix $\mathbf{P}$ (and the entrance vector $\mathbf{p_0}$) only describes the probabilistic sequence of component executions, it has to be slightly modified to allow for notions of execution time. Namely mean state-holding times $T_1, T_2, ...T_{N-1}$ for the $N-1$ states (the absorbing success state, here assumed state N, does not require an associated state-holding time) need to be defined which then allow to transform the discrete model into a continuous time Markov chain where the generator matrix $\mathbf{Q}$ is just obtained via correct adjustment of the main-diagonal of the matrix

$$\mathbf{Q}^* = \mathrm{diag}\,(1/T_1, ..., 1/T_{N-1}, 0) \cdot \mathbf{P},$$

such that the row-sums of $\mathbf{Q}$ are all equal to zero.

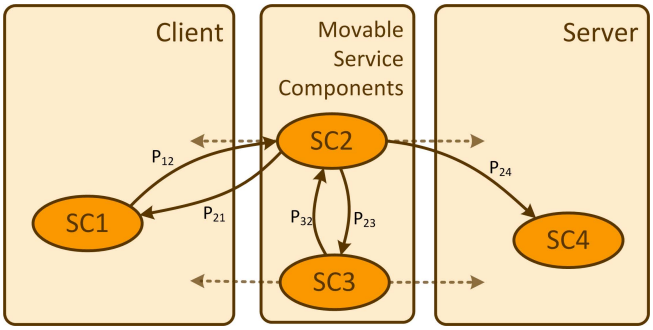

Fig. 2. Decomposed service with movable components.

Some of the service components cannot be freely migrated between server and client side. Typical examples include user-interface components that naturally have to reside on the client, or service completion states that require centralized storage of the result in the application server, hence are fixed to reside on server side. See Section 5.1 for an example. Other service components can be migrated between client and server side, as illustrated in Figure 2. The vector $\mathbf{c} \in [0, 1]^N$ represents a specific placement of components on client and server side; here we use $c(i) = 0$ for a client-side placement of component i. If the service execution flow passes from a component i to another component j, this transition requires network communication, if and only if these two components are located on different physical entities, i.e., $c(i) \neq c(j)$.

[1] Note, that we use bold fonts for matrices and vectors to improve readability.

The goal of this section is to come up with quantitative models that allow to calculate application reliability and performance for specific static configurations $\mathbf{c}$; the process of how such configurations are created, e.g., the download of the component to the client, is not considered. These models are developed in the following subsections.

4.2 Service Success/Failure Model

In the first scenario, we describe a modification of the discrete time Markov chain $\mathbf{P}$ such that the modified model $\mathbf{P}'(\mathbf{c})$ allows to compute the probability that the application is successfully completed given a certain component placement described by $\mathbf{c}$. As we consider the modified model for a specific given configuration, we drop the dependence on $\mathbf{c}$ in the following for notational convenience. The properties of the communication network are assumed to be described by a simple Bernoulli process, i.e., whenever network communication is needed upon transitions of the execution flow to a component placed on the different physical entity, the network is operational with probability $1 - p_f$ and the transition to the new service component succeeds. If network communication is not successful, the new service component cannot be executed and service execution fails.

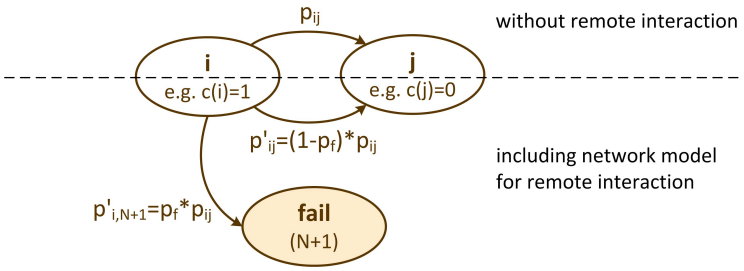

Fig. 3. Extended service component model including network failure.

The modifications of the Markov chain to capture such behavior in the extended model $\mathbf{P}'$ are illustrated in Figure 3. The matrix $\mathbf{P}'$ contains one more state, state number $N+1$, which resembles an absorbing service failure state. Every transition $i \rightarrow j$, where $i, j = 1, ..., N$ between service components placed on different entities is partially forked off to the fail state with probability p_f. The probability of a service failure can be computed as the probability of reaching the absorbing fail state, i.e.,

$$Pr(\text{service failure}) = \left(\lim_{k \to \infty} \mathbf{p_0} \cdot \mathbf{P'}^k \right) \mathbf{e'_{N+1}},$$

where $\mathbf{e'_{N+1}}$ is a column vector with all components set to 0 except component $N + 1$ which is set to 1. The service failure probability can hence be computed numerically, see Section 5.2 for examples.

4.3 Execution Time Model

The Markov model in the previous section allows to calculate service success probabilities defined by the probability that the network communication is available for remote component interactions in a probabilistically chosen execution sequence of service components. If the network is not available (which occurs according to a Bernoulli experiment with probability p_f when the execution flow is migrated to a remotely placed component), the service execution is stopped and considered failed. There are however cases of elastic or delay-tolerant services in which a temporarily unavailable network connection just creates additional delay. Another variant is that the network connectivity is not completely unavailable but rather in a degraded state which leads to longer communication delays. In the following, we describe a Markov model transformation which allows to analyze the impact of such additional network disruption delay on the distribution of the service execution time for different placements of the components.

We use the continuous time version of the service model, i.e., a Continuous Time Markov Chain (CTMC), described by the generator matrix $\mathbf{Q}$, see Section 4.1. The service execution time without considering component placement and network interaction is then the phase-type distribution [7, 8] described by the first $N-1$ states.

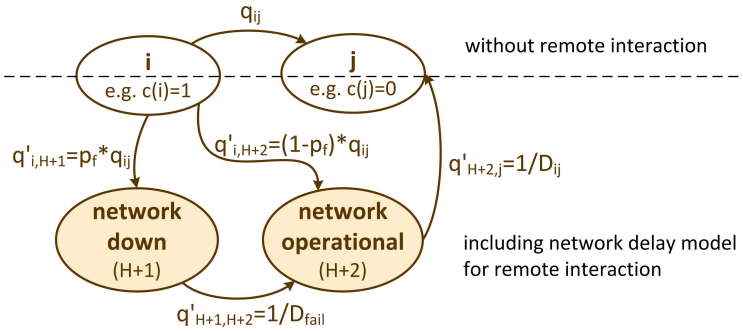

Fig. 4. Extended service component model including network failure and execution time.

The following model of the execution time behavior for the client-server configuration $\mathbf{c}$ of the service components is employed: First all software components that are executed on the client side are assumed to execute more slowly by a factor of k_{client}. This is reflected by scaling all corresponding rows of $\mathbf{Q}$ by a factor of $1/k_{client}$. For the communication behavior, the following two input parameters are required in addition to the network failure probability p_f: *(i)* A matrix $\mathbf{D}$, whose elements $D_{i,j}$ specify the mean communication delay for the activation of component j from the remote component i. *(ii)* The mean time until network recovery D_{fail}. The generator matrix of the CTMC for the distributed client-server implementation under such assumptions on the remote communication

delays is then obtained by adding two additional delay states for each transition $i \to j$ with $Q_{i,j} \neq 0$ and $c(i) \neq c(j)$. Let's assume these two additional delay states obtain labels $H+1$ and $H+2$, then the following modified transition rates are employed in the extended matrix $\mathbf{Q}'$ (illustrated in Figure 4):

$$Q'(i,j) = 0; \quad Q'(i,H+1) = p_f Q(i,j); \quad Q'(i,H+2) = (1-p_f)Q(i,j)$$

$$Q'(H+1,H+2) = 1/D_{fail}, \quad Q'(H+2,j) = 1/D_{i,j}.$$

The diagonal elements of $\mathbf{Q}'$ need to be adjusted accordingly. If component i and j are placed on the same entity ($c(i) = c(j)$), then $Q'(i,j) = Q(i,j)$. Note that using a matrix for the remote communication delays allows to distinguish between components that may have different sizes of parameters/data associated with their remote call. For the numerical examples in Section 5.3, we however employ $D_{i,j} = 1$ for all i, j.

The extended generator matrix $\mathbf{Q}'$ then contains the phase-type distribution (time until reaching state N, which is assumed to be the service success state), for which the standard matrix calculations for moments, tail probabilities, or density values can be applied, see [7, 8]. Numerical results are presented in Section 5.3.

Note, that many variants of the Execution Time Model can be defined: For instance, the current approach in Figure 4 assumes that the network is operational with probability $1-p_f$ and in that case the remote component call can be successfully finalized. One could of course also consider the case that the network connection can fail during the remote component call, which would correspond to a transition from state $H+2$ to state $H+1$ in the figure. Similarly, more general network down times than exponential can be represented by replacing state $H+1$ by a phase-type box of states.

5 Numerical Results

In the following we present numerical results to illustrate the service failure and execution time models for the example of a text editor service.

5.1 Text Editor Example Service

The editor example described below is used in Sections 5.2 and 5.3 to exemplify results of the introduced Markov models. Figure 5 shows the discrete time Markov model of the editor, including the values of the transition probabilities.

The transition probabilities are chosen so that they approximately resemble average user behavior: Component 2, the *Editing Framework*, is used most frequently as it processes the input of the user. Any key press or menu bar activity is communicated from the *UI* to the *Editing Framework*. Thus, the transitions between *UI* and *Editing Framework* component are most frequently

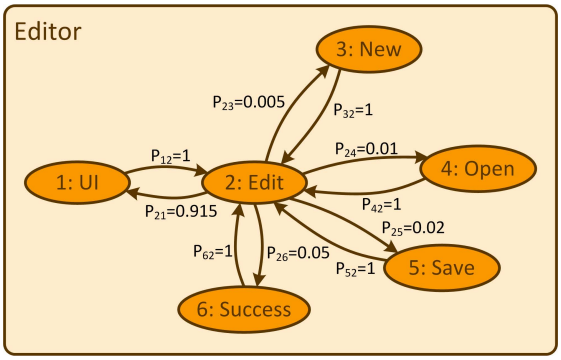

Fig. 5. Example use case text editor service.

taken. Creating, opening, or saving a document (components 3 to 5) are less likely operations compared to keystrokes. Components 1 (user interface) and 6 (service success) are special with respect to placement in the client/server architecture. The user interface needs to be executed on the client, and the final success operation is assumed to include storage of the document in the server cloud, hence must be located at the server. This fixes two of the components in the configuration vector **c**.

	1: UI	2: Edit	3: New	4: Open	5: Save	6: Success
config 1	client	server	server	server	server	server
config 2	client	server	server	server	client	server
config 3	client	client	server	server	server	server
config 4	client	client	client	client	client	server

Fig. 6. Editor example configurations.

We consider four different static configurations to analyze the execution characteristics of the editor as summarized in Figure 6. For configuration 1, everything expect the UI is located at the server. This is a pure SaaS configuration. For configurations 2 and 3, exactly one component in addition to the UI is placed on the client (note, that the selected components are used with different frequencies). Configuration 4 is placing all movable components on the client, hence, this configuration puts the highest resource requirements to the client.

5.2 Numerical Results for Service Success Probability

The editor example service is now used to exemplify the Markov model capabilities and to show the type of analysis and conclusions that can be obtained from the service success model in Section 4.2. Figure 7 shows the calculated

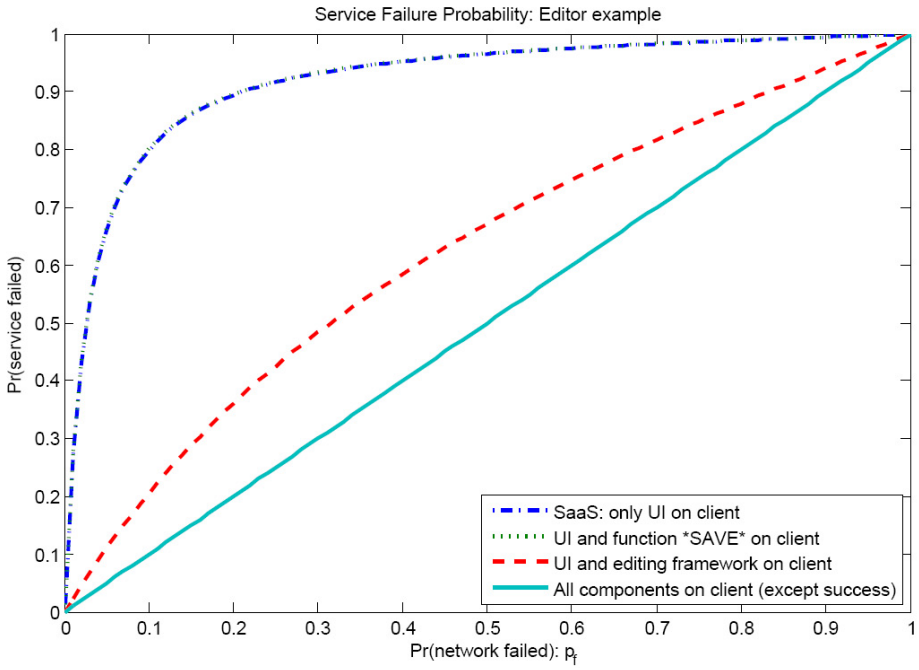

Fig. 7. Service failure probability of the editor service in four different component placement configurations.

service failure probabilities for the four different placement configurations of service components (Figure 6). The probability of network failure upon remote component interaction, p_f, is varied along the x-axis. The best possible scenario results when all editor components are placed on the client (solid line), so that only a single network interaction is necessary, namely the one connected to the transition to the success state (at which the edited file is stored at the server). As there is exactly one network interaction necessary in this case, the service failure probability is equal to p_f in this case.

At the other extreme, the full SaaS configuration in which only the user-interface is placed on the client (dashed-dotted line), frequent network interactions are necessary in particular for transitions between the *UI* and *Editing Framework* component leading to a rapid increase of service failure probability already for very small parameter ranges of p_f. Hence, the SaaS approach is in this example only useful for scenarios of good network connectivity (p_f well below 5%). Moving the service component *Save* to the client actually increases the service failure probability slightly due to the necessary interactions between editing (remaining on server) and saving (moved to the client), however hardly visible in Figure 7. Placing the *Editing Framework* instead on the client leads to a dramatic improvement: For instance, a service failure probability below 40% can be achieved also for network failure probabilities up to more than 20%.

Due to the simple structure of the editor example, the qualitative superiority of the configuration placing *UI* and *Editing Framework* both on the client is intuitively clear. However, the Markov model can be used to substantiate such choices with quantitative results and it can be argued whether moving a component might even worsen the failure rate. In particular for more complex service component interactions the Markov model can be used to make optimized choices about which component to place on client-side.

Note, that the four curves in Figure 7 never cross. Hence, when purely optimizing placement choices based on minimizing service failure probability, the network quality (expressed by p_f) does not influence the 'ranking' of the different placements.

5.3 Numerical Results for Execution Time Analysis

In the following we present numerical results to illustrate the application of the execution time CTMC from Section 4.3. The results use the same modular text editor service as previously for the service failure probability analysis. The mean state-holding time for the different states (assuming execution on the server) are:

$$T_{UI} = 1,\ T_{edit} = 0.1,\ T_{new} = T_{open} = T_{save} = 1.$$

Due to the possibility of rescaling time, we use configurable *units of time* in the investigations below; for illustration, seconds can be assumed.

The execution of the service component on the client is assumed to take $k_{client} = 10$ times as long as the execution on the server. The remote call of another module is for all module pairs the same, $D_{ij} = 1$. If the network connection is down (with probability p_f), the mean time to recovery is exponentially distributed with mean $D_{fail} = 20$. Figure 8 shows the mean application execution times for the same four component placement configurations as in the previous section. The full SaaS approach leaving all components on the server (dashed-dotted line) requires frequent network interactions, which degrades application execution time dramatically already for rather small probabilities p_f. When moving the *Save* component to the client, the execution time even increases showing that this configuration is not beneficial. Installing all components on the client (solid line) minimizes the impact of the network quality (as expressed by p_f). However, for parameter ranges of p_f smaller than approx. 15% in the calculated example, the solution of having both the *UI* and the *Editing Framework* executed locally on the client performs best. The latter is a consequence of the slow-down factor k_{client} of the processing at the client.

In summary, the calculation model can here be used to dynamically optimize the execution times via changes of the component placement depending on network quality. Note that the execution times grow linearly with p_f; as the network functionality does not change the execution flow through the modules (only its timing), the number of remote component invocations stays the same, hence p_f linearly scales into mean service execution times.

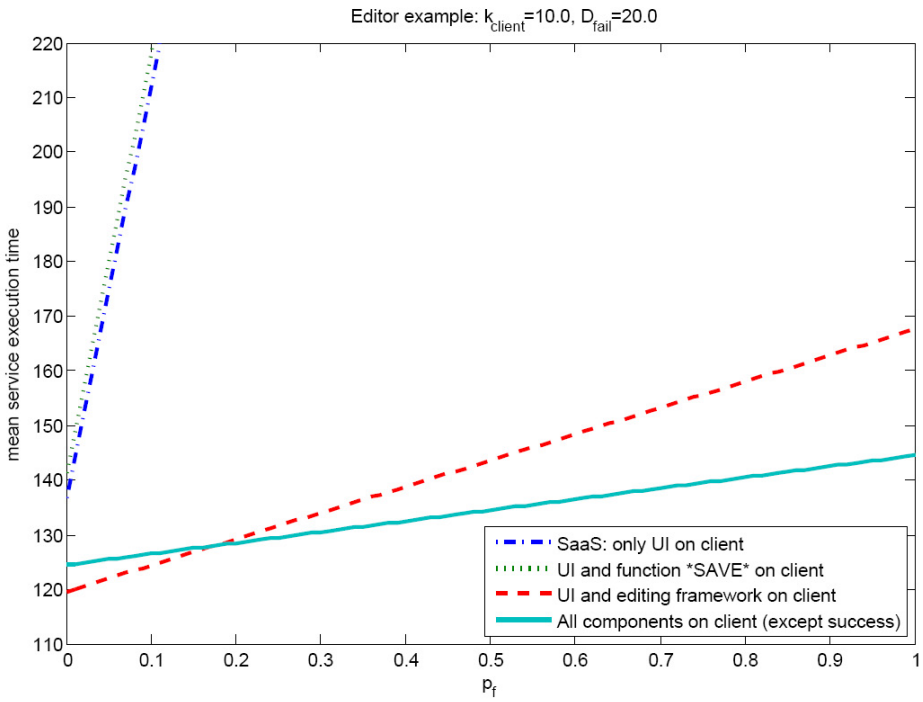

Fig. 8. Mean service execution time [units of time] of the editor example in four different configurations.

The representation of the execution time as phase-type distribution also allows to calculate numerically the density, tail probabilities, and higher moments of the execution time distribution. For the example configurations, we calculated the coefficient of variation (variance normalized by the square of the mean) of the execution time distribution for all configurations. The results showed that placing all components on the server not only dramatically increases the mean time, but also shows a higher variability in the application execution time. (The variance can be a useful input for an M/G/1 queuing type of analysis, as then the mean queue-length and system time only depend on the first two moments of the service time, e.g., P-K formula [6]).

6 Conclusions

In this paper, an architecture and modeling approach for movable service components has been presented targeting the Software as a Service paradigm. Moving service components from a server cloud to the mobile clients allows to tolerate disconnection periods, which are likely to occur in mobile scenarios. First, we described the concept of moving crucial service components from the server cloud to the client. Second, we presented analytical models to investigate the

potentials of proactive placement of components. The models are generic for disruption tolerant computing based on movable components and allows to give insights for various, even complex services.

The usefulness of the analytical models has been demonstrated for a sample editor use case service, consisting of network intensive and non-network intensive components. In this use case and realistic parameter settings, evaluation results in terms of failure rate and mean service execution time showed indeed the potential benefits of moving service components to the client in case of expected frequent networking failures. These results are encouraging for extending the approach in future work both in terms of proposing means for triggering proactive service component migration and investigating the trade-off between messaging overhead and decreased service failure rate.

Acknowledgments This work has been supported by the Austrian Government and by the City of Vienna within the competence center program COMET.

References

1. K. Bennett, P. Layzell, D. Budgen, P. Brereton, and M. Munro L. Macaulay. Service-based Software: The Future for Flexible Software. In *7th Asia-Pacific Software Engineering Conference*, pages 214–221. IEEE Computer Society Press, 2000.
2. W. Chou and L. Li. WIPdroid A Two-way Web Services and Real-time Communication Enabled Mobile Computing Platform for Distributed Services Computing. In *International Conference on Services Computing*, pages 205–212. IEEE Computer Society Press, 2008.
3. M. Chuah, P. Yang, and Y. Xi. How Mobility Models Affect the Design of Network Coding Schemes for Disruption Tolerant Networks. In *29th International Conference on Distributed Systems Workshop*, pages 172–177. IEEE Computer Society Press, 2009.
4. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions of Software Engineering*, 24(5):342–361, 1998.
5. T. Imielinski and B.R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, 1994.
6. Leonard Kleinrock. *QUEUEING SYSTEMS, Volume I: Theory.* John Wiley & Sons, New York, 1975.
7. Lester Lipsky. *QUEUEING THEORY: A Linear Algebraic Approach.* MacMillan Publishing Company, New York, 2009.
8. Marcel Neuts. *MATRIX-GEOMETRIC SOLUTIONS IN STOCHASTIC MODELS, Revised Edition.* Dover Publications, London, 1995.
9. J. Ott and L. Xiaojun. Disconnection Tolerance for SIP-based Real-time Media Sessions. In *6th International Conference on Mobile and Ubiquitous Multimedia.* ACM Press, 2007.
10. J. Su, J. Scott, P. Hui, J. Crowcroft, E. de Lara, C. Diot, A. Goel, M. H. Lom, and E. Upton. Haggle: Seamless Networking for Mobile Applications. *Krumm, J., Abowd, G.D., Seneviratne, A., Strang, T. (eds.) UbiComp 2007. LNCS*, 4717(5):391–408, 2007.