Reducing User Perceived Latency with a Middleware for Mobile SOA Access

Andreas Göb, Daniel Schreiber
Telekooperation
TU Darmstadt, Germany
{goeb,schreiber}@tk.informatik.tu-darmstadt.de

Louenas Hamdi
SAP Research
SRC Montreal, Canada
louenas.hamdi@sap.com

Erwin Aitenbichler, Max Mühlhäuser
Telekooperation
TU Darmstadt, Germany
{erwin,max}@tk.informatik.tu-darmstadt.de

***Abstract*—Network latency is one of the most critical factors for the usability of mobile SOA applications. This paper introduces prefetching and caching enhancements for an existing SOA framework for mobile applications to reduce the *user perceived latency*. Latency reduction is achieved by proactively sending data to the mobile device that could most likely be requested at a later time. This additional data is piggybacked onto responses to actual requests and injected into a client side cache, so that it can be used without an additional connection. The prefetching is done automatically using a sequence prediction algorithm. The benefit of prefetching and caching enhancements were evaluated for different network settings and a reduction of *user perceived latency* of up to 31% was found in a typical scenario. In contrast to other prefetching solutions, our piggybacking approach also allows to significantly increase battery lifetime of the mobile device.**

***Keywords*-Web-based services; Mobile communication systems; Distributed applications; Buffering; Low-bandwidth operation; Human-centered computing;**

I. Introduction

Employees spend more and more of their work time away from their desk, e.g., visiting customers, performing on-site maintenance, or similar activities. Therefore, access to Enterprise Information Software (EIS) on mobile devices becomes increasingly important. Today, most EIS like Customer Relationship Management (CRM) or Enterprise Resource Planning (ERP) are composed from Service Oriented Architecture (SOA) services. Hence, frameworks like [1]–[4] emerged, aiming to support mobile SOA access. These frameworks make use of several techniques to support the peculiarities of mobile scenarios. For example, they introduce a client-side cache to bridge temporal loss of network connectivity, or use data compression to avoid long download times in low-bandwidth mobile networks.

However, the usability of mobile SOA is still greatly affected by the high latency of mobile network connections. A typical use case for an EIS requires several roundtrips from the mobile device to the SOA infrastructure, as multiple services need to be accessed. Each roundtrip causes a relatively large overhead during connection setup. This leads to noticeable and disturbing delays in the User Interface (UI) [5], reducing the usability of mobile SOA. Leaving the network connection open over long periods to avoid this overhead is not an option, as this consumes too much battery power, which is a scarce resource for mobile devices [6].

In this paper we present prefetching and caching enhancements for an existing framework for mobile SOA access [1], reducing the *user perceived latency*. The *user perceived latency* is the latency measured at the UI level using a typical use case of mobile SOA access, as suggested in [7].

Our enhancements do not decrease battery lifetime for the mobile device, as the prefetched information is sent along with legitimate requests. The prefetching is done at a proxy server, that employs a sequence prediction algorithm to predict future requests. Using a sequence prediction algorithm for prefetching avoids the need to hand-craft prefetching functionality at application level. At the client, the prefetched data is stored in a cache. Thus, no communication overhead occurs, should the data be actually requested by the user at a later point in time.

The rest of the paper is organized as follows. Section II gives an overview of related work on prefetching and caching with a focus on how it applies to the mobile SOA access scenario. Section III describes our approach to reduce the *user perceived latency*. In section IV we explain how we introduce our enhancements in an existing middleware. Section V presents the experimental setup we used to evaluate our approach and describes our findings with regard to improvements in *user perceived latency* and the effects on battery lifetime. Finally, Section VI concludes the paper and gives some perspectives for future work.

II. Related work

We consider the scenario shown in the lower part of Figure 1 to analyze mobile SOA access. A *client* accesses a *proxy* server via a low-bandwidth, high-latency connection, e.g., EDGE and the *proxy* accesses one or more *servers* via high-bandwidth, low-latency connections, e.g., corporate LAN. Although the underlying network latency cannot be reduced, the *user perceived latency* at the client can be reduced by using idle times for prefetching content, either at the client or the proxy. This has been suggested for accessing the web in non-mobile settings e.g., in [8], [9] or [10]. However, these approaches do not apply to the mobile setting we target, as they assume leaving a connection open for long periods of time and continuously using the available bandwidth has no adverse effects. This is clearly

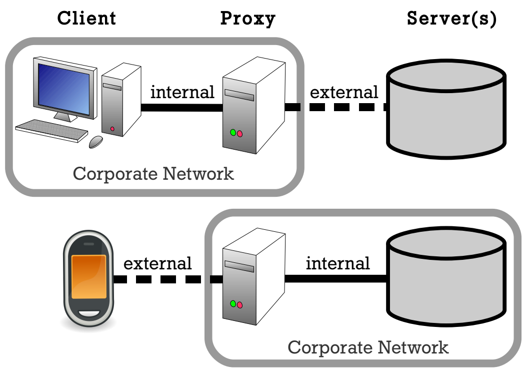

Figure 1. Mobile (lower part) and non-mobile (upper part) setup. In a mobile setup, the position of the low-bandwidth, high-latency link is the opposite compared to the non-mobile setup.

not the case in mobile settings, where energy considerations are important and prohibit persistent network connections. Another difference between mobile and non-mobile settings that needs to be accounted for, is the nature of the network. As shown in Figure 1, the positions of the low- and high-latency links are interchanged.

The importance of considering energy efficiency in mobile settings is also stressed in [6]. However, the prefetching and caching techniques proposed in [6] assume a broadcast scenario, in which clients have to decide when to activate the network connection and actively prefetch data they observe from the broadcast channel. This is not applicable to SOA settings, where single clients invoke several services. Nonetheless, [6] shows that caching and prefetching are appropriate means for achieving both usability and power efficiency in mobile environments. The importance of client-side caching is further supported by the findings in [5], which show that even on high-bandwidth, low-latency connections, the impact of caching is still perceived by the user.

Elbashir et al. propose a general approach for transparent caching of web services for mobile devices in [11]. They note that strong cache consistency is unachievable in a mobile environment, because connectivity cannot be guaranteed. To improve client cache coherency, they propose an architecture where the proxy can notify its clients about data changes and trigger the invalidation of the clients' cache entries. The approach includes connectivity awareness, request rerouting, message queuing and other advanced functionality, which has to be statically configured at the proxy. Therefore, it requires sophisticated configuration on a per-application basis to yield good results. In contrast, providing a generic algorithm for prefetching as part of the middleware at the proxy allows to reduce the *user perceived latency* for all applications, without having to implement this functionality for every application. Also, [11] assumes the client is built using the *.NET* framework, which is still not available on most mobile devices. In our approach the client is completely implemented in JavaScript and

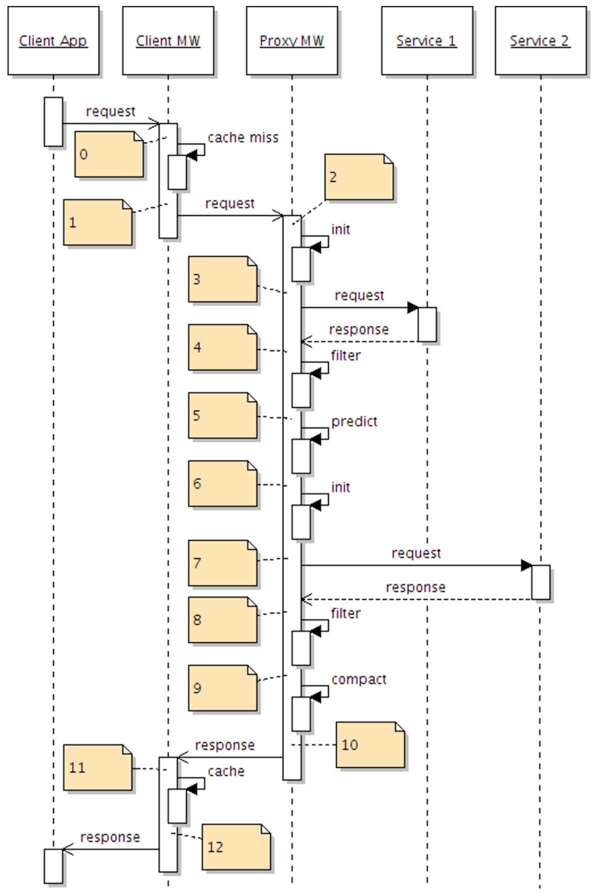

Figure 2. This diagram shows how a request is handled in our architecture and at which checkpoints we took time measurements.

runs on all major mobile browsers (e.g., Google Android, Opera Mobile, Nokia S60 browser, iPhone Safari and Pocket Internet Explorer).

To summarize, reducing *user perceived latency* by caching at the client is possible and already widely used in middleware for mobile SOA access. The approaches for non-mobile web access show, that the *user perceived latency* can be further reduced by prefetching techniques. However, these must be adapted and extended for mobile SOA access to deal with the energy constraints of mobile devices and the highly dynamic content delivered by SOA services.

III. Reducing user perceived latency in mobile SOA access by caching and prefetching

We assume a middleware for mobile SOA access is implemented in a distributed way, with a client part and a proxy server, as shown in Figure 1. This setup is common for such a scenario, as it allows to use compressed protocols instead of XML/SOAP on the mobile network connection. In addition, we assume a client-side cache is used to reduce the need for network requests.

Our improvements to this basic architecture are twofold:

- We introduce an algorithm for predicting client requests in an application-independent way as part of the middleware.
- We introduce a request piggybacking mechanism, allowing the proxy to proactively fill the client cache.

As a result of these improvements the *user perceived latency* of applications is reduced. Requested data may already be in the client cache, as it was proactively added to the cache, piggybacked to the response of a previous request. This improvements are achieved

- without forcing to the application programmer to manually implement prefetching of service for each application
- without the adverse effects on battery lifetime that come with existing approaches for prefetching.

The general approach is as follows. At the proxy, we employ a machine learning algorithm for online sequence prediction to predict future requests of the client. If requests can be predicted with sufficient confidence, the response to these requests is sent along with the response to the original request of the client using a piggybacking technique. The client stores these additional responses in a cache. Thereby, the prefetching happens transparently to the application that cannot distinguish whether the response originates from the cache or the proxy. As we use a compact communication protocol between client and proxy, the overhead of sending the additional piggybacked data over the network is negligible compared to the gained efficiency due to additional cache hits. To verify this hypothesis, we implemented our prefetching and caching enhancements on top of the "AjaxWeaver" platform[1] [1] using an adapted version of FxL [12] as sequence prediction algorithm. It relies on a simple model that is shown to perform very well in practice [12].

We implemented several improvements to the naïve approach sketched above, i.e., to avoid prefetching of critical services and to maintain a consistent request sequence at the proxy and the client.

IV. Implementation

As stated above, our approach is universal and applicable in different scenarios. However, to show its practical value, we implemented it on top of an existing framework. "AjaxWeaver" is an end-to-end infrastructure for composing and consuming mobile AJAX applications, that are connected to SOAP-based web services. In addition to compatibility with many different AJAX-enabled browsers, "AjaxWeaver" provides client runtime and middleware components to facilitate SOA consumption on mobile device browsers. "AjaxWeaver" provides simple methods to reduce SOAP messages and to mitigate the network latency via increased client-side processing.

[1]formerly known as MobileWeaver AJAX

The main parts of the "AjaxWeaver" platform are the client side library, the proxy, and the actual applications, described as XML documents that are interpreted at runtime by the client and the proxy. The client part of the platform runs on all major mobile browsers (e.g., Google Android, Opera Mobile, Nokia S60 browser, iPhone Safari and Pocket Internet Explorer). When an "AjaxWeaver" application is accessed, the client library downloads the application description. The description is interpreted and rendered as HTML UI by the JavaScript library inside the browser. The user is now able to use this application, e.g., a CRM system, to make further requests to the enterprise SOA, e.g., to query customer information. All further requests are routed through the library, where the caching is performed. Requests to the proxy are performed using AJAX techniques.

Every request to the enterprise SOA is processed as shown in Figure 2. In the following, the timepoints T_i refer to the time at the corresponding checkpoint i in Figure 2, the intervals t_i refer to the timespan $T_i - T_{i-1}$.

First, the client application forwards the request to the JavaScript library on the client (T_0). Next, the library decides whether this request can be fulfilled from the client-side cache (T_1). If so, it proceeds at T_{12} by returning the cached result to the application, where the UI is updated. If the request cannot be answered from the local cache, the request is forwarded to the proxy (t_2).

The proxy server is responsible for forwarding all requests to the respective backend services. The reason to use a proxy is twofold. For security reasons, JavaScript in the browser is only allowed to access the server it was loaded from (same origin policy). Therefore, this server has to act as a proxy to forward requests to other servers. In addition, the proxy compresses the responses to save bandwidth [1].

The proxy is implemented as follows. The request from the client is received at T_2. Default parameters, that are left out by the client to save network bandwidth, are substituted and a correct URL for the target service is generated during the `init` step. After this, the request is forwarded to the target service at T_3. We assume this service can be reached via the fast corporate LAN. The service's response arrives at T_4. It is then filtered at the proxy. In this step, e.g., data fields that are not used at the client are filtered out, again to save bandwidth. This step finishes at T_5.

With prefetching enabled, the prefetching algorithm is used to determine whether another service request should be automatically executed in advance. If this is the case, another cycle of request initialization, server request and result filtering is performed between T_5 and T_9. Without prefetching, the steps between T_5 and T_9 are skipped.

Finally, the reply to the client is composed. This reply is available at T_{10}. It consists of the response to the original request and, if prefetching was used, piggybacks the data for the prefetched requests. A condensed text format is used for this reply instead of XML/SOAP, again to save network bandwidth. After generating the reply at the proxy, it is transferred to the client. Finally, the JavaScript library at the client unpacks the results, caches them locally so that they are available for future requests and updates the application. For our analysis, we consider

- the total time used at the proxy: $t_p = t_3 + t_5 + t_6 + t_7 + t_9 + t_{10}$
- the time spent in the backend services: $t_b = t_4 + t_8$
- the time spent at the client: $t_c = t_1 + t_{12}$
- the network time: $t_n = t_2 + t_{11}$

Thus, the total *user perceived latency* t_u decomposes as: $t_u = t_c + t_n + t_p + t_b$

Details of our caching and prefetching enhancements and their effects on the components of t_u are described in the next section.

A. *Caching and Prefetching Enhancements*

The "AjaxWeaver" platform already provides a client-side cache for requests. Whenever a request is made, it is stored in a hashmap using the request URL as key. Our enhanced middleware aims at minimizing the impact of erroneously prefetched data by piggybacking the prefetched data onto responses to legitimate requests of the user. The piggybacked data contains the prefetched response as well as the request URL that was used to generate the response. This information is then stored in the cache so that it is available without another network request, should the prefetched request really be made by the user. Piggybacking the additional information leads to a longer reply, which increases the time t_{11} for transferring this reply over the network. It potentially increases the processing time at the client t_{12}, as a longer response string has to be parsed and the local cache has to be filled with more than one response.

1) Applying Sequence Prediction to SOA prefetching: Our approach uses an online learning algorithm for predicting the requests to prefetch. The prediction itself and the prefetching of additional server responses increase the time t_p taken at the proxy. To predict the next request a user is likely to make, we employ the FxL sequence prediction algorithm [12]. Using a machine learning algorithm avoids the need to reimplement prefetching behavior on a per-application basis. In addition, dynamically adapting to the individual user at hand outperforms statically preprogrammed models [13].

The original FxL algorithm predicts the next action a_{i+1} from a set of possible actions A, knowing the last k actions $a_{i-k} \dots a_i$. To adapt the FxL algorithm for our purposes, we define an action a as the combination of a request q from the client and a response r from a server. Our adapted version thus tries to predict the next request q_{i+1} knowing the sequence $a_{i-k} \dots a_i$, with $a_j = (q_j r_j)$.

The *frequency* $fr(s)$ of an action sequence s denotes the number of times this exact sequence has been observed so far. Following the definition in [12], the score of an action a is computed as:

$$score(a) = \sum_{j=1}^{k} w(j) fr(a_{i-j+2} \dots a_i \circ a)$$

with the weight $w(j)$ being the sequence length, including the action a: $w(j) = j$.
Knowing these scores, it is easy to compute the probability $p(a)$ of an action a by simply summing up the scores of all possible actions:

$$p(a) = \frac{score(a)}{\sum_{a' \in A} score(a')}$$

If a request q_{i+1} has been successfully predicted, we store this request together with the corresponding response r_{i+1} in the history. The history structure is the same for all users across all applications using the same proxy server. This way, even a completely new user can be provided with prefetched data, although the implementation is not able to adapt to a single user's individual profile.

2) Restricting the services eligible for prefetching: Prefetching should only be applied to uncritical, e.g., read-only services, as it could be dangerous if a request due to a prefetch operation would cause irreversible changes that are not desired by the user. Therefore, we augmented the middleware with an option to flag a service as safe for prefetching. As this property is not part of the Web Services Description Language (WSDL) standard, it has to be manually provided by the application developer for every service used. This effort is minimal compared to completely implementing the prefetching at application level. Our experiments showed, that excluding requests and responses to non read-only services from the history $a_{i-k} \dots a_i$ used for prediction of the next request, improves the results. This is probably due to the fact, that the order of read-only requests is relatively stable, and thus can be predicted with good results, whereas the requests to non read-only services are more random and need to be treated as noise.

3) Parameters: The prediction can be controlled by means of several parameters. The *maximum sequence length* k denotes the length of the session history window that is used to construct the prediction model. Greater values may achieve better prediction quality, but they also lead to higher memory requirements and probably to overfitting as well.

The *minimum confidence* value denotes a threshold for the prediction confidence. If the probability for a request is below this threshold, it will not be considered for prefetching, even if it is the best ranked request. This avoids sending unnecessary data in cases where the next request cannot be predicted precisely.

Another control variable is the *maximum number of prefetched requests*. This value limits the number of requests that are processed for prefetching and piggybacked onto the response. The limitation of this value can save processing

#	t_3	t_5	t_6	t_7	t_9	t_{10}	t_p	t_3	t_5	t_6	t_7	t_9	t_{10}	t_p
1	24.46	20.63	0	0	0	1.21	46.3	25.69	19.57	0.79	18.95	29.03	0.97	95
2	40.29	41.51	0	0	0	0.25	82.05	0	0	0	0	0	0	0
3	19.88	40.47	0	0	0	0.25	60.6	16.19	34.2	0.43	11.57	16.07	0.39	78.85
4	25.07	20.44	0	0	0	0.34	45.85	0	0	0	0	0	0	0
5	11.25	10.83	0	0	0	0.27	22.35	12.32	10.45	0.72	11.41	17.99	1.59	54.48
6	26.92	29.52	0	0	0	1.95	58.39	0	0	0	0	0	0	0
7	19.87	38.71	0	0	0	0.23	58.81	18.05	34.03	0.43	11.25	12.54	0.43	76.73
8	25.59	19.37	0	0	0	0.36	45.32	0.57	0.11	0.01	0	0	0	0.69
9	11.44	12.36	0	0	0	0.5	24.3	11.92	15.04	0.56	0	0	1.2	28.72
10	36.2	47.84	0	0	0	0.28	84.32	39.65	48.71	1.07	0	0	0.28	89.71
11	12.23	1.48	0	0	0	0.17	13.88	9.88	1.33	0.43	0	0	0.08	11.72

Table I
PROXY OVERHEAD MEASURED WITH AND W/O PREFETCHING.

time on the middleware and most importantly also limits the amount of data sent to the mobile device. In our evaluation we have limited the number of prefetched requests to one.

All these parameters can be configured in the global proxy settings and apply for all applications using the proxy.

4) Cache Hit Notification: One major problem of proxy based prefetching is the fact that the proxy is not informed of cache accesses, which occur on the client. So once a data object is sent to the client, the proxy does not know whether it is used or not. This means that the request sequences maintained at the proxy do not necessarily reflect the real client access sequences. To overcome this problem, we follow the approach of [9] and keep a separate sequence history on the client to record cache hits and transfer this history to the proxy with the next request. This implies additional complexity and data transfer, but to a very small extent which is far outweighed by the achieved improved prediction quality. Adding this information to the request from the client slightly increases t_1 as well as t_2.

V. EVALUATION

Evaluating our approach with real usage data was not possible, as EIS and corresponding access logs are mostly treated as confidential data by any company. Therefore, we evaluated our enhancements in a scenario based on services provided by the *Enterprise Service Workplace* in the *SAP Community Network*. After a free user registration, these services are publicly available for testing purposes. For this evaluation, three of the provided ERP services were used: The first one is a customer query, which was used to implement a keyword search over all customers. It returns customer names along with their IDs. The second service returns customer data and contact information for the specified customer ID, while the third service returns a list of sales arrangements for a specified customer ID. For the evaluation we used these services in the following way, which resembles a typical use case:

1) Search for the term “chemical”.
2) View the details of one specified customer and close.
3) View the details of another specified customer.
4) View her sales arrangements and close.
5) Search for the term “espresso”.
6) Search for the term “turner”.
7) View the details of one specified customer.
8) View her sales arrangements and close.
9) Search for the term “hollywood”.
10) View the details of a random customer in the list.
11) View her sales arrangements and close.

During each request we record timestamps at the 12 checkpoints shown in Figure 2. Using a fixed scenario does not allow us to evaluate the performance of the sequence prediction algorithm. Thus, the effect of our improvements will be different for realistic scenarios. However, the evaluation shows, that the overhead of prefetching is so marginal, that our approach should be beneficial even with very modest success rates of the sequence prediction algorithm.

An evaluation of the entire system for each interesting combination of clients, networks and services proved to be very complex. Therefore the system evaluation was split up into the following parts:

- Measurement of the middleware overhead caused by the proxy (t_p)
- Measurement of the middleware overhead caused by the client (t_c)
- Measurement of the overhead caused by the backend services (t_b)
- Estimation of the network delay (t_n)
- Calculation of the total delay experienced by the user and speedup caused by the enhancements

A. Proxy Overhead

For this part of the evaluation the proxy server was run on a MacBook with an Intel Core2Duo 2,4 GHz processor and 2 GB of RAM. The backend services were substituted by dummy services running on the same machine. We ran a script in the Firefox 3.0.5 browser on the same machine that automatically went through the 11 steps of the scenario 100 times. Thus, all communication was performed via the loopback network device. The average of the measured times can be found in Table I. The first column denotes the step

#	t_1	t_{12}	t_1	t_{12}
1	1.08	18.07	1.09	18.19
2	1.15	2.70	1.43	0.16
3	1.21	2.99	1.28	5.75
4	1.24	2.74	1.46	0.15
5	1.27	2.51	1.62	35.74
6	1.20	20.10	1.44	0.19
7	3.76	2.91	1.24	6.19
8	1.24	3.39	1.38	0.21
9	1.20	10.76	1.22	10.58
10	1.24	2.73	1.19	2.71
11	1.19	1.43	1.29	1.55

Table II
CLIENT OVERHEAD FOR CACHING MEASURED WITH AND W/O PREFETCHING.

in the scenario from above. All times are average times from 100 evaluations given in milliseconds. The left half of the table contains the values with prefetching disabled. Note the zeros in columns four, five and six. These steps are skipped without prefetching. The right part of the table contains the corresponding values with prefetching enabled. Note the zero rows four and five, and the near zero row eight. In these cases, the request has been predicted in the previous step and was prefetched. Accordingly the client does not request it from the proxy, but retrieves it from the cache. Column eight was not always predicted as a prefetch, as the online learning algorithm had to adapt to the scenario.

B. Client Overhead

To measure the effect of our prefetching enhancements at the client, we used an Apple iPod Touch with firmware 2.2. We accessed the proxy via WLAN and ran the client part of the middleware in the built-in Safari browser. We again ran a test script, performing 100 runs through the scenario with and without prefetching enabled. The results for the measured client overhead t_1 and t_{12} with and without prefetching are shown in Table II. The first column denotes the step in the scenario from above. All times are average times from 100 evaluations given in milliseconds. The left half of the table gives the values with prefetching disabled. The right half of the table give the corresponding times with prefetching enabled. Note the relatively large difference in lines five and six. With prefetching, the request no. 6 is already sent and filled in the cache at step no. 5. Consequently, the time needed for caching in step six is much shorter with prefetching enabled, as no response from the server is parsed.

As one can see, only a very small fraction from the overall *user perceived latency* is due to the client library. This is the case although even the baseline does perform client-side caching, thus emphasizing the efficiency gain provided by simple client-side caching. The additional overhead caused by our prefetching enhancements at the client are nearly non-measurable.

Preset	min delay [ms]	max delay [ms]	Upstream [kbit/s]	Downstream [kbit/s]
GPRS	150	550	40.0	80.0
EDGE	80	400	118.4	236.8
UMTS	35	200	128.0	1920.0
HSDPA	35	200	348.0	14,400.0
local	0	0	∞	∞

Table III
NETWORK CHARACTERISTICS ACCORDING TO [14]

C. Backend Services

The timing of the backend services of course depends on the nature of the corporate LAN and the actual services used. For the services used in our scenario we measured 16 ms as the average time for a single service request. While the overhead without t_b will be used to calculate the speedup achieved in the middleware, the total, including t_b serves as a hint to what the speedup will be for the user in an application scenario. Note, that if a prefetch occurred, t_b is $2*16$ ms, as the time for accessing a backend service has to be added twice, once for the original request and a second time for the prefetched request.

D. Network Delay

To simulate different wireless networks, the network characteristics shown in Table III were used to calculate different response times based on the results of the locally measured values. To do this, we measured the amount of data that had to be transferred between the proxy and the client for each of the eleven requests from the scenario. Given the network delay for an empty packet t_{d0}, the upstream data rate r_{up} and the downstream data rate r_{down}, the overall network delay for a request with size s_{req} and a reply with size s_{repl} is then calculated using the following equation:

$$t_n = t_{d0} + s_{req}\frac{1}{r_{up}} + s_{repl}\frac{1}{r_{down}}$$

E. Results

To determine the average *user perceived latency* caused by the middleware for the various network settings, we added the values measured for the client and the proxy overhead to the network delay, computed from the network characteristics. We thus were able to compute the values for four typical mobile network conditions. To get the total *user perceived latency*, we further added the time spent in the backend services.

Figure 3 shows how the latency decomposes for requests three and four from the scenario for the UMTS network condition. The figure highlights the difference between non-prefetching behavior - roughly equal latency for both requests - and prefetching behavior - slightly increased latency for request three where the prefetching is performed and no

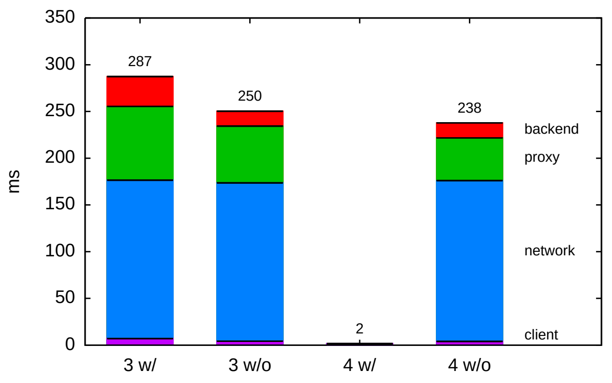

Figure 3. Time comparison for two requests with and without prefetching. Depicted are the UMTS average times for requests no. 3 and 4 of the scenario.

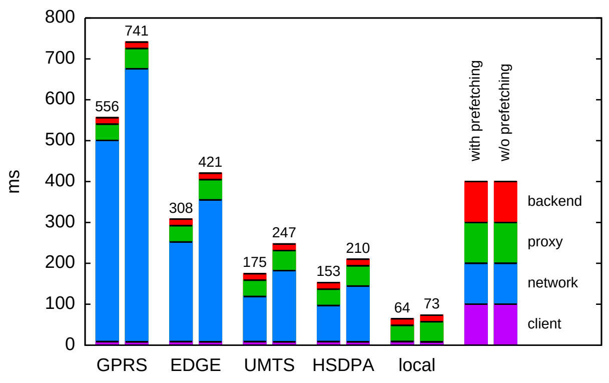

Figure 5. Comparison of user perceived latency in different networks, averaged across all eleven requests of the scenario.

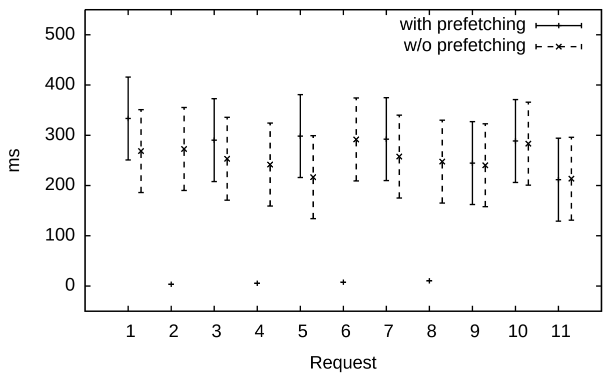

Figure 4. Comparison of prefetching and baseline behavior using UMTS.

noticeable latency for request four that can be fulfilled from the client cache.

Figure 4 contrasts the results of prefetching and non-prefetching behavior over the whole scenario for the UMTS network condition. Prefetching results in a zig-zag behavior with a lower mean latency than in the non-prefetching condition. The lower end of the bars indicates the minimum latency, the tick in the middle the mean latency, and the upper end the maximum latency.

Figure 5 shows the results averaged over all eleven requests from the scenario with all considered network profiles. For the GPRS network profile with average latency, we found an overall speedup of 26%. For the HSDPA profile with average latency the speedup was 29% and 31%, if we take only the middleware into account and neglect the time spent in the backend services.

To estimate the effect of our enhancements on the energy consumption on the mobile devices, we analyzed the average accumulated network times for the scenario. Shorter network times directly correspond to power savings. As shown in Figure 6, the network time is indeed shorter with prefetching enabled, thus prefetching helps to save battery power.

F. Discussion

The differences in *user perceived latency* without prefetching are solely caused by the differences in the response size and the corresponding network time. The impact of prefetching is clearly visible (see Figure 4): On one hand, some of the requests show response times of close to zero, obviously caused by cache hits for prefetched requests. On the other hand, the requests in between show slightly higher latency, caused by the prefetching process in the middleware. Requests 9, 10 and 11 have similar latency in both cases. This is caused by the selection of a random customer in step 10, so that no prefetching candidate exceeds the minimum confidence threshold, and therefore no prefetching is performed.

The average latency per request is smaller when prefetching is enabled for all network conditions. The gained speedup depends on the network latency (larger latency increases the speedup effect) and network bandwidth (larger bandwidth increases the speedup effect). The reason for the former is obvious, as the higher the network latency, the more time is saved by prefetching a request. The latter is due to the increased response size caused by the prefetching. Thereby, larger bandwidth reduces the overhead of prefetching, which also improves the speedup.

The speedups achievable for realistic use cases depend on the performance of the sequence prediction algorithm. However, our implementation shows, that prefetching can be implemented with nearly no drawbacks. Therefore, a badly performing prefetching algorithm does not affect the *user perceived latency*, while a well performing prefetching algorithm will greatly improve performance.

To compute the rate at which the prefetching overhead needs to be compensated by additional cache hits, we divide the average overhead caused by prefetching by the average duration of a single request without prefetching. This yields the needed accuracy for the sequence prediction algorithm. For the UMTS condition in our scenario, we compute a value of 37.98 ms for the average overhead of prefetching (steps 1, 3, 5 and 7 from the scenario) and 253.40 ms for the average duration of a request in the non-prefetching condition. Thus, our approach will have a positive effect in a realistic scenario, if a request is predicted correctly at only a rate of 15%, which is a realistic number for FxL [12].

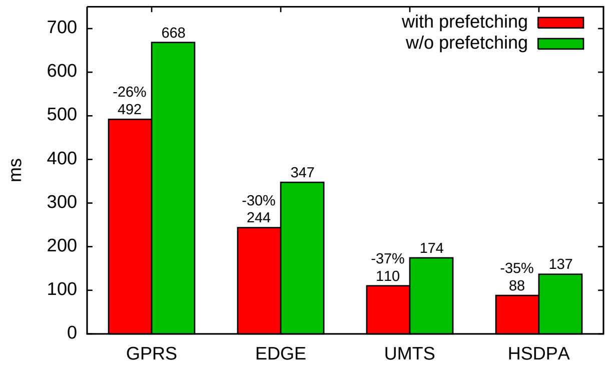

Figure 6. Comparison of accumulated average network times. Reduction of network time directly corresponds with battery power saving.

VI. Conclusions

Our results show, that prefetching at the proxy for a mobile SOA middleware can indeed improve user experience by decreasing the *user perceived latency*. The results for different networks are summarized in Figure 5. It can be implemented without negative effects on the client's battery lifetime and often will even have a positive effect. Furthermore, prefetching at the proxy comes at almost no additional cost for the developer, as a sequence prediction algorithm can be used to dynamically detect services to prefetch. Manual configuration as in [11] is not necessary with this approach. This should generally justify the adoption of prefetching at the proxy for mobile SOA access.

While we were motivated by giving mobile employees access to EIS, this approach applies to other scenarios as well. Quick response times and a high degree of usability are important in scenarios like mobile disaster management [15]. Enabling access to the infrastructure via off-the-shelf mobile phones could be a valuable approach to distribute access to important information as quickly as possible.

In our future work we aim to improve the middleware by parallelizing the process at the proxy server. While the request to the second, prefetched service is still performed, the proxy can already transfer the result of the first service to the client, which should further reduce the overhead of prefetching. However, such an approach is difficult to implement if we want to stay compatible to all major mobile browser platforms. Also, we aim to improve the sequence prediction algorithm using real sets of usage data.

Acknowledgement

We thank SAP CEC Darmstadt for funding part of this research in the AUGUR project. We further thank SAP SRC Montreal for providing the "AjaxWeaver" middleware.

References

[1] L. Hamdi, H. Wu, S. Dagtas, and A. Benharref, "Ajax for Mobility: MobileWeaver AJAX Framework," in *Proceedings of the WWW*. ACM, 2008, pp. 1077–1078.

[2] Y. Natchetoi, V. Kaufman, and A. Shapiro, "Service-oriented Architecture for Mobile Applications," in *Proc. of the 1^{st} Intl. Workshop on Software Architectures and Mobility*. Leipzig, Germany: ACM, 2008, pp. 27–32.

[3] R. Tergujeff, J. Haajanen, J. Leppanen, and S. Toivonen, "Mobile SOA: Service Orientation on Lightweight Mobile Devices," in *IEEE ICWS*, 2007, pp. 1224–1225.

[4] H. Wu, J. Grégoire, E. Mrass, C. Fung, and F. Haslani, "MoTaskit: A Personal Task-Centric Tool for Service Accesses from Mobile Phones," in *Proc. of the 1^{st} Workshop on Mobile Middleware*. Leuven, Belgium: ACM, 2008, pp. 1–5.

[5] M. Pervilä and J. Kangasharju, "Performance of Ajax on Mobile Devices: A Snapshot of Current Progress," in *2^{nd} Intl. Workshop on Improved Mobile User Experience*, 2008.

[6] G. Cao, "Proactive Power-Aware Cache Management for Mobile Computing Systems," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 608–621, 2002.

[7] J. Domènech, A. Pont, J. Sahuquillo, and J. A. Gil, "A User-Focused Evaluation of Web Prefetching Algorithms," *Computer Comm.*, vol. 30, no. 10, pp. 2213–2224, 2007.

[8] V. N. Padmanabhan and J. C. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," *ACM SIGCOMM Review*, vol. 26, no. 3, pp. 22–36, 1996.

[9] L. Fan, P. Cao, W. Lin, and Q. Jacobson, "Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance," in *ACM SIGMETRICS*, 1999, pp. 178–187.

[10] A. N. Eden, B. W. Joh, and T. Mudge, "Web Latency Reduction via Client-Side Prefetching," in *IEEE ISPASS*, 2000, pp. 193–200.

[11] K. Elbashir and R. Deters, "Transparent Caching for Nomadic WS Clients," in *IEEE ICWS*, 2005, pp. 177–184.

[12] M. Hartmann and D. Schreiber, "Prediction Algorithms for User Actions," in *Proc. of Lernen Wissen Adaption, ABIS 2007*, A. Hinneburg, Ed., Sep. 2007, pp. 349–354.

[13] M. Hartmann, D. Schreiber, and M. Mühlhäuser, "Tailoring the Interface to Individual Users," in *5^{th} Intl. Workshop on Ubiquitous User Modeling*. ACM, 2008.

[14] Google Inc. (2009, Mar.) Android Emulator Documentation. [Online]. Available: http://developer.android.com/guide/developing/tools/emulator.html

[15] L. Juszczyk and S. Dustdar, "A Middleware for Service-Oriented Communication in Mobile Disaster Response Environments," in *MPAC Workshop*. ACM, 2008, pp. 37–42.