Implementing Offline Work in Web Applications for Rich Domains

Edgar E.M. Gonçalves  
INESC-ID/Technical University of Lisbon  
Portugal  
Email: edgar.goncalves@ist.utl.pt

António Menezes Leitão  
INESC-ID/Technical University of Lisbon  
Portugal  
Email: aml@gia.ist.utl.pt

Abstract—Web applications are the preferred information system platform for many organizations. Some tasks, though, must be done offline, either for connectivity costs and reliability or security constraints. Still, offline work isn’t a common feature in Web applications, for it demands a more complex software architecture and a difficult development process. Existing applications are even harder to adapt with offline support, usually implying writing alternate versions of its code.

We provide a novel development methodology that speeds up the implementation of offline work in a Web application with a rich domain. It is based on the deployment of the server functionalities on the local machine. Replicating all data to the local server isn’t practical for application’s with large domain. We overcome this by enhancing the local server with an adaptive prefetcher that, while online, keeps fetching useful data from the remote server. To select the best suggestions, heuristics are applied to profiled data.

We discuss the application of this methodology in a Web server prototype. We discuss the choice of heuristics, and enhance the prototype with a statistics collector, to measure their usefulness. We also show how we can identify functional modules and find errors on the application by using the collected information.

Keywords—offline; web application; software engineering

I. INTRODUCTION

Nowadays Web based interfaces are taking control over most organization’s applications. These applications usually require a permanently available Internet connection. However, expensive mobile access and unreliable connectivity motivate the need to work offline. Enhancing existing functionalities with offline capability isn’t feasible with a single-step transformation: not only there are certain functionalities that cannot be performed on a disconnected environment, but also the application’s initial development was, most likely, oblivious of this new requirement. This represents our main motivation for providing a new development methodology.

We propose a novel replication system, from the remote server to the local machine, that will ultimately allow the latter to work offline. This is supported by an adaptive prefetching system that applies heuristics to profiled user interactions, suggesting the most useful information for the client’s next actions. This way, before disconnecting, the client is able to prefetch data that the server suggests it is going to need to work offline. Statistical analysis becomes possible in the presence of large user bases: since most execution flows tend to be stereotyped, common use cases are bound to be performed several times, and this tendency can be used to prioritize suggestions.

Our work benefits application’s users, in that the contents suggested to them will be more and more accurate with time (as the statistics collected into their personal profile get more significance, the suggestions become more relevant). New users may also benefit: they will automatically make use of a global profile, where statistics are collected from other users of the same role/type. Application development is also improved because our results may be used to identify functional modules, coding flaws, and to fine-tune the adaptive mechanism to a given application domain. This constitutes an alternative method for testing the application correctness.

We proceed by presenting an illustrative scenario to motivate our work, and describe current problems related to offline work on Web applications. Afterwards, we describe our solution in detail, and we show how the choice of heuristics impacts the usefulness of the suggestion system. We also discuss the results taken from implementing a working prototype with our methodology. We finalise the paper with our conclusions and future work directions.

II. CASE STUDY - FEARS

FeaRS (available at https://fears.ist.utl.pt/) is a feature request Web application implemented using a J2EE-based framework and Software Transactional Memory. Its functionalities are self-contained, and thus, good candidates for enabling offline work. All use cases revolve around issues suggested by users to projects. Those issues can be voted and commented on by all users, and go through a series of states until the issue is solved. All use cases currently work online only and, with some exceptions (e.g., retrieving the up-to-date comments) they can be adapted to work offline. The resulting workflow enables a user to work online for a period, disconnect and work offline and synchronize his work after he reconnects.

III. EXISTING PROBLEMS

Current approaches to develop offline work either deviate from the Web browser[1], [2] or focus on making development from scratch easier[3]. Tackling existing applications, specially with large and rich domains, doubles the
programming effort, as this is currently done by creating a client-side scripted version of the server. Also, some services need to be rewritten, since most application designs are based on a central service repository, where each service can safely access both code and data and these may not be present locally. Easily adding offline work on existing online applications remains an unsolved task.

Moving from a simple request-response workflow to an offline workflow requires adding a replica of the remote server, to the local machine. This local server acts as a proxy when online, while storing enough information to allow the user to disconnect and keep working. Web caches and prefetching systems (as explained in [6]) can both be used to this effect. The local server becomes the active replica and registers all executed operations, so that when the connections gets back, both servers may synchronize their states. This is a multi-step operation: 1) All work done by the user is merged into the remote server; 2) Unless there are merge conflicts, the local state is refreshed with remote data that was modified while the user was offline; 3) Information that was missed during offline work is fetched to the client, thus avoiding the causes for the already experienced execution flow stops. Synchronization issues arise when we consider concurrent accesses to the application — the entire offline activity should be treated as a long transaction, and merge conflicts should be handled so that work loss is minimized. These are research subjects on database systems and are addressed on [6].

Disconnected operation isn’t exclusive to Web applications. In fact, it is already contemplated in distributed systems, but their approaches aren’t suited to the Web architecture.

One issue in this offline transformation is how to select what functionalities are suitable to work offline, and what information they require. Replicating all information on the remote server is not feasible, due to the user’s access rights. Workflow research, based on colored Petri Nets, propose the fragmentation of processes in isolated activities, but this must be done at design time (e.g., the programmer must know beforehand what data the local server is going to use), and to do this on existing large applications would require a significant effort from the programmers. Another approach is to use code coverage tools — along with unit testing, to make sure all code has been explored — to isolate functional dependencies on the application code. However, not only it is impossible to identify optimal usage of a given code, we can’t assure the presence of unit testing on most Web application.

IV. PROPOSED SOLUTION

We propose a novel development methodology to enhance existing Web applications, with offline work for elected functionalities. The remote server is adapted to profile user activity. It is replicated to the local machine. Finally, it is enhanced with an adaptive prefetching mechanism that will help populate the local storage with information specifically picked for the user and for his current activity trends. With time, the user is able to disconnect and work exclusively with the local server, for the most common functionalities.

A. Requirements

Our target application must be coded in an Object-Oriented paradigm, with clear separation of the data access, business logic, domain, and user interface layers. We require the entire data graph to be traversable (e.g., using an Object Relational Mapper). Access control must also be done at the object property level. We also require the server code to be instrumentable (e.g., in the J2EE platform you could use annotations or code injection to intersect classes and methods descriptions).

This approach works best with a large, rich domain, with many users, since profiled data will be much more relevant. Note, however, that for a functionality to work offline it must be able to run isolated (i.e., without external dependencies such as real-time online data). Thus, we focus on domain-centered functionalities (e.g., content management tasks).

B. Server Profiling

We have to find out what code and information we are going to require to work offline. We can achieve this, based on the premise that a large user base will, eventually, perform all the relevant execution flows the application offers. We register run-time activity on the service and on the domain layer, and store it into user profiles, as well as on an aggregated global profile (to make inferences based on, e.g., a user role).

We collect data on each service execution (keeping, e.g., call stacks). Doing so can be simple if the application has a single point of service execution, like a service manager, or if each functionality follows the Command pattern and thus is implemented as first-class instances of an instrumentable class. We also instrument the code related to data access, at the object properties level. This can usually be achieved if all domain objects inherit from a common type, if we are able to attach behavior to this super-type such as method observers on accessors and instances initializers. Byte-code injection is another alternative to get this.

Profile data durability becomes an issue with server code upgrades, as collected data may loose meaning. Pessimistic solutions would invalidate the entire profile and start over, but that would feel awkward for users. Optimistic approaches assume most code upgrades don’t disrupt the entire code-base: users use collected data until one of them leads to an invalid situation. Then, we perform a fine-grain invalidation, keeping untouched and unused code as is.
C. Server Replication

We propose the deployment of the remote server on the local machine. This avoids having to code another one from scratch in a different language, and it is feasible given current client machines are capable of running most run-time engines. However, we must enhance both servers so that they can serialize, transfer and merge data between them. Also, since the data is stored in an object graph that can’t always be fully transportable, we require a proxy system that lazily fetches the required data from the remote server; lightweight proxy objects are kept in their place, locally, until an access attempt is made. Proxies allow data splitting at the property level, enabling restricted access to an instance when required. The local server starts with its storage empty. It operates on two modes: it acts as a message forwarder while accessing the remote server (and updating local state accordingly), and it replaces the remote server when offline.

D. Prefetching Mechanism

To populate local storage we add a prefetching system on the local server, at the instance level (i.e., the local server asks for instances remotely stored without needing them). This can use idle times, following the Predictive Fetch design pattern[24]. Implementations may make use of periodic asynchronous HTTP requests, Comet frameworks or web workers/worker pools (see [25]). Inter-server communication code must be adapted by making the remote server produce suggestions and the local sever handle them.

E. Adaptive suggestion producer

Election of the instances that are to be prefetched is done using a suggestion providing mechanism. We propose the use of heuristic functions that, based on statistics gathered from the profiled data and current context, indicate remotely stored instances that the local server will benefit from prefetching. The resulting behavior is that the local server becomes populated with data that would likely be used on following actions. To provide feasible counts of the suggested instances set, we propose limiting the output length of each heuristic and/or using a suggestion buffer.

Heuristics must be validated. By extending the servers to monitor both suggestion creation and usage, we can gather statistics and determine whether the used heuristics are being useful. Since prefetching is an information retrieval problem, we monitor the number of suggestions, hits (proxy objects resolved to locally stored objects) and misses (failed proxy resolution attempts) for each offline interaction. We can then derive computed metrics[26] such as hit/miss-ratios, recall and precision. Analysis of such information will indicate if we need to change/fine-tune the used heuristics.

F. Development Aspects

By analysing collected statistics (relative to both domain and heuristics) over time, we validate the prefetching’s success (e.g., decreasing miss-rates). We also have information to detect isolated modules: services, data types and queries required to perform an activity. We can detect application errors, e.g., when there’s an unexpected spike on the number of accesses made during a service to a certain data class. This type of statistical testing isn’t meant to replace unit testing, but can be used along with it. Detecting what code/data is being accessed by each user type can also be used to validate both access control and intellectual property constraints.

V. Implementation and Observations

We have put the proposed development methodology in practice, on a prototypical environment. We have implemented a FeaRS clone in a Web server, and reengineered it following the steps mentioned in the last section. In order to make both servers able to swap definitions on the fly, we implemented an on-demand loading mechanism of both services and classes - this way the local server can start empty and be filled with the required/authorized definitions only. Inter-server communication wasn’t hard to implement, since both servers already accepted HTTP requests. For the client to know about the local server, we had to make minor changes to its user interface (we used centralized server calling mechanism, so changes weren’t scattered). We created a graphical analysis page, to watch all collected information and manage heuristics. To simplify prefetching, instead of using a web worker or setting up a Comet framework, we took advantage of all regular requests, piggybacking suggestions with them. On the server side, most changes were located on the data types and the service calling mechanism, using the Adapter and Command design pattern. Changes did not depend on the application domain size/complexity, but on the framework structure. The changes to the application itself were only to disable some features when offline (such as sending an email).

The implemented heuristics gave us a clear notion that usability depends on whether they are domain-aware (such as suggesting all comments for the accessed issues) or simply access instances properties (such as created time and data type). Other heuristics can be further explored, by prioritizing suggestions based on their statistical “success”.

VI. Conclusions

In this document we’ve proposed a novel development methodology to enhance existing Web applications with an adaptive data prefetching system. Our target applications, with intensive and rich domains, are replicated between a local and a remote server. They’re enriched with an adaptive suggestion system, supported by a statistical data collection system, from profiled executions. Before disconnecting, the client is modified in order to prefetch such suggestions.

We showed how we can take advantage of the collected information to perform both isolated modules and error detection. We also applied the methodology on a prototype
environment, and discussed the relevant aspects on this experience; offline support was working as expected.

We claimed that by fine-tuning heuristics based on the domain and by having statistics collected from functionalities’ execution by several users, it becomes possible for a client to automatically (and transparently) prefetch all the information he is going to require to perform its usual functionalities while disconnected from the remote server. The implementation showed us that this was indeed possible.

In a prototype environment we focused on the development process. We will apply our solution to FeaRS, a J2EE Web application in production, taking advantage of a larger user-base. This will allow us to extract statistical comparisons between several heuristics, and study which kind (general-purpose, domain-specific or statistics-based) is best suited to that application’s domain. In order to assert our solution’s feasibility, we will compare performance before and after the methodology application. We intend to use a long transaction to support all offline work, joining efforts with ongoing work at our research group. For future work we leave the implementation of a conflict detection and reconciliation system.

REFERENCES


