Application Specific Communication Stack for Computationally Intensive Market Research Internet Information System

Peter Kurz, Andrzej Sikorski

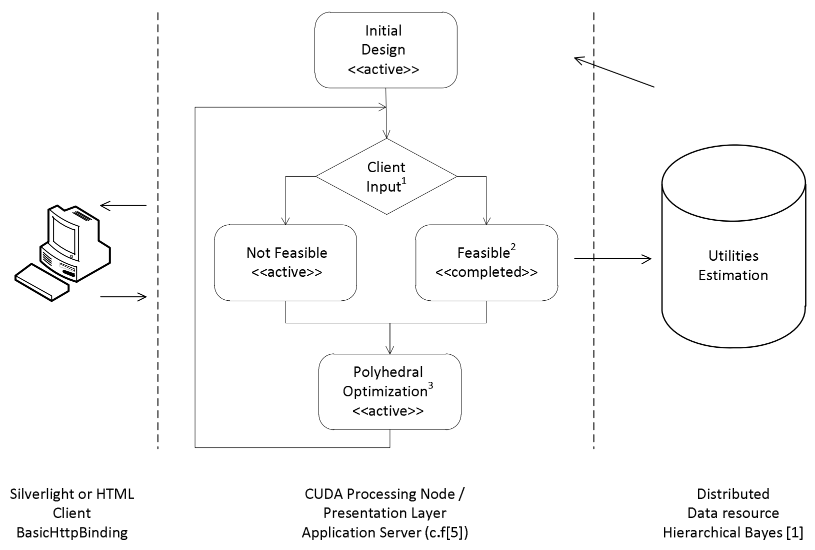
Models and Methods, TNS Infratest Forschung GmbH, Landsberger Str. 284
D-80687 München, Germany
peter.kurz@tns-infratest.com
Technical University Poznań, Faculty of Electrical Engineering, Piotrowo 3A,
60-965 Poznań, Poland
andrzejs@et.put.poznan.pl

Abstract. A robust and reliable transactional processing is a key quality factor in computationally intensive, multilayer information systems. We give three design patterns that model reliable session and transaction management in transactional web applications. These are: session timeout, server default action and split client-server state representation. Only the first design pattern can be successfully implemented with the standard WCF communication facilities available for Silverlight .NET subset. Therefore we include also a simple, robust and WCF compatible communication stack. Surprisingly, overriding the standard WCF facility has resulted in an almost factor 30 performance boost for local calls.

Keywords: web services, transactions, session management

1 Introduction

Choice-based conjoint (CBC) analysis is considered an essential tool in market research industry [1]. It is a simple yet powerful survey method, such that respondents are presented with multiple product profiles and asked to choose the preferred one (maximizing the respondent utility). The main focus among the workers in the field has been to improve the design of product profiles with the aim of maximizing the information gained from the survey. Important academic research in the field led to efficiency improvements, yielding more information from fewer respondents. However, that gains come at a cost of huge computational complexity, that requires adoption of sophisticated algorithms and computational power. In this paper we give a report on the web service integrated respondent survey system based on the Sonnevend [2] polyhedral convex optimization referencing a global, hierarchical state maintained in a distributed database (on hierarchical aspects of utility estimation c.f. [1],[3]).


Initial Design <<active>>
Client Input[1]
Not Feasible <<active>>
Feasible[2] <<completed>>
Utilities Estimation
Polyhedral Optimization[3] <<active>>
Silverlight or HTML Client BasicHttpBinding
CUDA Processing Node / Presentation Layer Application Server (c.f[5])
Distributed Data resource Hierarchical Bayes [1]

[1] The respondent state may yield an estimation that violates computational feasibility/this requires reliable communication between the client and the presentation layer application server
[2] Only feasible utility estimates update the global state
[3] A new profile (conjoint choice tasks) is generated with Sonnevend algorithm

Fig. 1. The architecture an adaptive Market Research multilayer survey system. The application server includes CUDA computational and Silverlight presentation layer components. The survey session can be either in completed (feasible) or active state.

From the software system perspective the adaptive survey process and the underlying convex optimization and can be viewed as a regular database transaction that includes multiple reads and updates (Fig.1.). The transaction status is a combination of user interaction, client application state variables (Silverlight/c# and html/javascript) and the current state of the optimization process. To handle the state transitions successfully reliable communication channels is required. There are two problems with the standard WCF subsystem of Silverlight, however. Firstly, if the application terminates abruptly , which is the case when the browser shuts down, the standard WCF channels (service proxies) are no longer functional. In consequence, the client has no way to send the finalization message to the application server. There seems to be no workaround for this issue within the communication mechanism offered in Silverlight [6]. Secondly, all communication WCF channels are asynchronous. These might seem to be superior to the synchronous counterparts, and in most cases they are. However, the synchronization and sequential control cannot be easily implemented with *Threading* objects (e.g. *AutoResetEvent*, *ManualResetEvent*), as the initialization and completion routines of an asynchronous call run in the single UI thread [7].

2 Completion Design Patterns

Let us now consider 3 simple but effective design patterns that allow reliable completion of database transactions initiated by a Web/Silverlight client application. These are: timeout based session termination, default server-side action and, the most featured one, state representation split between client and server. The first one, session timeout is in fact the only option presently available within the standard Silverlight communication framework. The two other require the web service call method described in this work.

The state model, assumed in this work, of a transactional Silverlight client is given in Fig. 2. The client sends SOAP messages to the application server, that can either initiate a transaction (*BeginTransaction*) or complete it. There are 2 possible states: *Active* or *Completed*. This can be also understood in a more general way, with *Active* state representing any pool of allocated resources, not just an active database transaction, and the locks acquired thereby. In particular an active transaction models a optimization process in an infeasible state.

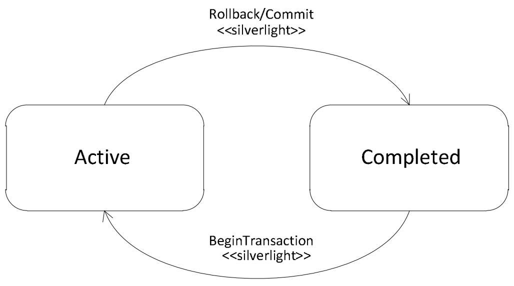


Fig. 2. The generic model of the application server state (c.f. [8]). It is assumed that the client session can be either in a active or completed transactional state. Our objective is to ensure that when the client terminates proper finalization is performed, in particular when browser is abruptly closed.

Operation other that *BeginTransaction*, *Rollback*/*Commit* are irrelevant to our model, as we are only interested in actions that require some finalization. Our objective is to ensure that after a transaction is initiated and the application server goes into the Active state, it will be eventually finalized in a proper manner, according to the observed business rules.

2.1 Timeout based session termination

The first option considered herein is to maintain a timer for each active session. This timer is started at the server for each newly created session and restarted each time a new message from a client arrives. If the time after the last message exceeds the designated interval the Timeout event occurs Subsequently the proper actions on the client side are taken and the session terminates. This solution does not require any cooperation from the client.

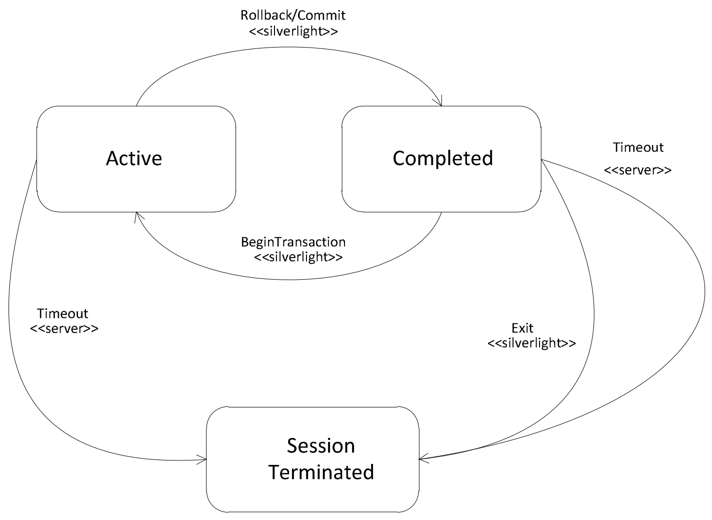


Fig. 3. The state diagram for the timeout design pattern. For each client session a timer is maintained, which fires an event when the designated time elapses.

If the session does not include an active transaction, timeout can be considered irrelevant. Nevertheless it can still occur for a session without any active transaction (i.e. *Completed* state) . In such a case timeout offers an opportunity to remove obsolete sessions and possibly perform some additional housekeeping. The *Timeout* transition between *Completed* and *Session Terminated* states represents this. In both cases this transition is tagged with «server» stereotype, which reflects the fact that it is server side responsibility to handle it.

If the client transaction in a session is not active, the session can terminate leaving the database, or other resource, in a consistent state. The *Exit* transition in Fig. 3. represents a regular WCF asynchronous call that informs the application server that the client is terminating.

2.2 Default server-side action

Another option is to designate a default action that should be performed at server side when a client shuts down. This method requires a reliable communication channel, which must be still available after the WCF facilities has been already closed. This channel is necessary to pass *Finalize* message to the application server. In response the server initiates the default action, that can possibly be a transaction rollback.

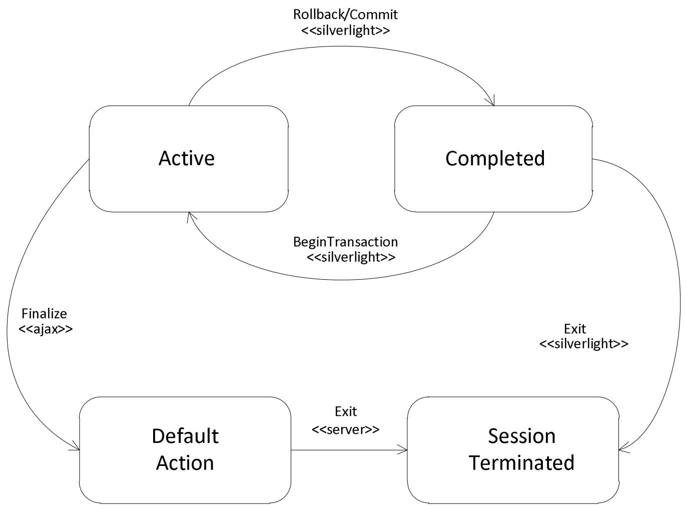


Fig. 4. Default action design pattern. When the Finalize message arrives at the server, default action for the session is taken and the session subsequently terminates.

Evidently the preferred way to shut down an application in the *Active* state is to go through the *Rollback*/*Commit* and *Exit* transitions. In this case the finalization is triggered solely by the standard WCF communication as both transition arcs are tagged with «silverlight». Unfortunately such a scenario is not what Silverlight and the web browser can guarantee.

The client application can be shut in an inadvertent way firing the Silverlight *Exit* event when the session is in the *Active* state. Although WCF is now not available the client can send its *Finalize* message through the channel available within the web browser – which is still active even after WCF shuts down. Because *Finalize* is a regular SOAP message, it can be send either through «ajax» channel (web browser) or standard WCF (not present in Fig.4.). We are going to discuss this opportunity in Sec. 7.

After Finalize message is received, the server performs appropriate handling (*Default Action* state) then the internal event *Exit* is issued and session terminates. Evidently default action for each session can be designated independently. It can be viewed as just another business rule implemented by the application server being a part of a regular data processing. It is also worth noting, that it can be modified by regular SOAP calls as whenever new message arrives a correct completion action may vary according to the current session state.

Seemingly, this pattern is only a minor modification to the session timeout as these two state charts look similar. Yet, there are two important performance gains. Firstly, the server session is terminated immediately after client shuts down, releasing all allocated resources, in particular database locks. Secondly, the session management gets much simpler. To handle the timeout properly, the session manager must either perform a continuous polling of all active entries or order the entries after the pending expiration time.

2.3 Split client-server state handling

The 3rd pattern is the most featured and enables the most flexible processing. We named it “a split state” as now the session state is maintained independently at both ends of the channel. This is reflected in Fig. 5. with labels *Client* - and *Application Server processing*. The idea is that each time the desired completion action changes, client part of the session is updated. This is correct, provided a reliable state transfer is guaranteed. See now, that the *Finalize* method includes one parameter (which can be a composite one) – representing, at the moment when the client is closing, its state.
The 2 arcs corresponding to switching between *Rollback* and *Commit* states represent events occurring at the client side – thus involves no WS calls.

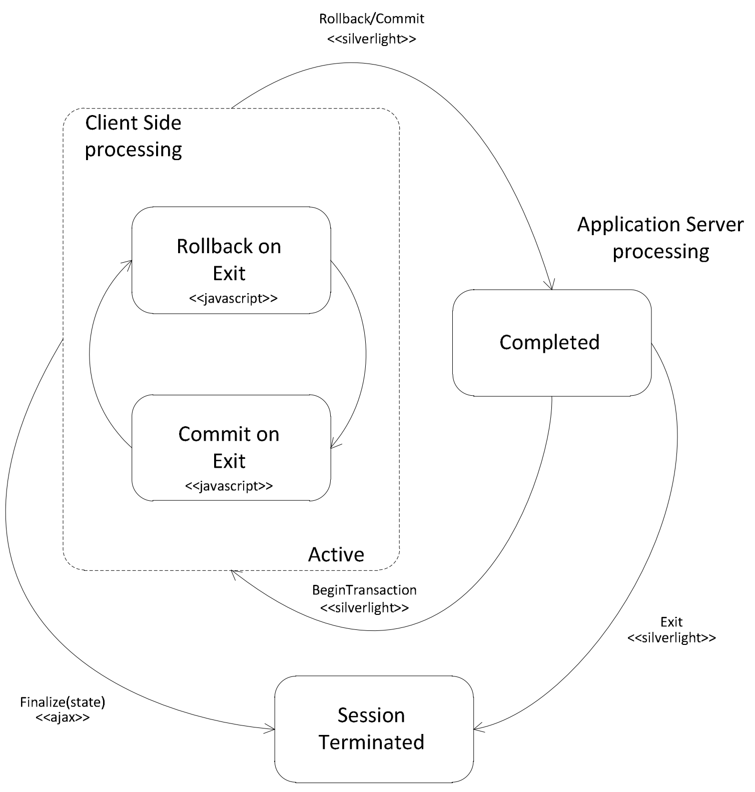


Fig. 5. Split-state management. The client updates its state which is guaranteed to be eventually submitted to the application server.

The client side state can be maintained in 2 ways. First method is to keep the state data in javascript variables, updated each time the session state changes. Silverlight code can easily access the members of its host html page and manipulate them in any way. The evident advantage of this approach is that the sending of *Finalize* message will be now solely the web browser responsibility. When browser window is closed the unload event is fired, therefore a javascript code can handle it, sending *Finalize* message which now includes the representation of the client state. The other option is to keep the current state in the application variable and make it available to java javascript code when needed.

Silverlight supports communication between its application and hosting html page and the state representation can be easily passed. One option is to implement an application class exposing accessors tagged with the *ScriptableMember* attribute. However in our production system we have chosen another way, that is, the current state is passed directly to the javascript routine when *Exit* event occurs.

3 Synchronous Web Service Calls

In this section we are going to describe the main technical component of our solution, namely the utilization of the AJAX /HttpXML object as a communication facility for the reliable message passing. Due to the limitation of Silverlight subset of the .NET framework (COM/DCOM including IDispatch automation is excluded from Silverlight [9]), the AJAX object is not available from within the client application. This is not an issue, as javascript has not such limitation and, as we have already pointed in the previous section, the application and html page can freely communicate.

Our objective is to maintain the maximum compatibility between WCF and AJAX communication. That is, the client side of the communication channel has to remain transparent to the application server. Consequently, the Silverlight SOAP compliant web service are unaware of the call method used at the client side. Both methods (WCF and AJAX) can consume services published via *basicHttpBinding* [9] Fig.6.

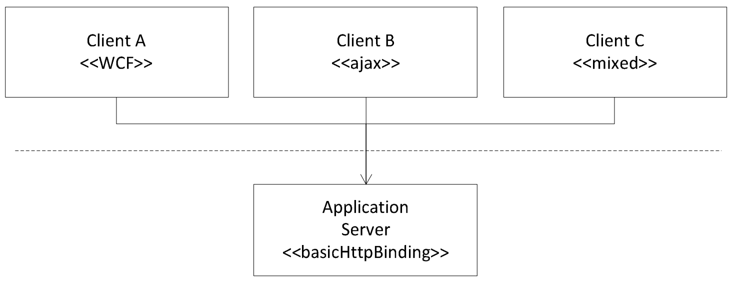


Fig. 6. Different types of client can access a service published via uniform binding (basicHttpBinding corresponds to SOAP in Microsoft terminology). The Client C uses both methods, thus stereotyped «mixed».

Client A is a standard WCF application, Client B uses our method while Client C (stereotyped as mixed) uses both of them (Fig. 6.). The third type of application probably makes little sense as we will see in the next section AJAX call performs much better than WCF. Thus, with AJAX facility already there, it is reasonable to give up WCF at all. As far as, asynchronous calls are concerned, these are still possible as AJAX is , by nature, asynchronous (Asynchronous Javascript and XML) and callbacks can be easily implemented with ScriptableObject classes.

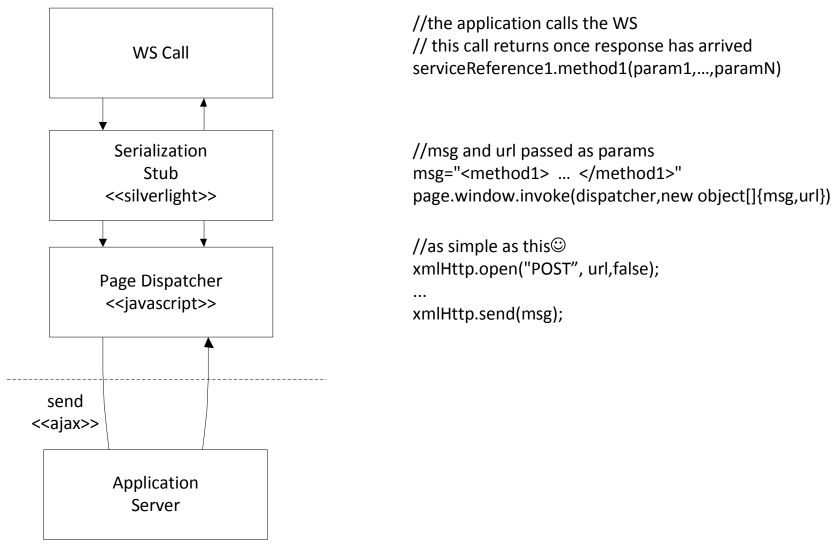
Fig. 7. (c.f. [11]) gives the platform/browser independent initialization code of an AJAX object. For our purposes we need only one such object per client, which runs a single threaded and synchronous process, however nothing prevents the application to create as many as needed such objects. The xmlHttp variable is global and valid until the application terminates. The dispatch routine references this variable in the process of forwarding SOAP messages to the application server (c.f. Fig. 8.).

```
function initCall() {
    try { xmlHttp = new XMLHttpRequest();} catch (e) {
            try {
                xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
            } catch (e) {
             try { xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");}
               catch (e) {return false;}
            }
        }
        return xmlHttp;
    }
```

Fig. 7. The platform independent initialization of the AJAX communication channel [11]

The decisive advantage of AJAX object is that it is available and fully functional even when WCF is closed and Exit event fired. Moreover, it offers also a significant performance boost, contrary to the expectations that javascript could perform worse than C#/Silverlight. Another advantage of AJAX/HttpXML is the flexibility, as the calls can be configured as either asynchronous or synchronous (c.f. [13]).

Fig. 8. outlines the control flow in AJAX based service call. From the application perspective nothing changes with respect to the conventional WCF communication, except that there is no completion routine. The processing is now synchronous. After the call is issued the caller blocks until the response message arrives. Afterwards the response is deserialized and passed back to the caller and the application resumes its execution.


WS Call
Serialization Stub <<silverlight>>
Page Dispatcher <<javascript>>
send <<ajax>>
Application Server

```
//the application calls the WS
// this call returns once response has arrived
serviceReference1.method1(param1,...,paramN)

//msg and url passed as params
msg="<method1>  ...  </method1>"
page.window.invoke(dispatcher,new object[]{msg,url})

//as simple as this☺
xmlHttp.open("POST", url,false);
...
xmlHttp.send(msg);
```

Fig. 8. The communication stack for AJAX based synchronous web service calls. To optimize the performance, the javascript layer of the stack is made extremely compact.

The entity named in Fig. 8. *Serialization Stub* which is a direct equivalent of WCF service reference can be generated automatically in a Visual Studio manner. This is possible with our tool which takes either a WSDL file or an Web Service endpoint as an input and creates the appropriate class definition. This AJAX oriented stub is much smaller and less complex than the standard one, opening up further opportunities for software developers, as generated routines can be subsequently modified or enhanced with application specific code.

4 Performance Evaluation

It is interesting to see how the AJAX based solution performance compares to the standard WCF asynchronous facility. Quite surprisingly, we have observed a significant performance boost when messages are dispatched via html/javascript. Our measurements were performed in 2 variants: local and network call. The predictable differences between them were confirmed by the experiments. (See Tab. 1.)

Table 1. The performance analysis. The network round trip simply adds to the total processing time. The server time is negligible, when the call returns immediately.

	AJAX (synchronous)	WCF	performance ratio
local call	4 ms	120 ms	30.00
network call (round trip 120 ms)	124 ms	240 ms	1.94
network call (round trip 300 ms)	304 ms	420 ms	1.38

To measure the performance of the WCF calls, we have implemented a completion routine that reissues the asynchronous call each time it completes. Each time the response arrives, the client increases the counter and finally, when designated number of call has been performed, it reports the total execution time. The code is given in Fig.9.

```
client.DoWorkCompleted += (s, ev) =>
 {
               if (++i < designatedIterations) client.DoWorkAsync();
               else
               {
                   double d = (DateTime.Now - dt).TotalMilliseconds;
                   status.Text = d.ToString();
               }
   };
dt = DateTime.Now;
designatedIterations = 1000;
client.DoWorkAsync();
```

Fig. 9. The performance measurement loop is simulated with a completion routine which repeatedly initiates an asynchronous call until the desired number of iterations is performed.

The code in Fig. 9. is simple but one thing is noteworthy. The completion is now executed in the main thread of the application. This is an evident departure from the former Silverlight asynchronous call architecture, where the completion code had been always scheduled in either a newly spawned thread or in a designated member of a thread pool [9]. Our guess is that this departure is a result of a confusion among Silverlight developers, annoyed by the requirement to access UI via the dispatcher. The time measurement of synchronous calls is trivial thus omitted.

5 Conclusions and Future Work

The presented solution was initially planned as a mere workaround, intended to remedy the Silverlight/WCF shortcomings with respect to finalization and synchronization. However, being both more efficient and flexible it became a viable candidate to replace the standard WCF in our Silverlight and WCF applications. To make this solution complete, in the future work we are going to enhance our utility tool (Sec. 2) with asynchronous interface and make it source level code compatible with WCF, facilitating greatly migration from WCF to AJAX.

The presented design patterns and communication stack has been already used in multiple market research applications, both internet and intranet. The market research applications relying on sophisticated computational infrastructure and databases are liable to pose considerable challenges for communication subsystem. The challenges present in our implementations included a proper integration of a computationally intensive multi-stage convex optimization (underlying an adaptive conjoint survey c.f. [3],[4]) and the reliable processing of hierarchical database updates. Both components (optimization and database) were published via an uniform, transactional *BeginTransation*, *Commit*, *Rollback*, *Update* interface, what motivated our research in the area of reliable, transaction oriented communication channels. In our future work we are going to give a detailed report on the transactional aspects of numeric optimization.

References

1. Rossi P.E., Allenby G.M., McCulloch R.: Bayesian Statistics and Marketing (Wiley Series in Probability and Statistics), Wiley, Hoboken (2006)
2. Sonnevend G.: An Analytic Center for Polyhedrons and new Classes of global algorithms for linear (smooth, convex) Programming. Proc. of 12th IFIP Conference on System Modeling and Optimization, Budapest (1985)
3. Toubia O., Simester D.I., Hauser J.R., Dahan E.: Fast Polyhedral Adaptive Conjoint Estimation MARKETING SCIENCE, Vol. 22, No. 3, Summer 2003, 273--303, (2003)
4. Nesterov Y., Nemirovskii A.: Interior point polynomial algorithms in convex programming SIAM, Philadelphia (1997)
5. NVIDIA CUDA Conpute Unified DeviceArchitecture, NVIDIA Corp
6. Beres J., Bill Evjen B., Devin Rader D.: Professional Silverlight 4: Wrox, Birmingham (2010)
7. Vaughan D.: Synchronous Web Service Calls with Silverlight: Dispelling the async-only myth. http://www.codeproject.com/KB/silverlight/SynchronousSilverlight.aspx, (2008)
8. Hill D., Webster B., Jezierski E.A., Vasireddy S., Al-Sabt M., Wastell B., Rasmusson J., Gale P., Slater P.: Smart Client Architecture and Design Guide, Patterns and Practices. Microsoft, Redmond (2004)
9. John Papa J.: Data-Driven Services with Silverlight 2: O'Reilly Media, 2008
10. Cleeren G., Dockx K.: Microsoft Silverlight 4 Data and Services Cookbook. PACKT Publishing, Mumbai (2010)
11. Darie C., Brinzarea B., Chérecheș-Toșa F., Bucica M.: AJAX and PHP Building Responsive Web Applications. PACKT Publishing, Mumbai (2006)
12. Zakas N.C., McPeak J., Fawcett J.: Professional Ajax, 2nd Edition. Wrox, Birmingham (2007)
13. Wenz C.: Programming ASP.NET AJAX: Build rich, Web 2.0-style UI with ASP.NET AJAX, O'Reilly Media, Sebastopol (2007)