STUDIA INFORMATICA 2011
Volume 32 Number 2A (96)

Andrzej SIKORSKI
Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej

ENSURING ROBUST TRANSACTIONAL PROCESSING IN SILVERLIGHT DATABASE APPLICATIONS

Summary. In this paper a compact, robust alternative to the WCF is proposed. Its purpose is to remedy the shortcomings of the WCF infrastructure. Firstly, WCF channels close immediately after browser exits, compromising session management and transactional processing. Secondly, WCF channels operate only asynchronously, hindering straightforward implementation of sequential business rules. Design patterns of transactional session processing are given. Surprisingly overriding the standard WCF facility results in a factor 30 performance boost for local calls.

Keywords: transactions, session management, web services

NIEZAWODNE PRZETWARZANIE TRANSAKCYJNE W BAZODANOWYCH APLIKACJACH SILVERLIGHT

Streszczenie. Zaproponowano zwięzłą i niezawodną alternatywę dla WCF. Rozwiązano dwa problemy: przedwczesne zakończenie aplikacji, mające negatywny wpływ na niezawodne zatwierdzanie transakcji oraz brak trybu synchronicznego. Zaproponowano wzorce programowe reprezentacji stanu sesji. Uzyskano znaczną poprawę wydajności, w skrajnym przypadku 30-krotną.

Słowa kluczowe: transakcje, zarządzanie sesjami, usługi webowe

1. Introduction

Silverlight web applications are inherently multitier with user interface and interaction residing on the client and business logic and data store on the application server. This is the task of communication subsystem to ensure successful integration and cooperation of these various client and server components. Therefore robust, reliable, efficient and also flexible methods of message passing constitute a crucial factor to the overall system performance and quality. To enable such a message passing, the .Net/Silverlight framework offers a rich set of classes, packaged as the WCF subsystem, Windows Communication Foundation, which is (in Microsoft own words) *designed to offer a manageable approach to distributed computing, broad interoperability, and direct support for service orientation* [1].

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. Secondly, all communication WCF channels are asynchronous [4]. 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[2,3].

2. Completion Design Patterns

Let us now consider 3 simple but effective design patterns that allow reliable completion of database transactions initiated by a 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 given in this work.

All three design patterns assume that some session management is maintained cooperatively by the client and the server. Usually, such a HTTP state management is based on cookies. Indeed Silverlight and WCF offer some support for this technique with HttpContext and HttpSessionState classes. Nevertheless, in our applications (experimental and production), we have purposely overridden the standard ASP compliant facility, making state maintenance leaner and more application oriented.

The state model, assumed in this work, of a transactional Silverlight client is given in fig. 1. 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 (c.f. [5]).

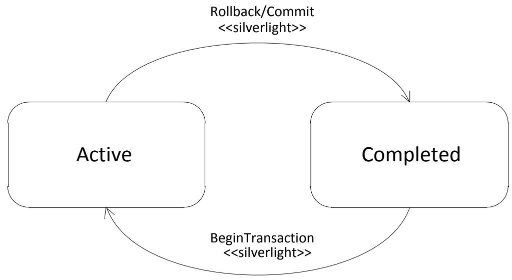

Fig. 1. The generic model of the application server state. It is assumed that the client session can be either in a active or completed transactional state. Our objective is to ensure that when the client terminates proper finalization is performed, in particular when browser is abruptly closed [5]

Rys. 1. Ogólny model stanu serwera aplikacji. Zakłada się, że sesja może być w dwóch stanach. Postawionym celem jest zapewnienie niezawodnego zakończenia transakcji w przypadku przedwczesnego zamknięcia przeglądarki [5]

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.

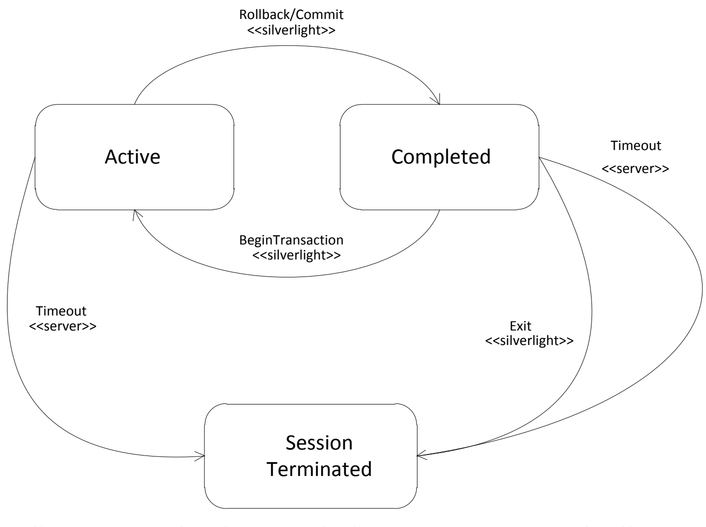

Fig. 2. The state diagram for the timeout design pattern. For each client session a timer is maintained, which fires an event when the designated time elapses

Rys. 2. Diagram stanów dla przeterminowania sesji. Dla każdej sesji utrzymywany jest *timer* lub używa się kolejki globalnej

If the session does not include an active transaction, timeout can be considered irrelevant. Nevertheless it can still occur for an active session without any transaction (i.e. it is in the *Completed* state). In such a case timeout offers an opportunity to remove obsolete sessions and, possibly, performs some additional housekeeping (e.g. finalization of resources). 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. 2 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 the *Finalize* message to the application server. In response the server initiates the default action, that can possibly be a transaction rollback.

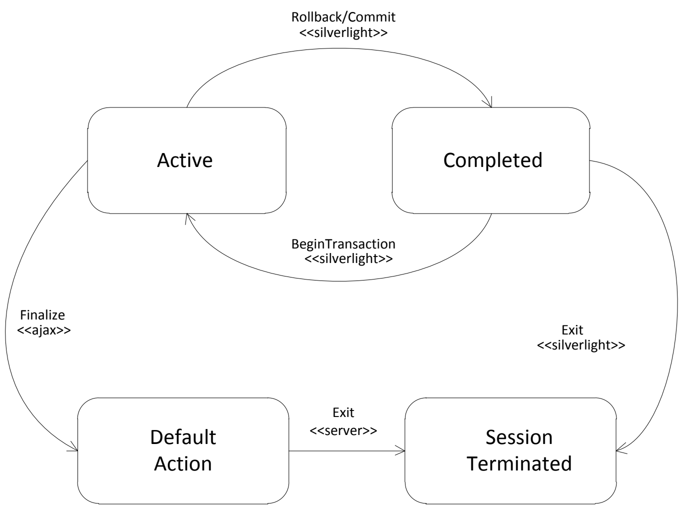

Fig. 3. Default action design pattern. When the Finalize message arrives at the server, default action for the session is taken and the session subsequently terminates
Rys. 3. Domyślna akcja po stronie serwera, podejmowana w momencie przyjścia zawiadomienia od klienta o wygaśnięciu sesji

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 the figure). We are going to discuss this opportunity later, in Sec. 4.

After Finalize message is received, the server performs appropriate handling (*Default Action* state) then internal event *Exit* is issued and the 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 sort the entries by the pending expiration time.

2.3. Split client-server state handling

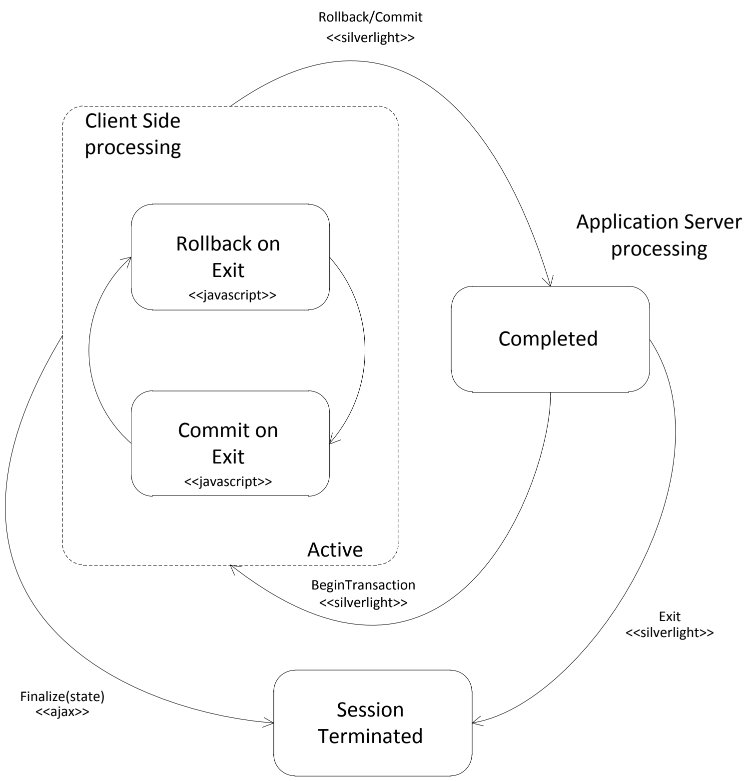

Fig. 4. Split-state management. The client updates its state which is guaranteed to be eventually submitted to the application server

Rys. 4. Rozdzielona reprezentacja stanu. Aktualizacja części stanu odbywa się tylko po stronie klienta, z zagwarantowaniem niezawodnego transferu

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. 6 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. These do involve neither any network communication nor server side action as during these switches transaction remains active.

The client side state can be maintained in 2 ways. The 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.

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 a reliable message passing. Due to the limitation of Silverlight subset of the .NET framework (COM/DCOM including IDispatch automation is excluded from Silverlight c.f. [2]), the AJAX object is not available from within the client application. Fortunately, 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 [1]. See fig.5.

Client A is a standard WCF application, Client B uses our method while Client C (stereotyped as mixed) uses both of them (fig. 5). 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. Asynchronous calls are still possible as AJAX is , by nature, asynchronous (c.f. [7,8]).

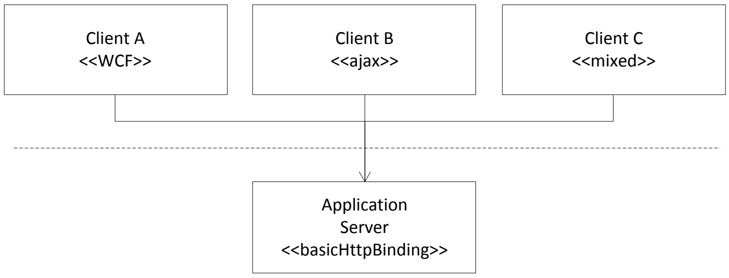

Fig. 5. Different types of client can access a service published via uniform binding (basicHttpBinding corresponds to SOAP in Microsoft terminology). The Client C uses both methods, thus stereotyped <<mixed>>
Rys. 5. Różne typy aplikacji klienckich mogą korzystać z usługi Web. Stereotyp <<mixed>> oznacza, że wykorzystano obydwa typy stosów komunikacyjnych

Fig. 6 gives the platform/browser independent initialization code [6] 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.

```
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. 6. The platform independent initialization of the AJAX communication channel, after [6]
Rys. 6. Platformowo niezależna inicjalizacja kanału AJAX [6]

The decisive advantage of AJAX object is that it is available and fully functional even when WCF is closed and Exit event is 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.

Fig. 7 outlines the control flow in AJAX based stack. 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.

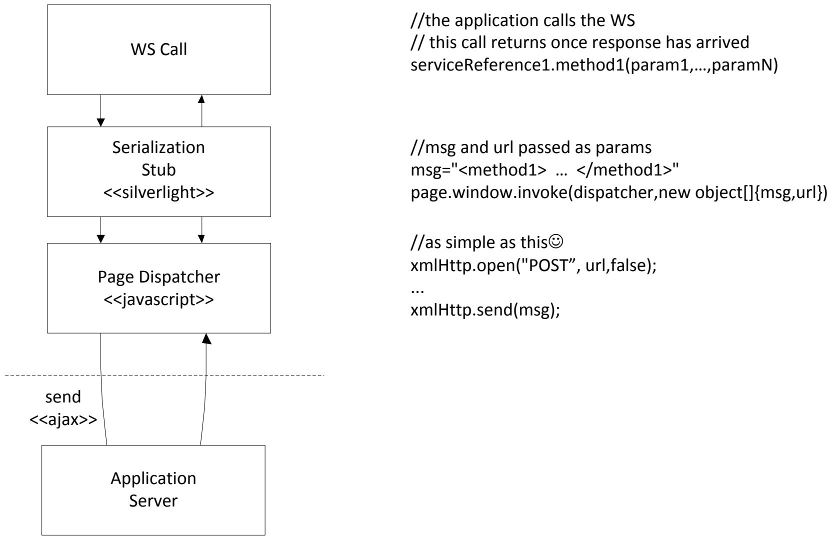

Fig. 7. The communication stack for AJAX based synchronous web service calls. To optimize the performance, the javascript layer of the stack is made extremely compact
Rys. 7. Struktura stosu komunikacyjnego dla wywołania opartego na AJAX. W celu zapewnienia optymalnego czasu wykonania kod w javascript ograniczono do minimum

The entity named in fig. 7 *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 (fig. 8).

	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

Fig. 8. The performance analysis. The network round trip simply adds to the total processing time
Rys. 8. Testy wydajności. Czas sieci ewidentnie, amortyzuje zysk wydajnościowy

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

Rys. 9. Symulacja pętli testującej wydajność wywołania asynchronicznego

The code in fig. 9 is simple but one thing is noteworthy. The completion code is now executed in the main thread of the application. This is an evident departure from the former Silverlight asynchronous call architecture (c.f. [2]), where the completion code had been always scheduled in either a newly spawned thread or in a designated member of a thread pool. 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 (c.f. Sec. 3) 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 [9,10,11,12]) 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.

BIBLIOGRAPHY

1. Windows Communication Foundation – Conceptual Overview. MSDN – Visual Studio 2008, Microsoft, 2008.
2. Beres J., Evjen B., Rader D.: Professional Silverlight 4: Wrox, 2010.
3. Cleeren G., Dockx K.: Microsoft Silverlight 4 Data and Services Cookbook. Packt, 2010. Papa J.: Data-Driven Services with Silverlight 2: O'Reilly Media, 2008.
4. Vaughan D.: Synchronous Web Service Calls with Silverlight: Dispelling the async-only myth. http://www.codeproject.com/KB/silverlight/SynchronousSilverlight.aspx, 16 November 2008.
5. 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, 2004.
6. Darie C., Brinzarea B., Cherecheş-Toşa F., Bucica M.: AJAX and PHP Building Responsive Web Applications. PACKT Publishing, 2006.
7. Zakas N. C., McPeak J., Fawcett J.: Professional Ajax, 2nd Edition. Wrox, 2007.
8. Wenz C.: Programming ASP.NET AJAX: Build rich, Web 2.0-style UI with ASP.NET AJAX, O'Reilly Media, 2007.
9. Rossi P. E., Allenby G. M., McCulloch R.: Bayesian Statistics and Marketing (Wiley Series in Probability and Statistics), Wiley, 2006.
10. 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.
11. Nesterov Y., Nemirovskii A.: Interior point polynomial algorithms in convex programming SIAM Philadelphia 1997.
12. Toubia O., Simester D. I., Hauser J. R., Dahan E.: Fast Polyhedral Adaptive Conjoint Estimation MARKETING SCIENCE, Vol. 22, No. 3, Summer 2003, p. 273÷303.

Recenzenci: Dr inż. Michał Kozielski
Dr inż. Dariusz Mrozek

Wpłynęło do Redakcji 15 stycznia 2011 r.

Omówienie

Osadzony w przeglądarkach Web podzbiór platformy .NET, jakim jest Silverlight służy do implementacji oprogramowania, które jest z natury wielowarstwowe i rozproszone. Dlatego też, cechy operacyjne podsystemu komunikacji mają istotny wpływ na jakość i stabilność oprogramowania, działającego na tejże platformie. W przypadku Silverlight [1] WCF (Windows Communication Framework) stanowi taki podsystem.

Chociaż WCF udostępnia bogatą bibliotekę klas, umożliwiającą korzystanie z rozmaitych mechanizmów połączeń sieciowych (gniazda, usługi Web, klient i serwer http), to jednak nie rozwiązuje on wszystkich problemów związanych z pomyślną implementacją transakcyjnej aplikacji, opartej na technologii Silverlight [3]. Po pierwsze, asynchroniczne kanały transmisyjne zostają natychmiast wyłączone w przypadku nagłego zamknięcia przeglądarki, co jest niepożądane w przypadku, gdy klient przetwarza aktywną transakcję na serwerze aplikacji lub gdy dokonał alokacji pewnych zasobów systemowych. Drugą wadą WCF jest brak przetwarzania synchronicznego [4]. Należy zwrócić uwagę, że reguły biznesowe typowych aplikacji baz danych na ogół mają charakter sekwencyjny. W związku z tym pożądana jest możliwość standardowej synchronizacji, wynikającej z zakończenia przetwarzania przez usługę Web.

W artykule zaproponowano wzorce programowe, mające na celu reprezentację stanu transakcji (czy też w ogólności zasobów zaalokowanych na serwerze aplikacji) w sposób umożliwiający pomyślną jej finalizację (a w ogólnym przypadku zwolnienie zasobów). Tymi wzorcami programowymi są: przeterminowanie sesji, domyślna obsługa przedwczesnego zakończenia oraz stan rozdzielony między serwerem i aplikacją klienta.

Wykorzystanie dwóch z zaproponowanych wzorców programowych (domyślna obsługa i stan rozdzielony) wymaga zastąpienia standardowych kanałów transmisji dostępnych WCF mechanizmem, zaproponowanym w niniejszym artykule. Mechanizm ten korzysta z obiektu AJAX/XMLHttp i polega na wygenerowaniu komunikatu SOAP (lub dokumentu XML) przez *proxy*, rezydujące po stronie klienta.

Zaprezentowana architektura (wzorce oraz stos komunikacyjny) zostały z powodzeniem użyte w rozproszonych aplikacjach badania rynku i opinii – integrujących w sobie komponenty bazodanowe oraz zaawansowane procedury optymalizacji numerycznej (por. [9,10,11,12]). Oprócz korzyści wynikających ze zwiększonej niezawodności oraz uproszczonej implementacji reguł biznesowych uzyskano znaczny wzrost wydajności. Raport dotyczący doświadczeń implementacyjnych, uzyskanych z wykorzystaniem zaproponowanej architektury, będzie przedmiotem dalszych artykułów. Planowane jest również rozszerzenie funkcjonalności narzędzi programistycznych umożliwiających prostą migrację z WCF do AJAX (dla trybu asynchronicznego).

Address

Andrzej SIKORSKI: Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, ul. Piotrowo 3A, Poznań, Polska, andrzejs@et.put.poznan.pl.