Tutorial 8 – Build resilient, responsive and scalable web applications with SocketPro

Contents:

1. Introduction

As software developers, we want our applications to be responsive and scalable to make users really engaged and pleased, but it's usually a challenge to write code that behave like that. Responsiveness and scalability are essential quality-of-service attributes of

Web applications. If your site responds to user requests sluggishly, your customers will not return, and search for other company products instead. Scalability is equally significant when your site traffic becomes large. Bad responsiveness and scalability are rooted in one or more problems due to web browser side, web server front-tier, middle-tier, backend database tier and hard ware.

SocketPro provides a number of unique ways to help solve these problems especially in web server front-tier and middle-tier as well as web browser side. If your web server or middle tier requires local data caching from backend databases, SocketPro also is extremely helpful because you can easily realize real-time update on data caching through pushing changes from database by use of triggers, stored procedures

This tutorial sample projects are located at the directory ../socketpro/tutorials/(csharp|vbnet|ce)/rado.

2. SocketPro ways for resilient, responsive and scalable web applications

Figure 1 below visually summarizes SocketPro features for helping you create responsive and scalable web applications.

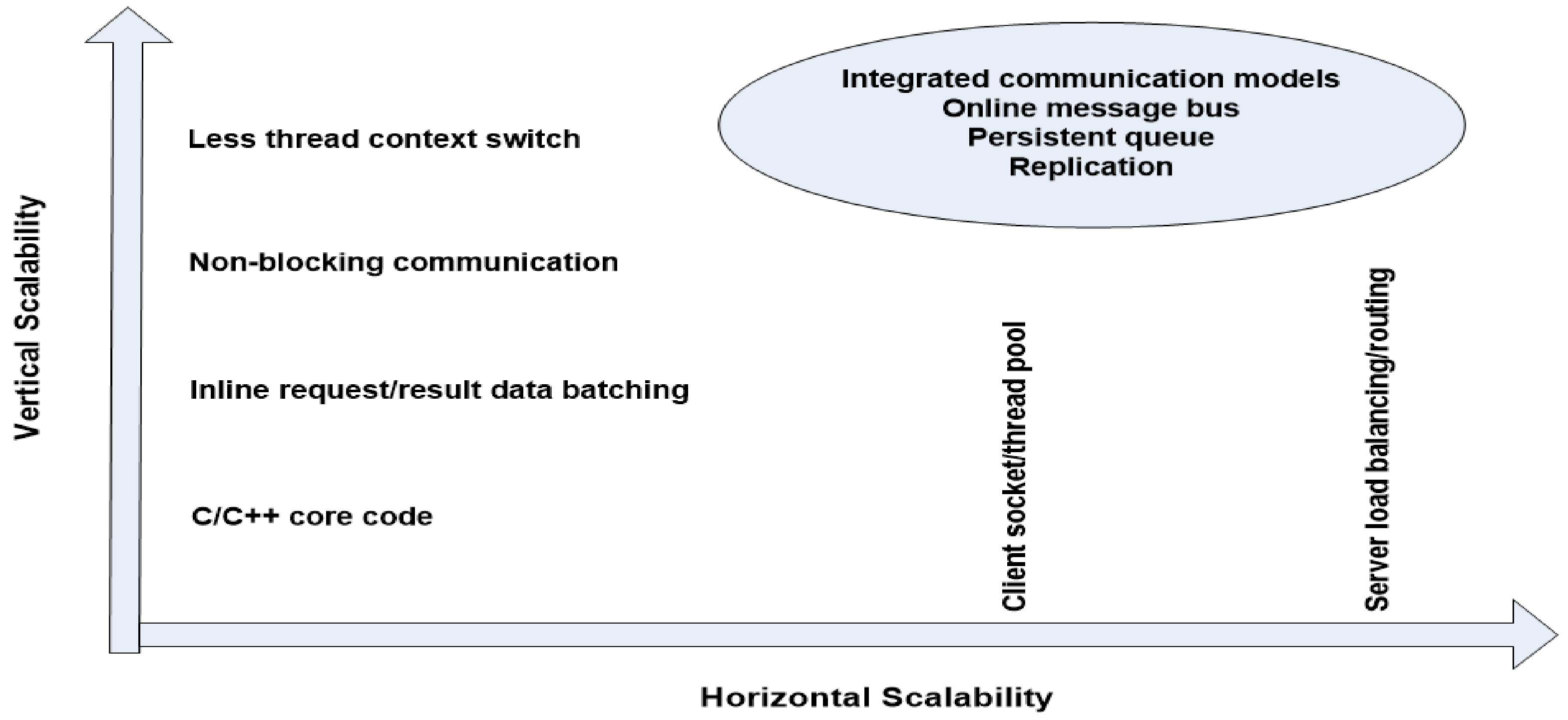

Vertical Scalability
Less thread context switch
Non-blocking communication
Inline request/result data batching
C/C++ core code
Integrated communication models
Online message bus
Persistent queue
Replication
Client socket/thread pool
Server load balancing/routing
Horizontal Scalability

Figure 1: SocketPro ways for better responsiveness and scalability of any types of applications

Vertical scalability: SocketPro adapters are created for all supported development languages, but its two core components, usocket and uservercore libraries, are written with direct use of C/C++ and raw system socket APIs, and developed over many years of experience in non-blocking communication just like many top quality software applications do. SocketPro does not have many dependency layers, which may also help application performance.

Furthermore, requests and returning results can be easily batched either manually or by internal batching algorithm as long as you use non-blocking communication. Request batching improves not only network efficiency but also processing speed at both client and server sides as shown in previous tutorials. Request batching works very efficiently especially on low-bandwidth and high-latency networks such as WIFI, DSL/Cable modems, satellite and remote communication.

SocketPro always uses non-blocking communication, although SocketPro does support synchronous requests indirectly by use of method *WaitAll*. Non-bolocking communication enables client and server computations to happen concurrently, which is also beneficial to application performance if there are many requests queued.

We measured thread-context switches with tools on window platforms. It is found that SocketPro shows much fewer thread context switches than MS WCF framework with a given number of requests.

Horizontal scalability: SocketPro has a number of features to ascertain that you can easily make your application scalable horizontally.

As depicted in the previous three tutorial samples, SocketPro uses socket pool at client side to build a set of socket connections to the same or different servers. We are going to asynchronously retrieve ADO.NET objects in parallel on master-slave architecture as shown in the below Figure 2.

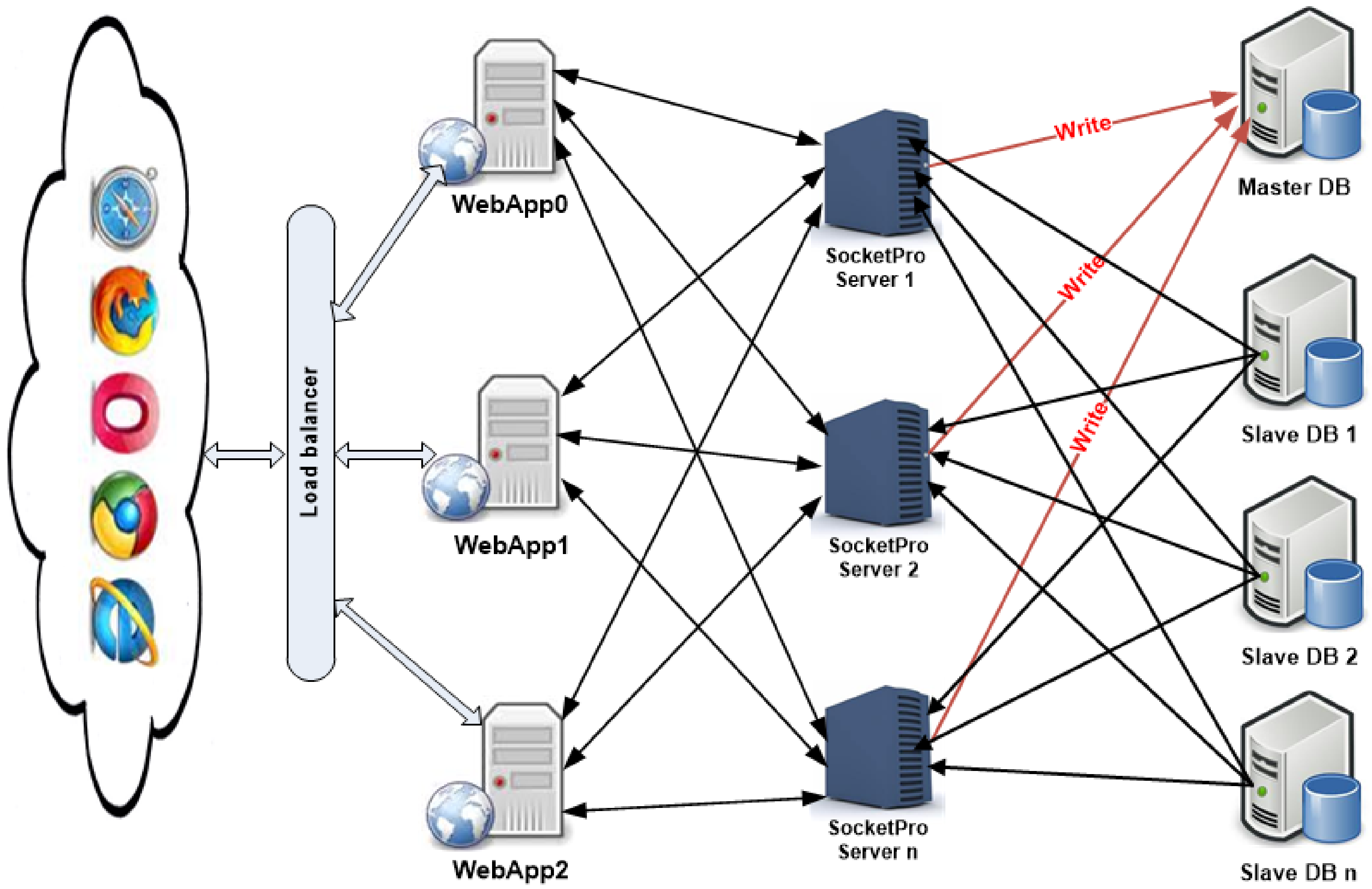


Figure 2: An architecture for resilient, responsive and scalable web application with socket pool and master-slave databases

In addition to socket pool at client side, you can also use SocketPro's server routing feature for load balancing. This feature was demonstrated by [tutorial 4: *Treat SocketPro server as router for load balancing*].

Additionally, SocketPro server also supports HTTP and websocket protocols. You can use this feature to enable communication between SocketPro and any devices as well as development environments through either HTTP or/and new web socket protocol. We used tutorial 6: *Access SocketPro by use of HTTP and WebSocket* to show you its procedures on web browser by JavaScript.

Caching, collaboration and real-time update: Assuming you like to use caches at WebApp0 through WebApp2 and SocketPro server 0 through SocketPro server n to avoid cross-machine trips as much as possible, you may need all of the caches to be collaborated and updated in real-time fashion from databases or else. You can do so through SocketPro built-in chat service or/and persistent message queue.

In regards to SocketPro built-in chat service, please refer to tutorial 2: *SocketPro secure communication and built-in bi-directional message pushes*.

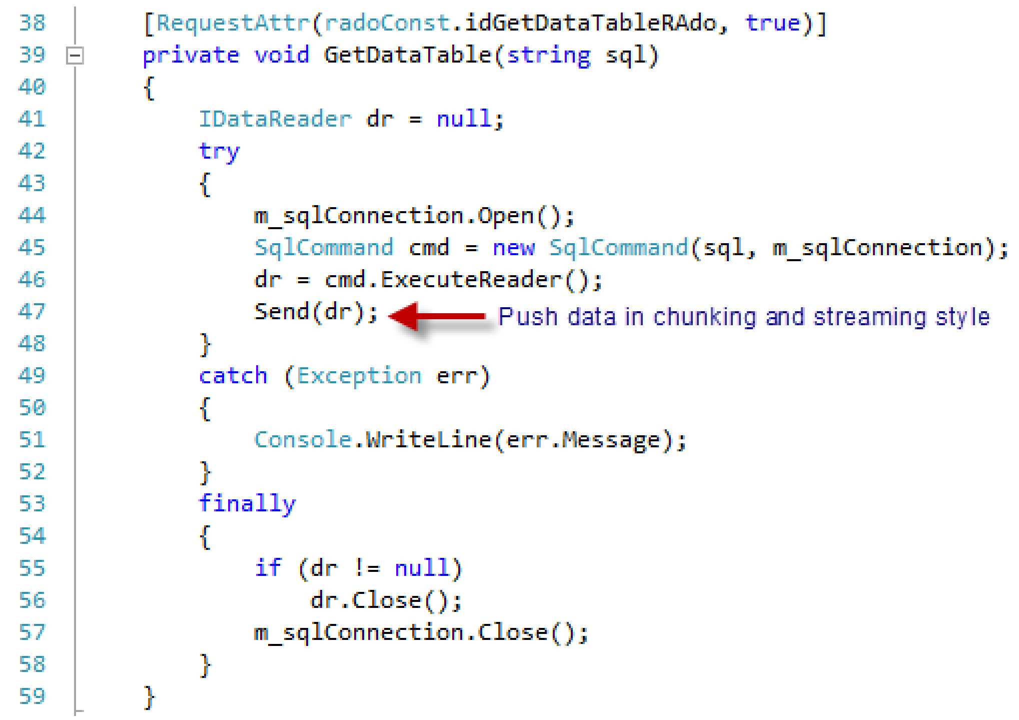
In regards to SocketPro persistent message queue, please refer to tutorial 1: *Hello world for a simple client/server application* and tutorial 3.

In addition to the two features above, you can use SocketPro's replication feature to safely execute any changes in databases or to these caches in transaction style as exemplified in tutorial 5: *Data replication by SocketPro persistent message queue*.

3. Push any sizes of ADO.NET objects in chunk and streaming style from server to client

Now, let's witness how codes work with the method Send at server side on the architecture of the Figure 2.

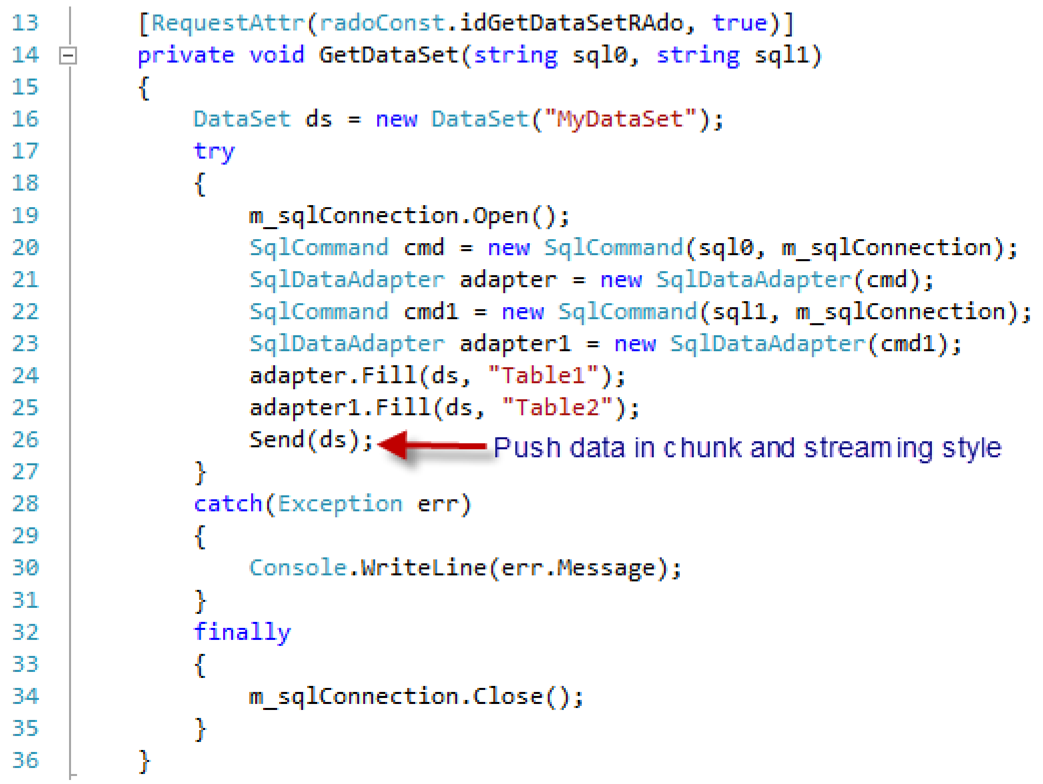
DataReader: See the code snippet below in Figure 3 below. At line 47 in the code snippet, SocketPro provides a method Send to push all record data within an ADO.NET data reader from server to client. Internally, SocketPro pushes data in chunk and streaming style; each of chunks has a size about 10 kilobytes or more so that client and server are able to unpack and pack data respectively, concurrently and in parallel at the same time. Note that there it is unnecessary to pre-load all records in memory. All these unique features make your application much faster in comparison to MS WCF framework. Our performance tests show that SocketPro will be three times faster than MS WCF framework in general. If a data reader has many records with large data size, SocketPro will be ten or more times faster!



```
[RequestAttr(radoConst.idGetDataTableRAdo, true)]
private void GetDataTable(string sql)
{
    IDataReader dr = null;
    try
    {
        m_sqlConnection.Open();
        SqlCommand cmd = new SqlCommand(sql, m_sqlConnection);
        dr = cmd.ExecuteReader();
        Send(dr);
    }
    catch (Exception err)
    {
        Console.WriteLine(err.Message);
    }
    finally
    {
        if (dr != null)
            dr.Close();
        m_sqlConnection.Close();
    }
}
```

Figure 3: Push all records of data within DataReader from server to client in chunk and streaming style

DataSet: You can push a *DataSet* to a client in chunk and streaming style too as shown in the below Figure 4. Code snippet has already remarked on how simple it is to use.



```
[RequestAttr(radoConst.idGetDataSetRAdo, true)]
private void GetDataSet(string sql0, string sql1)
{
    DataSet ds = new DataSet("MyDataSet");
    try
    {
        m_sqlConnection.Open();
        SqlCommand cmd = new SqlCommand(sql0, m_sqlConnection);
        SqlDataAdapter adapter = new SqlDataAdapter(cmd);
        SqlCommand cmd1 = new SqlCommand(sql1, m_sqlConnection);
        SqlDataAdapter adapter1 = new SqlDataAdapter(cmd1);
        adapter.Fill(ds, "Table1");
        adapter1.Fill(ds, "Table2");
        Send(ds);
    }
    catch(Exception err)
    {
        Console.WriteLine(err.Message);
    }
    finally
    {
        m_sqlConnection.Close();
    }
}
```

Figure 4: Push all data within DataSet from server to client in chunk and streaming style

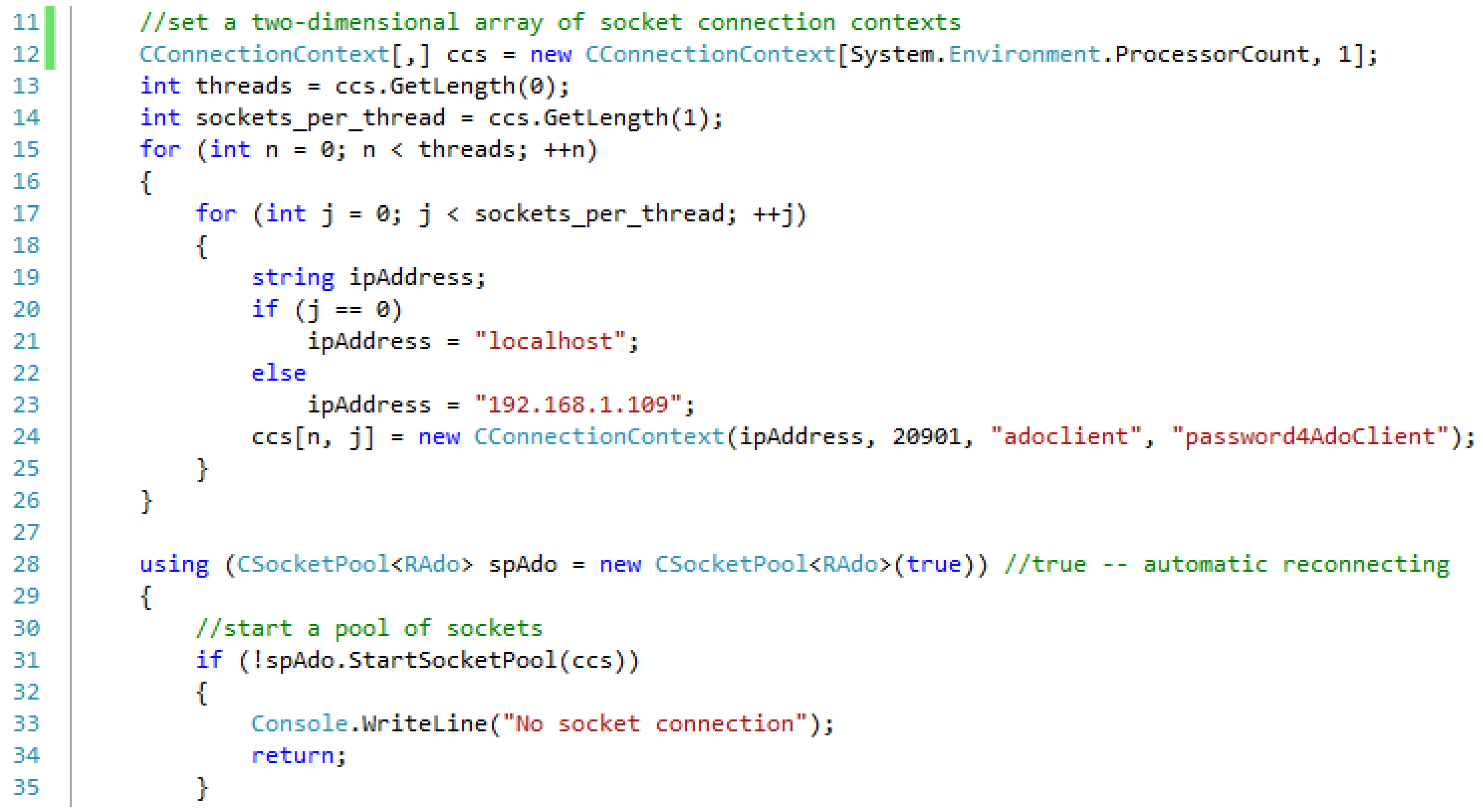
DataTable: Similarly, you can push all records within *DataTable* from server to client as you can see, even though we don't, provide the entire code snippet in this figure.

Overall, we recommend you use *DataReader* approach instead of *DataSet* and *DataTable* if proper, because use of *DataReader* avoids pre-loading all records in memory, and reduces memory footprint if your application has to deal with many records.

4. Sync and async ways to get ADO.NET objects from remote server

You can obtain ADO.NET objects either synchronously or asynchronously. If you have to get multiple ADO.NET objects, you can query them from server to client asynchronously and in parallel through different socket connections. We are going to demonstrate to you by code snippet at the conclusion of this tutorial. However, let's show how to establish a pool of sockets pointing to different servers.

A pool sockets pointing to different SocketPro servers: See the below Figure 5.

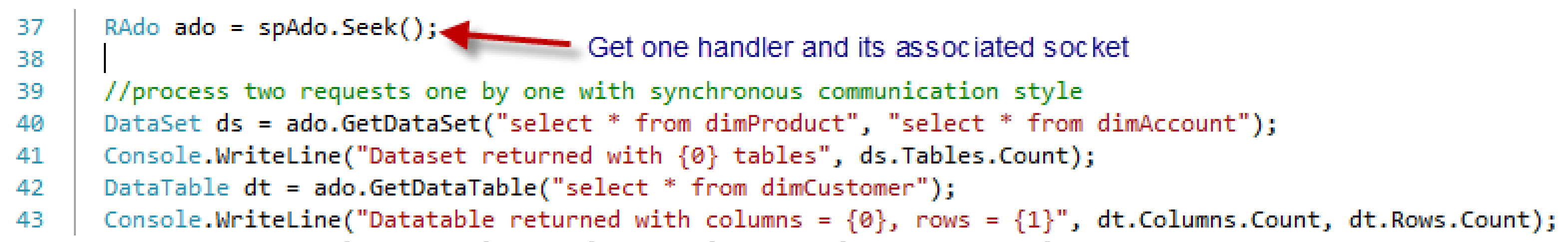


```
11      //set a two-dimensional array of socket connection contexts
12      CConnectionContext[,] ccs = new CConnectionContext[System.Environment.ProcessorCount, 1];
13      int threads = ccs.GetLength(0);
14      int sockets_per_thread = ccs.GetLength(1);
15      for (int n = 0; n < threads; ++n)
16      {
17          for (int j = 0; j < sockets_per_thread; ++j)
18          {
19              string ipAddress;
20              if (j == 0)
21                  ipAddress = "localhost";
22              else
23                  ipAddress = "192.168.1.109";
24              ccs[n, j] = new CConnectionContext(ipAddress, 20901, "adoclient", "password4AdoClient");
25          }
26      }
27
28      using (CSocketPool<RAdo> spAdo = new CSocketPool<RAdo>(true)) //true -- automatic reconnecting
29      {
30          //start a pool of sockets
31          if (!spAdo.StartSocketPool(ccs))
32          {
33              Console.WriteLine("No socket connection");
34              return;
35          }
```

Figure 5: Prepare a two-dimensional array of connection contexts for starting a pool sockets pointing to different servers

First of all, we use the codes in lines 12 through 26 to prepare a two-dimensional array of connection contexts pointing to different servers. Afterwards, we start a pool of sockets in line 31.

Sync: After establishing a pool of socket connections, we can use the following code to get ADO.NET objects synchronously one-by-one using one socket connection.

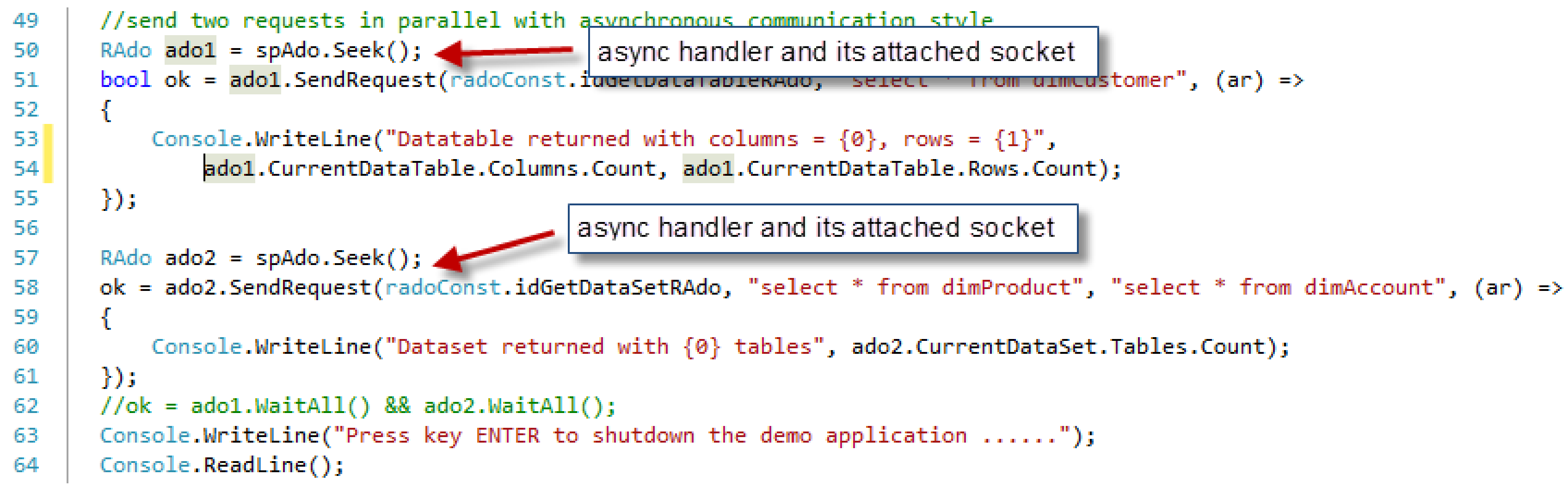


```
37  RAdo ado = spAdo.Seek();
38  
39  //process two requests one by one with synchronous communication style
40  DataSet ds = ado.GetDataSet("select * from dimProduct", "select * from dimAccount");
41  Console.WriteLine("Dataset returned with {0} tables", ds.Tables.Count);
42  DataTable dt = ado.GetDataTable("select * from dimCustomer");
43  Console.WriteLine("Datatable returned with columns = {0}, rows = {1}", dt.Columns.Count, dt.Rows.Count);
```

Figure 6: Get ADO.NET objects synchronously in one-by-one style using one socket connection

As shown in Figure 6 above, we search for an asynchronous handler from a pool first. Afterwards, we send one request in line 40 and another in line 42 to get *DataSet* and *DataTable* synchronously using the same socket connection, respectively.

Async, parallel and wait: The following code snippet is preferred over the one above in case you have multiple requests for ADO.NET objects or others.


async handler and its attached socket
async handler and its attached socket

```
//send two requests in parallel with asynchronous communication style
RAdo ado1 = spAdo.Seek();
bool ok = ado1.SendRequest(radoConst.idGetDataTableRAdo, "select * from dimCustomer", (ar) =>
{
    Console.WriteLine("Datatable returned with columns = {0}, rows = {1}",
        ado1.CurrentDataTable.Columns.Count, ado1.CurrentDataTable.Rows.Count);
});

RAdo ado2 = spAdo.Seek();
ok = ado2.SendRequest(radoConst.idGetDataSetRAdo, "select * from dimProduct", "select * from dimAccount", (ar) =>
{
    Console.WriteLine("Dataset returned with {0} tables", ado2.CurrentDataSet.Tables.Count);
});
//ok = ado1.WaitAll() && ado2.WaitAll();
Console.WriteLine("Press key ENTER to shutdown the demo application ......");
Console.ReadLine();
```

Figure 7: *Fetch ADO.NET objects from server asynchronously and in parallel by use of multiple socket connections*

We get two asynchronous handlers and their socket connections in lines 50 and 57. Afterwards, we send two asynchronous requests in lines 51 and 58 for *DataTable* and *DataSet* objects, both respectively in parallel. Note that two socket connections may point to either the same or different servers, which fully take advantages of multiple SocketPro servers and multiple master-slave databases as shown in the previous architecture of Figure 2.

Finally, it is optional if you choose to use the method *WaitAll* to put a barrier as shown in line 62. The above approach can be easily coupled with .NET asynchronous task as shown previously in tutorial one.