Cross-platform Communication Using Custom Channel Sinks

RNDr. Tibor Hrnko
Vero Partners, s.r.o.
Italská 13
120 00, Prague, Czech Republic
Thrnko@veropartners.com

Mgr. Jan Kašpar
Vero Partners, s.r.o.
Italská 13
120 00, Prague, Czech Republic
Jkaspar@veropartners.com

ABSTRACT

The paper discusses the design of a distributed application that consists of a connection to independent provider of message transport and logic based on text messages. It is scalable, extendable and its communication is based on remote method calls. Due to the dominance of WIN32 based systems on the computer market and due to the remarkable extension of the possibilities of its development environment provided by introduction of the .NET platform, this particular environment is very suitable for the creation of an effective solution. C# language is appropriate for easy code porting to other object oriented standards (e.g. J2EE).

The general data flow of the solution can be described in four main steps:

- The application logic receives and writes data asynchronously using application queues. Queue messages are serialized application specific objects. Scalability is implemented by the possibility of multiple application instances.
- An arbitrary number of connectors to service providers can access the queues picking only the appropriate messages. The object model on the connector level is uniform with regard to application objects.
- Method calls of the connectors are translated into individual data streams crossing the domain boundaries using the custom channel sink mechanism of the .NET platform. The other end of the communication can be totally independent from the originating platform.
- Arbitrary specialized functionality can be introduced into individual connectors through the insertion of specialized sinks. Unrestricted manipulation of the received data on the application level can be introduced by the modification of the application queue data.

Keywords

WIN32; .NET; C#; Cross-platform; GSM; SMS; Remoting; Application queues; Custom sinks

APPLICATION DESCRIPTION

SMS application platform

The goal of application platform creation was to prepare a base for text oriented games with an arbitrary connection to transport service provider. This specification covers a wide range of trivia type quiz games, chats and information services. Individual applications should differ only in their specialized functionality. Common communication object at the application level was defined as SMS message with the properties specified by ETSI standards [ETSI SMS]:

- SmsDateTime
- SmsMessage
- SmsMsgStatus
- SmsPhoneBookEntry
- SmsReferenceId
- SmsSenderId
- SmsTimeZone

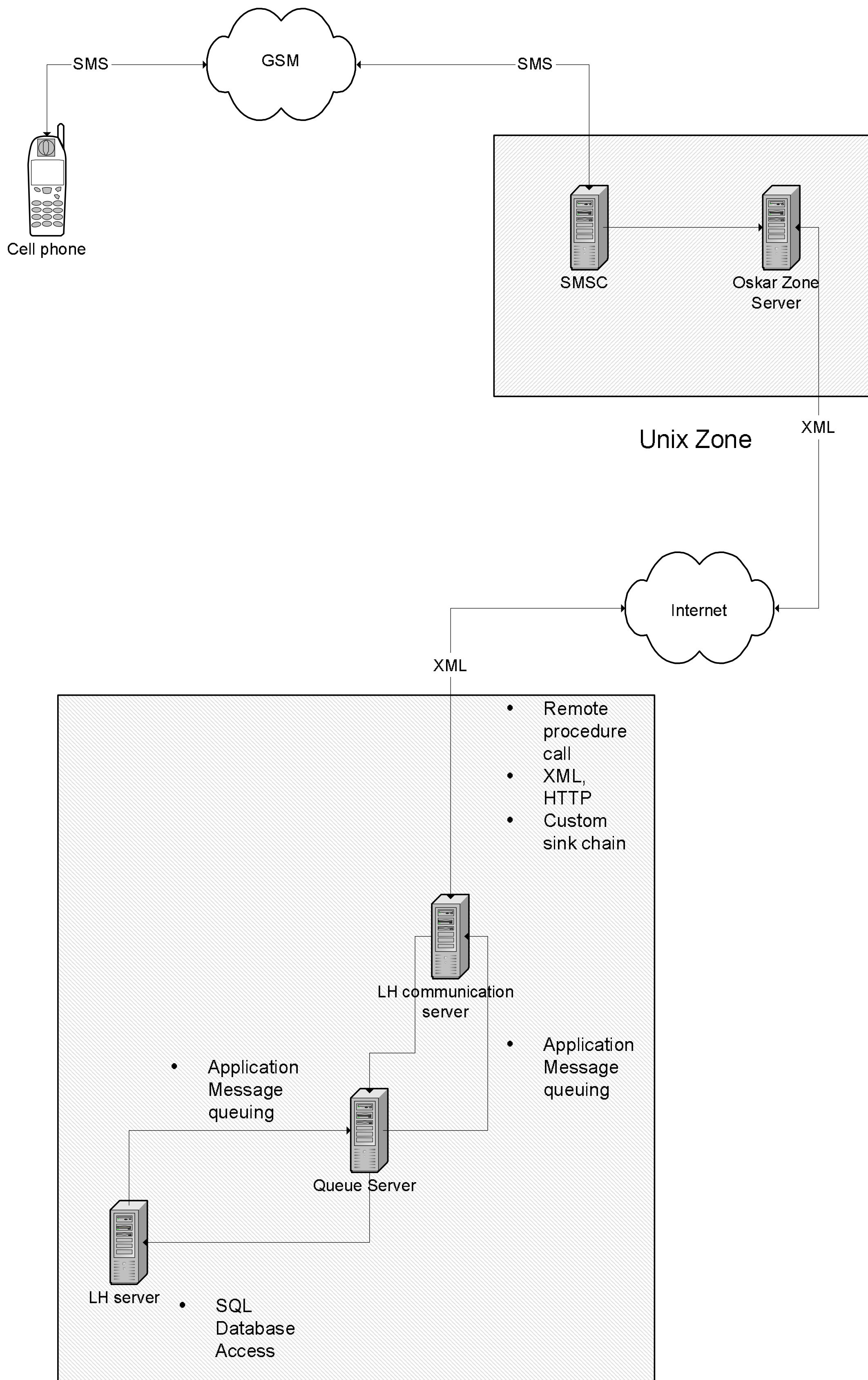
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1st Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing

February 6-8, 2003, Plzen, Czech Republic.

Copyright UNION Agency – Science Press

ISBN 80-903100-3-6



Picture 1 General View of the Application

other properties are general, concerned with transferred information status and reference.

Any platform application receives objects of this type from application queues. The objects are pushed to a queue by individual connectors.

Connection to a service provider

This part of the system should provide connection to arbitrary individual protocols on the provider side, create a common SMS object and store it to proper application queue on the application side.

Applying widespread approach of writing specialized communication for each provider would mean an object mess, as the objects of those providers are as far appart from each other as are the individual providers' approaches. It would mean a lot of overhead in maintaining this code too.

Channel based connections have much more common code and provide an extra feature of masking the connection complexity behind single method call. This approach is detailed in section 3.

1. USE OF APPLICATION QUEUES

Scalability Issues

Generaly accepted rule of today's application is to keep interprocess communication down to minimum. This is argumented by overhead imposed by serialization and deserialization of the objects, transported between the processes. It may be wise approach for computation intensive processes, but we consider it not so well founded in communication between high level objects transferring messages that contain a lot of computational effort in a command that contains limited data amount or is not time critical. Typical example of such message is a communication of an application with human user.

In-process manipulation of objects is generically quick but self-contained. Inter-process manipulation tends to be slower, but is distributed by its nature. It provides for both the scalability introduced by distributed computation and the implementation of functionality provided by any computer with access rights for the queue.

Up Time And Application Monitoring

We take it for granted, that there is no complex application that is uncrashable. Commercial applications with a short lifetime are even more prone to this problem, as they tend to be created quickly.

Use of queue lessens this problem by providing easy means of monitoring the queue state and staling of the messages that are stored. It can be achieved on the system level by setting the properties of the queue or by a code monitoring the arrival time of the queue messages. The monitoring code can even dynamically spawn new instances of the code according to the queue length.

Exception handling does not have to be so bullet proof; as the crush does not mean necesarily the loss of arrived data and new instances of the same functionality can work well with the rest of the queue messages. It is necessary only to spawn the new process by an independent agent.

Figure 1 is a code snippet that checks inbound queue and sends messages to application users notifying them about the service problems.

Security Considerations

Queued solution is easy to implement on computers located behind a firewall, but separated from the rest of the domain. Individual connectors should have only a limited access to the queue server. Thus the malicious attack chances are rather reduced with respect to the internal workstations and servers.

Strict typing of queue data almost excludes possibility of running outsider code as all the messages are deserialized by individual data consumers. That means that even inherently insecure practice of calling selects into database by dynamical construction of the clause is separated from the arrived data and is invisible from outside world.

Independent Data Storage

It is possible to store all data in a queue itself or use more sophisticated mechanisms like SQL database. This choice depends solely on the application needs.

Let us pinpoint, that queuing concept incorporates even transactional processing and referencing of the managed messages. C# objects implement IEnumerable interface so implementation of in memory indices and sorts is straightforward.

2. CHANNEL AND SINK DEFFINITIONS

Predefined .NET Channels

There are two channel types shipped with .NET. Http channel formats its messages according to the SOAP standard. TCP channel formats them into binary messages. The later is useful for remoting based on .NET framework applications sitting on both sides of the communication channel. The former is convenient for writing applications using web services. Further information on the channel

```
// Defined outside the presented code
// System.Messaging.MessageQueue inQueue;
// System.Messaging.XmlMessageFormatter ftr
// System.MarshalByRefObject marshallObj
// SMSClass and Ucp51 proprietary classes
// used for the connection to Oskar
private void checkInQueueLh()
{
  System.Messaging.Message m;
  SmsGame.SMSClass lhm;

  while(isProgramRunning())
  {
    Thread.Sleep(5000); //milliseconds
    IEnumerator e=inQueue.GetEnumerator();
    while (e.MoveNext())
    {
      try
      {
       m = ((Message)e.Current);
       if((isTooOld(m))&&(isNotDelayMsg(m))
       {
         m.Formatter = ftr;
         lhm = new SMSClass(m);
         Ucp.Ucp51 u = new Ucp.Ucp51();
         u.to = lhm.SmsSenderId;
         u.message=MESSAGE_STILL_QUEUE;
         marsahllObj.sendUCP(u));
         m = inQueue.ReceiveById(m.Id);
         inQueue.Send(m,LH_DELAYED_MSG);
       }
      catch(Exception e1)
      {
       Debug.WriteLn (e1.StackTrace());
      }
    }
  }
}
```

Figure 1 Monitoring inboud queue

concept can be found in a number of books. We have found [Advanced Remoting] a very instructive one.

Channel processing

Channel starts processing once a method call is invoked on object that is a local proxy for a remote object. First sink in a chain of client side sinks serializes data of a local object, rest of the sinks adds its own data, but can not change the stream created by formatter sink. Last sink on the client side, called a transport sink, generates appropriate transport streams and sends data to a remote server. Server side sinks are responsible for recreation of the original object, execution of its functionality and returning result to the caller.

Sink chains are arbitrary on both the client and the server side. Any user sink can be inserted into it at any point, if it adheres to the basic rules of the chain:

- Any object change is done prior to calling formatter sink method. Changes to object properties are not propagated to the server if made in later stages of the processing
- Formatter sink has to serializes object data to an existing stream or create a new one if none is available
- No client side sink called after formatter sink changes serialized properties.
- Server side sink is responsible for deserialization of the provided data.

In common language the implictions are these:

- Data encryption or any other object modification prior to call has to be done before serialization takes place.
- When implementing a custom client side sink, custom server side sink is likely to be needed.

Creation of Custom Sinks

The task of creation the custom sinks is presented by examples. Code is based on Vero Partners company connection to Český Mobil GSM provider. Servers on the operator side run a JSP application on UNIX servers, accepting custom XML stream, defined by their IT division. Servers on Vero side run .NET channel application. The Figure 2 is a simplified code for client side formatter sink. The Figure 3 is a simplified code for server side sink.

3. EXCEPTIONS, AND TIMEOUTS

Fortunately enough, there are not too many beasts hidden in the implementation apart from understanding the channel mechanism. Exception handling can be simply expressed by the fact, that no sink could throw an exception. Easiest way to achieve this is to bracket all custom sink code into try catch block. Timeouts are slightly more difficult because they are not described in the documentation. But once you know that it is only necessary to change properties of the transport sink (provided by the framework for commonly used http and TCP channels), it is quite straightforward. The simple code for a custom channel sink that is a predecessor of the transport sink is:

```
_nextChanelSink.Properties["timeout"]=10000;
```

4. CONCLUSIONS

Channel sinks are suitable for handling inter process communication. They are flexible and offer seamless integration into object model of an application. Use of application queues is a good way to introduce scalability, monitoring and safety features with a minimal amount of programming effort.

5. REFERENCES

[1] [ETSI SMS] ETSI TS 127 005 v4.0.0 (2001-2003)

[2] [Advanced Remoting] Ingo Ramirez, Advanced .NET Remoting, Apress 2002, ISBN (pbk): 1-59059-025-2

```
namespace XmlConnector
{
  public class OskarClientFormatterSink : IClientFormatterSink
  {
    ListDictionary _properties = new ListDictionary();
    IClientChannelSink _nextChanelSink;
    Stream _stream;
    IMessageSink _nextSink = null;
    ITransportHeaders _responseHeaders;
    Stream _responseStream;

    public OskarClientFormatterSink(
      IClientChannelSink nextSink)
    {
      _nextChanelSink = nextSink;
    }
    public IDictionary Properties
    {
      get{return _properties;}
    }
    public IClientChannelSink NextChannelSink
    {
      get{return _nextChanelSink;}
    }
    public void AsyncProcessRequest(
      IClientChannelSinkStack sinkStack,
      IMessage msg,
      ITransportHeaders headers,
      Stream stream)
    {
      //todo: add async code here
    }
    public void AsyncProcessResponse(
      IClientResponseChannelSinkStack sinkStack,
      object state,
      ITransportHeaders headers,
      Stream stream)
    {
      //todo: add async code here
    }
    public Stream GetRequestStream(
      IMessage msg,
      ITransportHeaders headers)
    {
      Stream s;
      if (_stream == null)
      {
        if ((s = NextChannelSink.GetRequestStream(msg,headers))==null)
          return _stream = new MemoryStream();//no stream from predecessor sink
        else
          return s;
      }
      else
        return _stream; //subsequent calls return stream set by first call
    }
    public void ProcessMessage(
      IMessage msg,
      ITransportHeaders requestHeaders,
      Stream requestStream,
      out ITransportHeaders responseHeaders,
      out Stream responseStream)
    {
      responseHeaders = requestHeaders;//No modification during message processing
      responseStream = requestStream;
    }
public IMessageSink NextSink
    {
      get
      {
        return _nextSink;
      }
    }
```

Figure 2 Custom client side formatter sink

```
public IMessageCtrl AsyncProcessMessage(
      IMessage msg, IMessageSink replySink)
    {
      return null; //todo: add async code here
    }

    public IMessage SyncProcessMessage(
      IMessage msg)
    {

      try
      {
        TransportHeaders requestHeaders = new TransportHeaders();//New transport headers
        IMethodCallMessage m = (IMethodCallMessage) msg;//cast msg to what it actually is
        Stream stream = GetRequestStream(msg, requestHeaders);//obtain a request stream
        // Handling of application specific data
        if (m.InArgs[0].GetType() == typeof(Ucp.Ucp51))
        {
          ((Ucp.Ucp51)m.InArgs[0]).WriteXml(stream);
        }
        // \Handling of application specific data

        //Forward for processing using ProcessMessage. Never TimesOut
        NextChannelSink.ProcessMessage(
          msg, requestHeaders, stream,
          out _responseHeaders,
          out _responseStream);

        //clean-up stuff
        stream.Close();
        _stream = null;
        byte[] buf = new byte[1000];
        _responseStream.Read(buf,0,1000);
        _responseStream.Close();
        string inputStr = new System.Text.ASCIIEncoding().GetString(buf);
        string resultStr;
        if (inputStr.IndexOf("result=\"ACK\"")>=0)
          resultStr = "ACK";
        else
        {
          resultStr = "NACK";
          EventLog eventLog = new EventLog();
          eventLog.Source = "XMLConnector";
          eventLog.WriteEntry("NACK for message");
        }
        // \handling of debug messages and event logs
        return new ReturnMessage(resultStr,null,0,null,m);
      }
      catch (Exception e)// No exception possible in channel processing
      {
        Debug.WriteLine("Exception in syncProcessMessage");
        Debug.WriteLine(e.Message);
        try
        {
          _stream.Close();
        }
        catch(Exception){}
        _stream = null;
        return new ReturnMessage("ERROR",null,0,null,null);
      }
    }
}
```

Figure 2 Custom client side formatter sink (continued)

```
namespace XmlConnector
{
  public class OskarServerFormatterSink:IServerChannelSink
  {
    private const int OK = 1;
    private const int ERR = -1;
    private ListDictionary _properties = new ListDictionary();
    private IServerChannelSink _nextChanelSink;

    public OskarServerFormatterSink(IServerChannelSink nextSink)
    {
      _nextChanelSink = nextSink;
    }
    public IDictionary Properties
    {
      get{return _properties;}
    }
    public IServerChannelSink NextChannelSink
    {
      get{return _nextChanelSink;}
    }
    public void AsyncProcessResponse(
      IServerResponseChannelSinkStack sinkStack,
      object state,
      IMessage msg,
      ITransportHeaders headers, Stream stream)
    {//add async handling here
    }
    public Stream GetResponseStream(
      IServerResponseChannelSinkStack sinkStack,
      object state,
      IMessage msg,
      ITransportHeaders headers)
    {//should test sink chain !
      return null;
    }
    public ServerProcessing ProcessMessage(
      IServerChannelSinkStack sinkStack,
      IMessage requestMsg,
      ITransportHeaders requestHeaders,
      Stream requestStream,
      out IMessage responseMsg,
      out ITransportHeaders responseHeaders,
      out Stream responseStream)
    {
      Object result;
      responseMsg = null;
      responseHeaders = new TransportHeaders();
      responseStream = new MemoryStream();
      result = new Object();
      try
      {
        switch (validateInput(requestStream))
        {
          case ERR: //invalid should handle proper response
            Debug.WriteLine("NACK for batch");break;
          case OK://valid any useful code for validated input here
            Debug.WriteLine("ACK for batch ");break;
        }
        responseMsg = new ReturnMessage(result, null, 0, null, null);
      }
      catch(Exception e)
      {
      }
      return ServerProcessing.Complete;
    }
    private int validateInput(Stream stream)
    {
      return OK//any validation goes here OK, ERR
    }
}
```

Figure 3 Custom server side sink