IMPLEMENTING LARGE DATASET SYNCHRONIZATION ON RESOURCE-LIMITED MOBILE TERMINALS

Constantin Pistol

Computer Dept., Faculty of Automation, Computers and Electronics, University of Craiova
Str. Lapus Nr. 5, Craiova, Romania
costi@xts.ro

Abstract: Today's automation systems are shifting more and more toward small, portable, mobile terminals that provide users a full interface on large Enterprise Resource Planners or database systems. With more and more flexibility and ease of use being incorporated in such terminals, the complexity and amount of data that is transferred also increased significantly. Having wireless terminals that are capable of working both online and offline, the fast synchronization of large amounts of data has become an issue. The terminals (like PocketPCs) are considerably resource restricted, and so memory space and CPU cycles have to be carefully optimized in order to achieve usable (fast) systems. The paper presents a combination of custom XML parser and PocketPC local database access that allows for fast synchronization (table duplication) between PocketPC terminals and external (PC) data sources.

Keywords: Mobile, PDA, Syncronization,XML

INTRODUCTION

We will consider in this paper only the aspects regarding the implementation of a fast data handling / data synchronization on the PocketPC terminal. The communication and PC server aspects are not detailed.

A PocketPC terminal (or PDA – Personal Digital Assistant) is basically a hand-held personal computer, running a stripped-down version of Microsoft Windows , namely Windows CE. A typical computer of this type has a hardware configuration similar to the following :

CPU : Strong ARM 206 MHz

Memory : 64 MB RAM, 32MB ROM

Display : 320x240 pixel, 64.000 colors, touch sensitive

Data Comm : Standards-based Spectrum24 IEEE 802.11 wireless adapter (data rate 2 to 11 Mbit)

No hard-drives are supported, mainly due to size restrictions, but Flash Cards are available for expanding the available memory space.

It should be noted that a 200 MHz Strong ARM CPU can not be directly compared (speed-wise) to a standard 200 MHz desktop CPU (like an Intel Pentium). The performance of the former is at least an order of magnitude lower than that of the latter, in spite of having the same clock rate.

We'll consider that data is transmitted in the form of eXtended Markup Language (XML). We can define a markup language as a set of rules which impose a syntax used to define, delimit, and describe text or data within a document through the use of tags

(literal strings). Under the above definition, XML is a markup language that is used to convey metadata (information about data). The language depicts the structure as well as the meaning of data contained within an XML document. However, XML is not concerned with formatting but with structure and semantics. It is a simple, standard way to delimit text data in a self-describing way.

A sample of XML formatted data :

```
<item>
        <name>IR Sensor</name>
        <code>12355</code>
        <availability>Out of stock</availability>
</item>
```

XML imposed as a widely used standard mostly because it is non-proprietary (open) and easy to read and write by both humans and computers.

It proved an excellent format for the interchange of both data among different applications and computing platforms (platform independent). Another plus is the self-describing nature of XML data that makes it an effective choice in the field of distributed applications. XML allows for easy transfer of data over standard Internet protocols, such as HTTP (making it firewall friendly). It is also extensible and supported on virtually every platform, being usable basically from any programming language.

For the actual communication, an XML based protocol like Server Object Access Protocol (SOAP) can be used (although it is not mandatory). Simple Object Access Protocol (SOAP) is a lightweight, extensible protocol based on XML designed for facilitating information exchange in decentralized, distributed environments. Most importantly, SOAP defines rules for message structuring, as well as a message processing model. A convention for making remote procedure calls together with a set of encoding rules for serializing data are also defined. SOAP provides the base for many modules and protocols running over a multitude of underlying protocols (HTTP being the most common).

MAIN ACTIONS

Given these aspects, we can encapsulate (from a logical standpoint) remote communication in a function with the prototype:

XMLDataType GetRemoteData (CmdType cmd)

Given the XML containing the data (a list of records), there are two steps that need to be taken in order to have the data inside a local PocketPC database:

- *parsing* the XML and extracting the data records
- *inserting* each record in the corresponding table of a local database

Parsing

For the the communication layer the freely available library PocketSOAP (www.pocketsoap.com) was used. It presents itself as an open source SOAP client PocketPC COM component, with a straight forward, easy to use interface for SOAP based communication. The package includes a HTTP 1.1 transport for making HTTP based SOAP requests.

Although the PocketSOAP library provides a serializer/deserializer that could perform the parsing functionality required, it was ruled out because of speed/resource limitations. PocketSOAP's built-in XML parser creates an in-memory tree object model of the received XML. In our target applications, the volume of data is very high , with XML files in the order of megabytes. Under these conditions, the built-in parser fails to do it's job because of out-of-memory errors (because total device memory is only 64 MB, with roughly half of it usually available).

Given the situation, a custom XML parser was implemented. Light-weight and fast, it uses an optimized SAX-style parsing, allowing for minimal memory footprint (a few KB more than the XML file itself).

The core of the parser is the Boyer-Moore algorithm, one of a larger family of algorithms designed for performing exact string searches, approximate string searches, 'sounds-like' string searches, and other types of textual comparisons.

This parser (we'll call it DAX Parser) goes through the XML sequentially, creating directly a list of records without the intermediary tree representation.

But, even this can be further optimized (in some particular cases). The cases in which this is true is when large amounts of XML are expected, and the processing can be done at a record-by-record level. Synchronizing a local database with a remote database is such a data intensive, repetitive task.

Let's see how will DAX Parser be used in such a case, first in the standard way, and then using an optimized interface to the parser.

Standard use

We will consider that 20.000 records must be retrieved from a remote tabel and inserted in a local one. The data retrieval function to be used in this case is (pseudocode) :

```
List<RecordType>  GetData( CommandType cmd) {
        Xml = GetRemoteData(cmd);
        List = Parse(Xml);
        Return List;
}
```

This function will get called with the corresponding select-type SQL query as parameter. After the data is received as XML (we'll consider it to be about 5 MB), it is internally parsed by DAX Parser and a List<RecordType> is built and returned. After GetData() has returned, the application will start looping through the returned list, inserting each record in the local database. The steps of this action (and their corresponding memory use) can be represented in the following manner :

Step	Action	Max Memory
1	Call to GetData	0 MB
2	XML received	XML : 5 MB
3	Parsing and list building	XML,part. list : 15MB max
4	GetData() returns list	Record list : 10 MB
5	Loop through list and insert each record	Record list : 10 MB

Fig.1 Standard parser call and insertion steps

Memory use peaks at 15MB, and it takes two full loops through the data records until they get inserted into the local database (at step 3 and 5).

Optimizing with callbacks

The optimization implemented for such scenarios relays on the so-called callback functions mechanism. We modify the parser (and GetData()) to accepts as parameter a pointer to a function – a function that will be called to handle each XML record as it is parsed. No list is built in this case.

The handler must have a prototype like the following:

Bool HandleRecord(RecordType rec);

A pointer to such a function is passed to the GetData() call as an optional parameter :

List<RecordType> GetData(CommandType cmd, PHANDLE pHandleRec=NULL) ;

When pHandleRec is not NULL, an empty list is returned. However, as soon as a full record is parsed, we get a chance to process it. In the case presented, we'll consider that the HandleRecord() function inserts the record received into the local database. The steps in a GetData()call using this approach are :

Step	Action	Max Memory
1	Call to GetData	0 MB
2	XML received	XML : 5 MB
3	Loop – [1. parse one record 2. call (*pHandleRec)(rec) to insert it]	XML+1 rec ~ 5MB
4	GetData returns empty list	0 MB

Fig.2 Callback parser call and insertion steps

The gains compared to the standard call are significant :

1. Memory optimization - Peak memory use down from 15 MB to 5 MB (the size of the XML data)

2. Speed optimization - One loop through the records instead of two (removal of step 5)

The HandleRecord function returns a Boolean that allows for early parse termination. If the value returned is true then the next record is parsed , otherwise the process is halted and GetData() returns immediately. This comes in very handy for instance when all we want from the received data is a specific record that meets a certain criterion. HandleRecord() will check for the criterion and return false after it is found. This further optimizes execution time (for such cases).

Of course, the HandleRecord() can do other things than database insertion, like filling a list of the display (or something else that might need to be done). Generally speaking, it is advised in the case of large operations that are record oriented rather than list oriented. Database synchronization is a perfect match for this optimization.

LOCAL DATABASE ACCESS

Microsoft ADO CE : the problem

Microsoft offers a standard database access layer for their Windows CE (PocketPC) operating system in the form of ADO CE (ActiveX Data Objects for CE). ADO CE proved to be adequate for a small database activity, but under medium to heavy use it failed to deliver a good level of performance reliability.

When ADO CE was used for data intensive tasks such remote-to-local database synchronization, it proved to be an Achilles' heel. Memory leaks within ADO CE's code were causing major application slowdowns, and the performance itself (in terms of speed) was disappointingly low.

After (unsuccessfully) trying various alternatives of using ADO CE's functionality, a decision was made to scrap it altogether and find an alternative. Unfortunately, the only standard layer on which we can build upon in the absence of ADO CE is OLEDB – which is quite a low level layer.

The data access library: XDB

So, a new interface called XDB was created as a wrapper around OLEDB. It provides the ADO CE functionality that is needed, and it improves by adding parameterized query support and exception-based error handling. The interface XDB creates is designed along the lines of ADO.NET, with separate Connection, Data Provider, Data Reader and Command classes provided.

The boost in performance gained by switching to XDB was significant. The data-intensive synchronizations times were reduced by a factor of 2-3x. An insertion test was performed with records containing 3 strings and 1 integer in a table with no indexes defined. The test configuration was a Symbol PPT 2800 PDA, with Windows CE 3.0 , using ADO CE 3.1 and Microsoft SQL Server CE 2.0. It yielded the following results :

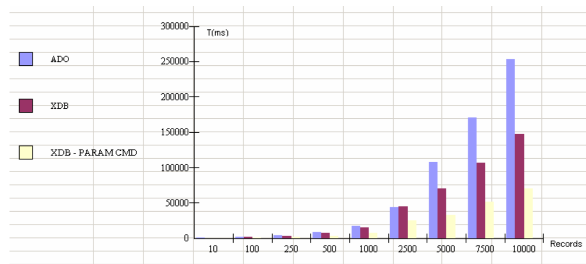

Fig.3 XDB insertion speed comparison

The yellow bars show the speed XDB has when using parameterized queries (a feature not supported by ADO CE). It basically allows to tell the database engine the query you want to use, but using parameters instead of actual data to be inserted. Of course, this query does no insertion, but allows the database to make an execution plan and store it. Afterwards, you start doing inserts by simply specifying those parameters. Because the database had already "compiled" what it needs to do to execute the insertion, things go faster, with less overhead on each insertion. All this is done at OLEDB layer, so it is as close to the database core as you can get.

Used in the HandleRecord() callback function for inserting records in the local database during a sync, the speed improvement is dramatic compared to "normal" inserts.

FINAL REMARKS

The framework was implemented and used in a Sales Force Automation (SFA) application. The picture below describes one screen from the user interface built namely the selection of the client from the local database.

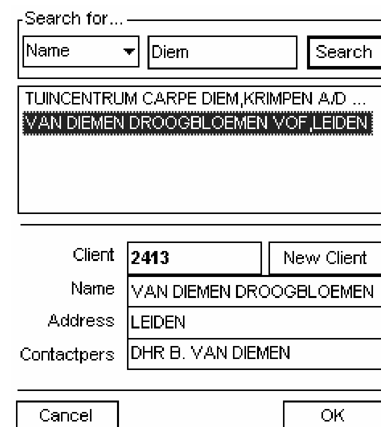

Fig. 4 GUI Example – search for a client

It delivers a complete, all-around solution, capable of wireless networking for instant information update as well as full-featured offline mode for field use. As such, it is intended to be extended and used on future, similar projects.

REFERENCES

Andrew Binstock and John Rex (1995), *Practical Algorithms for Programmers*, Addison-Wesley.

Alfred V. Aho (1990). *Algorithms for finding patterns in strings*. Handbook of Theoretical Computer Science, chapter 5, pages 254-300. Elsevier Science Publishers B. V.

James Snell, Doug Tidwell, Pavel Kulchenko (2002), *Programming Web Services with SOAP*, O'Reilly & Associates, Inc.