Shared Events Part 2: Redesign

by Primoz Gabrijelcic

Today we will return to the shared event system which I described in Issue 97. I explained the higher levels, but skimmed over the implementation details. For example, you don't know yet how shared tables (subjects, events, etc) are stored internally. I haven't told you why the system is so slow. To understand that, you must know the answer to the first question, so let's start with it.

XML Storage

When designing shared structures for the shared event system I wanted to keep it open for any future extensions. With that in mind, I decided to use the storage that is both standardised and open, the extensible markup language, or XML for short.

Each table is internally stored as a quite simple XML document. A typical example would be the Subjects table, which you can see in Figure 1 (slightly edited for readability). At the moment when the snapshot was made, the table contained two producers and one listener (for the other tables, see the implementation section of the GpSharedEventsImpl unit).

As the tables must be common to all applications in the same namespace, they are stored in shared memory blocks. The storage mechanism is simple. When saving, the document is converted into a string and stored inside this shared memory. To read it, the code parses the shared memory block and converts it back to the internal XML representation. As you may have guessed, this is not exactly the fastest operation imaginable. The shared event system does, however, implement some trickery to reduce the number of times XML document is parsed.

XML-accessing logic is encapsulated in the TGpSharedXML class (OmniXMLShared.pas). While most methods in this class are quite simple, one deserves special attention. Acquire (Listing 1), which is called whenever the XML needs to be accessed for reading or writing, first acquires the shared memory block holding the XML document and then loads (parses) that document. The result is stored in the internal variable sxXMLDoc. On the next Acquire, the underlying memory block is first acquired and then the code checks if it was modified in the meantime (the TGpSharedMemory object handles that check for us). If no modifications were made to the shared memory, there is no need to reparse the contents and Acquire can simply return the cached sxXMLDoc.

Accessing the shared XML document for reading is then fairly trivial. The Read method only calls Acquire and then immediately releases the shared memory block. After that, the XML document is available in the sxXMLDoc variable (which is also accessible through the read-only property XML). As the shared memory block is already released when Read returns, there is no need for the caller to do anything when it has completed the reading.

Modifying the XML is slightly more complicated. An application must call BeginUpdate to access the writable copy of the document and EndUpdate when it is done writing. During that time, the underlying shared memory block is locked for all access. EndUpdate is also responsible for saving the XML document into the string and storing it into the shared memory block.

For the XML implementation, I'm using OmniXML (see www.omnixml.com). It is simple to use, fast, doesn't require any external

➤ *Figure 1: XML representation of the Subjects table.*

```
<SubjectsTable>
  <Subjects>
    <Subject>
      <Handle>1</Handle>
      <Name>Producer 1</Name>
      <Token>Gp/GpToken/...
         -9EF9ACC83F2B</Token>
      <IsProducer>1</IsProducer>
      <Interest>60</Interest>
    </Subject>
    <Subject>
      <Handle>2</Handle>
      <Name>Listener 1</Name>
      <Token>Gp/GpToken/...
         -E777548EE858</Token>
      <IsProducer>0</IsProducer>
    </Subject>
    <Subject>
      <Handle>3</Handle>
      <Name>Producer 2</Name>
      <Token>Gp/GpToken/...
         -3E309EAAAFC9</Token>
      <IsProducer>1</IsProducer>
    </Subject>
  </Subjects>
</SubjectsTable>
```

➤ *Listing 1: Accessing a shared XML document.*

```
function TGpSharedXML.Acquire(forWriting: boolean;
  timeout: DWORD): TGpXMLDoc;
begin
  Result := nil;
  if sxSharedMemory.AcquireMemory(forWriting, timeout)
    <> nil then begin
    if (not sxSharedMemory.Modified) or
      sxXMLDoc.LoadFromStream(sxSharedMemory.AsStream)
    then
      Result := sxXMLDoc;
    if not assigned(Result) then
      sxSharedMemory.ReleaseMemory;
  end;
end; { TGpSharedXML.Acquire }
function TGpSharedXML.Read(timeout: DWORD): TGpXMLDoc;
begin
  Result := Acquire(false, timeout);
  EndUpdate;
end; { TGpSharedXML.Read }
function TGpSharedXML.BeginUpdate(timeout: DWORD):
  TGpXMLDoc;
begin
  Result := Acquire(true, timeout);
end; { TGpSharedXML.BeginUpdate }
procedure TGpSharedXML.EndUpdate;
begin
  if sxSharedMemory.IsWriting then
    sxXMLDoc.SaveToStream(sxSharedMemory.AsStream);
  if sxSharedMemory.Acquired then
    sxSharedMemory.ReleaseMemory;
end; { TGpSharedXML.EndUpdate }
```

```
{:Subject entry in the Subjects table.}
TGpSESubject = class(TGpSEXMLData)
public
  constructor Create(subjectNode: IXMLNode); override;
  property Handle : TGpSEHandle index 0
    read GetXMLPropSEHandle write SetXMLPropSEHandle;
  property Name : string
    index 1 read GetXMLProp write SetXMLProp;
  property Token     : string
    index 2 read GetXMLProp write SetXMLProp;
  property Interest : integer
    index 3 read GetXMLPropInt write SetXMLPropInt;
  property IsProducer: boolean
    index 4 read GetXMLPropBool write SetXMLPropBool;
end; { TGpSESubject }
{:Data for the Subjects table.}
TGpSESubjectList = class(TGpXMLDocList)
public
  constructor Create; reintroduce;
end; { TGpSESubjectList }
constructor TGpSESubject.Create(subjectNode: IXMLNode);
begin
  inherited Create(subjectNode);
  InitChildNodes(['Handle', 'Name', 'Token', 'Interest',
    'IsProducer'],
    [integer(CInvalidSEHandle), '', CInvalidSEToken, 0,
      false]);
end; { TGpSESubject.Create }
constructor TGpSESubjectList.Create;
begin
  inherited Create('SubjectsTable', 'Subjects', 'Subject',
    TGpSESubject);
end; { TGpSESubjectList.Create }
```

➤ *Listing 2: Implementation of the Subjects table.*

DLLs, works in Kylix (although the shared memory implementation is Windows only at the moment), and I helped write it.

To simplify the code, the shared event system doesn't work with the XML directly, but uses the class mapping code from the `OmniXMLProperties` unit (part of the OmniXML code). For example, the `Subjects` table is implemented with only a few lines of code (Listing 2). All the XML manipulation logic is provided by the ancestor classes `TGpSEXMLData` (a small wrapper around `OmniXMLProperties.TGpXMLData`) and `TGpXMLDocList`. (Actually, you'll see in the code that `TGpSESubjectList` contains many other methods. As they are only small helper routines, not directly related to the way XML storage is implemented, I have removed them to simplify the example in Listing 2.)

As this is getting rapidly off topic, I won't pursue XML binding further. Maybe in a future article, if the Editor will allow, I can describe OmniXML and the accompanying units in more detail.

Speed Issues

Now we can already guess where the general sluggishness of the shared event system comes from. The trouble lies in the constant XML loading and saving. Most of the shared event system tables are quite static, but that doesn't hold for the `Event Queue` table, where new entries are inserted, modified and deleted all the time.

To verify this assumption I fired up my trusty GpProfile profiler (http://gp.17slon.com/gpprofile). I set up two instances of the `testGpSharedEvents` application from the previous issue: one sending and another receiving messages. Only the sending part was instrumented for profiling. Then I sent 500 events (in bursts of ten) over the system. After a few attempts (the trouble with profiling is that it is very easy to profile too little or too much) I got the result that verified my assumptions: the program spent half its time loading and saving XML streams (Table 1). With some clicking in the profiler I also found out that the `Event Queue` table is the one that is constantly calling those two methods. Another trafficky table is `Counters`, but it is smaller and less frequently accessed, and doesn't contribute to the slowness nearly as much.

The results of this first profiling session also reveal one strange anomaly that points to a problem in the shared event code. Of course, I missed it completely at that time and only found it a few days later. If you don't see it either, don't worry. I'll come back to it later.

Shared Tables

It seemed to me that the best way to implement the `Event Queue` table would be to store it in a sort of table (in the database sense), of course stored in the shared memory. It's kind of obvious, but then most good ideas are. In my usual way, I have wrapped the shared table logic into a reusable class `TGpSharedTable` (stored in GpSharedTable.pas).

➤ *Table 1: Original code, ten most CPU-intensive methods.*

Procedure	Time	Time(sec)	Calls	Avg/Call(sec)
OmniXMLUtils.XMLLoadFromStream	40.70%	1.07495	123	0.00874
OmniXMLUtils.XMLSaveToStream	10.30%	10.30%	202	0.00134
GpSharedMemory.TGpBaseSharedMemory.GetSize	8.10%	0.21469	564532	0
GpSharedMemory.TGpSharedMemory.AcquireMemory	7.80%	0.20704	251	0.00083
OmniXMLShared.TGpSharedXML.Acquire	7.50%	0.1981	249	0.0008
GpSharedMemory.TGpSharedStream.Read	4.60%	0.12232	281455	0
GpSharedMemory.TGpBaseSharedMemory.GetAsStream	4.10%	0.10816	539485	0
OmniXMLUtils.GetNodeText	1.60%	0.04207	4331	0.00001
GpSharedEvents.TCustomGpSharedEventSubject.DoEventSent	1.40%	0.03765	33	0.00114
GpSharedMemory.TGpSharedStream.GetCurrentData	1.10%	0.02828	281178	0

Interface to the shared memory table allows you to add and remove entries (rows for you SQL people), modify data, search for specific values and iterate over stored entries.

A shared table is of course stored in the shared memory block, implemented by the ubiquitous `TGpSharedMemory`. At the beginning of this shared memory lies a small header. It is not explicitly declared in the code but we can still think of it as being declared as a short packed record containing five fields, a signature (used to verify if the memory block indeed contains a shared table), the number of entries (rows), a small alignment filler, and two headers (one for the free list and another for the used list: I'll be discussing those in a moment.).

This header is followed by the table entries. They are not positioned sequentially after the header, but can appear anywhere in the shared memory, in any order. The code can find all the entries not because their addresses are stored in some kind of table, but because they are linked in a doubly-linked list: the first 4 bytes in each entry contain the address of the next entry and the next 4 bytes contain the address of the previous entry.

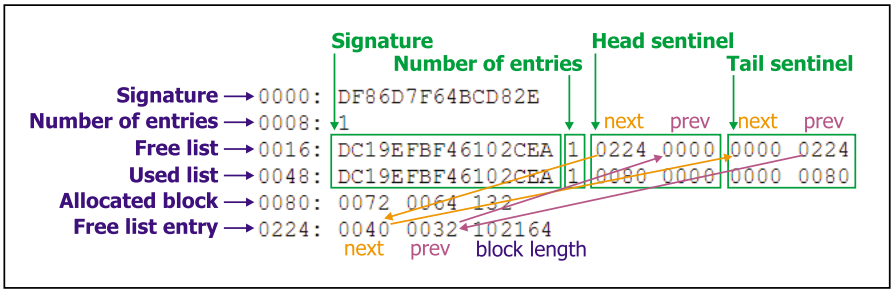
Those addresses are absolute offsets inside the shared memory block (ie, the first byte of the free list header has address 16). The other option I considered was to use an offset from the start of the list header (the first byte of the free list header would have address 0; the same for the first byte of the used list header, as both would be calculated relative to the start of the respective header), but I chose the former approach simply because it is easier to debug.

In order to easily find free space for new entries, the shared table manages all the free space in a free list, which contains free blocks, again doubly-linked. In fact, the same class (`TGpSharedLinkedList` from GpSharedMemory.pas) is responsible for both lists.

`TGpSharedLinkedList` implements the doubly-linked list by using the first eight bytes of each linked block as a placeholder for the pointers to the next and previous blocks in the list. Both ends of the list are protected by sentinels, which are part of the header. Although it is never declared as such, the shared list header could be represented by a packed record holding a signature, the number of entries in the list, a four-byte alignment filler, a head sentinel (containing next and previous pointers) and a tail sentinel.

When the shared table is first created, the used list is empty and the free list contains only one block, spanning the entire free memory (even if it is not committed yet: `TGpSharedMemory` does that for us automatically anyway).

➤ *Figure 2: Memory snapshot of an empty shared table*




```
function TGpBaseSharedTable.AllocateRow(dataSize: integer): integer;
begin
  dataSize := ((dataSize + CEntryHeaderSize - 1) div 16 + 1) *
    16 - CEntryHeaderSize;
  Result := FindSmallestFreeBlock(dataSize);
  if (DataLength[Result] - dataSize) >= (16 - CEntryHeaderSize + 16) then
    stFreeList.EnqueueAfter(Result, SplitEntry(Result, dataSize));
  stFreeList.Dequeue(Result);
end; { TGpBaseSharedTable.AllocateRow }
function TGpBaseSharedTable.AppendRow(rowStream: TStream): integer;
var baseAddress: integer;
begin
  baseAddress := AllocateRow(rowStream.Size);
  if baseAddress = 0 then
    raise EGpSharedTable.CreateFmt('Shared memory %s is full', [Name]);
  rowStream.Position := 0;
  rowStream.Read(DataAddress[baseAddress]^, rowStream.Size);
  stUsedList.EnqueueTail(baseAddress);
  Result := NumberOfEntries;
  NumberOfEntries := NumberOfEntries + 1;
end; { TGpBaseSharedTable.AppendRow }
```

➤ *Listing 3: Appending new entry to the shared table.*

When the application calls `Add` to allocate new entry, nothing much happens. The `Add` method only clears the internal instance of the object representing one table entry and returns it. (Of course, the application must have already called `BeginUpdate` before that). The application can then manipulate the entry, which keeps its data in memory streams and doesn't manipulate the shared memory contents at all. Only when it is absolutely necessary that the shared memory blocks reflect the true state (for example when application calls `EndUpdate`) is the entry stored in the table.

At that moment, the `AppendRow` method (Listing 3) is called. Its only parameter contains the data from the entry the application just manipulated. After allocating a new block of memory from the free list (`AllocateRow`) the code writes the entry directly into the shared memory at the appropriate address (`rowStream.Read`). It puts a new block at the end of the used list (`stUsedList.EnqueueTail`) and updates the number of entries. The state of the memory at that moment is shown in Figure 2.

Another interesting part of the code is the `AllocateRow` method. Its main function is to find a free block that can hold at least the specified number of bytes. At the same time it also tries to reduce memory fragmentation by implementing two measures. The first is to round the data size up to the nearest multiple of 16. The calculation in the code is

quite convoluted because the code must actually allocate 12 more bytes than specified: 4 for the pointer to the next entry, 4 for the pointer to the previous entry, and 4 to hold the length of the allocated area.

This free memory block is then split into two parts: one for the newly allocated data and another to be kept in the free list. If the free block is only slightly larger than the requested size, it won't be split at all (this is the second anti-fragmentation measure). In all cases, the free memory block is removed from the free list (`stFreeList.Dequeue`) and its base address is returned as a function result.

When updating existing data, `UpdateRow` is called instead of `AppendRow`. It first locates the base address of the entry being updated (function `LocateBase` converts the entry index into its base address). Then it checks whether the new entry is not larger than the already allocated block. If so, the shared memory is updated without resizing the block (again, to minimise fragmentation). If the entry size grows the code must, however, allocate a new block (by calling `AllocateRow`). It then puts the new data into this block, adds it to the end of the used list, and moves the original block from the used list to the free list.

In a similar manner, the `Delete` method (not shown) locates the base address for the entry, moves the block from the used list to the free list, and decrements the field holding the number of entries.

Free List Fragmentation

The important thing here is that `Delete` does not merge the newly created free block with the other free blocks, even if it is adjacent to another block. The free memory inside the shared table therefore gets fragmented during the table lifetime.

To fight these fragmentation problems, `EndUpdate` calls the `Defragment` method immediately before releasing the memory block. `Defragment` first checks if defragmentation is required and quickly returns if not. Currently, the table is deemed 'too fragmented' if the average free block size is not at least twice as big as an average used block (if you have any better suggestion, I'm all ears).

If defragmentation is required, the code builds a list of all the used blocks and sorts it by the base address. It then clears the used lists, moves each block as near to the beginning of the shared memory block as possible, and rebuilds the used list at the same time. Finally the free list is cleared and replaced with one entry containing all the space left.

This method may reorder entries: you have to live with that or write a better `Defragment`.

Entry Address Cache

As you can see, the code extensively uses `LocateRow` to convert entry indices into base addresses. This code becomes even more important if we take a look at a `GetItem` method (Listing 4), which handles indexed access to stored data. Basically, it locates the base address of an entry, loads its contents from the shared memory, and returns the entry object.

As the application may examine and modify fields in any order, the shared table class needs to have a fast mechanism for converting an entry index into a base address. On the other hand, an application may only want to access a few entries. In that case, it would not be appropriate to create an entire mapping table (indices to addresses) each time memory is accessed.

To solve both problems at once, `TGpSharedTable` uses a semi-permanent cache, which is only loaded on demand. In other words, the code only locates entries that the application needs, but stores all the resolved entries in the cache. The cache is only cleared when the shared memory block is modified by another `TGpSharedTable` object.

To achieve that, `LocateBase` method first asks the cache to retrieve the base address for the specified entry index, or at least an address of the highest known index. In the latter case, `LocateBase` starts with the address returned from the cache and uses the used list pointers to walk the entries. It does that until it reaches the end of the list or the requested entry. In addition, it stores all newly found entries into the cache.

Fields

Each entry in the shared table is represented by the `TGpSharedTableEntry` object (or a descendant: you can pass the class to be generated to the `TGpSharedTable` constructor), which is not generated dynamically, but stays static through the lifetime of the `TGpSharedTable`. This object should only be accessed through the `Items` property or `Add` method.

➤ *Listing 4: Mapping entry index into base address.*

```
function TGpBaseSharedTable.LocateBase(rowIndex: integer):
  integer;
var lastLowestItem: integer;
begin
  if not stItemCache.RetrieveBase(rowIndex, Result,
    lastLowestItem) then begin
    while lastLowestItem < rowIndex do begin
      if lastLowestItem < 0 then
        Result := stUsedList.Head
      else
        Result := stUsedList.Next(Result);
      Inc(lastLowestItem);
      if Result = 0 then
        break // while
      else
        stItemCache.Insert(lastLowestItem, Result);
    end; //while
  end;
end; { TGpBaseSharedTable.LocateBase }
```

```
function TGpBaseSharedTable.GetItem(idxItem: integer):
  TGpSharedTableEntry;
var
  baseAddress: integer;
  rowStream  : TGpFixedMemoryStream;
begin
  if stInstance.Index <> idxItem then begin
    baseAddress := LocateBase(idxItem);
    if baseAddress <> 0 then begin
      rowStream := TGpFixedMemoryStream.Create(
        DataAddress[baseAddress]^,
        DataLength[baseAddress]);
      try
        stInstance.Reload(idxItem, rowStream);
      finally FreeAndNil(rowStream); end;
    end;
  end;
  Result := stInstance;
end; { TGpBaseSharedTable.GetItem }
```

```
procedure TGpBaseSharedTableEntry.Reload(rowIndex: integer; rowStream: TStream);
var
  fieldLength: integer;
begin
  InternalClear(tesCopy, rowIndex);
  while (rowStream.Read(fieldLength, 4) = 4) and (fieldLength >= 0) do
    if fieldLength > 0 then
      Field[steFields.Count].CopyFrom(rowStream, fieldLength)
    else
      Field[steFields.Count]; // touch to create
  AfterFetch;
end; { TGpBaseSharedTableEntry.Reload }
function TGpBaseSharedTableEntry.AccessField(fieldIndex: integer): TMemoryStream;
begin
  while steFields.Count <= fieldIndex do
    steFields.Add(TMemoryStream.Create);
  Result := TMemoryStream(steFields[fieldIndex]);
end; { TGpBaseSharedTableEntry.AccessField }
```

➤ *Listing 5: Loading shared table entry.*

From the application viewpoint, `TGpSharedTableEntry` is pretty formless. It contains fields that can hold formless (blob) data, and are only accessed by the index (starting from 0). In addition, all the table entries don't have to contain the same number of fields. It is entirely possible to store one field in one entry and ten in another. All in all, the structure is pretty flexible and could be used as a basis for a more rigid shared database manager. Usually, it is best to wrap it in a descendent class that defines stricter access to the fields. To help create such a class, `TGpSharedTableEntry` contains some helper getters and setters that all map to `GetField` and `SetField` methods.

So how is the entry stored in the shared memory? It's quite trivial: each field data item is prefixed by the field data length (which may be 0, meaning the field is empty). The end of the field data is signalled by a length of -1. Usually, there will be some unused space at the end of the entry, because the block allocator allocates the memory in blocks of 16 bytes.

To load the entry into working memory, `TGpSharedMemory` calls the entry's `Reload` method (Listing 5), passing it the index of the entry and a stream containing the entry's data. `Reload` loads the length of each field from the stream, automatically creates a `TMemoryStream` belonging to the entry, by accessing the `Field` property, and loads the field's data from the stream into the `TMemoryStream`. It stops when the field length is less than 0. Finally it calls `AfterFetch`, which is a do-nothing method in the base class but may be overriden in the derived class to add some additional processing (we'll be using this in the `Event Queue` table).

The automatic creation of per-field memory streams is implemented in the `Field` getter: `AccessField`. This method simply adds `TMemoryStreams` to the internal `steFields`: `TObjectList` object until it has reached the index requested. In the current version of the code, the fields are always created sequentially, from 0 onward, and `AccessField` could be much simpler, but I prefer to write generic code whenever it does not hurt performance.

In fact, this generic approach allows me to add fields on the fly without further complications. For example, let's say that application accessed an entry with five fields (indices 0 to 4). It can then freely access `Field[10]`. That will automatically create empty fields [5] to [9], which will then of course be saved into the shared table. Because all the magic is done in `AccessField`, the `GetField` and `SetField` methods are very simple.

Shared Tables And Shared Event System

Besides implementing the `Event Queue` table as a shared memory table instead of a shared XML document, I have also implemented the `Counters` table as a shared table; or, to be more exact, as a `TGpSharedNamedCounters`, which is a simple wrapper around the shared table. `TGpSharedNamedCounters` manages a table of counters, distinguished by names. Each counter has one integer value attached. The application can query the current value of the counter and increment it by one.

The most important aspect of the shared counters class is its educational aspect: it shows how to create a descendant of the `TGpSharedTableEntry` class (Listing 6). As you can see, no code was required to construct a useful class with named access to the fields: we only had to declare two properties that use accessor methods nicely provided by the `TGpSharedTableEntry` class.

The declaration of the `Event Queue` entry is quite similar, with two additional twists (Listing 7). Instead of working directly with the `GetFieldInt` and `SetFieldInt` accessors, it declares the methods `GetFieldHandle` and and `SetFieldHandle`, which simply remap the `TGpSEHandle` type into an integer and back. It also implements a `Listeners` field, which is a list of integers (and can be manipulated as such by an application), but is stored as a formless blob. Conversion between the field data and the list of integers is done in the `AfterFetch` and `BeforePost` methods, which are called at the appropriate time from the `TGpSharedTable` code.

Profiling, Take Two

After all those modifications, it is time for a new profiling session. The results are encouraging.

➤ *Listing 6: Definition of the shared named counters table entry.*

```
const
  CCountersFieldName  = 0;
  CCountersFieldValue = 1;
type
  TGpSharedNamedCountersEntry = class(TGpSharedTableEntry)
  public
    property Name: string index CCountersFieldName
      read GetFieldStr write SetFieldStr;
    property Value: integer index CCountersFieldValue
      read GetFieldInt write SetFieldInt;
  end; { TGpSharedNamedCountersEntry }
```

```
const
  CRecHandle       = 0;
  CRecEvent        = 1;
  CRecProducer     = 2;
  CRecData         = 3;
  CRecDataSize     = 4;
  CRecSharedMemory = 5;
  CRecListeners    = 6;
type
  TGpSEEventQueueEntry = class(TGpSharedTableEntry)
  private
    eqeListeners: TGpSEHandleList;
  protected
    procedure AfterFetch; override;
    procedure BeforePost; override;
    function  GetFieldHandle(idxField: integer):
      TGpSEHandle;
    procedure SetFieldHandle(idxField: integer; value:
      TGpSEHandle);
  public
    constructor Create(owner: TGpBaseSharedTable); override;
    destructor  Destroy; override;
    property Handle: TGpSEHandle index CRecHandle
      read GetFieldHandle write SetFieldHandle;
    property Event: TGpSEHandle index CRecEvent
      read GetFieldHandle write SetFieldHandle;
    property Producer: TGpSEHandle index CRecProducer
      read GetFieldHandle write SetFieldHandle;
    property Data: string index CRecData
      read GetFieldStr write SetFieldStr;
    property DataSize: cardinal index CRecDataSize
      read GetFieldCardinal write SetFieldCardinal;
    property SharedMemory: cardinal index CRecSharedMemory
      read GetFieldCardinal write SetFieldCardinal;
    property Listeners: TGpSEHandleList index CRecListeners
      read eqeListeners;
  end; { TGpSEEventQueueEntry }
procedure TGpSEEventQueueEntry.AfterFetch;
var
  strListeners: TMemoryStream;
begin
  inherited;
  strListeners := TMemoryStream.Create;
  try
    GetFieldStream(CRecListeners, strListeners);
    strListeners.Position := 0;
    eqeListeners.LoadFromStream(strListeners);
  finally FreeAndNil(strListeners); end;
end; { TGpSEEventQueueEntry.AfterFetch }
procedure TGpSEEventQueueEntry.BeforePost;
var strListeners: TMemoryStream;
begin
  strListeners := TMemoryStream.Create;
  try
    eqeListeners.SaveToStream(strListeners);
    if not EqualsField(CRecListeners, strListeners.Size,
      strListeners.Memory^) then begin
      strListeners.Position := 0;
      SetFieldStream(CRecListeners, strListeners);
    end;
  finally FreeAndNil(strListeners); end;
  inherited;
end; { TGpSEEventQueueEntry.BeforePost }
```

➤ *Listing 7: Shared Event table entry.*

The time is spent more evenly and, what is even more important, the total time spent in the profiled code is much shorter. `XMLLoadFromStream` uses only 0.06 second. In the first test, this number is greater than one second.

What surprised me was that there were still so many calls to that method present. After some clicking in the profiler, I discovered that most calls come from the `PopulateListenersList`, via `FilterSubjects` and `BeginUpdate`. While it is true that `FilterSubjects` *may* need write access, it is also true that in most cases it requires only read access. Because `FilterSubjects` acquires the shared XML document for write access, the memory access code modifies the 'dirty' flag and the XML document is reloaded on the next read access. (By the way, that was the anomaly that was visible in the first profiling session too, but which I hadn't noticed until this point.)

The problem is easily solvable, though. I have modified `FilterSubjects` to only request read access by default. The code will acquire write access only if it locates some problems that require the subject table to be modified (which is usually not the case).

The next profiling session showed that there were now only 17 calls to the `XMLLoadFromStream` code and that it used only 13.3% of the total time. With some more research I have found out that it was only called when subjects and events were registered and unregistered.

For the comparison, I have also profiled the sending of 500 messages (in bursts of 10) from one application to another. While the `XMLLoadFromStream` was still called 17 times, it didn't even make it into the top 10.

Let's take a look at the four most CPU-intensive methods. We can't do much about the `AcquireMemory` (it uses a slow Windows API), `EventSend` (which spends most of its time in the `OnEventSent` handler), and `GetXMLPropInt64` (which is fast, but is called many times), but maybe we can speed up `Reload`.

Most of the time `Reload` is called from `Locate`, the function that iterates over the table and compares the given field in each entry with some value. The trouble with `Locate` is that it is programmed exactly as described in the previous sentence. It iterates over the table, loads each entry in turn (with a call to `Reload`) and asks the loaded entry to check if the field passed to the `Locate` contains the given value.

Loading the entire entry to access only one field is not a very effective way to solve the problem, so I modified `Locate`, which now works in a completely different manner. It iterates over the used list, passes the base address of each entry to the `TGpSharedTableEntry` class, and asks it to return the position and size of the field we are interested in. After that, it compares the value at the given position to the value passed to the `Locate` method.

The advantage of this approach is that entries are not fully loaded into memory. The new code is, however, slightly more complicated and not much faster compared to the older one.

That is all for today. We have now got a much faster event system (more than ten times faster than the old one) and also three new useful classes (shared table, shared linked list and shared named counters). Not bad for one issue's work, don't you think?

Primoz Gabrijelcic is R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com