Shared Events Part 1: Rationale, Design, Implementation

by Primoz Gabrijelcic

Hello again. Today I won't bother you with the history of this occasional series: you surely know what I have been writing about in previous issues of this great magazine (if not, reread Issues 86, 88, 91 and 95). In this article I'll describe a system that allows multiple applications running on one computer to communicate freely, without the limitations of the Windows message-passing system and without the complexity of the shared memory or shared data approach.

Motivation

I'm not usually writing code just for the sake of coding. Sometimes, yes, because I like to create and I like to code, but most of the time my code is written with a purpose in mind. Today's classes are no different. I was writing a framework for an all-in-one service, configuration and monitoring program; that is, a single exe that can be installed as a Windows NT service, can be started as a GUI application to configure this service, can run in the tray and display the service state, and can even be used to install, uninstall, start and stop the said service. I stumbled into a big problem. Both parts of the user interface, monitoring and configuration, had to know the current state of the service. The former needs it to display an appropriate icon and the latter to optionally restart the service when new settings are applied. Sure, both could be written by simply querying the service controller, but that would mean the monitoring part must execute this query once every few seconds. Whenever possible, I stay away from polling: it is a bad programming habit.

I looked into several possibilities, from the simple message-based approach (`PostMessage`) to a TCP/IP server implemented inside the service, but in the end I decided on a distributed serverless solution using basic Windows synchronisation primitives to communicate and shared memory to hold the shared data. I designed it as an event broadcasting system, where applications can announce their ability to provide some kind of an event (*publish* the event) or their interest in the same event (*monitoring* the event).

The application would then *broadcast* the event to all interested parties, which will receive this event some short time after. The application that is publishing the event (the *producer*) does not have to know how many monitoring applications (the *listeners*) are alive: one or more, or maybe even none. On the other hand, the listener doesn't have to know if the producer (or producers, as there can be more than one publisher of that event) is alive or not, it just has to register its interest in the event.

Shared event system, as I nicknamed this approach, is implemented as a shared set of in-memory tables, which are accessed and manipulated by producers and listeners. The important part is that there is no dedicated server: housekeeping is distributed between the producers and listeners.

In the first approach to the problem, each producer could see all the listeners on the same computer and vice versa. I have, however, decided against that approach. I think that it is best to contain producers and listeners in small communities that can't interact with each other. The shared event system belonging to one project (which can, of course, consist of many applications) can be completely separated from the shared event system of another project, even if both are running at the same time on the same computer.

To facilitate that, I have introduced the well-known concept of the *namespace*. Each *subject* (producer or listener) has a namespace (a simple string) assigned and only subjects within the same namespace can see each other. I usually use hierarchically named namespaces with a GUID inserted somewhere to guarantee uniqueness, for example:

```
/Gp/SampleProject/854BAE86-
  4E63-48B8-AC92-A0BE7BD51E41
```

but you are free to use anything inside this string.

Such a complex system surely has some limitations. The most important one is that it is kind of slow. I don't recommend it for activities that are expected to trigger more than one event per second, on average (or a few events per second on faster computers). That may change in future releases, though, so stay tuned. The second limitation is that the shared event system is a local solution: you cannot use it to communicate between computers, but then it wasn't designed with that usage in mind.

If you are interested in the all-in-one service, monitoring and configuration framework, I'm afraid you will have to wait a few months: I'll describe it in a future article.

Simple Demonstration

For starters, let's see how the code described in this article is used. When you install the package from this month's code archive, you'll get two additional components on the `Gp` tab: `GpSharedEventProducer` and `GpSharedEventListener`. Start a new project and drop both components on its main (and only) form (or open the Intro1 project available in this month's code).

What we want to do in this demonstration is to send a notification from an application to all listening applications each time the user clicks a button. This is not a complicated task, I admit, but still one that nicely demonstrates the use of the shared event components.

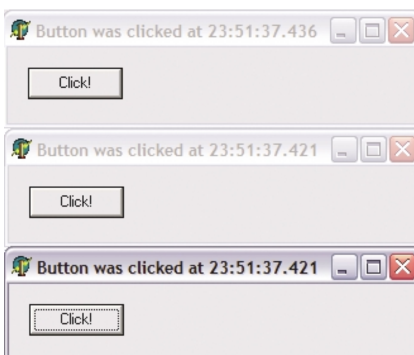
For starters, we have to connect the listener and producer. This is established by setting the `Namespace` property in both of the components to the same value; for example `GpSharedEvents/Introduction`.

Next, we must select the name for the event that will be triggered by the user. We can, for example, name it `ButtonClicked` (If we compare the shared event system with Delphi events, this name would roughly correspond to the name of the event handler, for example `OnClick`). But before we can trigger this event, we must *publish* it. To do so, just select the `Producer` component, double-click on the `PublishedEvents` property, and enter `ButtonClicked` into the editor.

That allows us to *broadcast* the event with a call to the `BroadcastEvent` method. To test it, put a button on the form, double-click it and enter the one-liner shown in Listing 1.

This simple code tells the `Producer` component to broadcast the event `ButtonClick` to all interested listeners. To declare its interest, the listener must *monitor* the event. Monitoring an event is as simple as publishing it: just select the `Listener` component, double-click on the `MonitoredEvents` property, and enter `ButtonClicked` into the editor.

Figure 1




```
procedure TForm1.Button1Click(Sender: TObject);
begin
  GpSharedEventProducer1.BroadcastEvent('ButtonClicked');
end; { TForm1.Button1Click }
```

Listing 1: Broadcasting an event.

```
procedure TForm1.GpSharedEventListener1EventReceived(Sender: TObject;
  producerHandle: TGpSEHandle; const producerName, eventName, eventData: String);
begin
  if eventName = 'ButtonClicked' then
    Caption := 'Button was clicked at '+FormatDateTime('hh:mm:ss.zzz', Now);
end; { TForm1.GpSharedEventListener1EventReceived }
```

Listing 2: Receiving an event.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  GpSharedEventProducer1.BroadcastEvent('ButtonClicked',
    FormatDateTime('hh:mm:ss.zzz', Now));
end; { TForm1.Button1Click }
procedure TForm1.GpSharedEventListener1EventReceived(Sender: TObject;
  producerHandle: TGpSEHandle; const producerName, eventName, eventData: String);
begin
  if (eventName = 'ButtonClicked') and
    (producerHandle <> GpSharedEventProducer1.ProducerHandle) then
    Caption := 'Button was clicked at '+eventData;
end; { TForm1.GpSharedEventListener1EventReceived }
```

Listing 3: Changes in the Intro2 application.

Now we only have to react to this event. To do so, write the `OnEventReceived` event handler (see Listing 2).

As you can see, the shared events infrastructure doesn't yet allow you to write a specific event handler for each monitored event. You must write a generic handler and then test for possible events inside it. This is not a very good design and will be most probably be fixed very soon (hopefully before the next article in this series comes out). You can be sure, however, that `OnEventReceived` will stay: it will still have some uses (for example, to log all the events in a central place).

Compile this application and start three instances of it. Click the button in any of those instances and you'll see that the caption bar in all three instances will update (see Figure 1).

Let's iron out two little glitches in this application (which again may not be glitches at all: it all depends on your viewpoint). The first little 'problem' is that all instances are notified on the button click, even the one that broadcasts the event. The second problem is that the time displayed in receiving instances differs, which is quite reasonable as it is sampled on the listener side and not generated on the producer.

To solve the latter problem, we should record the time on the producer and broadcast it to listeners. The `BroadcastEvent` method facilitates this by providing a second parameter that is by default set to an empty string. This second parameter is passed to all `Listeners` in the form of the `eventData` parameter of the `OnEventReceived` handler.

The former problem can be solved in more than one way. The code that is broadcasting the event could, for example, remove `ButtonClicked` from the list of monitored events before broadcasting (and then add it back). While good enough for this demonstration program, this could cause a problem inside a real-world application. If another application broadcasts the same event while it is disabled in the first application, it (the first application) won't see that broadcast.

A better approach is to check the producer's handle (the ID assigned to each producer and listener, which is unique in the namespace) and compare it with

the handle of the GpSharedEventProducer1. If they are the same, this event was generated internally and should be ignored (see Listing 3). Both changes are implemented in the project Intro2.

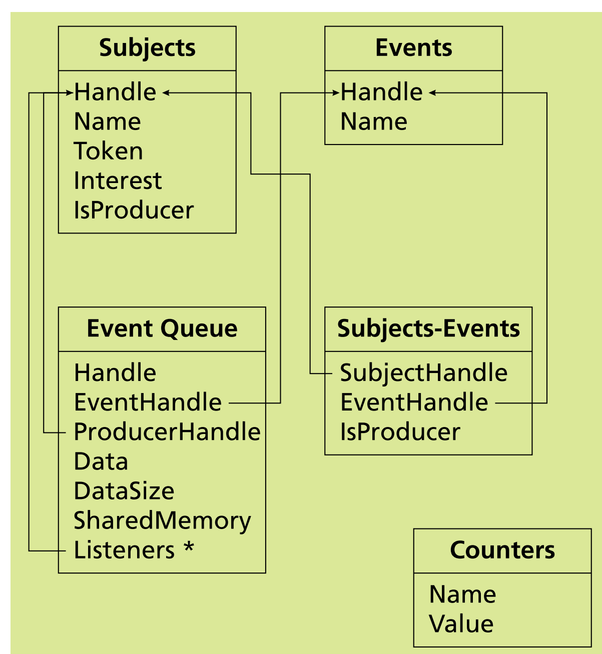
The last issue I have with this code is that the name ButtonClick is a literal string, which we have entered four times: this is a great opportunity to introduce some hard to find error. It would be better to make it into a constant. While that causes no problem with the Button1.OnClick handler and the GpSharedEventListener1.OnEventReceived handler, we have to add some code to cover the initialization. We are not able to modify the PublishedEvents and MonitoredEvents properties in the Object Inspector (because we cannot enter the name of the constant holding the event name here), so we must do that in code. Instead of working on those properties directly (both of them are pretty simple TStringLists), we can just call the PublishEvent and MonitorEvent methods (see Listing 4). Project Intro3 implements this change.

While we're on the topic I should mention that there are also UnpublishEvent and IgnoreEvent methods which remove the event from the PublishedEvents and MonitorEvents lists.

```
const
  CButtonClicked = 'ButtonClicked';
procedure TForm1.Button1Click(Sender: TObject);
begin
  GpSharedEventProducer1.BroadcastEvent(CButtonClicked,
    FormatDateTime('hh:mm:ss.zzz', Now));
end; { TForm1.Button1Click }
procedure TForm1.GpSharedEventListener1EventReceived(Sender: TObject;
  producerHandle: TGpSEHandle; const producerName, eventName, eventData: String);
begin
  if (eventName = CButtonClicked) and
    (producerHandle <> GpSharedEventProducer1.ProducerHandle) then
    Caption := 'Button was clicked at '+eventData;
end; { TForm1.GpSharedEventListener1EventReceived }
procedure TForm1.FormCreate(Sender: TObject);
begin
  GpSharedEventProducer1.PublishEvent(CButtonClicked);
  GpSharedEventListener1.MonitorEvent(CButtonClicked);
end; { TForm1.FormCreate }
```

➤ *Listing 4: Event name as a constant.*

➤ *Figure 2*



Data Model

Before we plunge into the depths of the shared events code and discover how it works without a dedicated server process, let's examine the data model. For the time being, just imagine that there is a server (an SQL server, if you want) that manages a few tables for us (see Figure 2).

Shared data is split into four main tables (which, I must immediately admit, are not really normalised as you would expect database tables to be) and one helper table. Of the main tables, Subjects contains all the active producers and listeners, Events holds all the published and/or monitored events, Subjects-Events specifies connections between Subjects and Events and Event Queue keeps all the events currently being broadcasted. A helper table, Counters, just contains name/value pairs, and helps implement global counters, used to allocate unique handles for our entities.

The Subjects table contains information on active producers and listeners. Each has a unique handle, a (potentially empty) name, which is used only for cosmetic purposes (for example, when you want to enumerate all producers) and a flag that specifies whether this is a producer or listener (IsProducer). Each subject also contains a unique token (tokens are systemwide unique strings, in case you skipped Issue 86.) The last field, Interest, specifies the event classes this subject is interested in. In short, this field allows the event engine to optimize some system broadcasts, for example when a new subject appears in the system. I'll explain this process in more detail later.

A much simpler table, Events, contains the names of all registered events (either published or monitored) and their associated handles.

Connections between subjects and events are stored in the Subjects-Events table. Each published event creates a new entry containing producer and event handles. In a similar manner, each monitored event creates an entry with listener and event handles. As a simple optimisation, the IsProducer field contains a Boolean value indicating whether the table row belongs to the producer or listener. This table is used when an event is broadcasted. To build the list of interested listeners, the publisher iterates over the table and selects all listeners (ie, where IsProducer = false) that registered interest in that event.

Finally, the most important and busiest table, Event Queue, contains all the active events. Those are the events that were broadcasted and are still waiting to be processed by at least one listener. Each broadcast is represented with one entry in this table and

contains a unique handle, which the code uses internally during the event dispatch (and which, incidentally, is also returned as a result of the `BroadcastEvent` call). As we need to know who broadcasted the event and which event is in question, each entry also contains a subject handle and an event handle. There can be more than one recipient of the broadcast, therefore we need a list of listeners (their handles), which is stored in the `Listeners` field.

That covers the housekeeping needs, but we still have to store the message data somewhere. Small messages (under 1Kb) are kept in the table, in the `Data` field, while larger messages are offloaded into a shared memory block (one per event queue entry), whose name is stored in the `SharedMemory` field. In both cases, `DataSize` stores the message data size. This message offloading is actually a dirty hack, designed to speed up message delivery, and is tightly connected to the fact that all the tables are stored internally as XML document. In fact, this XML representation is not really suitable for the `Event Queue` and I suspect that internal representation of this table will be changed in the future releases.

Broadcasting

To better understand how those tables are used, let's take a look at what happens when you run the Intro3 program. (Note that all the component-related code is stored in the GpSharedEvents.pas unit, while the GpSharedEventsImpl.pas unit contains the implementation of the shared event system.)

During the main form creation, the program creates an instance of the `TGpSharedEventProducer` component. When it is activated (the `Active` property is loaded or set in code), the code creates a token and registers producer in the `Subjects` table. The registration code first checks if that token exists in the table (if it does, the existing handle is returned), then allocates a new handle (with the help of the `Counters` table) and inserts a new entry into the table.

A moment later, `TGpSharedEventListener` is created and registered in the `Subjects` table in the same manner.

The next part of the initialisation happens in the form's `OnCreate` handler. First, `PublishEvent` is called, which registers the event in the `Events` table. Event registration is very similar to subject registration: if an event with that name already exists in the table, the existing handle is returned, otherwise a new handle is allocated. After that, `PublishEvent` adds a new entry containing the producer handle and event handle into the `Subjects-Events` table.

When `MonitorEvent` is called, the event will already exist in the `Events` table and only its handle will be retrieved. Of course, a new entry containing the listener handle and event handle will still be added to the `Subjects-Events` table.

That concludes the initialisation phase. Now we have two entries in

```
function TGpSharedEventManager.InternalBroadcastEvent(
  out eventQueueHandle: TGpSEHandle; const eventName,
  eventData: string; excludeListener, sendOnlyTo:
  TGpSEHandle; interestFlags: integer): boolean;
var
  eventDataCopy: string;
  eventDataSize: cardinal;
  eventHandle  : TGpSEHandle;
  listeners    : TGpSEHandleList;
  publishes    : boolean;
  sharedMemory : TGpSharedMemory;
begin
  eventDataCopy := eventData;
  eventDataSize := Length(eventDataCopy);
  listeners := TGpSEHandleList.Create;
  try
    Result := PopulateListenerList(listeners, eventName,
      sendOnlyTo, excludeListener, interestFlags,
      eventHandle);
    if Result then begin
      publishes := true;
      if (eventName <> CSystemEvent) and
        (not emSEMappings.SubjectPublishes(emSubjectHandle,
        eventHandle, true, publishes)) then
        Result := SetError(emSEMappings)
      else if not publishes then
        Result := SetError(Ord(semErrInvalidEvent),
          sSubjectDoesntPublishEvent, [eventName,
          eventHandle])
      else begin
        if listeners.Count > 0 then begin
          OffloadLargeMessage(eventDataCopy, sharedMemory);
          if not emEventQueue.Insert(eventHandle,
            emSubjectHandle, eventDataCopy, eventDataSize,
            listeners, sharedMemory, eventQueueHandle) then
            Result := SetError(emEventQueue)
          else if NotifyListeners(listeners) then
            Result := ClearError;
        end else begin // nothing to do = all done
          if eventName <> CSystemEvent then
            if assigned(emOnEventSent) then
              emOnEventSent(self, eventQueueHandle,
                eventHandle, eventName);
          Result := ClearError;
        end;
      end;
    end;
  finally FreeAndNil(listeners); end;
end; { TGpSharedEventManager.InternalBroadcastEvent }
```

➤ *Listing 5: Broadcasting.*

the `Subjects` table (one for the producer and another for the listener), one entry in the `Events` table (for the `ButtonClicked` event) and two entries in the `Subjects-Events` table (one for the producer and another for the listener). If we start a new instance of the Intro3 application, we'll get two new entries in the `Subjects` table and two in the `Subjects-Events` table.

Interesting things start happening when a user clicks a button and triggers a `BroadcastEvent` call, which quickly passes control to the `InternalBroadcastEvent` (see Listing 5). This first creates a list of all listeners interested in the event. If there is no such listener, the code triggers the producer's `OnEventSent` event (a Delphi event, to make sure we understand each other: the classic kind you can set up in the Object Inspector) and exits. From the viewpoint of the producer, the event was just sent to all interested parties.

If there are listeners interested in that event, the code checks the size of the event data (the data associated with the event: the second parameter of the `BroadcastEvent` method) and offloads the data into a newly created shared memory block if required.

At the end, the broadcasting code inserts all information on this broadcast into the `Event Queue` table, allocating a new event queue handle in the process, and notifies all interested listeners that a new event is waiting in the event queue. The newly allocated event queue handle is also returned as the `BroadcastEvent` result.

How exactly does this 'notifying the listeners' work? Simply by using Windows events (`CreateEvent`). Each subject (either producer or listener) creates one event with a well-known name. The name of this event is constructed from the subject's token (which other subjects can look up in the `Subjects` table) with the suffix `$NotificationEvent` (see Listing 6). To prevent problems when signalling over the NT desktop (for example, from a service to a GUI application), this event is created with 'allow all' access (see Issue 91 for more detail on NT access protection). In addition, each subject creates one internal window (created with `AllocateHWnd`) and a thread which waits on this event. Whenever the event gets signalled, the thread sends a windows message to its owner.

Receiving

So where are we now? The `Producer` has inserted a new entry into the `Event Queue` table and has finished its job by signalling the listener's notification event. This event wakes up the listener's notification thread, which sends an internal message to the listener's internal window. This message is processed by the window's `WndProc` method, which calls the `InternalProcessAndRemove` method (Listing 7).

➤ *Listing 6: Notifying all listeners.*

```
class function TGpSharedEventManager.MakeNotificationEventName(const tokenName:
  string): string;
begin
  Result := tokenName + '$NotificationEvent';
end; { TGpSharedEventManager.MakeNotificationEventName }
function TGpSESubjects.NotifyListeners(listeners: TGpSEHandleList): boolean;
var
  iListener        : integer;
  iSubject         : integer;
  notificationEvent: THandle{CreateEvent};
  subjects         : TGpSESubjectList;
begin
  subjects := AccessSubjects;
  if not assigned(subjects) then
    Result := SetError(Ord(semErrNotAcquired), sFailedToAcquiredSharedMemory,
      [seSubjData.Name, CSubjectsTimeout])
  else begin
    for iListener := 0 to listeners.Count-1 do begin
      iSubject := subjects.IndexOf(listeners[iListener]);
      if iSubject >= 0 then begin
        notificationEvent := OpenEvent(EVENT_MODIFY_STATE, false,
          PChar(TGpSharedEventManager.MakeNotificationEventName(
            subjects[iSubject].Token)));
        try
          if notificationEvent <> 0 then
            SetEvent(notificationEvent);
        finally CloseHandle(notificationEvent); end;
      end;
    end; //for
    Result := ClearError;
  end;
end; { TGpSESubjects.NotifyListeners }
```

```
procedure TGpSharedEventManager.InternalProcessAndRemove(
  eventQueueHandle: TGpSEHandle; doProcess: boolean);
var
  eqEntries       : TGpSEHandleList;
  eventData       : string;
  eventHandle     : TGpSEHandle;
  eventProducer   : TGpSEHandle;
  eventQueue      : TGpSEEventQueueList;
  eventQueueEntry : TGpSEEventQueueEntry;
  iEventQueue     : integer;
  sendNotification: boolean;
begin
  eventQueue := emEventQueue.BeginUpdate;
  if not assigned(eventQueue) then begin
    if emEventQueue.LastError = Ord(semErrNotAcquired) then
      // Timeout - work around it. We really _must_ remove
      // the listener from the event queue.
      Trigger(emMsgRemoveFromEventQueue,
        WPARAM(eventQueueHandle), 0)
    else
      raise EGpSharedEventManager.Create('Failed to allocate
        event queue');
  end else begin
    sendNotification := false;
    eventProducer := 0; // to keep Delphi happy
    try
      eqEntries := TGpSEHandleList.Create;
      try
        eventQueue.GetAllEntriesForListener(emSubjectHandle,
          eqEntries);
        for iEventQueue := 0 to eqEntries.Count-1 do begin
          eventQueueEntry :=
            eventQueue.Locate(eqEntries[iEventQueue]);
          eventQueueHandle := eventQueueEntry.Handle;
          eventProducer := eventQueueEntry.Producer;
          eventHandle := eventQueueEntry.Event;
          if doProcess then begin
            ReloadLargeMessage(eventQueueEntry, eventData);
            if eventHandle = CInvalidSEHandle then
              ProcessSystemEvent(eventData, eventProducer)
            else
              ProcessUserEvent(eventHandle, eventProducer,
                eventData);
          end;
          sendNotification := RemoveListener(eventQueue,
            eventHandle, eventProducer, eventQueueHandle,
            emSubjectHandle);
        end; //for iEventQueue
      finally FreeAndNil(eqEntries); end;
    finally emEventQueue.EndUpdate; end;
    if sendNotification then
      NotifyEventSent(eventProducer, eventQueueHandle);
  end;
end; { TGpSharedEventManager.InternalProcessAndRemove }
```

➤ *Listing 7: Receiving*

At that moment, the listener cannot know how many events are waiting to be processed in the `Event Queue`. It is entirely possible that more than one producer has signalled its notification event before the listener's thread could be awaken. The listener must therefore scan the entire `Event Queue` and process all the events which have its handle listed in the `Listeners` field.

If by any chance the `Event Queue` cannot be locked for exclusive access (this can happen if lots of producers and listeners are passing events around), an internal message is generated. This message will reach the `WndProc` and `InternalProcessAndRemove` will be triggered again. Therefore we can be assured that each event will reach all the listeners sooner or later.

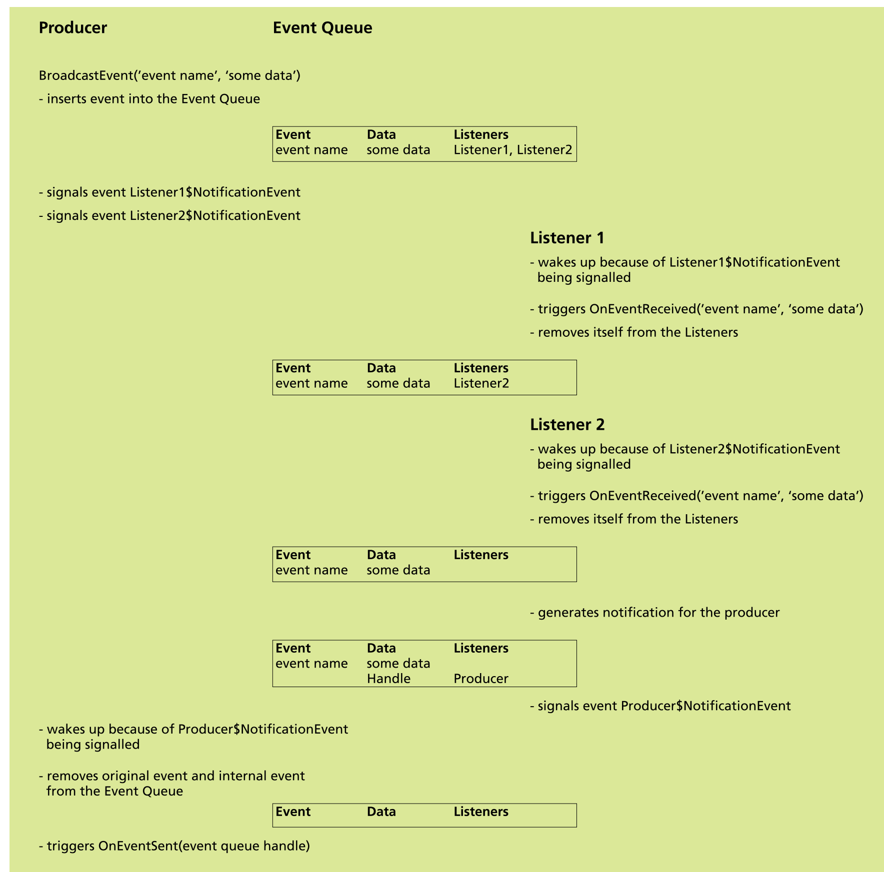
For each event queue entry the listener first retrieves the event data from the shared memory (if the data was offloaded there), then it triggers the `OnEventReceived` event and removes itself from the `Listeners` field. If this was the last listener in the list, the code will notify the producer that the event was sent to all intended receivers. This notification uses the same mechanism as ordinary event broadcasting: the listener generates an internal event (which, unlike user events, has an empty name associated with the otherwise invalid handle), with the event queue handle stored in the event data, and sends it back to the producer.

Of course, this means that now the listener must insert a new entry into the `Event Queue` table and signal the producer's notification event. The producer's notification thread receives this event and posts an internal message to the producer's internal window, where it is processed by the `WndProc`. Just like when the producer broadcasted the event in the first place, just reversed.

The code that processes this internal message in the producer just removes the appropriate entry from the `Event Queue` and destroys the shared memory (if it was created in the first place). At the end, it triggers the producer's `OnEventSent` event. Somewhere along the line, it also removes the internal event from the `Event Queue` table.

Oh, yes, I almost forgot: how does the application know which

➤ *Figure 3*



event was just fully sent when `OnEventSent` is triggered? Simple: the same event queue handle that was returned as a `BroadcastEvent` result is passed to the `OnEventSent` parameter.

The whole broadcasting process is depicted in Figure 3.

That concludes our tour. Really. Except that there are some additional bells and whistles built into the shared event system which I have not yet got around to discussing.

(Not So) Hidden Extras

Although the last section of this article covers event sending and receiving in detail, some parts of the code were skipped for the sake of clarity. Most notably, I have not yet mentioned the validity scans. When some part of the code detects that the remote subject does not exist anymore (by checking the validity of subject's token), it triggers a full validity re-scan.

The validity checker iterates over the `Subjects` table and verifies existence of all subjects. Along the way, it creates a list of all invalid subjects. After that, it iterates over the `Subjects-Events`, `Event Queue` and `Subjects` tables and removes invalid subjects from all the relevant fields (see Listing 8).

```
procedure TGpSharedEventManager.ValidityRescan;
var
  subjectList: TGpSEHandleList;
begin
  subjectList := TGpSEHandleList.Create;
  try
    try
      if not emSubjects.CollectInvalidSubjects(subjectList) then
        Abort;
      if subjectList.Count > 0 then begin
        if not emSEMappings.RemoveInvalidSubjects(subjectList) then
          Abort;
        if not emEventQueue.RemoveInvalidSubjects(subjectList) then
          Abort;
        if not emSubjects.RemoveInvalidSubjects(subjectList) then
          Abort;
      end;
    except
      on EAbort do
        TriggerValidityRescan;
    end;
  finally FreeAndNil(subjectList); end;
end; { TGpSharedEventManager.ValidityRescan }
```

➤ *Listing 8: Validity checks.*

Another feature I have not yet mentioned is support for registration and deregistration notifications. When a subject registers into the system or deregisters from it, when an event is published or monitored (or unpublished or ignored), a special internal message is sent to all listeners that show interest in such an event.

From the user's (programmer's) viewpoint not much has to be done to enable these notifications. When you set up the appropriate event (such as `OnProducerRegistered`, `OnEventPublished`, etc) in the Object Inspector, the property setter updates the value of the `Interest` field in the `Subjects` table. When a subject makes a change that causes registration or deregistration of some kind, it also collects all subjects that are interested in that kind of change (by checking the `Interest` field for all registered subjects) and then generates an internal message describing the change that is sent only to interested subjects. This approach minimises event flow through the system while still allowing every producer and listener to be informed of changes inside the system.

I also haven't mentioned the `SendEvent` method. It allows you to send an event to only one listener. This is especially useful in the `OnEventMonitored` handler. Whenever a producer detects an interest in the event it provides, it can immediately send the interested party some data. For example, that allows my service, configuration and monitoring program to inform the monitoring part of the current service state as soon as the monitor shows interest in the appropriate event.

There is also a way to access all the registered producers (the `Producers` property), listeners (the `Listeners` property), published events (`PublishedEvents`), monitored events (`MonitoredEvents`) and all registered events (`Events`). All are simple `TStringLists`.

All features of the shared event system are demonstrated in the testGpSharedEvents application (which is included in this month's source code, of course).

Next time we will take a look at how the shared tables are implemented internally and we'll try to make the whole shared event system work faster. Till then, have fun with the current implementation and feel free to comment on it via email.

Primoz Gabrijelcic is R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com. All code in this article is freeware and may be freely reused in your own applications. Check http://gp.17slon.com for updates to the published code.

SourceConneXion
Check In, View differences, and much more, all directly from Delphi!
Supports SourceSafe, StarTeam, PVCS, SourceOffSite, Perforce, ClearCase, Surround, MKS, and others
www.epocalipse.com