Put It In A Tree

by Primoz Gabrijelcic

We programmers sure want to complicate our lives. Just think of the ways you are displaying data. Most of the time, `TListBox` is enough. But every now and then, things get complicated. You have to display hierarchical data (or at least your customer thinks so) and there is no way it can be stuffed into a simple list. When that happens, you need a tree. What could be simpler? The internet is full of Delphi components to display trees (in most cases you'll only need Lischke's excellent Virtual Treeview: www.delphi-gems.com). Sometimes, this is enough: take the data and display it. But sometimes this is when your troubles start.

I'll give you a simple example. You've written software to manage incidents. To facilitate incident solving, a discussion can be attached to it. Discussions can be big and they can branch in many directions. Of course, your software has to show them in a hierarchical way. It's just that the data (messages, etc) is stored in the database, which can only provide a flat output. You have to rebuild the tree structure from it, but how?

This is not so rare a problem as you might think. Hierarchical data appears everywhere and most of the time it must be rebuilt from simple non-hierarchical storage. Think about mail, news, CVS repositories and databases, not to mention groupware and chat forums. Trees are everywhere.

To recap, we want to generate something similar to Figure 1, and we want to do it from the flat, non-hierarchical, unconnected data. We will start with a bunch of messages (or other atomic data items) and apply a message threading algorithm (a group of related messages is called a *thread* in most forum software). This algorithm returns a list of top-level messages with child messages attached in a tree structure. Displaying this will be left as an exercise for the reader. (I like to solve problems, not push the pixels on the screen!)

To be as reusable as possible, we'll do it in a high-level way. No pointers will be used, so the code runs on both Win32 and .NET.

Ground Rules

Before we start, let's lay down few rules. Each message (or any other atomic data) contains a unique identifier. In concordance with current internet standards, this identifier is a simple string with an unknown internal structure. Each message also contains an identifier for its parent, which can be an empty string if this is a top-level (root) message. For example, in the email world, these identifiers are usually called `MsgID` (message ID) and `InReplyTo` (ID of the message we are replying to, in other words an ID of the parent message).

Instead of implementing a message as an object, I decided to make it an interface, mostly because we are implementing a *process*, which must somehow attach to the existing infrastructure. It is much easier to implement an additional interface in the basic object (which you are doubtless already using to represent your atomic data) than re-derive this basic object from a different parent. In some cases, the latter option is totally unacceptable.

The `IGpThreadable` interface, which you'll have to implement in your threading-enabled objects, is shown in Listing 1. Besides providing access to a unique ID and the parent's ID (properties `ID` and `ParentID` respectively), it declares the properties `Parent` and `Items`. The former is a link from the child to the parent and the latter contains links from the parent to its children. Both will be filled by the threading method.

The second interface, `IGpThreadableContainer`, specifies access to a container of threadable items. This container is used in a top level (we'll pass such container to the threading method) and again as wrapped storage for the item's children (`IGpThreadable.Items`).

The last interface, which is only used to simplify testing of various threading algorithms, is called `IGpThreader`. It declares just one method called `Thread`, taking one parameter: a container we want to convert into a tree. This is the method we have to develop.

All three interfaces are declared in unit GpThreadable.pas, which also adds two simple base classes: one implementing `IGpThreadable` and the other implementing `IGpThreadableContainer`.

Threading 101

First we have to state the requirements for the threading algorithm. It will take a container of objects which implement `IGpThreadable` with `ID` and `ParentID` set to appropriate values. It then rearranges those objects so that only root (top-level) items are stored in the container. All non-root (child) items are stored in the `Items` property of their respective parents. All child items point back to the parents via the `Parent` property. The original order for each level must also be preserved.

➤ *Figure 1: XanaNews display of a borland.public.delphi.language.basm newsgroup.*

```
IGpThreadable = interface(IUnknown)
['{F6403548-FD09-48D7-AD69-E13A908E59D6}']
  function  GetID: string;
  function  GetItems: IGpThreadableContainer;
  function  GetParent: IGpThreadable;
  function  GetParentID: string;
  procedure SetID(const value: string);
  procedure SetParent(const value: IGpThreadable);
  procedure SetParentID(const value: string);
  property ID: string read GetID write SetID;
  property Items: IGpThreadableContainer read GetItems;
  property Parent: IGpThreadable read GetParent
    write SetParent;
  property ParentID: string read GetParentID
    write SetParentID;
end; { IThreadable }
{:Container for threadable items.
  @since   2005-01-31
}
IGpThreadableContainer = interface(IUnknown)
['{1C3706A7-31F4-46D9-A81F-A9163FC3376F}']
  function  Add(item: IGpThreadable): integer;
  procedure Clear;
  procedure Delete(idxItem: integer);
  function  GetCount: integer;
  function  GetItem(idxItem: integer): IGpThreadable;
  procedure SetItem(idxItem: integer; value: IGpThreadable);
  procedure SetSize(numItems: integer);
  property Count: integer read GetCount;
  property Items[idxItem: integer]: IGpThreadable
    read GetItem write SetItem; default;
end; { IThreadableContainer }
{:Threader for threadable containers.
  @since   2005-02-02
}
IGpThreader = interface(IUnknown)
['{F2E5D0C3-4B5B-4C24-9F4A-70C062B4705D}']
  procedure Thread(container: IGpThreadableContainer);
end; { IGpThreader }
```

➤ *Listing1: Threading interfaces.*

To understand the problem, we can start with a naive implementation. It will take messages one by one (as they appear in the original container). For each message, it will try to insert it into an already-built tree. If this fails, it will leave the message in the container and consider it a root message.

Besides being very quadratic by nature (for each message we have to check all the already-processed items), this threading algorithm exposes a big problem: it doesn't handle out-of-order messages.

In the real world (and especially when dealing with email messages), child objects can appear before the parents. If you are subscribed to at least one mailing list sometimes replies arrive before the message they are replying to.

The naive threader we have just described doesn't handle this situation well. It will take a response, try to find a parent and fail, leaving the response as a root item. When it later encounters the parent it won't know that it has to attach an existing child item to this parent.

We can extend the algorithm to walk over the already-built trees, searching for items that must be inserted into the child list of the currently processed item, but that will slow down this already sluggish method even more.

This simple algorithm is implemented in the unit GpQuadraticThreader.pas and doesn't deserve to appear in the printed article. It is nevertheless included in the demonstration program so you can see for yourself how slow it is. It needs 0.3 seconds to thread 1,000 messages on my Athlon 1266, which is fine, but fails miserably with 10,000 messages, requiring more than 50 seconds for the task. This is simply not acceptable.

Hashed Threading

Let's step back and look at the problem from a distance. We have to build a forest (as such a collection of trees is usually called, in nature and in computer science), step by step. We have to take a message, *decide what to do with it* and place it somewhere.

The next big question is how we can speed up this decision. It would be great if we could just look at the message's `ID` and `ParentID` and automatically filter out the already-built trees that are of no use to us. In other words, we want to connect each `ID` to the trees that contain this `ID`, either as a unique `ID` or as a `ParentID`. In other words, for each `ID` we want to keep a list of candidate trees.

We could build a `TStringList` descendant where we would insert `ID`s into the `Strings[]` property and associated lists of trees (maybe represented as `TInterfaceLists`) into the `Objects[]` property. Operations on such a list (inserting and searching) would be quite slow, however. We'd do much better with a more advanced structure, like a hash table.

As hash tables are not really a topic for this article, we'll make do with a very simple implementation that comes with Delphi: `TStringHash` from IniFiles.pas. If you want to learn more on hashing, read Julian Bucknall's articles and search the internet (http://burtleburtle.net/bob/hash is a good starting point). Suffice to say that a hash table allows us to treat long strings as if they were represented by a numbers, stored in a structure that allows for fast insertion and searching.

Like the rejected naive implementation, this algorithm (let's call it *hashed threading*) takes items from the container one by one (see the method `InternalThread` in the Listing 2). For each item, it calculates a hash of the `ParentID` identifier and looks into a hash table to retrieve a list of all the trees encountered so far that contain this `ParentID`.

Next it checks if the currently processed item can be inserted into one of these trees as a child of an existing item. The method `ListToExistingParent` walks over all trees in this list and, for each tree, checks to see if it contains an item that is a parent to the currently processed item. If a match is found, the program adds the currently processed item to the child list of this parent. It also maps this tree to the unique ID of the message we are processing, by adding it to its tree list. We may need this link later when someone wants to become a child of this item.

As there can be only one parent for each item, there is no need to check the rest of the trees in the list once we find a match.

The harder part is to connect already built trees as the children of the currently processed item (see the method `ListToExistingChildren`, shown in Listing 2). As we have seen before, children can appear before parents so there can be more than one tree waiting to be connected to the currently

processed item. To find such trees, we must calculate a hash of the item's unique identifier and retrieve the associated list of trees. Then we must check all these trees. If the root node of a tree wants to be a child of the current item, we must connect them. At the end, we must find the root of the tree containing the current item (this may differ from the current item as it is possible that it was connected to an existing tree in the previous step) and add it to the hash table.

As there can be more than one child waiting to be connected to the item, we have to check all the trees in the list.

If both previous steps failed, we simply leave the item as a root and create tree lists for its `ID` and `ParentID`. Both lists will initially contain only the current item.

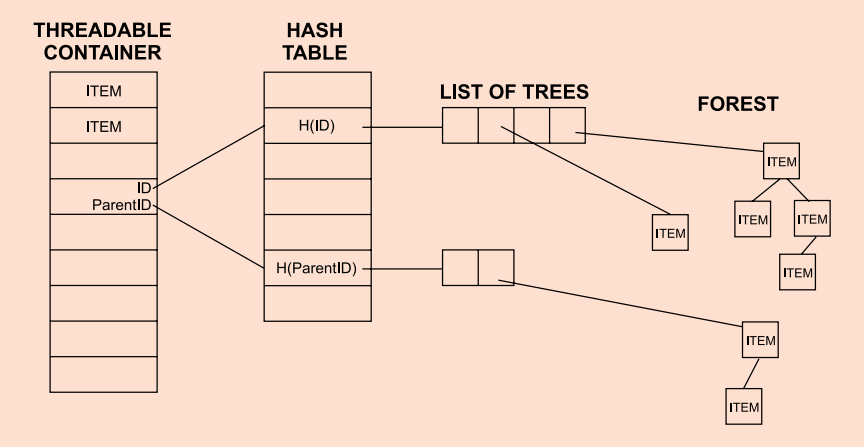
When all the entries are processed, we have to remove all the non-root items from the container's main list, which happens in the `DeleteChildren` method.

To get a better feel for the various data structures used by this algorithm, a simplified version is shown in Figure 2.

This algorithm, implemented in unit `GpHashedThreader`, is much faster than the naive implementation. Even more, it does the job right. Its time requirements are in the order of $O(n*m)$ where n is the number of items being threaded and m is the average size of a parent-child tree encountered during the construction.

Testing on the same data as before showed great improvement: 10,000 messages were threaded in less than 0.2 seconds and 100,000 messages in 2.8 seconds. But this is still not perfect.

➤ *Figure2: Structures used by the hashed threading algorithm.*



Fast As Lightning

I would probably live quite happily with the hashed threader if a smarter guy didn't publish a better algorithm. At the beginning of this article, I was talking about internal incident reporting and discussion systems. Well, wouldn't you believe, I was not making things up. The threading problem was published on Slovenian Delphi forum (www.delphi-si.com, if you think you can cope with the language) by a guy working on just such a system. Some time later he published his solution, which I slightly changed to fit in the framework and added it to my collection of threading algorithms as `GpLeeThreader` (after Lee, its author).

Lee's threader also uses hash tables to speed up the searches, but in a much simpler manner. While processing the items, it will only check the `ParentID` side of the equation (ie, the same part I was

➤ *Listing2: Hashed threading*

```
procedure TGpHashedThreader.InternalThread(container:
  IGpThreadableContainer);
var
  iItem     : integer;
  item      : IGpThreadable;
  itemLinked: boolean;
begin
  for iItem := 0 to container.Count-1 do begin
    item := container[iItem];
    itemLinked := false;
    if LinkToExistingParent(item, GetRootList(
      item.ParentID)) then itemLinked := true;
    if LinkToExistingChildren(item, GetRootList(item.ID))
      then itemLinked := true;
    if not itemLinked then AddToHash(item);
  end;
  DeleteChildren(container);
end; { TGpHashedThreader.InternalThread }
function TGpHashedThreader.LinkToExistingParent(item:
  IGpThreadable; rootList: TGpThreadableList): boolean;
var
  iRoot     : integer;
  parentItem: IGpThreadable;
begin
  Result := false;
  for iRoot := 0 to rootList.Count - 1 do begin
    parentItem := FindItemByID(rootList[iRoot],
      item.ParentID);
    if assigned(parentItem) then begin
      parentItem.Items.Add(item);
      item.Parent := parentItem;
      InsertIntoList(item.ID, parentItem);
      Result := true;
      break; //for
    end;
  end; //for iProxy
end; { TGpHashedThreader.LinkToExistingParent }
```

```
function
TGpHashedThreader.LinkToExistingChildren(item:
  IGpThreadable; rootList: TGpThreadableList): boolean;
var
  iRoot    : integer;
  root, topParent: IGpThreadable;
begin
  Result := false;
  iRoot := 0;
  while iRoot < rootList.Count do begin
    root := rootList[iRoot];
    if root.ParentID = item.ID then begin
      item.Items.Add(root);
      root.Parent := item;
      topParent := FindTopParent(item);
      if not rootList.Contains(topParent) then
        rootList.Add(topParent);
      Result := true;
    end;
    Inc(iRoot);
  end;
end; { TGpHashedThreader.LinkToExistingChildren }
procedure TGpHashedThreader.DeleteChildren(container:
  IGpThreadableContainer);
var
  iItem   : integer;
  item    : IGpThreadable;
  lastRoot: integer;
begin
  lastRoot := 0;
  for iItem := 0 to container.Count-1 do begin
    item := container.Items[iItem];
    if not Assigned(item.Parent) then begin
      container.Items[lastRoot] := item;
      Inc(lastRoot);
    end;
  end;
  container.SetSize(lastRoot);
```

```
procedure TGpLinearThreader.Thread(container:
IGpThreadableContainer);
begin
  ltIDHash := TStringHash.Create(GetGoodHashSize(
   container.Count) div 2);
  try
    CreateIDHash(container);
    ReparentItems(container);
    DeleteChildren(container);
  finally FreeAndNil(ltIDHash); end;
end; { TGpLinearThreader.Thread }
procedure TGpLinearThreader.CreateIDHash(container:
  IGpThreadableContainer);
var
  iItem: integer;
begin
  for iItem := 0 to container.Count-1 do
    ltIDHash.Add(container[iItem].ID, iItem);
end; { TGpLinearThreader.CreateIDHash }
```

```
procedure TGpLinearThreader.ReparentItems(container:
  IGpThreadableContainer);
var
  idxParent: integer;
  item     : IGpThreadable;
  iItem    : integer;
  parent   : IGpThreadable;
begin
  for iItem := 0 to container.Count-1 do begin
    item := container[iItem];
    if item.ParentID <> '' then begin
      idxParent := ltIDHash.ValueOf(item.ParentID);
      if idxParent >= 0 then begin
        parent := container[idxParent];
        parent.Items.Add(item);
        item.Parent := parent;
      end;
    end;
  end; //for iItem
end; { TGpLinearThreader.ReparentItems }
```

➤ *Listing3: Linear threading.*

solving in `LinkToExistingParent`). If a parent is not found in the list of already processed nodes, it will add the item to a temporary container and remember its location in another hash table so it can be quickly retrieved when the appropriate parent comes by. If you look at the implementation, you'll see that this approach is also on the order of *O(n*m)* (*n* and *m* being the same values as before).

When I benchmarked this algorithm, I couldn't believe my eyes. It sure beat my hashed implementation, threading 100,000 messages in under a second and 1,000,000 messages in 7.5 seconds (the hashed threader required more than 19 seconds to thread through a million messages).

I couldn't explain the difference so I fired up my trusty profiler (for details see http://gp.17slon.com/gpprofile) and applied it to the hashed threader. The result was truly surprising: about 50% of the time was spent creating and destroying `TInterfaceList` objects.

As it turns out, the hashed threader creates a `TInterfaceList` for each ID encountered, and then keeps them mostly empty. The reason is simple: when it encounters a leaf node (an item that will never be a parent to another item) it must still create an associated tree, because it doesn't know yet that this will be a leaf node. However, the only item ever appearing in this list will be the leaf node.

Besides that, a `TInterfaceList` is *ssssssslowwww*. Think molasses: that slow. Internally, it uses a `TThreadList` and locks/unlocks it (using a critical section) for each operation, which makes it safe, but really really (really!) slow. That is why `TGpThreadableContainer`, a simple implementation of `IGpThreadableContainer` from the unit GpThreadable.pas, is not really that simple. Instead of `TInterfaceList` it implements a storage using a dynamically resized array of `IGpThreadable` items.

KISS

Time to step back and think. Again. Obviously, Lee was onto something, it is much easier to connect a child to its parent than a parent to its children, simply because in the former case, there can be only one parent. The problem is that we don't always know the exact location of this parent at the moment when the child is being processed. By now you already know why: this happens when a child is encountered before its parent. To work around this problem, Lee's threader uses a temporary container and the hashed threader uses a bunch of `TInterfaceLists`.

There is another workaround. We can pretend that the problem doesn't exist. OK, that is not a good programming practice, but we can do the next best thing: we can make sure that it doesn't occur.

The solution is incredibly simple. First we calculate hashes for all the IDs in the container and store the location (the array index) of each ID in the hash table. Then we iterate over the container again and for each `ParentID` check the hash table. If the `ParentID` is stored in the hash table, the hash entry will contain an index to the parent and we can connect the item to it. If there is no such entry in the hash table, the parent doesn't exist and we can just leave the item in its place and treat it as a root. At the end, the third pass is done to remove all non-root items from the container, just as in the Lee's solution and the hashed threader. Three loops over the container are needed, that is all, and they are shown in Listing 3.

Finally, we have an algorithm of order *O(n)*. Still, it is not noticeably faster then Lee's algorithm. In fact, the measured numbers are almost the same, up to two million messages (after that, the measly 1Gb of memory in my computer can't hold the required data structures any more). Still, it is much simpler and as such is preferred. At least by me.

Our task is therefore complete. We now have a very fast tree generator. No, we have two! You can use whichever one you prefer to thread the data.

If you will be using this knowledge in the real world, I must warn you that things can be messier out there. At least check out www.jwz.org/doc/threading.html where real-life message threading (as used in the Netscape Navigator and IMAP servers) is discussed.

This article would never habe appeared without www.delphi-si.com, a guy called Lee_Nover (at least on the internet), Philip Glass, Keith Jarret and WA Mozart: thanks to them all!

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