# An Internal Application Message Bus in vb.net

Modern programming languages and multi-core CPUs offer very efficient multi-threading. Using multithreading can improve performance and responsiveness of an application, but working with threads is quite difficult as they can make things much more complicated. One way to organise threads so that they co-operate without tripping over each other is to use a messaging mechanism to communicate between them.

## Why use messages?

Messages provide a good model for communication between independent processes because we humans use them all the time. We naturally coordinate and co-operate by sending messages to each other. The messages can be synchronous (like a conversation) or asynchronous (like an email or a letter) or broadcast (like a radio programme). The messaging *paradigm* is easy for us to imagine and understand. In particular, it provides a natural way for us to think about how things interact. We can easily imagine a process that is responsible for a particular action, which is started when it receives a message. The message may contain information needed for the action. When the action is complete, the process can report the result by sending another message. We can imagine a simple, independent process, all it has to do is wait for the arrival of a message, carry out a task, and send a message saying it has finished. What could be simpler?

## What is a message bus?

A *message bus* provides a means of message transmission in which the sender and receiver are not directly connected. For example, Ethernet is a message bus – all senders put all their messages on the bus and, locally at least, all receivers receive every message.

When messages are sent on a bus, there needs to be a way for the receiver(s) to select the messages they need to process. There are various ways to do this, but for the message bus implemented in this article we allow a sender to label a message with a *sender role*, a *subject* and a *message type*. These may appear to be quite arbitrary properties, but they fit in with the way in which the bus is used and provide a straightforward way for receivers to filter messages so that they process only those messages that are relevant.

This form of messaging is usually referred to as a *Publish and Subscribe* model.

## How our bus will work

These are the essential characteristics of our message bus:

* The message bus operates within a single application, to send messages between independent worker threads.
* Any worker thread in the application can access the message bus.
* Any worker thread may send and receive messages using the bus.
* Messages are *broadcast ,* so every receiver that is listening will get every message.
* The bus does not store messages so a receiver will not get any messages that were sent before it connects to the bus.
* The thread that sends a message is separated from the thread(s) that receive it, so sending and receiving are always *asynchronous*.
* A receiver can set a filter to select only relevant messages for delivery – subscribing to a subset of the messages sent on the bus.
* Worker processes that send and receive messages are not held up by other worker threads when they do so. We want our senders and receivers to be working at their tasks without having to wait for messages to be delivered and processed by other threads.

# Classes of the message bus

These are the classes which make up the bus:

cBus

The base class of the bus and all the other classes. This class is never instantiated directly, but holds class (*Shared*) variables and methods that provide some core functions of the bus.

cBusLink

A component that provides the mechanism for delivering messages to receivers.

cThread

A component that provides and controls a thread for use within the sender and receiver classes.

cSender

The class that is used by senders to put messages into the Message bus. Each worker process that sends messages uses a cSender object.

cReceiver

The class that is used by worker processes to subscribe and take delivery of messages from the bus.

cFilter

Class used to apply subscription filters to incoming messages within a cReceiver.

cMessage

Objects of this class are sent and received. In our system the message content is a string, but the class could be extended through inheritance to provide richer content.

## cBus and cBusLink – the core of the message bus

CBus is the base class for all the other classes in the implementation. cBus is a *virtual* class – it is never itself instantiated. It contains only one class member, *oBusLink,* a shared instance of **cBusLink**. oBusLink is *protected*, which means it is accessible only to derived (child) classes of cBus.

**cBus** and **cBusLink**, which are central to the whole message bus, are very simple (see Listing 1).

**Listing 1 – cBus and cBusLink classes**

Public Class cBus

'// ///////////////////////////////////////

'// The BusLink class is used only as a means of

'// propagating publication of a message from

'// senders to receivers.

Protected Class cBusLink

'// Event published with new message

Public Event NewMessage(ByVal oMessage As cMessage)

'// Event published when bus is stopped

Public Event StopBus()

'// Flag to indicate that the bus has been

'// stopped. Provides orderly shutdown

Private bStopped As Boolean = False

'// Method to publish a message

Public Sub PublishMessage(ByVal oMessage As cMessage)

If bStopped Then Exit Sub

RaiseEvent NewMessage(oMessage)

End Sub

'// Method to stop the bus, for orderly shutdown

Public Sub StopBusNow()

bStopped = True

RaiseEvent StopBus()

End Sub

End Class

'// Global shared single instance of cBusLink

'// used to send messages to all receivers

Protected Shared oBusLink As New cBusLink

'// Global shared flag indicating the bus has

'// been stopped

Protected Shared bStopped As Boolean = False

'// ///////////////////////////////////////

'// ID generator is used by other classes to

'// generate unique sequence numbers

Protected Class cIDGenerator

Private \_ID As Long = 0

Public Function NextID() As Long

\_ID += 1

Return \_ID

End Function

End Class

'// ////////////////////////////////////

'// Public method to stop the bus before

'// closedown. Ensures orderly closedown.

Public Shared Sub StopBusNow()

bstopped = True

oBusLink.StopBusNow()

End Sub

End Class

The class cBusLink is at the core of the message bus and is responsible for delivering messages to every recipient through the *NewMessage* event. As we shall see later, every cReceiver object holds a reference to a single shared cBusLink object and they all subscribe to its *NewMessage* event. When this event is fired, every cReceiver object is given a reference to the new message.

## cMessage

Objects of the **cMessage** class carry the message data from sender to recipient. In our implementation, the class has only a single string payload, see Listing 2 – but you can implement sub-types of cMessage with additional properties and methods for more sophisticated communication between senders and receivers.

**Listing 2 – cMessage class**

Public Class cMessage

Inherits cBus

'// /////////////////////////////////

'// This class is a container for allocating

'// unique message ids to each mec

Private Shared \_oMsgID As New cIDGenerator

'// Properties of the message, accessible to derived

'// classes

Protected \_SenderRole As String = ""

Protected \_SenderRef As String = ""

Protected \_Subject As String = ""

Protected \_Type As String = ""

Protected \_Content As String = ""

'// Message ID is private, it cannot be changed,

'// even by derived classes

Private \_MsgID As Long

'// /////////////////////////////

'// Default constructor used only for

'// derived classes

Protected Sub New()

\_MsgID = \_oMsgID.NextID

End Sub

'// /////////////////////////////

'// Public constructor requires key message

'// properties to be supplied. The message

'// cannot be modified thereafter.

Public Sub New(ByVal Sender As String, \_

ByVal Subject As String, \_

ByVal Type As String, \_

Optional ByVal Content As String = "")

\_SenderRole = Sender

\_Subject = Subject

\_Type = Type

\_Content = Content

\_MsgID = \_oMsgID.NextID

End Sub

'// /////////////////////////////////////////////////

'// Property accessors - all read-only so values

'// cannot be changed by any recipient.

Public ReadOnly Property SenderRole() As String

Get

Return \_SenderRole

End Get

End Property

Public ReadOnly Property Subject() As String

Get

Return \_Subject

End Get

End Property

Public ReadOnly Property Type() As String

Get

Return \_Type

End Get

End Property

Public ReadOnly Property MsgID() As Long

Get

Return \_MsgID

End Get

End Property

Public ReadOnly Property Content() As String

Get

Return \_Content

End Get

End Property

'//

'////////////////////////////

End Class

This class implementation is mostly straightforward, but some aspects are worth looking at more closely:

* The class inherits **cBus** to gain access to the protected class **cIDGenerator** which is declared in the base class.
* All the variables that store property values, except for MsgID, are declared *Protected* so that they can be accessed within in a child class. MsgID is declared *Private* so its value cannot be changed by a child class.

## cSender

**cSender** and its counterpart **cReceiver** do all the hard work. **cSender** is the class used by a worker thread to add messages to the bus. Before we look under the hood, let’s examine the public members of the class that a sending process will use.

*Using the cSender class*

First, a worker process that wants to send messages must instantiate an instance of cSender, providing the *sender’s role* as a parameter. The role allows for the possibility that there might be multiple worker threads performing the same role within the application. A recipient can filter messages based on the role of the sender, but does not need to know that there is more than one sender acting in that role.

**:**

Dim oSender as New cSender("clock")

:

Once instantiated, the cSender object can be used to send messages on the bus:

:

Dim oMsg as New cMessage("time", "hourchange", "10>11")

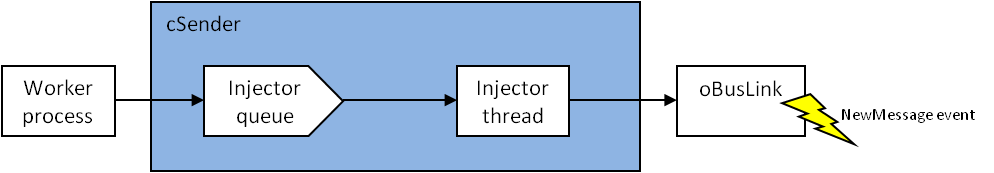
oSender.SendMessage oMsg

:

In this case, the message has the type "time", the subject "hourchange" and the content "10>11".

*Under the hood of the cSender class*

The cSender implementation uses a queue to separate the sender process from the bus. When the worker thread sends a message it is written to the *injector queue*, from where it is picked up by a separate injector thread and published through the bus link:



The injector runs on a separate thread, so that placing a message on the bus does not hold up the worker process. The injector thread is provided by a cThread object which runs only when messages are waiting in the injector queue. The implementation of the class is shown in Listing 3.

**Listing 3 – cSender Class**

Public Class cSender

Inherits cBus

'// //////////////////////////////////////////

'// Queue of messages waiting to be injected

'// into the message bus. Each sender has its

'// own private injector queue

Private \_oMsgQ As New System.Collections.Generic.Queue(Of cMessage)

'// /////////////////////////////////////////

'// Reference to the global BusLink instance, used

'// only to pick up the BusStopped event published

'// by the bus when stopped.

Private WithEvents oMyBusLink As cBusLink

'// /////////////////////////////////////////

'// Event to inform owner the bus has stopped

Public Event Stopped()

'// Sender role, used to identify the sender and

'// provide the key for filtering messages

'// at the receiver.

Private \_Role As String

Public ReadOnly Property Role() As String

Get

Return \_Role

End Get

End Property

#Region "Construct and destruct"

'// //////////////////////////////////////////

'// Constructor with role (mandatory)

Public Sub New(ByVal sRole As String)

\_Role = sRole

'// Set the reference to the buslink to the

'// shared instance of the single buslink. We

'// need this reference to pick up the stop event

oMyBusLink = oBusLink

End Sub

'// //////////////////////////////////////////////

'// This method is called when the bus is closed down

Private Sub oBusLink\_StopBus() Handles oMyBusLink.StopBus

SyncLock \_oMsgQ

RaiseEvent Stopped()

End SyncLock

End Sub

#End Region

#Region "Sending messages"

'// /////////////////////////////////////////

'// Method used by worker thread to place a

'// new default cMessage object on the injector

'// queue.

Public Function SendNewMessage(ByVal Type As String, \_

ByVal Subj As String, \_

Optional ByVal Ref As String = "", \_

Optional ByVal Content As String = "") As cMessage

If BusStopped Then Return Nothing

Dim oM As New cMessage(\_Role, Type, Subj, Ref, Content)

SendMessage(oM)

Return oM

End Function

'// //////////////////////////////////////////

'// Method used by worker thread to place message

'// object on the injector queue.

Public Sub SendMessage(ByVal pMessage As cMessage)

If BusStopped Then Exit Sub

'// We do not allow Nothing to be sent

If pMessage Is Nothing Then

'// Do nothing

'// We could throw an error here

Else

SyncLock \_oMsgQ

\_oMsgQ.Enqueue(pMessage)

'// Start the thread only if

'// one message on the queue.

If \_oMsgQ.Count = 1 Then

\_oInjectorThread.Start()

End If

End SyncLock

End If

End Sub

'// ////////////////////////////////////////

'// Holds up the caller thread until all the messages

'// have been injected into the bus

Public Sub Flush()

Do Until \_oMsgQ.Count = 0

Threading.Thread.Sleep(2)

Loop

End Sub

#End Region

#Region "Message Injector"

'// //////////////////////////////////////////

'// Functions run by the thread for injecting messages

'// into the bus. The thread runs only when at

'// least one message is waiting in the injector queue.

Private WithEvents \_oInjectorThread As New cThread

'// //////////////////////////////////////////

'// Injector Thread fires Run event to place

'// messages on the queue

Private Sub \_oInjectorThread\_Run() Handles \_oInjectorThread.Run

InjectMessagesNow()

End Sub

'// ///////////////////////////////////////////

'// When the injector thread runs, this function

'// is called to push all the queued messages into

'// the bus.

Private Sub InjectMessagesNow()

Dim oM As cMessage

'// Loop until all messages in the

'// queue have been injected into the

'// bus.

Do

'// Check if stopped flag was set while

'// going round loop.

If BusStopped Then Exit Sub

'// Get the next message off the

'// injector queue

SyncLock \_oMsgQ

If \_oMsgQ.Count > 0 Then

oM = \_oMsgQ.Dequeue()

Else

oM = Nothing

End If

'// Release the lock so that the worker

'// process can add new messages to

'// the queue while we are publishing

'// this message on the bus

End SyncLock

If oM Is Nothing Then

'// Queue is empty, so finish the

'// loop

Exit Do

End If

'// Now we have got the message, we can

'// send it using the single global

'// cBusLink which is instantiated in the

'// base class cBus.

SyncLock oBusLink

oBusLink.PublishMessage(oM)

End SyncLock

Loop

End Sub

#End Region

Protected Overrides Sub Finalize()

'// Close down the injector thread

\_oInjectorThread.StopThread()

MyBase.Finalize()

End Sub

End Class

The method *SendMessage* is used by a worker process to place messages on the injector queue. The queue class is not threadsafe, so *SyncLock* is used to protect the queue from simultaneous use by another thread. The injector thread is started only when a message is added to an empty queue, and this fires the event *cThread.Run*.

The private method *\_oInjectorThread\_Run* handles the injector thread *Run* event. The method takes all the waiting messages from the injector queue, placing them in turn on the bus by using the BusLink's PublishMessage method. When the method exits, the thread is blocked in within cThread until another message is placed on the empty queue. If a message is added to the injector queue while an earlier message is being sent on the bus, it will be included in the sending loop without needing the *Run* event to fire again.

## cReceiver

Objects of this class are used by worker processes to receive messages from the bus.

The process that creates the cReceiver object can choose to set filters so that only relevant messages are delivered. More detail on filtering is given below.

When the receiver object connects to the bus, it sets its own private member variable \_BusLinkRef to refer to the shared member oBusLink. \_BusLinkRef is declared *WithEvents* so that the *NewMessage* event of the cBusLink can be handled.

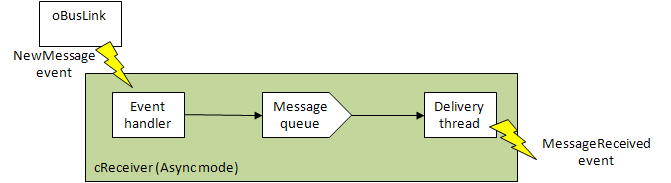
The thread that owns the receiver can set a cFilter object on the receiver. Then every message received through the *NewMessage* event is checked against the filter and, if it passes, it is added to the receiver's incoming message queue, waiting to be delivered. The filter can be changed during the run.

Messages are delivered and processed in one of three ways:

* The worker thread calls *GetNextMessage* to return the next message from the queue. If there are no messages waiting, the method returns **Nothing**.
* The worker thread calls *DeliverMessages* to deliver all queued messages through the *MessageReceived* event. The events are raised on the worker thread.
* The creator/owner calls the *StartAsync* method to request that the receiver object provides a separate worker thread to raise the *MessageReceived* event, when new messages arrive. The event is raised on a thread provided by a cThread object within cReceiver.

Using *GetNextMessage* or *DeliverMessages* means that the receiver worker thread must set up its own processing loop, for example by having its own timer to repeat the loop. This is appropriate when, for example, the thread needs to interact with the GUI – using a Timer component on a form could provide the thread.

In contrast, using *StartAsync* means that the cReceiver object will create its own internal worker thread that raises the *MessageReceived* event.



Listing 4 – cReceiver class

Public Class cReceiver

Inherits cBus

'// //////////////////////////////////////

'// Id generator for all cReceiver objects

Private Shared \_oRecId As New cIDGenerator

'// //////////////////////////////////////

'// Event used to deliver a message to the

'// message handler function

Public Event MessageReceived(ByVal oMessage As cMessage)

'// //////////////////////////////////////

'// Event used to indicate the bus has stopped,

'// used to ensure orderly shutdown of the bus

Public Event Stopped()

Public ReadOnly Property IsStopped() As Boolean

Get

Return BusStopped

End Get

End Property

'// //////////////////////////////////////

'// Message queue holding the messages

'// waiting to be delivered

Private \_MQueue As New System.Collections.Generic.Queue(Of cMessage)

'// ///////////////////////////////////////////

'// Filter set by the recipient to select

'// messages. Fileter can be by specific role(s),

'// subjects(s) or type(s) or using more specialised

'// filters. Filters can be changed at any time. The

'// default no filter allows all messages through.

Public Filter As cFilter = Nothing

'// //////////////////////////////////////////

'// Reference to the single global buslink

'// so that the receiver can pick up published

'// messages from the bus

Private WithEvents \_BusLinkRef As cBusLink

'// Flag to indicate that this object has been

'// finalised and is closing.

Private \_Closing As Boolean = False

Private \_RaiseStopEvent As Boolean = False

'// /////////////////////////////////////////

'// Unique identifier of this receiver object

Private \_ID As Long

'// /////////////////////////////////////////

'// Counts of number of messages received

'// and delivered

Private \_BCount As Long = 0 ' Messages from the Bus

Private \_RCount As Long = 0 ' Messages received onto the queue

Private \_DCount As Long = 0 ' Messages delivered to the worker

'// //////////////////////////////////

'// Constructor

Public Sub New()

\_ID = \_oRecId.NextID

End Sub

'// ///////////////////////////////////

'// Establishes connection to the bus so that

'// message delivery can start

Public Sub Connect()

'// /////////////////////////////////////////

'// Set the buslink variable to refer to the

'// shared buslink so that it delivers

'// messages through the event handler

\_BusLinkRef = oBusLink

'// NOTE: oBus is a direct reference to

'// the protected shared class member.

End Sub

'// ////////////////////////////////////////

'// Breaks the connection with the bus

'// so that messages are no longer

'// received.

Public Sub Disconnect()

\_BusLinkRef = Nothing

End Sub

'// /////////////////////////////////

'// Accessor methods for the readonly

'// properties

Public ReadOnly Property BCount() As Long

'// Bus message count

Get

Return \_BCount

End Get

End Property

Public ReadOnly Property RCount() As Long

'// Received message count

Get

Return \_RCount

End Get

End Property

Public ReadOnly Property DCount() As Long

'// Delivered message count

Get

Return \_DCount

End Get

End Property

Public ReadOnly Property QCount() As Long

'// Queued (waiting) message count

Get

If \_MQueue IsNot Nothing Then

Return \_MQueue.Count

Else

Return 0

End If

End Get

End Property

Public ReadOnly Property ID() As Long

'// Unique ID number of this receiver

Get

Return \_ID

End Get

End Property

Public Function MessagesWaiting() As Boolean

'// Helper property returns true if there

'// are messages waiting

Return QCount > 0

End Function

#Region "Message arrival"

'// //////////////////////////////////

'// This method handles the new message

'// event from the bus. The message is

'// queued for delivery.

Private Sub oBusLink\_NewMessage( \_

ByVal oMessage As cMessage \_

) Handles \_BusLinkRef.NewMessage

'// Discard message if closing, or the bus has stopped

If \_Closing Then Exit Sub

If BusStopped Then Exit Sub

\_BCount += 1

'// ////////////////////////////

'// Check against the filter.

'// The message must be included by the filter

'// otherwise it will not be delivered.

Select Case True

Case Filter Is Nothing, Filter.bInclude(oMessage)

'// ///////////////////////////////

'// New message has passed the filter, so

'// add it to the message queue waiting

'// for delivery to the worker process.

AddToQueue(oMessage)

End Select

End Sub

'// ////////////////////////////////

'// Method used to add messages

'// to the message queue when they arrive

'// from the message bus.

Private Sub AddToQueue(ByVal oMessage As cMessage)

'// ////////////////////////////////////////////

'// Check if the queue exists - if not, then

'// exit without adding a message.

If \_MQueue Is Nothing Then Exit Sub

'// ////////////////////////////////////////////

'// Check if closing or stopped, if so exit

If BusStopped Then Exit Sub

If \_Closing Then Exit Sub

Dim bStartDelivery As Boolean

'// ////////////////////////////////////////////

'// SyncLock the queue to guarantee exclusive

'// access, then add the message

SyncLock \_MQueue

\_RCount += 1

\_MQueue.Enqueue(oMessage)

'// ////////////////////////////////////////////////

'// We start the delivery thread if async AND

'// this is the first message in the queue

bStartDelivery = \_AsyncMode And \_MQueue.Count = 1

End SyncLock

'// //////////////////////////////

'// Check if we need to start the delivery thread

'// which we do only in async mode and if this is

'// the first message in the queue

If bStartDelivery Then

\_DeliveryThread.Start()

End If

End Sub

#End Region

#Region "Message delivery"

'// ////////////////////////////////

'//

'// Message delivery can be made in these

'// ways:

'// \* Asynchronously on a provided thread

'// - call StartAsync to enable this

'// - messages are delivered through MessageReceived event

'//

'// \* By a call from the worker thread

'// - use GetNextMessage to retrieve the message

'//

'// GetNextMessage returns the next

'// message as the function result.

'// It returns Nothing if

'// there is no message in the queue

'//

'// ////////////////////////////////

'// Delivery thread is used with asynch delivery only

Private WithEvents \_DeliveryThread As cThread = Nothing

Private \_AsyncMode As Boolean = False

'////////////////////////////////////

'// Starts Asynchronous delivery through the NewMessage event.

'// Called by the creator/owner to initiate a new thread delivering

'// messages from this receiver.

Public Sub StartAsync()

'// Do nothing if closing, stopped or already in asyinc mode.

If \_Closing Then Exit Sub

If BusStopped Then Exit Sub

If \_AsyncMode Then Exit Sub

\_AsyncMode = True

'// Create and start the delivery thread.

If \_DeliveryThread Is Nothing Then \_DeliveryThread = New cThread

\_DeliveryThread.Start()

End Sub

'// ///////////////////////////////////////////////

'// Picks up the next message from the queue

'// if any and returns it. Returns Nothing

'// if there is no message.

Public Function GetNextMessage() As cMessage

'// Do not return anything if closing or stopped

If \_Closing Then Return Nothing

If BusStopped Then Return Nothing

Dim oM As cMessage

'// Lock the queue and get the next message

SyncLock \_MQueue

If \_MQueue.Count > 0 Then

oM = \_MQueue.Dequeue

\_DCount += 1

Else

oM = Nothing

End If

End SyncLock

'// Return the message (if any)

Return oM

End Function

'// ///////////////////////////////////////////////

'// This event handler is called when the thread runs

'// - only when messages are waiting to be delivered in

'// async mode

Private Sub \_DeliveryThread\_Run() Handles \_DeliveryThread.Run

DeliverWaitingMessages()

End Sub

'// ///////////////////////////////////////////////

'// Delivers all the messages in the incoming

'// message queue using the MessageReceived event

Public Sub DeliverWaitingMessages()

'// Raise the stop event if the bus has been stopped

If BusStopped Then

'// Inform the delivery thread

If \_RaiseStopEvent Then

RaiseEvent Stopped()

\_RaiseStopEvent = False

End If

Exit Sub

End If

'// Do nothing if closing

If \_Closing Then Exit Sub

'// The queue may be nothing , so simply

'// exit and try again on the cycle

If \_MQueue Is Nothing Then Exit Sub

Dim oM As cMessage

'// Retrieve all the messages and deliver them

'// using the message received event.

Do

'// Lock the queue before dequeuing the message

SyncLock \_MQueue

If \_MQueue.Count > 0 Then

oM = \_MQueue.Dequeue

Else

oM = Nothing

End If

End SyncLock

'// ///////

'// After releasing the lock we

'// can deliver the message.

If oM IsNot Nothing Then

\_DCount += 1

RaiseEvent MessageReceived(oM)

End If

'// If the queue was not empty then loop back for the

'// next message

Loop Until oM Is Nothing

End Sub

#End Region

#Region "Stats Report"

'////////////////////////////////////////////////

'// This sub simply publishes a message of

'// stats about this receiver.

Public Sub StatsReport()

If BusStopped Then Exit Sub

Dim sRpt As String

sRpt = "Report from Receiver #" & Me.ID

sRpt &= "|BUS=" & \_BCount

sRpt &= "|REC=" & \_RCount

sRpt &= "|DEL=" & \_DCount

sRpt &= "|Q=" & \_MQueue.Count

sRpt &= "|Closing=" & \_Closing

Dim s As New cSender("Receiver#" & ID)

s.SendNewMessage("STATS", "STATS", sRpt)

s.Flush()

s = Nothing

End Sub

#End Region

'// ///////////////////////////////////

'// Handler for the stopbus event. Do

'// not deliver any more messages once the

'// bus has been stopped.

Private Sub oBusLinkRef\_StopBus() Handles \_BusLinkRef.StopBus

\_Closing = True

'\_DeliveryTimer = Nothing

\_AsyncMode = False

\_RaiseStopEvent = True

End Sub

'// ////////////////////////////////////

'// Finalise to tidy up resources when being disposed

Protected Overrides Sub Finalize()

\_DeliveryThread.StopThread()

\_Closing = True

\_AsyncMode = False

\_MQueue = Nothing

MyBase.Finalize()

End Sub

End Class

## cThread

The cThread class provides a thread and the control methods needed to block and release the thread as required.

By default, the thread is blocked. The class provides a method, *Start()*, which unblocks the thread. The thread immediately raises the *Run* event to carry out the processing required, and then blocks again until the *Start* method is called again, when it repeats the *Run* event.

In our message bus, cThread is used in cSender to inject messages onto the bus, and in cReceiver to deliver messages, when operating in Async mode. In both of these classes the *Run* event handler picks messages off a queue until it is empty, then exits. It is quite likely that new messages are added to the queue while the handler is running, and these are picked up in the handler loop. Eventually, the queue is empty and if *Start* has not been called again, the thread blocks until it is.

The implementation of the class is shown in Listing 5.

Listing 5 – cThread class

Public Class cThread

Inherits cBus

Private WithEvents \_BusLinkRef As cBusLink = oBusLink

Private Shared iThreadCount As Long = 0

'// Event fired to execute the thread's

'// assigned processes.

Public Event Run()

'// Thread object provides the thread

Private \_Thread As New Thread(AddressOf RunThread)

'// Signal object to block the thread

'// when there are no messages to be delivered

Private \_Signal As New EventWaitHandle(False, EventResetMode.AutoReset)

'// Flag to indicate thread has been stopped

Private bThreadStopped As Boolean = False

'// Start the thread on creation of the object

Public Sub New()

\_Thread.Start()

End Sub

'// Start called by owner to

'// unblock this thread.

Public Sub Start()

If \_Thread.ThreadState = ThreadState.Unstarted Then \_Thread.Start()

SyncLock Me

\_Signal.Set()

End SyncLock

End Sub

'// Stop called by owner to close

'// down thread

Public Sub StopThread()

bThreadStopped = True

\_Signal.Set()

End Sub

'// Method executed by the thread. This is

'// a repeated loop until the bus is stopped

Private Sub RunThread()

Do

'// The signal blocks the thread until

'// it is released by the Start method

\_Signal.WaitOne()

If bThreadStopped Then

Exit Sub

End If

'// Raise the thread event that will

'// do the work.

RaiseEvent Run()

Loop

End Sub

Private Sub \_BusLinkRef\_StopBus() Handles \_BusLinkRef.StopBus

StopThread()

End Sub

End Class

## cFilter

cFilter objects are used by cReceiver to apply filtering to incoming messages. The base cFilter class is declared *Must Override* so cannot be instantiated. It is only by defining a child class to apply some filtering logic that messages get filtered. This is how it works:

* The base class, cFilter defines a *Protected Must Override* method *bMatches*, which takes a Message object as a parameter. In a child class this method is overridden to implement specific filtering logic.
* cFilter defines a *Public* method, *bInclude*, which takes a message object as a parameter and returns true if the message is to be included, and false if not. This is the method used by cReceiver to check if a message passes the filter. Apart from testing its own *bMatches*() value, this method also contains the logic to check other cFilter objects that have been attached in And / Or collections.
* Four further methods, *And\_()*, *And\_Not()*, *Or\_()* and *Or\_Not()* provide the means to add other filter objects to the And/Or collections of this filter.

Using the *And\_()* and *Or\_()* etc. methods makes it easy to build compound logical conditional tests using basic filter components. For example, if I have two filter objects FilterA and FilterB, they can be combined as FilterA.Or\_(FilterB), or FilterA.And\_(FilterB). It is also possible to combine several chains of filters. For example, FilterA.And\_(FilterB.Or\_Not(FilterC)) implements the filter condition **A and (B or not C)**.

#### Actual filtering classes implemented

Various specialised classes of cFilter are implemented to provide filtering on sender role, message type and subject. These include, for example, cRoleEquals, cTypeEquals and cSubjectEquals. As their names suggest, these filters check that the key fields of the message match a given string.

A worker process that uses cReceiver can apply filters to the incoming message simply by setting the Filter property of the receiver:

:

Dim oReceiver as new cReceiver

oReceiver.Filter = new cRoleEquals("monitor")

:

#### Inside cFilter and its derived classes

The base *cFilter* class defines the protected *MustOverride* method *bMatches*. The derived classes override *bMatches*, providing the appropriate code to determine the match. For example, in the case of the cSubjectContains class, the overriding bMatches method is:

:

Public Overrides Function bMatches(ByVal oMessage As cMessage) As Boolean

Return oMessage.Subject.Contains(FilterString)

End Function

:

If you need to have a more specialised filtering mechanism in your application, it is easy to define a derived class of cFilter that implements whatever logic you need in *bMatches.*

Listing 5 – cFilter class and derived classes

'// The filter base class is used to implement

'// message filtering on incoming messages

'// at each receiver. Filters can be grouped in

'// AND and OR groups - the message is

'// included if it matches all filters in the

'// AND group or any filter in the OR group.

Public MustInherit Class cFilter

Inherits cBus

'// A collection of filters which this filter must AND

'// with to allow the message through

Private oAnds As New System.Collections.Generic.List(Of cFilter)

'// A collection of filters which this filter must OR

'// with to allow the message through

Private oOrs As New System.Collections.Generic.List(Of cFilter)

'// Check if the message is included by this filter

Public Function bInclude(ByVal oMessage As cMessage) As Boolean

Dim bResult As Boolean

'// First, test this filter alone

bResult = bMatches(oMessage)

Dim oFF As cFilter

'// If this filter matches, then check all the

'// ANDs to see if they also match

If bResult Then

For Each oFF In oAnds

bResult = oFF.bMatches(oMessage)

'// As soon as we find the first failure to

'// match we know the result is a non-match

'// for this filter and all its ANDs

If Not bResult Then Exit For

Next

End If

'// If all the ANDS were true, then the whole result

'// is true regardless of the OR result.

If bResult Then Return True

'// The ANDs did not match, so now

'// we find if any one OR matches, and if so

'// the result is true

For Each oFF In oOrs

bResult = oFF.bInclude(oMessage)

If bResult Then Return True

Next oFF

'// No match on any of the ORS, so

'// the message does not match this filter

Return False

End Function

'// ///////////////////////////////////

'// This method must be overridden in child

'// classes to implement the matching test.

Protected MustOverride Function bMatches( \_

ByVal omessage As cMessage) As Boolean

'// ///////////////////////////////////

'// These methods add a given filter to the

'// ANDs or ORs collections to build filtering

'// logic.

Public Function And\_(ByVal oFilter As cFilter) As cFilter

oAnds.Add(oFilter)

Return Me

End Function

Public Function Or\_(ByVal ofilter As cFilter) As cFilter

oOrs.Add(ofilter)

Return Me

End Function

Public Function Or\_Not(ByVal ofilter As cFilter) As cFilter

oOrs.Add(Not\_(ofilter))

Return Me

End Function

Public Function And\_Not(ByVal oFilter As cFilter) As cFilter

oAnds.Add(Not\_(oFilter))

Return Me

End Function

'//

'// ///////////////////////////////////////

'// ///////////////////////////////////////

'// Class and function to provide negation

'// of a filter condition

Private Class cNot

Inherits cFilter

Private oNotFilter As cFilter

Public Sub New(ByVal oFilter As cFilter)

oNotFilter = oFilter

End Sub

Protected Overrides Function bMatches(ByVal omessage As cMessage) As Boolean

Return Not oNotFilter.bMatches(omessage)

End Function

End Class

Private Function Not\_(ByVal oFilter As cFilter) As cFilter

Return New cNot(oFilter)

End Function

'//

'// /////////////////////////////////////////////

End Class

#Region "Filter implementations"

'// /////////////////////////////////////////

'// Derived specialised classes for implementing

'// different specific filters.

Public Class cTypeContains

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.Type.Contains(FilterString)

End Function

End Class

Public Class cTypeEquals

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.Type = FilterString

End Function

End Class

Public Class cRoleContains

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.SenderRole.Contains(FilterString)

End Function

End Class

Public Class cRoleEquals

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.SenderRole = FilterString

End Function

End Class

Public Class cSubjectContains

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.Subject.Contains(FilterString)

End Function

End Class

Public Class cSubjectEquals

Inherits cFilter

Public FilterString As String

Public Sub New(ByVal sFilter As String)

FilterString = sFilter

End Sub

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.Subject = FilterString

End Function

End Class

Public Class cRoleTypeSubjectFilter

Inherits cFilter

Public sRole As String = ""

Public sType As String = ""

Public sSubject As String = ""

Protected Overrides Function bMatches( \_

ByVal oMessage As cMessage) As Boolean

Return oMessage.Type = sType \_

And oMessage.SenderRole = sRole \_

And oMessage.Subject = sSubject

End Function

End Class

'//

'///////////////////////////////////////////////

#End Region

# A demo application

The demo application included in the zip file is a simple windows forms application that includes a number of components that communicate with each other via the MessageBus:

* The main control form provides buttons for opening the other form types
* A mouse tracker form, that monitors mouse movements over the form and sends mouse movement messages on the bus
* A clock object that sends a time message whenever the time ticks past a tenth of a second, a second, a minute or an hour.
* A mouse follower form, that monitors mouse movement messages and positions a red box on the form at the position indicated by the message, and also displays the time as sent out by the clock object.
* A message sender form, which can generate bus messages of different types a frequency set by the user
* A message receiver form, that lists messages received, filtered on attributes set by the user

The user can open as many sender forms, receiver forms and mouse follower forms as they wish, and can set the message types to be sent and received. Each of the forms operates independently of the others.

**Word counts**

Intro: 577

Classes intro: 159

cBus/cBusLink: 75+65=140

cMessage: 48+74=122

sSender: 272+147=419

cReceiver: 305

cThread: 167

cFilter: 405

demo app: 185

**Total: 2479**