Guide to Implementing Custom TPL Dataflow Blocks

Zlatko Michailov

Microsoft Corporation

1. Introduction

TPL Dataflow comes with a set of built-in blocks and a variety of extensibility options with a different degree of flexibility. That way, develepers can choose the right balance between flexibility and complexity.

The simplest form of creating a custom block is encapsulating existing blocks using the static DataflowBlock.Ecapsulate[1] method which allows you to treat a network of two or more blocks as a single block.

Alternatively, the most flexible, yet most complex, way to create a custom block is to implement the necessary block interfaces - ITargetBlock<TInput>, ISourceBlock<TOutput>, or

[1] All types are assumed to reside in the System.Threading.Tasks.Dataflow namespace unless specified otherwise.

IPropagatorBlock<TInput, TOutput> (which combines the former two interfaces.) If you choose to implement ISourceBlock<TOutput>, you might consider implementing IReceivableSourceBlock<TOutput> which would allow external code to efficiently receive messages from the source block without having to link a special-purpose target. Some aspects of a block's implementation may be done by wrapping one of the built-in blocks, or these interfaces could be implemented entirely from scratch.

2. Prerequisites

In order to compile and run TPL Dataflow-based applications, you need .NET 4.5 or the .NET 4-based CTP DLL. In your project, you have to reference the System.Threading.Tasks.Dataflow.dll assembly. You may import the System.Threading.Tasks.Dataflow namespace in your source files for convenience.

3. Encapsulation

Dataflow block encapsulation is a feature that allows you to create a new virtual dataflow block that consists of two or more existing blocks. This comes in handy when you already have most of the desired functionality implemented by an existing block or a network of blocks, and you want to reuse it with some minor augmentation.

Encapsulation is done through the static Encapsulate method of the DataflowBlock utility class:

```
public static class DataflowBlock
{
    public static IPropagatorBlock<TInput, TOutput> Encapsulate<TInput, TOutput>(
            ITargetBlock<TInput> target, ISourceBlock<TOutput> source)

}
```

What this method requires is two existing blocks - a target block (an instance of a class that implements ITargetBlock<TInput>) and a source block (an instance of a class that implements ISourceBlock<TOutput>). What Encapsulate does is create a new instance of an internal class that wires the ITargetBlock<TInput> interface members to the target parameter and the ISourceBlock<TOutput> interface members to the source parameter.

There is one subtlety – both ISourceBlock<TOutput> and ITargetBlock<TInput> derive from IDataflowBlock. How are the IDataflowBlock interface members wired? The rule of thumb throughout TPL Dataflow is that block completion should be explicitly passed from sources to targets. Therefore the IDataflowBlock.Complete() and IDataflowBlock.Fault() methods are wired to target while the IDataflowBlock.Completion property is wired to source.

Note: It is up to you as the user of Encapsulate to ensure that when your target half completes, the source half gets completed in whatever manner is most appropriate. For example:

```
target.Completion.ContinueWith(completion => source.Complete());
```

Or, if you want to propagate the completion type, you can use this more sophisticated form:

```
target.Completion.ContinueWith(completion =>
{
    if (completion.IsFaulted)
```

```
        ((IDataflowBlock)batchBlock).Fault(completion.Exception);
    else
        batchBlock.Complete();
});
```

Another important thing you have to explicitly provision is the message propagation from target to source. The benefit of this explicit wiring is that you have the freedom to perform any unconstrained processing between the two encapsulated blocks. You may do that either by encoding the necessary processing into the blocks' delegates if the blocks take delegates, or by embedding a sub-network of blocks between them. The easier way is to use a block that take delegates, e.g. ActionBlock, TransformBlock, TransformManyBlock (if applicable), or a custom block.

4. Embedding

Embedding a TPL Dataflow block is essentially implementing TPL Dataflow interfaces while replacing some of the body code with a call to the embedded block. Thus this topic is covered by the next section – Interface Implementation.

5. Interface Implementation

5.1. Guidelines for Implementing TPL Dataflow Interfaces

When choosing whether to implement a member implicitly or explicitly, the following rule applies: *If a member is meant to be used by general code (as opposed to block implementations), it should be implemented implicitly*. Such members are:

- IDataflowBlock
  - Completion
  - Complete
- ISourceBlock<TOutput>
  - LinkTo
- IReceivableSourceBlock<TOutput>
  - TryReceive
  - TryReceiveAll

Alternatively, *if a member is meant to be used by block implementations, it should be implemented explicitly*. Such members are:

- IDataflowBlock
  - Fault
- ITargetBlock<TInput>
  - OfferMessage
- ISourceBlock<TOutput>
  - ReserveMessage
  - ConsumeMessage
  - ReleaseReservation

5.2. Block Threading

In general, blocks should be *asynchronous*, i.e. when a block is being offered a message it should return control to the calling source as soon as possible. Any processing of the message as well as

further propagation down the network should be scheduled on other threads. Those threads may be dedicated, individually created, or reused from the framework's thread pool.

There are cases, however, when a block may reuse the calling source's thread to *synchronously* process and propagate offered messages. Synchronous blocks are much easier to implement from scratch than asynchronous blocks.

5.3. Implementing a Synchronous Filter Block

A synchronous filter block intercepts the traffic to a single given target and invokes a predicate for each message. If the predicate passes, the message is synchronously forwarded to the target. Otherwise DataflowMessageStatus.Declined is returned to source.

This block has several constraints that we can take advantage of in order to simplify the block's implementation:

a. There is exactly one target linked to this block at any given moment and that target never changes.
b. The messages are not transformed, i.e. the type of the output messages is the same as the type of the input messages.
c. [Because of b.] This block can forward the original source reference, i.e. the synchronous propagator need not pretend to be a source.

Our block can be outlined as this:

```
    public sealed class SyncFilterTarget<TInput> : ITargetBlock<TInput>
    {
        /// <summary>The ITargetBlock{TInput} to which this block is
propagating.</summary>
        private readonly ITargetBlock<TOutput> m_target;

        /// <summary>The predicate.</summary>
        private readonly Func<TInput, bool> m_predicate;


        /// <summary>
        /// Constructs a SyncFilterTarget{TInput}.
        /// </summary>
        /// <param name="target">Target block to propagate to.</param>
        /// <param name="predicate">Predicate.</param>
        public SyncFilterTarget(ITargetBlock<TInput> target, Func<TInput, bool>
predicate)
        {
            if (target == null) throw new ArgumentNullException("target");
            if (predicate == null) throw new ArgumentNullException("predicate");

            m_target = target;
            m_predicate = predicate;
        }

        ...
    }
```

Now we need to implement the ITargetBlock<TInput> interface and its base interface – IDataflowBlock. Let's start with the latter as it is really simple. Simply forward all members to m_target:

```
        // *** IDataflowBlock implementation ***
        public Task Completion
        {
            get
            {
                return m_target.Completion;
            }
        }

        public void Complete()
        {
            m_target.Complete();
        }

        void IDataflowBlock.Fault(Exception exception)
        {
            m_target.Fault(exception);
        }
```

Finally, implement ITargetBlock<TInput> itself which has only one method:

```
        // *** ITargetBlock<TInput> implementation ***
        DataflowMessageStatus ITargetBlock<TInput>.OfferMessage(DataflowMessageHeader
messageHeader, TInput messageValue, ISourceBlock<TInput> source, bool consumeToAccept)
        {
            // Evaluate the predicate over the passed-in value
            var passed = m_predicate(messageValue);

            // Propagate the offering synchronously if necessary
            if (passed)
                return m_target.OfferMessage(messageHeader, messageValue, source,
consumeToAccept);
            else
                return DataflowMessageStatus.Declined;
        }
```

And that is our first custom block implemented from scratch albeit very constrained.

5.4. Implementing a Synchronous Transform Block

Next we'll relax some of the constraints around block we are building. Let's say we need a block that does some lightweight transformation on the fly that causes the type of the output messages to be different from the type of the input messages.

We can still count on constraint a. from above, but constraint b. is no longer present which also defeats constraint c. What that means is we cannot forward the same source reference in the call to m_target.OfferMessage(). That will not even compile, because m_target expects a reference to ISourceBlock<TOutput> while what is being passed in is ISourceBlock<TInput>.

First, notice that the ISourceBlock<TOutput> implementation that we pass to m_target.OfferMessage() need not be public, i.e. there is no need to make our block a source. We can, and we should, implement ISourceBlock<TOutput> as a private class inside our block.

Second, we need to create a source "proxy" per actual source. It's OK to create multiple proxies per single source if simplicity is very important, but you should remember that excessive object allocations put unnecessary pressure on the garbage collector which should be avoided. In this implementation, we'll use a single proxy per actual source and we'll use a System.Collections.Concurrent.ConcurrentDictionary<ISourceBlock<TInput>, SyncTransformSource> to maintain that mapping.

The synchronous transform propagator looks very much like the synchronous filter propagator except that the input message type and the output message type may be different and there is an additional mapping filed:

```
    public sealed class SyncTransformTarget<TInput, TOutput> : ITargetBlock<TInput>
    {
        /// <summary>The ITargetBlock{TOutput} to which this block is
propagating.</summary>
        private readonly ITargetBlock<TOutput> m_target;

        /// <summary>The transformation function.</summary>
        private readonly Func<TInput, TOutput> m_transform;

        /// <summary>The actual-to-proxy source mapping.</summary>
        private readonly ConcurrentDictionary<ISourceBlock<TInput>, SyncTransformSource>
m_sourceMap;


        /// <summary>
        /// Constructs a SyncTransformTarget{TInput, TOutput}.
        /// </summary>
        /// <param name="target">Target block to propagate to.</param>
        /// <param name="transform">Transformation function.</param>
        public SyncTransformTarget(ITargetBlock<TOutput> target, Func<TInput, TOutput>
transform)
        {
            if (target == null) throw new ArgumentNullException("target");
            if (transform == null) throw new ArgumentNullException("transform");

            m_target = target;
            m_transform = transform;
            m_sourceMap = new ConcurrentDictionary<ISourceBlock<TInput>,
SyncTransformSource>();
        }
        ...
    }
```

Implementing IDataflowBlock is exactly the same:

```
        // *** IDataflowBlock implementation ***
        public Task Completion
        {
            get
            {
```

```
                return m_target.Completion;
            }
        }

        public void Complete()
        {
            m_target.Complete();
        }

        void IDataflowBlock.Fault(Exception exception)
        {
            m_target.Fault(exception);
        }
```

Implementing ITargetBlock<TInput> is slightly different – we need to pass down an appropriate source proxy – either find an existing one or create and store one:

```
        // *** ITargetBlock<TInput> implementation ***
        DataflowMessageStatus ITargetBlock<TInput>.OfferMessage(DataflowMessageHeader
messageHeader, TInput messageValue, ISourceBlock<TInput> source, bool consumeToAccept)
        {
            // Make sure the source is uniquely mapped if it is non-null
            SyncTransformSource sourceProxy = source != null ?
m_sourceMap.GetOrAdd(source, src => new SyncTransformSource(this, src)) : null;

            // Transform the value
            var transformedValue = m_transform(messageValue);

            // Propagate the offering synchronously
            return m_target.OfferMessage(messageHeader, transformedValue, sourceProxy,
consumeToAccept);
        }
```

Now we have to implement the private source proxy class – SyncTransformSource. It is simple – it keeps two references: one to the actual source and one to the owning block. The only important thing here is to create a continuation from the original source's Completion property to remove the mapping between the original source and this proxy. Everything else is straightforward:

```
        private sealed class SyncTransformSource : ISourceBlock<TOutput>
        {
            /// <summary>The ISourceBlock{TInput} to which this block is
propagating.</summary>
            private readonly ISourceBlock<TInput> m_source;

            /// <summary>The parent SyncTransformTarget{TInput, TOutput}.</summary>
            private readonly SyncTransformTarget<TInput, TOutput> m_parent;


            internal SyncTransformSource(SyncTransformTarget<TInput, TOutput> parent,
ISourceBlock<TInput> source)
            {
                Contract.Requires(parent != null);
                Contract.Requires(source != null);
```

```
                m_parent = parent;
                m_source = source;

                // Remove the proxy from the map as soon as the original source completes
                SyncTransformSource proxySource;
                source.Completion.ContinueWith(_ => parent.m_sourceMap.TryRemove(source,
out proxySource));
            }


            // *** IDataflowBlock implementation ***
            public Task Completion
            {
                get
                {
                    return m_source.Completion;
                }
            }

            public void Complete()
            {
                m_source.Complete();
            }

            void IDataflowBlock.Fault(Exception exception)
            {
                m_source.Fault(exception);
            }


            // *** ISourceBlock<TOutput> implementation ***
            IDisposable ISourceBlock<TOutput>.LinkTo(ITargetBlock<TOutput> target, bool
unlinkAfterOne)
            {
                throw new NotSupportedException(ERROR_NoDynamicLinking);
            }

            TOutput ISourceBlock<TOutput>.ConsumeMessage(DataflowMessageHeader
messageHeader, ITargetBlock<TOutput> target, out bool messageConsumed)
            {
                // Consume the original value and transform it for the target
                var originalValue = m_source.ConsumeMessage(messageHeader, m_parent, out
messageConsumed);
                return messageConsumed ?
                    m_parent.m_transform(originalValue) : default(TOutput);
            }

            bool ISourceBlock<TOutput>.ReserveMessage(DataflowMessageHeader
messageHeader, ITargetBlock<TOutput> target)
            {
                return m_source.ReserveMessage(messageHeader, m_parent);
            }

            void ISourceBlock<TOutput>.ReleaseReservation(DataflowMessageHeader
messageHeader, ITargetBlock<TOutput> target)
            {
                m_source.ReleaseReservation(messageHeader, m_parent);
            }
```

```
        }
```

5.5. Implementing an Asynchronous Block

5.5.1. A Word of Caution

Implementing an autonomous block contains lots of pitfalls related to locking, block completion, transferring of message ownership, and non-greediness if applicable. Before starting to implement an autonomous block from scratch, do a thorough investigation whether you can accomplish what you need through encapsulating an existing block eventually combined with a custom synchronous propagator block as described in the preceding sections.

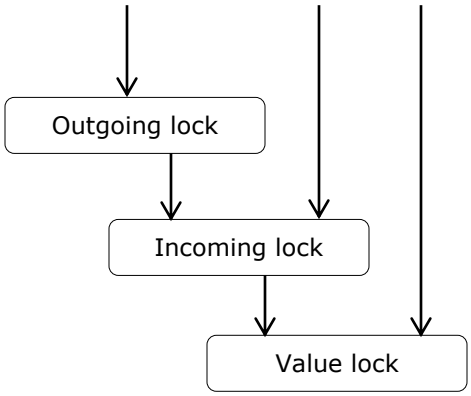
5.5.2. Locking

Since asynchronous blocks schedule calling their source/target parties on separate threads, a block must synchronize its target- and source-side. A possible locking scheme is the one described below. It is the one the built-in blocks use. You may implement a different one as longs as it allows you to correctly implement the message propagation protocol:

- Target-side
    - Incoming lock
        - A lock that is used to synchronize all target-related activity, with the main purpose of ensuring that multiple messages offered to the block at the same time are processed serially. For example, if accepting one message triggers the block to decline permanently, all future messages will be correctly declined.
        - If the SingleProducerConstrained option is passed in, the block may try to avoid using this lock altogether because the user is promising to offer messages serially.
- Source-side
    - Outgoing lock
        - A lock that is used to synchronize all source-related activity, ensuring that propagation of data out of the block is processed serially. For example, during message propagation to target blocks, a receive operation or an attempt to reserve a message will be delayed until propagation completes.
        - This lock may also be used to protect the data structure that stores information about linked targets.
        - This lock must be reentrant. This is needed because a target may call, as part of its OfferMessage implementation, a method on the source that takes this lock, e.g. ConsumeMessage(), ReserveMessage(), or LinkTo() which will cause a deadlock if the lock is not reentrant.
    - Value lock
        - A lock that protects the internal state of the source. It is separated from the Outgoing lock in order to allow the target-side of a block to manipulate aspects

of the source-side without having to acquire the Outgoing lock, which could lead to deadlock.

Whenever multiple locks are to be acquired, a locking order must be followed strictly. Otherwise deadlocks may occur. In order to enable a block to participate in network cycles, i.e. the block to be either directly linked to itself or indirectly – through one or more other blocks, the following order of lock taking must be followed:



5.5.3. Block Completion

Here are some clarifications around block completion policy:

- A block should stop accepting messages and should start returning DecliningPermanently as soon as the Complete() or Fault() method has returned.
- A block should allow the Complete() or Fault() method to be called multiple times without throwing an exception even if the block's Completion property is already in Completed state.
- A block should not put its Completion property into Completed state before all buffered messages have been propagated unless the block has been canceled or faulted.

A block should end up in a Faulted state if it encounters an exception during message processing even if the Complete() method has already been called. (This doesn't mean that having Fault called after Complete has been called should put the block in Faulted state.)

A block should complete its Completion property without holding a lock, because there may be synchronous continuations off the task returned from the Completion property which will execute before the completion status change call returns, i.e. under the lock, which may potentially cause a deadlock.

The task returned from a block's Completion property may be lazily initialized, i.e. it may not be constructed until needed, but once returned the block must keep returning the same task.

When a block's Complete() or Fault() method has been called, the block should not try consume postponed messages. Instead, it should try to "release" those postponements, because some sources stop offering when a message has been postponed. Since there is no direct API to accomplish that, the following pattern may be used:

```
if (sourceOfPostponedMessage.ReserveMessage(postponedMessageHeader, thisTarget))
{
    sourceOfPostponedMessage.ReleaseReservation(postponedMessageHeader, thisTarget);
}
```

Once a block completes, it should iterate over its linked targets and should propagate completion to the targets that have been linked with the PropagateCompletion option. Then the block should unlink its linked targets.

5.5.4. Non-Greediness

The key feature of the non-greedy behavior is that messages remain owned by the sources until the target can consume all necessary messages to produce a group. This implies that a target should be directly postponing messages from OfferMessage, and it should be consuming them asynchronously.

Targets should also try to preserve the order within output groups if an order is applicable.

A target may keep track of only the last message offered by a given source. If a source continues to offer messages, it means other targets must have consumed the previous messages. So they are no longer available.

Once a target detects a chance to produce an output group, it should use the following pattern to consume postponed messages (checks, BoundingCapacity in particular, are skipped for simplicity):

```
// PHASE 1: Reserve messages
while (reservedInfoTemp.Count < m_groupSize)
{
    SourceMessageInfo info;

    // Take the lock while accessing the m_postponedInfo queue
    lock (IncomingLock)
    {
        if (!m_postponedInfo.TryPop(out info)) break;
    } // Release the lock.  It must not be held while calling Reserve/Consume/Release.

    var source = info.Source;
    var messageHeader = info.MessageHeader;
    if (source.ReserveMessage(messageHeader, m_thisBlock))
    {
        // Maintain a temp list of reserved messages.
        reservedInfoTemp.Add(postponedInfoTemp);
    }
}

// PHASE 2: Consume reserved messages (or release reservations if the necessary number of
messages was not reserved)
if (reservedInfoTemp.Count == m_groupSize)
{
    // We have enough messages. CONSUME them.
    for (int i = 0; i < reservedInfoTemp.Count; i++)
    {
        var source = reservedInfoTemp[i].Source;
        var messageHeader = reservedInfoTemp[i].MessageHeader;
        bool consumed;
        var consumedValue = source.ConsumeMessage(messageHeader, m_thisBlock, out
consumed);
        if (!consumed)
        {
            // The protocol broke down, so throw an exception, as this is fatal.
            // Before doing so, though, null out all of the messages we've already
            // consumed, as a higher-level event handler should try to release
            // everything in the reserved temp list.
```

```
            for (int prev = 0; prev < i; prev++)
                reservedInfoTemp[prev] = default(SourceMessageInfo);
            throw new InvalidOperationException(FailedToConsumeReservedMessage));
        }

        var consumedInfo = new SourceMessageInfo(source, messageHeader, consumedValue);
        reservedInfoTemp[i] = consumedInfo;
    }

    // Enqueue the consumed mesasages
    lock (IncomingLock)
    {
        foreach (var info in reservedInfoTemp)
        {
            m_bufferedMessages.Enqueue(info.Message);
        }
    }

    reservedInfoTemp.Clear();
}
else
{
    // We don't have enough messages. RELEASE what we have reserved.
    for(int i = 0; i < reservedInfoTemp.Count; i++)
    {
        var source = reservedInfoTemp[i].Source;
        var messageHeader = reservedInfoTemp[i].MessageHeader;
        reservedInfoTemp[i] = default(SourceMessageInfo);
        if (source != null && messageHeader.IsValid)
        {
            // If the source throws, let the exception bubble to an outer handler.
            source.ReleaseReservation(messageHeader, m_thisBlock);
        }
    }

    // We should not try to put back the range of postponed messages, because
    // once ReleaseMessage has been called, the source will re-offer it.
}
```

5.5.5. Offering Messages and Linking Sources to Targets

Message offering and linking of sources to targets should be synchronized in most cases. That is because calls to ConsumeMessage(), ReserveMessage(), and ReleaseReservation() may affect the offering process. For instance, ConsumeMessage() may cause a target to be unlinked if the MaxMessages limit for that target has been reached.

It is up to the source to choose what disposable link object to return, and what that disposable link object should do upon disposal. In most cases that would be a removal of the target from the source's linked target list. Keep in mind that this operation should also be synchronized with LinkTo(), message offering, ConsumeMessage(), ReserveMessage(), and ReleaseReservation().

6. References

For additional information on TPL Dataflow and how to implement custom blocks, consider the following resources:

TPL Dataflow Home on the DevLabs Portal
http://msdn.microsoft.com/en-us/devlabs/gg585582#

TPL Forum on MSDN:
http://social.msdn.microsoft.com/Forums/en-US/tpldataflow/threads

PFX Team Blog:
http://blogs.msdn.com/b/pfxteam/