TPL Dataflow

Growing Interest In …

Dataflow

Processing Pipelines

Network of agents (actors)

Dispatching / routing / "flow"

Coordination

Message passing

Concurrency

- … Moving beyond simple asynchrony and multithreading

What's Wrong With Current Approaches

Very complicated

Few software developers can write concurrent software

- And very few can do it well

Difficult to debug

Long-term evolution is error prone

Yet …

- There is an ever growing need to more successful utilize available compute resources

Move Towards …

Asynchrony

- Parallelism implies asynchrony
- The inverse is not necessarily so

Message Passing

- Lots of messages flowing everywhere, need to maintain order
- (by default, in-process, could manually extend this to out of process)

Data flow and agents

- Contrast with work flow
- Contrast with control flow

Success Of Task Abstraction

.NET 4.0 introduced the Task abstraction to represent a quantum of work to be performed sometime in the future

- System.Threading.Tasks
- Nice API which has become very popular

Task could complete

- Successfully
- In error
- May have been canceled

A task's completion may be awaited

A task can be scheduled using a TaskScheduler

Introducing TPL Dataflow

“TPL Dataflow (TDF) is a new .NET library for building concurrent applications. It promotes actor/agent-oriented designs thorugh primitives for in-process message passing, dataflow and pipelining”

- Stephen Toub

Works well with:

- Async C#
- Task Parallel Library (TPL)
- Reactive Extensions (Rx)

A little overlap with Rx but mostly aimed for different areas

Installing TPL Dataflow

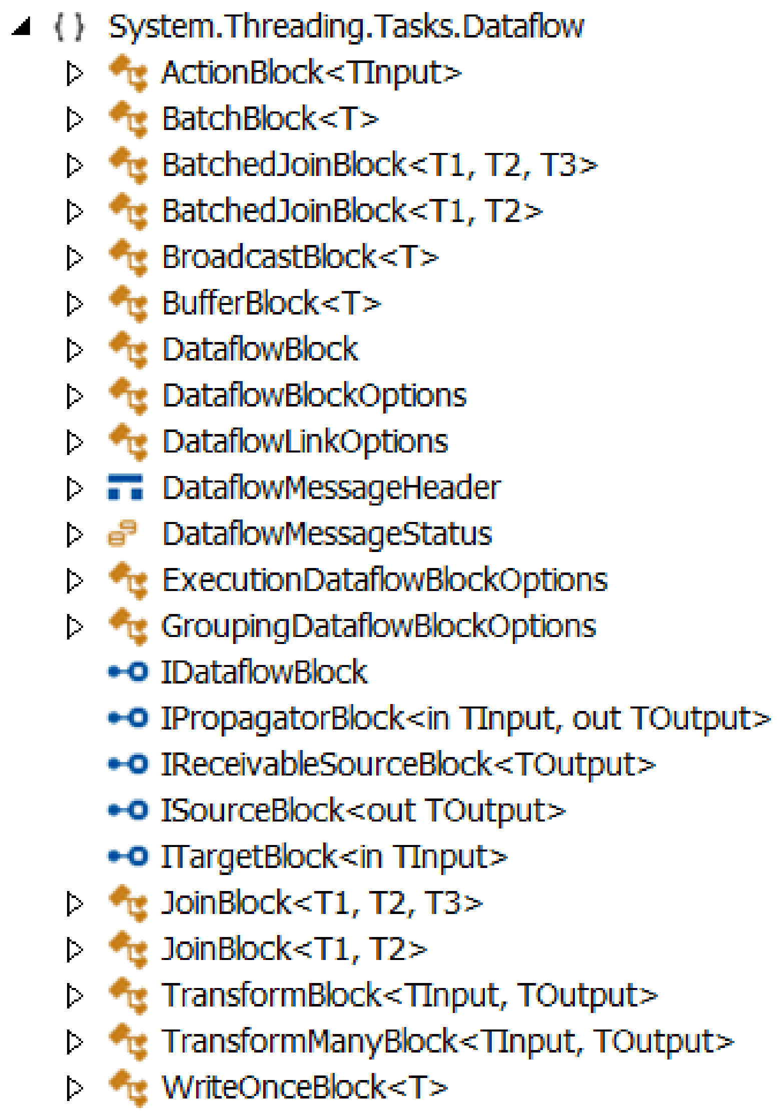
TPL Dataflow is delivered as a single assembly

- System.Threading.Tasks.Dataflow.dll

Installed available via two routes

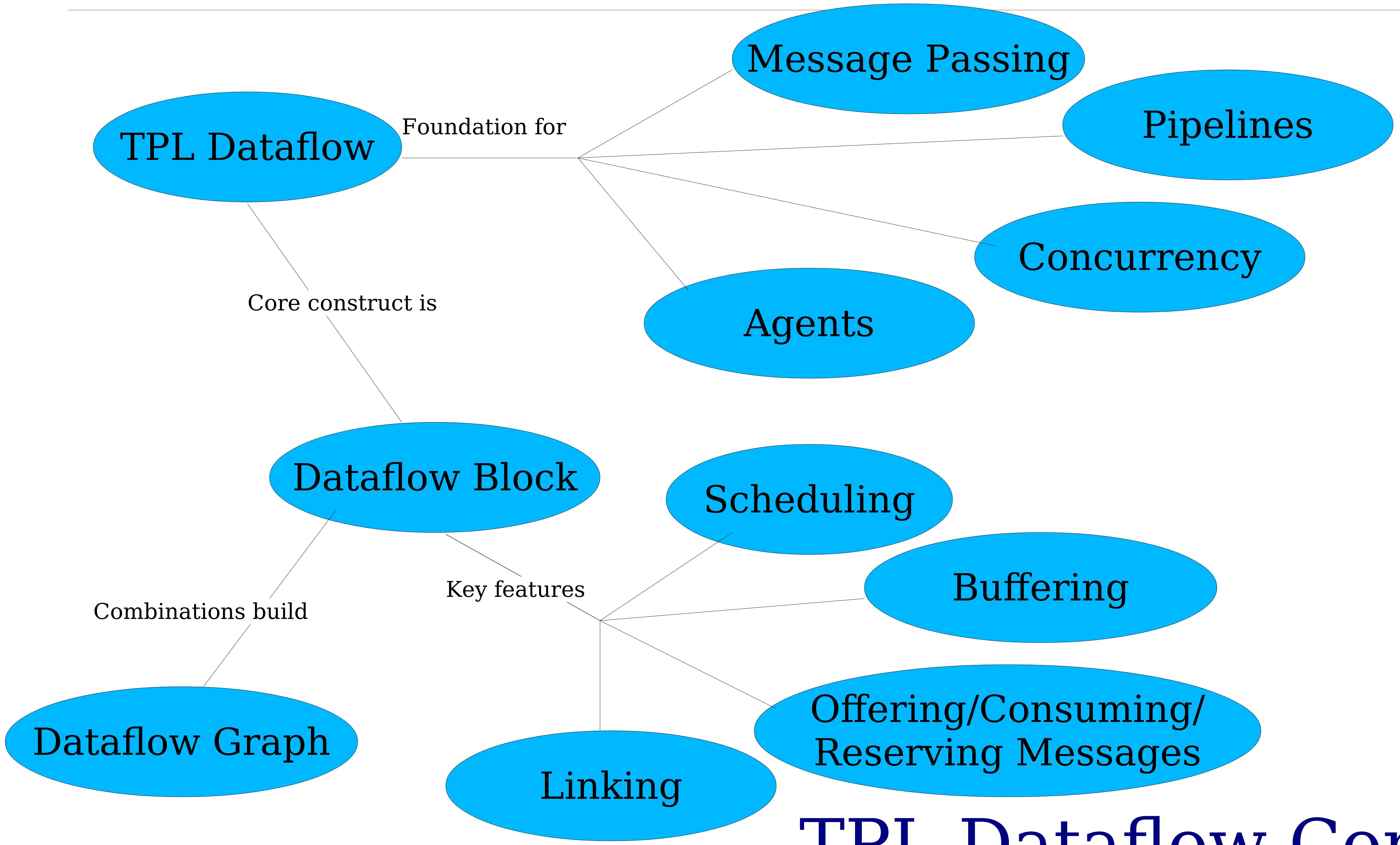
- Via nuget: System.Threading.Tasks.Dataflow
- Via .NET Core CLI: dotnet add package System.Threading.Tasks.Dataflow

What is in the TPL Dataflow Assembly?



```
◢ { } System.Threading.Tasks.Dataflow
    ▷ ActionBlock<TInput>
    ▷ BatchBlock<T>
    ▷ BatchedJoinBlock<T1, T2, T3>
    ▷ BatchedJoinBlock<T1, T2>
    ▷ BroadcastBlock<T>
    ▷ BufferBlock<T>
    ▷ DataflowBlock
    ▷ DataflowBlockOptions
    ▷ DataflowLinkOptions
    ▷ DataflowMessageHeader
    ▷ DataflowMessageStatus
    ▷ ExecutionDataflowBlockOptions
    ▷ GroupingDataflowBlockOptions
      IDataflowBlock
      IPropagatorBlock<in TInput, out TOutput>
      IReceivableSourceBlock<TOutput>
      ISourceBlock<out TOutput>
      ITargetBlock<in TInput>
    ▷ JoinBlock<T1, T2, T3>
    ▷ JoinBlock<T1, T2>
    ▷ TransformBlock<TInput, TOutput>
    ▷ TransformManyBlock<TInput, TOutput>
    ▷ WriteOnceBlock<T>
```

TPL Dataflow Concepts



TPL Dataflow Concepts

Messages

Message

- A “message” (or “element” or “item”) flows through one or more dataflow blocks

Offering Messages

Consuming Messages

Reserve / Release Messages

Options

Maintaining Order

Synchronous and Asynchronous

What Is A Task?

A task is a unit of work that we wish to have completed

- A task can be scheduled
- A task can be canceled
- A task can complete successfully, faulted or be canceled

What are the differences between Thread and Task?

Tasks are scheduled using a TaskScheduler

- Default uses the .NET threadpool
- Can create custom schedulers

Covariance And Contravariance

Why needed?

- Helps with the use of generics with inheritance hierarchies

Covariance, defined by out keyword, is like polymorphism

- Instance of more derived generic type can be used instead of base

Contravariance, defined by in

- Instances of less derived generic type can be used instead of derived type

Additional meaning for out

- Note out keyword in a C# method signature has a distinct meaning

Types Used in TPL Dataflow

IEquatable<>

IDisposable

Tuple

IList

IObserver/IObservable

CancellationToken

Action<>, Func<> And Predicate<>

The custom code that runs inside blocks can be supplied as actions or funcs

- Actions and funcs are delegates with multiple overloads that take varying number of parameters
- Action has no return value
  public delegate void Action()
  public delegate void Action<in T>(T obj)
- Func has a return value (whose type is supplied as type parameter)
  public delegate TResult Func<out TResult>()
  public delegate TResult Func<in T, out TResult>(T arg)

Predicates are used for filtering

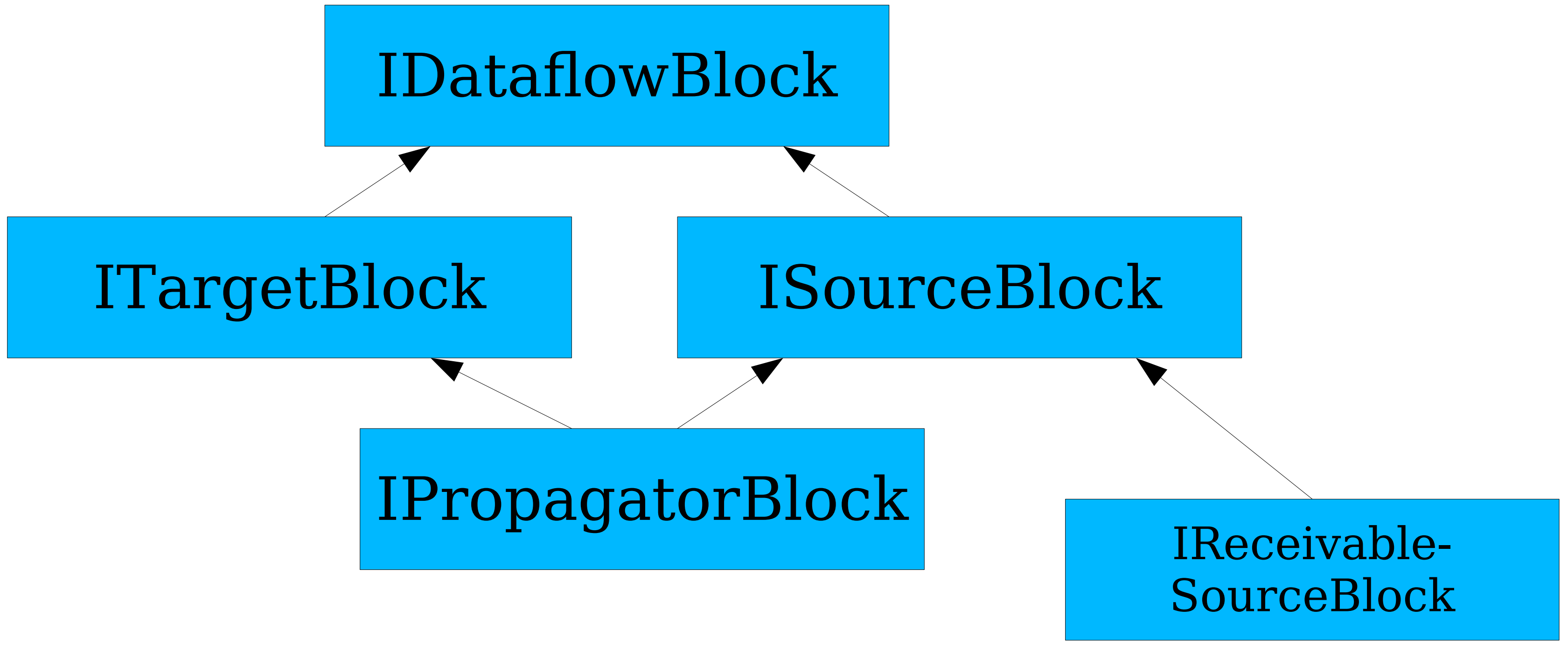
- Some dataflow block operations allow you to filter messages
- e.g. only messages that comply with certain criteria are passed
- public delegate bool Predicate<in T>(T obj)
- (All defined in System namespace)

TPL Dataflow Interfaces

Dataflow Interfaces

TDF defines 5 interfaces:

- IDataflowBlock
- ITargetBlock
- ISourceBlock
- IPropagatorBlock
- IReceivableSourceBlock



Notes

- They use two helper types – DataflowMessageHeader and DataflowMessageStatus

DataflowMessageHeader

In the interaction between a target and a source messages can be:

- Offered
- Consumed
- Reserved
- Released

A message header is used to identify a particular message

- Basically, it is just an integer ID, generated by the source
- Unique between a particular source and target at that time
- A valid message header is any with a non-zero ID
- A .NET struct does not support inheritance, so cannot add custom header information (if required, create an envelope containing message header(s) + message body)

DataflowMessageHeader Definition

```
public struct DataflowMessageHeader : IEquatable<DataflowMessageHeader>{
  public DataflowMessageHeader(long id);
  public static bool operator !=(
              DataflowMessageHeader left, DataflowMessageHeader right);
  public static bool operator ==(
              DataflowMessageHeader left, DataflowMessageHeader right);
  public long Id { get; }
  public bool IsValid { get; }
  public bool Equals(DataflowMessageHeader other);
  public override bool Equals(object obj);
  public override int GetHashCode();
}
```

DataflowMessageStatus

When a message is offered to a target, the result is an instance of DataflowMessageStatus

Indicates whether the message:

- Was accepted immediately
- Was declined
- Was postponed
- Was not available
- Was permanently declined

DataflowMessageStatus Definition

```
public enum DataflowMessageStatus {
    Accepted = 0,
    Declined = 1,
    Postponed = 2,
    NotAvailable = 3,
    DecliningPermanently = 4,
  }
```

IDataflowBlock

IDataflowBlock is the base interface

- (all other interfaces derive from this one)

Its role is to facilitate the concept of block completion

- A block “completes” when it no longer accepts incoming messages or forwards outgoing messages
- There can be a time lage between the time a block is told to complete/fault, and when it actually stops processing, since messages already inside the block will still be processed

IDataflowBlock provides a Task used to determine when a block has completed

- Allows a block to indicate it has completed successfully, faulted or canceled

IDataflowBlock Definition

```
public interface IDataflowBlock {
  Task Completion { get; }
  void Complete();
  void Fault(Exception exception);
}
```

IDataflowBlock provides methods for successful & faulted completion

- Note that a number of blocks implement this interface (and other interfaces) explicitly, so a cast may be required in order to execute an interface method
- Separately, CancellationToken (see options) used for canceled result

ITargetBlock

A block that implements ITargetBlock is offered messages

- A message flows into a target (so target's type parameter is named TInput)

ITargetBlock adds one method (OfferMessage) to IDataflowBlock

- Think of a dataflow network as a directed graph of blocks
- In TPL Dataflow's case, these blocks implement some/all of the 5 TDF interfaces

Two ways a target block can be offered messages

- From other (source) blocks in the networks (from linked source blocks)
- Or messages can be injected into the network when a TargetBlock at the edge

ITargetBlock Definition

```
public interface ITargetBlock<in TInput> : IDataflowBlock {
   DataflowMessageStatus OfferMessage(
                                      DataflowMessageHeader messageHeader,
                                      TInput messageValue,
                                      ISourceBlock<TInput> source,
                                      bool consumeToAccept);
}
```

ISourceBlock

The role of a block that implements ISourceBlock's is to supply messages

- A message flows out of a source (so source's parameter type is named TOutput)

Three features

- Linking
- ConsumeToAccept
- Reserve/Consume/ReleaseReservation

ISourceBlock Definition

```
public interface ISourceBlock<out TOutput> : IDataflowBlock {
  TOutput ConsumeMessage(DataflowMessageHeader messageHeader,
          ITargetBlock<TOutput> target, out bool messageConsumed);

  IDisposable LinkTo(ITargetBlock<TOutput> target,
                bool unlinkAfterOne);

  void ReleaseReservation(DataflowMessageHeader messageHeader,
                ITargetBlock<TOutput> target);

  bool ReserveMessage(DataflowMessageHeader messageHeader,
                ITargetBlock<TOutput> target);

}
```

Linking

Dataflow blocks can be linked to form an (in-process) network

ISourceBlock.LinkTo is used to create one link to a target

- Can call this on blocks that are both source and target
- Hence a network can be created

Messages pushed into blocks that are linked to others are forwarded to others

- After processing as the block semantics require

To unlink

- Automatically, set unlinkAfterOne to true (usually false)
- The return value of LinkTo is IDisposable – call its dispose() method
- Complete block

ConsumeToAccept

Some blocks support cloning

- e.g. broadcasting the same message to multiple targets
- Can be expensive to clone

Sometimes the target will accept an offered message, and sometimes will postpone or decline it

- Only wish to incur cloning cost if message will be accepted
- By setting consumeToAccept parameter in ITargetBlock.OfferMessage to true, we are telling target to call ISourceBlock.ConsumeMessage in order to accept message
- Only needs to do so if it is prepared to accept message
- Cloning happens inside ISourceBlock.ConsumeMessage, so cost only incurred as needed

Reserving a Message

Sometimes a block wishing to process a message, but not just now

- Can reserve it (ISourceBlock.ReserveMessage)

Later, can either:

- Consume message (ISourceBlock.ConsumeMessage)
- Release reservation (ISourceBlock.ReleaseReservation)

IPropagatorBlock Interface

IPropagatorBlock's role is to:

- Act as both target and source
  (derives from both ISourceBlock and ITargetBlock)
- It adds no functionality beyond that
- Useful when you wish to have a strongly collection of dataflow blocks and you know they can act as both sources and targets

IPropagatorBlock Definition

```
public interface IPropagatorBlock<in TInput, out TOutput> :
  IdataflowBlock, IsourceBlock<TOutput>, ItargetBlock<TInput> {}
```

IReceivableSourceBlock

IReceivableSourceBlock's role is to:

- Provide additional optional functionality for a source block
  (derives from ISourceBlock and adds two methods)

TryReceive receives the first available message (with an optional filter)

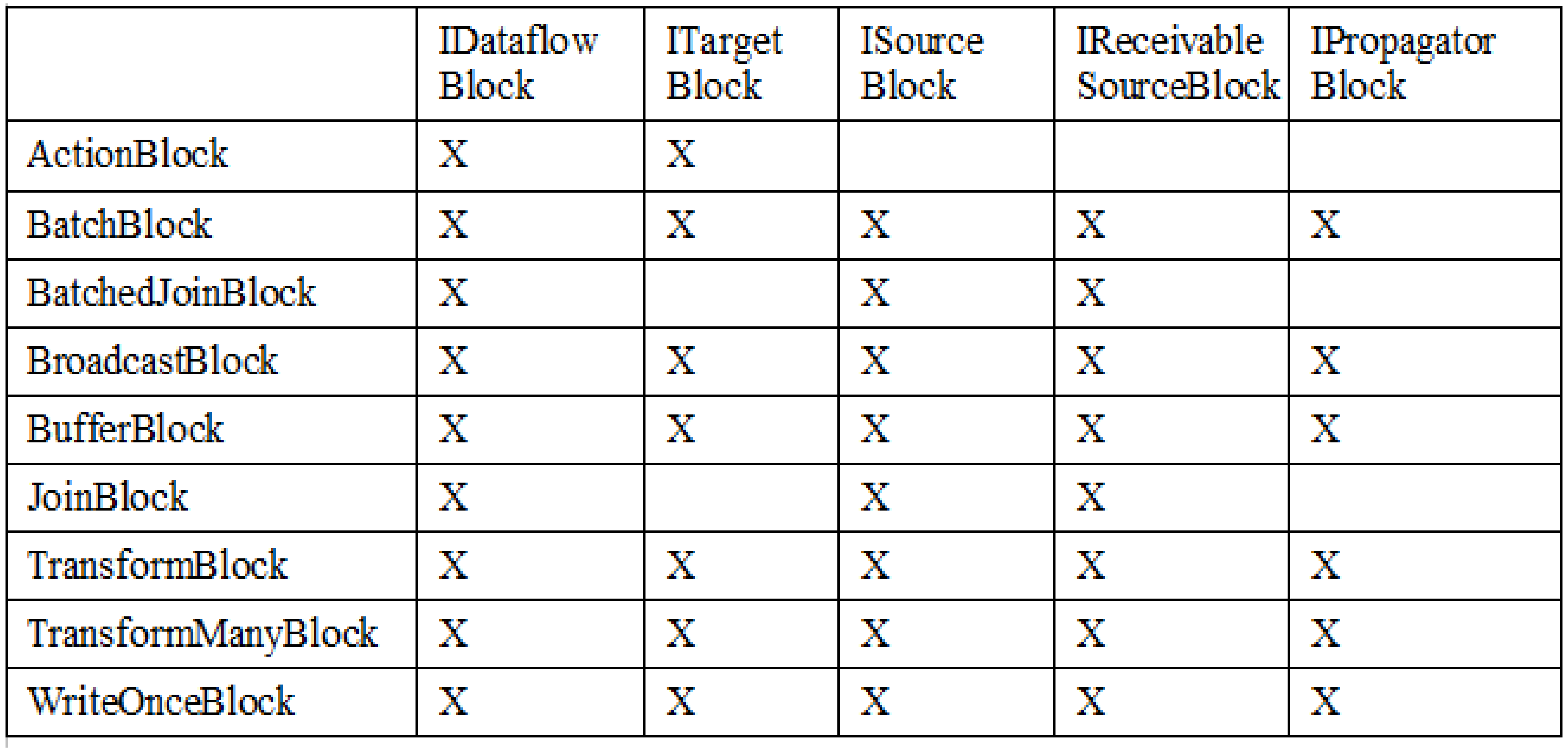
TryReceiveAll receives all available messages

- These may return no message is none is available
- These are synchronous methods
- Don't need to be a target to receive messages
  (hence ideal for dataflow network edges)

IReceivableSourceBlock Definition

```
public interface IReceivableSourceBlock<TOutput>
                                : ISourceBlock<TOutput>, IDataflowBlock  {
  bool TryReceive(Predicate<TOutput> filter, out TOutput item);
  bool TryReceiveAll(out IList<TOutput> items);
}
```

Which Block Implements Which Interfaces?



	IDataflow Block	ITarget Block	ISource Block	IReceivable SourceBlock	IPropagator Block
ActionBlock	X	X			
BatchBlock	X	X	X	X	X
BatchedJoinBlock	X		X	X	
BroadcastBlock	X	X	X	X	X
BufferBlock	X	X	X	X	X
JoinBlock	X		X	X	
TransformBlock	X	X	X	X	X
TransformManyBlock	X	X	X	X	X
WriteOnceBlock	X	X	X	X	X

Buffering Blocks

DataBlock Implementations

Three are three categories of datablock implementations provided

Distinguished by the Options they takes

- DatablockOptions (default) – buffering
- ExecutionOptions – execution
- GroupingOptions – grouping

Custom Blocks And Custom Options

- You and third parties, can of course create more
- Create blocks that use the above standard options, or
- Create custom options and create blocks that use them

Buffering Dataflow Blocks

BufferBlock

- Buffer of messages, each supplied to at most one particular target

BroadcastBlock

- Clones messages to each linked target

WriteOnceBlock

- One message only (singleton), cloned and supplied to each linked targets

BufferBlock

A BufferBlock maintains an ordered buffer

- Incoming messages are placed at the end of a FIFO queue
- Outgoing messages are taken from the front of the FIFO queue and offered each linked target in turn, until first accepts, or reserves the message
- If there are no linked targets, or none accepts/reserves messge, it stays in queue

Two comon traits about all block implementations in TPL Dataflow

- All are sealed
- All have a constructor with an option parameter and one without (uses default options)

BufferBlock Definition

```
public sealed class BufferBlock<T> : IPropagatorBlock<T, T>,
       ITargetBlock<T>, IReceivableSourceBlock<T>,
       ISourceBlock<T>, IDataflowBlock {
    public BufferBlock();
    public BufferBlock(DataflowBlockOptions dataflowBlockOptions);
    public Task Completion { get; }
    public int Count { get; }
    public void Complete();
    public IDisposable LinkTo(
                    ITargetBlock<T> target, bool unlinkAfterOne);
    public bool TryReceive(Predicate<T> filter, out T item);
    public bool TryReceiveAll(out IList<T> items);
}
```

Notes

With BufferBlock, the message only exists once

No cloning or sharing happens

The message at all times is owned by one block and one block only

Count tells us how many messages are in buffer

Capacity of buffer can be set via options

By default, capacity is unbounded

BroadcastBlock

A broadcast block “broadcasts” a message to all linked targets

Keeps a record of current message (just one)

- Current message is overwritten each time a message flows through

Provided at least one message has flowed through BroadcastBlock

Can access current message

- Manually calling Receive
- Linking a target block afterwards

BroadcastBlock Definition

```
public sealed class BroadcastBlock<T> : IPropagatorBlock<T, T>,
        ITargetBlock<T>, IReceivableSourceBlock<T>,
        ISourceBlock<T>, IDataflowBlock {

    public BroadcastBlock(Func<T, T> cloningFunction);

    public BroadcastBlock(Func<T, T> cloningFunction,
                    DataflowBlockOptions dataflowBlockOptions);

    public Task Completion { get; }

    public void Complete();

    public IDisposable LinkTo(
                    ITargetBlock<T> target, bool unlinkAfterOne);

    public bool TryReceive(Predicate<T> filter, out T item);
}
```

Notes

The constructor can be passed in a cloning function

- This will be called to clone the message for each target and sent each a different clone
- Cloning function parameter may be null, in qhich case the same message instance is sent to each target
- Often use consumeToAccept = true for ITargetBlock.OfferMessage with BroadcastBlocks

Cloning function could be the identity function i=>i

- More efficient to pass in null

WriteOnceBlock

Think of a singleton, const or readonly variable

WriteOnceBlock contains a single message

- Can be written just once
- Can be read any number of times
- Constructor takes a cloning function (similar to BroadcastBlock

For WriteOnceBlock, DataflowOptions.BoundedCapacity is ignored

WriteOnceBlock Definition

```
public sealed class WriteOnceBlock<T> : IPropagatorBlock<T, T>,
        ITargetBlock<T>, IReceivableSourceBlock<T>,
        ISourceBlock<T>, IDataflowBlock {
    public WriteOnceBlock(Func<T, T> cloningFunction);
    public WriteOnceBlock(Func<T, T> cloningFunction,
                          DataflowBlockOptions dataflowBlockOptions);
    public Task Completion { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<T> target,
                          bool unlinkAfterOne);
    public bool TryReceive(Predicate<T> filter, out T item);
}
```

DataflowOptions

Good idea to pass a DataflowOptions instance to a block's constructor

- All TPL Dataflow blocks do
- Influences how block operates

Note DataflowOptions class is not sealed

- Can create custom options with additional parameters for custom blocks

DataflowOptions Definition

```
public class DataflowBlockOptions {
    public const int Unbounded = -1;
    public DataflowBlockOptions();
    public int BoundedCapacity { get; set; }
    public CancellationToken CancellationToken { get; set; }
    public int MaxMessagesPerTask { get; set; }
    public TaskScheduler TaskScheduler { get; set; }
  }
```

Creating Options

Default constructor creates a DataflowOptions instance with default parameters

Get and set accessors available for all properties

Value of properties at block construction should be used

For a particular block created with a particular options instance, later, when options are changed, these changes are not to be reflected in block

- Used only for new blocks created afterwards

Default Values

Bounded Capacity = Unbounded

CancellationToken = none

MaxMessagesPerTask = Unbounded

TaskScheduler = Default sechduler

Bounded Capacity

A question for blocks that buffer

- How many messages should the buffer contain
- By default, the answer is there is no limit (well, until memory is exhausted)

Can limit buffer size using:

- DataflowOptions.BoundedCapacity
- Useful for modulating buffer sizes throughout network
- Allowing multiple targets get some messages
- Allowing most recent message to be processed

MaxMessagesPerTask

Blocks spin up tasks to perform functionality

Can limit number of messages a particular task will process

- At the end of which, task will complete
- New task will spin up to continue

Supports fairness in scheduling of tasks

CancellationToken

A cancellation token can be used to cancel a task in-flight

Tasks must be specially written in order to cooperate with a cancellationToken

- Tasks in all TPL Dataflow blocks do support this

How it works

- A CancellationToken is passed in via options
- At some later point, client code can cancel the CancellationToken
- Task within a block notices this request
- Cancels activity
- Completion task completes

TaskScheduler

A TaskScheduler schedules task operations

Usually the .NET thread pool

Can use dataflowOptions to configure a different TaskScheduler

Might have different priority scheme, scheduling, instrumentation

- Anything

Also pay attention to ConcurrentExclusiveSchedulerPair

- A reader/writer lock for schedulers

Executing Blocks

Execution Blocks

ActionBlock

TransformBlock

TransformManyBlock

- All blocks that execute an action

ActionBlock

An ActionBlock is a target only

Has an input queue

- Size known via InputCount property

For each message, executes delegate specified in constructor

- Action<TInput>
- Func<TInput, Task>

Allows synchronous or asynchronous calling

ActionBlock Definition

```
public sealed class ActionBlock<TInput>
                              : ITargetBlock<TInput>, IDataflowBlock {
  public ActionBlock(Action<TInput> action);
  public ActionBlock(Func<TInput, Task> action);
  public ActionBlock(Action<TInput> action,
                     ExecutionDataflowBlockOptions dataflowBlockOptions);
  public ActionBlock(Func<TInput, Task> action,
                     ExecutionDataflowBlockOptions dataflowBlockOptions);
  public Task Completion { get; }
  public int InputCount { get; }
  public void Complete();
}
```

Notes

Once a message is processed by an actionBlock, it is lost

If you wish to pass on a message for further processing to a different block, using TransformBlock

TransformBlock

TransformBlock is both a source and a target

Maintains two queues

- Size of each can be known via InputCount and OutputCount properties

Two type parameters

- One for input, one for output, need not be the same

For each message, executes delegate specified in constructor

- Func<TInput, TOutput>
- Func<TInput, Task<TOutput> >

TransformBlock Definition (1)

```
public sealed class TransformBlock<TInput, TOutput> :
 IPropagatorBlock<TInput, TOutput>, ITargetBlock<TInput>,
 IReceivableSourceBlock<TOutput>, ISourceBlock<TOutput>,
 IDataflowBlock{
    public TransformBlock(Func<TInput, Task<TOutput>> transform);
    public TransformBlock(Func<TInput, TOutput> transform);
    public TransformBlock(Func<TInput, Task<TOutput>> transform,
                ExecutionDataflowBlockOptions dataflowBlockOptions);
    public TransformBlock(Func<TInput, TOutput> transform,
                ExecutionDataflowBlockOptions dataflowBlockOptions);
```

TransformBlock Definition (2)

```
    public Task Completion { get; }
    public int InputCount { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<TOutput> target,
                                                  bool unlinkAfterOne);
    public bool TryReceive(Predicate<TOutput> filter, out TOutput item);
    public bool TryReceiveAll(out IList<TOutput> items);
}
```

Processing Stages

Three stages:

- Filling input queue: as target, receives message into input queue
- Processing: extracts message at top of input queue and calls delegate, adding result to bottom of output queue
- Emptying output queue: sends message at head of output queue to linked targets

TransformManyBlock

Similar to TransformBlock, but output is an IEnumerable

For each input message, there may be zero or more messages in the output enumerable

Maintains two queues

- Size of each can be known via InputCount and OutputCount properties

Two type parameters

- One for input, one for output, need not be the same

TransformManyBlock Definition (1)

```
public sealed class TransformManyBlock<TInput, TOutput> :
    IPropagatorBlock<TInput, TOutput>, ITargetBlock<TInput>,
    IReceivableSourceBlock<TOutput>, ISourceBlock<TOutput>,
    IDataflowBlock {
  public TransformManyBlock(Func<TInput,
          IEnumerable<TOutput>> transform);
  public TransformManyBlock(Func<TInput,
          Task<IEnumerable<TOutput>>> transform);
  public TransformManyBlock(Func<TInput,
          IEnumerable<TOutput>> transform,
          ExecutionDataflowBlockOptions dataflowBlockOptions);
  public TransformManyBlock(Func<TInput,
          Task<IEnumerable<TOutput>>> transform,
          ExecutionDataflowBlockOptions dataflowBlockOptions);
```

TransformManyBlock Definition (2)

```
    public Task Completion { get; }
    public int InputCount { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<TOutput> target,
                                bool unlinkAfterOne);
    public bool TryReceive(Predicate<TOutput> filter,
                                out TOutput item);
    public bool TryReceiveAll(out IList<TOutput> items);
}
```

ExecutionDataflowBlockOptions

Options for blocks involved in executing actions

Adds one property to base DataflowBlockOptions

```
public class ExecutionDataflowBlockOptions : DataflowBlockOptions {
    public ExecutionDataflowBlockOptions();
    public int MaxDegreeOfParallelism { get; set; }
    public bool SingleProducerConstrained { get; set; }
}
```

MaxDegreeOfParallelism

The number of messages a block processes in parallel

- Defaults to 1
- Is a maximum
- The task scheduler has the ultimate say in how much parallelism there is

Contrast with MaxMessagesPerTask (From DataflowOptions)

Impact Of Parallelism On Ordering

TPL Dataflow blocks that are propagators output messages in the same order that they are received

- If a particular message cannot be delivered, it sits on the output queue
- Blocking delivery of other messages that follow it

For executing blocks (such as TransformBlock) that have MaxDegreesOfParallelism set to > 1, this ordering still holds

Grouping Blocks & DataflowBlock Extension Methods

Batching And Joining

GOAL: Making one output message from multiple input messages

BatchBlock

- Batching means combining instances of the same type into an array of instance of that type

JoinBlock

- Joining means combining instances of different types into a tuple

BatchedJoinBlock

BatchBlock

Sometimes we need a certain number of instances of a type in order to perform an operation

- Re-tyre a car needs four tyres
- Imagine we are receiving tyres from a warehouse
- Can't really do anything until we have all of them

Need to be able to specify:

- input type
- Batch size

Need to trigger a batch

- We decide to re-tyre a car just using what we have available (e.g. 2 tyres)

BatchBlock Definition

```
public sealed class BatchBlock<T> : IPropagatorBlock<T, T[]>,
        ITargetBlock<T>, IReceivableSourceBlock<T[]>,
        ISourceBlock<T[]>, IDataflowBlock {
    public BatchBlock(int batchSize);
    public BatchBlock(int batchSize,GroupingDataflowBlockOptions blockOptions);
    public int BatchSize { get; }
    public Task Completion { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<T[]> target, bool unlinkAfterOne);
    public void TriggerBatch();
    public bool TryReceive(Predicate<T[]> filter, out T[] item);
    public bool TryReceiveAll(out IList<T[]> items);
}
```

BatchBlock Notes

Note output type is an arry of the input type

- Type parameter to ITargetBlock and first type parameter to IPropagatorBlock is T
- Type parameter to ISourceBlock & IReceivableBlock and first type parameter to IPropagatorBlock is T[]

Batch size is a getter-only property

- Value provided in constructor
- To change batch size, complete block and create a new one

OutputCount is the number

- Value between 0 and BatchSize-1
- Goes back to 0 when a new output message is emitted

JoinBlock

Joins instances of distinct types into a tuple

There is a JoinBlock<T1, T2> and a JoinBlock<T1, T2, T3>

The JoinBlock type implements ISourceBlock but not ITargetBlock

- Type parameter to ISourceBlock is Tuple<T1, T2>
- Instead, has properties of type ITargetBlock
- One for each type
- Input messages are supplied to these, not to JoinBlock itself

JoinBlock Definition

```
public sealed class JoinBlock<T1, T2>
: IReceivableSourceBlock<Tuple<T1, T2>>,
  ISourceBlock<Tuple<T1, T2>>, IDataflowBlock {
    public JoinBlock();
    public JoinBlock(GroupingDataflowBlockOptions dataflowBlockOptions);
    public Task Completion { get; }
    public ITargetBlock<T1> Target1 { get; }
    public ITargetBlock<T2> Target2 { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<Tuple<T1, T2>> target,
                                                bool unlinkAfterOne);
    public bool TryReceive(Predicate<Tuple<T1, T2>> filter,
                                                out Tuple<T1, T2> item);
    public bool TryReceiveAll(out IList<Tuple<T1, T2>> items);
}
```

BatchedJoinBlock

A mix of the batch concept and the join concept

- There is a BatchedJoinBlock<T1, T2> and a BatchedJoinBlock<T1, T2, T3>

Joins batches of instances of distinct types

Implements ISourceBlock & IReceivableBlock

- Type parameter is Tuple< IList<T1>, IList<T2> >
- A tuple whose items are lists

Provides properties that implement ITargetBlock

- One each for T1, T2 and (for the T3 type) T3

BatchedJoinBlock Definition

```
public sealed class BatchedJoinBlock<T1, T2> :
        IReceivableSourceBlock<Tuple<IList<T1>, IList<T2>>>,
        ISourceBlock<Tuple<IList<T1>, IList<T2>>>, IDataflowBlock {

    public BatchedJoinBlock(int batchSize);

    public BatchedJoinBlock(int batchSize, GroupingDataflowBlockOptions options);

    public int BatchSize { get; }

    public Task Completion { get; }

    public int OutputCount { get; }

    public ITargetBlock<T1> Target1 { get; }

    public ITargetBlock<T2> Target2 { get; }

    public void Complete();

    public IDisposable LinkTo(ITargetBlock<Tuple<IList<T1>, IList<T2>>> target,
                                                  bool unlinkAfterOne);

    public bool TryReceive(Predicate<Tuple<IList<T1>, IList<T2>>> filter,
                                                  out Tuple<IList<T1>, IList<T2>> item);

    public bool TryReceiveAll(out IList<Tuple<IList<T1>, IList<T2>>> items);
}
```

BatchedJoinBlock Notes

BatchSize refers to list size for each type

To get output, need batchsize of instances of each target type

OutputCount is a count of what?

What is List<T> used here, whereas T[] is used with BatchBlock

GroupingDataflowBlockOptions

Used to influence how grouping blocks work

- If none provided, defaults used

```
public class GroupingDataflowBlockOptions : DataflowBlockOptions {
    public GroupingDataflowBlockOptions();
    public bool Greedy { get; set; }
    public long MaxNumberOfGroups { get; set; }
}
```

Greedy

A block is greedy when it accepts all messages offered to it

- Regardless of what it can do at that time
- Default is true (this block is greedy)

A block is not greedy when:

- It postpones accepting messages offered to it until it has been offered enough messages in order to create an output message
- When this happens, it tries to reserve all necessary messages
- If sucessfully, it consumes all messages and emits output message
- If unsuccessful, it releases reservations
  (think two-phase commit)

Is A Greedy Dataflow Block Good Or Bad?

For humans, sharing is considered good, being greedy is not

With dataflow, the picture is not so clear cut

- “Is a greedy block good or bad?” we need to ask

With a multi-core system, a good rule of thumb - if you are in a position to do productive work, do the work

- CPU cache lines are more likely to be populated
- Page cache more likely to match what you want
- Avoid context switching

If your block can't do productive work (e.g. need to wait for something extra before progressing), then don't cause problems for other blocks