TPL Dataflow

A brand new world for async

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

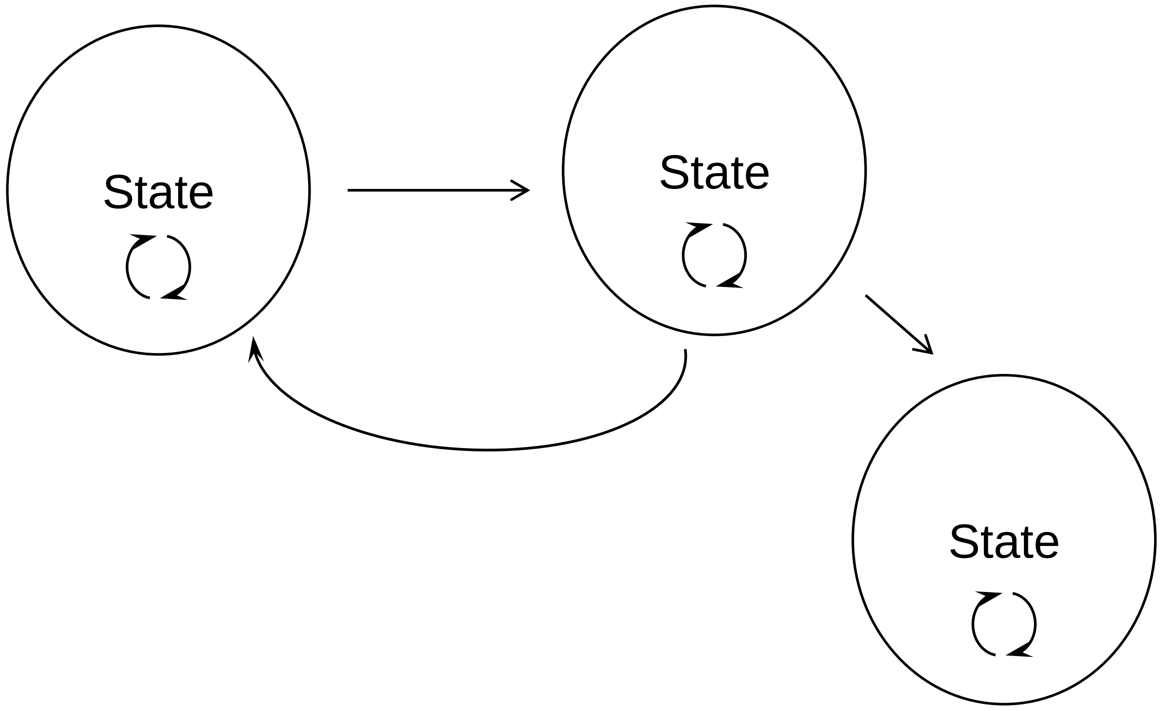
- Motivation
- What is TPL Dataflow, how to get it
- Asynchronous programming with blocks

Motivation

- Asynchronous programming evolved from synchronous programming
- Asynchronous programming = Synchronous programming
  + Tasks
  + Synchronization
- Evolution lead to complexity
- Real world is composed of many autonomous things
- Concurrent systems should perhaps more closely model the real world.

Imagine a world

- No mutable shared state
  - No need for locks and semaphores
- Just autonomous objects communicating via messages

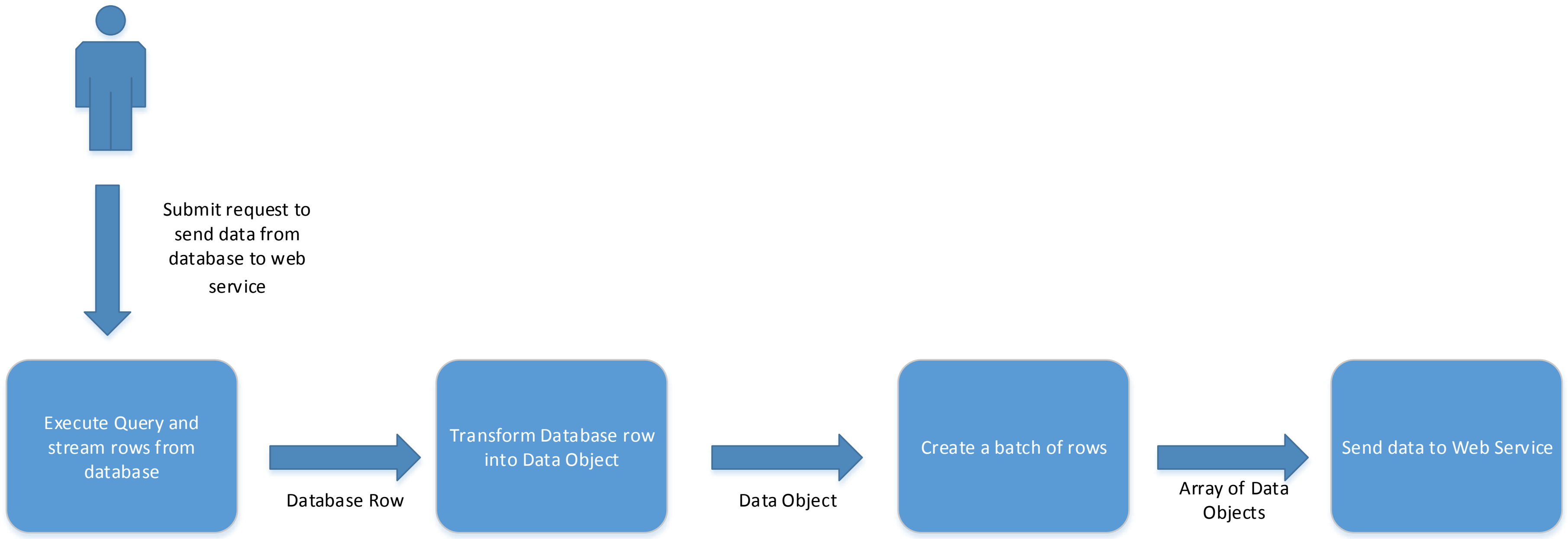


Row processing example

- Repeat until all rows processed
  - Load Row from database
  - Transform the data
  - Add to a batch
  - When batch reaches given size send to web server
- How to parallelise
  - Reading from the database is sequential
  - Can parallelise transformation
    - What if order matters ?
  - Need to safely collate results

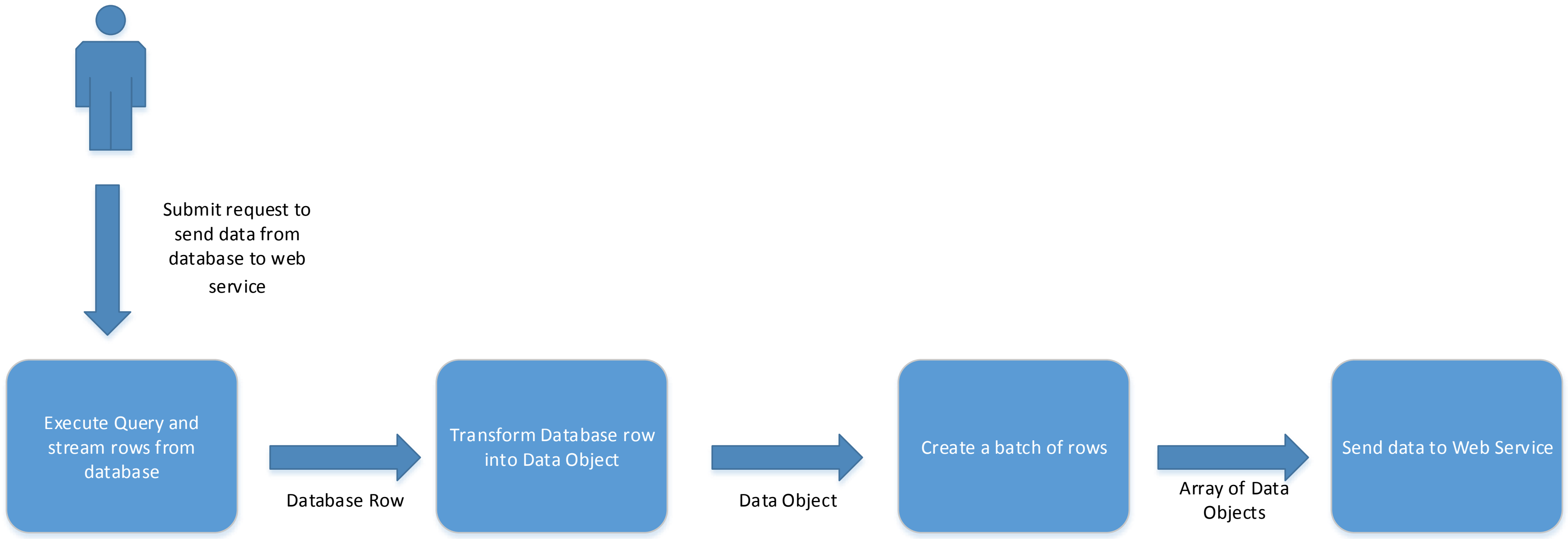
Autonomous blocks

- Each block has its own thread
- While a message is being transformed another is being fetched from the database.
- This is akin to Henry Ford's production line


Submit request to send data from database to web service
Execute Query and stream rows from database
Database Row
Transform Database row into Data Object
Data Object
Create a batch of rows
Array of Data Objects
Send data to Web Service

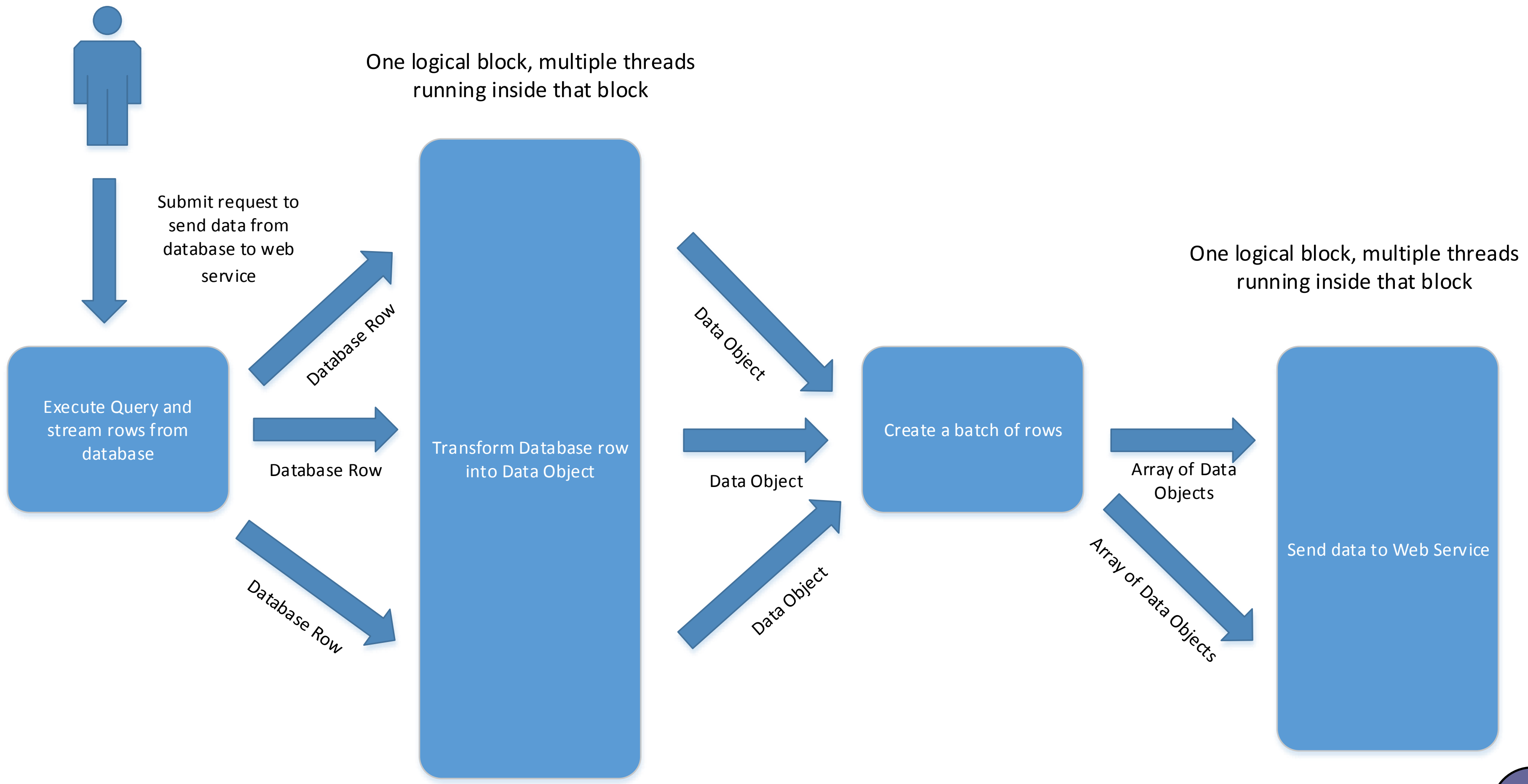
Blockages

- What if it took 3 times longer to transform a row than fetch it ?


Submit request to send data from database to web service
Execute Query and stream rows from database
Database Row
Transform Database row into Data Object
Data Object
Create a batch of rows
Array of Data Objects
Send data to Web Service

Balanced pipe line

- Same structure
  - Many threads per block
  - Message order is preserved $I_1, I_2, I_3 \Rightarrow O_1, O_2, O_3$



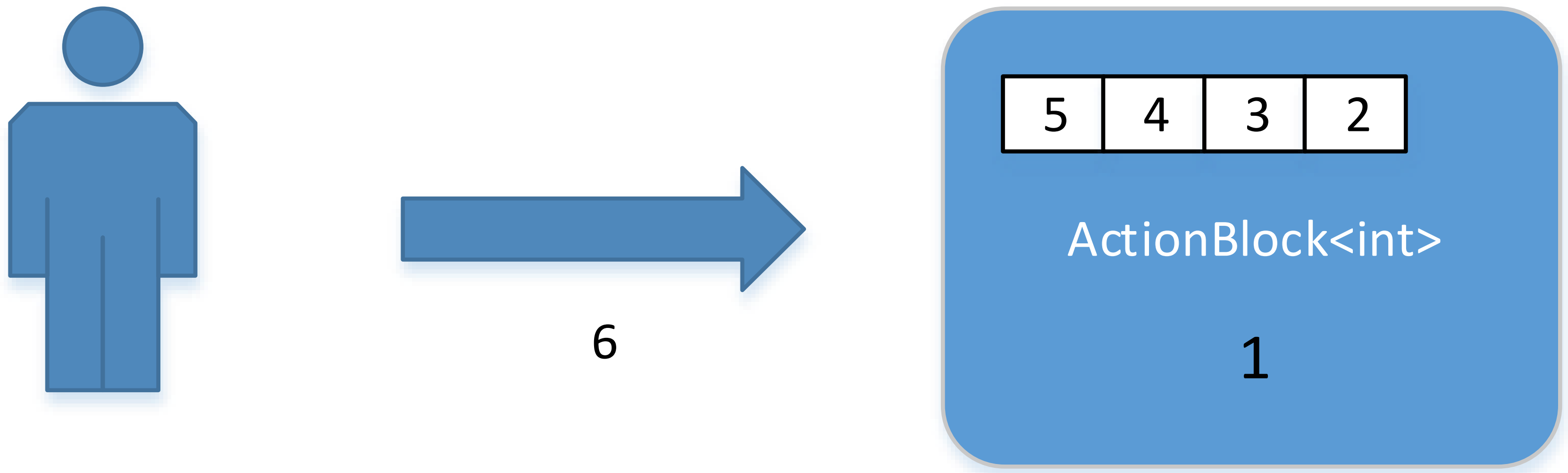
One logical block, multiple threads running inside that block
Submit request to send data from database to web service
Execute Query and stream rows from database
Database Row
Database Row
Database Row
Transform Database row into Data Object
Data Object
Data Object
Data Object
Create a batch of rows
One logical block, multiple threads running inside that block
Array of Data Objects
Array of Data Objects
Send data to Web Service

TPL Dataflow

- Does not ship as core part of framework
  - Nuget package
- Provides abstraction over TPL to implement data flow style programming

Blocks

- Can be both targets and sources of messages
- A block provides the logic to perform its behaviour
  - Sometimes parts supplied via delegates
- By default only a single task will execute in a block
- Contain an unbounded buffer to receive messages while processing previous message



Producer Consumer

- Post asynchronously sends a message to a block
- When busy queues messages
- When no messages to process, no task is running

```
var consumerBlock = new ActionBlock<int>(new Action<int>(Consume));

   for (int i = 0; i < 5; i++)
   {
      consumerBlock.Post(i);
      Thread.Sleep(1000);
   }

    // Tell the block no more items will be coming
    consumerBlock.Complete();
    // wait for the block to shutdown
    consumerBlock.Completion.Wait();

. . .
private static void Consume(int val) { ... }
```

Linking Blocks

- Isolated blocks not that interesting
- Blocks can be linked together to produce data flows
- Many types of blocks out of the box
  - Execution Blocks
    - ActionBlock<T>
    - TransformBlock<TInput,TOuput>
    - TransformManyBlock<Tinput,Toutput>
  - Glue Blocks
    - BufferBlock<T>
    - BatchBlock<T>
    - BroadcastBlock<T>
    - WriteOnce<T>
    - JoinBlock<T1,T2>
    - JoinBlock<T1,T2,T3>
    - BatchedJoinBlock<T1,T2>
    - BatchedJoinBlock<T1,T2,T3>

Image processing example

- Apply image processing to a file based image and then show on screen
- Obviously execute asynchronously to keep the UI running, could use raw TPL or Dataflow

```
var loadAndToGreyBlock = new TransformBlock<string, BitmapSource>(
                            (Func<string, BitmapSource>)LoadAndToGrayScale);

var publishImageBlock = new ActionBlock<BitmapSource>(
                        (Action<BitmapSource>) PublishImage,
                       new ExecutionDataflowBlockOptions(){
                        TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
                       });

loadAndToGreyBlock.LinkTo(publishImageBlock);
```

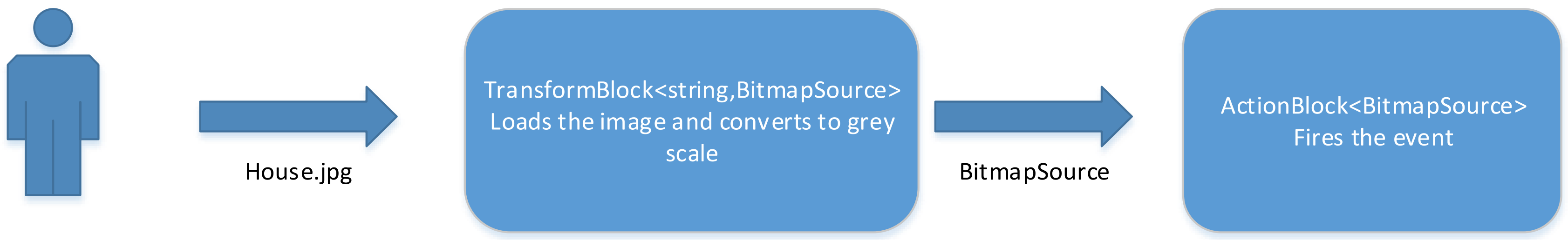
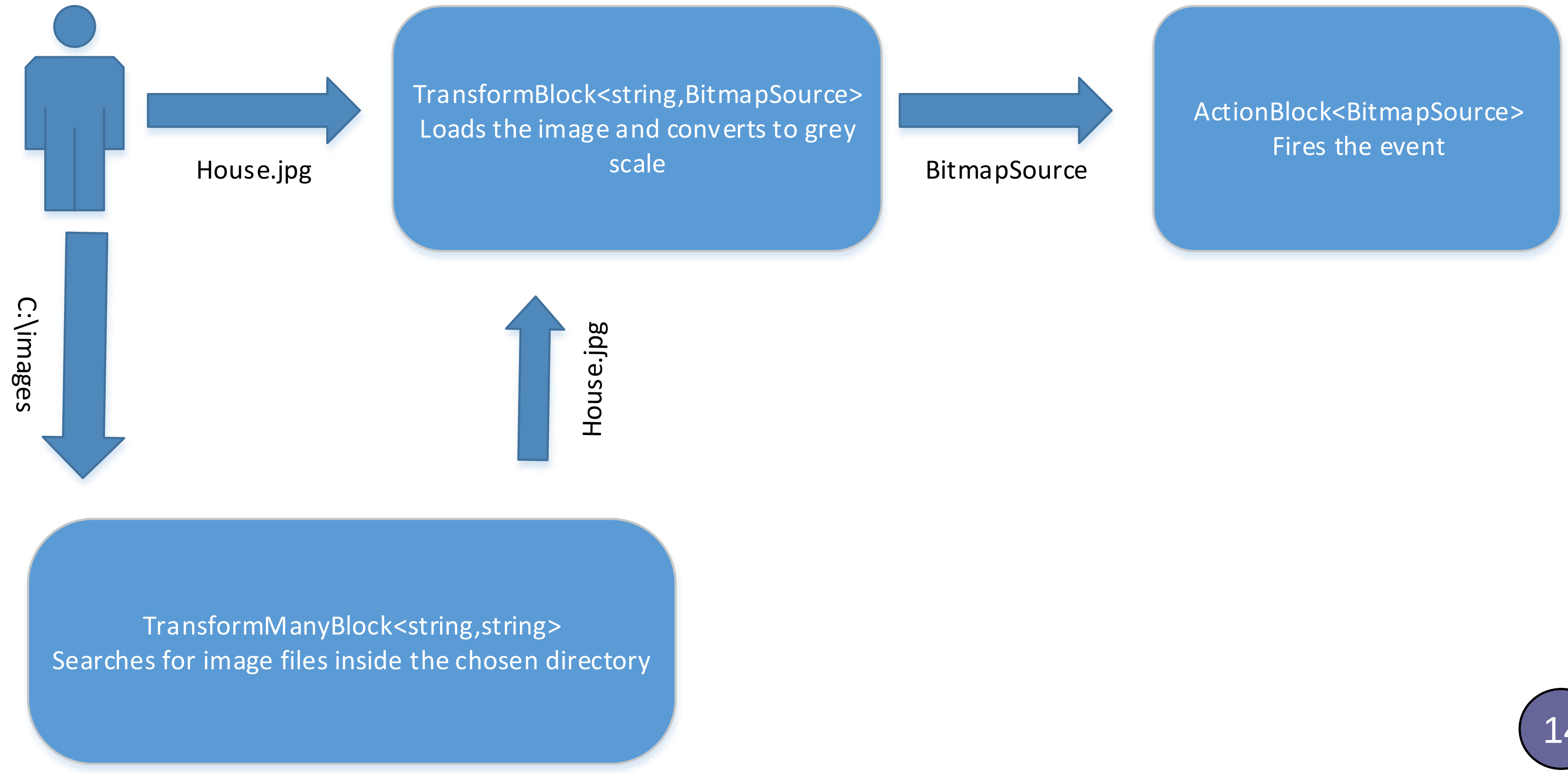

House.jpg
TransformBlock<string,BitmapSource>
Loads the image and converts to grey scale
BitmapSource
ActionBlock<BitmapSource>
Fires the event

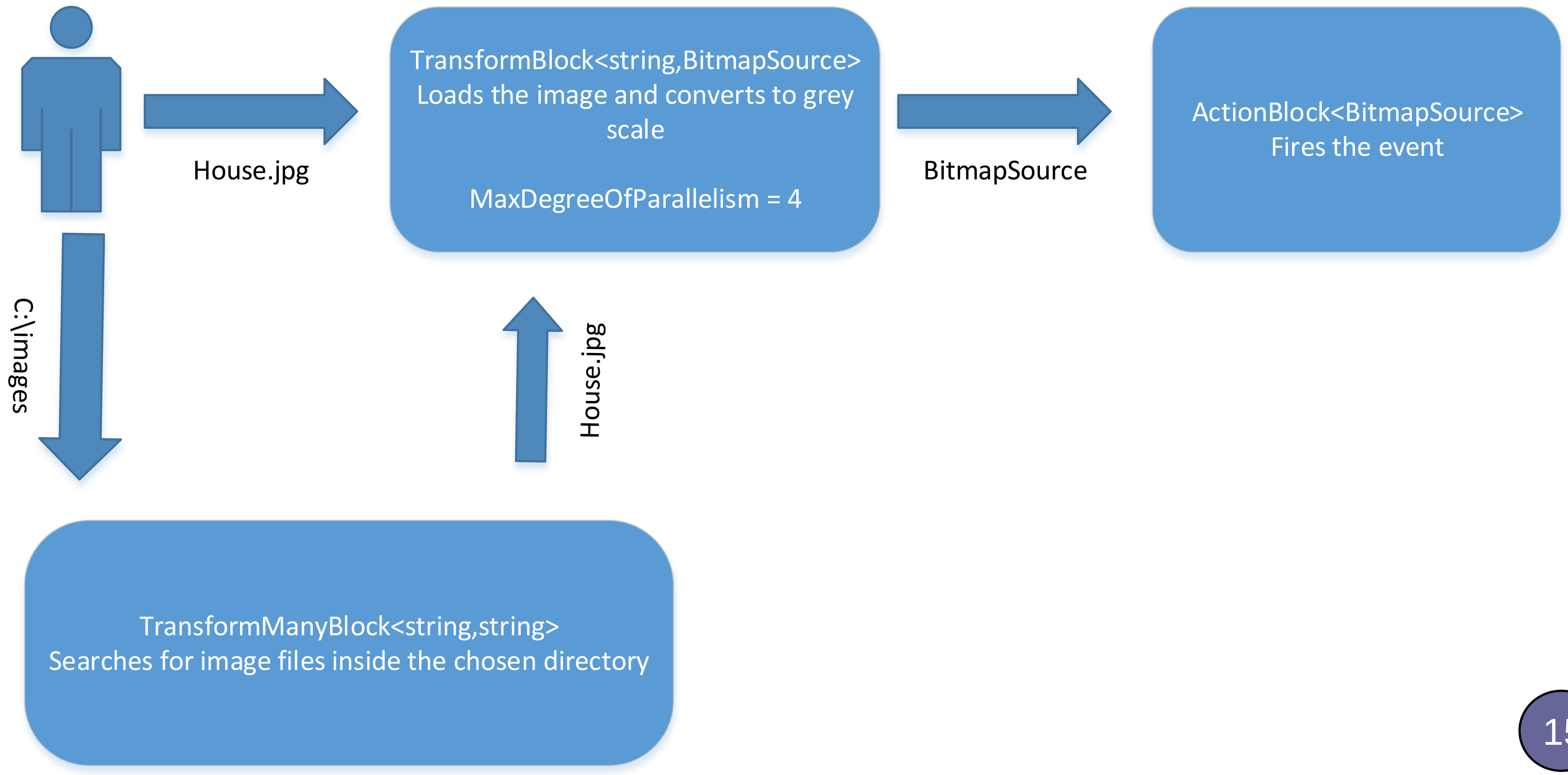
Image processing directories example

- Process directories of images asynchronously
- TransformMany equivalent to SelectMany in Linq


TransformBlock<string,BitmapSource>
Loads the image and converts to grey scale
ActionBlock<BitmapSource>
Fires the event
House.jpg
BitmapSource
C:\images
House.jpg
TransformManyBlock<string,string>
Searches for image files inside the chosen directory

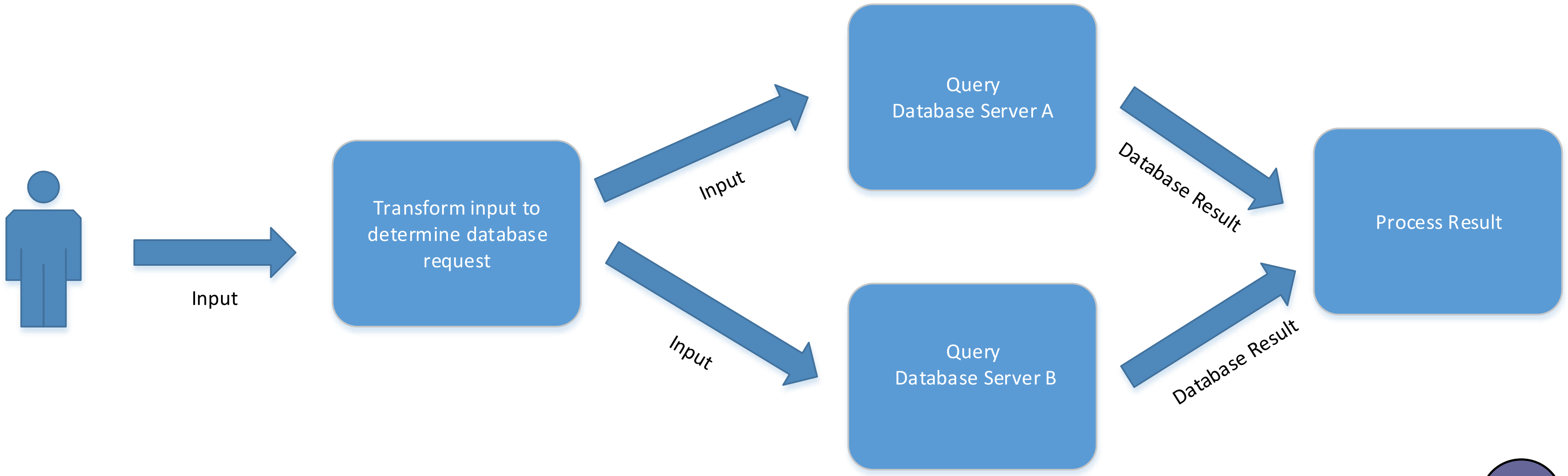
Preserving message order

- Multiple threads performing image processing
- Output order ALWAYS same as input order


TransformBlock<string,BitmapSource>
Loads the image and converts to grey scale
MaxDegreeOfParallelism = 4
House.jpg
BitmapSource
ActionBlock<BitmapSource>
Fires the event
C:\images
House.jpg
TransformManyBlock<string,string>
Searches for image files inside the chosen directory

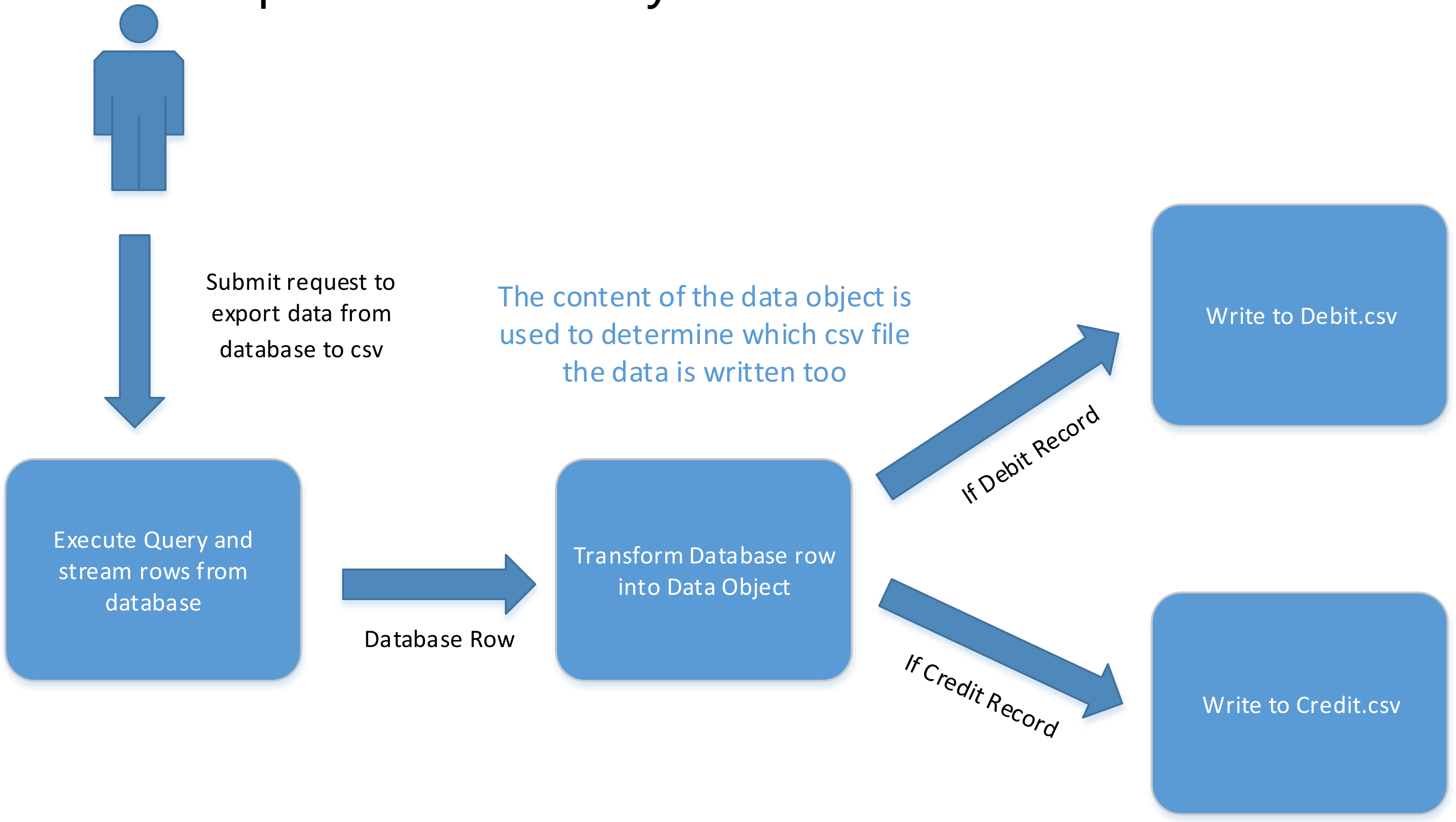
Linking to multiple targets

- Not just simple pipe lines
- Messages offered to each target in turn
  - Only one target gets the message
- Source will block until message has been delivered
- Blocks often configured greedy, which means always accepts
- Set BoundedCapacity to 1 to enable non greedy



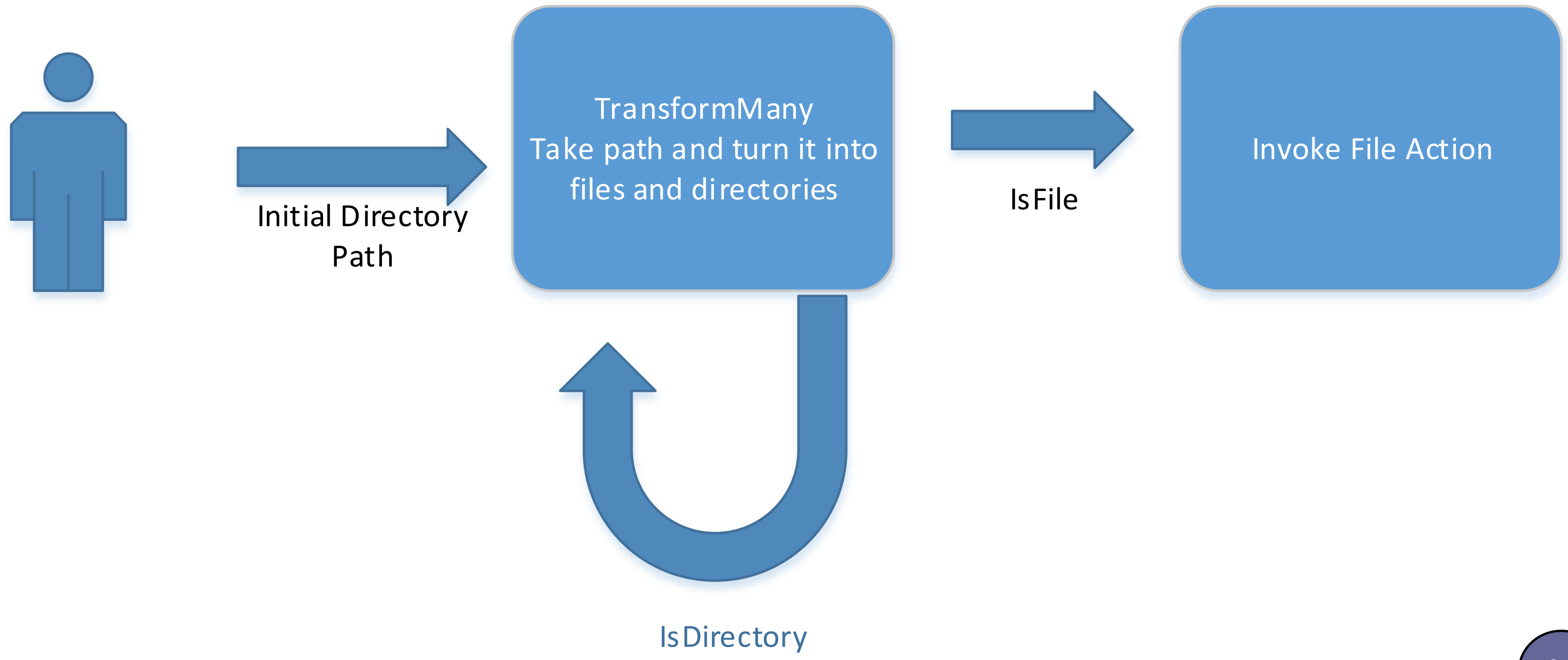
Conditional Linking if/elseif/else

- LinkTo method takes a predicate to decide if target should receive
- **WARNING** no matching target, source will block forever
  - Good practice to always have an unconditional link


Submit request to export data from database to csv
The content of the data object is used to determine which csv file the data is written too
Write to Debit.csv
If Debit Record
Execute Query and stream rows from database
Database Row
Transform Database row into Data Object
If Credit Record
Write to Credit.csv

Conditional Linking Recursion

- Sources can link back to themselves for recursive style programming



Graceful shutdown

- A block is told not to receive any more input by calling its Complete method
- Each block has a Task representing completion
  - Accessed via the blocks Completion property
  - Used to observe the outcome of the block
    - RanToCompletion,Faulted,Cancelled

```
var actionBlock = new ActionBlock<int>((Action<int>) Console.WriteLine);

for (int i = 0; i < 10; i++)
{
  actionBlock.Post(i);
}

 Console.WriteLine("Completing..");
            actionBlock.Complete();
            Console.WriteLine("Waiting..");
            actionBlock.Completion.Wait();
            Console.WriteLine(actionBlock.Completion.Status);
```

Propagating shutdown

- Calling complete on each block would be tedious
- Completed blocks can be configured to flow completion
  - on a per link basis
- Complete the start, wait for the end

```
var firstBlock = new TransformBlock<int, int>(i => i*2);
var secondBlock = new ActionBlock<int>( Console.WriteLine );


firstBlock.LinkTo(secondBlock, new DataflowLinkOptions()
                       {PropagateCompletion = true});
for (int i = 0; i < 10; i++)
{
  firstBlock.Post(i);
}

firstBlock.Complete();
secondBlock.Completion.Wait();
```

Error Handling

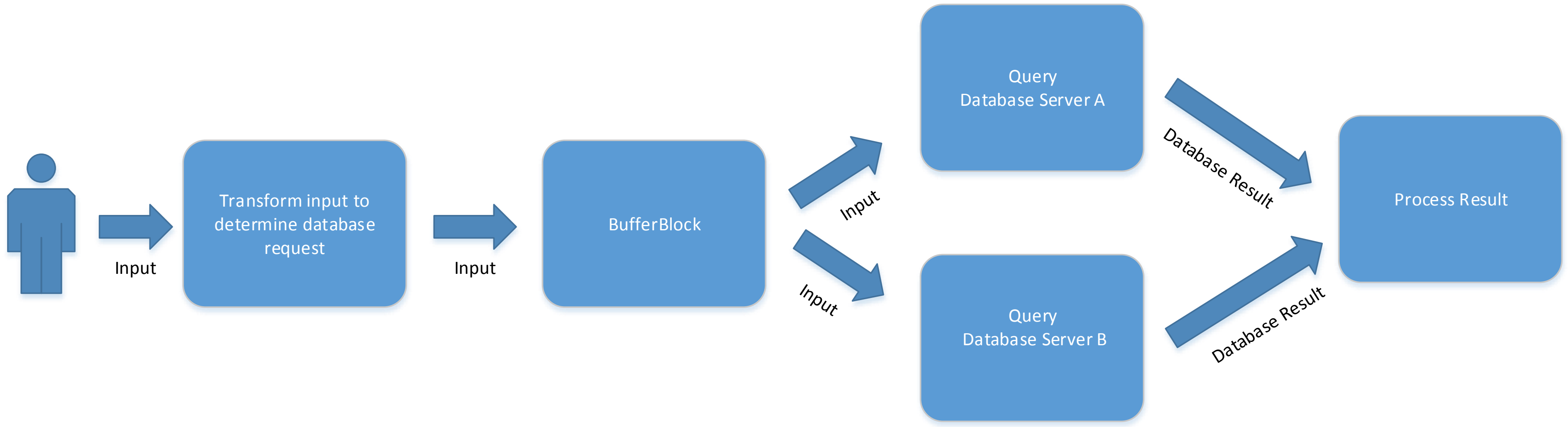
- Execution blocks run code and as such can fail with an exception
- How to handle an exception inside a block
  - If recoverable, try/catch inside the block and recover
  - If non recoverable let exception propagate from the block
    - Block is now in faulted state will not process any more messages
    - Use PropagateCompletion to pass on the error to linked blocks to provide ordered shutdown

Glue Blocks

- Glue blocks provide common network functionality
  - Shared buffer for load balancing
  - Batching messages for more efficient processing
  - Broadcasting many receivers, always get last result
  - WriteOnce, first result wins (readonly variable)

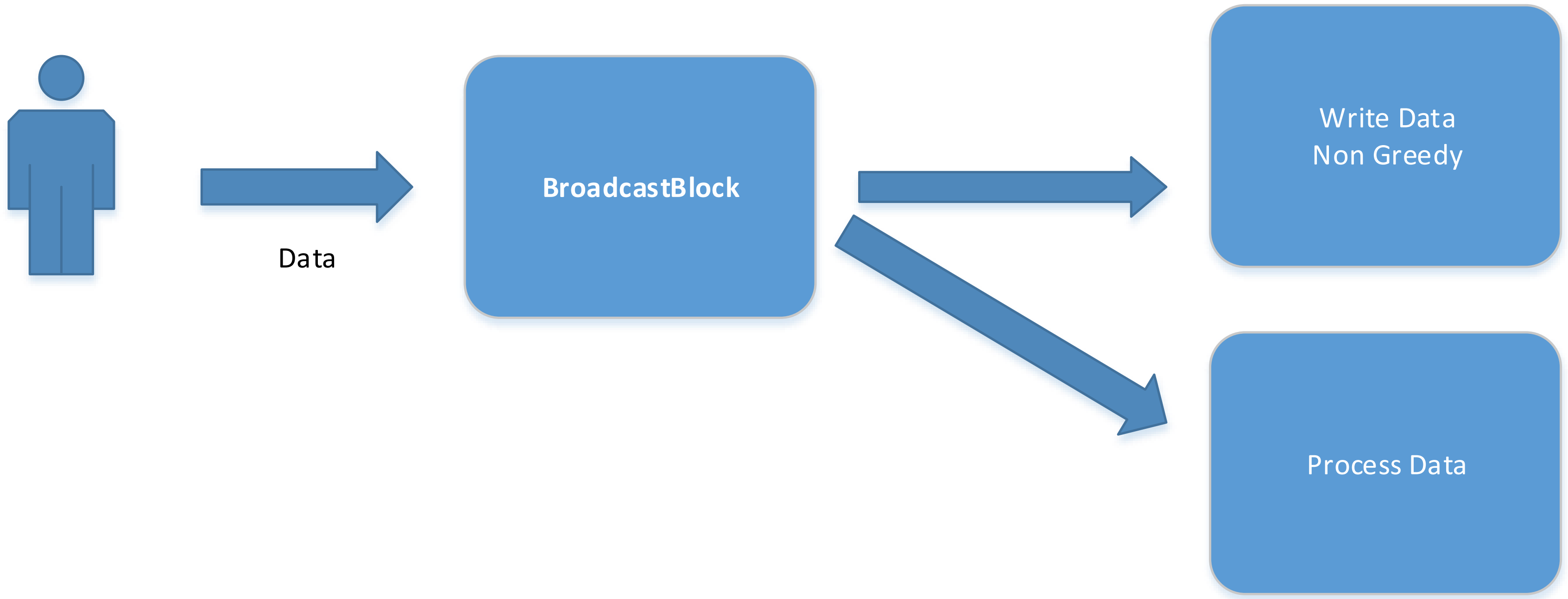
Buffer Block

- Shared buffer, enables load balancing when execution block is non greedy



Broadcast Block

- One of the only block that delivers identical message to multiple blocks
- Must provide copy method
- Useful for providing best effort in processing



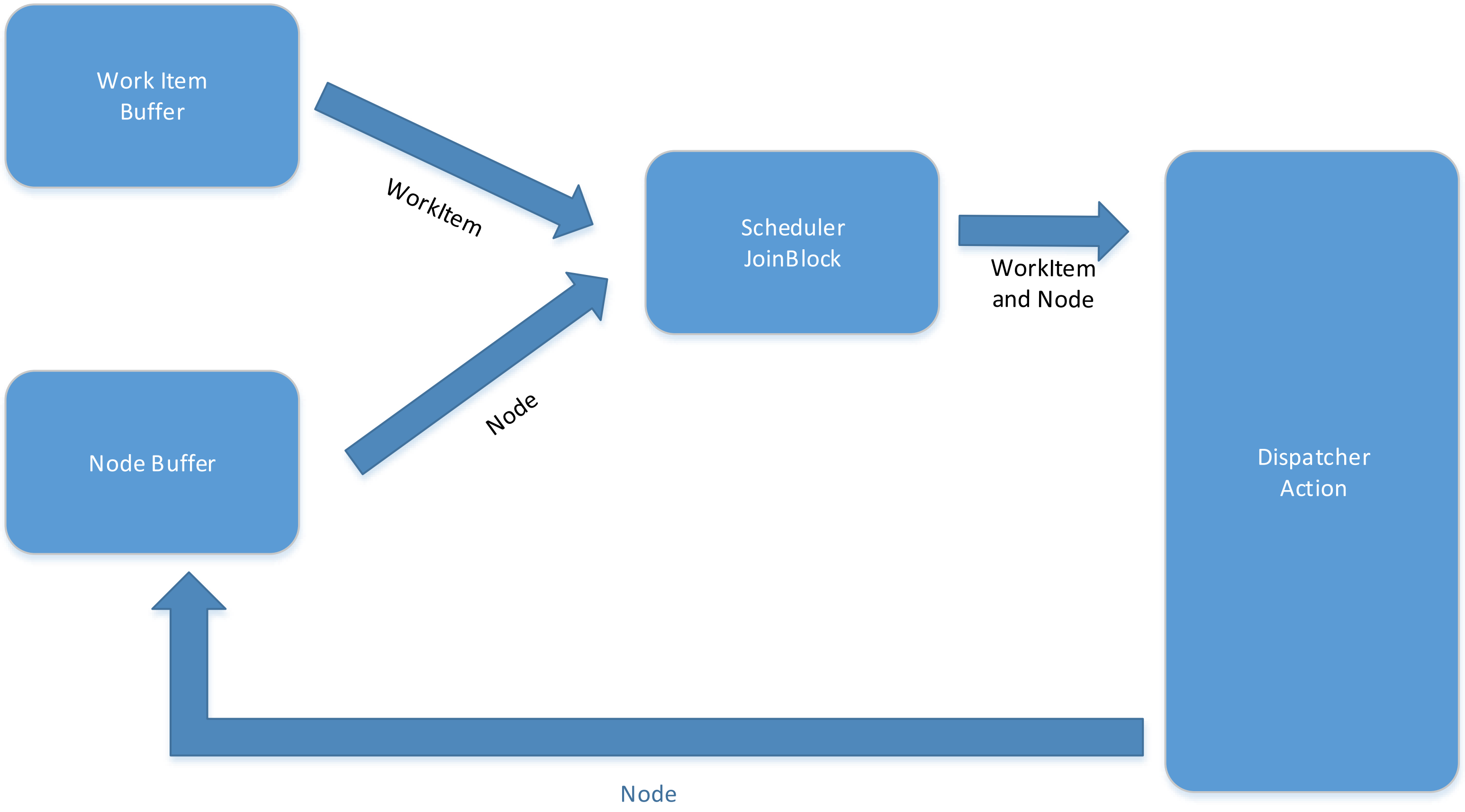
Batch Block

- More efficient to process collection of messages
  - Sending messages to web service
- Sampling

```
 int batchSize = 100;
 var batcher = new BatchBlock<int>(batchSize);
 var averager = new TransformBlock<int[], double>(
                       values => values.Average());

var currentAverage = new BroadcastBlock<double>(i => i);

batcher.LinkTo(averager);
averager.LinkTo(currentAverage);

var rnd = new Random();
while (true) {
 batcher.Post(rnd.Next(1, 100));

 if (Console.KeyAvailable &&
     Console.ReadKey(true).Key == ConsoleKey.A){
    Console.WriteLine(currentAverage.ReceiveAsync().Result);
 }
}
```

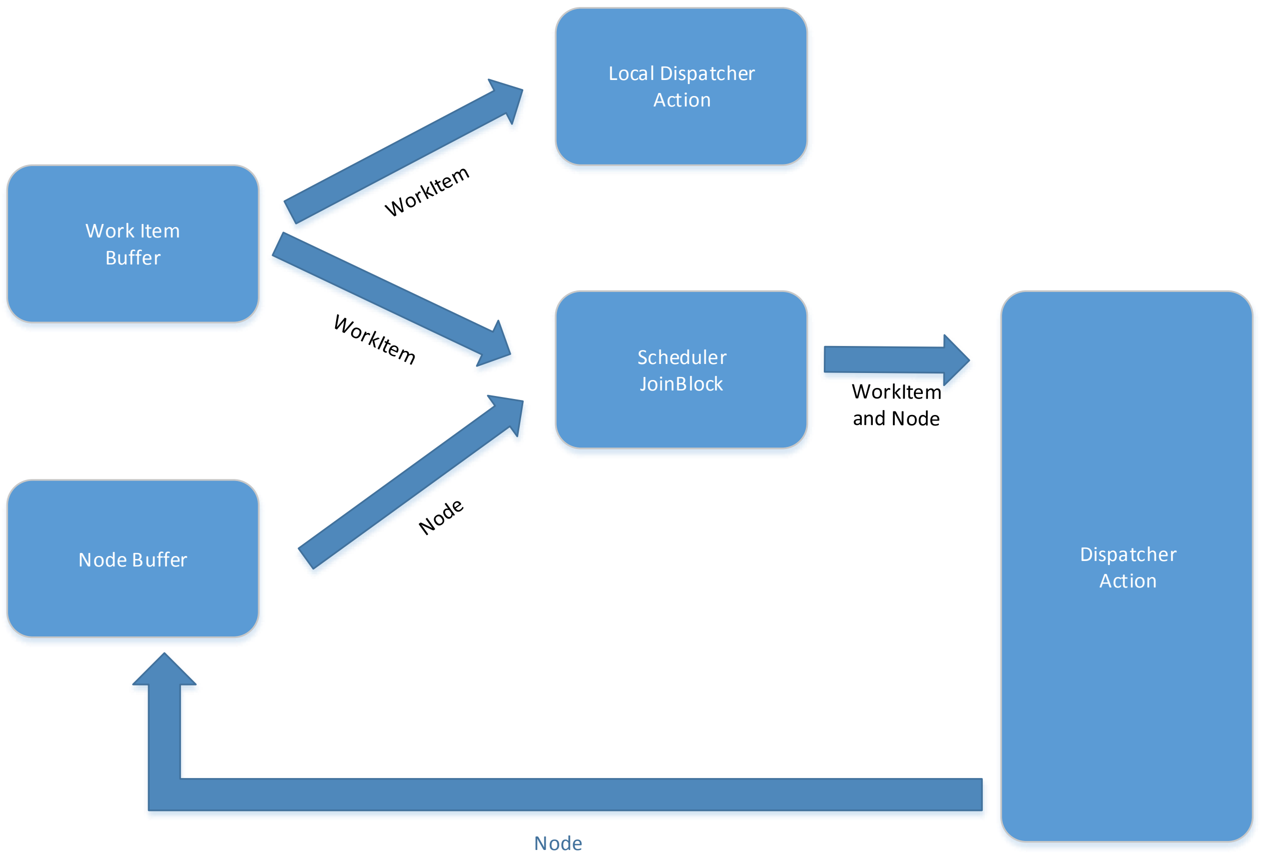
Join Block

- Requires 2 or 3 message sources to offer a message
- Produces a Tuple of combined messages



Be careful of greedy joins

- Scheduler could consume work item message even if no node message is available.
- Configure it to be non greedy, so that it only consumes if WorkItem and Node message available



Asynchronous blocks

- Execution blocks use tasks
  - Having a task block is not good mojo
- Execution blocks can take advantage of async/await
  - Enables block to give up thread while waiting for IO
- Still enforces MaxDegreeOfParallelism

```
var downloadAndPrintBlock =
      new ActionBlock<string>(async url =>
      {
        var client = new WebClient();
        string content = await client.DownloadStringTaskAsync(url);
        Console.WriteLine(content);
      });

downloadAndPrintBlock.Post("http://www.bbc.co.uk");
downloadAndPrintBlock.Post("http://www.google.com");
downloadAndPrintBlock.Post("http://www.develop.com");
Console.ReadLine();
```

Summary

- An alternative approach to classical multi threaded programming
- Code structure closer to real problem domain
  - Easier to visualize
- Simpler asynchronous programming
  - Look no locks
- Integrates with Reactive framework