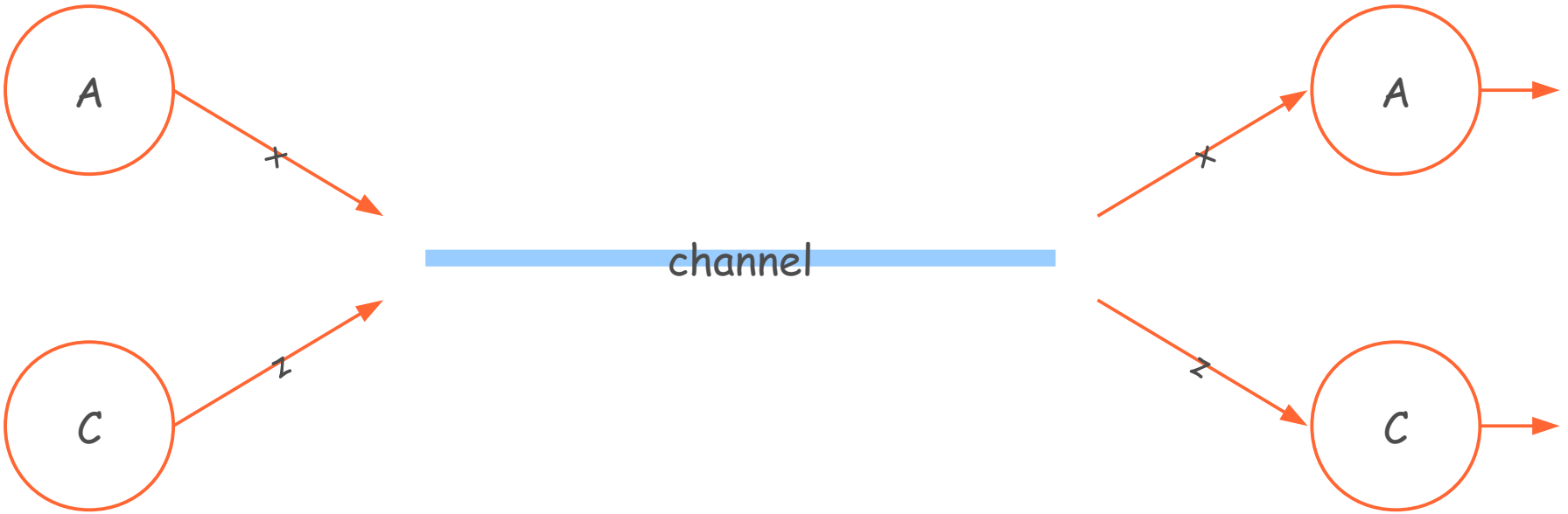
# A Framework for the New Modal and Sine Bank Modules

# Multiplexing
# in Reaktor

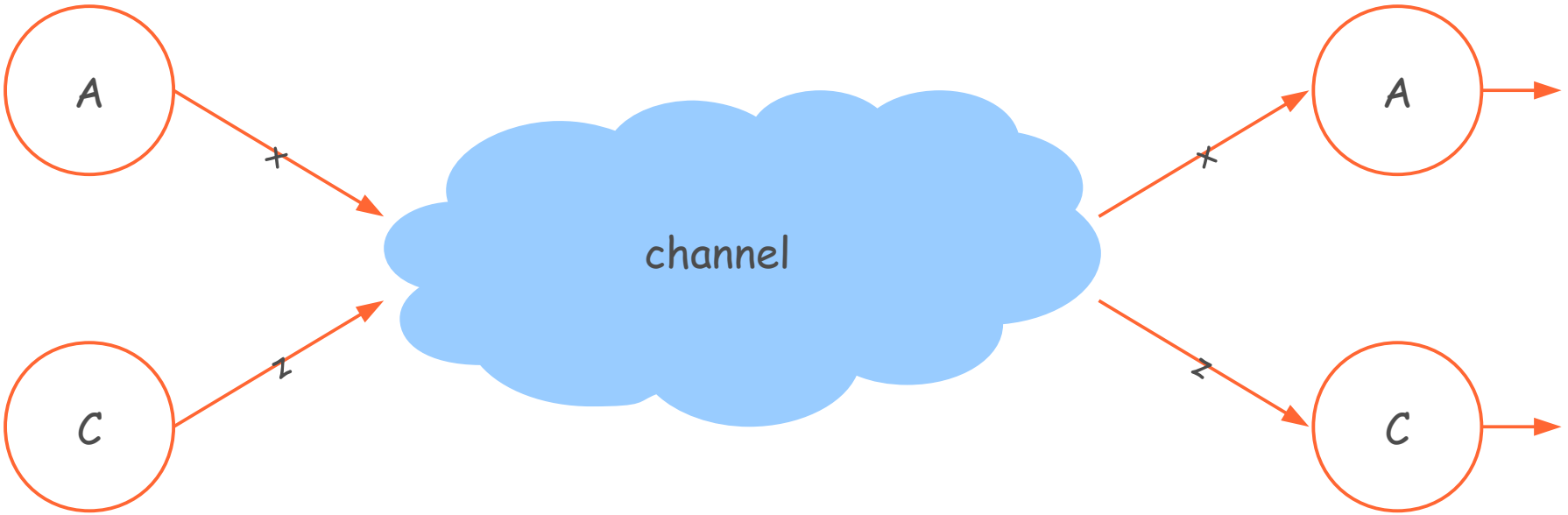
# What is Multiplexing?

Multiplexing is a technique for transmitting different types (A, C) of signals (x, z) over the same channel.



# What is Multiplexing?

A channel could basically be anything! In Reaktor, it might be a single wire, a bunch of wires, an Event Table, etc.



# Why Multiplexing?

A wire in Reaktor is a great thing. It visualizes data flow from one module to another module. Reaktor's approach is very pure - one wire equals one type of signal, one number.

# Why Multiplexing?

A wire in Reaktor is a great thing. It visualizes data flow from one module to another module. Reaktor's approach is very pure - one wire equals one type of signal, one number.

Sometimes that's not really what you want, though. Often the "one wire – one signal" approach provides more information than necessary which can make working with Reaktor slow, confusing and actually clashes with some of Reaktor's technical limitations.

# Why multiplexing ?

From a purely technical point of view, reducing the number of wires can become a necessity in Reaktor.

# Why Multiplexing?

From a purely technical point of view, reducing the number of wires can become a necessity in Reaktor.

As you know, a Core Cell has the limitation of 40 inports and outports. Usually this would limit the number of independent control signals we could send into the Core Cell.

# Why Multiplexing?

From a purely technical point of view, reducing the number of wires can become a necessity in Reaktor.

As you know, a Core Cell has the limitation of 40 inports and outports. Usually this would limit the number of independent control signals we could send into the Core Cell.

The only way to get more signals into the Cell we would pretty much have to send more independent signals over these 40 wires, right ?

# Why Multiplexing?

From a purely technical point of view, reducing the number of wires can become a necessity in Reaktor.

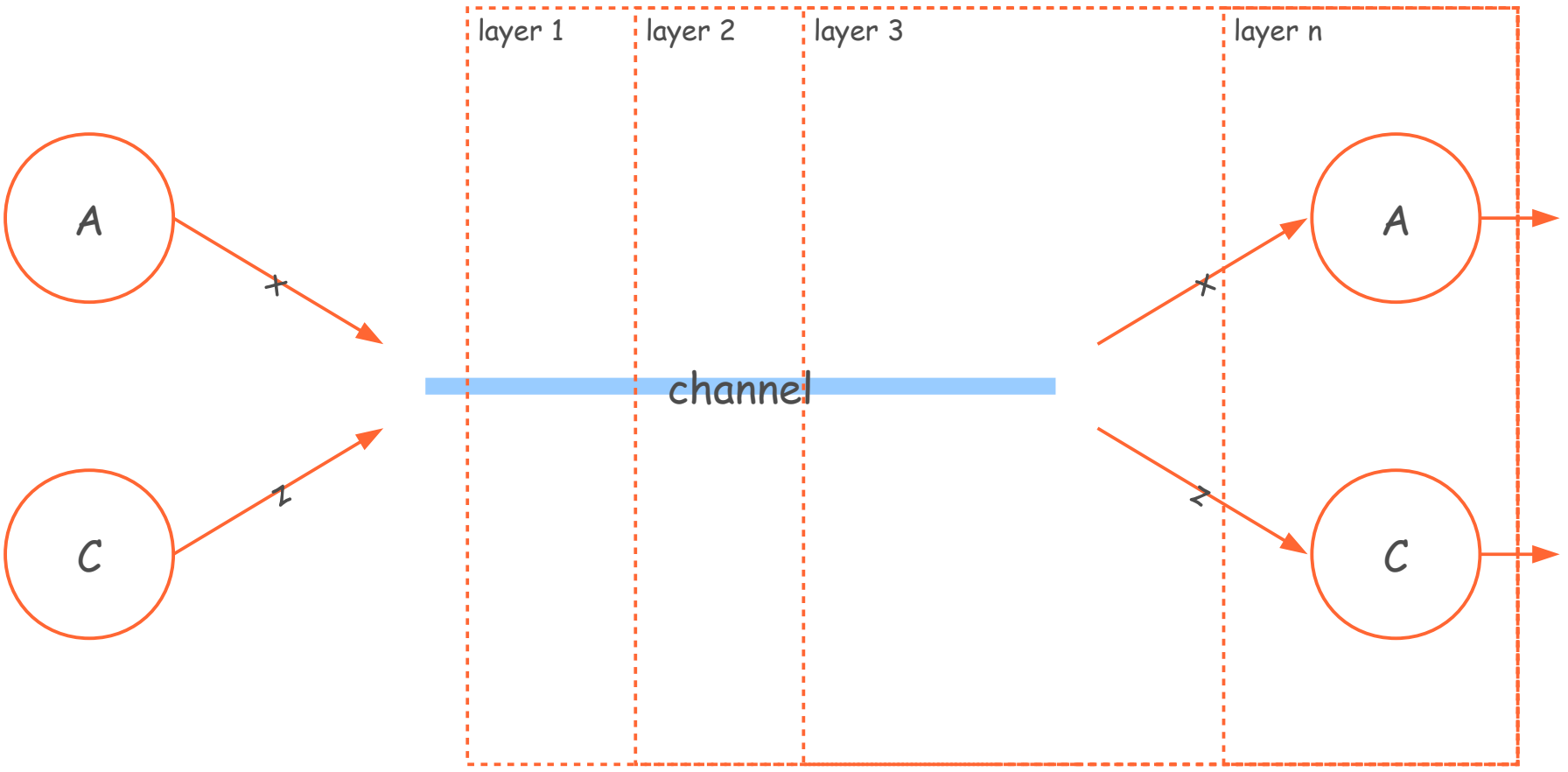
As you know, a Core Cell has the limitation of 40 inports and outports. Usually this would limit the number of independent control signals we could send into the Core Cell.

The only way to get more signals into the Cell we would pretty much have to send more independent signals over these 40 wires, right ?

Here is where multiplexing can come in handy. These techniques are used all over the framework to reduce the number of wires and to "hide" the information until it reaches the point where it belongs...

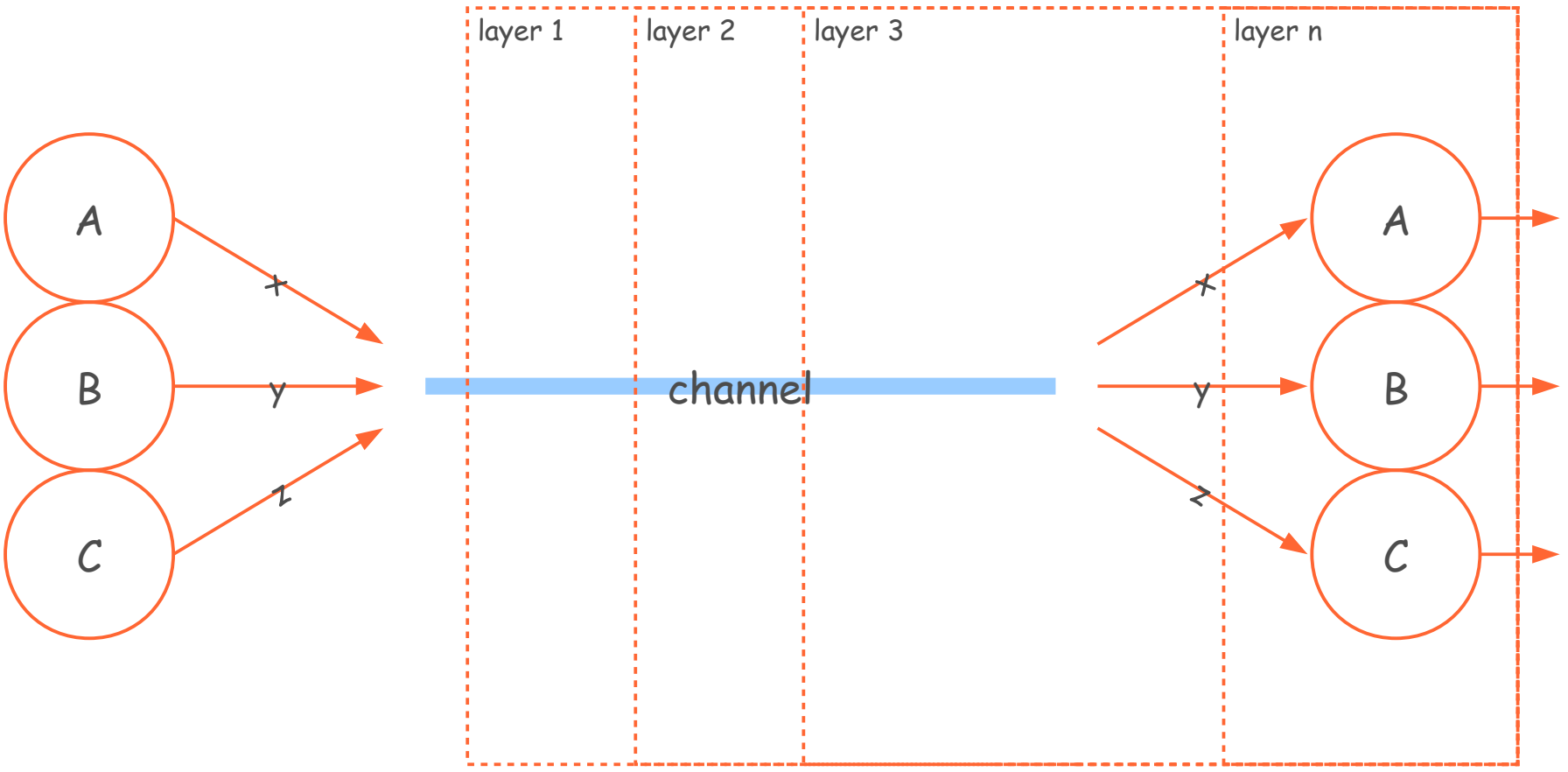
# Why Multiplexing?

Multiplexing allows, e.g. to provide the events of several event sources at a deeper Macro layer without the need for wires and ports for each event source in every layer.



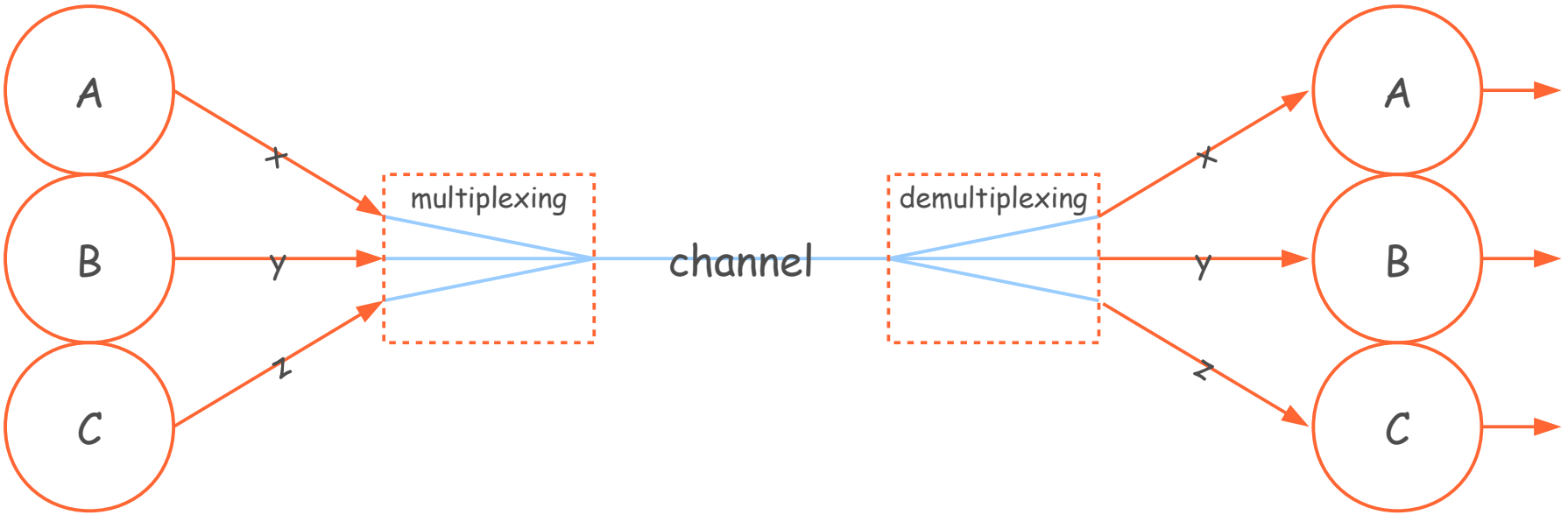
# Why Multiplexing?

Also, adding an event source to such a Structure would only require a change at both ends of the channel and not in the Macro layers in between.



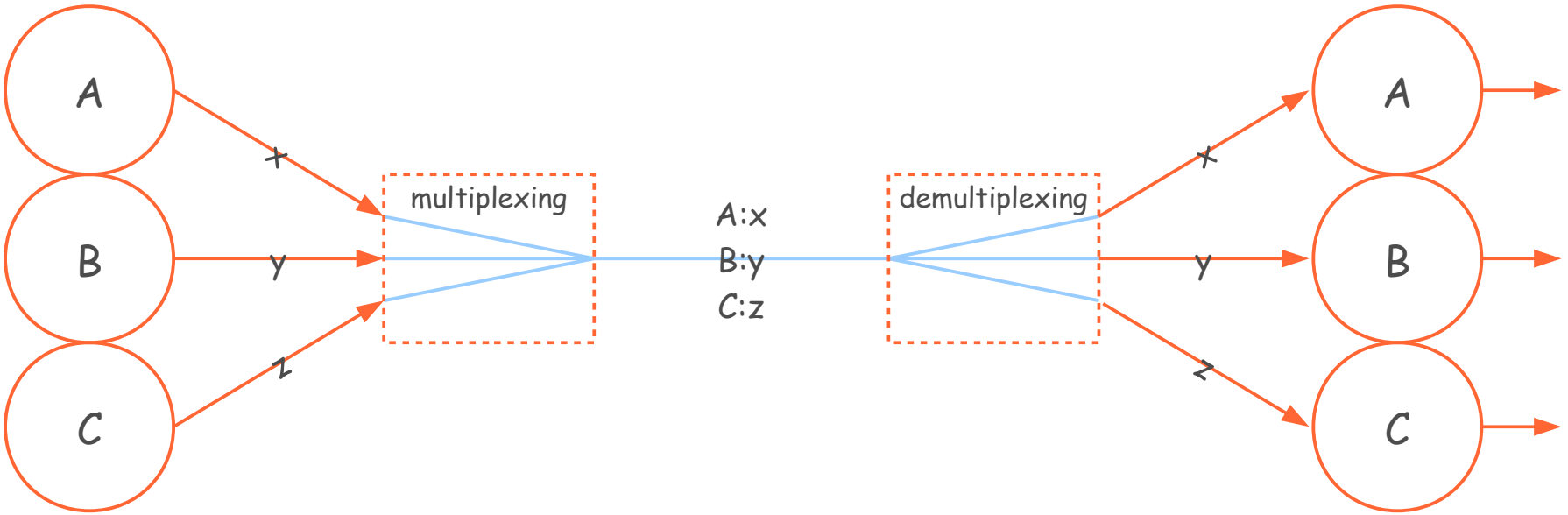
# Multiplexing Basics

To send different types of signals over the channel there have to be processing stages capable of merging and unmerging signals. These stages are called multiplexing (mux) and demultiplexing (demux) stage.



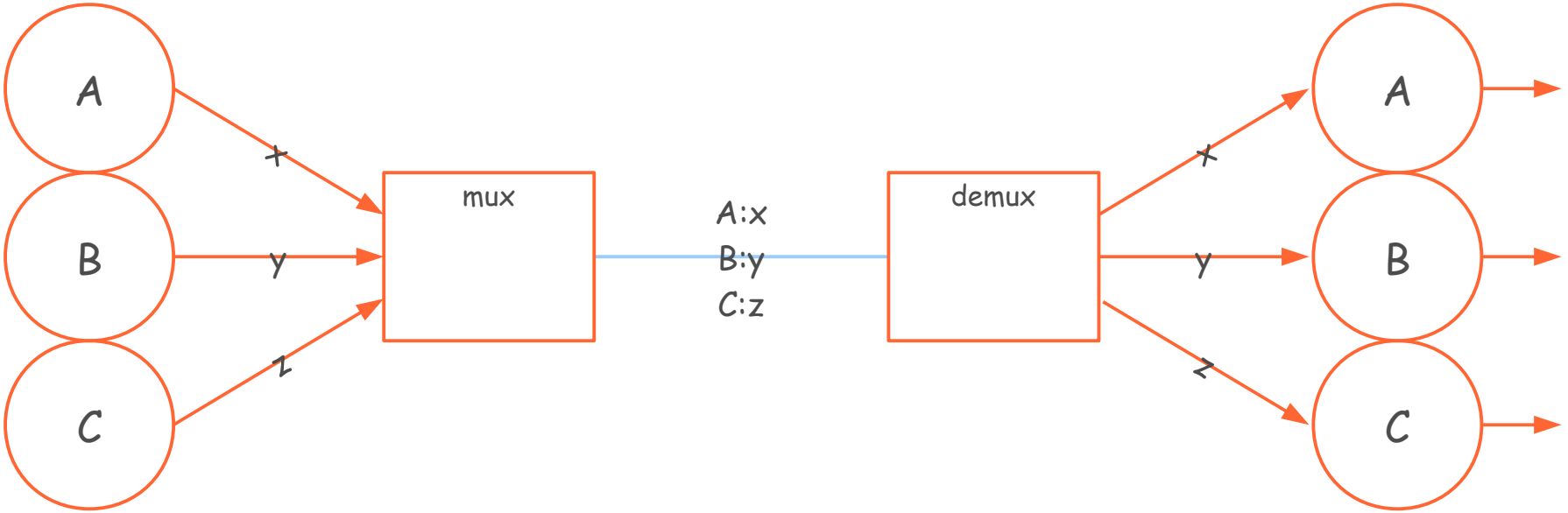
# Multiplexing Basics

The multiplexing stage has to wrap the data in a certain way to allow the demultiplexing stage to relate each signal (x, y, z) to its type (A, B, C), so these stages have to agree on some rule how multiplexed data has to look like.



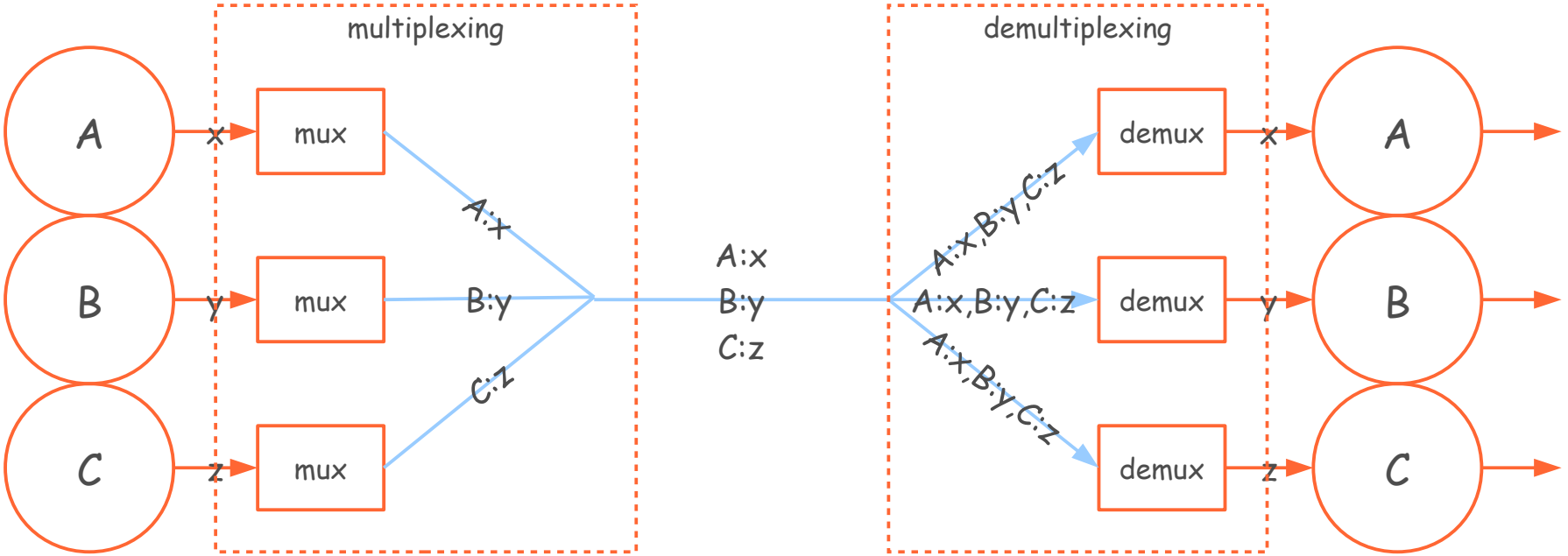
# Multiplexing Basics

These two stages might really be just two Macros working much like routers. This would be quite efficient but not that flexible...



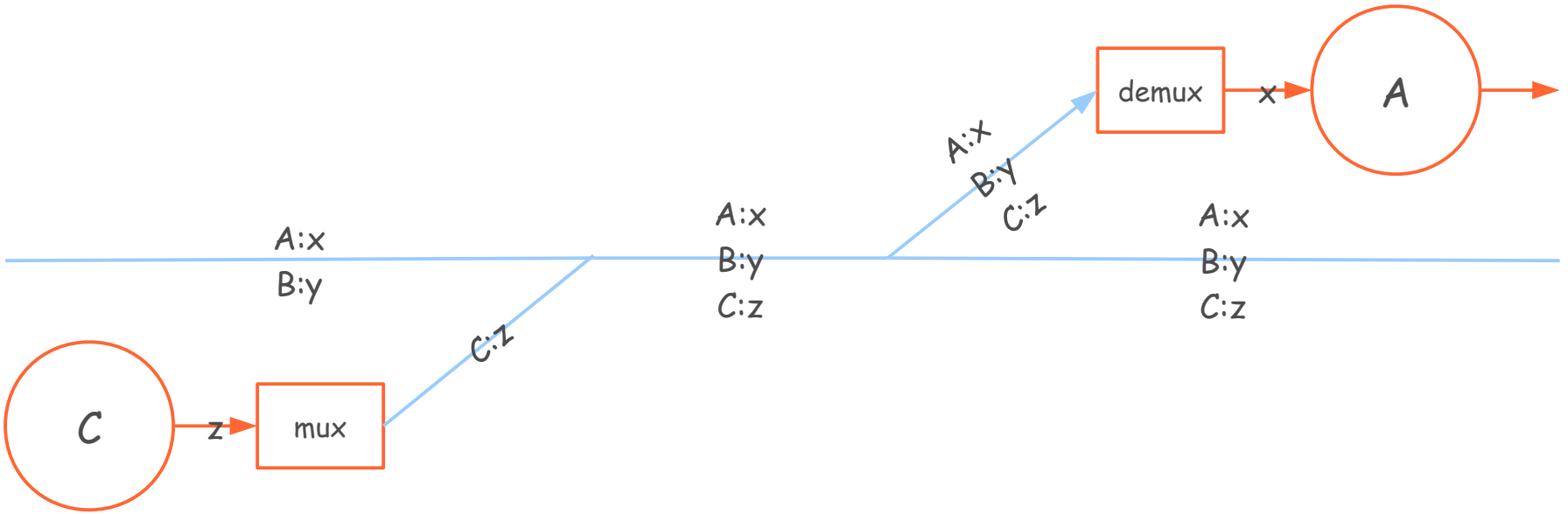
# Multiplexing Basics

... or there could be an approach where the two stages could be implemented as sets of independent Macros. I'll use the term decentralized for this approach.



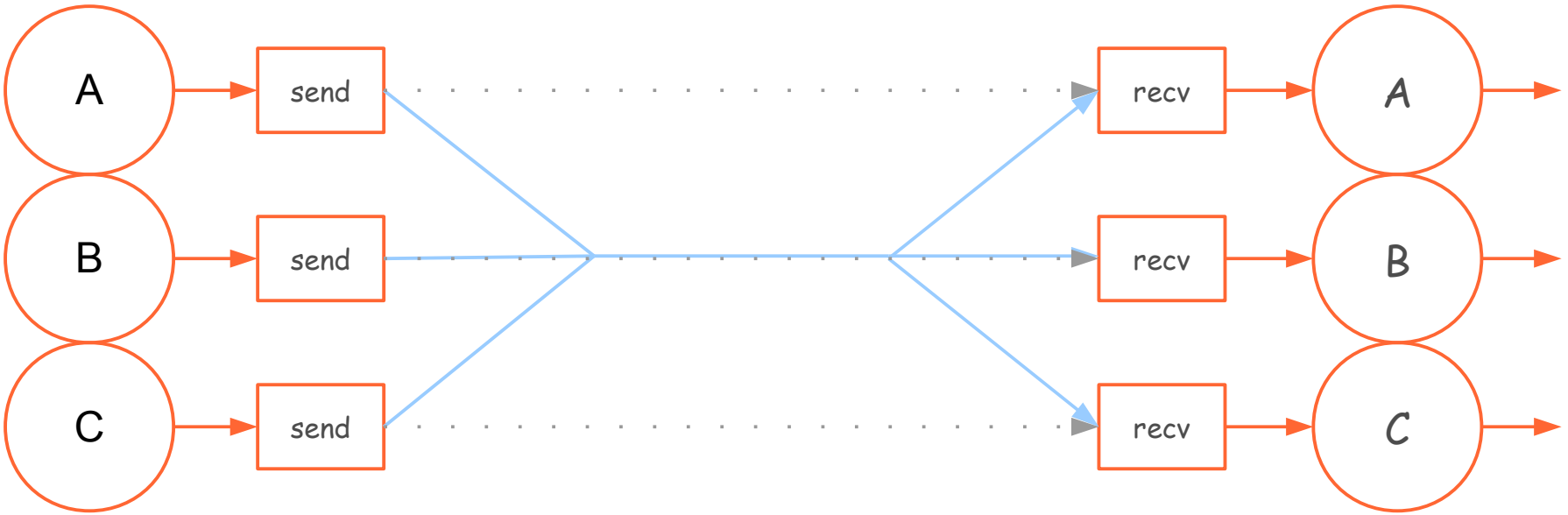
# Multiplexing Basics

Such a decentralized approach allows the demultiplexers to be used precisely where the unpacked value is needed. The demultiplexing can happen at different Macro layers and multiple times.



# Multiplexing Basics

A decentralized approach basically allows you to implement your own send and receive system.



# Possible Implementations...

There are several possibilities to implement multiplexing in Reaktor. Let's look at some of them...

# Trigger Multiplexing

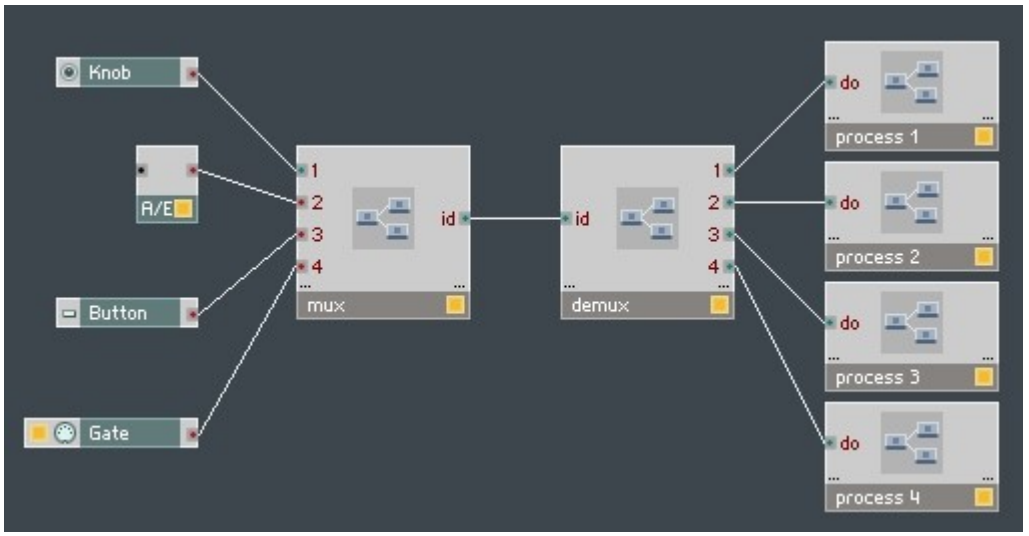
Let's start with a very simple example. Let's say there are four event sources and four different processes initiated by these sources, one for each source. Also, let's assume that the values of the events are of no importance for the processes.

# Trigger Multiplexing

This would be the direct approach without multiplexing...

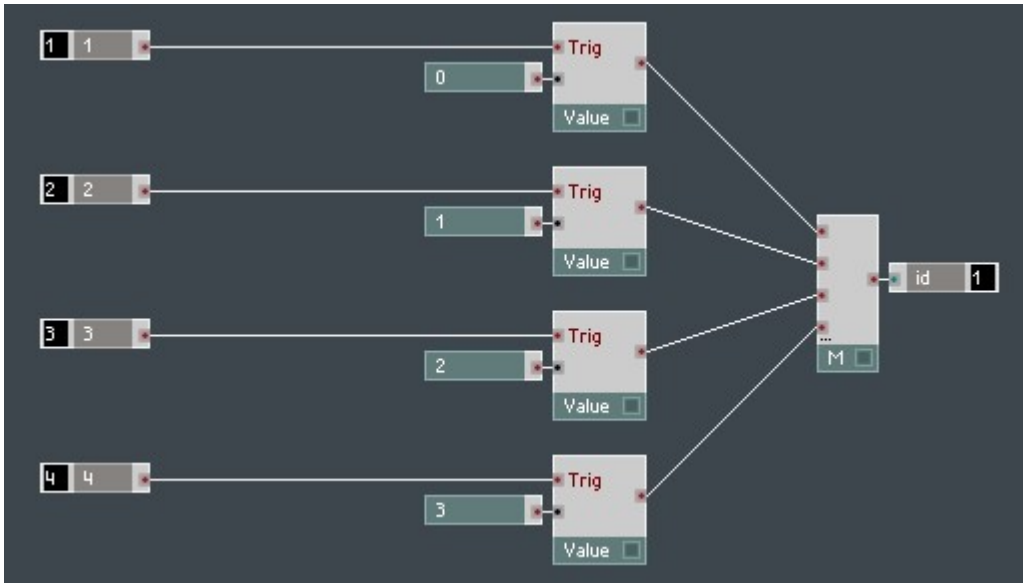
# Trigger Multiplexing

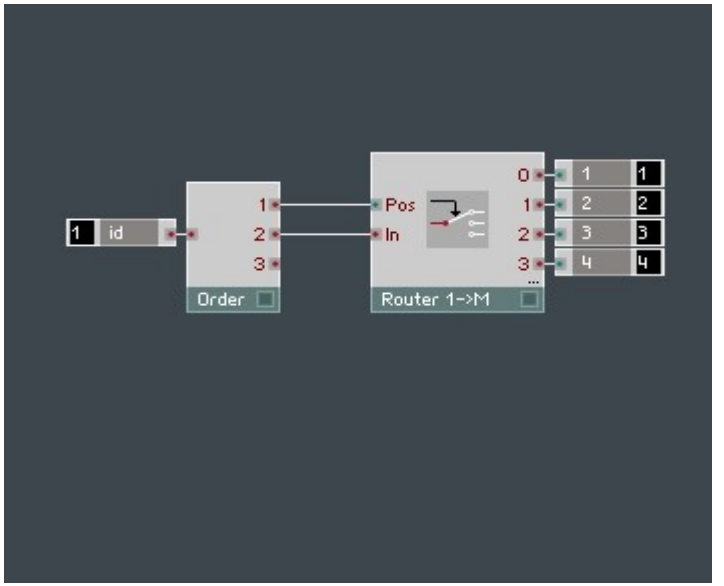
... here is the approach with multiplexing over a single wire.



# Trigger Multiplexing

How would the mux and demux Macros look like? Probably pretty simple, right? How about this...?



# Trigger Multiplexing

Well, there is a problem we have not considered so far: how does the multiplexing Structure behave during initialization?

Here is how it should behave:

# Trigger Multiplexing

Well, there is a problem we have not considered so far: how does the multiplexing Structure behave during initialization?

Here is how it should behave:

1. Any event arriving a the multiplexer needs to be available at the demultiplexer as well. Of course this also has to apply to events caused by initialization.

# Trigger Multiplexing

Well, there is a problem we have not considered so far: how does the multiplexing Structure behave during initialization?

Here is how it should behave:

1. Any event arriving a the multiplexer needs to be available at the demultiplexer as well. Of course this also has to apply to events caused by initialization.

2. If one of the ports is not connected, it must never cause an event at the corresponding port at the demultiplexer. Of course this also applies during initialization.

# Initialization

Working around the initialization algorithm of Reaktor is one of the more annoying problems you'll have to deal with when using multiplexing.

# Initialization

Working around the initialization algorithm of Reaktor is one of the more annoying problems you'll have to deal with when using multiplexing.

Any event module that merges two or more event streams (Add, Subtract, Merge, etc.) produces only one event at its output during the initialization stage. For our Structure this means that only the initialization event of the Gate module will pass! That's not what we want...

# Initialization

Working around the initialization algorithm of Reaktor is one of the more annoying problems you'll have to deal with when using multiplexing.

Any event module that merges two or more event streams (Add, Subtract, Merge, etc.) produces only one event at its output during the initialization stage. For our Structure this means that only the initialization event of the Gate module will pass! That's not what we want...

Imagine a situation where all GUI elements are multiplexed. Once you've started the Reaktor application the initialization algorithm will allow only one GUI event to reach its destinations when initializing!

# The Order Module

Fortunately, there is a work-around for this initialization behaviour: The 2nd or 3rd output of the Order module fire their events after the actual initialization stage. Events coming from these ports won't be dropped by merged wires.

# The Order Module

Fortunately, there is a work-around for this initialization behaviour: The 2nd or 3rd output of the Order module fire their events after the actual initialization stage. Events coming from these ports won't be dropped by merged wires.

Also, another very desirable behaviour of the Order module is that it doesn't send any initialization event on the 2nd or 3rd output if there is nothing connected to it's input.

# The Order Module

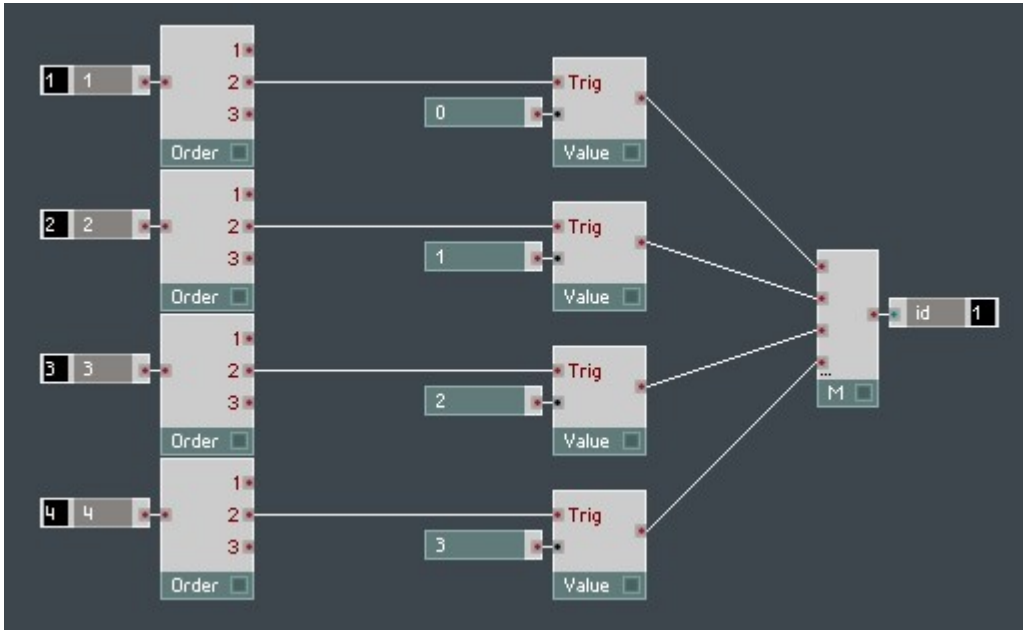
Fortunately, there is a work-around for this initialization behaviour: The 2nd or 3rd output of the Order module fire their events after the actual initialization stage. Events coming from these ports won't be dropped by merged wires.

Also, another very desired behaviour of the Order module is that it doesn't send any initialization event on the 2nd or 3rd output if there is nothing connected to it's input.

Without the Order module, multiplexing (and some other more advanced event processing) would be hard or impossible to realize in Reaktor.

# Trigger Multiplexing (Correct)

Here is the corrected multiplexer...



# Bit Multiplexing

In Reaktor the following idea may seem like an obscure technique but the principle is very common in other programming languages, especially when working at the very low level, close to the actual hardware.

# Bit Multiplexing

In Reaktor the following idea may seem like an obscure technique but the principle is very common in other programming languages, especially when working at the very low level, close to the actual hardware.

Since you can access the single bits of an integer using the "Bit AND", "Bit OR", "Bit >>" and "Bit <<" modules, it is easy to encode more than one information in a single integer number. In Reaktor Core an integer number is currently 32 bits long. That gives you plenty of combinations to consider sub ranges in the binary integer representation to encode different types of information. Take a look at some examples to get used to this idea...

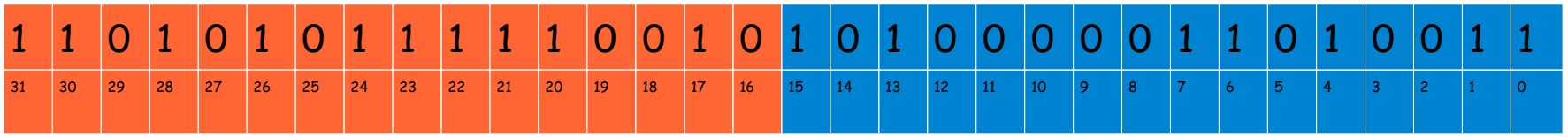
# Bit Multiplexing

32 times 1 or 0, e.g. "on" or "off"...

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

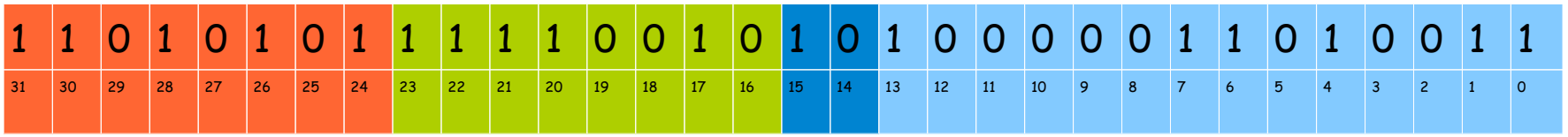
# Bit Multiplexing

Twice a range from 0 to 65535...



# Bit Multiplexing

Twice a range from 0 to 255, a range from 0 to 3 and a finally a range from 0 to 16383...



# How to?

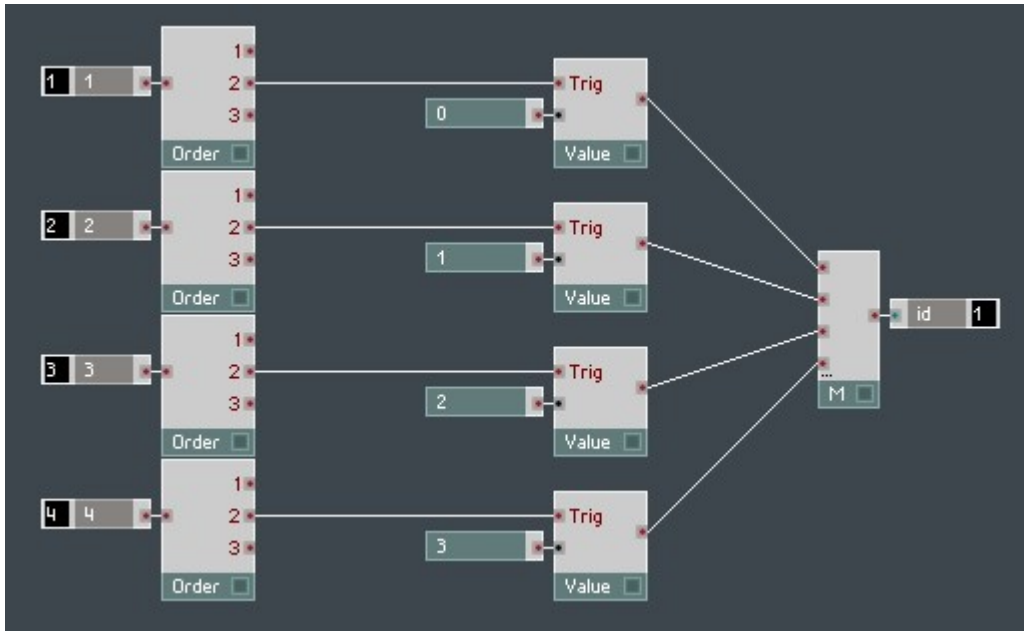
To check your understanding of this idea you should try to figure out the right usage of "Bit AND", "Bit OR", "Bit >>" and "Bit <<" needed to build the multiplexers and demultiplexers yourself... it's not hard, but a good exercise in working with integers and bits.

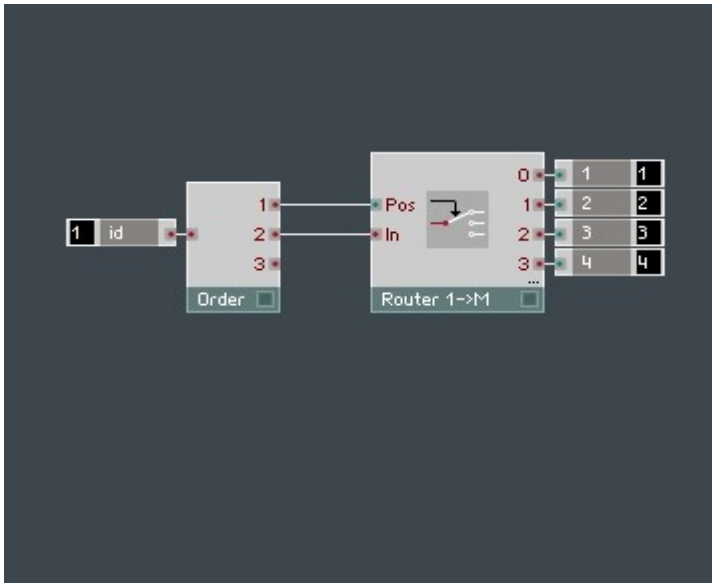
# Event Multiplexing

So far we haven't actually achieved the goal we've started with - allowing all possible events to travel the same channel and being able to identify the type (or source) of the event. The Trigger Multiplexer didn't allow us to send actual values and the Bit Multiplexing had the generally undesireable effect of reducing the value range for each multiplexed signal source with a growing number of signal sources.

# Event Multiplexing

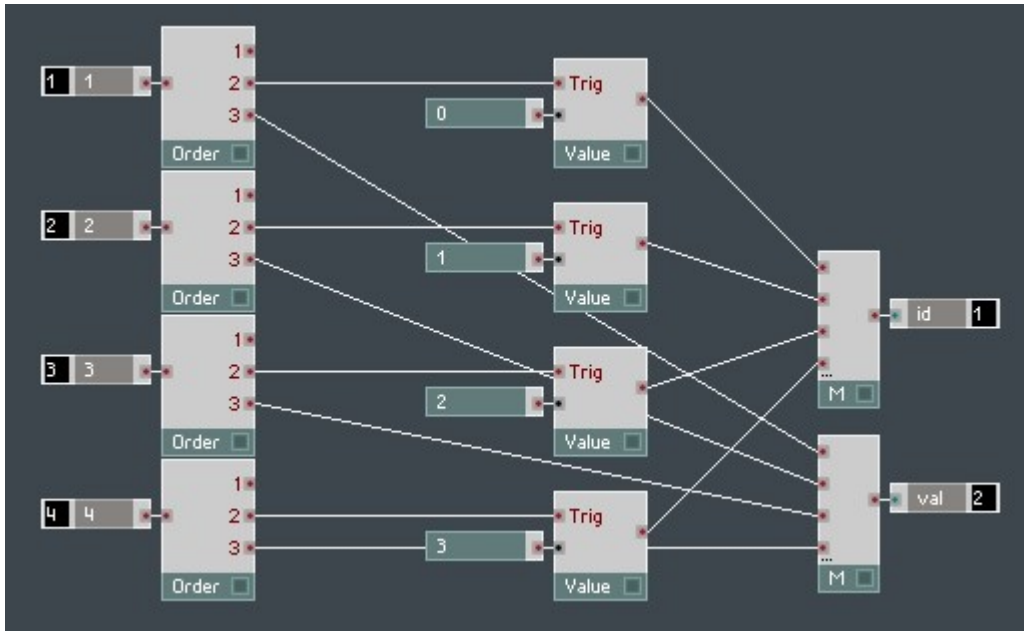
Actually half of what we need is already done, though. We can reuse most of the multiplexer and demultiplexer Structures from our Trigger Multiplexing approach. Remember?

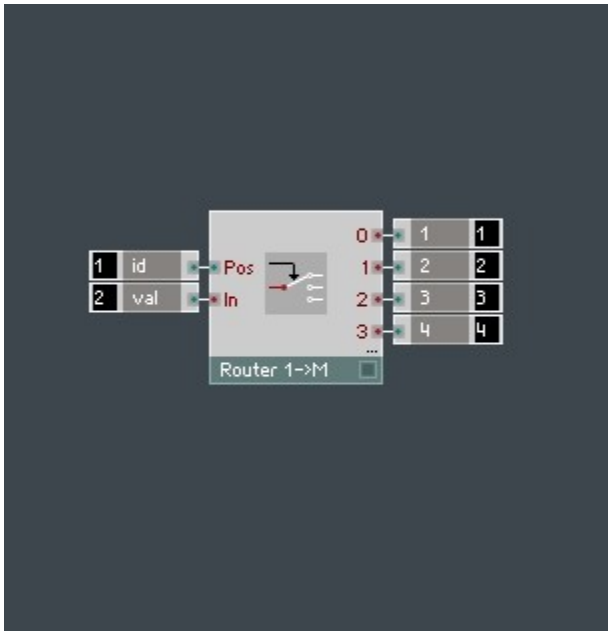


# Event Multiplexing

The only thing we need to change is add a second wire to transmit the actual value after we transmitted the id and use this as the value input for the Router:



# Improvements?

While this is a perfectly valid approach there are two things that could be improved:

1. It would be nice to need only one wire instead of two.

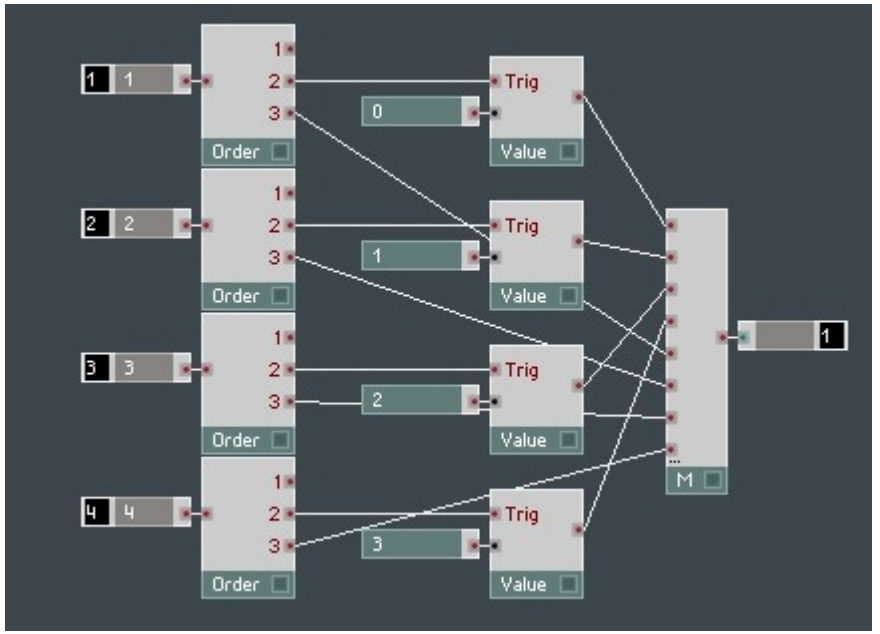
2. It would be nice to have a decentralized demultiplexer.

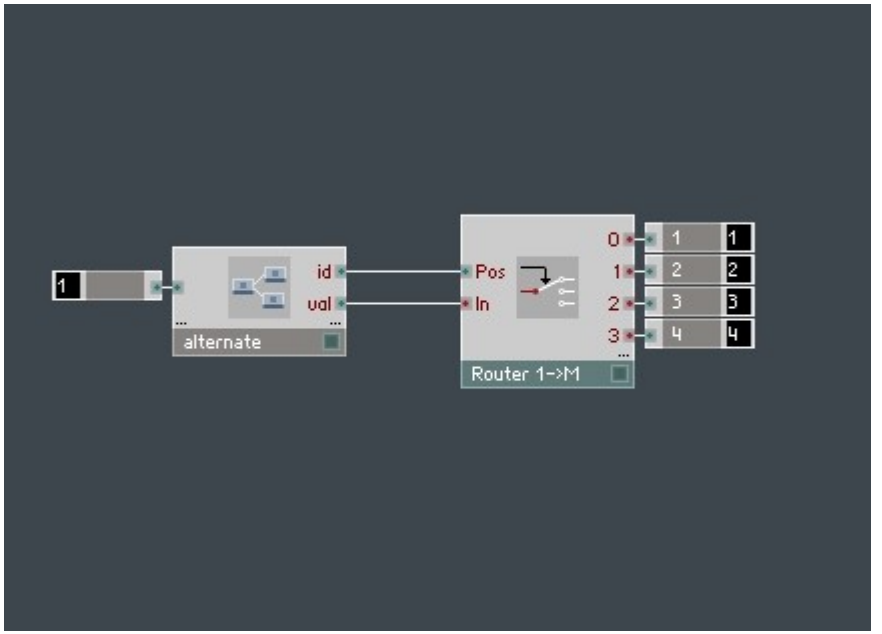
A decentralized demultiplexer is a lot more useful than a decentralized multiplexer, since one event source has usually many event sinks.

Also a decentralized multiplexer makes it difficult to fulfill the requirement of unique IDs (multiplexer chaining, anyone ?)!

# Single Wire Event Multiplexing

Multiplexing on a single wire (with the requirement that **all** 32 bits of a float value can be transmitted) can be achieved by sending both the ID and then the actual value over the same wire. The demultiplexer only needs to alternate between ID and value with every event it receives:



# Synchronizing Issues?

But isn't it possible that an id event is interpreted as a val event and that the system gets out of sync? Well, not if all events arriving at the multiplexer come from the same thread.

If all events come from the same thread this method is **just as safe** as using two wires.

If the events come from separate threads this method causes **significantly more trouble** than separate wires.

But since thread safeness is **always required** when using multiplexing anyway, I decided on the "single wire + forced thread safeness" approach for the framework. Right now would be a good point to read the presentation about thread safeness...

# Event Block Multiplexing

Let's modify the event multiplexing approach a little by allowing an arbitrary number of value events following the id event.

# Event Block Multiplexing

Let's modify the event multiplexing approach a little by allowing an arbitrary number of value events following the id event.

So instead of an event stream like

id, val, id, val, id, val, id, val, ...

we'll allow a stream like

id, n, val 1, val 2, ..., val n, id, m, val 1, val 2, ..., val m, id, ...

# Event Block Multiplexing

Let's modify the event multiplexing approach a little by allowing an arbitrary number of value events following the id event.

So instead of an event stream like

id, val, id, val, id, val, id, val, ...

we'll allow a stream like

id, n, val 1, val 2, ..., val n, id, m, val 1, val 2, ..., val m, id, ...

This can be achieved by a simple counter for each demultiplexer.

# Event Block Multiplexing

It is often necessary to implement a more complex "messaging" system on top this multiplexing scheme.

# Event Block Multiplexing

It is often necessary to implement a more complex "messaging" system on top this multiplexing scheme.

A standard application for this multiplexing scheme would be to read a range of values from an Event Table. So it would make sense to extend the event stream to something like this:

id, n, lo, val 1, val 2, val 3, ..., val n-1, id, ...

where lo and n are used to define the range of indices of values read from the Event Table.

# Event Block Multiplexing

It is often necessary to implement a more complex "messaging" system on top this multiplexing scheme.

A standard application for this multiplexing scheme would be to read a range of values from an Event Table. So it would make sense to extend the event stream to something like this:

id, n, lo, val 1, val 2, val 3, ..., val n-1, id, ...

where lo and n are used to define the range of indices of values read from the Event Table.

Since a demultiplexer needs to count the events anyway the counter can be used to create the necessary indices based on lo to write into a cell of a Core array. Read more about that in the tutorial about the { b } bus...

# Summary

Multiplexing allows for faster building and can result in more readable Structures. It is used to work around limitations like 40 inports and outports of Macros and Core Cells.

There are different kinds of multiplexing mentioned in this tutorial such as trigger multiplexing, bit multiplexing, event – and event block multiplexing.

There are more multiplexing techniques that were not discussed, like shared-memory solutions utilizing multi-client Event Tables in Primary and OBC connections in Core...