Everything

that can go wrong in

a message passing system

Just some things

that can go wrong in

a message passing system

Just some things

that can go wrong in

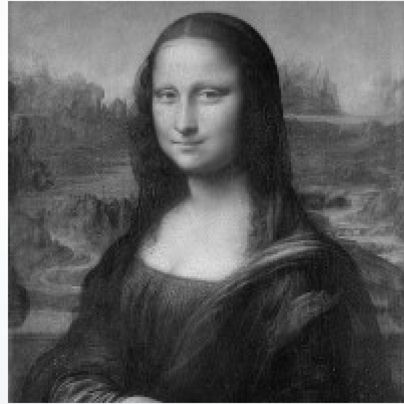
a message passing system

Message passing is nice

Pipeline

Input

Operation A



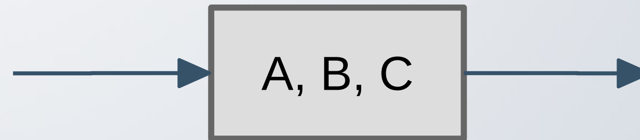
1ms

Operation B

3ms

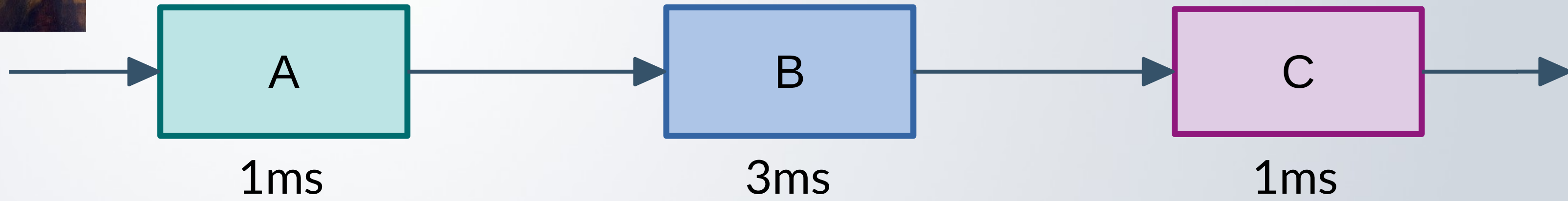
Operation C

1ms

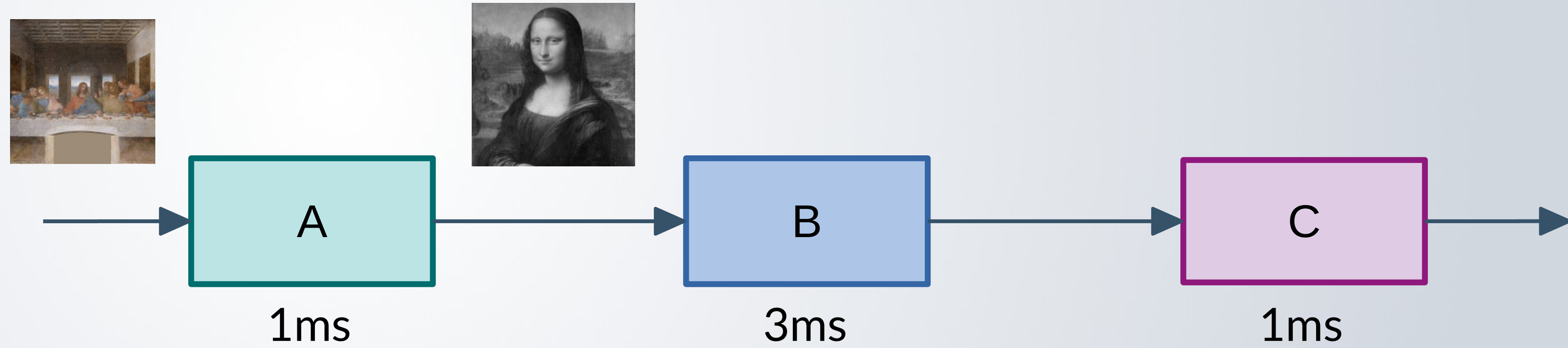


5ms

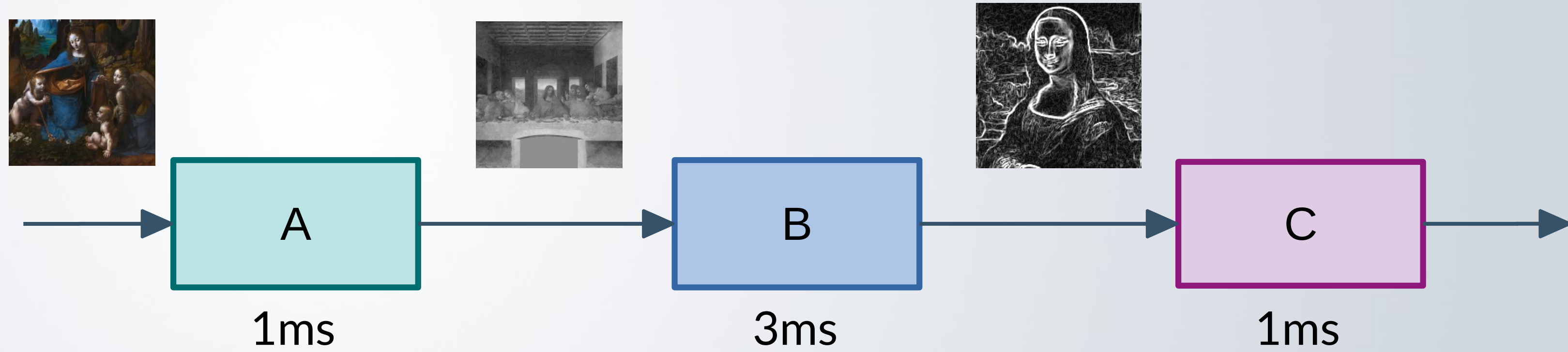
Pipeline



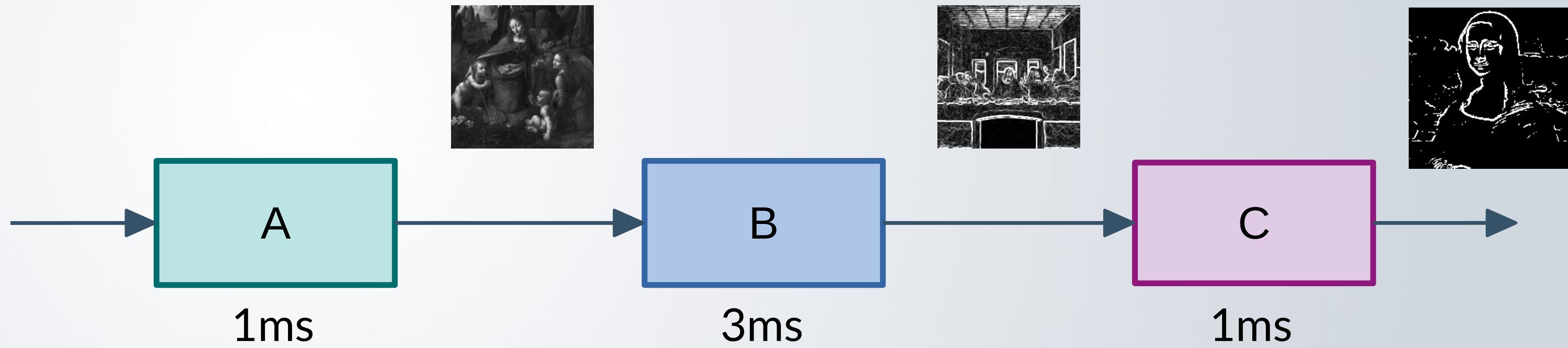
Pipeline



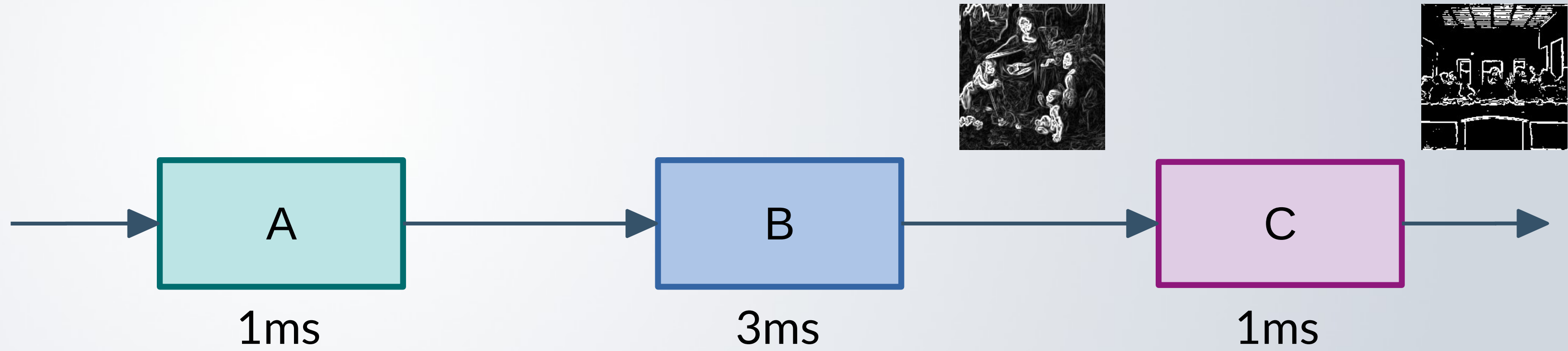
Pipeline



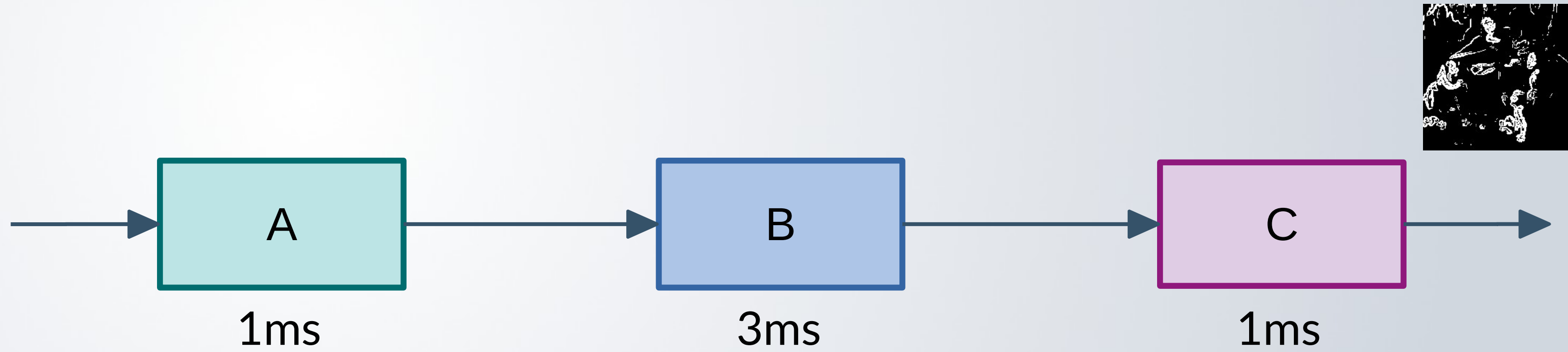
Pipeline



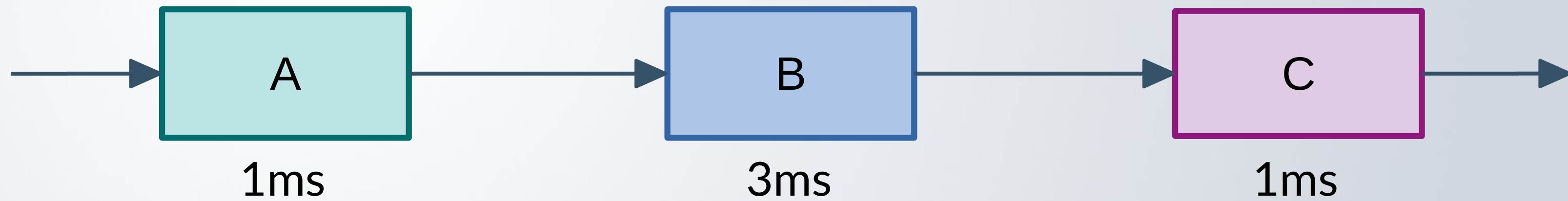
Pipeline



Pipeline

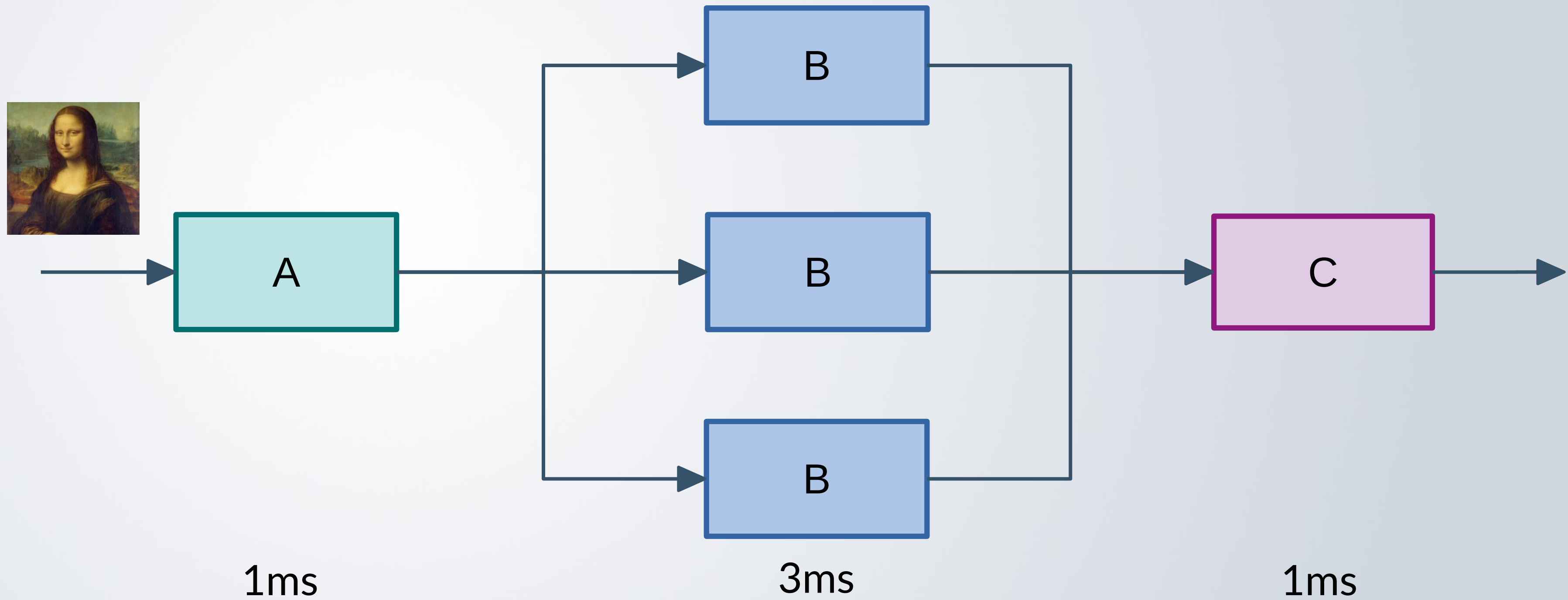


Pipeline

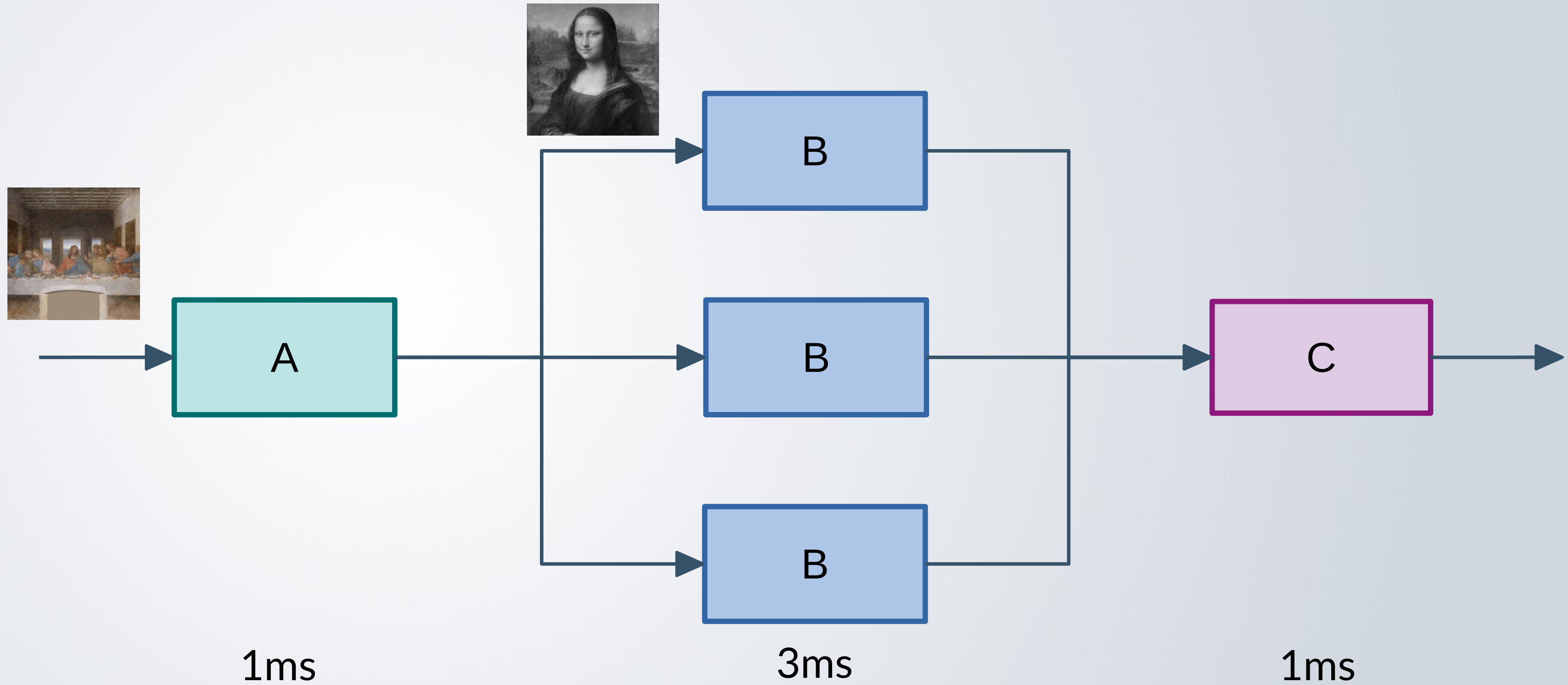


Total latency: 5ms
Total throughput: 1 img / 3ms

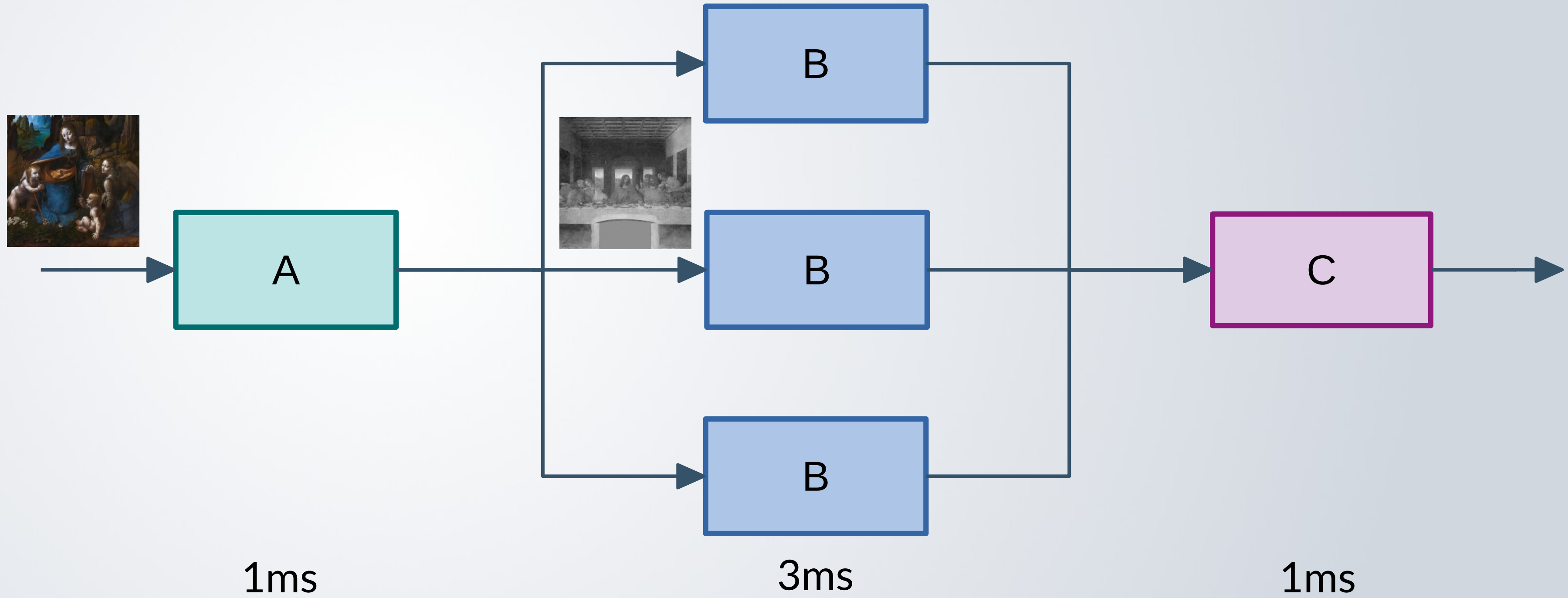
Pipeline with fan-out/fan-in



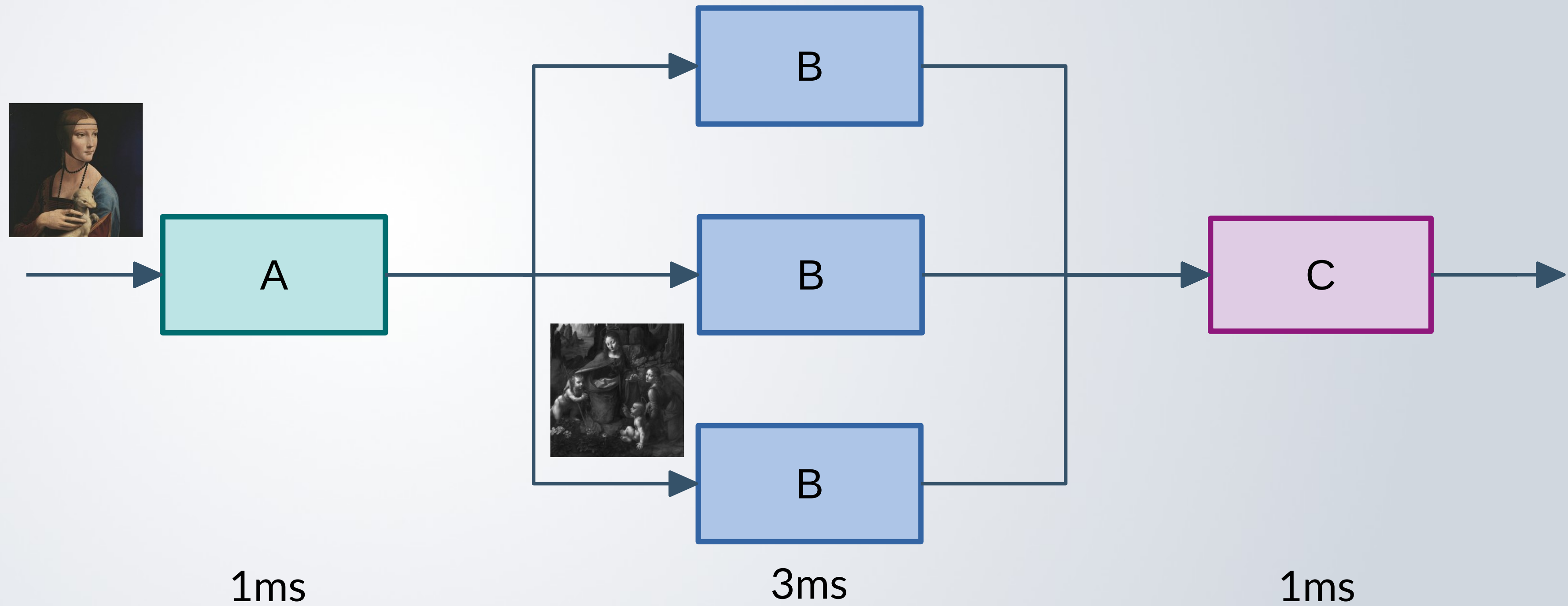
Pipeline with fan-out/fan-in



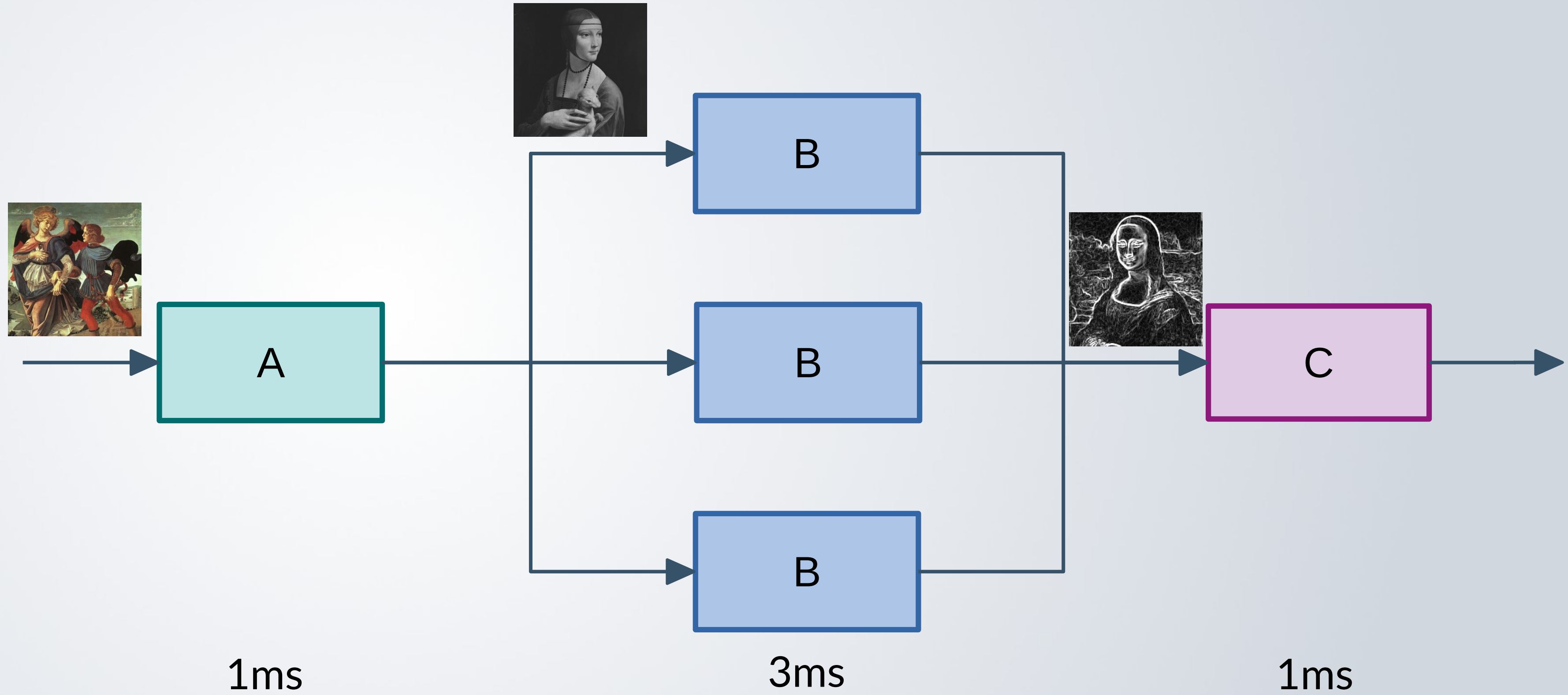
Pipeline with fan-out/fan-in



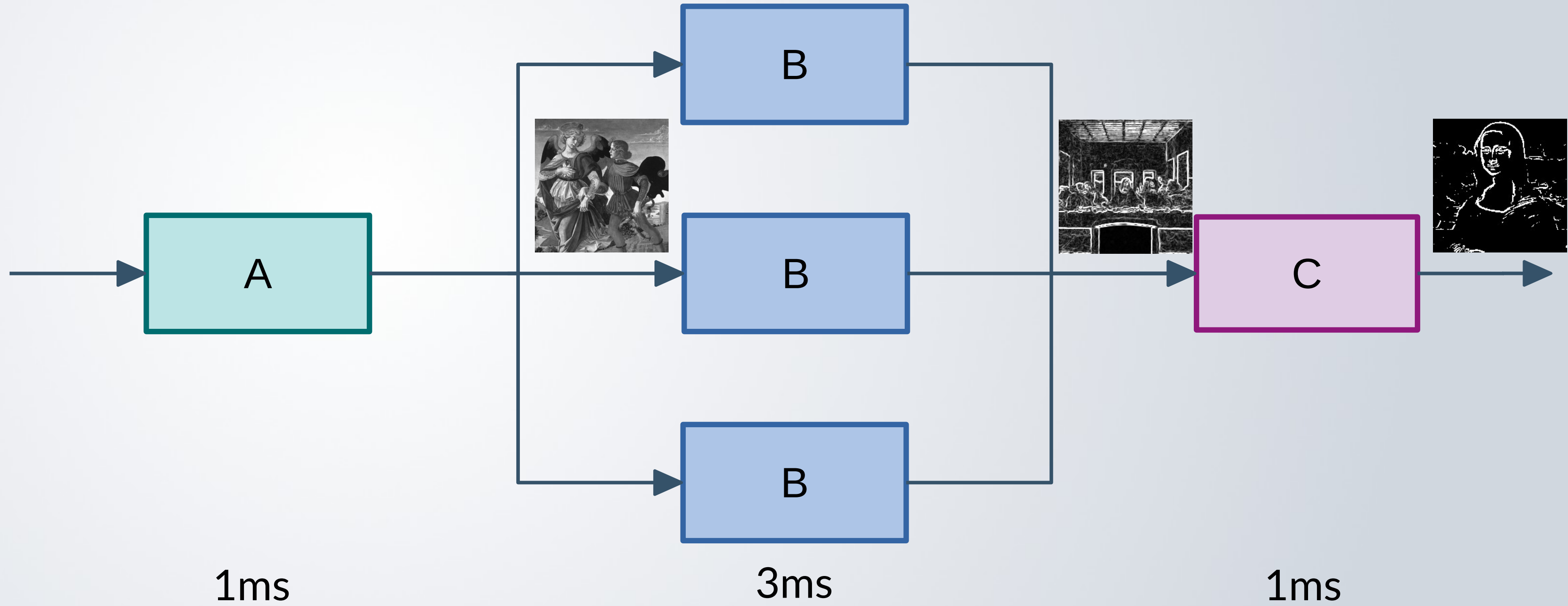
Pipeline with fan-out/fan-in



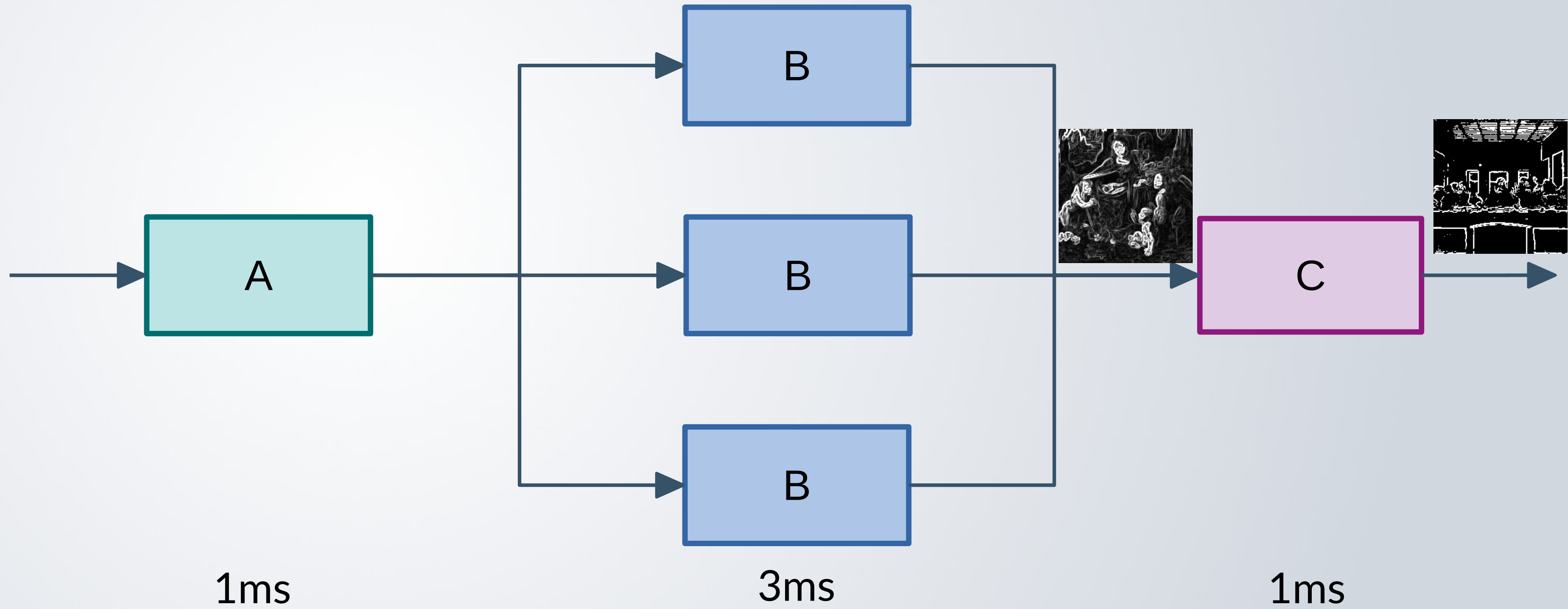
Pipeline with fan-out/fan-in



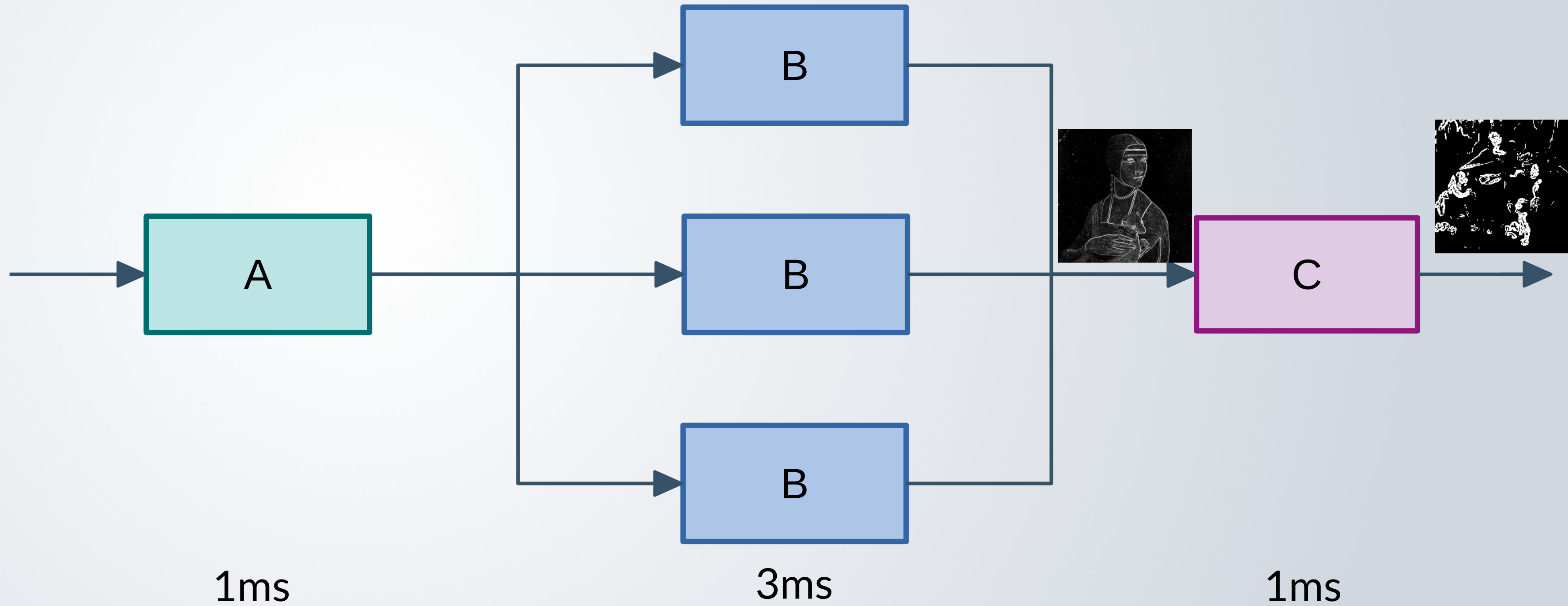
Pipeline with fan-out/fan-in



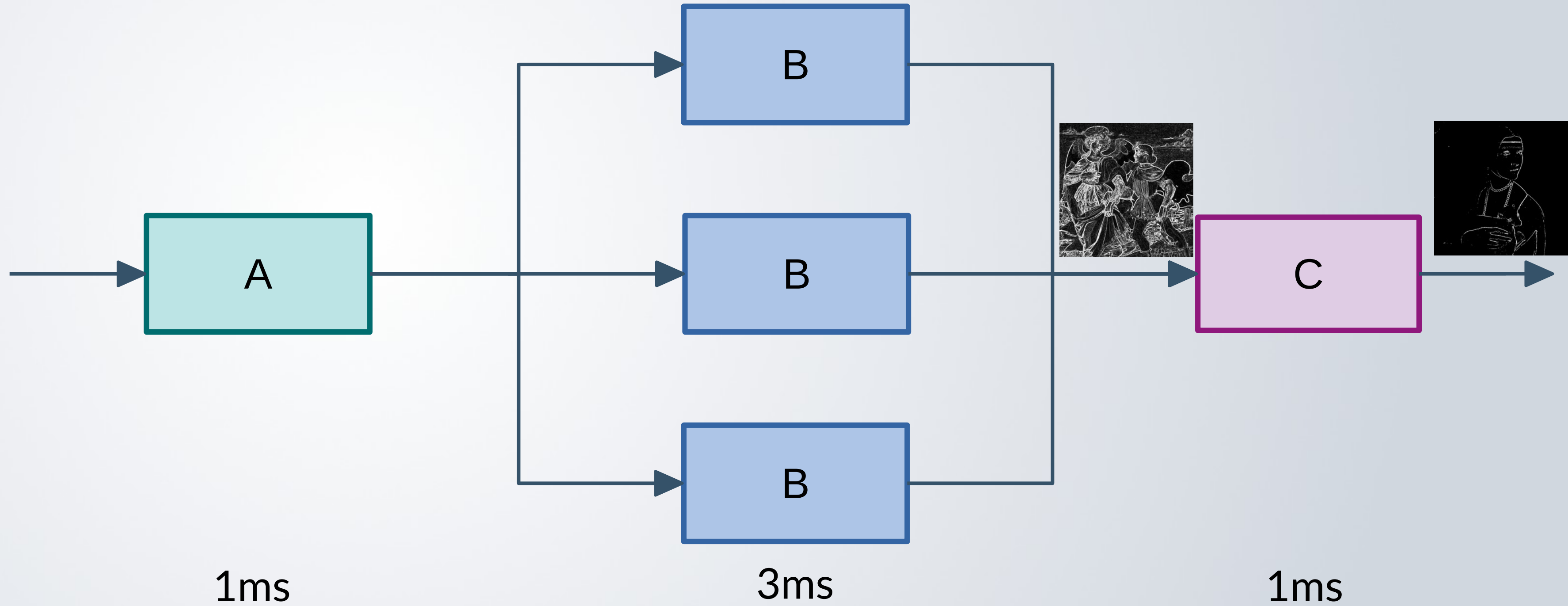
Pipeline with fan-out/fan-in



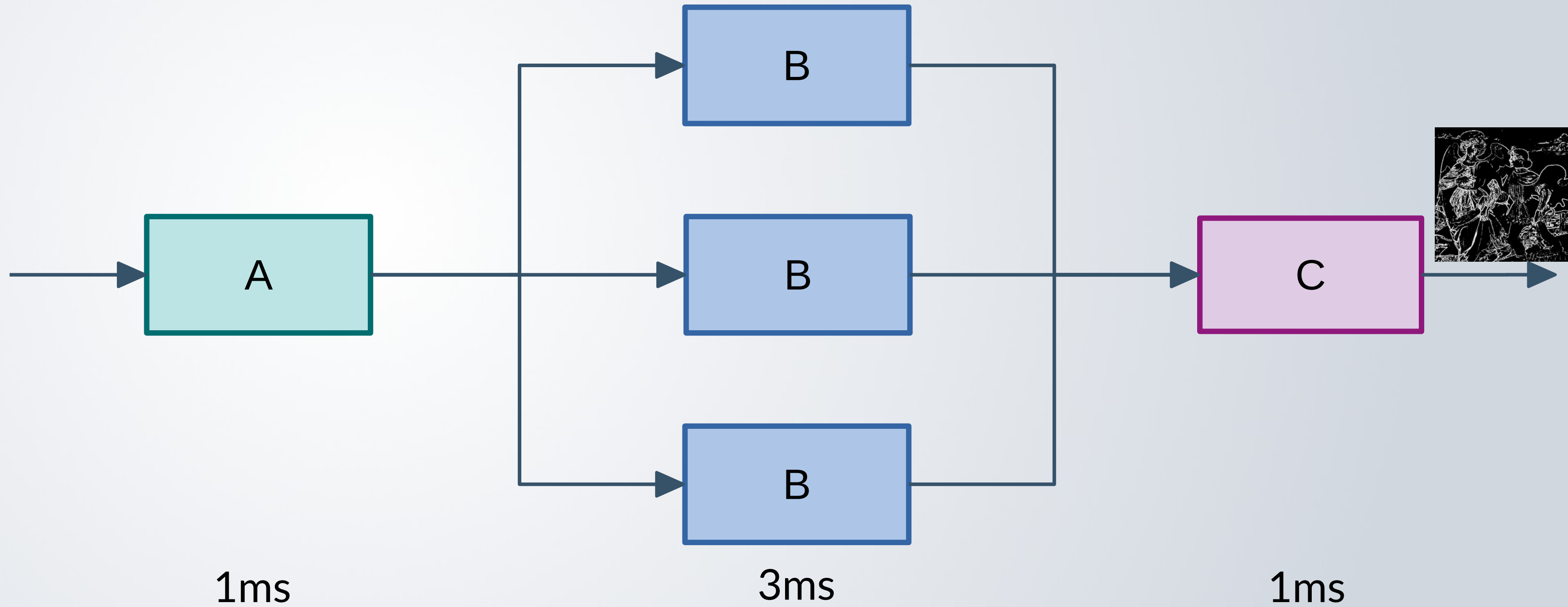
Pipeline with fan-out/fan-in



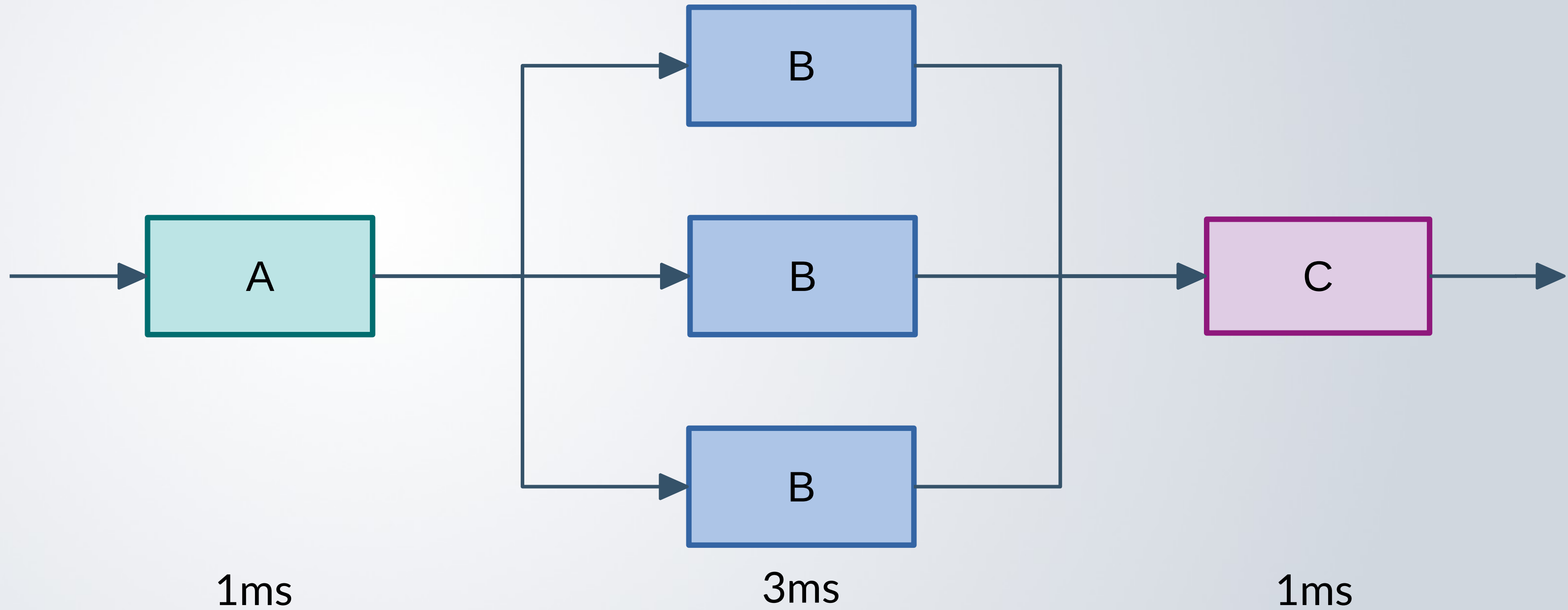
Pipeline with fan-out/fan-in



Pipeline with fan-out/fan-in



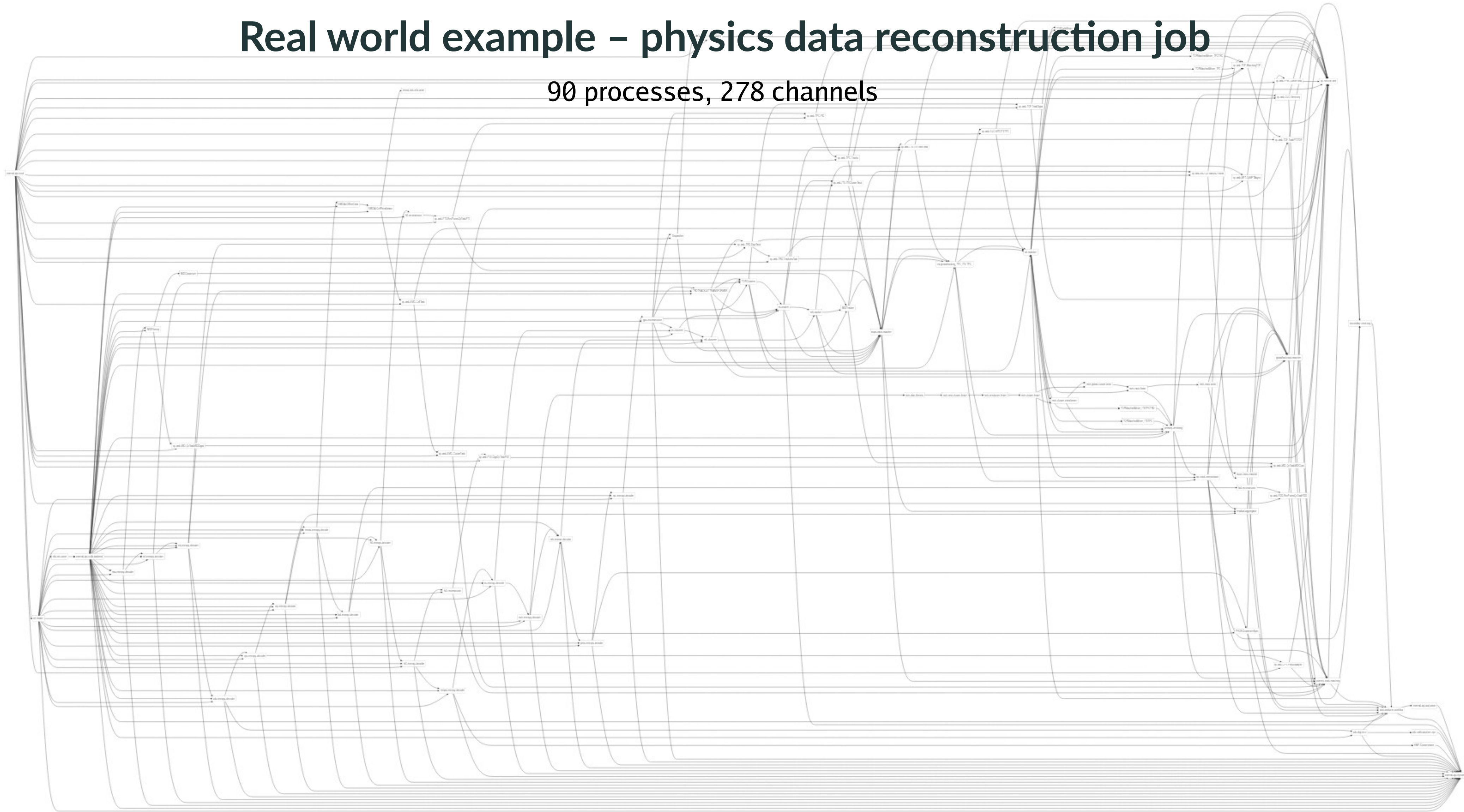
Pipeline with fan-out/fan-in



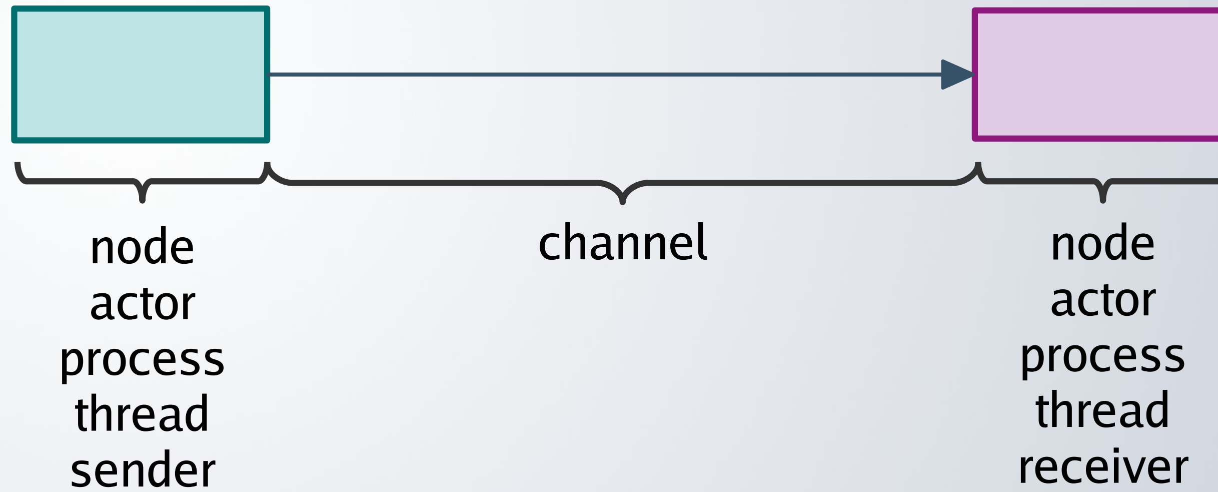
Total latency: 5ms
Total throughput: 1 img / 1ms

Real world example – physics data reconstruction job

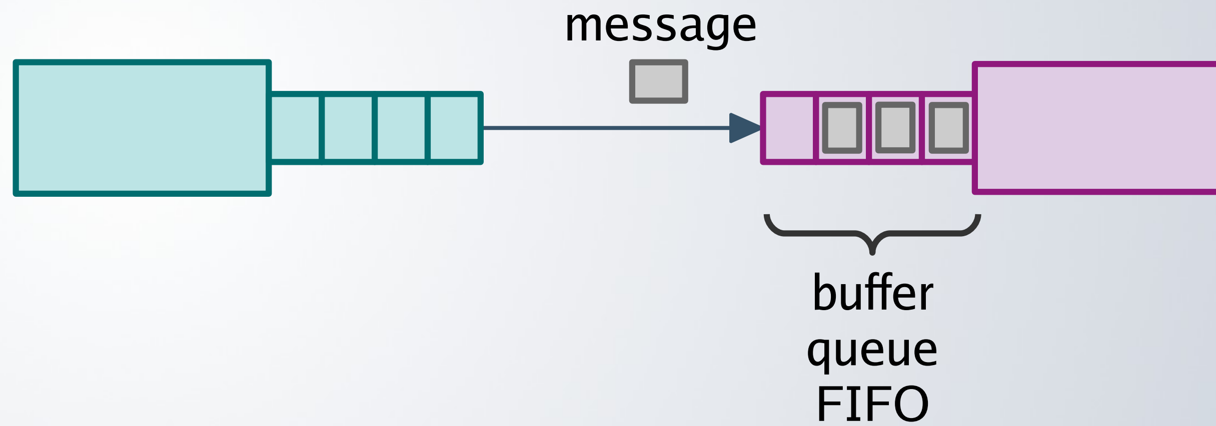
90 processes, 278 channels



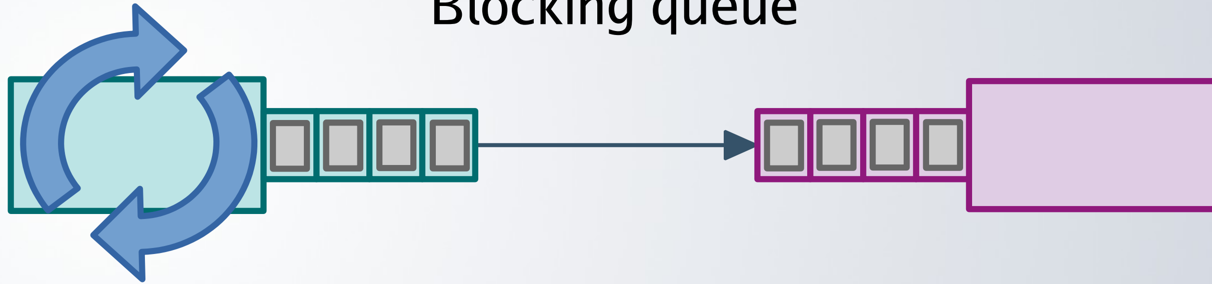
Queues



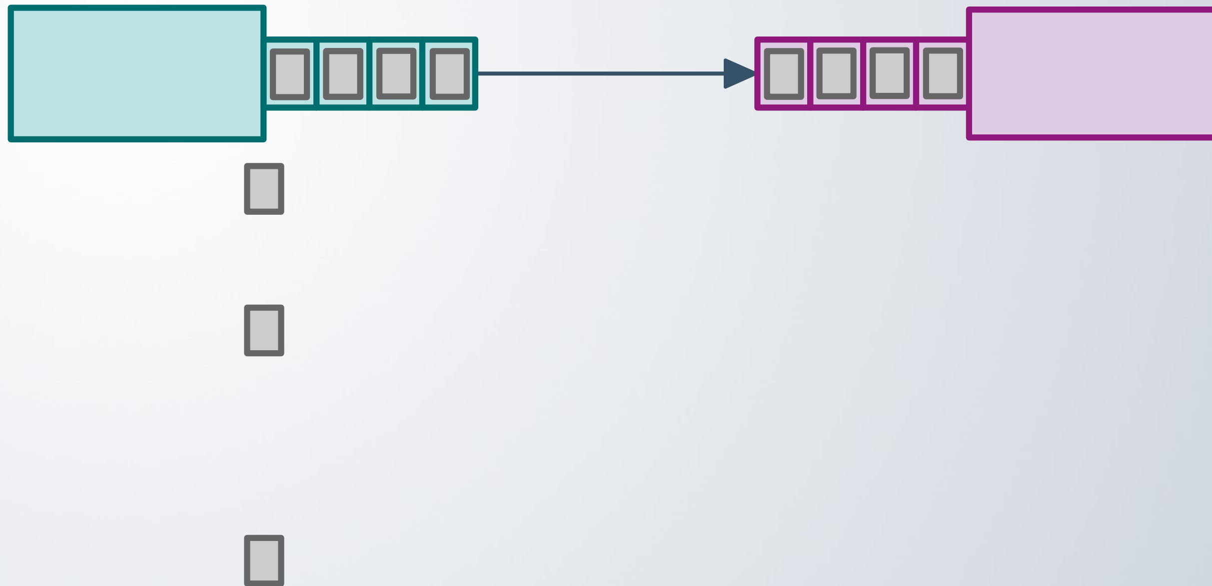
Queues



Queues



Queues



Message passing is nice

Message passing is nice

Scalability

Single
Responsibility

Multithreading

Modularity

Multiprocessing

Extensibility

Scalability

Single
Responsibility

Multithreading

Message passing is nice
Let’s break it

Modularity

Multiprocessing

Extensibility

Test setup

ØMQ

Problem 1

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.PUB)
s.bind("tcp://*:7891")


print("PUB: Sending a message...")
s.send_pyobj("hello")
print("PUB: Message sent.")
```

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.SUB)
s.connect("tcp://localhost:7891")
s.subscribe("")

print("SUB: Ready to receive...")
msg = s.recv_pyobj()
print("SUB: Received message:", msg)
```

Problem 1

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.PUB)
s.bind("tcp://*:7891")


print("PUB: Sending a message...")
s.send_pyobj("hello")
print("PUB: Message sent.")
```

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.SUB)
s.connect("tcp://localhost:7891")
s.subscribe("")

print("SUB: Ready to receive...")
msg = s.recv_pyobj()
print("SUB: Received message:", msg)
```

```
$ python3 01a_sub.py & python3 01a_pub.py
[1] 1272653
PUB: Sending a message...
PUB: Message sent.
SUB: Ready to receive...
```

Problem 1 - synchronization

```python
import zmq
import time
ctx = zmq.Context()
s = ctx.socket(zmq.PUB)
s.bind("tcp://*:7891")
time.sleep(1)

print("PUB: Sending a message...")
s.send_pyobj("hello")
print("PUB: Message sent.")
```

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.SUB)
s.connect("tcp://localhost:7891")
s.subscribe("")

print("SUB: Ready to receive...")
msg = s.recv_pyobj()
print("SUB: Received message:", msg)
```

Problem 1 - synchronization

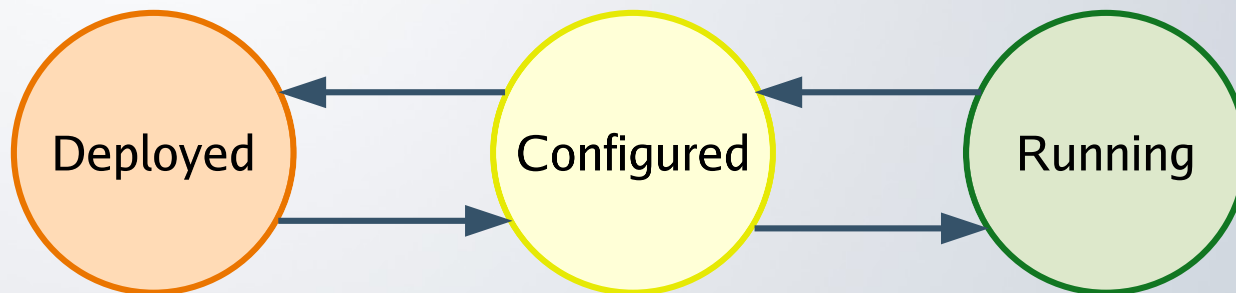
```python
import zmq
import time
ctx = zmq.Context()
s = ctx.socket(zmq.PUB)
s.bind("tcp://*:7891")
time.sleep(1)

print("PUB: Sending a message...")
s.send_pyobj("hello")
print("PUB: Message sent.")
```

```python
import zmq

ctx = zmq.Context()
s = ctx.socket(zmq.SUB)
s.connect("tcp://localhost:7891")
s.subscribe("")

print("SUB: Ready to receive...")
msg = s.recv_pyobj()
print("SUB: Received message:", msg)
```

```
$ python3 01b_sub.py & python3 01b_pub.py
[1] 1273119
SUB: Ready to receive...
PUB: Sending a message...
PUB: Message sent.
SUB: Received message: hello
[1]+  Done                    python3 01b_sub.py
```

Problem 1 - summary

- Do not rely on synchronization with sleep(), this is only for illustration
- Make sure all connections are working before passing data around
- Consider using a state machine
- ...or use a library which retains messages until they are processed



Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(100)

s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(100)

s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(100)

s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(100)

s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Queue capacity: 350, 950, 700, 850, 600

Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)

s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)

s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)

s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)

s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Queue capacity: 282, 183, 182, 183, 183

Problem 2 – limiting memory usage

```
tcp_rmem (since Linux 2.4)
       This is a vector of 3 integers: [min, default, max].
       These parameters are used by TCP to regulate receive
       buffer sizes.  TCP dynamically adjusts the size of the
       receive buffer from the defaults listed below, in the
       range of these values, depending on memory available in
       the system.
```

https://man7.org/linux/man-pages/man7/tcp.7.html

Problem 2 – limiting memory usage

```
$ cat /proc/sys/net/ipv4/tcp_wmem
4096   16384  4194304
$ cat /proc/sys/net/ipv4/tcp_rmem
4096   131072 6291456
```

The maximum sizes for socket buffers declared via the **SO_SNDBUF** and **SO_RCVBUF** mechanisms are limited by the values in the */proc/sys/net/core/rmem_max* and */proc/sys/net/core/wmem_max* files. Note that TCP actually allocates twice the size of the buffer requested in the setsockopt(2) call, and so a succeeding getsockopt(2) call will not return the same size of buffer as requested in the setsockopt(2) call. TCP uses the extra space for administrative purposes and internal kernel structures, and the */proc* file values reflect the larger sizes compared to the actual TCP windows. On individual connections, the socket buffer size must be set prior to the listen(2) or connect(2) calls in order to have it take effect. See socket(7) for more information.

https://man7.org/linux/man-pages/man7/tcp.7.html

Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)
s.setsockopt(zmq.SNDBUF, 100000)
s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)
s.setsockopt(zmq.RCVBUF, 100000)
s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Problem 2 – limiting memory usage

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)
s.setsockopt(zmq.SNDBUF, 100000)
s.bind("tcp://*:7891")
time.sleep(1)

print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

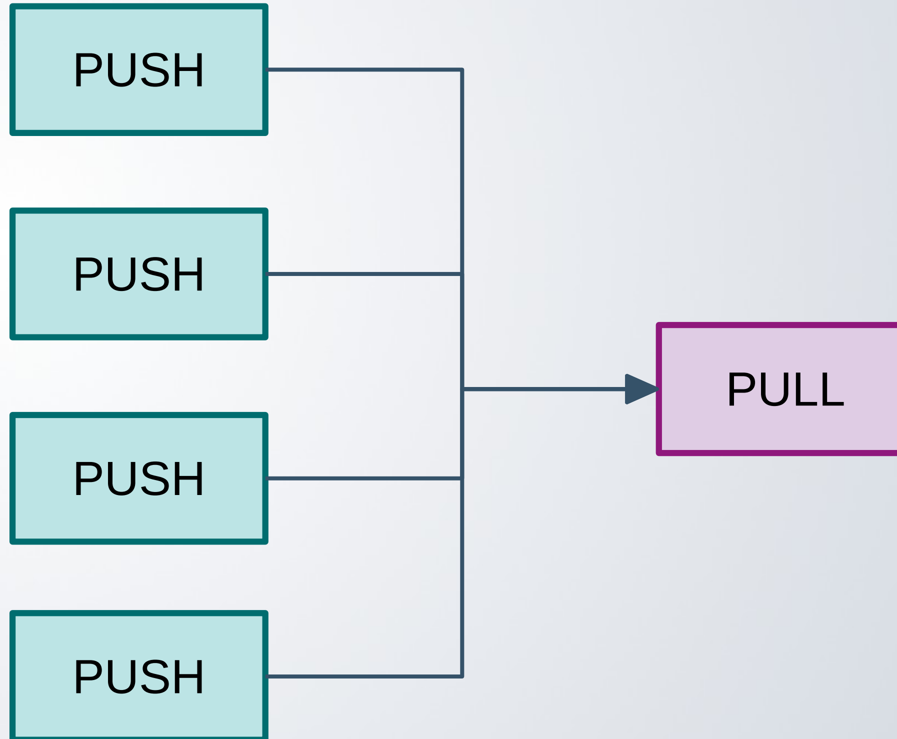
```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)
s.setsockopt(zmq.RCVBUF, 100000)
s.connect("tcp://localhost:7891")


print("PULL: Ready to receive, pausing")
signal.pause()
```

Queue capacity: 32, 31, 33, 31, 32

Problem 2 – limiting memory usage

Transport	Send High Water Mark [msgs]	Send Buffer [B]	Receive Buffer [B]	Receive High Water Mark [msgs]	Queue cap. [msgs]
TCP/localhost	100	default	default	100	350–950
TCP/localhost	1	default	default	1	182–282
TCP/localhost	1	100000	100000	1	31–32
TCP/localhost	1	4096	4096	1	4 - 5
TCP/localhost	100	4096	4096	1	100
TCP/localhost	1	4096	4096	100	104
TCP/network	1	4096	4096	1	5
IPC	1	65536		1	8 – 9

Problem 2 – limiting memory usage



Problem 2 – limiting memory usage

4x

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)
s.setsockopt(zmq.SNDBUF, 100000)
s.connect("tcp://localhost:7891")

time.sleep(1)
print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)
s.setsockopt(zmq.RCVBUF, 100000)
s.bind("tcp://*:7891")


print("PULL: Ready to receive, pausing the thread")
signal.pause()
```

Problem 2 – limiting memory usage

4x

```python
import zmq
import time

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.set_hwm(1)
s.setsockopt(zmq.SNDBUF, 100000)
s.connect("tcp://localhost:7891")

time.sleep(1)
print("PUSH: Sending messages...")
for i in range(1000):
  s.send(b'a' * 10000) # 10kB
  print(i+1)
```

```python
import zmq
import signal

ctx = zmq.Context()
s = ctx.socket(zmq.PULL)
s.set_hwm(1)
s.setsockopt(zmq.RCVBUF, 100000)
s.bind("tcp://*:7891")


print("PULL: Ready to receive, pausing
the thread")
signal.pause()
```

Queue capacity: 125 (32 + 31 + 31 + 31), 128 (33 + 32 + 32 + 31), 127 (32 + 32 + 32 + 31)

Problem 2 – limiting memory usage

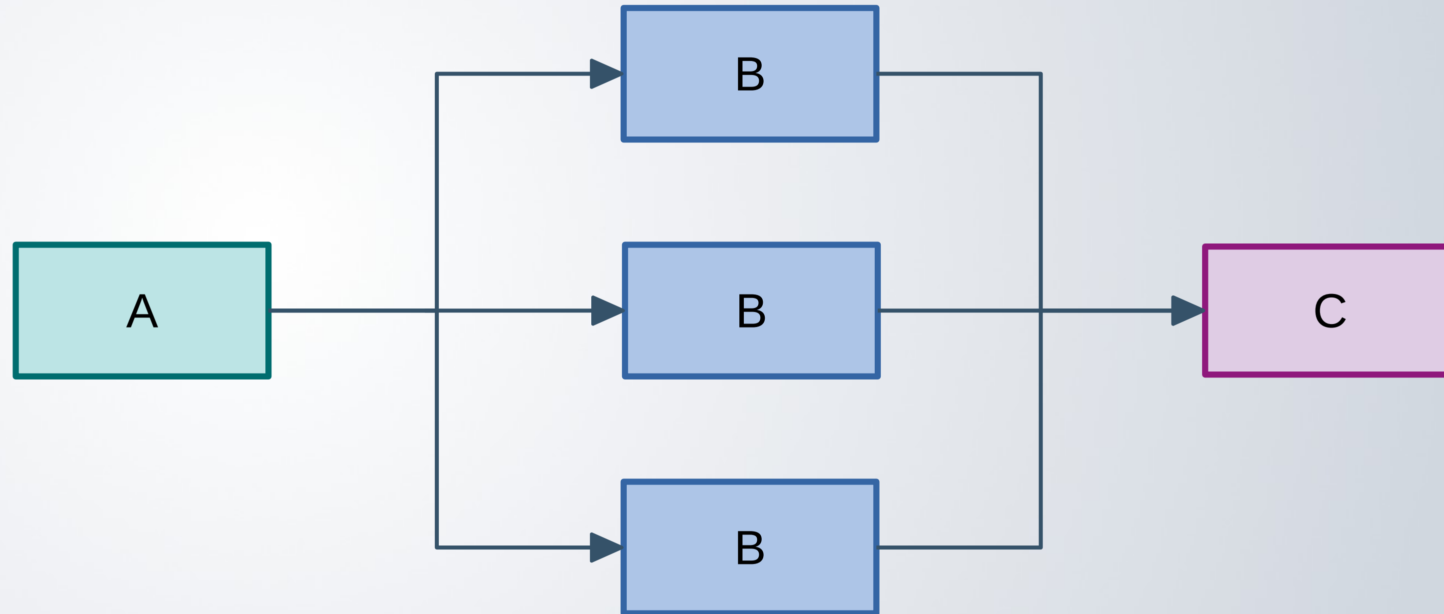
Problem 2 – limiting memory usage

CPU, memory and bandwidth spikes

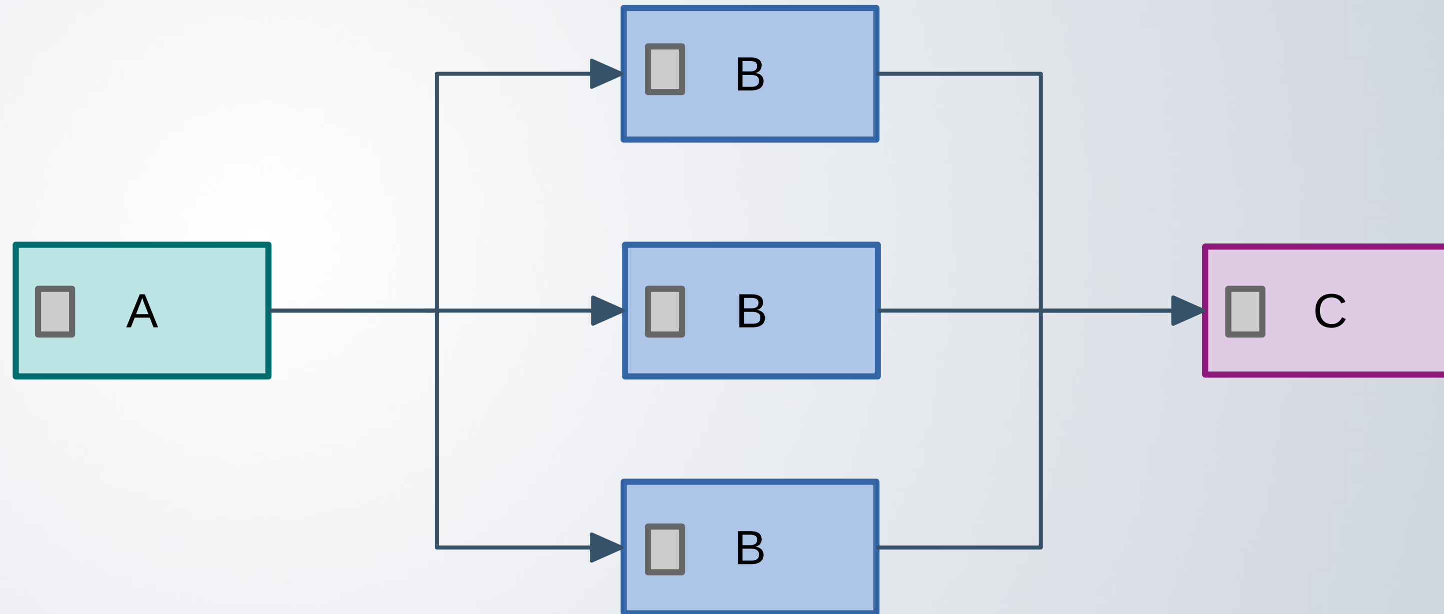
Problem 2 – limiting memory usage

Shifting publication time in phase evens out the receiver's load

Problem 2 – limiting memory usage



Problem 2 – limiting memory usage



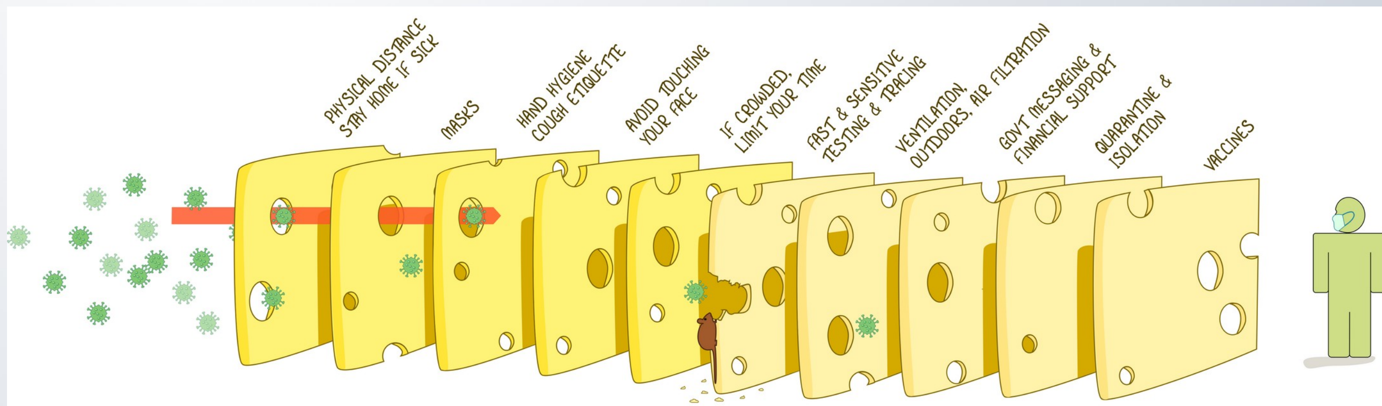
Do not introduce more messages in the topology than it is possible to handle

Problem 2 - summary

- Get familiar with your message passing library
  - Do the options work as you expect?
  - How are the data transferred?
  - Is the behaviour predictable?

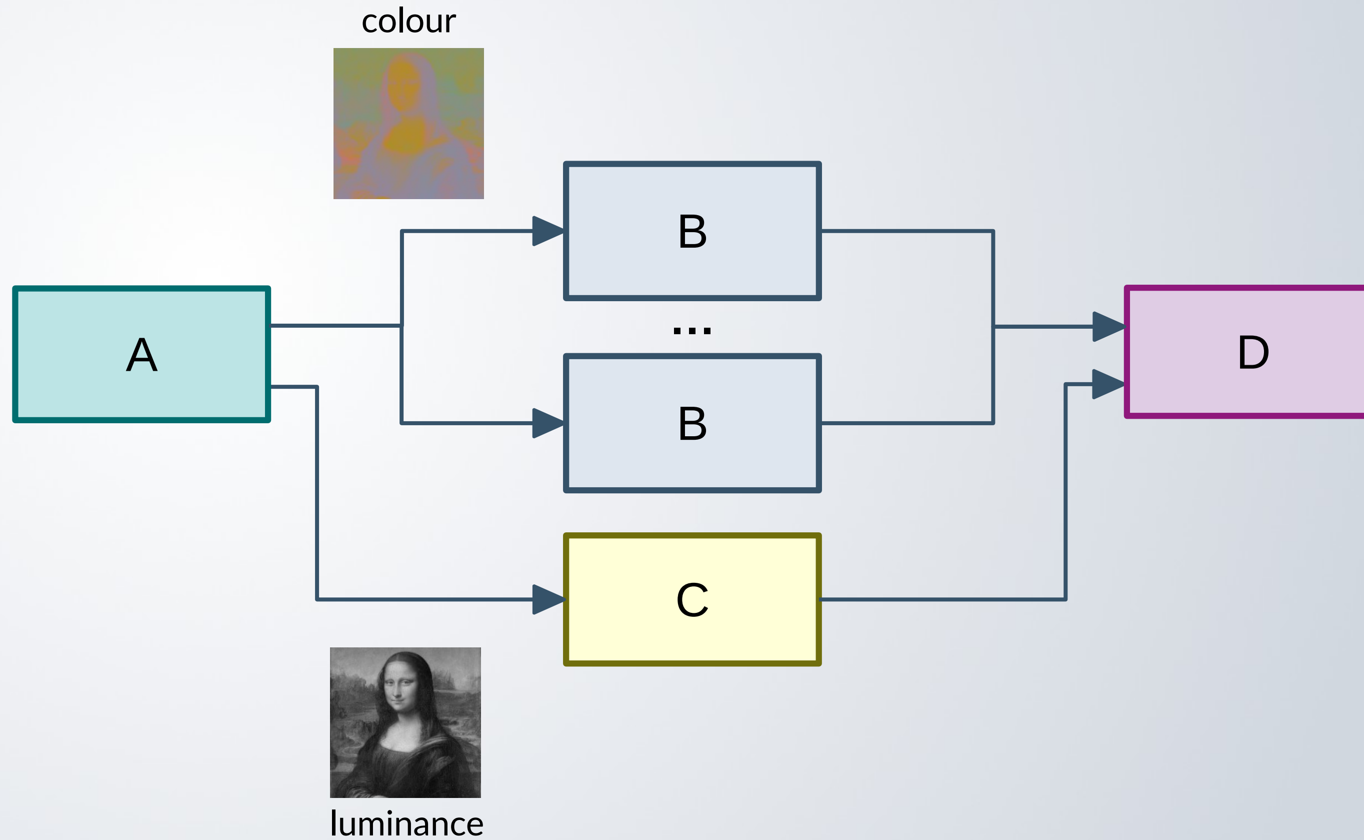
Problem 2 - summary

- Use multiple safety mechanisms for dealing with abnormal situations
  - Avoid putting more messages in topology than it can process
  - Avoid publishing all messages at the same time
  - Configure connections so that you can limit memory usage
  - Pacify memory–hungry processes before oom–killer does it for you
  - Configure oom–killer to kill these before the system is unresponsive

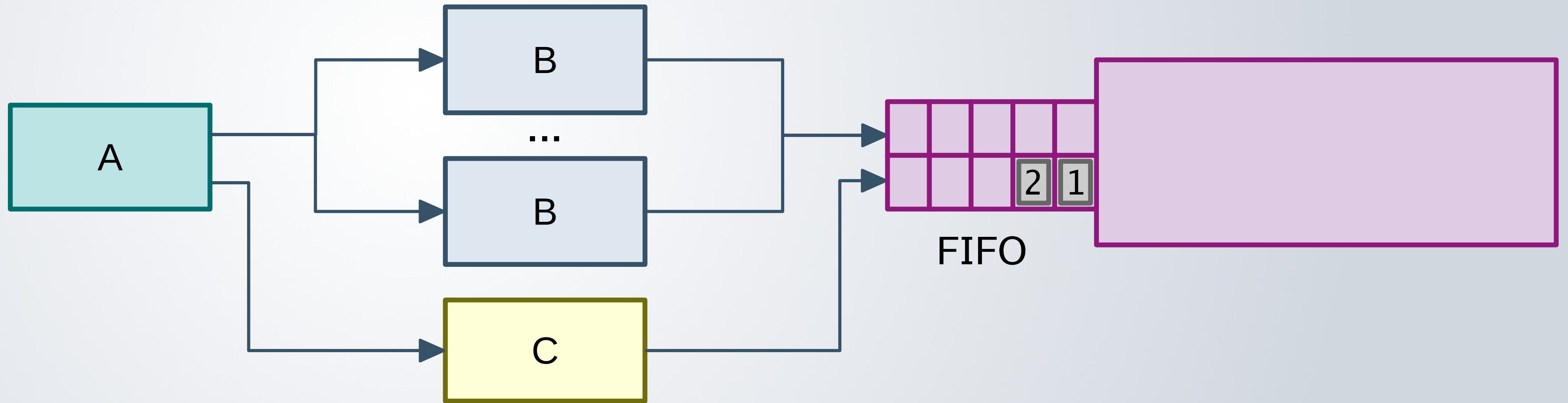


Ian M MacKay

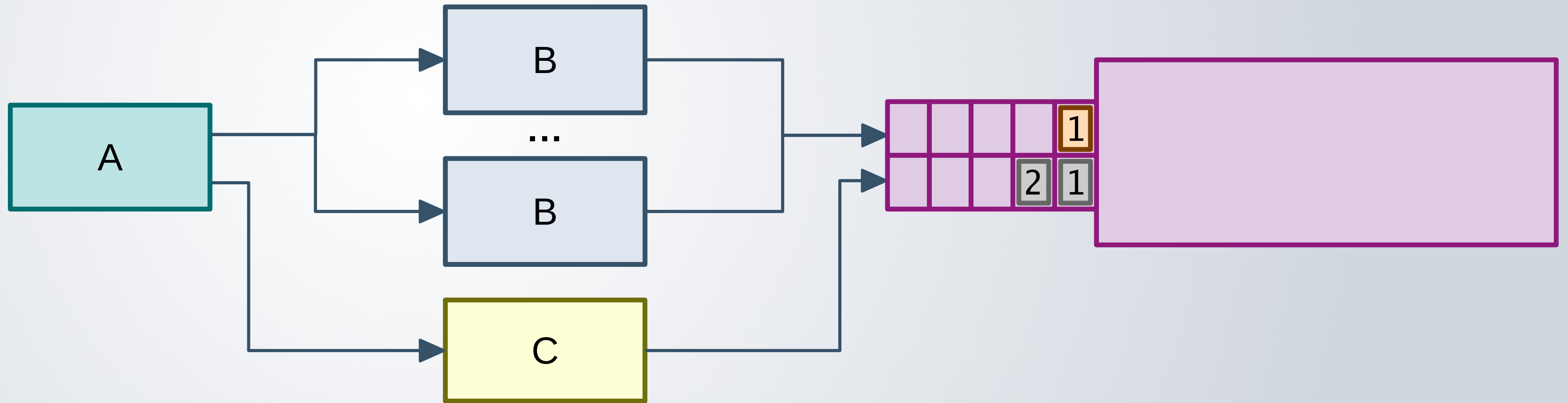
Problem 3 - synchronizing inputs



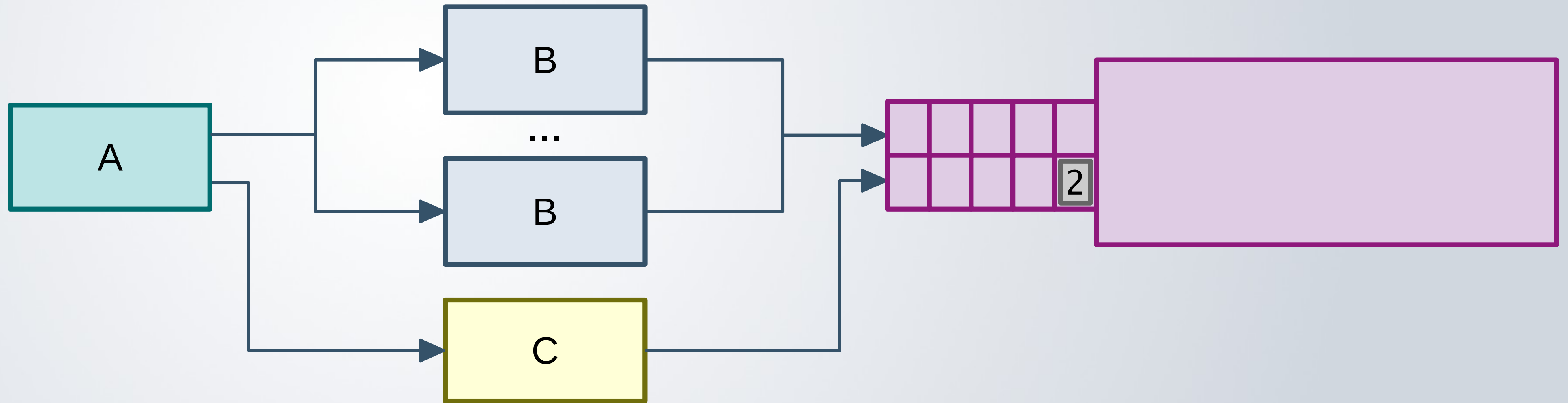
Problem 3 - synchronizing inputs (case 1)



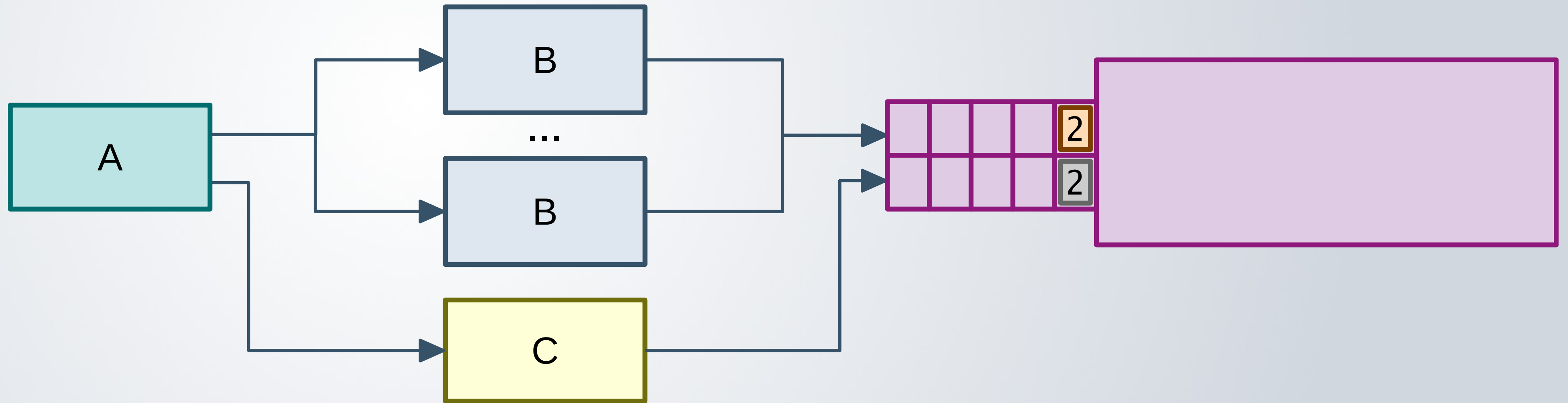
Problem 3 - synchronizing inputs (case 1)



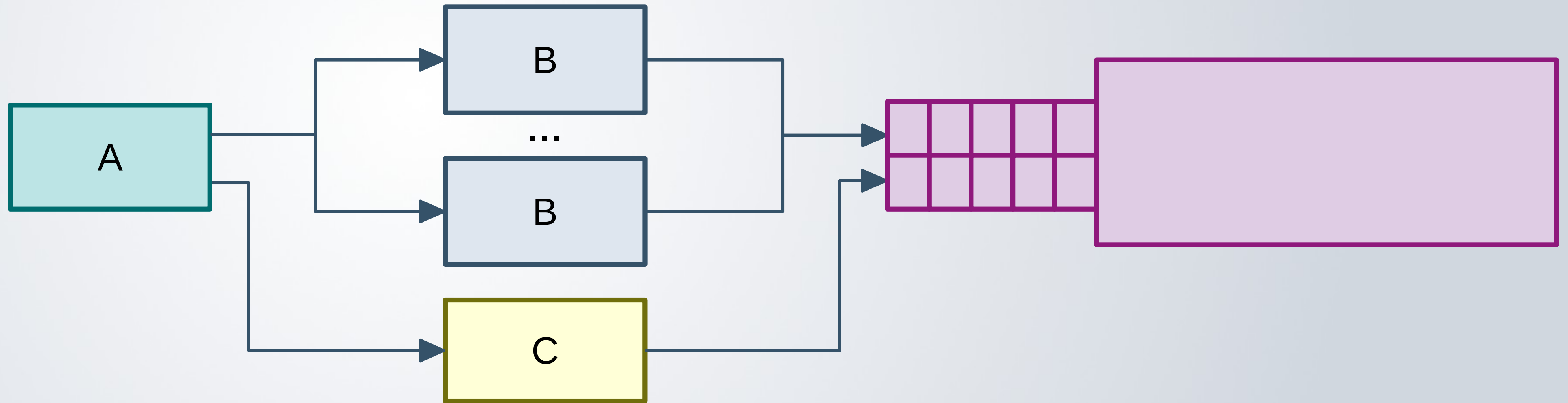
Problem 3 - synchronizing inputs (case 1)



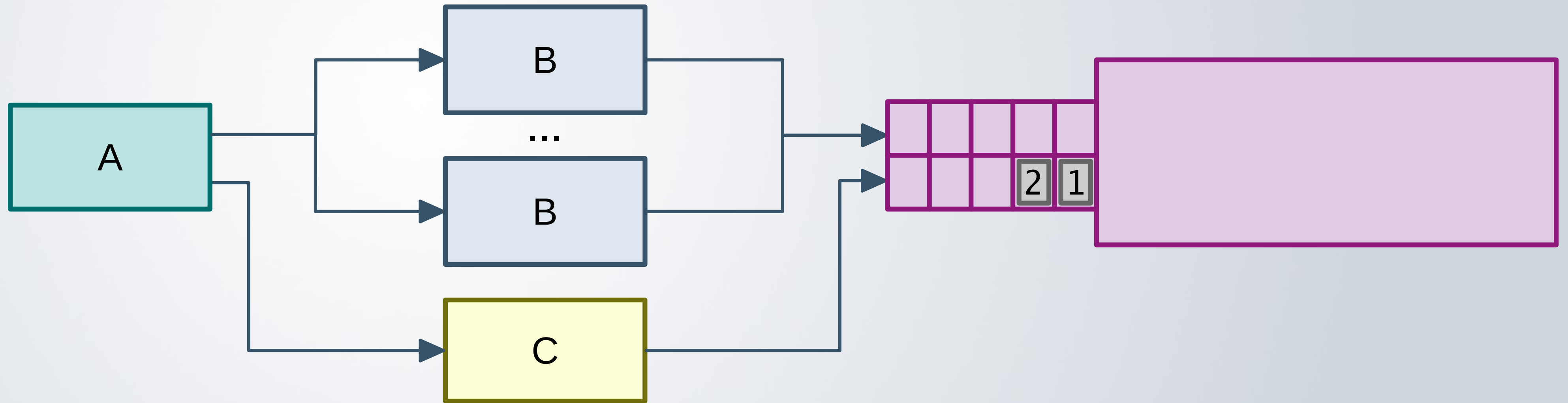
Problem 3 - synchronizing inputs (case 1)



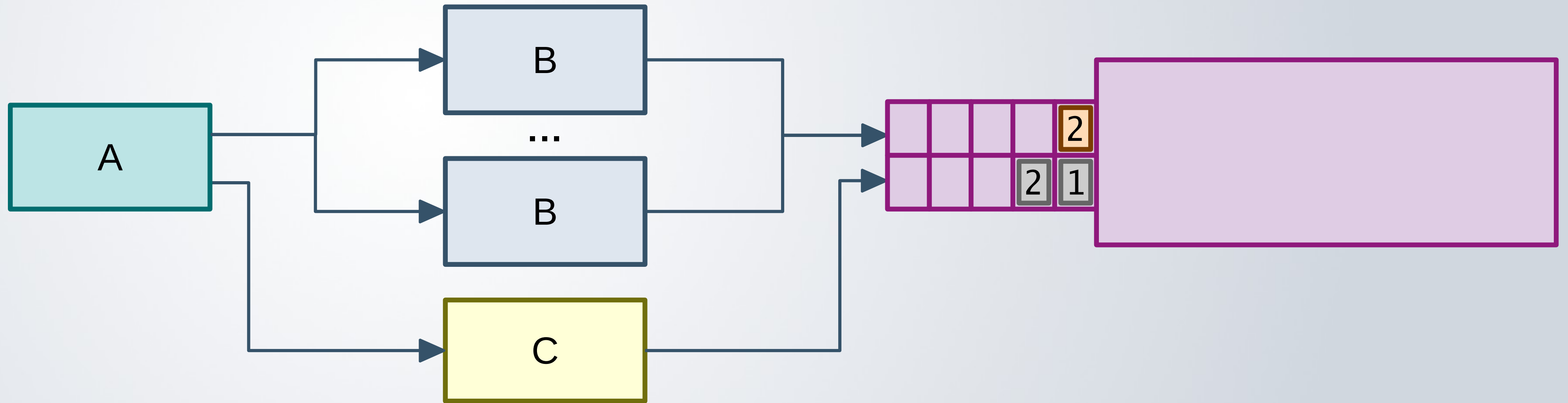
Problem 3 - synchronizing inputs (case 1)



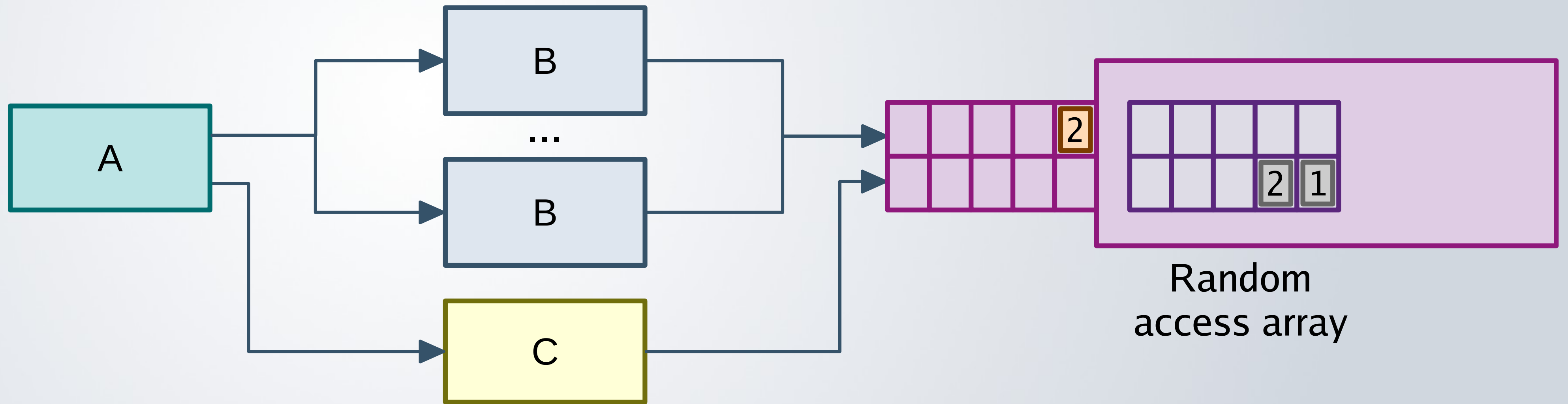
Problem 3 - synchronizing inputs (case 2)



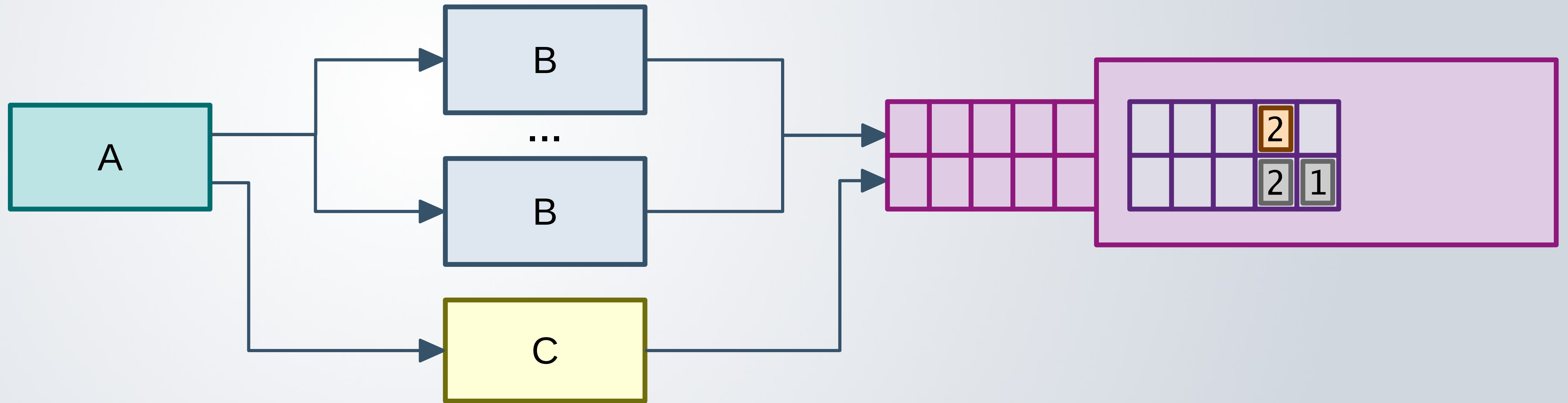
Problem 3 - synchronizing inputs (case 2)



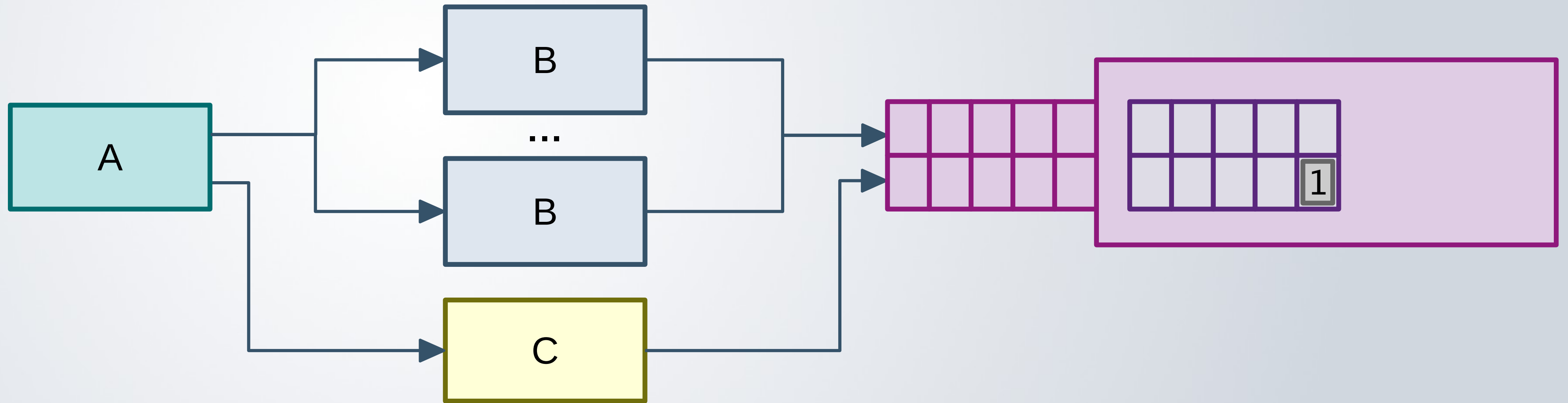
Problem 3 - synchronizing inputs (case 3)



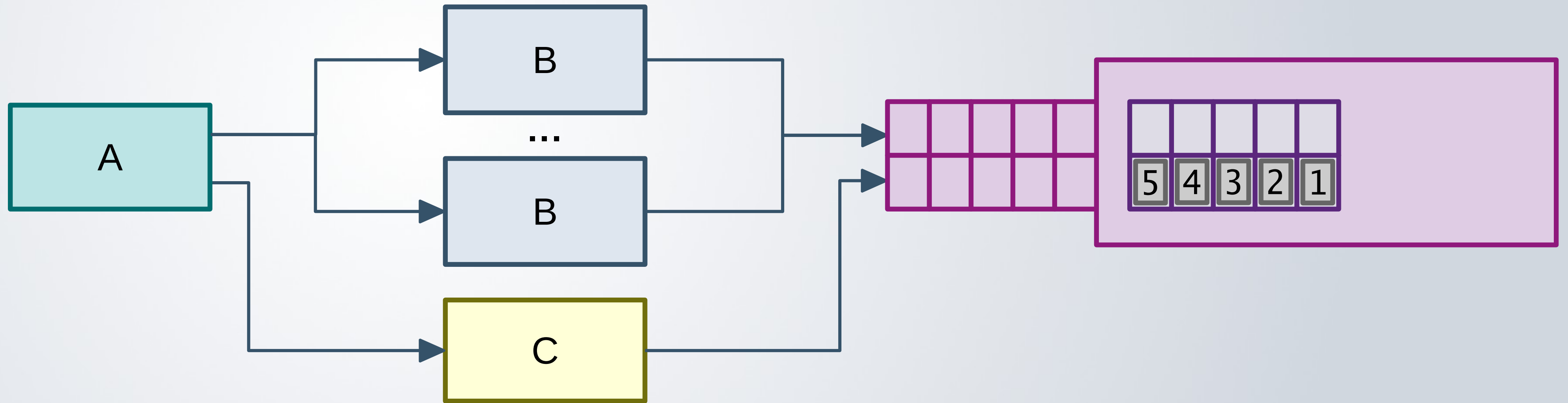
Problem 3 - synchronizing inputs (case 3)



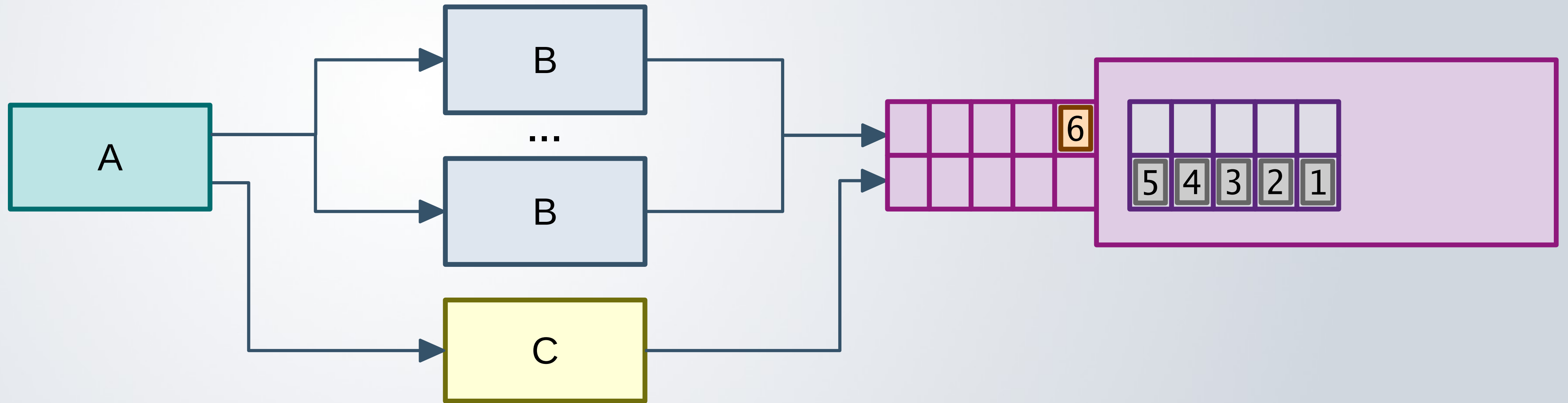
Problem 3 - synchronizing inputs (case 3)



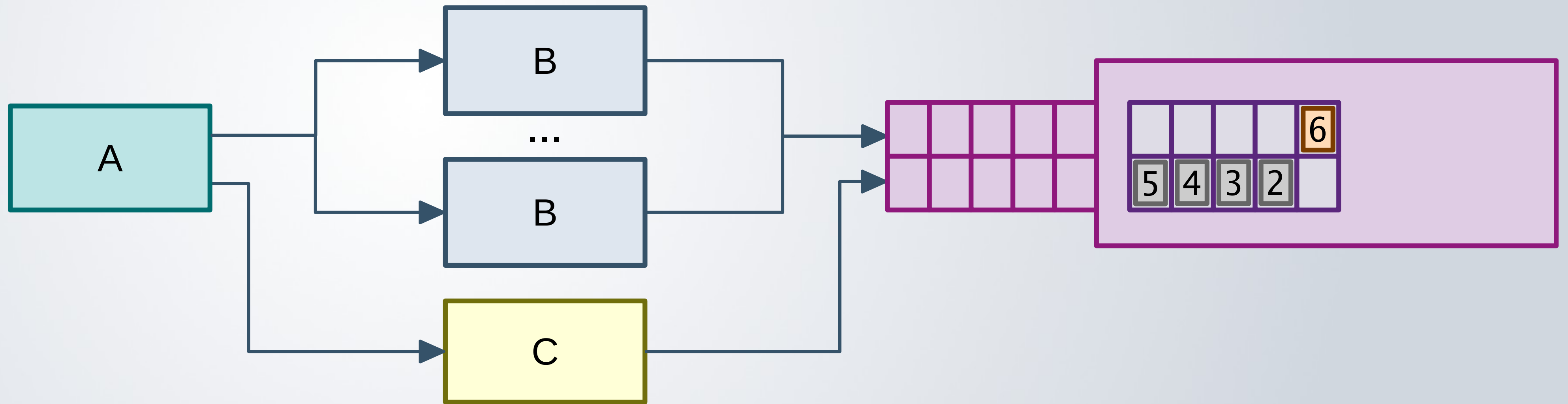
Problem 3 - synchronizing inputs (case 4)



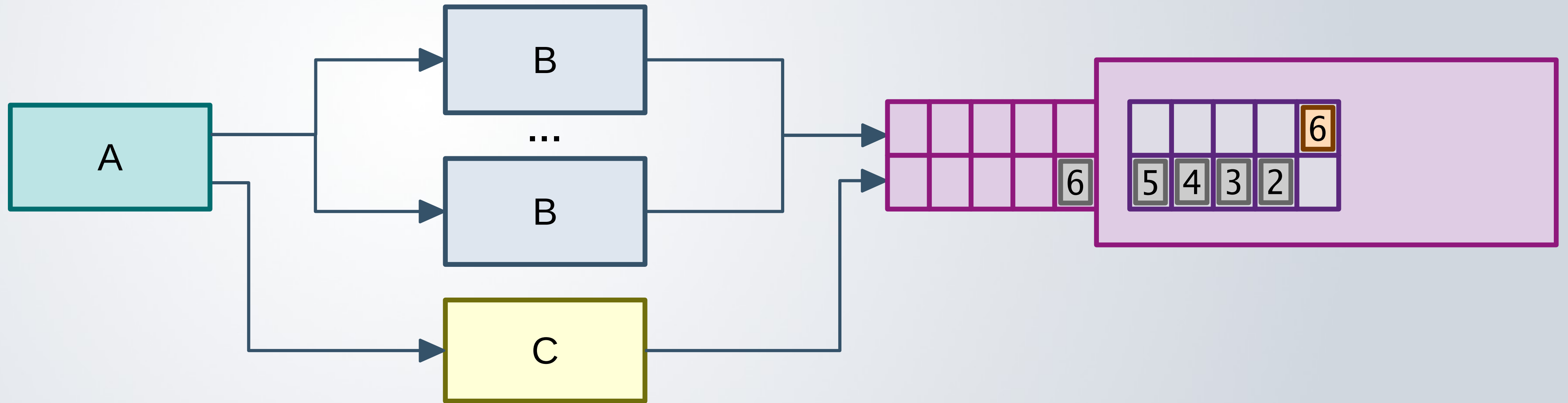
Problem 3 - synchronizing inputs (case 4)



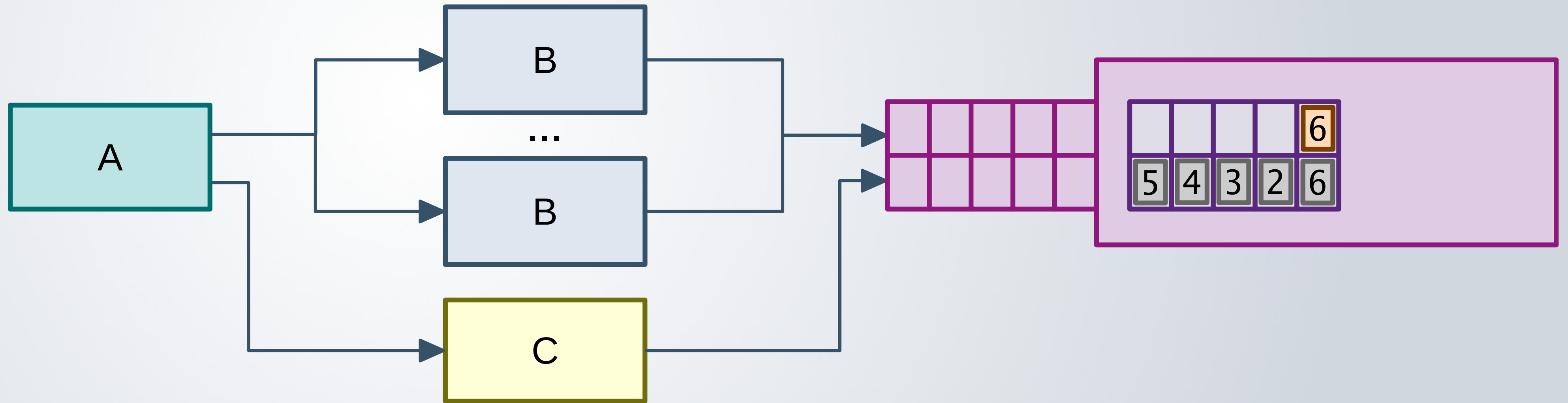
Problem 3 - synchronizing inputs (case 4)



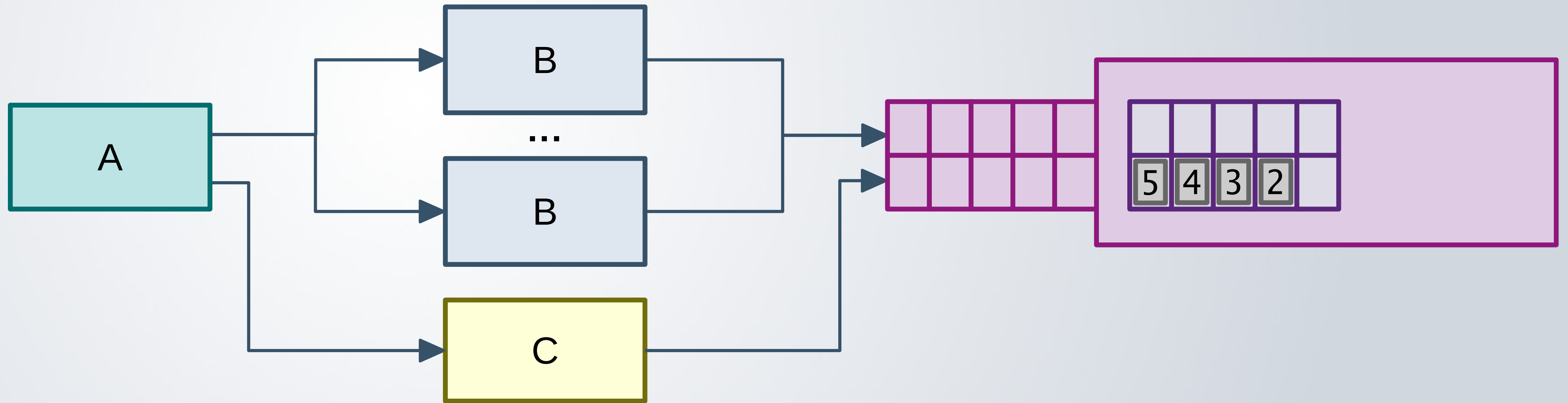
Problem 3 - synchronizing inputs (case 4)



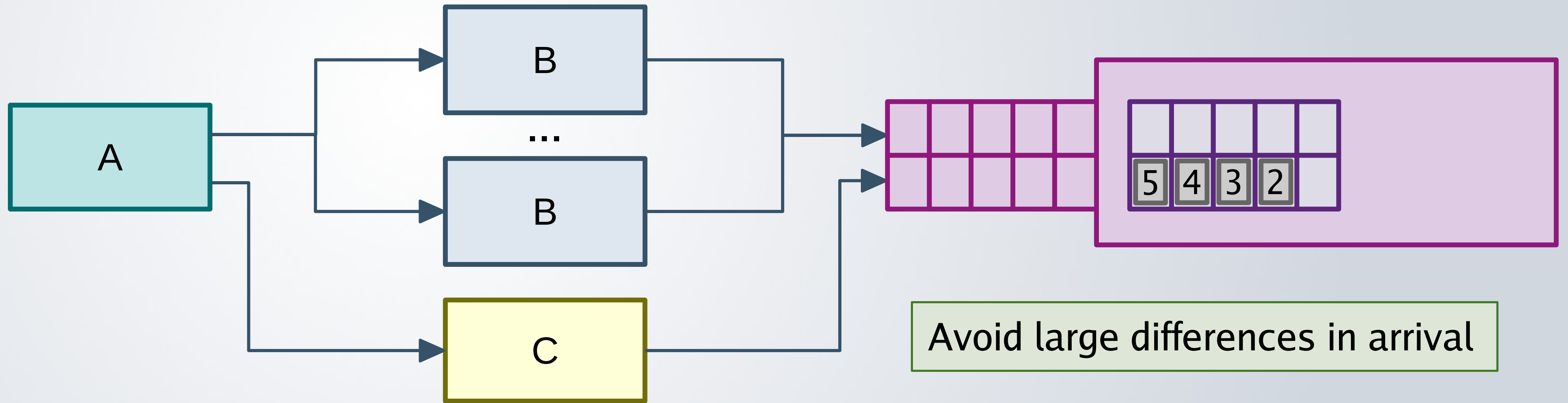
Problem 3 - synchronizing inputs (case 4)



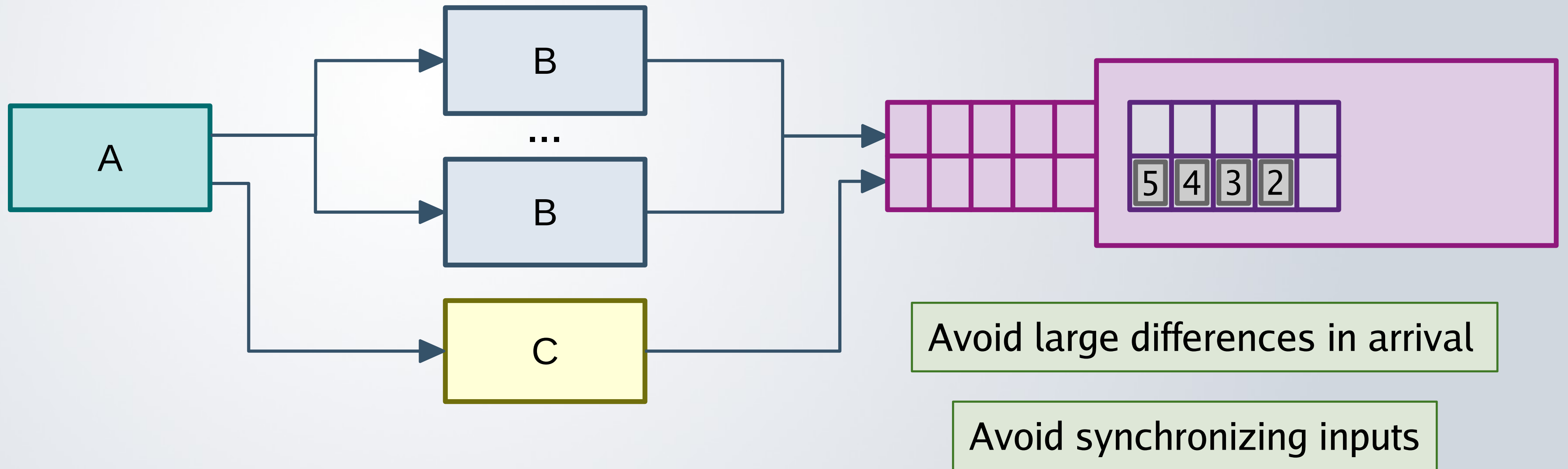
Problem 3 - synchronizing inputs (case 4)



Problem 3 - synchronizing inputs (case 4)

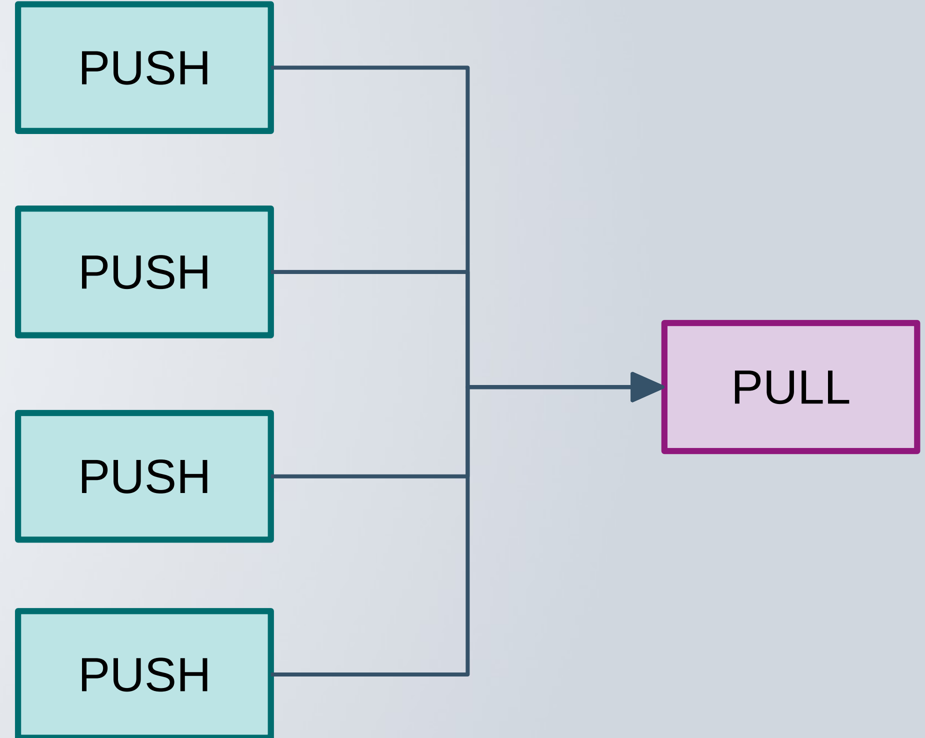


Problem 3 - synchronizing inputs (case 4)



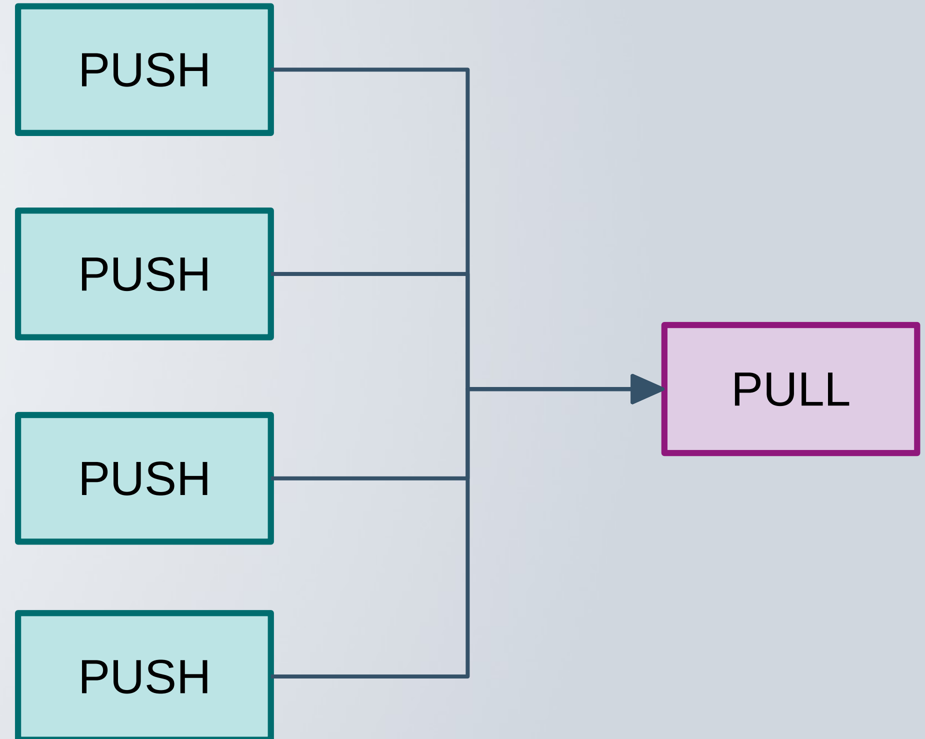
Problem 4

```python
import zmq


ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.connect("tcp://prod01:7891")

s.send_pyobj("hello")
```



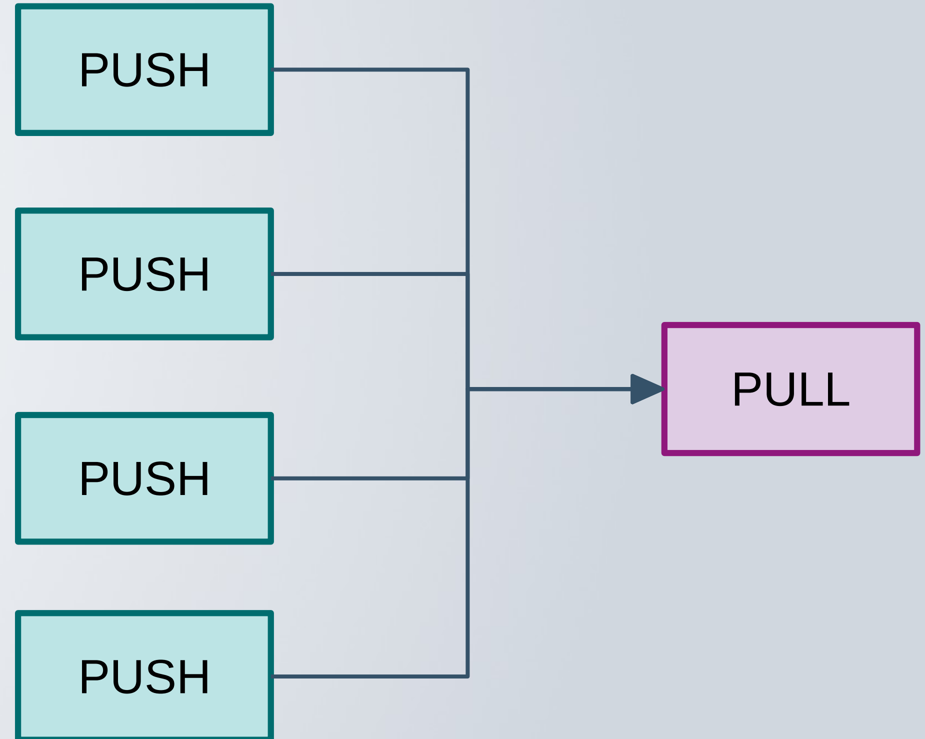
Problem 4 – splitting topologies

```
import zmq
import sys

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.connect(sys.argv[1])

s.send_pyobj("hello")
```



Problem 4 – splitting topologies

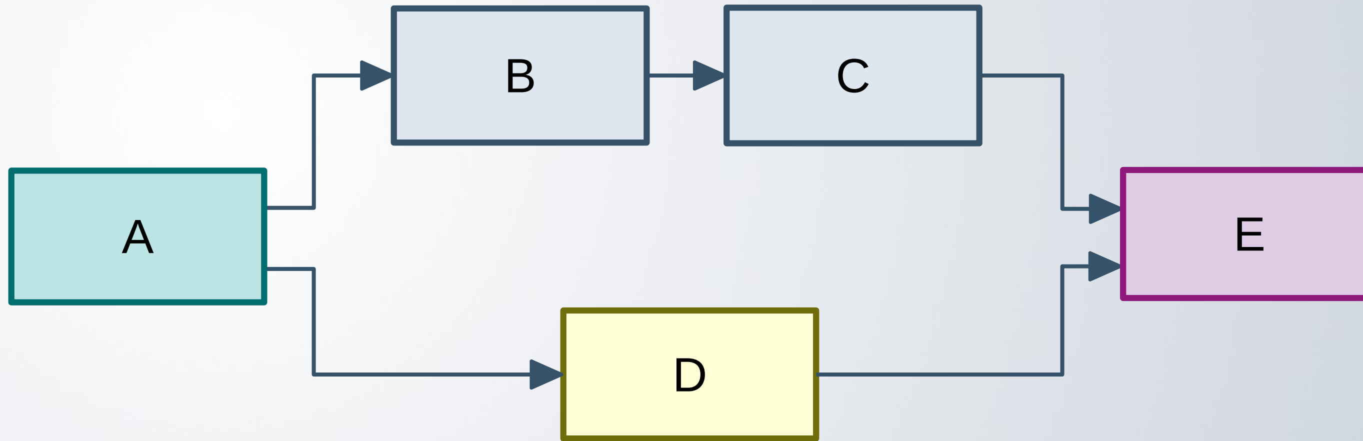
```
import zmq
import sys

ctx = zmq.Context()
s = ctx.socket(zmq.PUSH)
s.connect(sys.argv[1])

s.send_pyobj("hello id" + sys.argv[2])
```



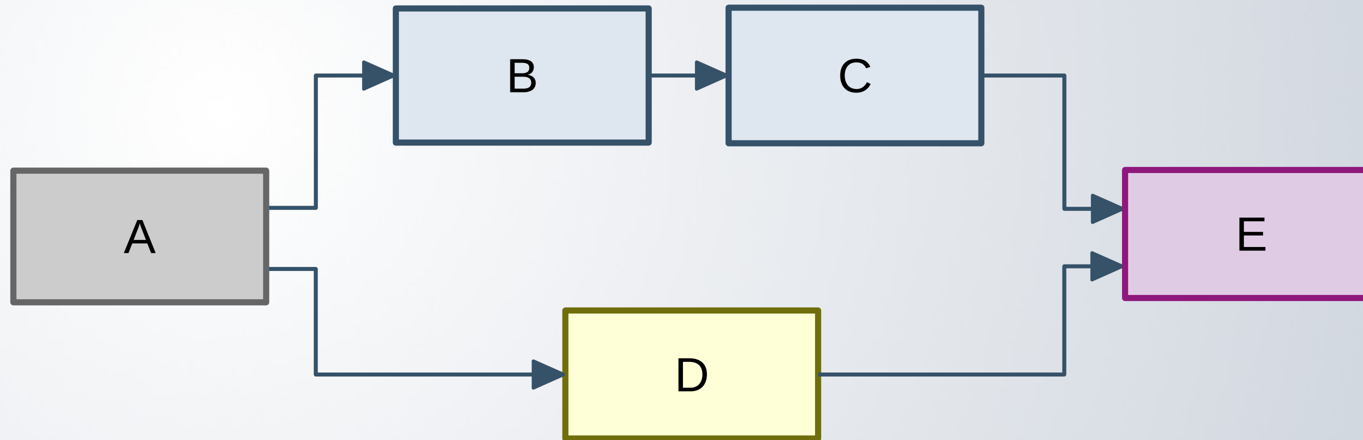
Problem 4 – summary

- Design the system in a way that test setups cannot access production
  - Separate services
  - Separate machines
  - Separate networks
- Check if your input data comes from the right data set
- Consider authenticating

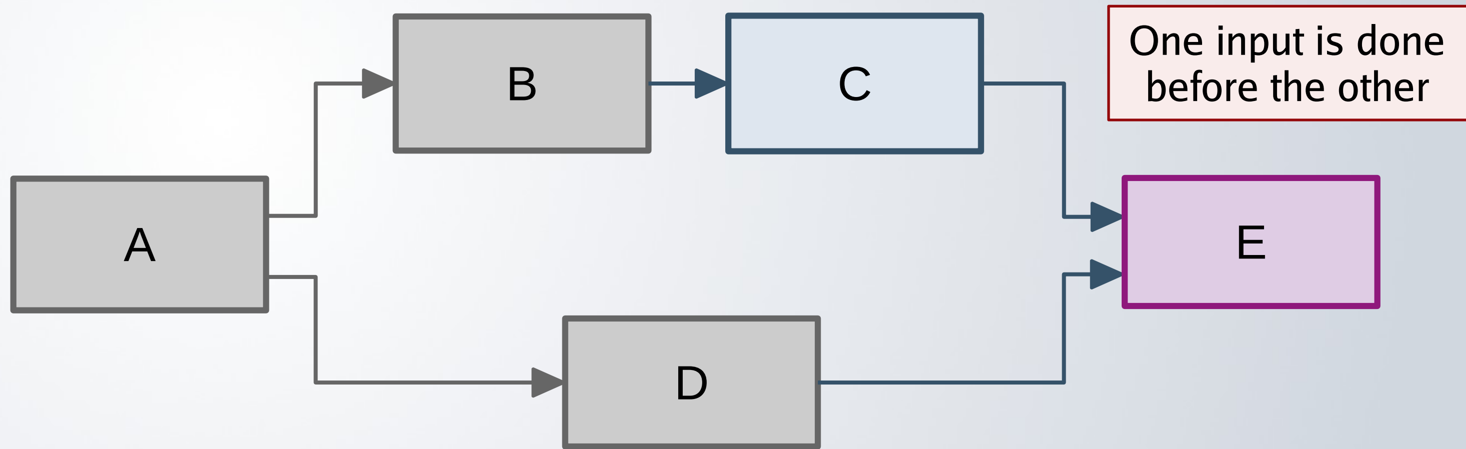
Problem 5 – end of data



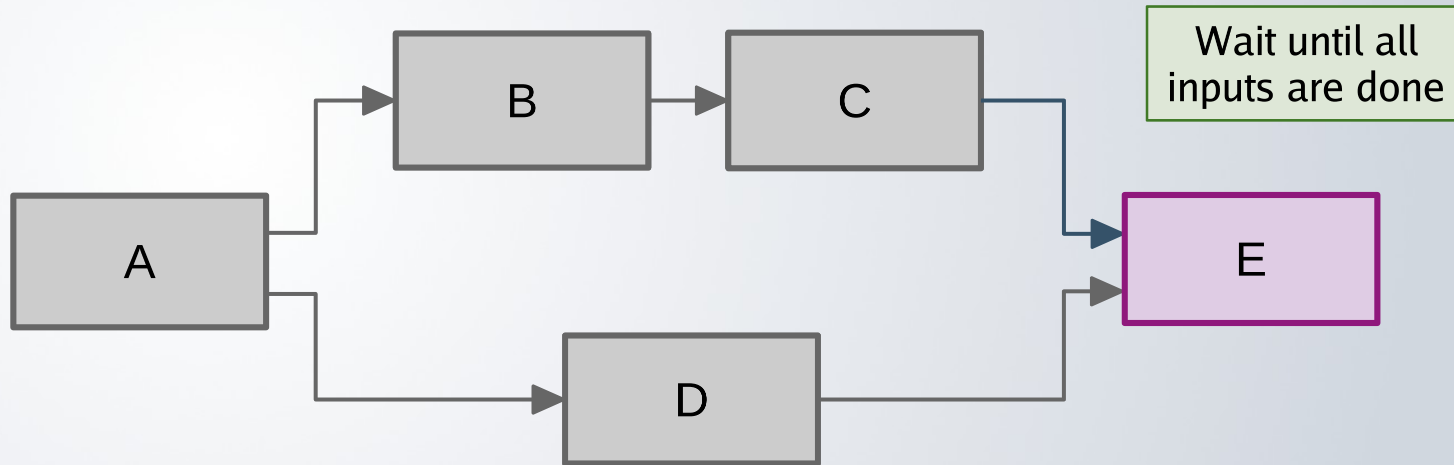
Problem 5 – end of data



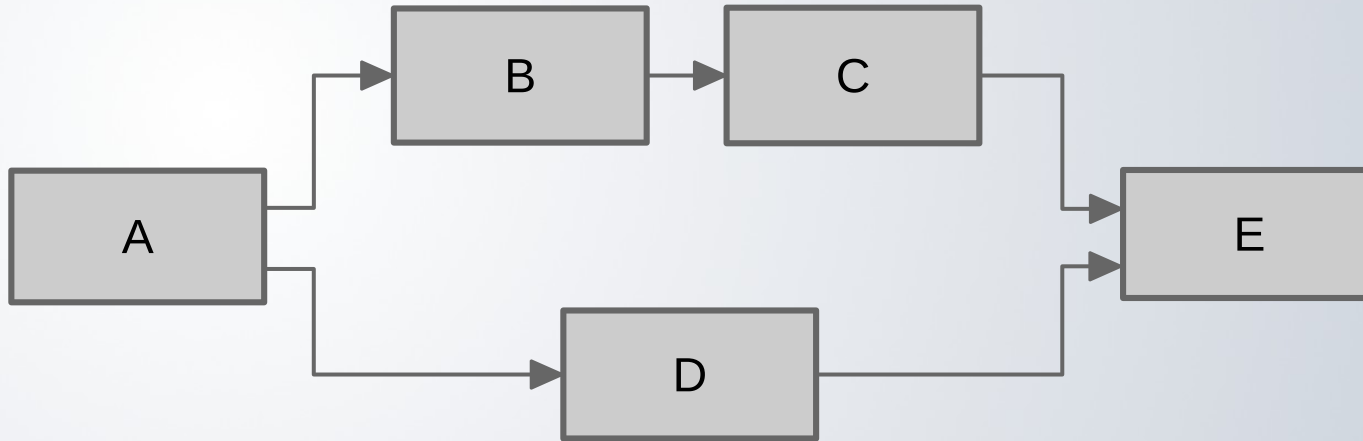
Problem 5 – end of data



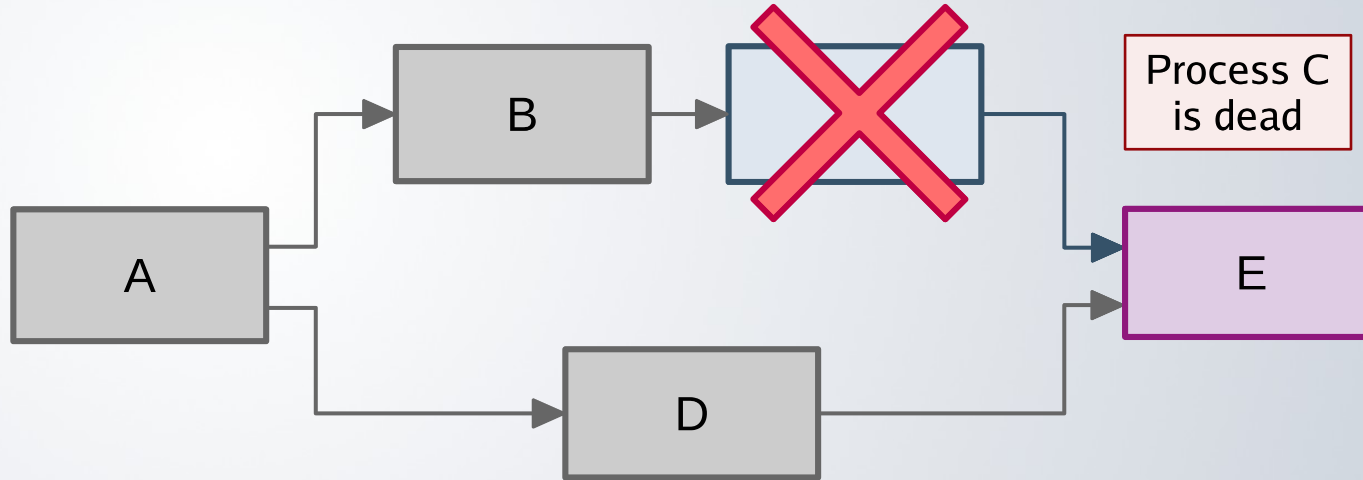
Problem 5 – end of data



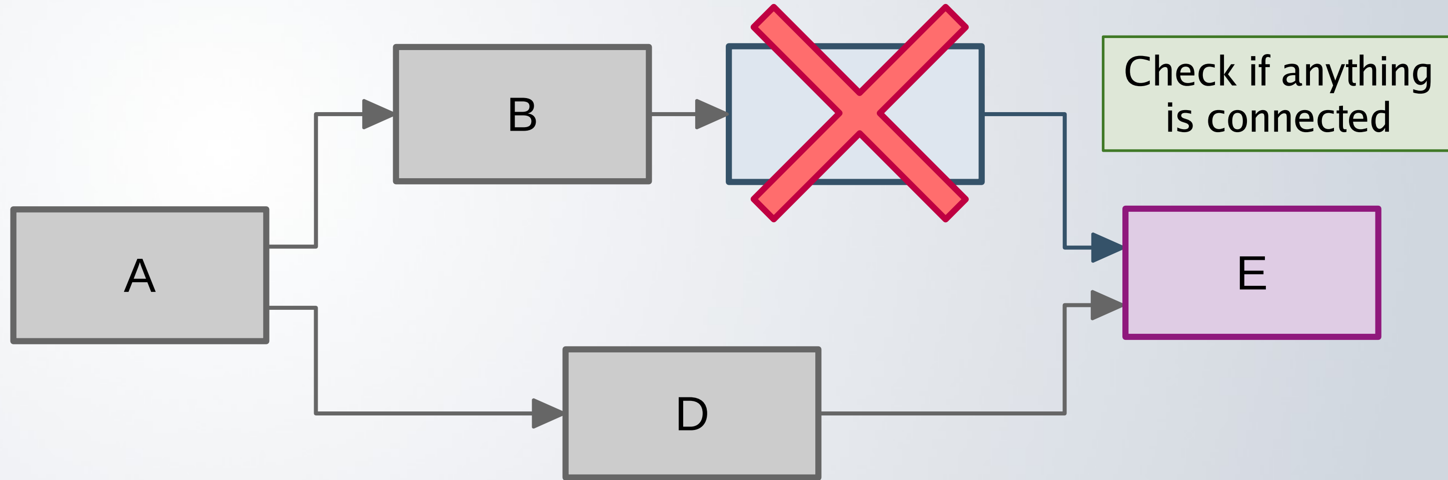
Problem 5 – end of data



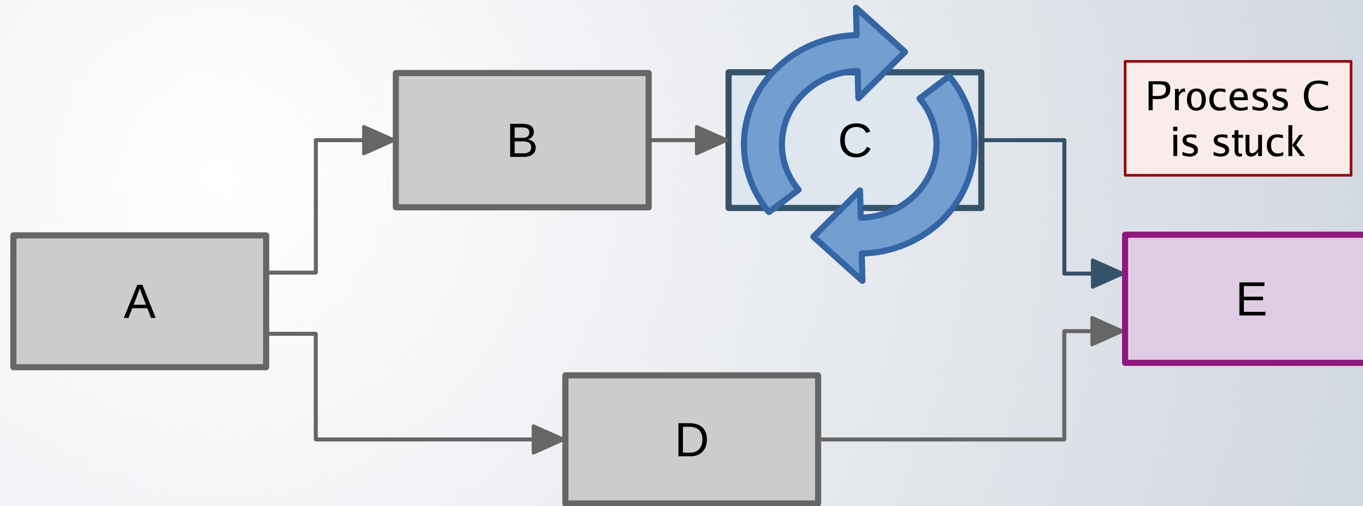
Problem 5 – end of data



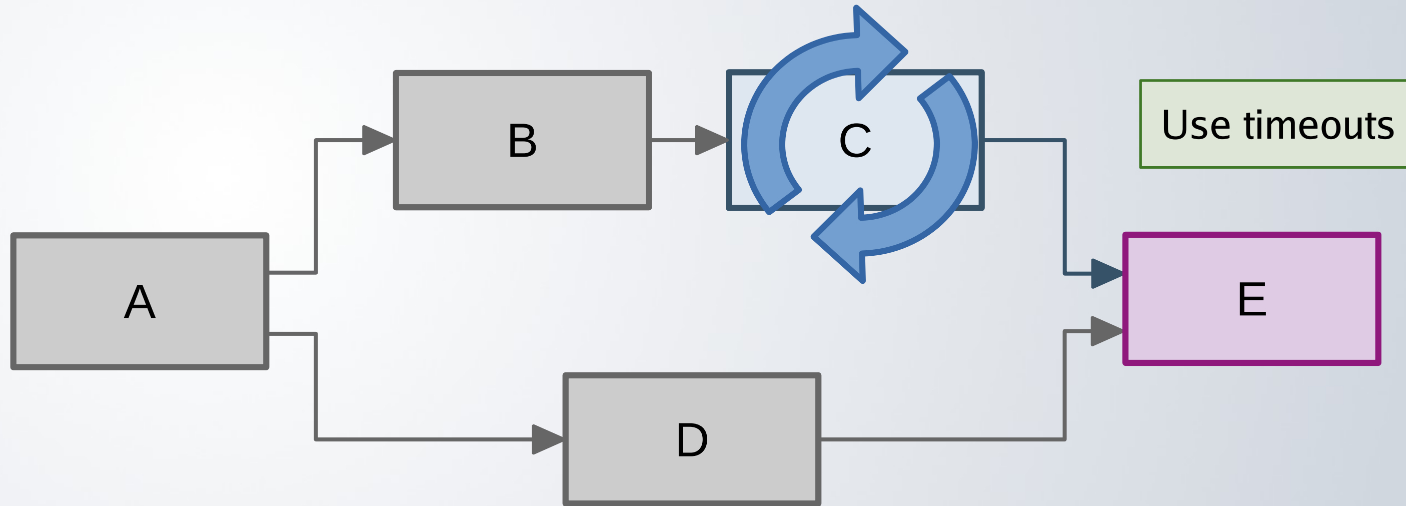
Problem 5 – end of data



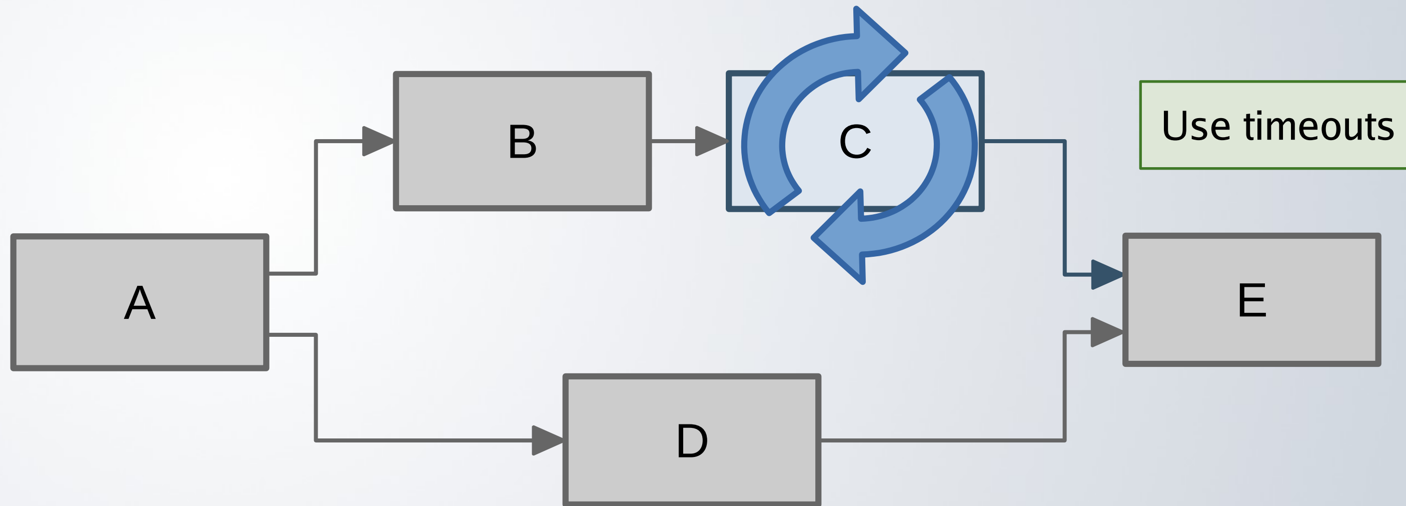
Problem 5 – end of data



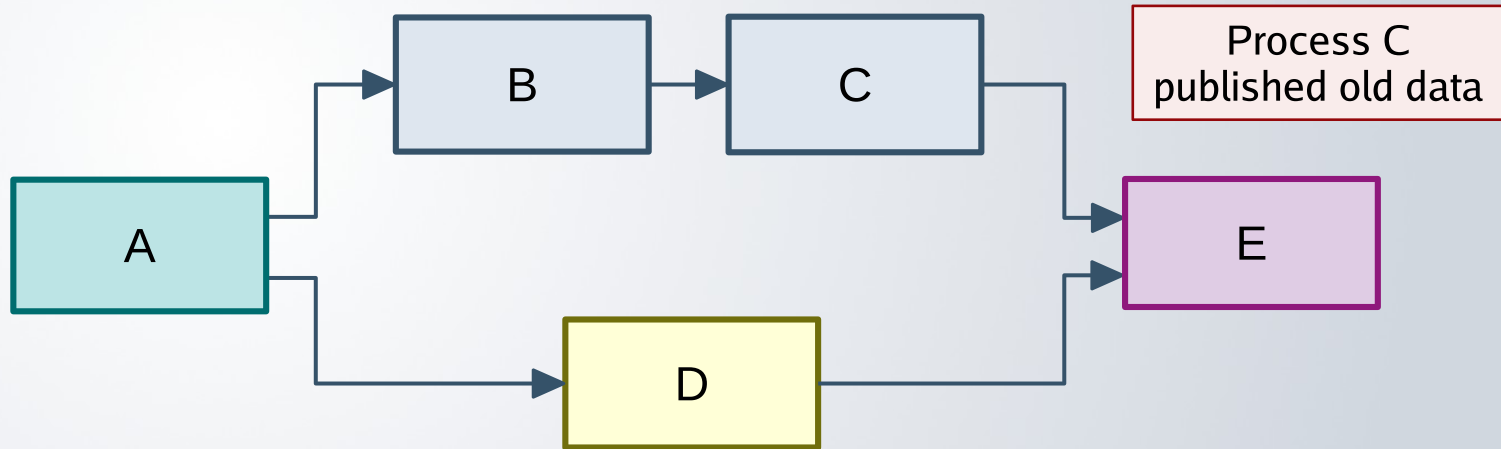
Problem 5 – end of data



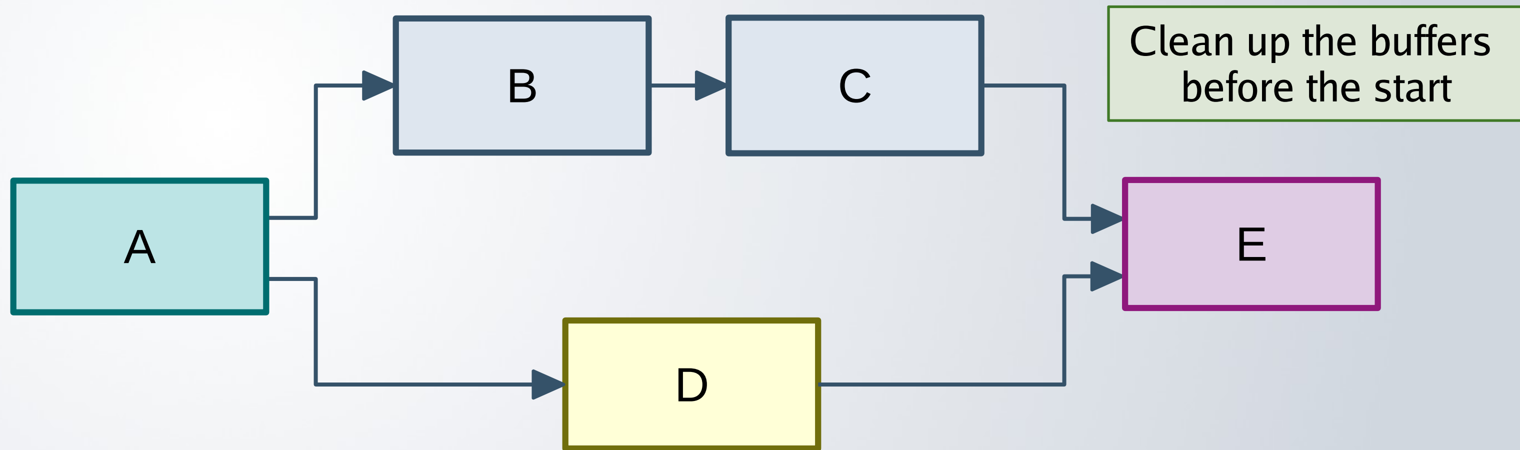
Problem 5 – end of data



Problem 5 – end of data & process reuse



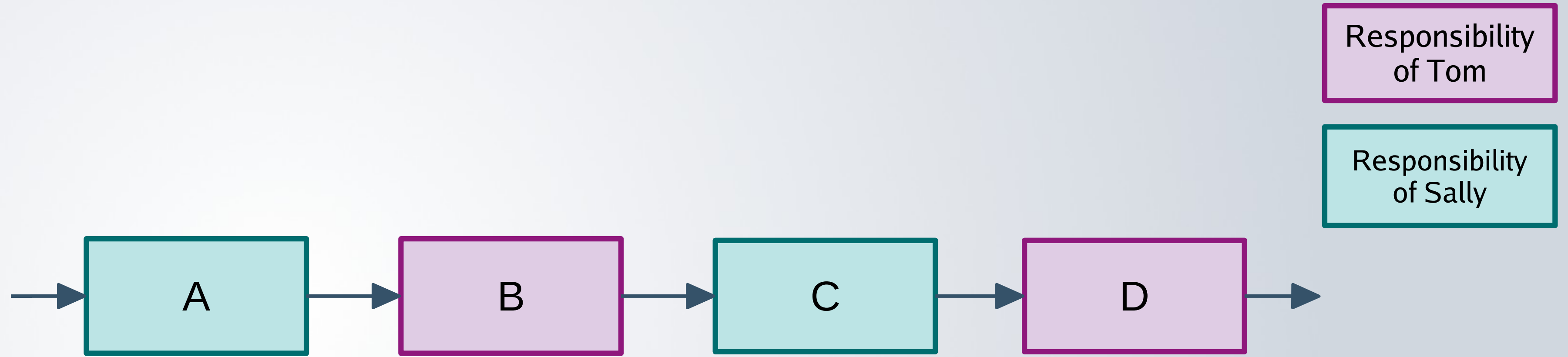
Problem 5 – end of data & process reuse



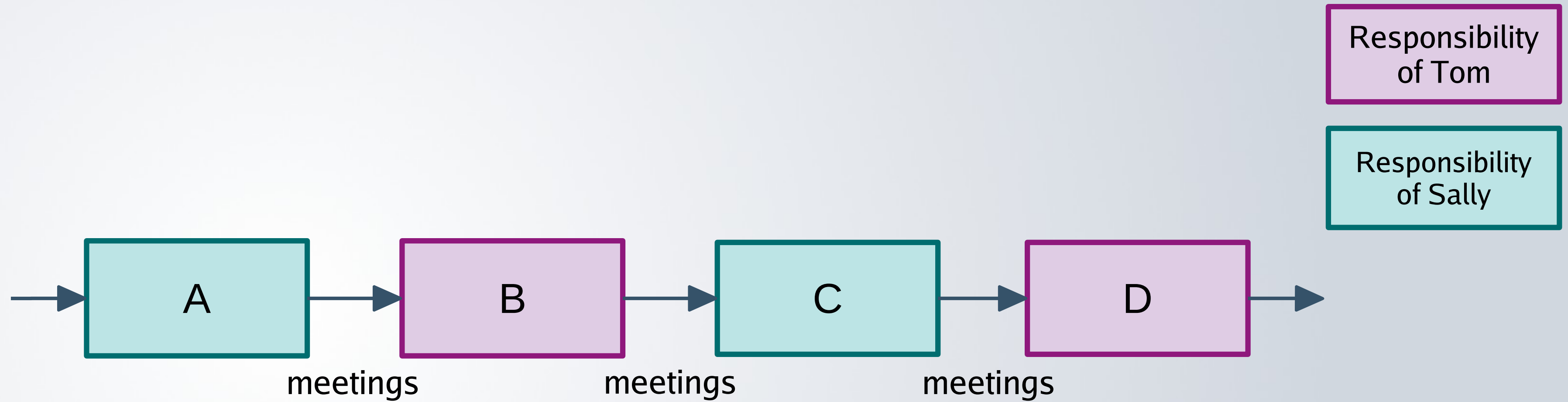
Problem 5 – summary

- End of processing sequence may include:
  - propagating information about end of data
  - checking validity of input channels (if your library allows for it)
  - using timeouts
- Start of processing sequence may include:
  - Cleaning up the queues

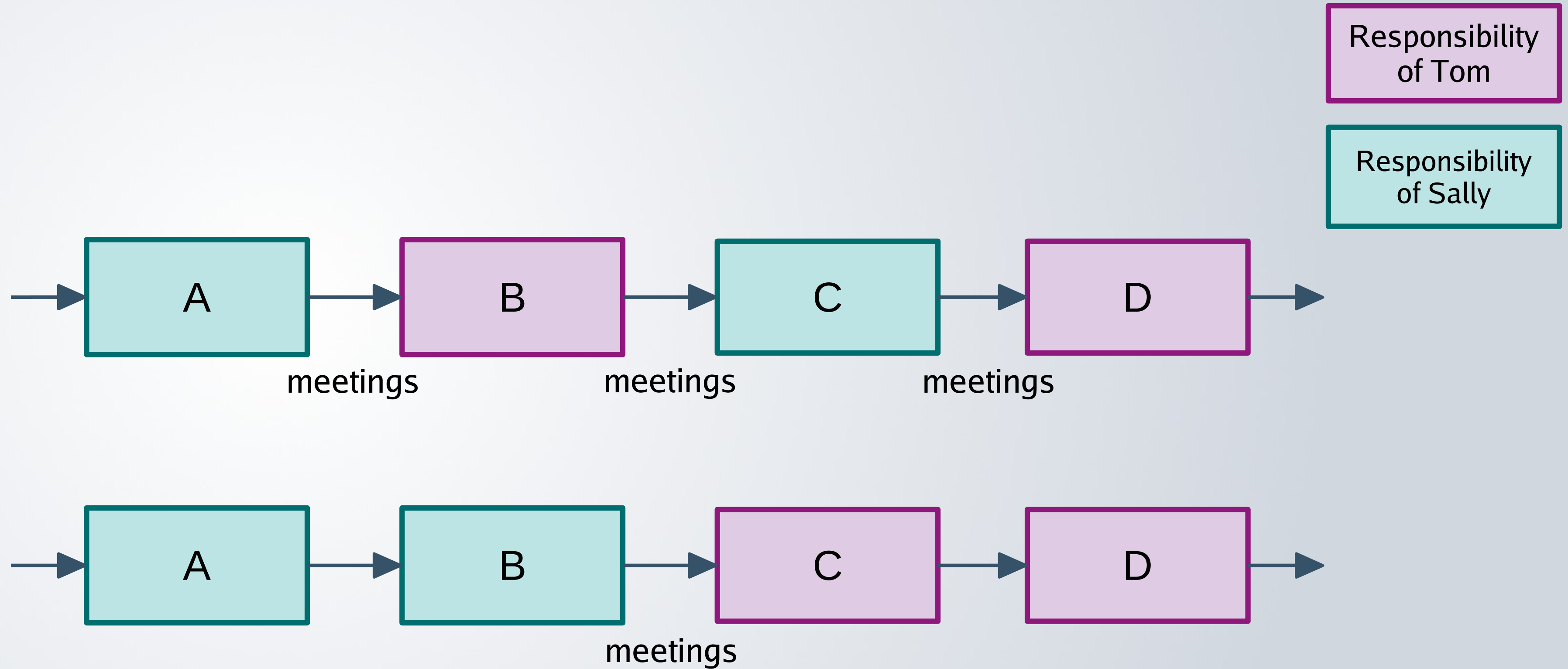
Problem 6 – collaborating on message passing topologies



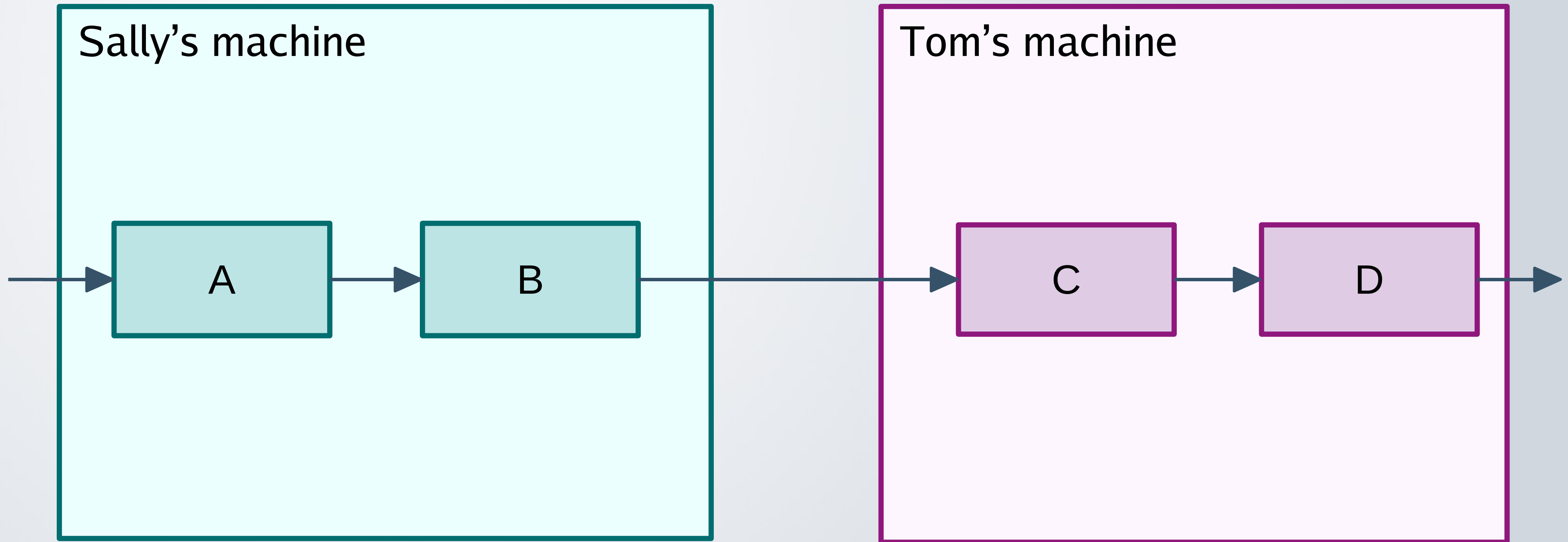
Problem 6 – collaborating on message passing topologies



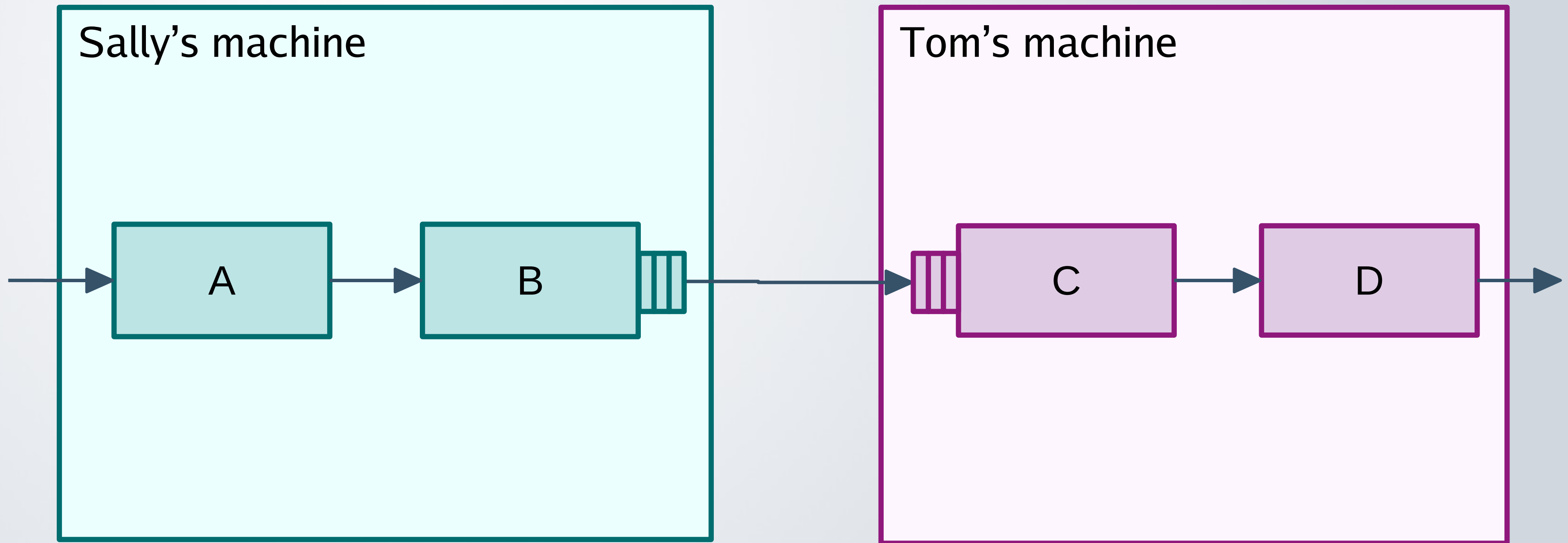
Problem 6 – collaborating on message passing topologies



Problem 6 – collaborating on message passing topologies



Problem 6 – collaborating on message passing topologies



Summary

- Understand the basic concepts of message passing and recognize where it may be applied
- Know your message passing library
- Expect that anything that may happen, will happen
- Consider what to do at the start and end of your application

Summary

- Understand the basic concepts of message passing and recognize where it may be applied
- Know your message passing library
- Expect that anything that may happen, will happen
- Consider what to do at the start and end of your application
- <u>This lecture was not exhaustive!</u>