Bounded buffer with asynchronous message passing

The presented solution (Algorithm 1) makes use of dedicated ports for messages that play a specific role in the communication scheme, reported in Figure 1.

The buffered data elements are kept in the receiving queue for port `indata_p`; the handshake between a *Producer* and *Buffer* is aimed at granting a permit to send only when the queued data elements are less than `MAX_N`.

Data elements are removed from the `indata_p` queue only upon the receipt of a request from a *Consumer*.

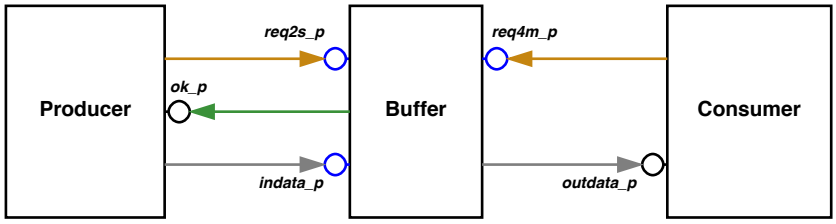

Figure 1: **Scheme for Bounded Buffer with Asynchronous Send.**

NOTES

In the code, both `send` and `receive` are supposed to be methods of and `EndPoint` object.

A `receive` is given the variable to store the incoming content, and it returns the `Process` corresponding to the sender.

An endpoint is returned by the factory method `EndPoint.newEP(proc, port)` given the corresponding process and port objects.

A Dijkstra's guard on receive is indicated by the syntax `rcvGuard( <condition>;<receive op.>) { <instructions> }`, and the corresponding repetition list is indicated by a `do/while(true)` loop.

A more classical approach may exploit an internal data structure (let's call it *List*) to store the buffered data elements. In such a case, the receive on `indata_p` can be performed directly by *Buffer* in the first guard after sending out the permit to send the data element. Insertion in *List* would be performed just after receiving a data element from a *Producer*, and extraction from *List* must precede any send of a data element to a *Consumer*.

Algorithm 1 Example in Pseudo-Java of a Bounded Buffer implementation with asynchronous message passing and Dijkstra's guarded receives.

```
//BUFFER process; data elements are kept on the indata port queue

Message data, req2s, req4m, ok; //messages

Proc me = new Proc(...),  //initialized process
   producer, consumer;    //not initialized

Port req2s_p, ok_p, req4m_p, //signal ports (initialization omitted here)
        indata_p, outdata_p; //data ports   (ditto)

EndPoint req2s_ep  = EndPoint.newEP(me, req2s_p);
EndPoint indata_ep = EndPoint.newEP(me, indata_p);
EndPoint req4m_ep  = EndPoint.newEP(me, req4m_p);
EndPoint producerok_ep;     //not initialized (not known yet)
EndPoint consumerdata_ep;   //ditto

int howmany=0; //nr of data elements currently held

do {   //repetitive structure for guarded commands;
  rcvGuard( howmany < MAX_N;
            producer = req2s_ep.receive(req2s) ) {
    howmany++;
    producerok_ep = EndPoint.newEP(producer, ok_p);
    producerok_ep.send(ok);
  }
  rcvGuard( howmany > 0;
            consumer=req4msg_ep.receive(req4msg) ) {
    producer = indata_ep.receive(data); //this is immediately done!
    howmany--;
    consumerdata_ep = EndPoint.newEP(consumer, outdata_p);
    consumerdata_ep.send(data);
} while(true);
```

```
//PRODUCER process:
Message data, req2s, ok;
Proc me,buf,otherp;//init omitted
Port data_p,req2s_p,ok_p;//ditto
//end points
EndPoint ok_ep =
  EndPoint.newEP(me,ok_p);
EndPoint breqs_ep =
  EndPoint.newEP(buf,req2s_p);
EndPoint bdata_ep =
   EndPoint.newEP(buf,data_p);
while(true) {
  data = produce_content();
  breqs_ep.send(req2s);
  //blocking
  otherp = ok_ep.receive(ok);
  bdata_ep.send(data);
}
```

```
//CONSUMER process:
Message data, req4m;
Proc me,buf,otherp;//init omitted
Port data_p,req4m_p;//ditto
//end points
EndPoint breqm_ep =
   EndPoint.newEP(buf,req4m_p);
EndPoint data_ep =
   EndPoint.newEP(me,data_p);


while(true) {
  breqm_ep.send(req4m);

  //blocking
  otherp=data_ep.receive(data);

}
```