

# **Memory Management and Buffer Sharing in DRI Early Binding Communication**

**Ken Cain, Mercury Computer Systems, Inc., [kcain@mc.com](mailto:kcain@mc.com)**

**August 23, 2001**

**“Skeleton” (working) Presentation to be used by the Data Reorganization Forum at  
the August 2001 meeting to assess current DRI interfaces**

# Outline

- 0 **What buffer sharing scenarios do we want to support?**
- 0 **Basic “tenets” of buffer sharing for early binding data reorgs**
- 0 **What can we actually do with the API as it exists in August 2001?**
- 0 **Potential API “issues” or problems**
- 0 **Recommendations**

# Which Buffer Sharing Scenarios Should We Support?

0 Scenarios are categorized according to their selection of each of the following 4 attributes:

- Memory allocated by: <user, library>
- Data parallel design: <clique, pipeline>
- Processing memory disposition: <in-place (IP), out-of-place (OOP)>
- Data Reorg memory disposition: <in-place (IP), out-of-place (OOP)>

0 Cases listed in (perceived?) increasing order of implementation difficulty (with respect to buffer sharing):

- User / Pipeline / OOP-proc / OOP-dr [OOP-dr is implied with pipeline]
- User / Pipeline / IP-proc / OOP-dr
- User / Clique / OOP-proc / OOP-dr
- User / Clique / IP-proc / OOP-dr
- User / Clique / OOP-proc / IP-dr
- User / Clique / IP-proc / IP-dr

■ - ... and then the same group of six combinations with **library allocated memory** instead of user allocated memory

■ This is what we will focus on in this working presentation

# Basic “Tenets” of Early-Binding Reorgs

- 0 *See August 2001 meeting minutes for discussion on these bullets (some “tenets” were disputed, modified)*
- 0 With respect to an individual DRI\_Reorg, buffers in a bufferset should be accessed in order
- 0 For DRI\_Reorgs that share the same bufferset, each will still access buffer 0 first, then buffer 1, then buffer 2
- 0 For performance, and for ease of implementation, we need to make the number of buffers in a bufferset the same on BOTH sides of a DRI\_Reorg
  - Currently, we only require the number of buffers in each process be the same as all other peer processes on the SAME side of a Reorg
- 0 Making these restrictions reduces the number of low-level DMA transfers that need to be set up in advance (a concern on embedded platforms)

# It All Seems to Boil Down to...

## Multi-Buffering Application Support

### 0 A choice between 2 approaches

- user explicitly “puts” buffer to the next DRI\_Reorg that will use it
  - This would modify the current get/put semantics in DRI
- library manages state of buffers in the buffersets (current approach)
  - which buffers are occupied by a DRI\_Reorg (and which one)
  - which buffers are occupied by the application
  - which buffers are available (for the application, or for ANY other DRI\_Reorg)

### 0 We need to consider 1 thread vs. multi-thread environments:

- Library managed buffer access approach
  - Difficult/impossible to make work properly in single-threaded environment?
  - a natural for multi-threaded, locking/synchronization handled in library
    - locking for access to the bufferset itself
    - locking / synchronization to maintain buffer order among DRI\_Reorgs
- User managed buffer access approach
  - a natural for conventional single-threaded “multi-buffering” model
  - Probably wouldn’t be used in multi-threaded applications, but would require the user to implement locking described above in the “library managed” approach

# Ok, So How Do We Organize This Mess?

## 0 For each of the 6 cases of interest

- **Library memory, clique or pipeline, IP or OOP proc, IP or OOP dr**
  - Show how user code would look if the library handled buffer sharing
  - Show how user code would look if buffer sharing was implemented manually

We will show code in yellow background that reflects how buffer sharing would be done using DRI under its current assumptions:

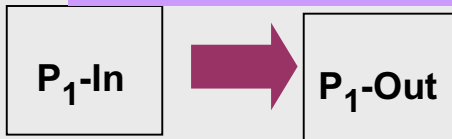
- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

We will show code in blue background that reflects how a user would manually use get, put, and a new "recycle" function to implement buffer sharing. This assumes:

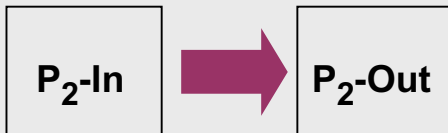
- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

# Legend for Diagrams

Processing stage 1,  
Input and output



Processing stage 2,  
Input and output



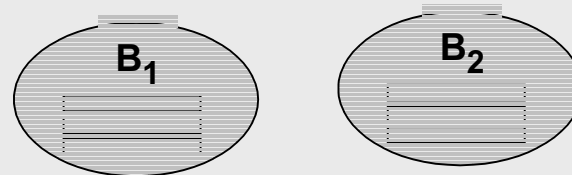
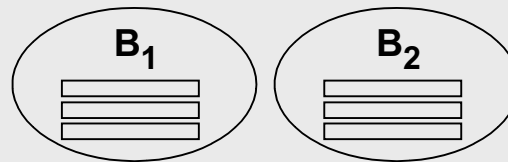
Data Reorg stage 1,  
Send and receive sides



Data Reorg stage 2,  
Send and receive sides



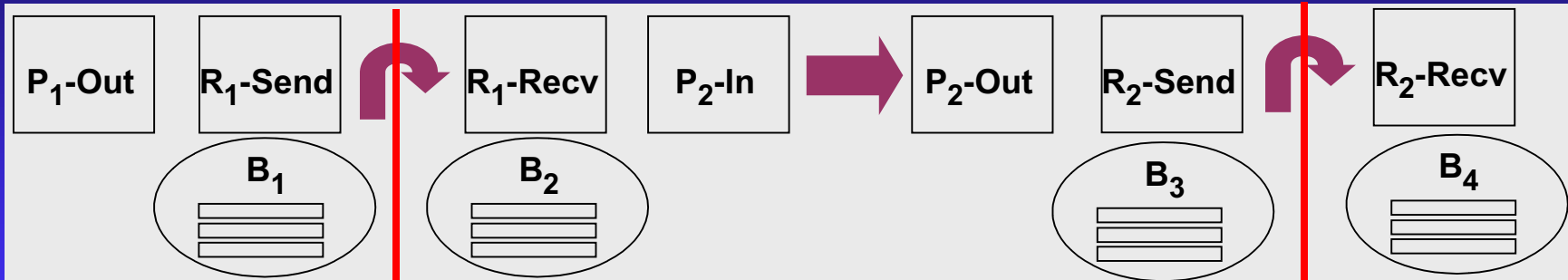
Buffersets 1 and 2



Buffersets 1 and 2 –  
Shaded indicates that it references  
A bufferset also shared by another  
DRI\_Reorg

— Represents process set boundary in a pipeline

# Lib-Mem / Pipeline / OOP-proc / OOP-dr [OOP-dr is implied in pipeline]



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

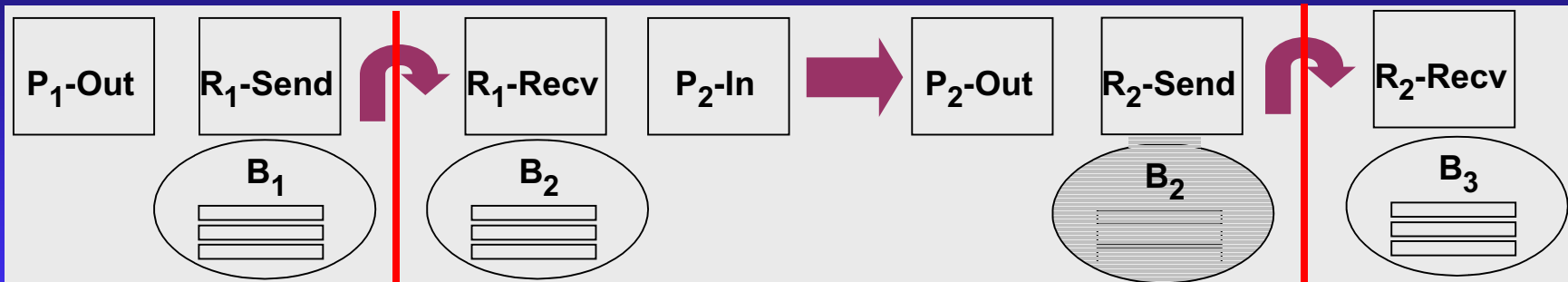
- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

— Represents process set boundary in the pipeline

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# Lib-Mem / Pipeline / IP-proc / OOP-dr [OOP-dr is implied in pipeline]



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

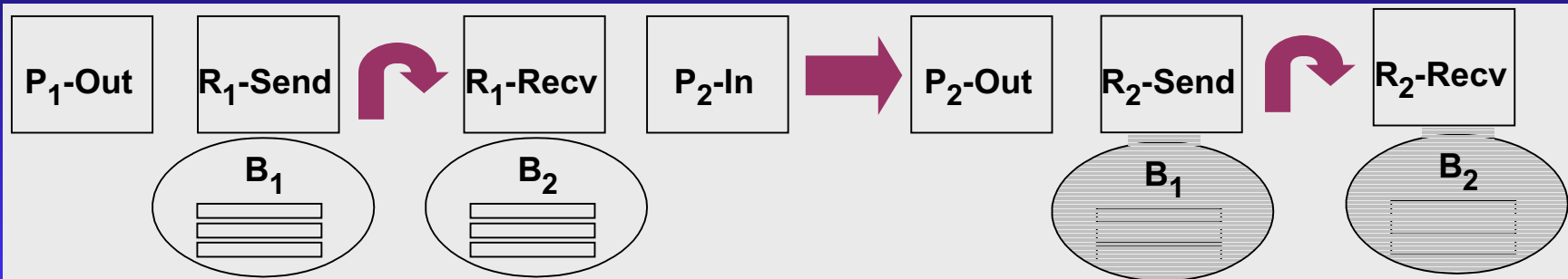
- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

— Represents process set boundary in the pipeline

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# Lib-Mem / Clique / OOP-proc / OOP-dr



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

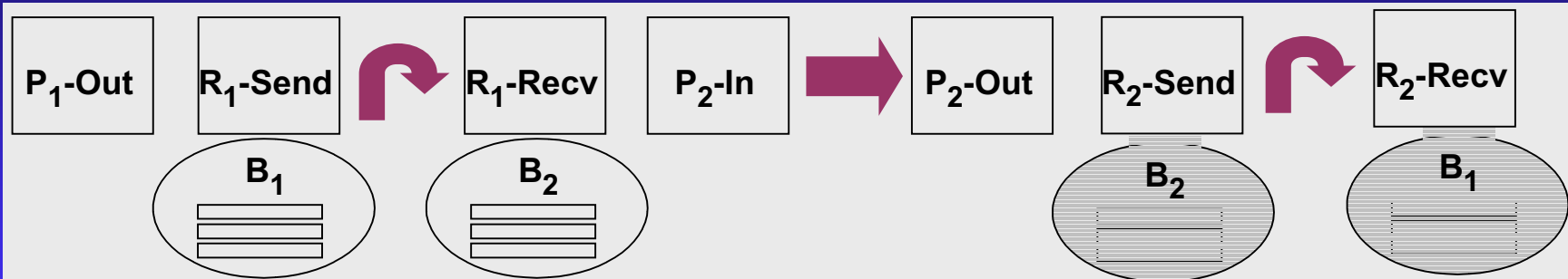
Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# Lib-Mem / Clique / IP-proc / OOP-dr



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

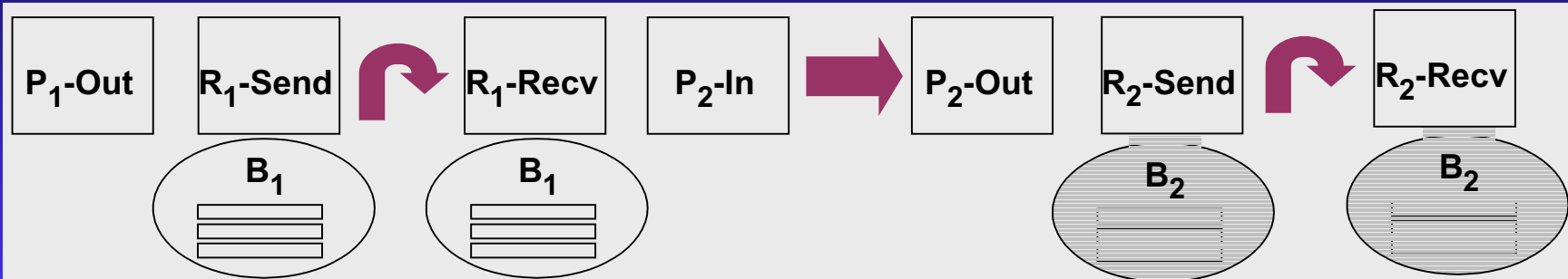
Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# Lib-Mem / Clique / OOP-proc / IP-dr



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

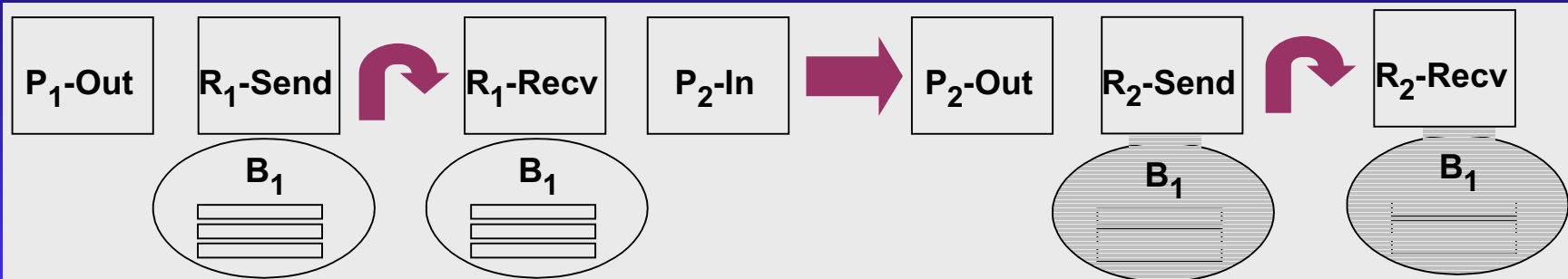
Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# Lib-Mem / Clique / IP-proc / IP-dr



Insert code here for current DRI spec Design. It assumes:

- 0 user specifies the bufferset sharing relationship at setup time
- 0 progress engine / multi-thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Insert here how the user code would look if user had to manually use get/put and a "recycle" function to implement buffer sharing. Assumes:

- 0 user does NOT specify sharing relationship at setup time
- 0 no progress engine / single thread
- 0 blocking + non-waiting puts
- 0 blocking gets

Coding approach using 08/2001 DRI Spec assumptions (multi-thread, automatic sharing)

Potential coding approach in single-threaded environments using manual get/put/recycle

# API Issues and Problems (1/2)

**0 Issue: “putting” a buffer back into a Reorg (current API approach in August 2001) is semantically confusing**

- **Currently, we have**
  - For send reorgs: get (an available buffer for producing), put (to transfer data)
  - For recv reorgs: get (receives data transfer), put (releases app use of buffer)
- **We used to have...**
  - For send reorgs: acquire/insert
  - For recv reorgs: extract/release
- **probably don’t want to go all the way back to that...**
- **could “put” buffer into exposed DRI\_Bufferset type in this case**
- **or... allow “putting” into next DRI\_Reorg, and “recycling”**

**0 Problem: For DRI\_Reorgs created with system memory:**

- **No way to express buffer sharing among DRI\_Reorg objects**
- **No way for implementation to make reasonable decisions on how to share buffers among the DRI\_Reorgs**
- **Letting the “implementation figure it out” probably not be the right way to go**

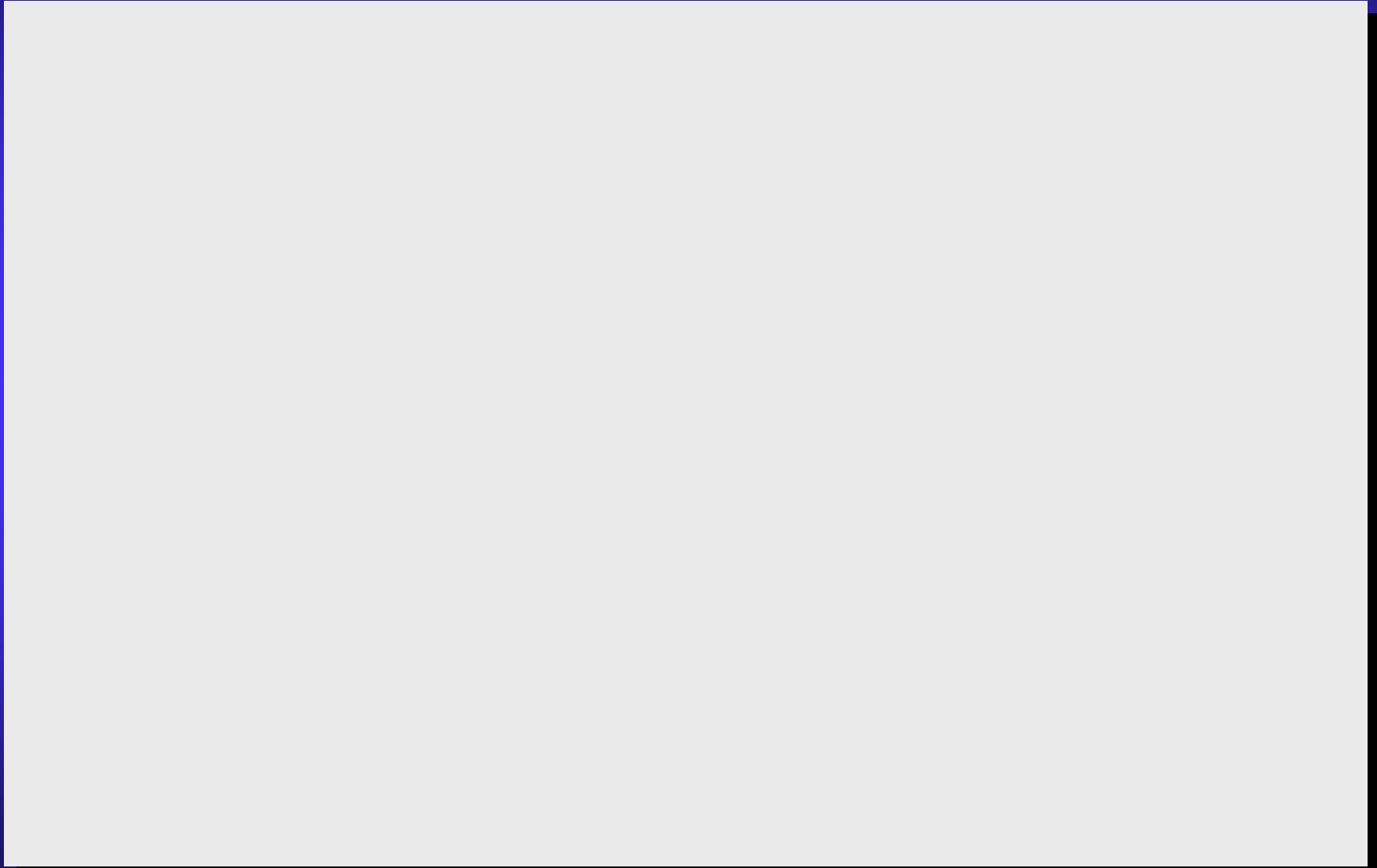
# API Issues and Problems (2/2)

## 0 Issue: Blocking/Non-blocking semantics

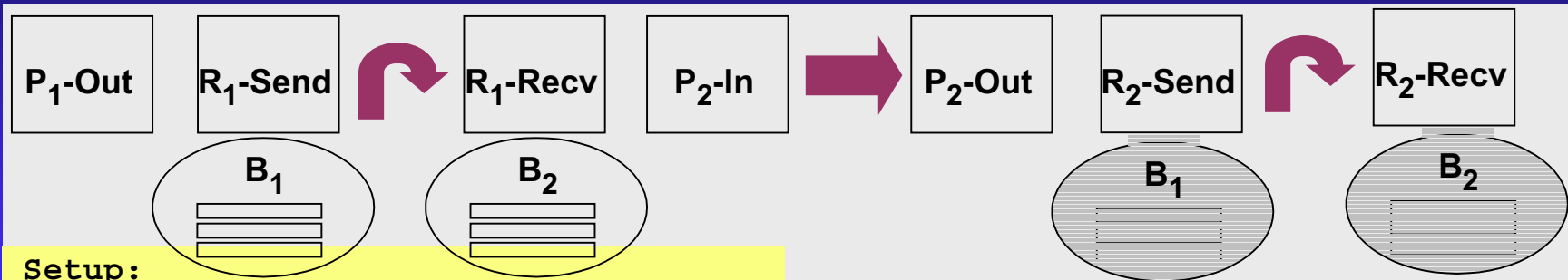
- We claim support for “non-blocking” put, and “blocking” get
- First of all, what do we mean by these terms
  - See below – there are 2 levels of synchronous/asynchronous behavior
- Is there value in supporting other forms?
- Propose the following semantics:
  - Generally:
    - blocking / non-blocking refers to whether synchronization happens
    - Waiting / non-waiting refers to the caller’s disposition wrt an operation
  - Blocking **put**: If receivers’ buffers not yet ready, wait until they are, then send
  - Non-Blocking **put**: Return status indication if receivers’ buffers not yet ready
  - **Waiting put: BLOCK until receivers’ buffers ready, then WAIT until xfer complete**
  - Not-Waiting **put**: BLOCK until receivers’ buffers ready, TRIGGER xfer
  - Blocking **get**: waits until data has arrived from all senders on other side of reorg
  - Non-blocking **get**:
- Given the above proposed semantics, DRI in its current form supports:

 – Blocking and Non-waiting put  
**We shouldn't support this mode (I think)**  
– Blocking get

# Backup Slides



# Lib-Mem / Clique / OOP-proc / OOP-dr



Setup:

<We don't specify who "owns" buffers>

Loop:

```
Reorg_get_buffer(R1-S) ==> B1,0
```

```
<produce data in B1,0>
```

```
B1,0 ==> Reorg_put_buffer(R1-S)
```

```
Reorg_get_buffer(R1-R) ==> B2,0
```

```
Reorg_get_buffer(R2-S) ==> B1,0
```

```
<process OOP B2,0 ==> B1,0>
```

```
B2,0 ==> Reorg_put_buffer(R1-R)
```

```
B1,0 ==> Reorg_put_buffer(R2-S)
```

```
Reorg_get_buffer(R2-R) ==> B2,0
```

```
<consume B2,0 somehow>
```

```
B2,0 ==> Reorg_put_buffer(R2-R)
```

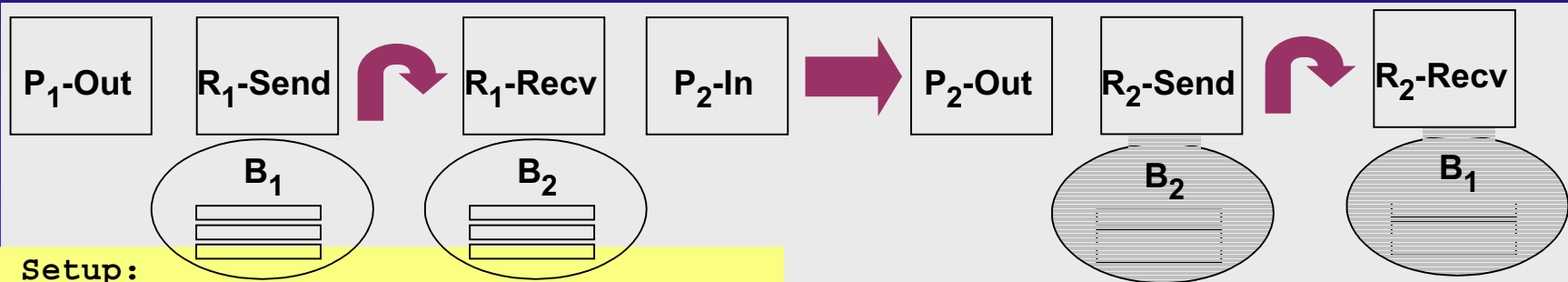
Should we say they start with R1-S?

Either user or lib needs to put B2,0 to R2-R for its next use

Either user or lib needs to put B2,0 back to R1-R for its next use

Coding approach using current DRI Spec (Aug 2001)

# Lib-Mem / Clique / IP-proc / OOP-dr



Setup:

Specify IP proc relationship R1,R2

<We don't specify who "owns" buffers>

Loop:

```
Reorg_get_buffer(R1-S) ==> B1,0
```

```
<produce data in B1,0>
```

```
B1,0 ==> Reorg_put_buffer(R1-S)
```

```
Reorg_get_buffer(R1-R) ==> B2,0
```

```
<process IP B2,0 ==> B2,0>
```

```
B2,0 ==> Reorg_put_buffer(R2-S)
```

```
Reorg_get_buffer(R2-R) ==> B1,0
```

```
<consume B1,0>
```

```
B1,0 ==> Reorg_put_buffer(R2-R)
```

Should we say they start with R1-S?

How will B1,0 be available for R2-R unless user or lib makes it so?

Even so, there is no way to make this happen with explicit put/get call (since buffer access must be in order)  
 User mgmt: "recycle" B1,0 (R1-S => R2-R)  
 lib mgmt: "recycle" behavior is hidden by normal lock/sync implementation

Coding approach using current DRI Spec (Aug 2001)