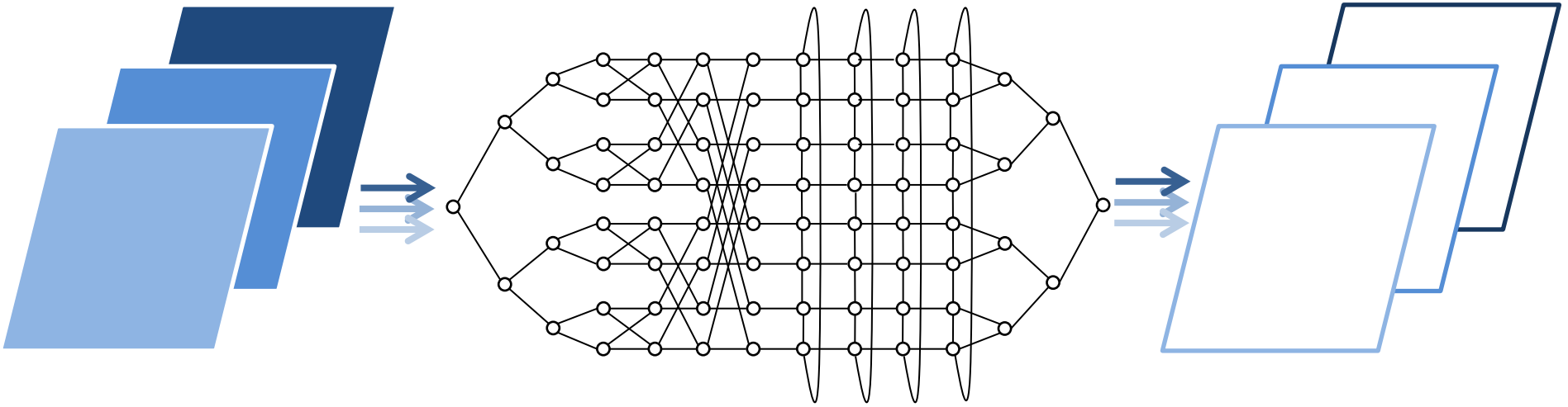
ENSIIE-HPC/BigData

Programmation Parallèle

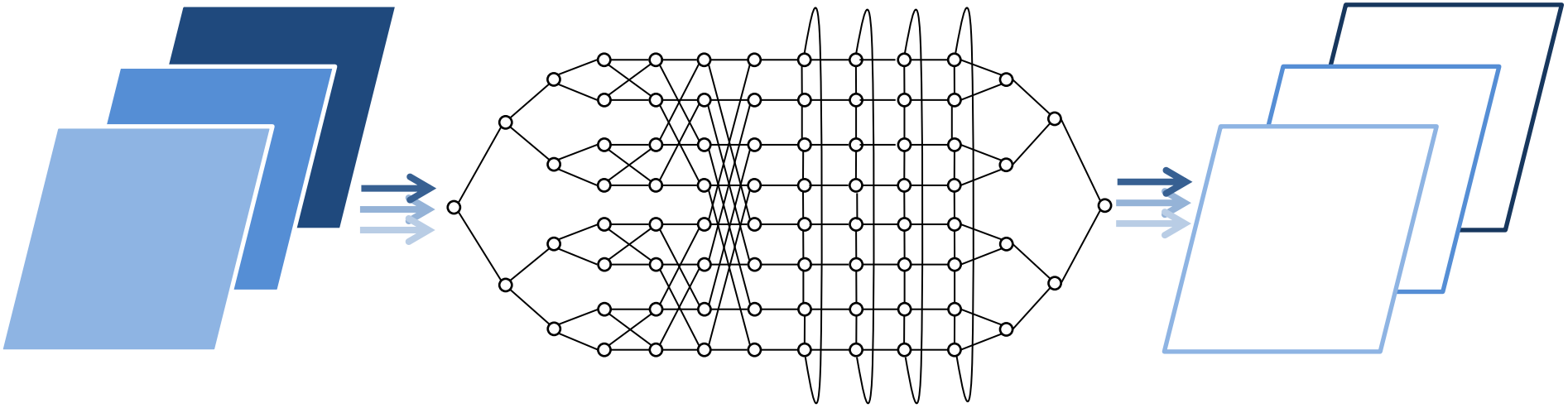
MPI: Message Passing Interface



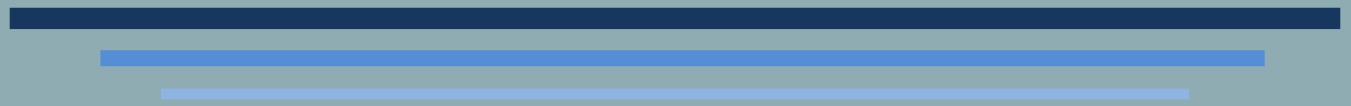
MPI RMA

Remote Memory Access

ENSIIE-HPC/BigData-PP-IPAR-Lecture 6



OVERVIEW OF RDMA


Hardware RMA

What is RDMA ?

- A (relatively) new method for interconnecting platforms in high-speed networks that overcomes many of the difficulties encountered with traditional networks such as TCP/IP over Ethernet.
  - new standards
  - new protocols
  - new hardware interface cards and switches
  - new software

- **R**emote **D**irect **M**emory **A**ccess

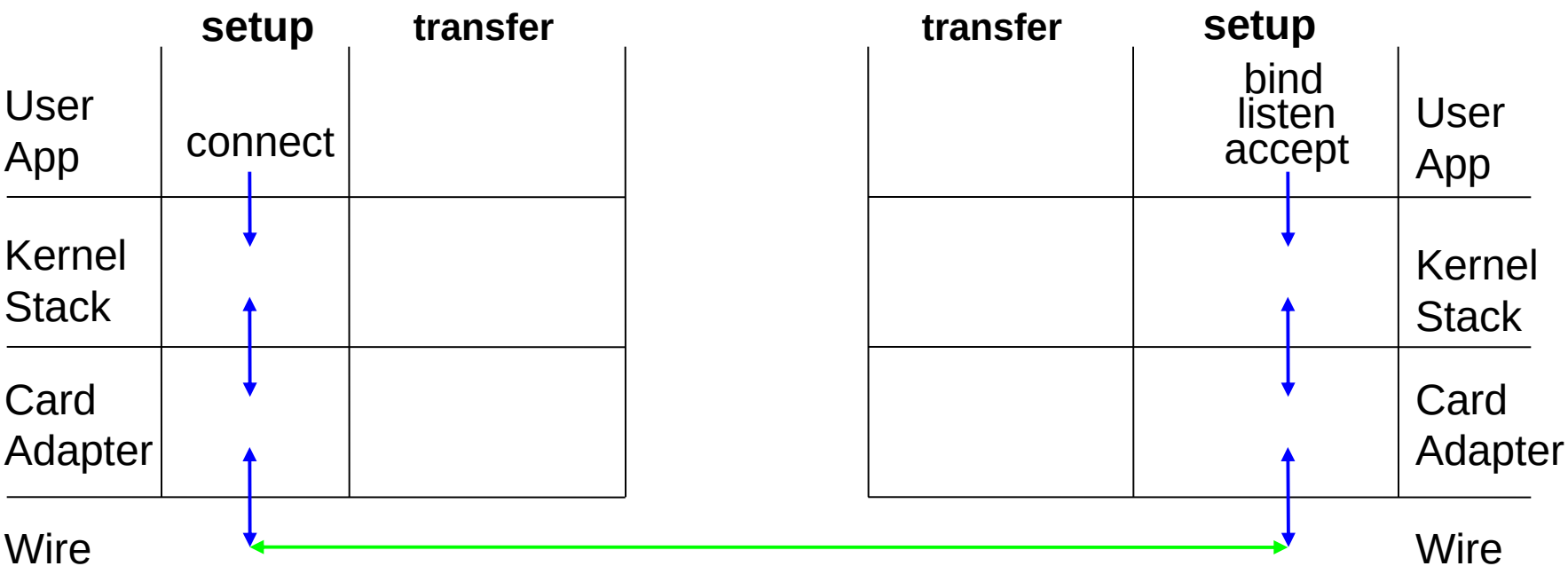
What RDMA stands for?

- Remote
  - data transfers between nodes in a network
- Direct
  - no Operating System Kernel involvement in transfers
  - everything about a transfer offloaded onto Interface Card
- Memory
  - transfers between user space application virtual memory
  - no extra copying or buffering
- Access
  - send, receive, read, write, atomic operations

How RDMA differs from TCP/IP

- “zero copy” – data transferred directly from virtual memory on one node to virtual memory on another node
- “kernel bypass” – no operating system involvement during data transfers
- asynchronous operation – threads not blocked during I/O transfers

Usual Message Transfer

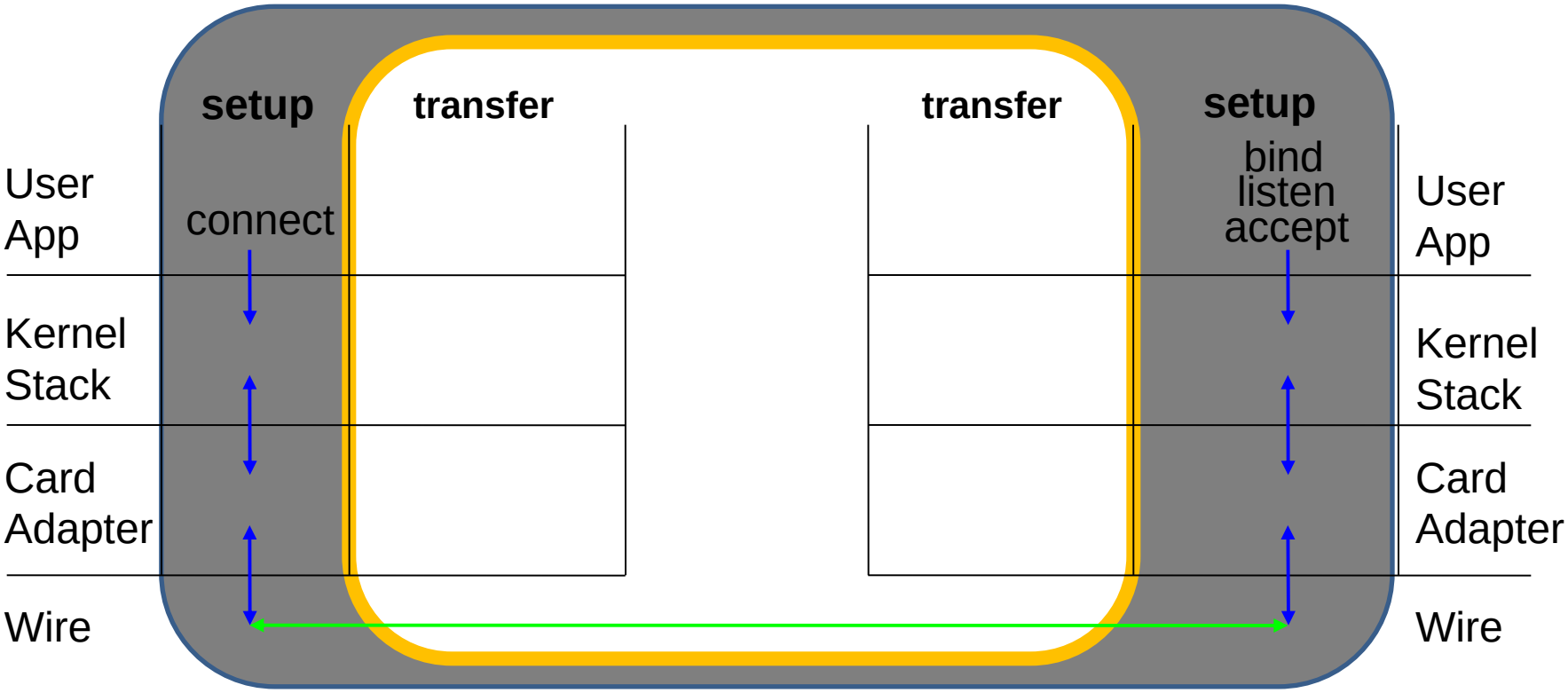


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

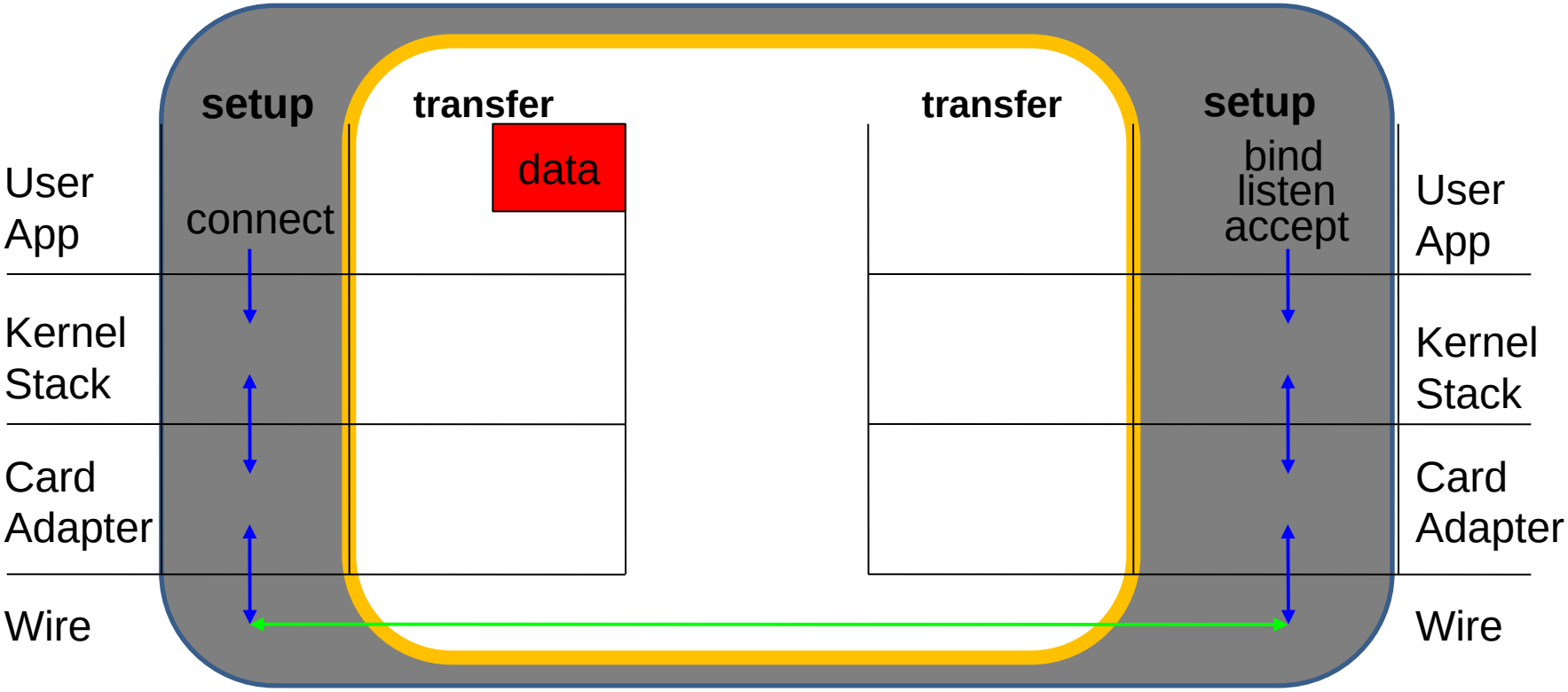


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

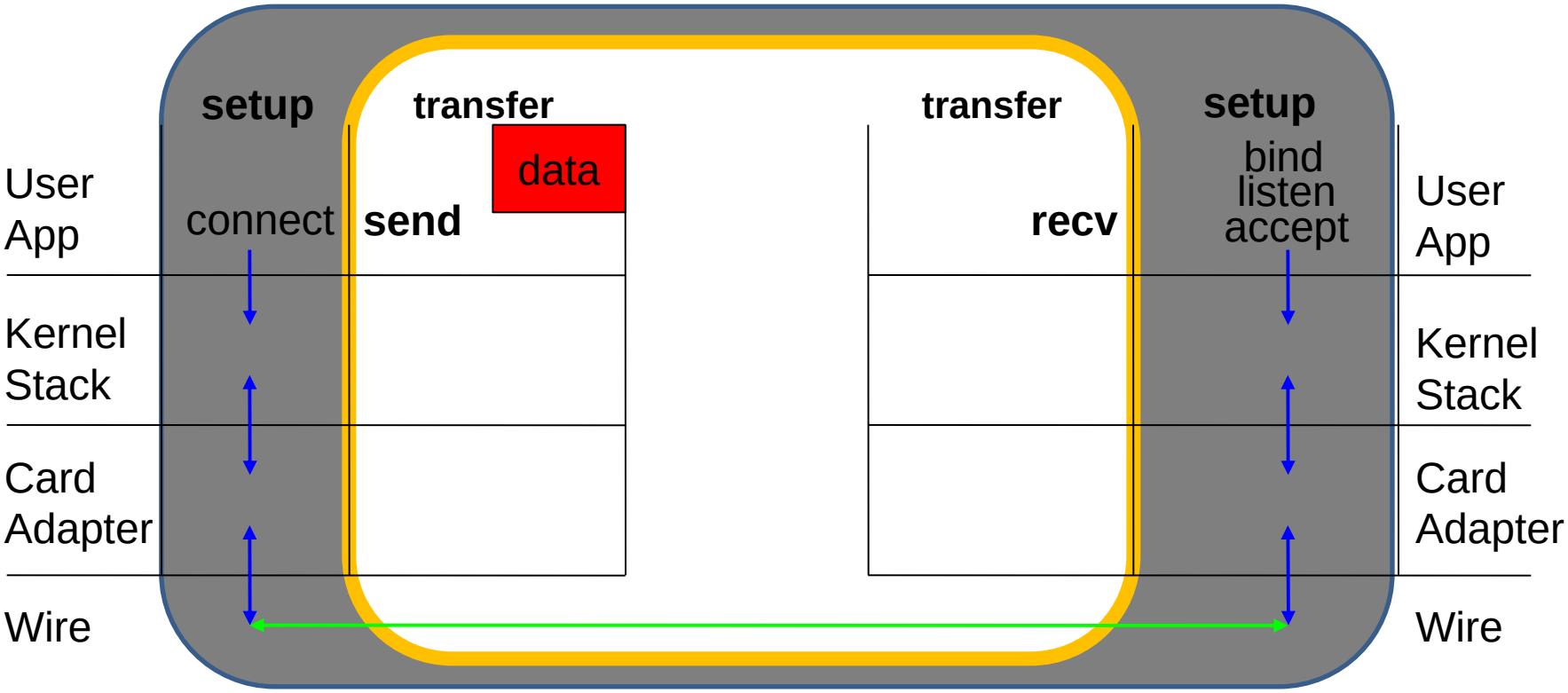


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

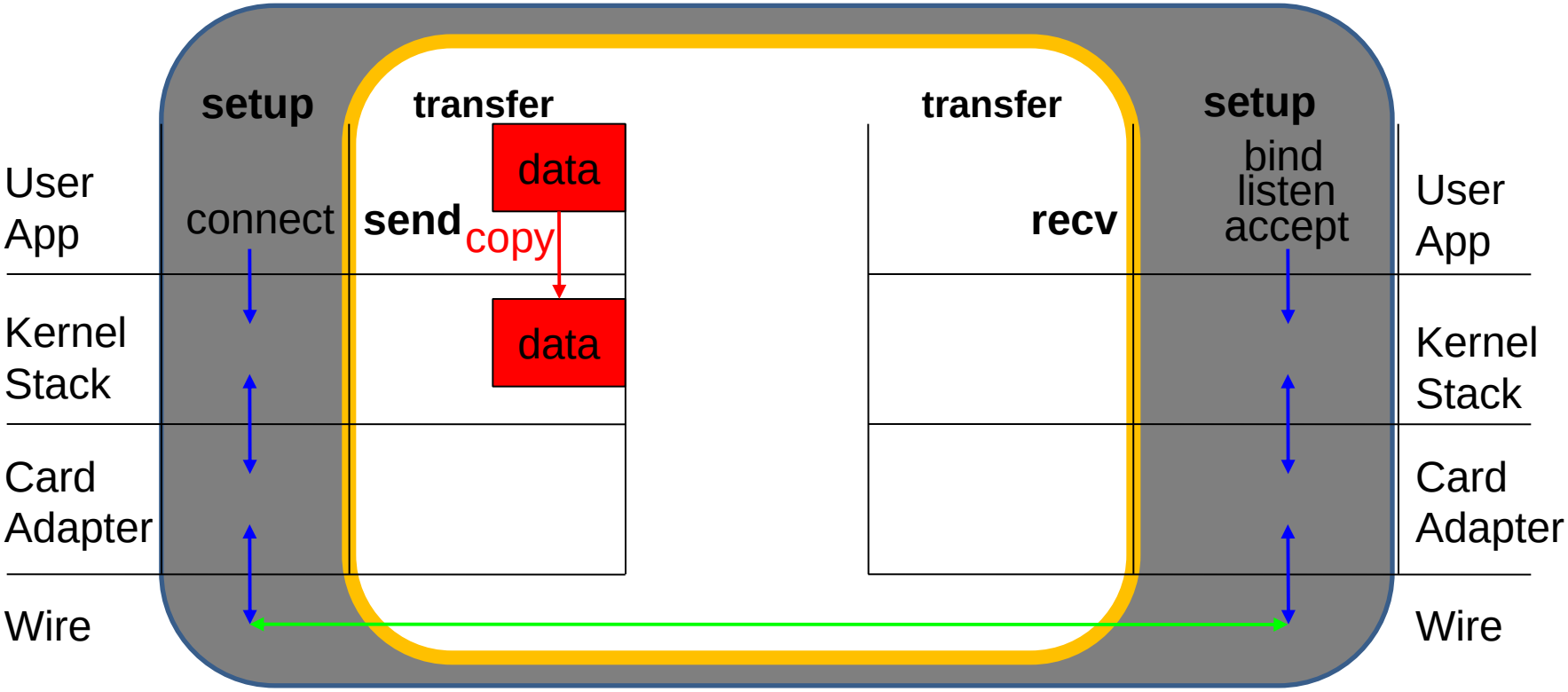


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

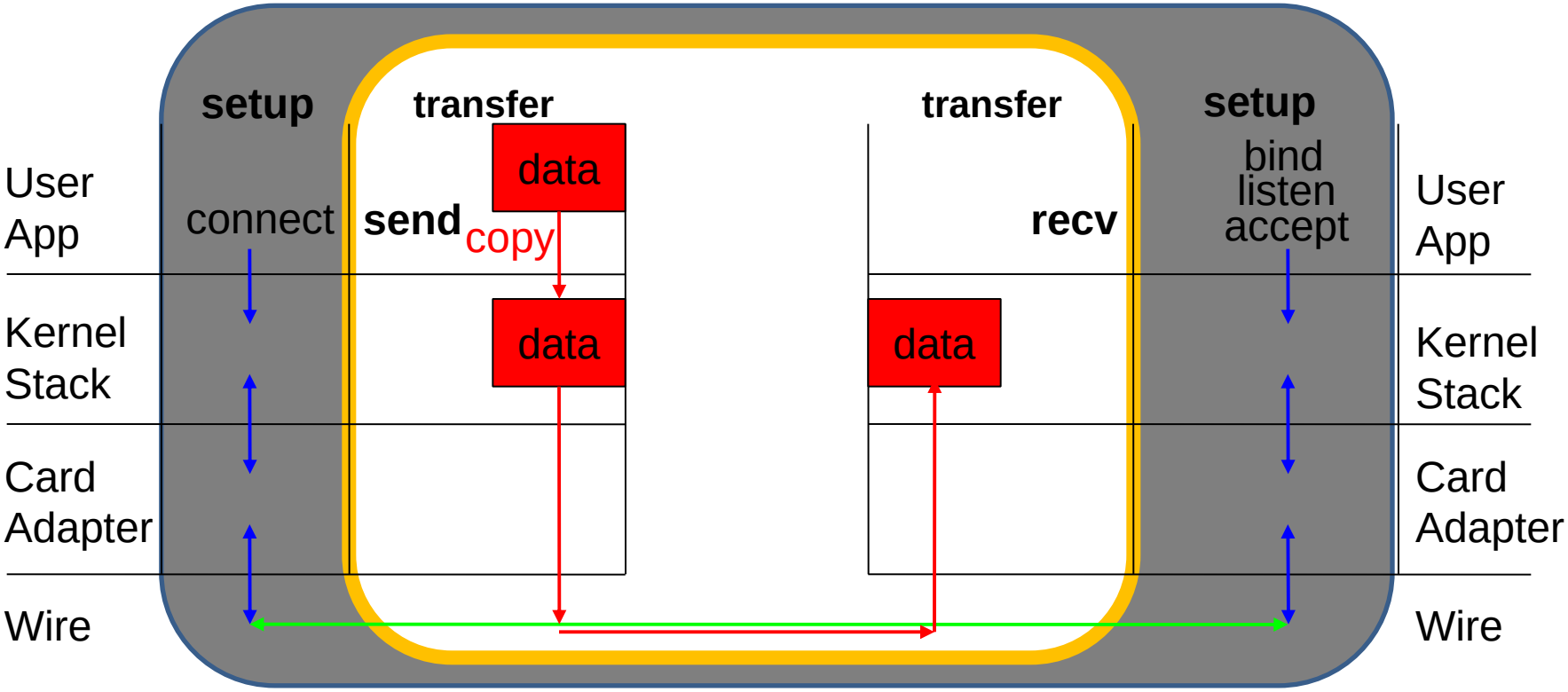


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

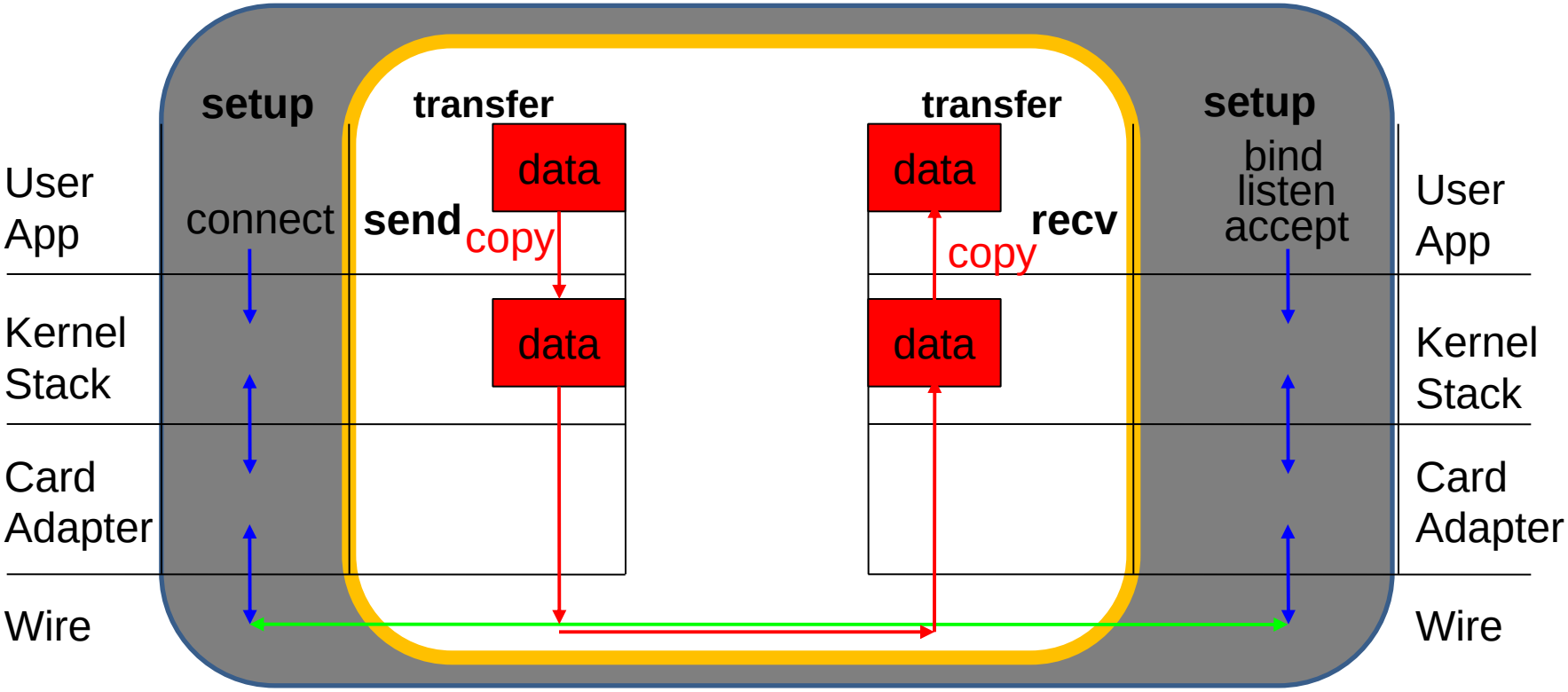


blue lines: control information

red lines: user data

green lines: control and data

Usual Message Transfer

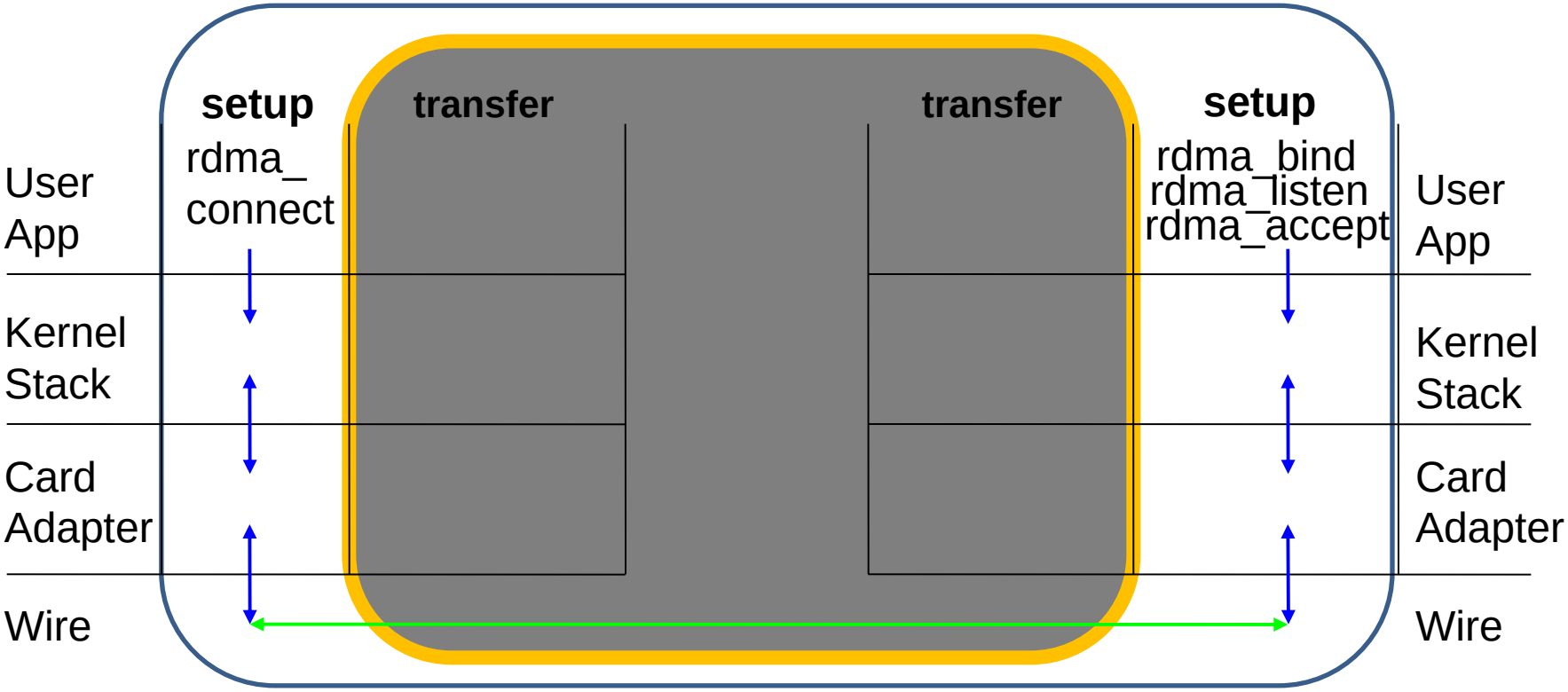


blue lines: control information

red lines: user data

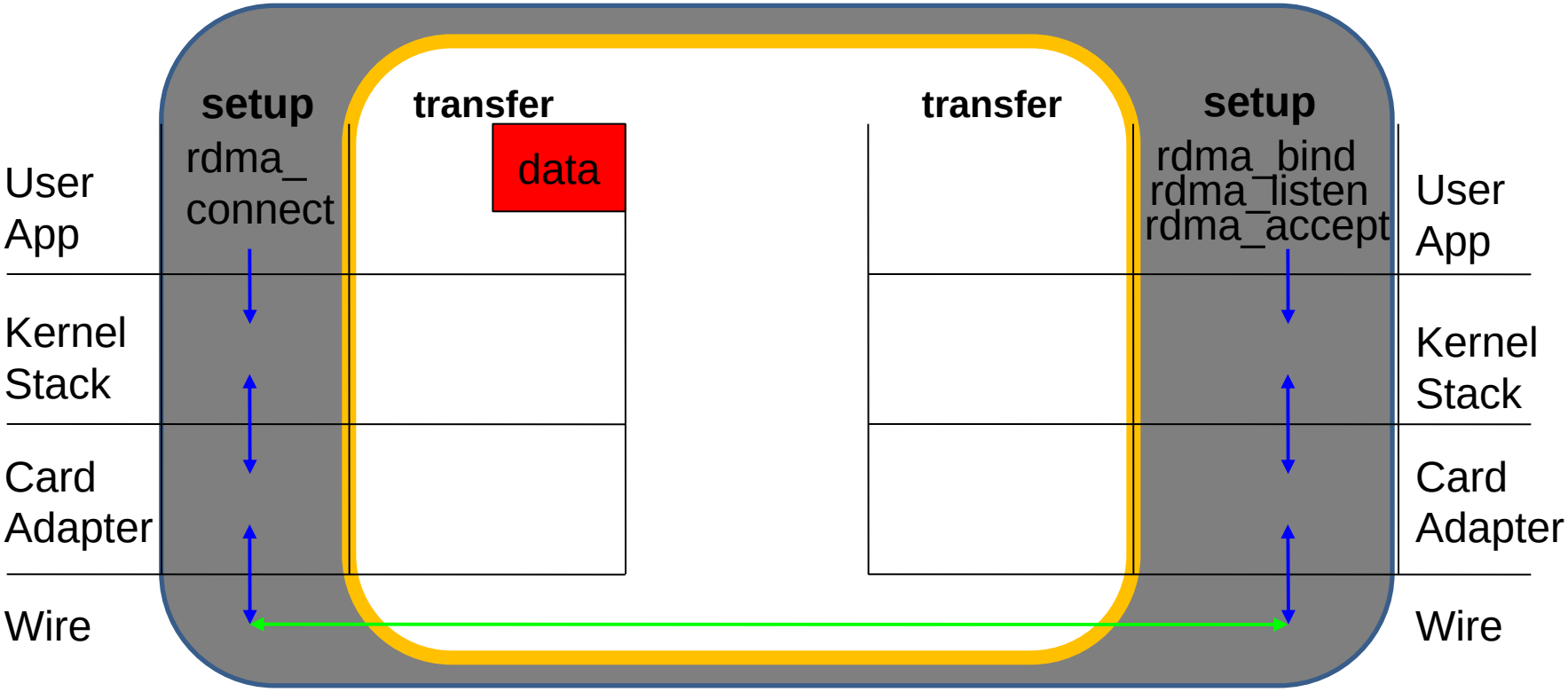
green lines: control and data

RDMA Transfer

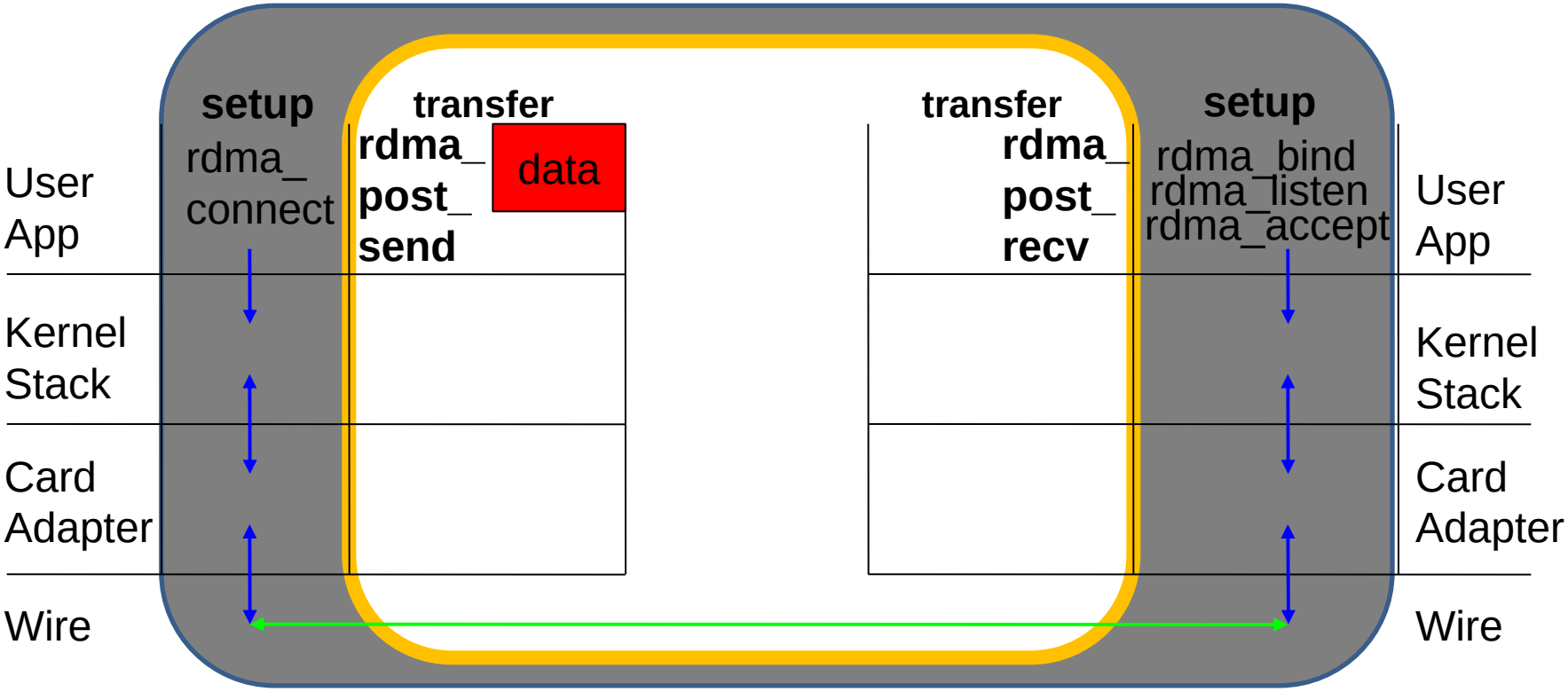


- Setup phase is the same

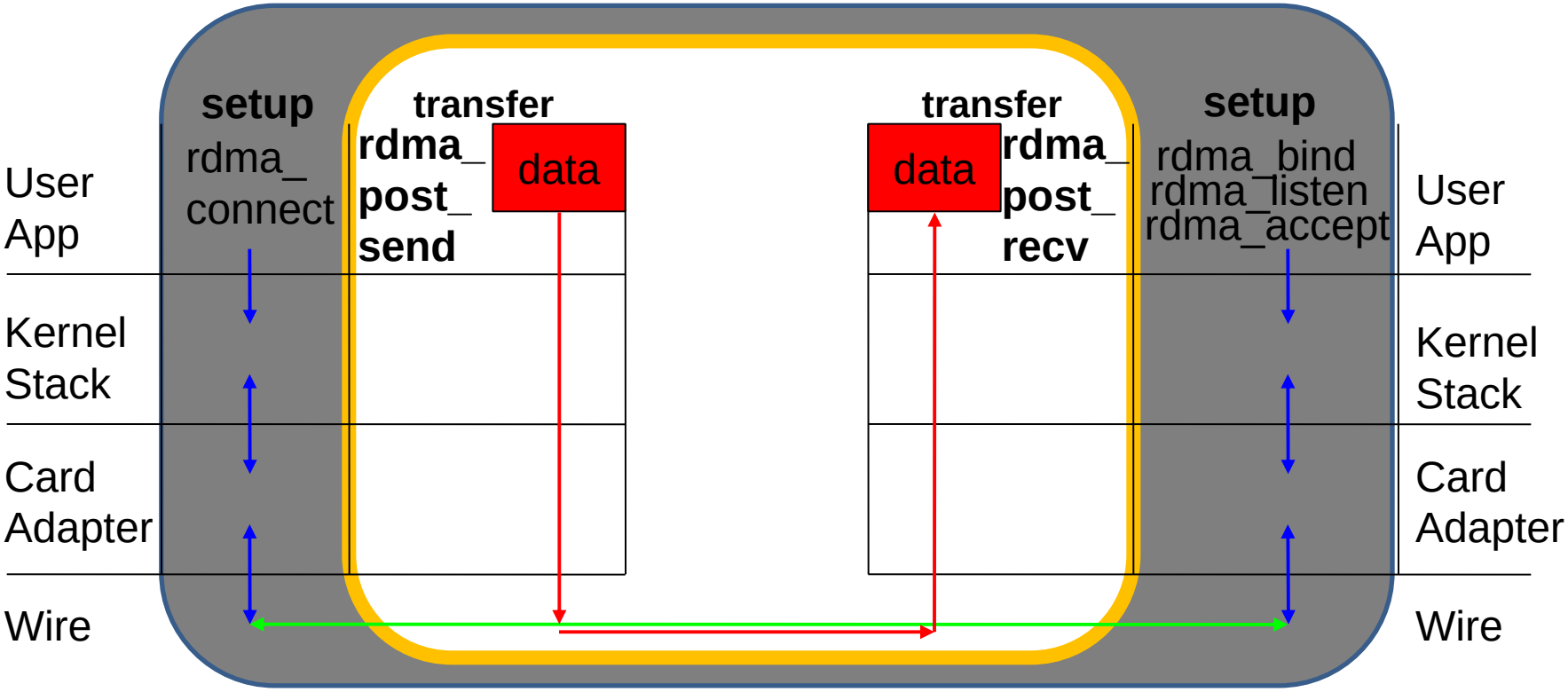
RDMA Transfer



RDMA Transfer

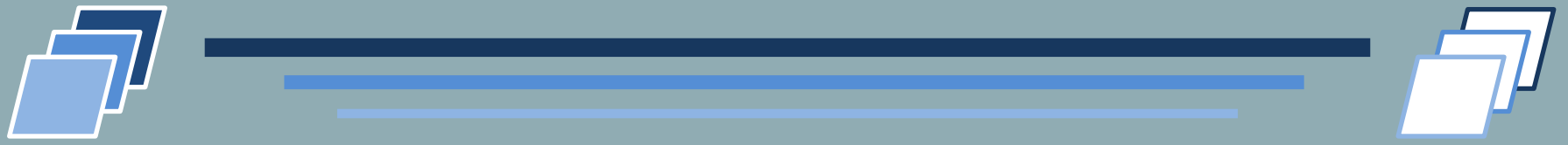


RDMA Transfer



- Write directly in user-space memory
- No need for copy in kernel space

RMA PROGRAMMING MODELS



One-sided communication model

- One-sided communication models were developed to take advantage of RDMA
  - Programming model directly maps network behavior
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
- Each communicating entities exposes a part of its memory to other entities
- These other entities can directly read from or write to this memory

Partitionned Global Address Space

- PGAS are programming languages proposed as an alternatives to message passing
- The basic idea is to consider the memory as a shared address space partitionned between all communicating entities
  - Each entity as its own address range
  - Every entity can directly access the other entities memory
- Multiple PGAS languages exist
  - UPC, XMP, GASPI…

Partitionned Global Address Space

- PGAS fits well with the idea of RMA
- Accessing a distant memory should only involve the communicating entity doing the access
  - This access should not impact the owner of the address
- Every communicating entity can access any address of the Global Address Space
  - In this case, the whole memory is exposed to be accessed by distant entities, not only a subpart

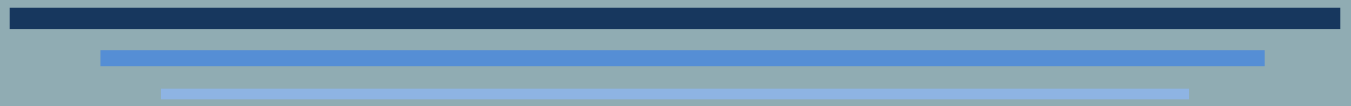
MPI RMA

- As we have seen so far, MPI communications mainly rely on send/recv message passing routines
  - Fits well with “Usual Message Transfer” shown in RDMA part
- Most communications in MPI are synchronizing
  - All ranks involved in a communication must be ready
  - Non-blocking routines try to alleviate this burden
    - But original communication scheme is still synchronizing
  - Synchronization hurts performance
- RMA offers “real” asynchronous communication
  - Remote Memory Access: MPI one-sided comm. model

MPI RMA

- Realizing synchronous-based message passing on RDMA enabled networks is suboptimal
  - The software arbitrarily inserts synchronization on a network free of these synchronizaton
- MPI RMA appeared in MPI 2 and has been refactored in MPI 3 to better support RDMA enabled network
- It is also possible to implement an RMA runtime on top of non RDMA enabled network
  - All synchronizing parts will happen in the runtime and remain hidden to the user

MPI RMA WINDOWS



MPI Windows

- One-sided communication are based on part of the memory exposed to and accessed by distant entities
- To enable one-sided communication, MPI ranks need to expose some part of their memory to other ranks
  - Then every ranks can directly read from or write to the exposed memory
- The exposed memory in MPI is called a **Window**.
- Multiple routines exist to create windows

Using existing buffer as a Window

- *MPI_Win_create*(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win);
  - IN base initial address of window
  - IN size size of window in bytes
  - IN disp_unit local unit size for displacements, in bytes
  - IN info info argument (handle)
  - IN comm intra-communicator (handle)
  - OUT win window object returned by the call (handle)
- Collective operation
- All processes in the communicator can use the exposed buffer to do RMA
  - Through the Window handle
- The disp_unit argument is provided to facilitate address arithmetic in RMA operations
  - the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

Creating a buffer for a Window

- *MPI_Win_allocate*(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);
  - IN size — size of window in bytes
  - IN disp_unit — local unit size for displacements, in bytes
  - IN info — info argument (handle)
  - IN comm — intra-communicator (handle)
  - OUT baseptr — initial address of window
  - OUT win — window object returned by the call (handle)
- Collective operation
- On all processes in the communicator allocates at least *size* bytes of memory
- The function returns a window associated with the exposed memory
- The function also returns a pointer for the calling rank to access and/or modify created buffer

Creating a shared buffer for a Window

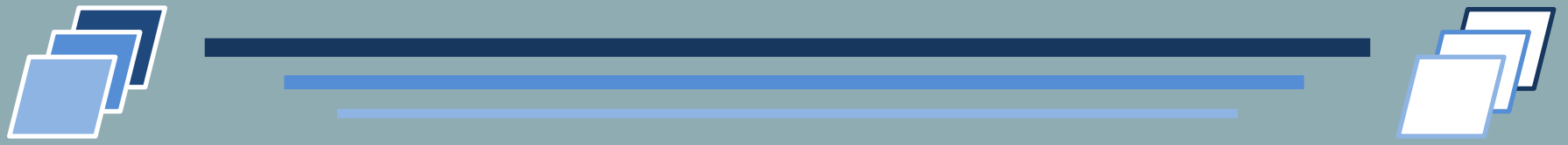
- *MPI_Win_allocate_shared*(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);
  - IN size — size of window in bytes
  - IN disp_unit — local unit size for displacements, in bytes
  - IN info — info argument (handle)
  - IN comm — intra-communicator (handle)
  - OUT baseptr — initial address of window
  - OUT win — window object returned by the call (handle)
- Collective operation
- Difference with MPI_Win_allocate: the memory segment allocated is shared among all processes in the communicator
  - The other processes can load/store directly in this memory
  - A call to MPI_Win_shared_query will returns the beginning address in the shared buffer assorciated with a specific (specified) rank
- It is the user responsibility to ensure that the processes in comm can allocate a shared memory buffer

Creating a Window with no buffer yet

- *MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win);*
  - IN info info argument (handle)
  - IN comm intra-communicator (handle)
  - OUT win window object returned by the call (handle)

- Collective operation
- Creates a window handle with no associated buffer (yet)
- The user need to call another function to attach a buffer to the local rank window handle
  - MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
    - Attach the buffer pointed to by *base* to the existing window
  - MPI_Win_detach(MPI_Win win, const void *base)
    - Detach the buffer pointed to by *base* from the existing window
    - The *win* and *base* arguments should match those of a previous MPI_Win_attach call

ONE-SIDED COMMUNICATIONS



One-sided communications

- One-sided communications are based on two main primitives:
  - As point-to-point and collective communications are based on send and recv
- Two main actions are required to access a remote memory address
  - One process may either read or write a specific data
- One crucial action is also necessary for performance: perform arithmetic operations on distant data
  - Avoid to read, compute then write
- More complex primitives are also available in MPI

Reading a distant data: Get

```
int MPI_Get (
  void *origin_addr(out),
  int origin_count(in),
  MPI_Datatype origin_type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  int target_count(in),
  MPI_Datatype target_type(in),
  MPI_Win win(in),
);
```

Arguments corresponding to reading process

Which process to read data from in the window

Arguments corresponding to the process being read

Reading a distant data: Get

- Transfer occurs as if the target process had issued an MPI_send with the target parameters and the original process had issued an MPI_recv with the original parameter
  - Hence, the target and original parameters follow the same restrictions as if they were arguments of MPI_send and MPI_recv functions respectively

- The starting address for the target buffer is window_base[target_rank]+target_disp x disp_unit[target_rank]
  - disp_unit is specified by the target rank at windows creation

Writing a distant data: Put

```
int MPI_Put (
  void *origin_addr(in),
  int origin_count(in),
  MPI_Datatype origin_type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  int target_count(in),
  MPI_Datatype target_type(in),
  MPI_Win win(in),
);
```

Arguments corresponding to reading process

Which process to write data to in the window

Arguments corresponding to the process being read

Writing a distant data: Put

- Transfer occurs as if the original process had issued an MPI_send with the original parameters and the target process had issued an MPI_recv with the target parameter
  - Hence, the original and target parameters follow the same restrictions as if they were arguments of MPI_send and MPI_recv functions respectively

- The starting address for the target buffer is window_base[target_rank]+target_disp x disp_unit[target_rank]
  - disp_unit is specified by the target rank at windows creation

Reduce local and distant buffers: Accumulate

```
int MPI_Accumulate (
  void *origin_addr(in),
  int origin_count(in),
  MPI_Datatype origin_type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  int target_count(in),
  MPI_Datatype target_type(in),
  MPI_Op op(in),
  MPI_Win win(in),
);
```

Arguments corresponding to reading process

Which process to write data to in the window

Arguments corresponding to the process being read

Which reduction operation to apply between original and target buffer

Reduce local and distant buffers: Accumulate

- Apply the reduction operation between original buffer and target buffer
  - For example, if *op* is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target buffer
- The reduction result is stored in the target buffer
- Only predefined datatypes or derived datatypes from predefined datatypes
  - Each datatype argument must relate to the same predefined datatype
  - The op operation applies on elements of this predefined datatypes
- Only predefined operations are allowed
  - No user-defined operations
- New predefined operations for RMA: MPI_Replace

Read and reduce: Get_accumulate

```
int MPI_Get_accumulate (
  void *origin_addr(in),
  int origin_count(in),
  MPI_Datatype origin_type(in),
  void *result_addr(out),
  int result_count(in),
  MPI_Datatype result_type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  int target_count(in),
  MPI_Datatype target_type(in),
  MPI_Op op(in),
  MPI_Win win(in),
);
```

Arguments corresponding to the buffer of reading process for the reduce

Arguments corresponding to the receiving buffer of the reading process

Which process to write data to in the window

Arguments corresponding to the process being read

Which reduction operation to apply between original and target buffer

Read and reduce: Get_accumulate

- Combine the behavior of a Get and an Accumulate
  - The same restrictions apply as for MPI_Get and MPI_Accumulate functions
- The original process first read the data in the target buffer
- Then an accumulate is perform between the original buffer and the target buffer
- The data read in the Get part of the function is stored in the “result” buffer of the original process

Read and reduce: Fetch_and_op

- Some processors have hardware design with special operations such as fetch_and_add or fetch_and_increment
- The generic nature of Get_accumulate may prevent the implementation to efficiently use these hardware support
  - Operations supported in hardware are often much more efficient than their software equivalent implemented on top of usual hardware operations
- On processor, operations apply on registers
  - Hence on only one element, or a very limited number of elements
- MPI provides function Fetch_and_op to do the same as Get_accumulate but on only one element in the buffers
  - May help MPI implementations to directly target these hardware operations

Read and reduce: Get_accumulate

```
int MPI_Get_accumulate (
  void *origin_addr(in),
  int origin_count(in),
  MPI_Datatype origin_type(in),
  void *result_addr(out),
  int result_count(in),
  MPI_Datatype result_type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  int target_count(in),
  MPI_Datatype target_type(in),
  MPI_Op op(in),
  MPI_Win win(in),
);
```

Arguments corresponding to the buffer of reading process for the reduce

Arguments corresponding to the receiving buffer of the reading process

Which process to write data to in the window

Arguments corresponding to the process being read

Which reduction operation to apply between original and target buffer

Read and reduce: Fetch_and_op

```
int MPI_Get_accumulate (
  void *origin_addr(in),


  void *result_addr(out),

  MPI_Datatype type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),


  MPI_Op op(in),
  MPI_Win win(in),
);
```

Datatypes for origin, result and target buffers

Which process to write data to in the window

Which reduction operation to apply between original and target buffer

Compare and swap

```
int MPI_Compare_and_swap (
  void *origin_addr(in),
  void *compare_addr(int),
  void *result_addr(out),
  MPI_Datatype type(in),
  int target_rank(in),
  MPI_Aint target_disp(in),
  MPI_Win win(in),
);
```

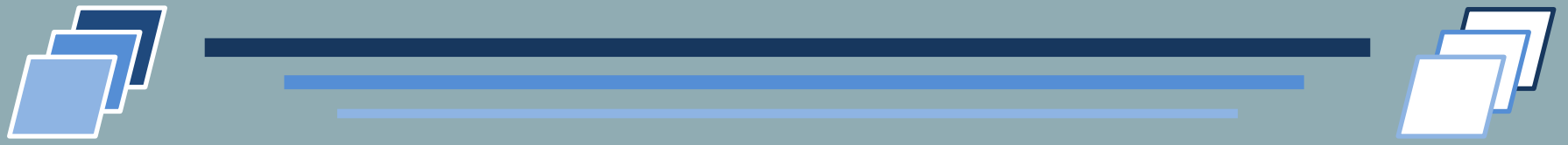
- origin_addr(in) → Data to copy to target if compare is ok
- compare_addr(int) → Data to compare to target buffer
- result_addr(out) → Buffer to receive data from target if compare is ok
- type(in) → Datatypes for origin, compare, result and target buffers
- target_rank(in) → Which process to compare and write data to in the window

Compare and swap

- Like Fetch_and_op, this function only operates on one element

- Compare the element pointed by *compare_addr* to the element specified in the target buffer
- If the two compared data are the same
- Then the original buffer reads the target data and store it in the result buffer…
- … and the target data is replaced by the original data pointed by *original_addr*

NECESSARY SYNCHRONIZATION



Why synchronizations?

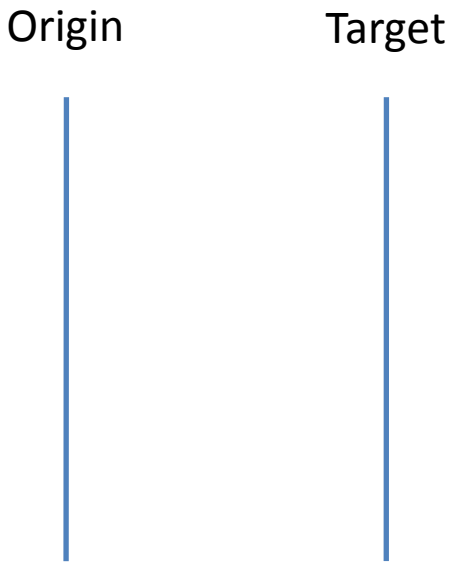
- As we already discussed in PP-C5 on MPI-IO, it is not a good idea for several entities to write the same data at the same time
- Synchronization are then necessary to target process to access its data while an origin process is accessing it
- MPI RMA defines “epochs” for that purpose
  - Exposure epoch: amount of time when target window is accessible by other processes
  - Access epoch: amount of time when RMA operations can be issued

Why synchronizations?

- The access epoch of the origin process matches the exposure epoch of the target process
- Epochs at a process on the same window must be disjoint.
- If multiple processes targets the same window in the same exposure epoch, data access are sequentialized and done in order
- There are two ways for creating epochs:
  - Active: the target takes part in the epochs creation
  - Passive: only origin handles the epochs creation

P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation

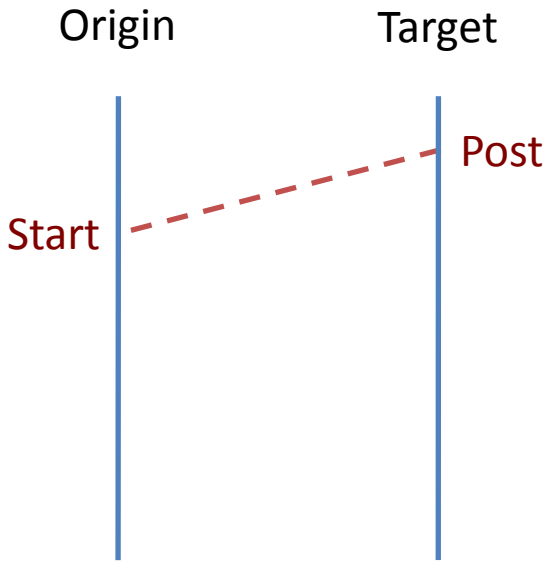


P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation
  - Target process calls MPI_Win_Post to open its exposure epoch

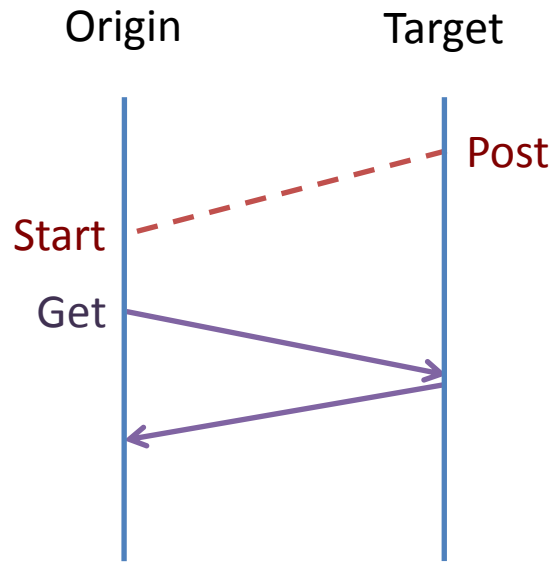
P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Start to open its access epoch



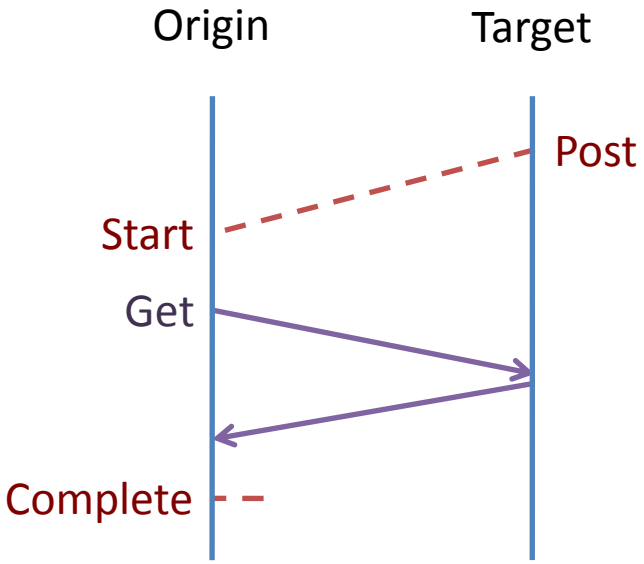
P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process can issue RMA calls



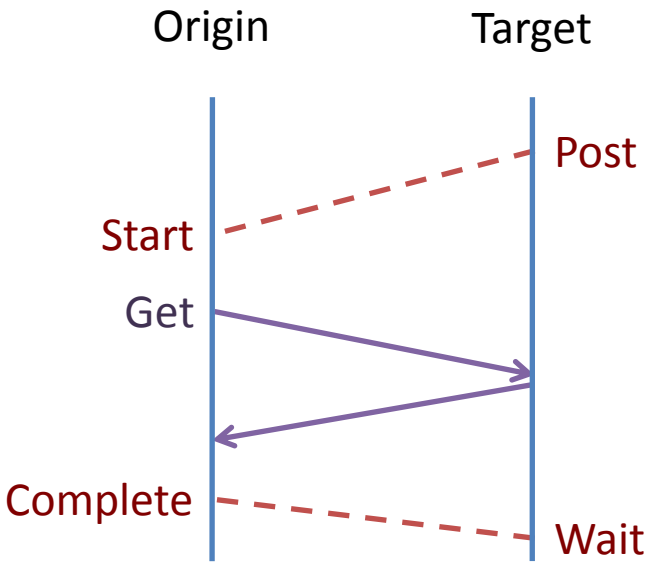
P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process can issue RMA calls
  - Origin process calls MPI_Win_Complete to close its access epoch



P2P active sync: PSCW

- PSCW stands for Post-Complete-Start-Wait
- The origin and target processes are both involved in epochs creation
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process can issue RMA calls
  - Origin process calls MPI_Win_Complete to close its access epoch
  - Target process calls MPI_Win_Wait to close its exposure epoch

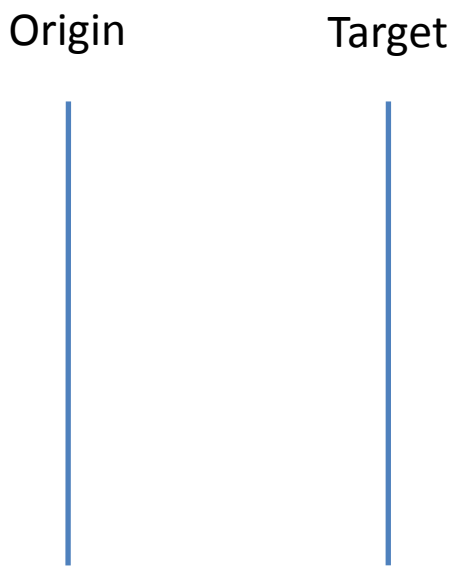


P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done

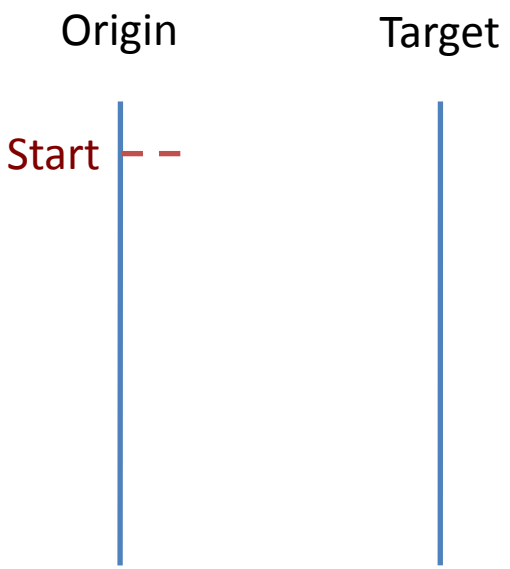
P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done



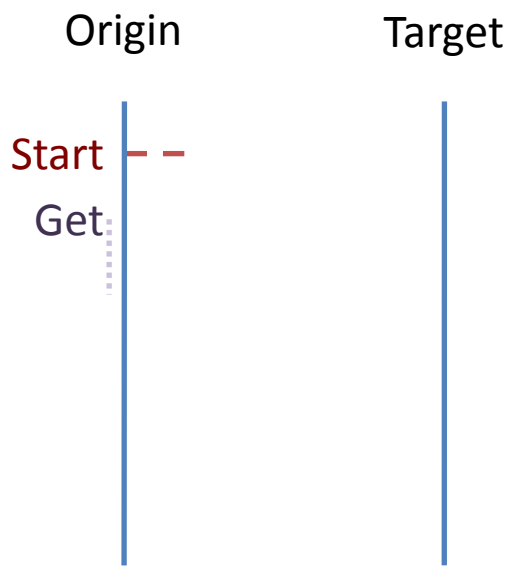
P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done
  - Origin process calls MPI_Win_Start to open its access epoch



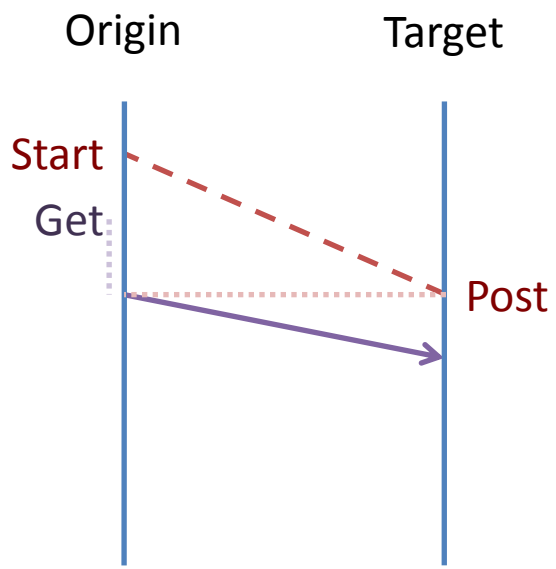
P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process issues a Get



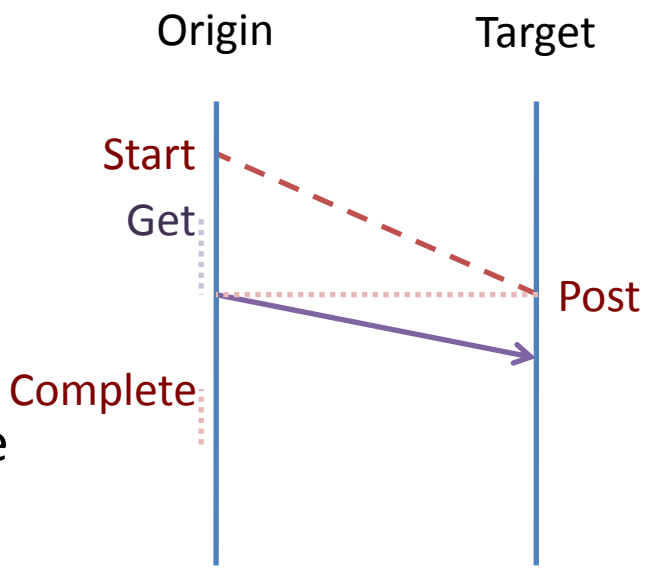
P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process issues a Get
  - Target process calls MPI_Win_Post to open its exposure epoch



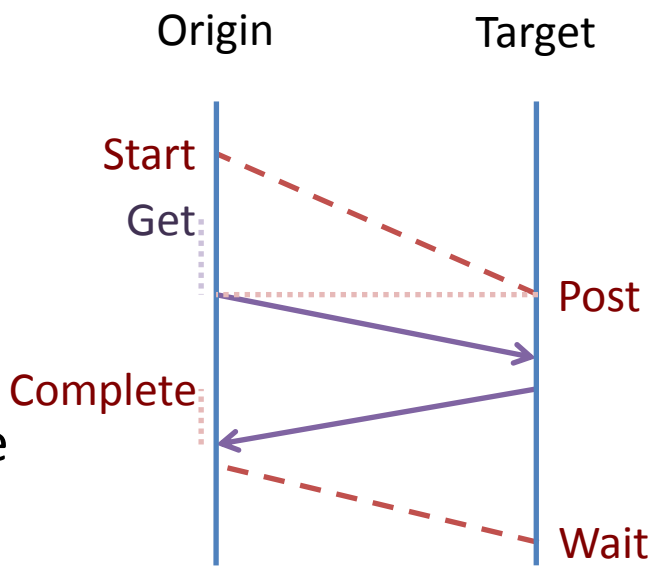
P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process issues a Get
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Complete to close access epoch before Get is done



P2P active sync: PSCW

- Start can be issued before Post
- Start not blocking, but first RMA call blocks until Post is called
- Wait blocks until all matching MPI_Win_Complete are called
- Complete blocks until all RMA are done
  - Origin process calls MPI_Win_Start to open its access epoch
  - Origin process issues a Get
  - Target process calls MPI_Win_Post to open its exposure epoch
  - Origin process calls MPI_Win_Complete to close access epoch before Get is done
  - Target process calls MPI_Win_Wait to close its exposure epoch



Collective active synchronization

- *MPI_Win_fence*(int assert, MPI_Win win);
  - IN assert program assertion (integer)
  - IN win window object (handle)

- Collective operation
- All processes associated to the windows must call MPI_Win_fence
- The first call to MPI_Win_fence opens exposure and access epochs for all processes
- A second call to MPI_Win_fence closes exposure and access epochs for all processes

P2P passive synchronization

- *MPI_Win_lock*(int lock_type, int rank, int assert, MPI_Win win);
  - IN lock_type — either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
  - IN rank — rank of locked window
  - IN assert — program assertion
  - IN win — window object
- Called from the origin process
- Open an exposure epoch at target rank and an access epoch at the calling rank
- MPI_Win_lock is just beginning of epochs
- If *lock_type* is MPI_LOCK_EXCLUSIVE, then behaves “like” a lock, and no other process can access concurrently to the window
- If *lock_type* is MPI_LOCK_SHARED, other processes also using MPI_LOCK_SHARED can concurrently access to the window
- Concurrent epochs to the same process are not allowed

P2P passive synchronization

- *MPI_Win_unlock*(int rank, MPI_Win win);
  - IN rank rank of locked window
  - IN win window object

- Called from the origin process
- Close the exposure epoch at target process and access epoch at origin process opened with a call to MPI_Win_lock
- MPI_Win_unlock is just closure of epochs
- All RMA operations will be done both at origin and at target when the call returns

Global passive synchronization

- *MPI_Win_lock_all*(int rank, MPI_Win win);
  - IN assert program assertion
  - IN win window object
- Called from the origin process
- Open an exposure epoch at all processes in the window, with MPI_LOCK_SHARED
- It is **NOT** a collective operation!
  - The all refers to all the processes in the windows

- *MPI_Win_unlock_all*(int rank, MPI_Win win);
  - IN win window object
- Called from the origin process
- Closes all epochs opened with MPI_Win_lock_all
  - As MPI_Win_unlock closes epochs created by MPI_Win_lock

Flushing operations in passive epochs

- *MPI_Win_flush*(int rank, MPI_Win win);
  - IN rank rank of locked window
  - IN win window object
- Called from the origin process
- Completes all outstanding RMA operations from origin process to target rank
- RMA operations are completed both at origin and at target

- *MPI_Win_flush_all*(MPI_Win win);
  - IN rank rank of locked window
  - IN win window object
- Called from the origin process
- Completes all outstanding operations to any target in the window
- RMA operations are completed both at origin and at target

MPI RMA and RMA models

- MPI RMA performances depends on RMA implementation
  - Is it emulated on p2p message passing?
  - Is it based on hardware features/instructions?
  - Is it based on RDMA enabled network?
- MPI RMA proposal has to deal with the burden of other MPI parts
  - Especially with the first-coming p2p and collective comm. models
- What about other RMA models such as PGAS?
  - Also depends on implementation
  - Some PGAS implementations still relies on MPI for communications between nodes
    - Once again, is it emulated on top of p2p communications or MPI RMA communications
- Programming models created for One-sided communications and with all the right supports (hardware, …) may provide better performance than MPI RMA or models implemented using MPI