FOSDEM'19 - Rust Devroom

Containing the RDMA plasma

Andrea Lattuada (@utaal)
andrea.lattuada.me / andrea@lattuada.me
Systems Group, Dept. of Computer Science, ETH Zürich

I'm supported by a Google PhD fellowship.

FOSDEM'19 - Rust Devroom

Containing the RDMA plasma

Andrea Lattuada (@utaal)
andrea.lattuada.me / andrea@lattuada.me
Systems Group, Dept. of Computer Science, ETH Zürich

I'm supported by a Google PhD fellowship.

Safe DMA with ownership

Safe DMA with ownership

Hardware

Safe DMA with ownership

Hardware **direct access to program-owned memory**

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Data Race doc.rust-lang.org/nomicon/races.html

- **two or more threads concurrently accessing a location of memory**
- **one of them is a write**
- **one of them is unsynchronized**

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Data Race doc.rust-lang.org/nomicon/races.html

- **two or more threads concurrently accessing a location of memory**
- **one of them is a write**
- **one of them is unsynchronized**

Hardware operations

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Data Race doc.rust-lang.org/nomicon/races.html

- **two or more threads concurrently accessing a location of memory**
- **one of them is a write**
- **one of them is unsynchronized**

Hardware operations as a thread of control

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Data Race doc.rust-lang.org/nomicon/races.html

- **two or more threads concurrently accessing a location of memory**
- **one of them is a write**
- **one of them is unsynchronized**

Hardware operations as a thread of control

Leverage ownership

Safe DMA with ownership

Hardware	**direct access to program-owned memory**
Safe Rust	**guarantees absence of data races**

Data Race doc.rust-lang.org/nomicon/races.html

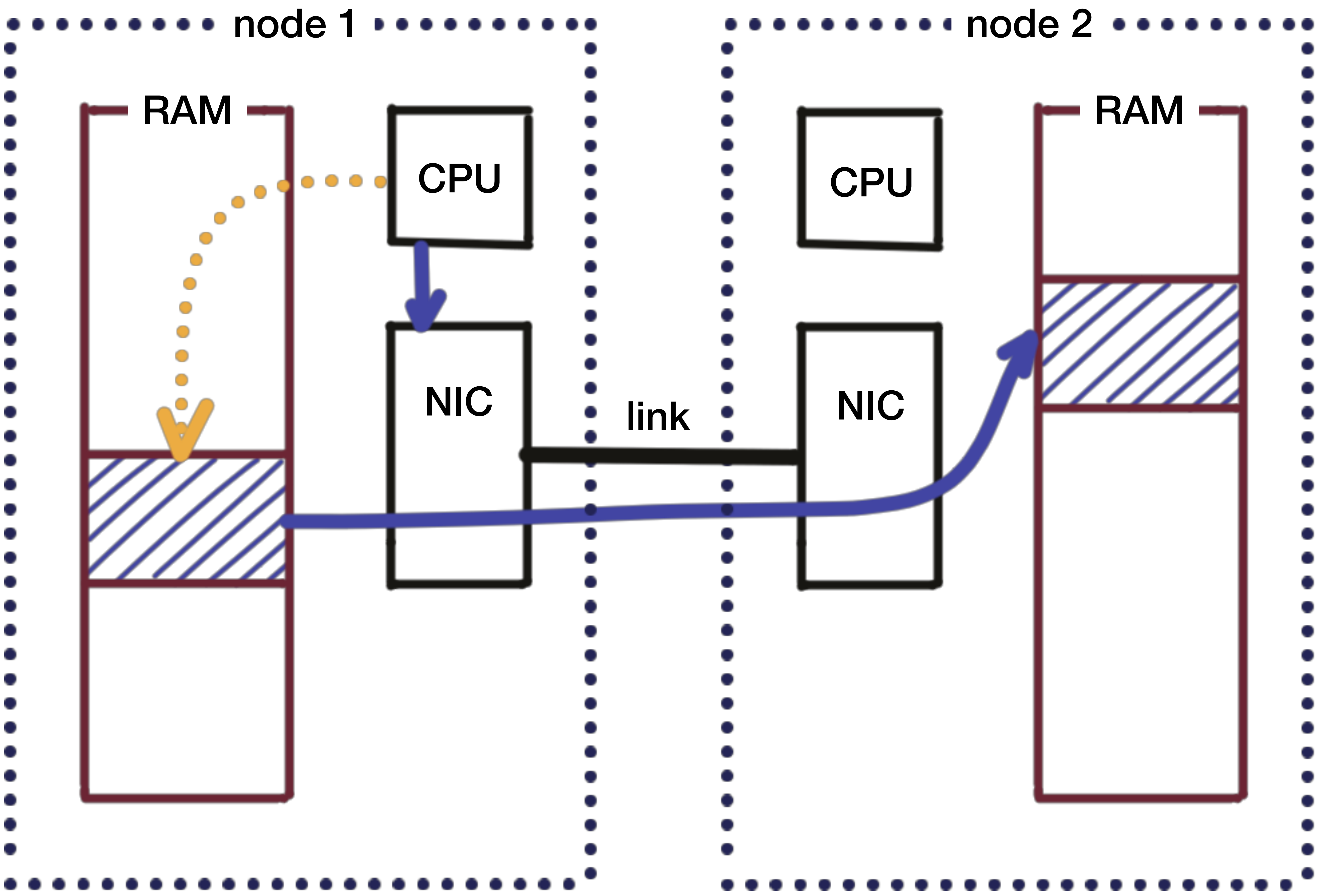
- **two or more threads concurrently accessing a location of memory**
- **one of them is a write**
- **one of them is unsynchronized**

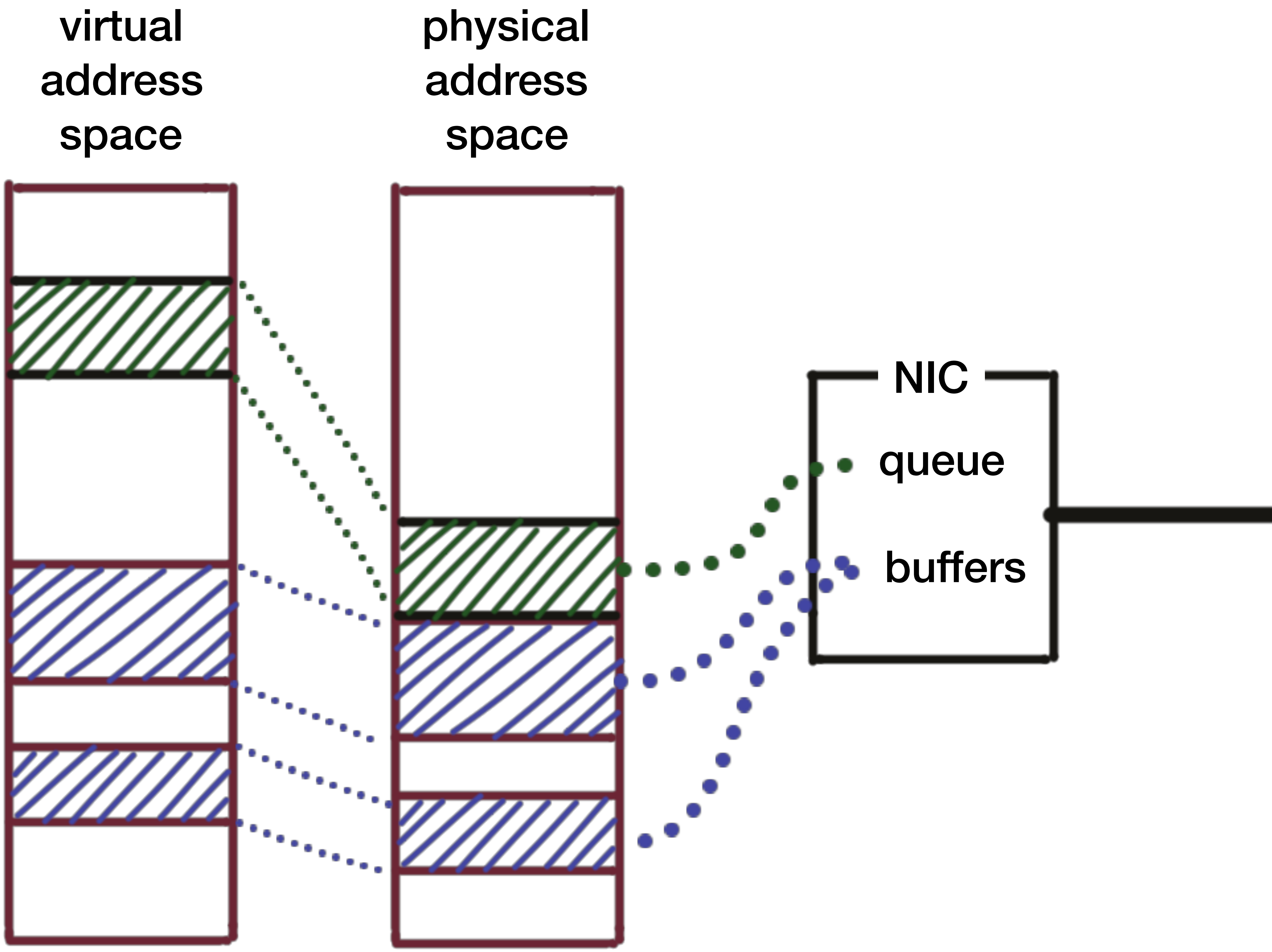
Hardware operations	**as a thread of control**
Leverage ownership	**prevent CPU⟷Hardware data races**

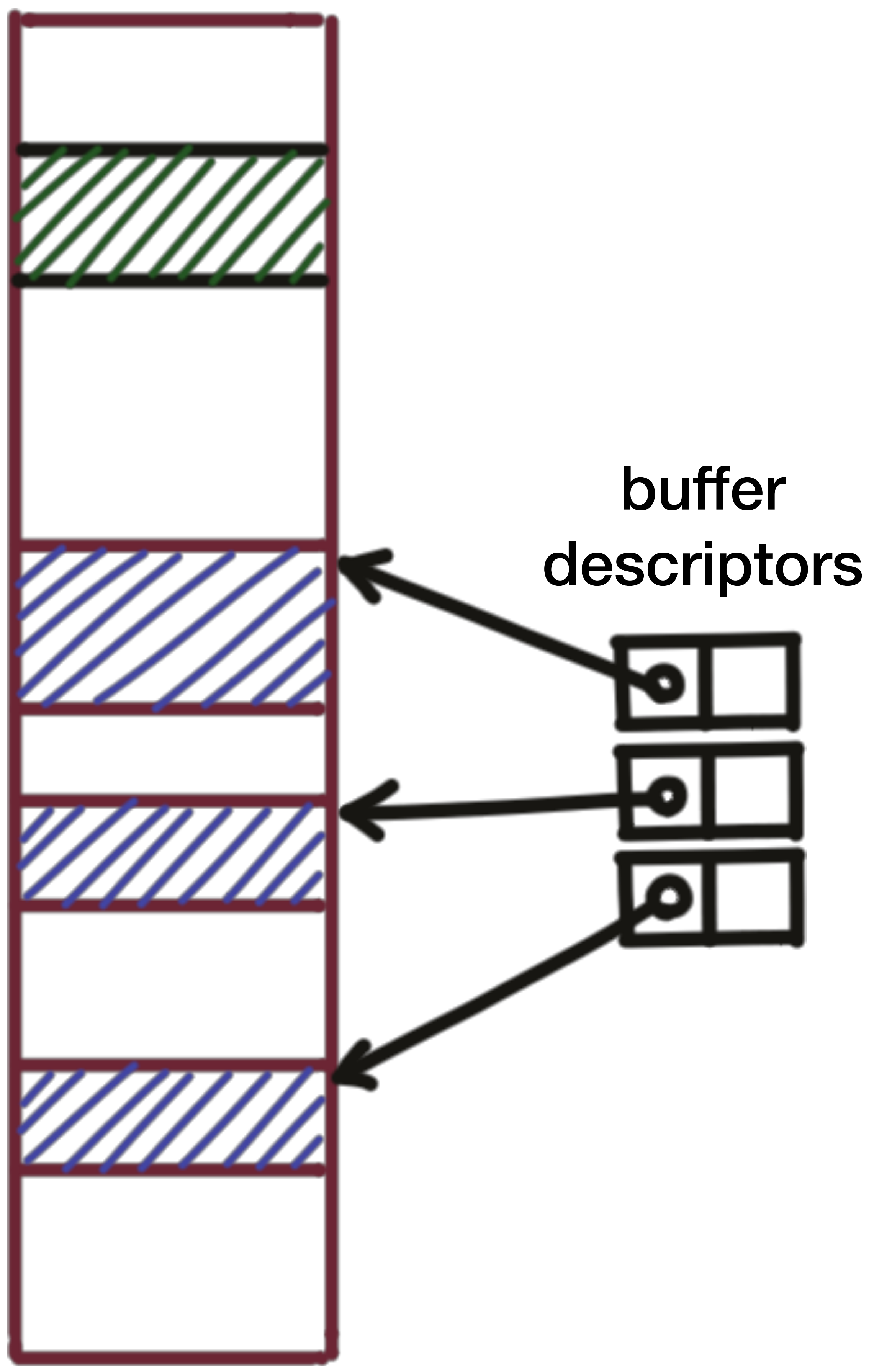
RDMA ibverbs

Remote Direct Memory Access

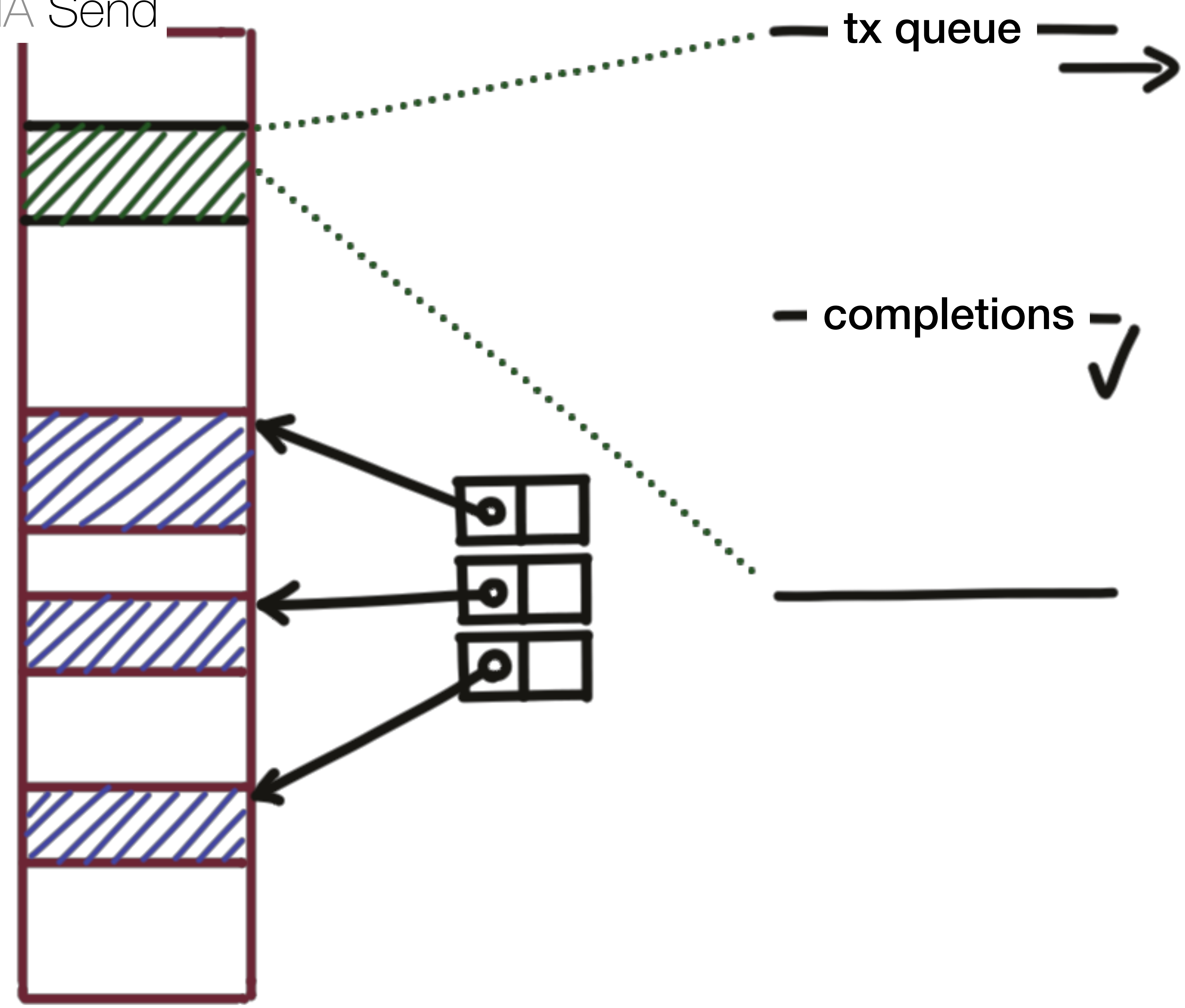
Access hardware RDMA verbs from userspace


node 1
RAM
CPU
NIC
link
node 2
CPU
NIC
RAM

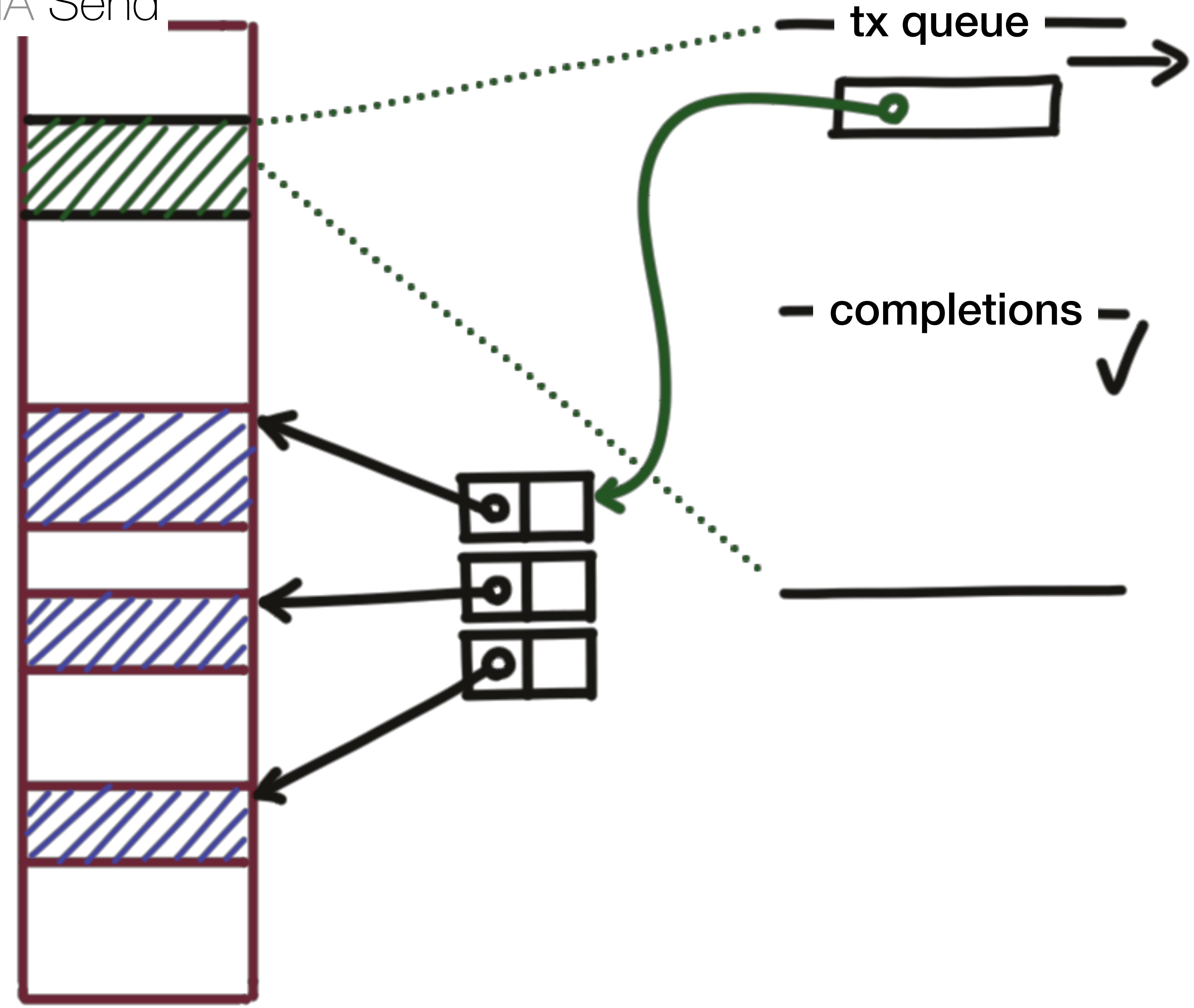

virtual address space
physical address space
NIC
queue
buffers


buffer
descriptors

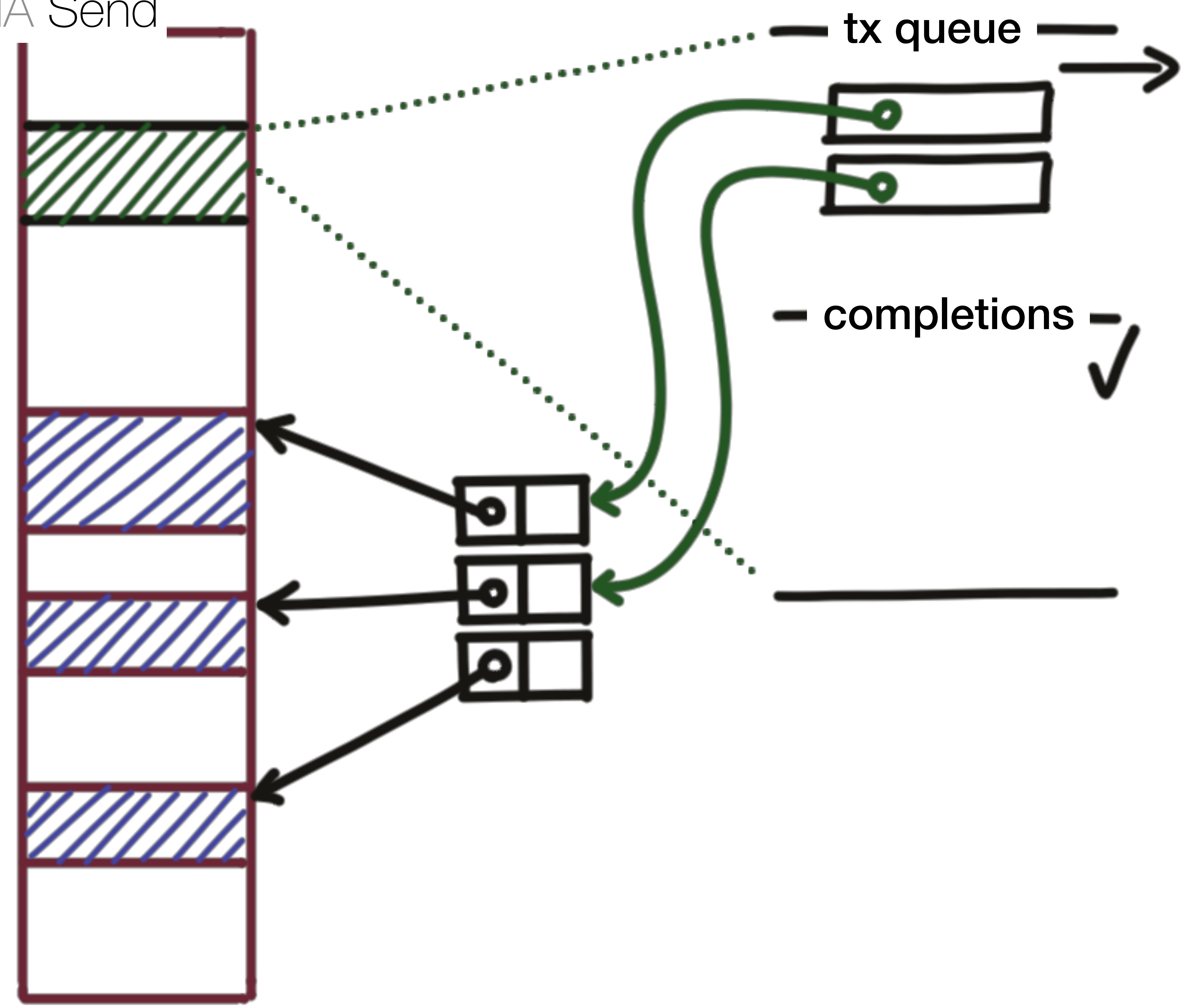
RDMA Send



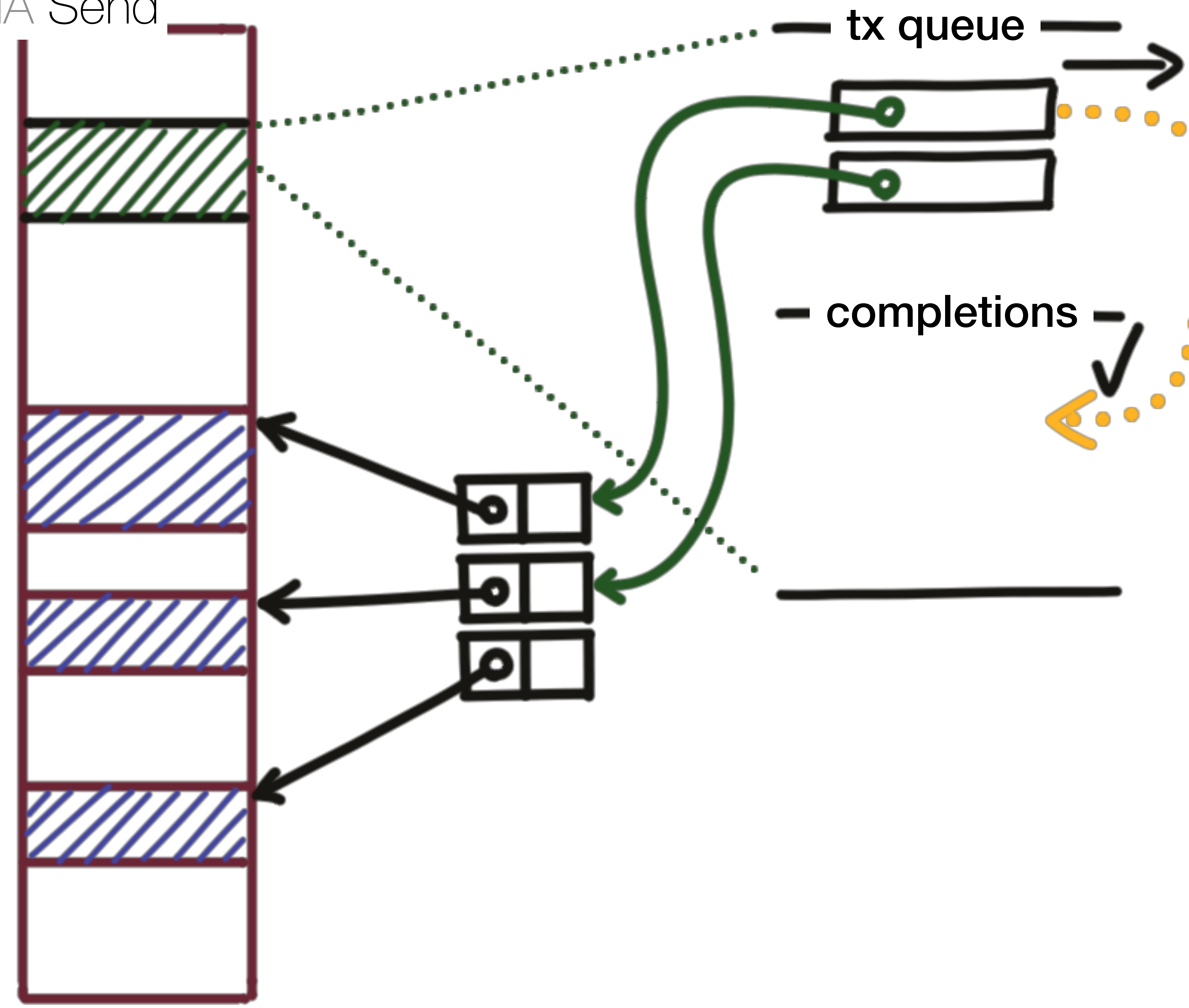
RDMA Send



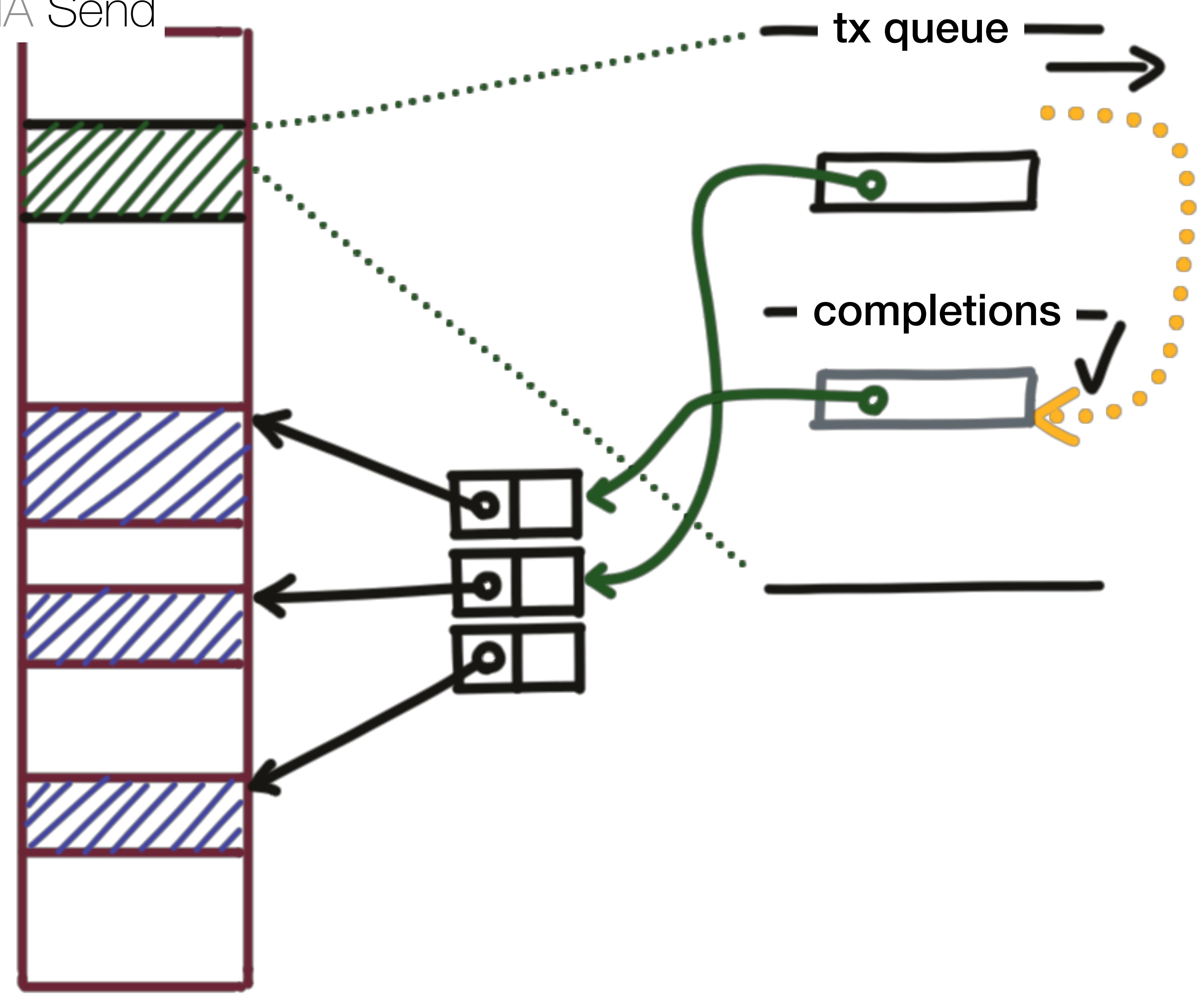
RDMA Send



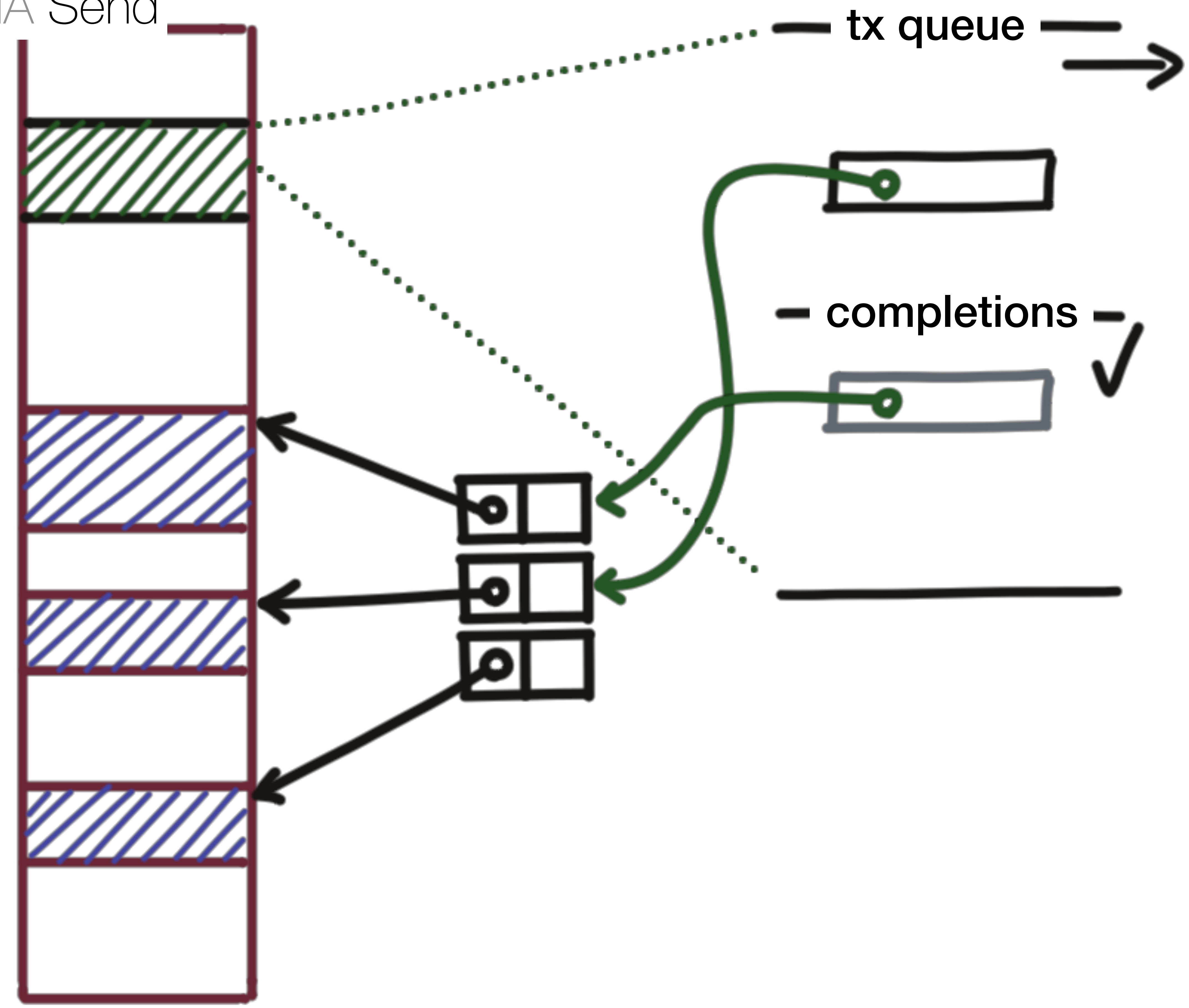
RDMA Send



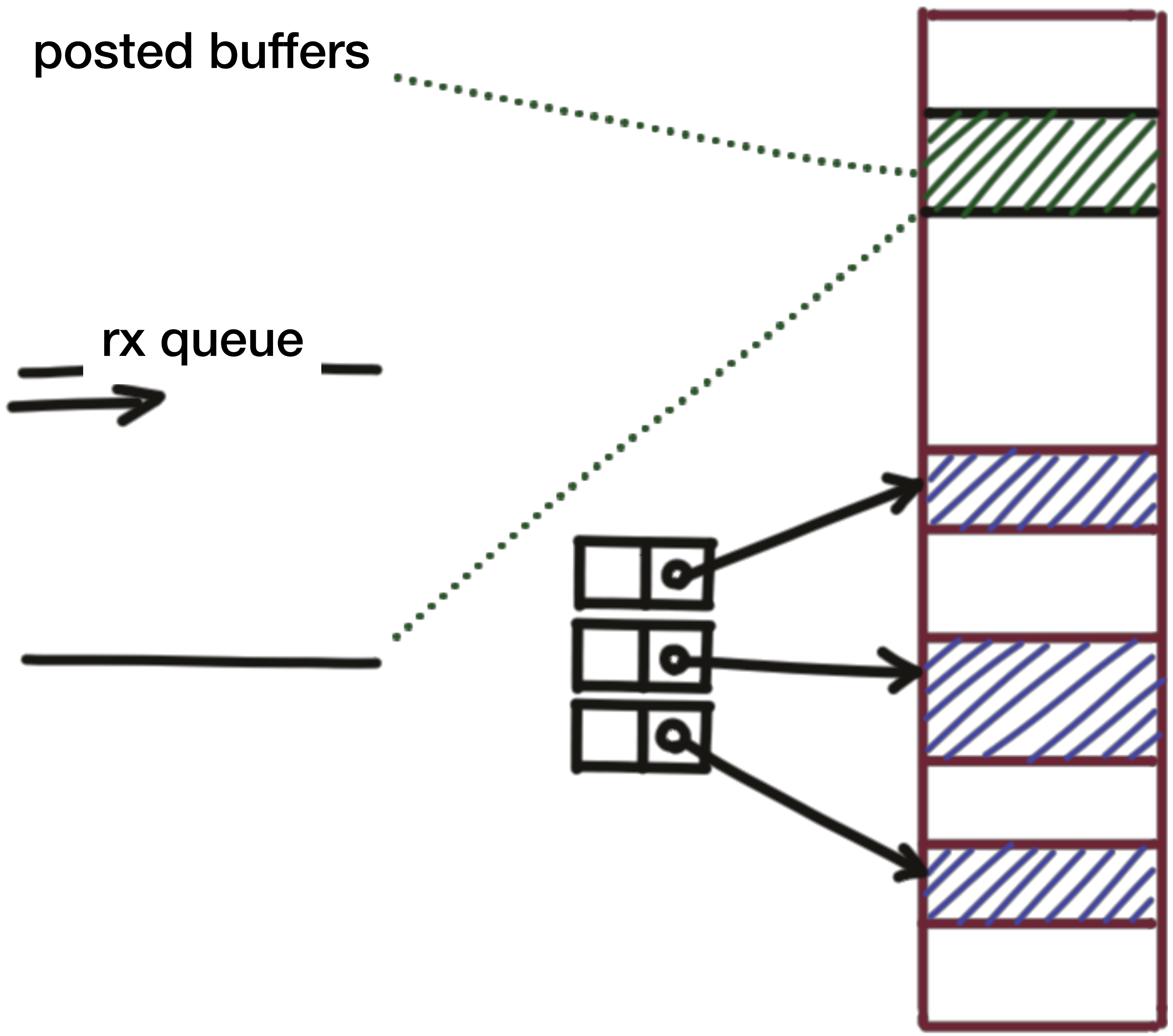
RDMA Send



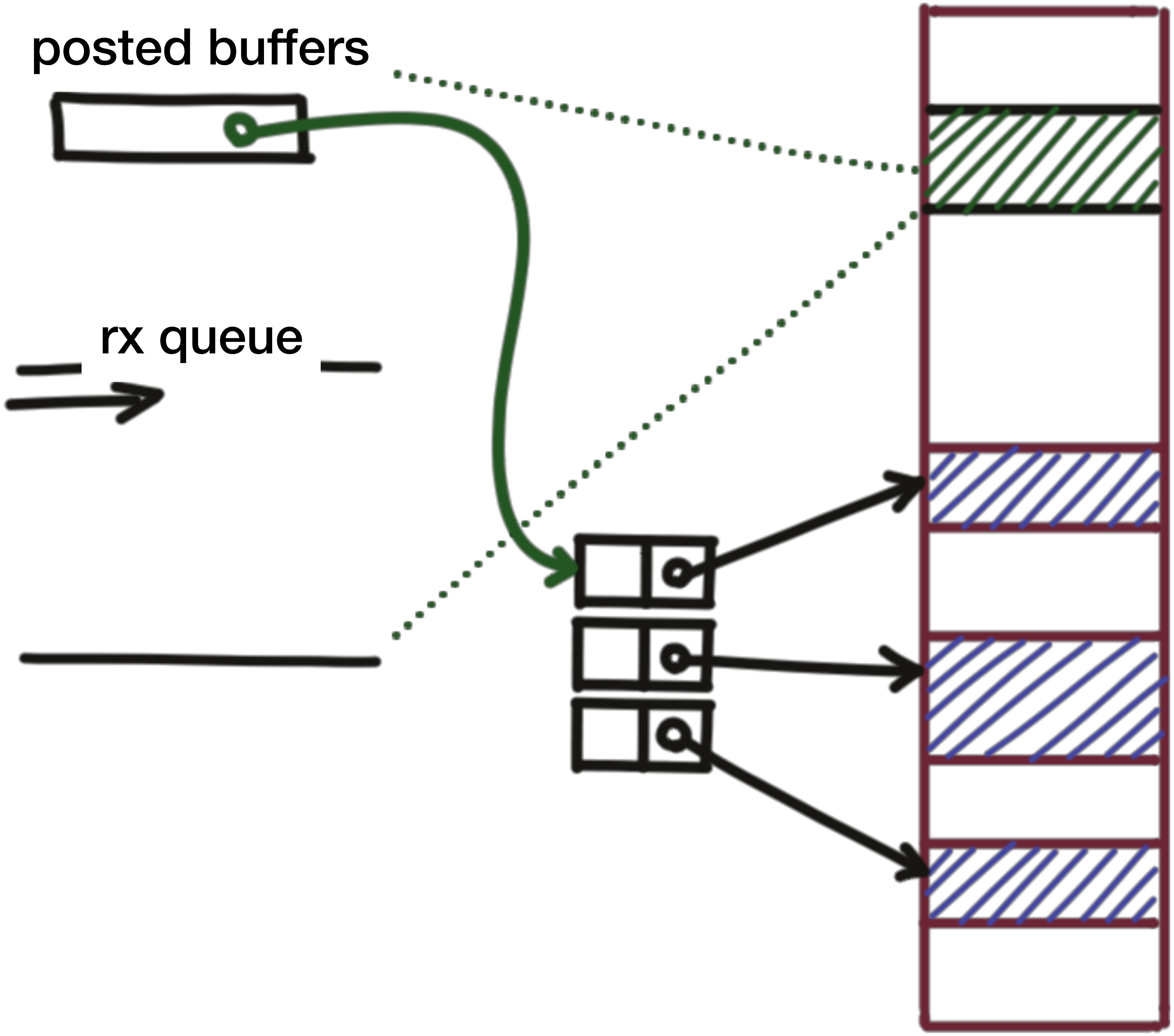
RDMA Send



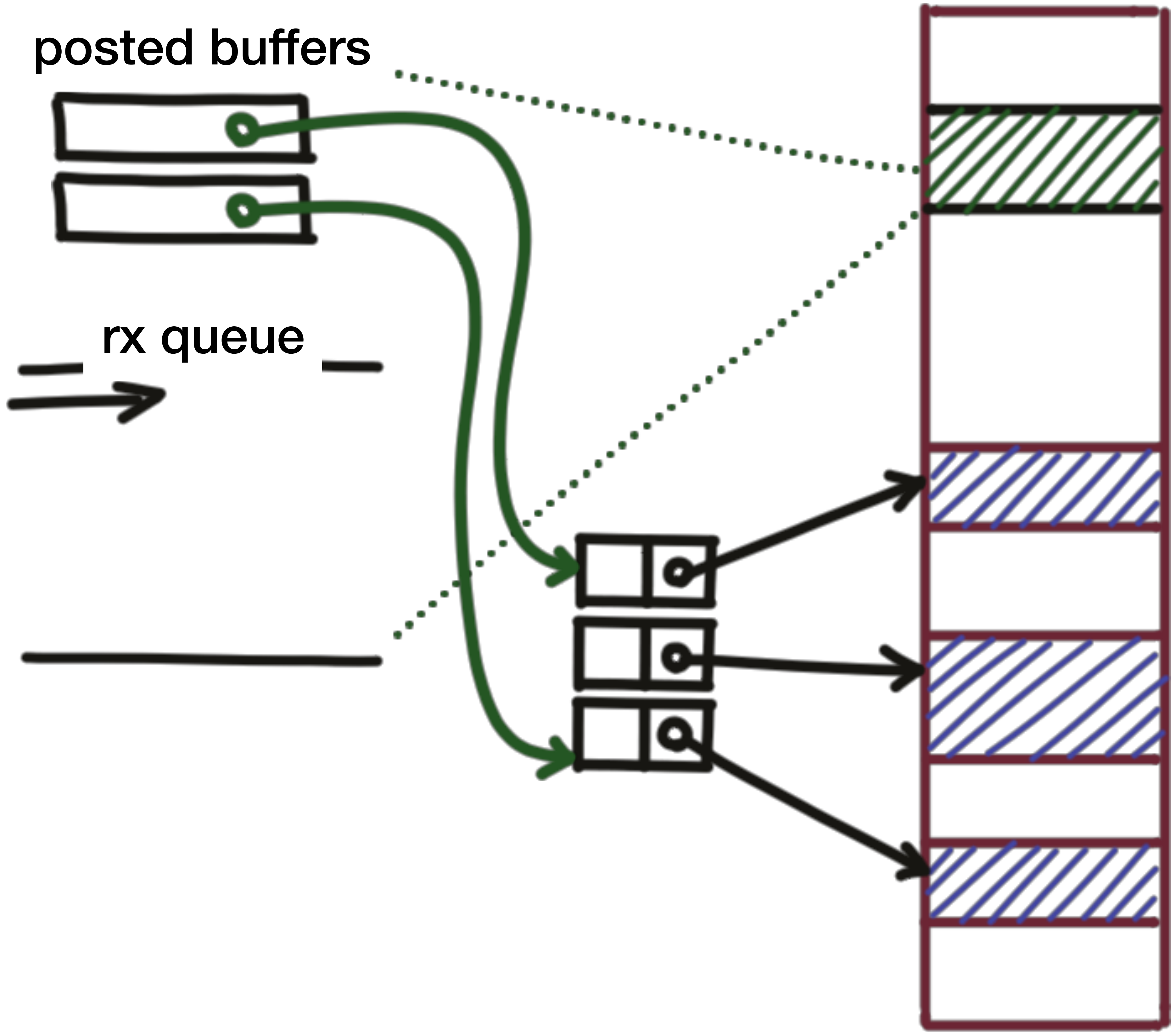
RDMA Receive



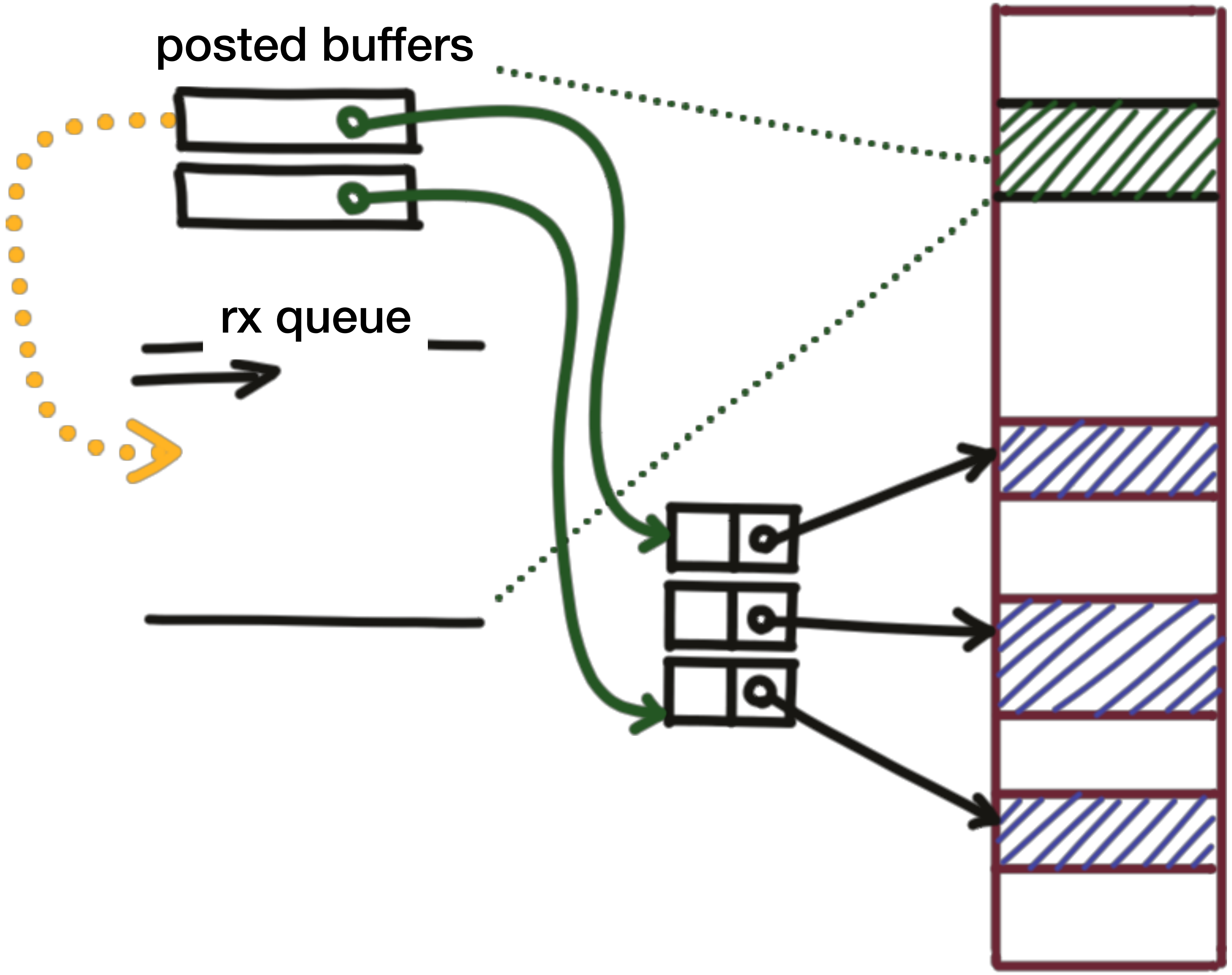
RDMA Receive



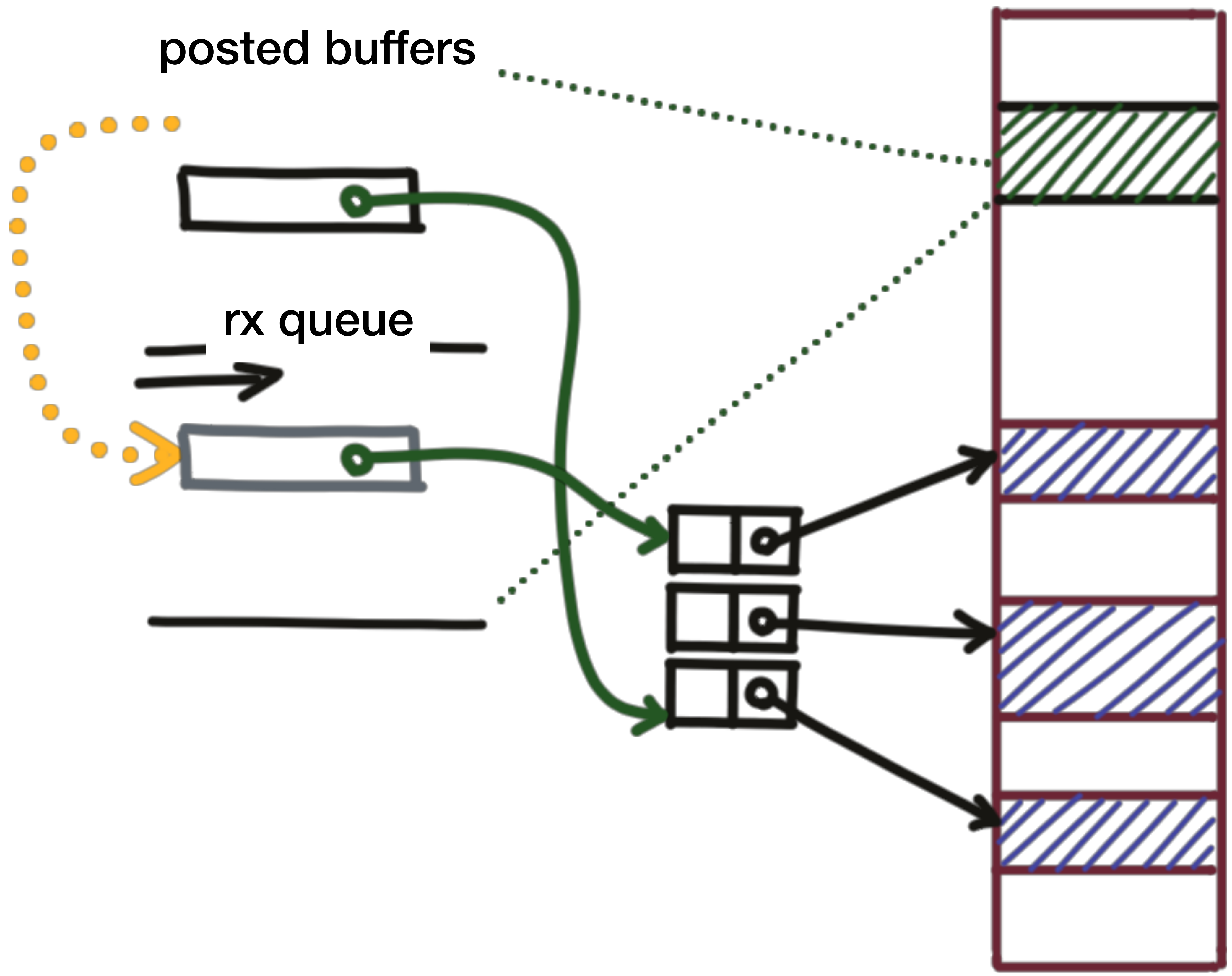
RDMA Receive



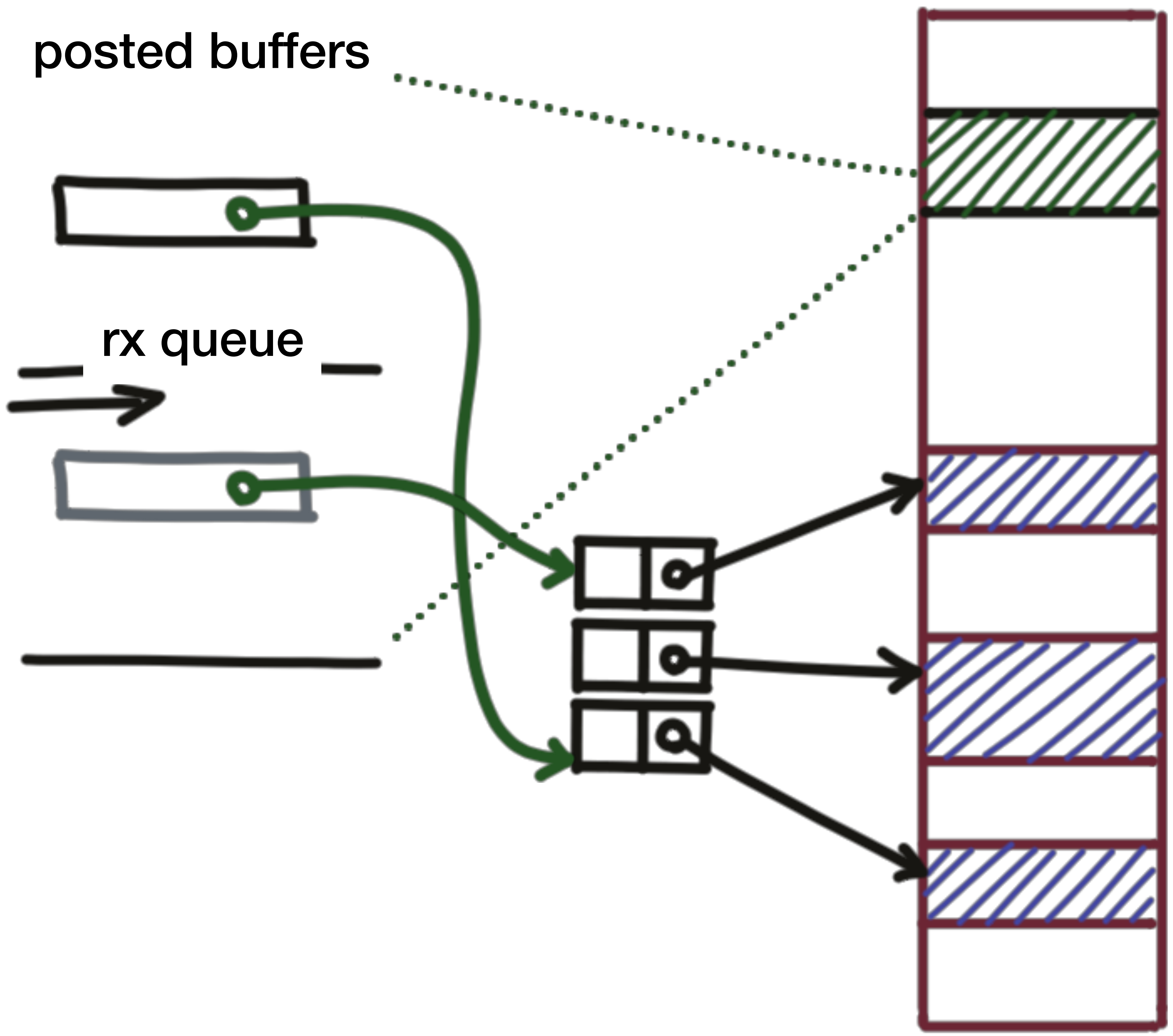
RDMA Receive



RDMA Receive



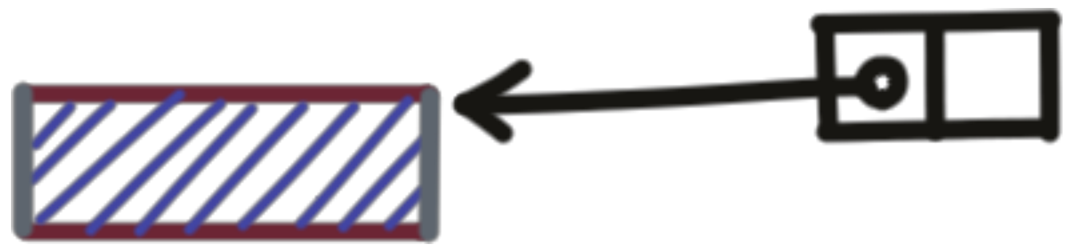
RDMA Receive



github.com/claudebarthels/infinity

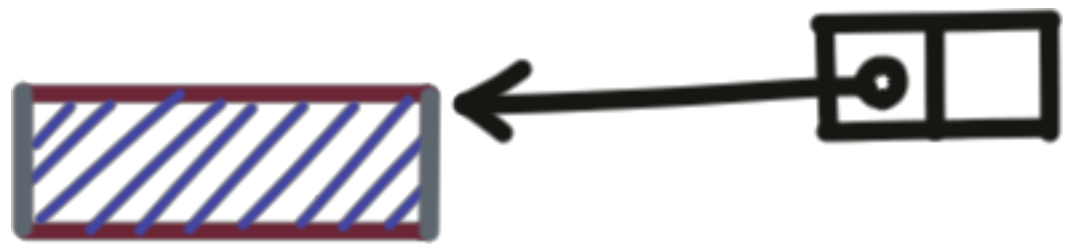
Infinity is a simple, powerful, object-oriented abstraction of ibVerbs.

claudebarthels/infinity



Buffer

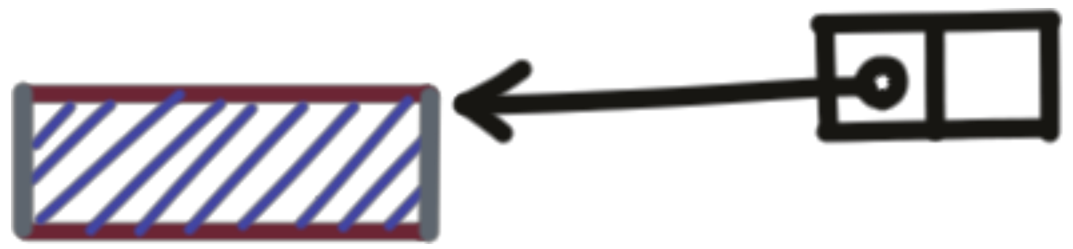
claudebarthels/infinity



Buffer

```
infinity::memory::Buffer *buffer =
    new infinity::memory::Buffer(context, 1024 * sizeof(char));
```

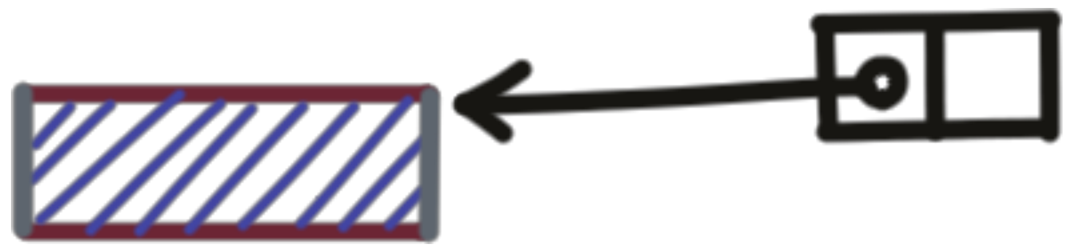
claudebarthels/infinity



Buffer

```
infinity::memory::Buffer *buffer =
    new infinity::memory::Buffer(context, 1024 * sizeof(char));

void* Buffer::getData() {
   return reinterpret_cast<void *>(this->getAddress());
}
```

claudebarthels/infinity



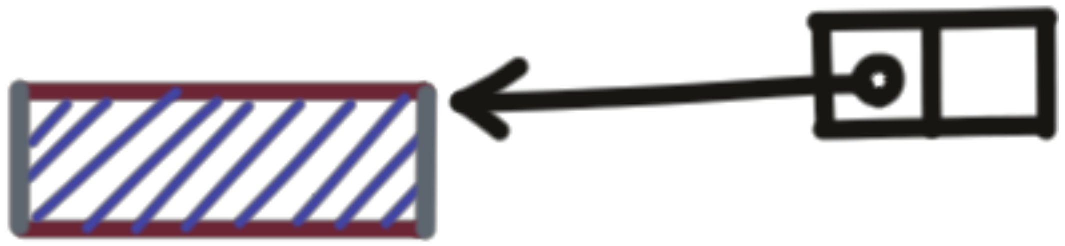
Buffer

```
infinity::memory::Buffer *buffer =
    new infinity::memory::Buffer(context, 1024 * sizeof(char));

void* Buffer::getData() {
   return reinterpret_cast<void *>(this->getAddress());
}
```

sender

receiver

claudebarthels/infinity



Buffer

```
infinity::memory::Buffer *buffer =
    new infinity::memory::Buffer(context, 1024 * sizeof(char));
```

```
void* Buffer::getData() {
   return reinterpret_cast<void *>(this->getAddress());
}
```

sender

receiver

```
context->postReceiveBuffer(buffer);

while(!context->receive(&result));
```

claudebarthels/infinity

Buffer

```
infinity::memory::Buffer *buffer =
    new infinity::memory::Buffer(context, 1024 * sizeof(char));

void* Buffer::getData() {
   return reinterpret_cast<void *>(this->getAddress());
}
```

sender

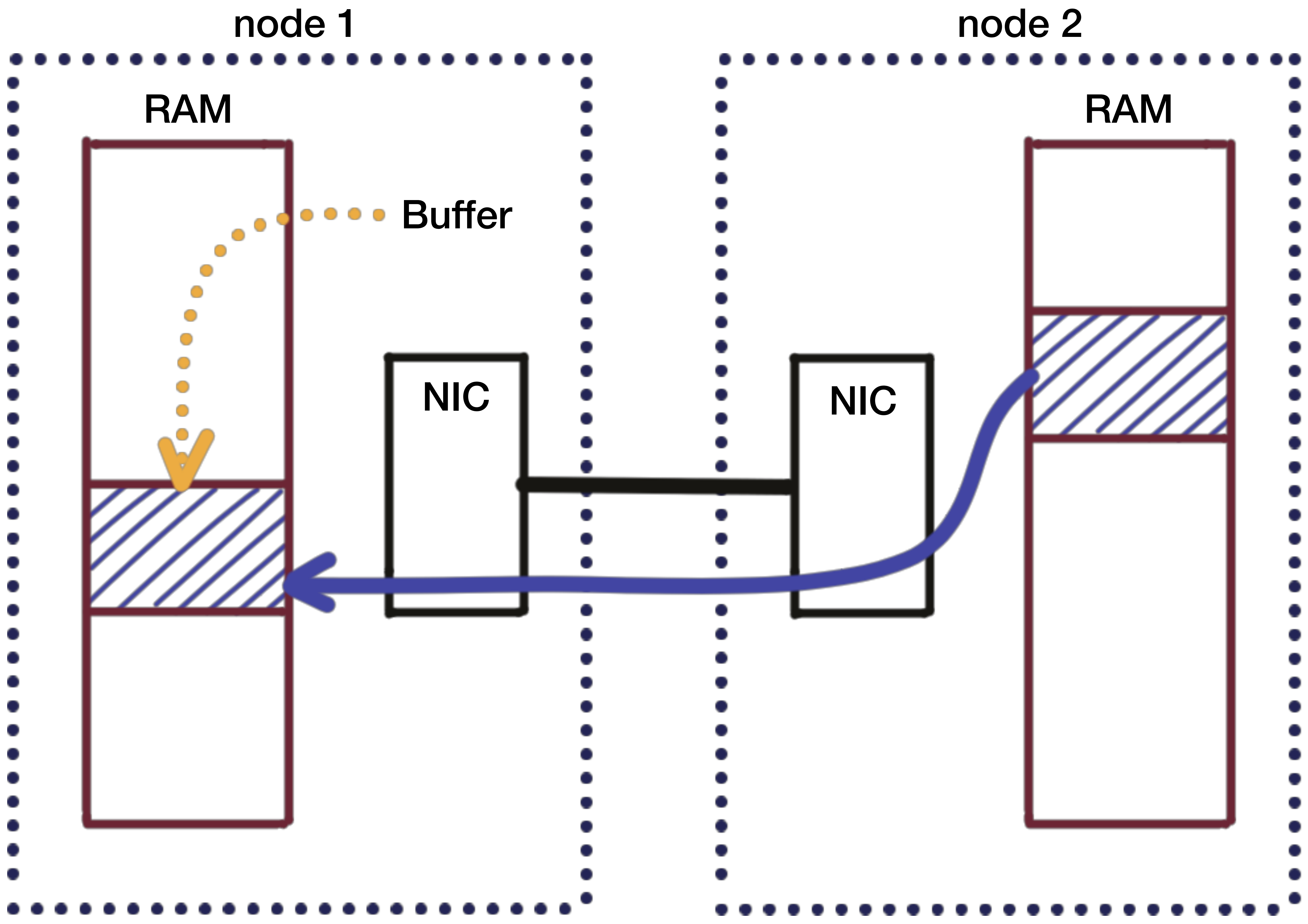
```
infinity::requests::RequestToken
    requestToken(context);

queue->send(buffer, &requestToken);

requestToken.waitUntilCompleted();
```

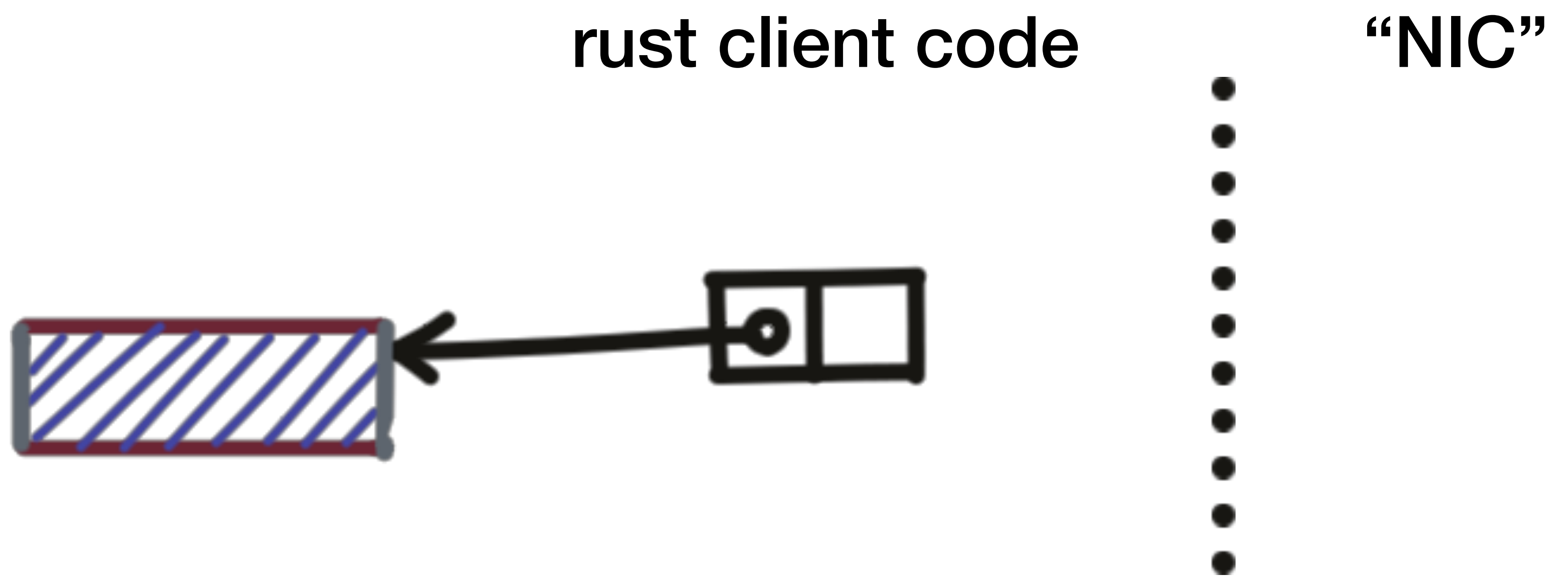
receiver

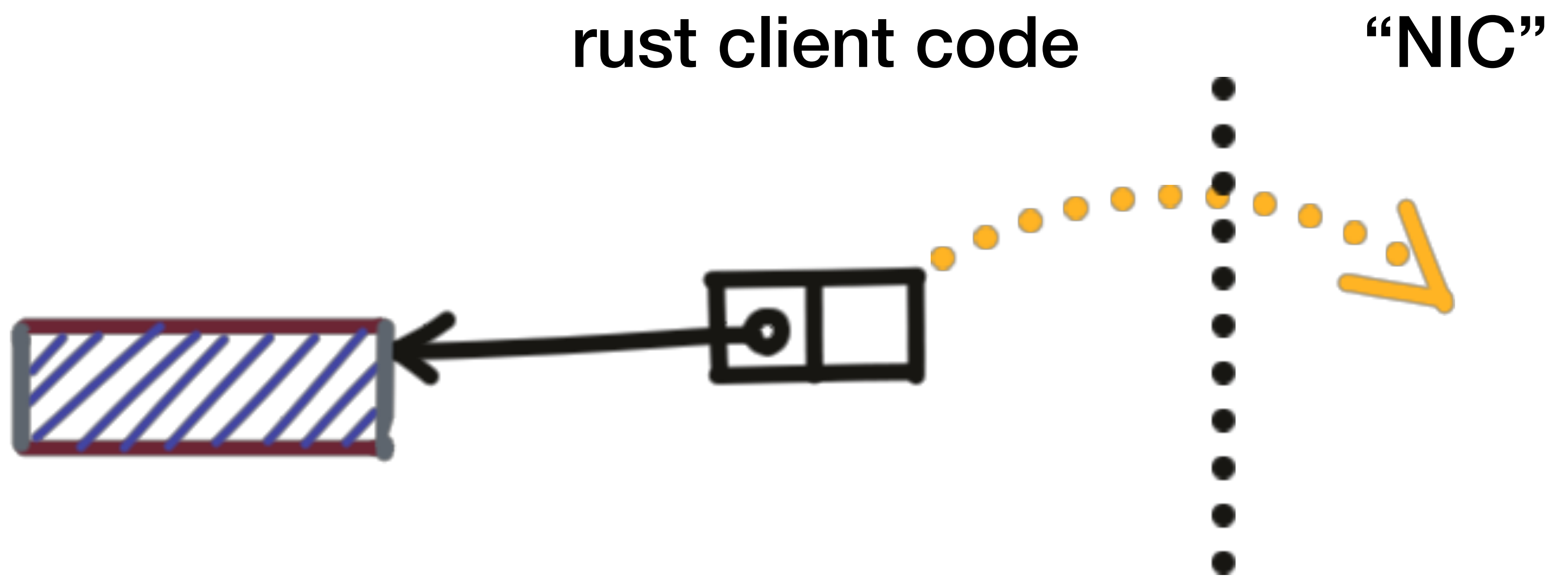
```
context->postReceiveBuffer(buffer);

while(!context->receive(&result));
```

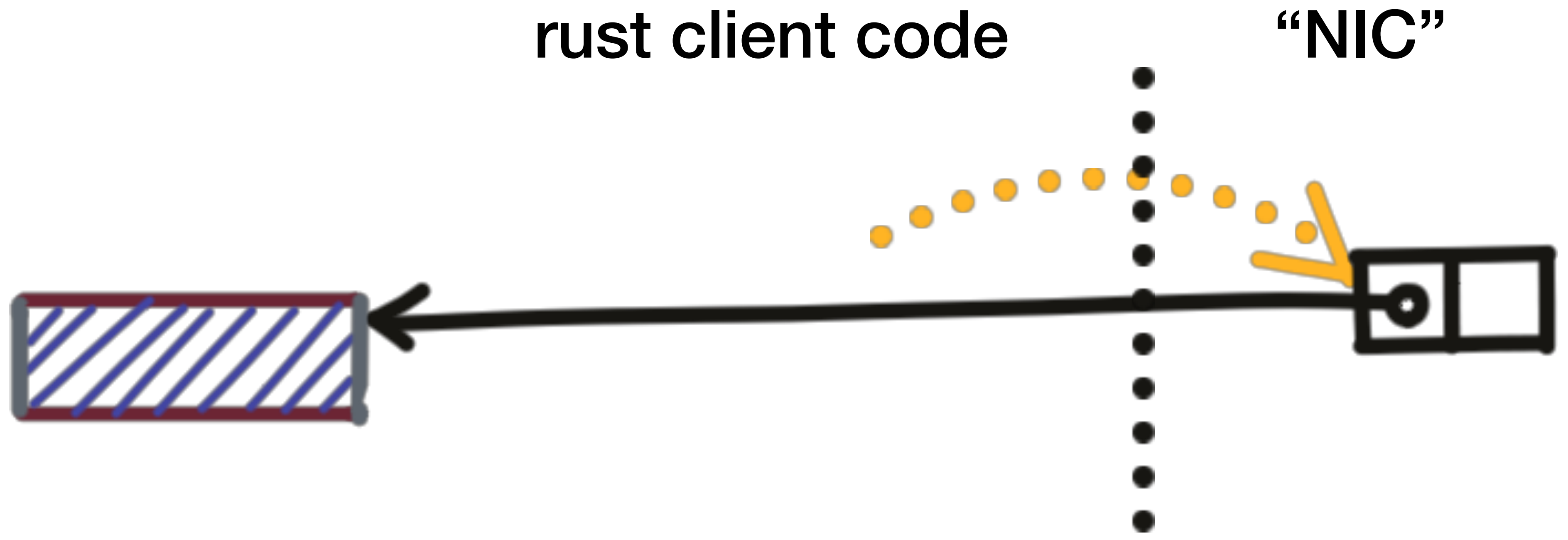
```
pub struct Buffer {
    _buffer: UnsafeCell<Box<ffi::infinity::memory::Buffer>>,
}

impl Buffer {
    pub fn new(context: &::core::Context, size: u64) -> Self {
        unsafe { Buffer {
            _buffer:
                UnsafeCell::new(Box::new(
                    ffi::infinity::memory::Buffer::new(context._context), size)),
        } }
    }
```

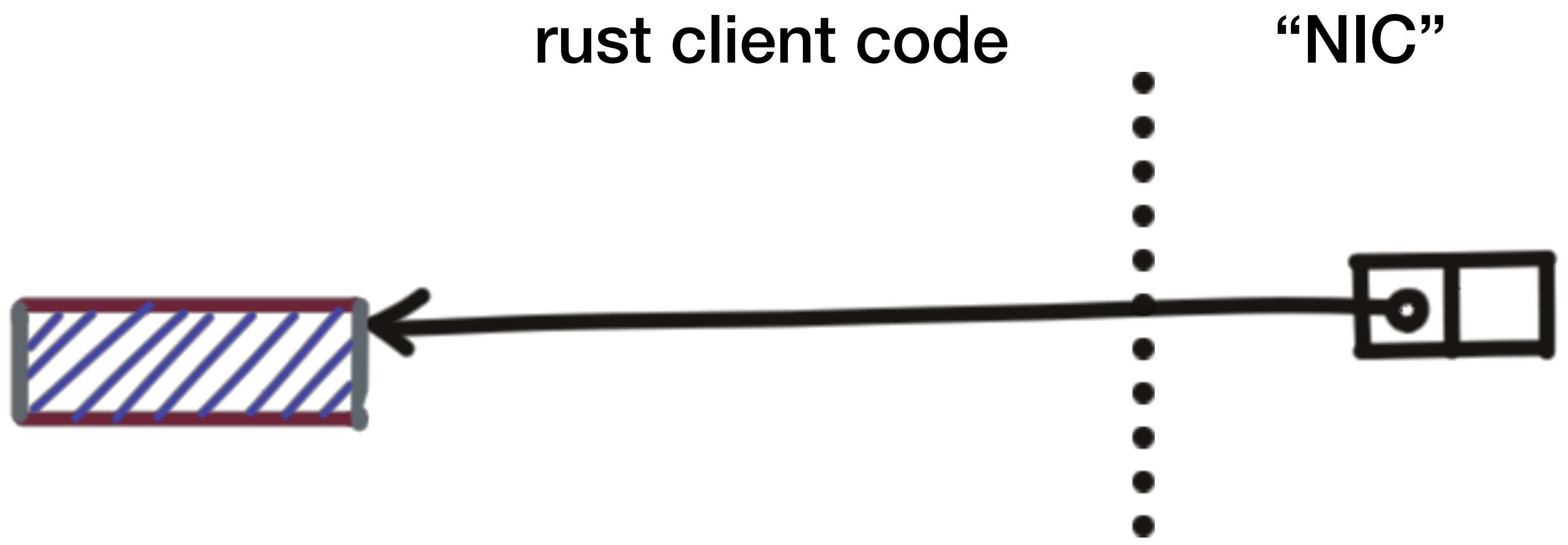
```
impl ::std::ops::DerefMut for Buffer {
    fn deref_mut(&mut self) -> &mut[u8] {
        unsafe {
            ::std::slice::from_raw_parts_mut(
                ::std::mem::transmute::<_, *mut u8>(
                    (*self._buffer.get()).getData()),
                (*self._buffer.get()).getSizeInBytes() as usize)
        }
    }
}
```

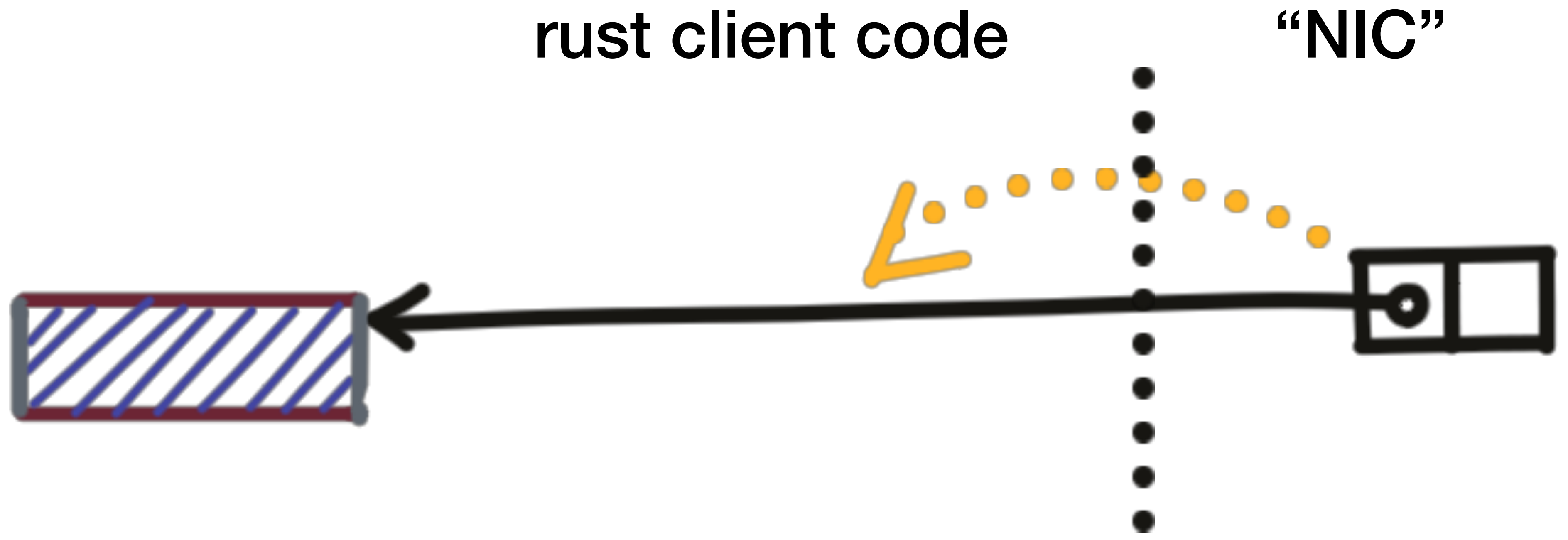

node 1
node 2
RAM
Buffer
NIC
NIC
RAM

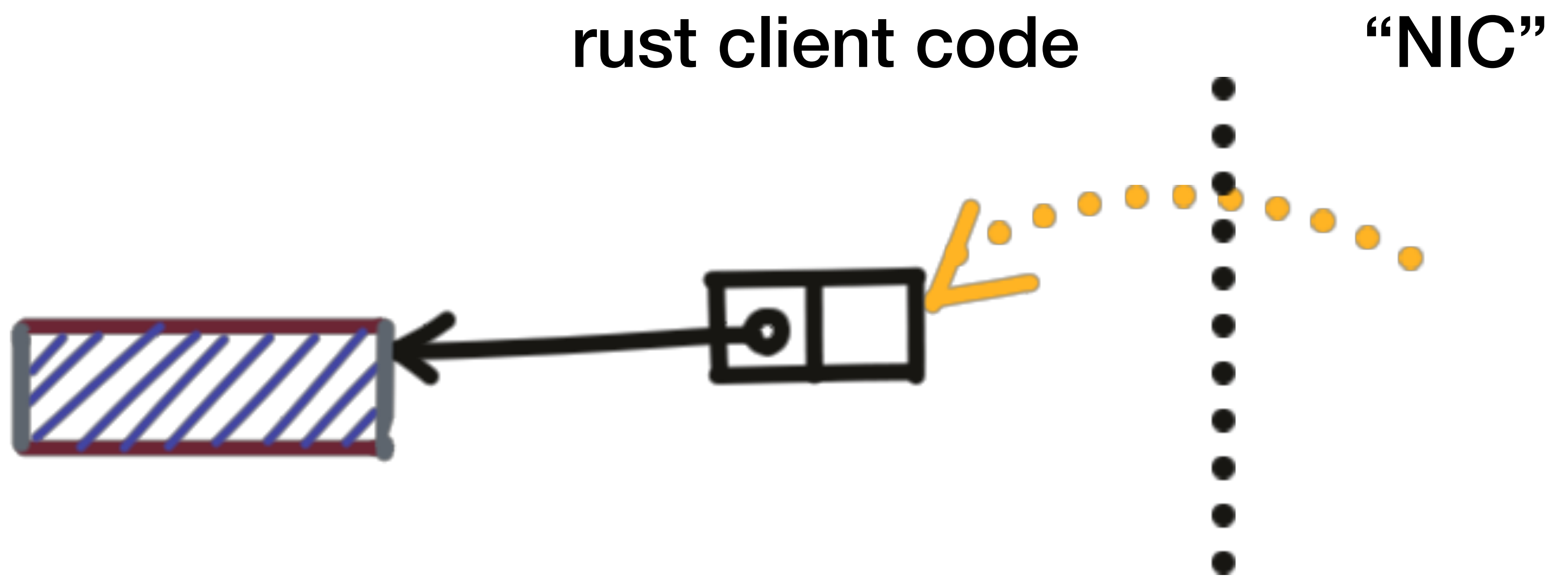

rust client code
“NIC”


rust client code
"NIC"


rust client code
“NIC”

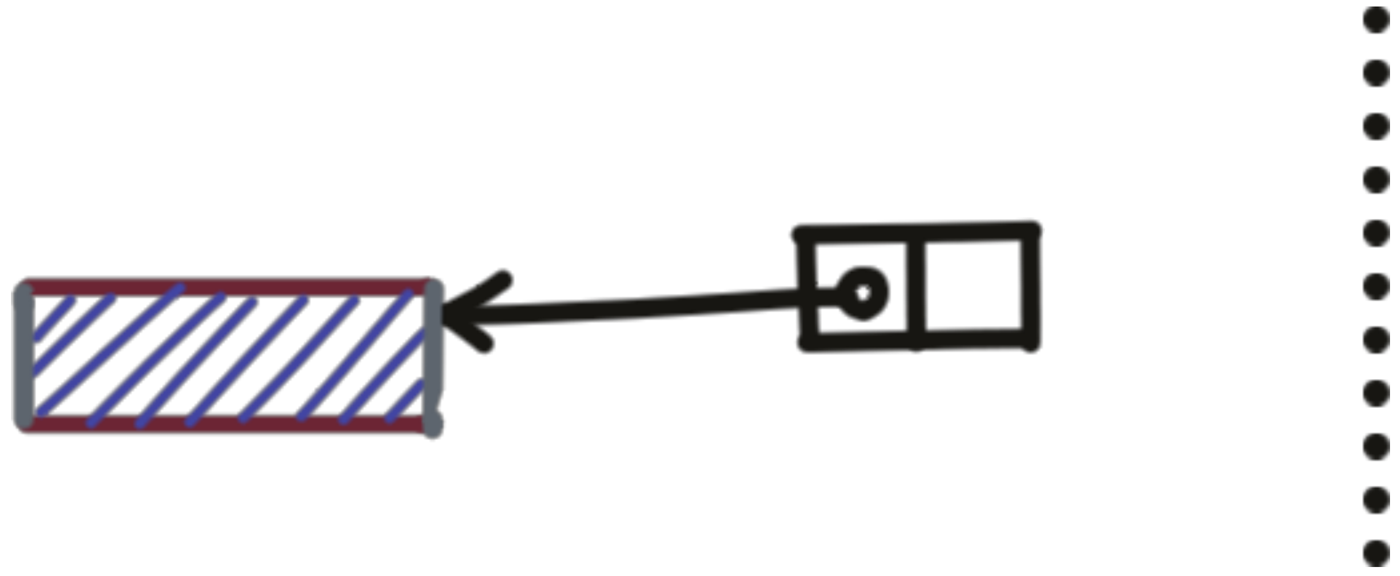

rust client code
"NIC"


rust client code
“NIC”


rust client code
“NIC”

rust client code

“NIC”



```
pub struct Buffer {
    _buffer: UnsafeCell<Box<ffi::infinity::memory::Buffer>>,
}
```

```
pub struct Buffer {
    _buffer: UnsafeCell<Box<ffi::infinity::memory::Buffer>>,
}
```

```
impl Buffer {

  pub(crate) unsafe fn into_raw(self) → *mut ffi::infinity::memory::Buffer {
      Box::into_raw(self.into_inner())
  }
```

```rust
pub struct Buffer {
    _buffer: UnsafeCell<Box<ffi::infinity::memory::Buffer>>,
}
```

```rust
impl Buffer {

  pub(crate) unsafe fn into_raw(self) -> *mut ffi::infinity::memory::Buffer {
      Box::into_raw(self.into_inner())
  }
```

```rust
  pub(crate) unsafe fn from_raw(
          buffer: *mut ffi::infinity::memory::Buffer) -> Self {
      Buffer {
          _buffer: UnsafeCell::new(Box::from_raw(buffer)),
      }
  }
```

```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) -> ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

        ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
            ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

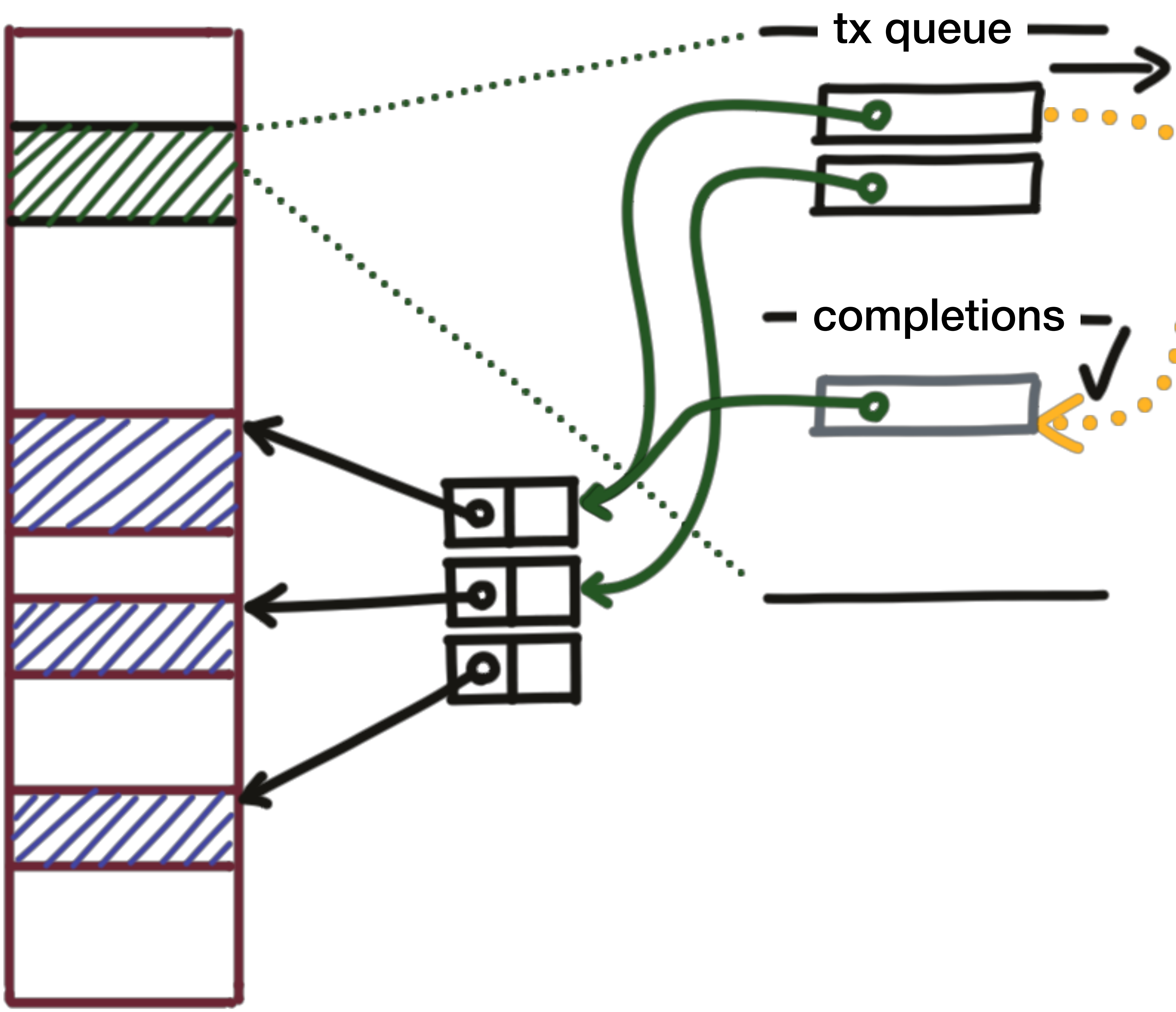
```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) -> ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

         ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
             ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) → ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

         ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
             ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) -> ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

         ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
             ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) -> ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

         ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
             ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

```
impl Queue {

pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) → ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

         …

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
             …
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```


tx queue
completions
✓

```
pub fn send(
        &mut self,
        mut buffer: ::memory::Buffer,
        options: SendOptions) → ::requests::RequestToken
```

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) → ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            …
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

```
pub fn send(
        &mut self,
        mut buffer: ::memory::Buffer,
        options: SendOptions) → ::requests::RequestToken
```

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) → ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            …
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

```
pub fn send(
        &mut self,
        mut buffer: ::memory::Buffer,
        options: SendOptions) -> ::requests::RequestToken
```

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) -> ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            …
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

```
pub fn send(
        &mut self,
        mut buffer: ::memory::Buffer,
        options: SendOptions) → ::requests::RequestToken
```

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) → ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            …
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

```
pub fn send(
       &mut self,
       mut buffer: ::memory::Buffer,
       options: SendOptions) → ::requests::RequestToken
```

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) → ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            ...
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

```
let mut context = infinity::core::Context::new(0, 1);
let mut qp_factory = infinity::queues::QueuePairFactory::new(&context);
let mut context = infinity::core::Context::new(0, 1);
let mut qp_factory = infinity::queues::QueuePairFactory::new(&context);
```

receiver

```
let mut buffer = infinity::memory::Buffer::new(&context, 128);
context.post_receive_buffer(buffer); // give up ownership of buffer
…
let infinity::core::ReceiveElement { buffer: (mut recv_buf, recv_len), .. } =
    loop {
        if let Some(el) = context.receive() {
            break el;
        }
    };
…
```

```
let mut context = infinity::core::Context::new(0, 1);
let mut qp_factory = infinity::queues::QueuePairFactory::new(&context);
```

receiver

```
let mut buffer = infinity::memory::Buffer::new(&context, 128);
context.post_receive_buffer(buffer); // give up ownership of buffer
...
let infinity::core::ReceiveElement { buffer: (mut recv_buf, recv_len), .. } =
    loop {
        if let Some(el) = context.receive() {
            break el;
        }
    };
...
```

sender

```
let mut buffer = infinity::memory::Buffer::new(&context, 128);
...
let request_token = qp.send(buffer, Default::default()); // give up ownership
...
let buffer = request_token.wait_until_completed();
```

The **unsafe** boundary

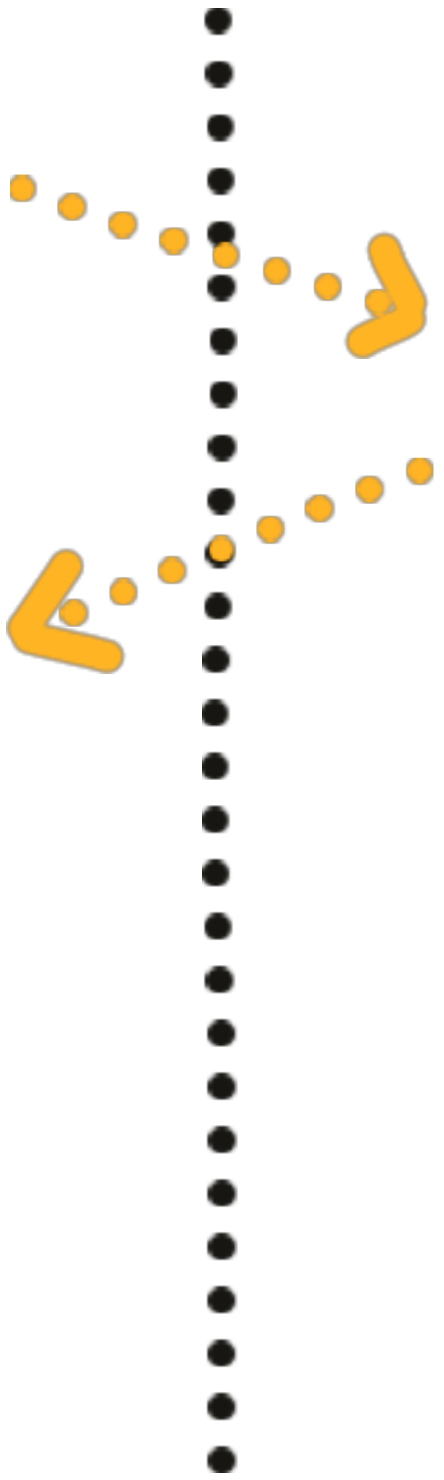
```
pub fn send(
    &mut self,
    mut buffer: ::memory::Buffer,
    options: SendOptions) -> ::requests::RequestToken {

    unsafe {
        let mut _request_token = Box::new(
            ffi::infinity::requests::RequestToken::new(self.context._context));

        ...

        (*self._queue_pair).send2(
            buffer.into_raw(),
            size_in_bytes as u32,
            ...
            &mut (*_request_token) as *mut _);

        ::requests::RequestToken {
            _request_token,
        }
    }
}
```

The **unsafe** boundary

```
pub struct RequestToken {
    pub(crate) _request_token: Box<ffi::infinity::requests::RequestToken>,
}

impl RequestToken {
    pub fn wait_until_completed(mut self) → ::memory::Buffer {
        unsafe {
            self._request_token.waitUntilCompleted();
            …
            Buffer::from_raw(self._request_token.buffer)
        }
    }
```

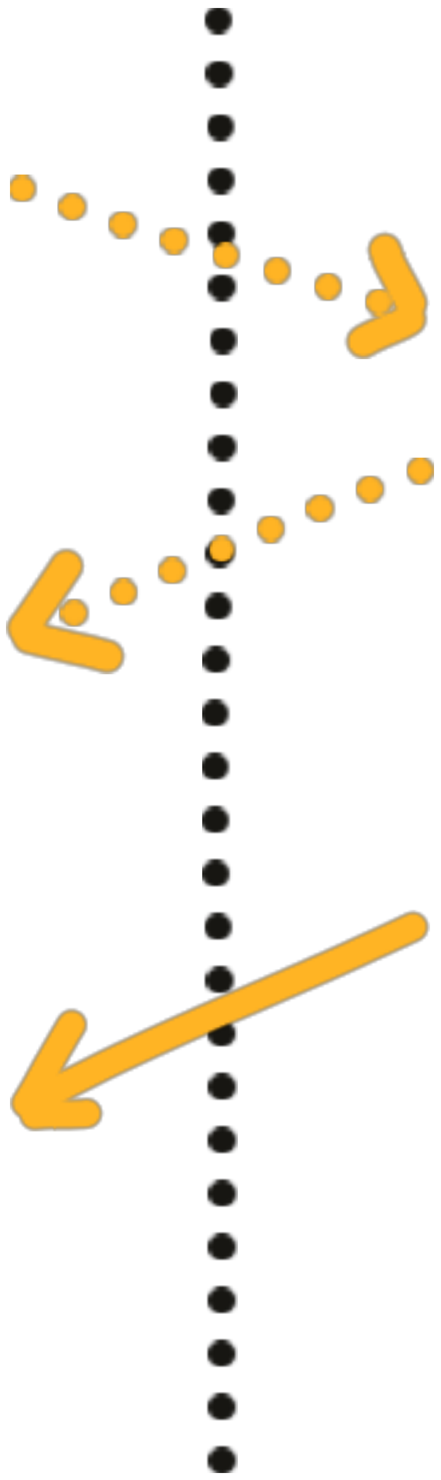
The **unsafe** boundary

```
impl Queue

    pub fn send(…, mut buffer: ::memory::Buffer, …)


impl RequestToken

    pub fn wait_until_completed(&mut self) → ::memory::Buffer
```



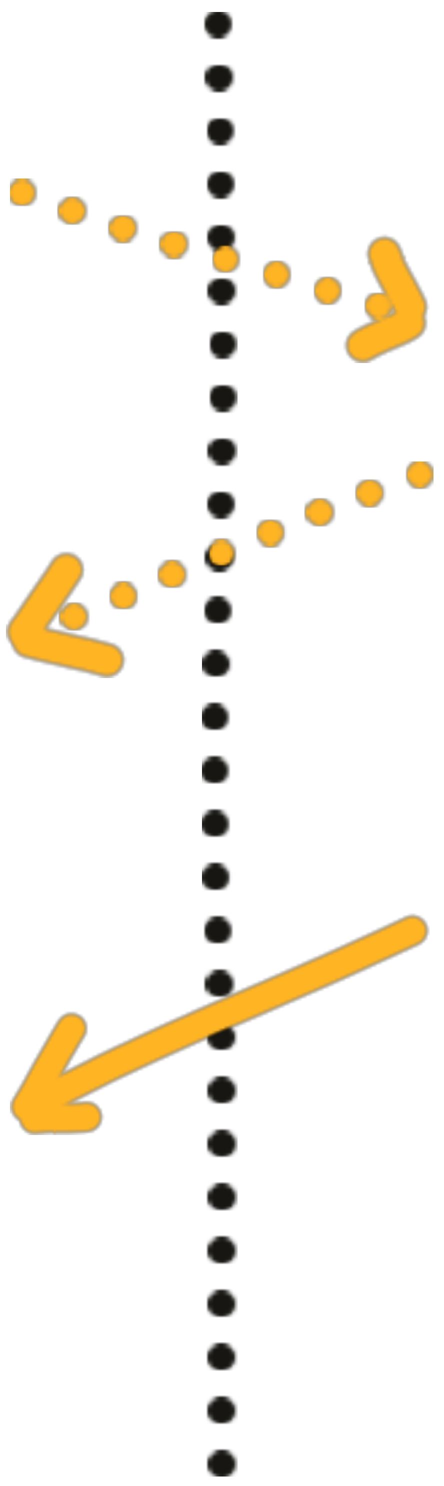
The **unsafe** boundary

```
impl Queue
    pub fn send(…, mut buffer: ::memory::Buffer, …)

impl RequestToken
    pub fn wait_until_completed(&mut self) → ::memory::Buffer



impl Clone for RequestToken
    fn clone(&self) → RequestToken
```



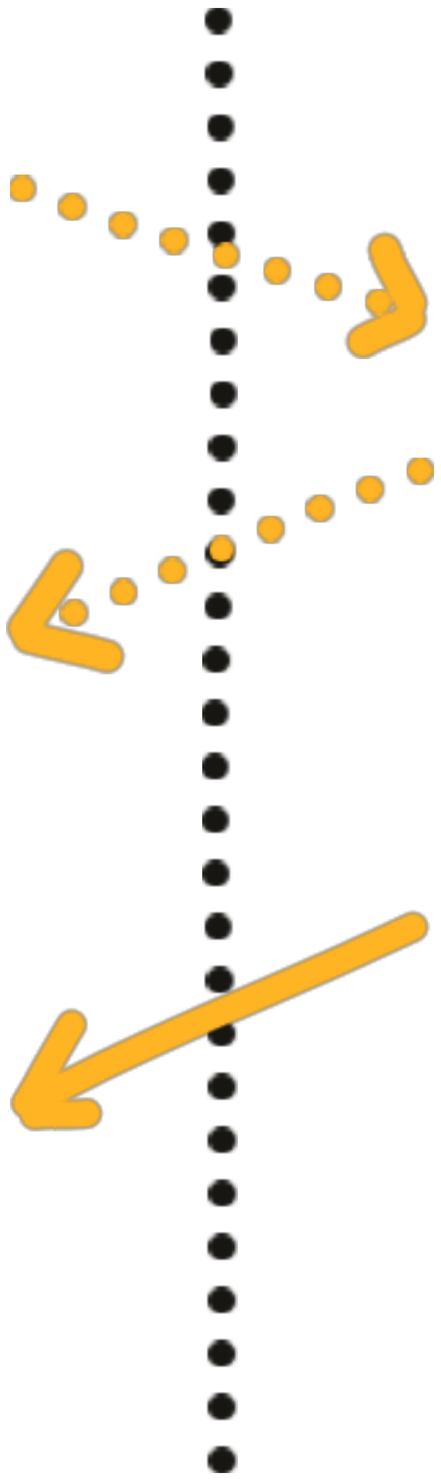
The **unsafe** boundary

```
impl Queue

    pub fn send(…, mut buffer: ::memory::Buffer, …)

impl RequestToken

    pub fn wait_until_completed(&mut self) → ::memory::Buffer



impl Clone for RequestToken

    fn clone(&self) → RequestToken
```



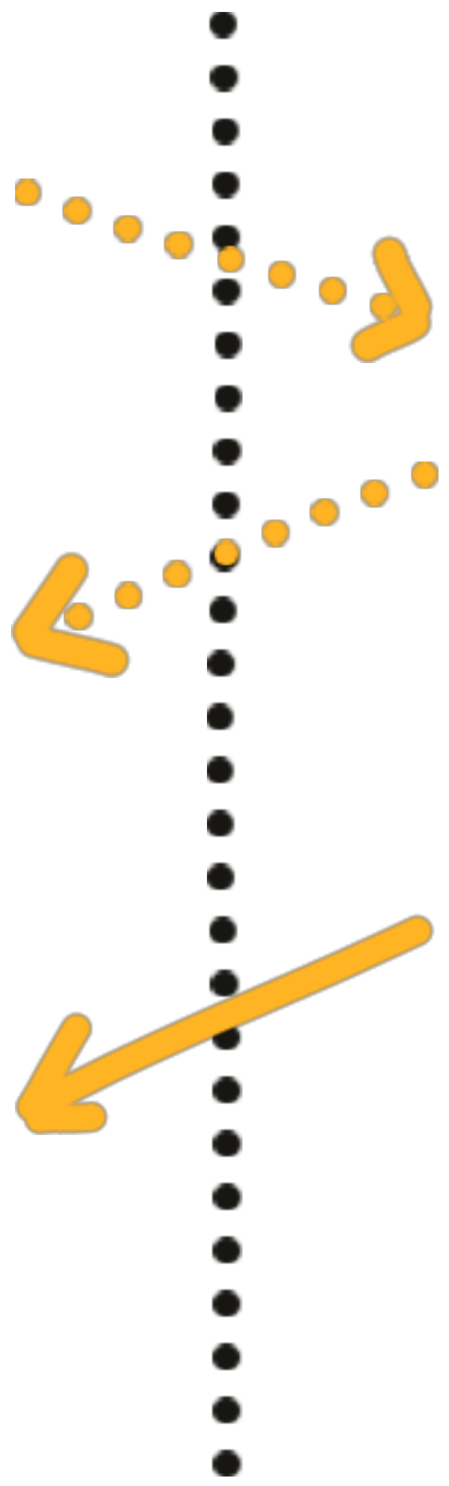
The **unsafe** boundary

```
impl Queue

    pub fn send(…, mut buffer: ::memory::Buffer, …)


impl RequestToken

    pub fn wait_until_completed(&mut self) → ::memory::Buffer

    pub fn wait_until_completed(mut self) → ::memory::Buffer


impl Clone for RequestToken

    fn clone(&self) → RequestToken
```



The **unsafe** boundary

```
impl Queue
    pub fn send(…, mut buffer: ::memory::Buffer, …)

impl RequestToken
    pub fn wait_until_completed(&mut self) -> ::memory::Buffer
    pub fn wait_until_completed(mut self) -> ::memory::Buffer


impl Clone for RequestToken
    fn clone(&self) -> RequestToken
```



Safety is non-local

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Introduce invariants

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Introduce invariants **rely on these invariants**

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Introduce invariants **rely on these invariants**

Safety depends on all of them

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Introduce invariants **rely on these invariants**

Safety depends on all of them

Use ownership and privacy

Safety is non-local

doc.rust-lang.org/stable/nomicon/working-with-unsafe.html

Introduce invariants **rely on these invariants**

Safety depends on all of them

Use ownership and privacy **control the scope of the invariants**

github.com/claudebarthels/infinity

Infinity is a simple, powerful, object-oriented abstraction of ibVerbs.

github.com/utaal/infinity-rust

an idiomatic, safe Rust wrapper of Infinity

doc.rust-lang.org/nomicon

The dark arts of advanced and unsafe Rust programming