



# Heterogeneous Communication API

Free and Open Source: Apache 2.0 License

<https://github.com/michaelboth/Takyon>

*Unification of RDMA, Sockets, and More...*

*Made Easy...*

*Without Compromising Performance*

# Takyon is a Message Passing API

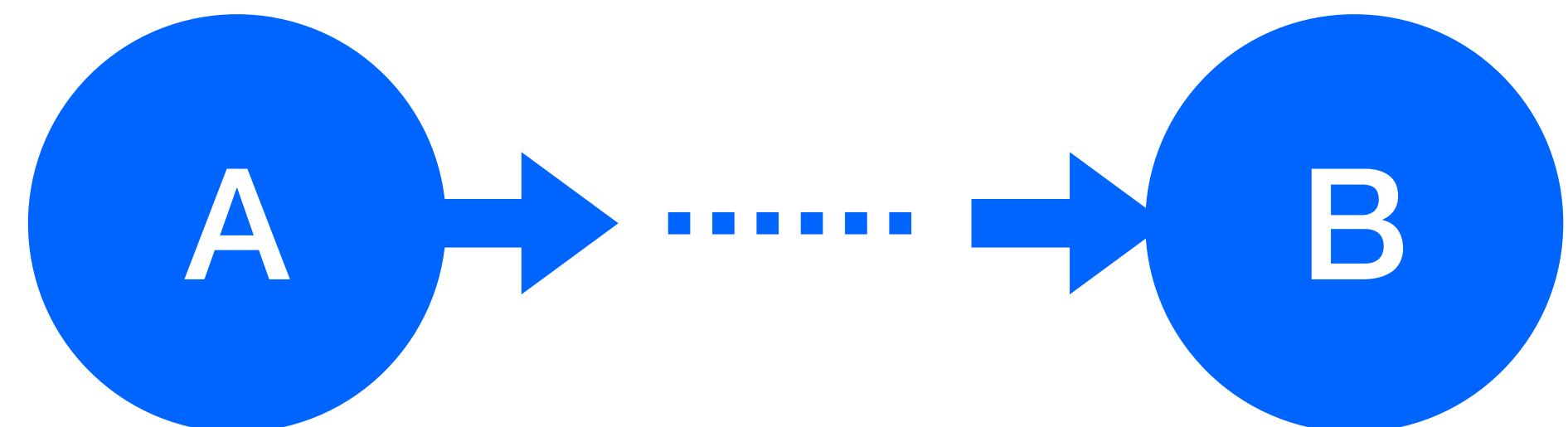
Based on almost 30 years of experience with heterogeneous systems

## Connected / Reliable



- Every byte matters
- E.g. downloading, distributed computation

## Unconnected / Unreliable



- Unicast & multicast packet streaming
- Packets may drop, come out of order, or be duplicated
- E.g. audio, video, sensors

# Takyon's Intended Audience



- HPC (High Performance Computing)
  - Dozens to millions of compute nodes
  - Mostly homogeneous
  - Collective communication is critical
  - Experts **ARE** focused on communication
  - E.g. Pixar render farms, physics simulation

HPC is better suited by  
MPI or libFabric



- eHPC (embedded HPC)
  - Edge computing, where **SWaP** may be critical
  - Small distributed system (< 100 nodes)
  - Heterogeneous; not a lot of symmetry
  - May real-time real-time, fault tolerance, certification
  - Experts are **NOT** focused on communication
  - E.g. Mil/Aero platforms, autonomous vehicles

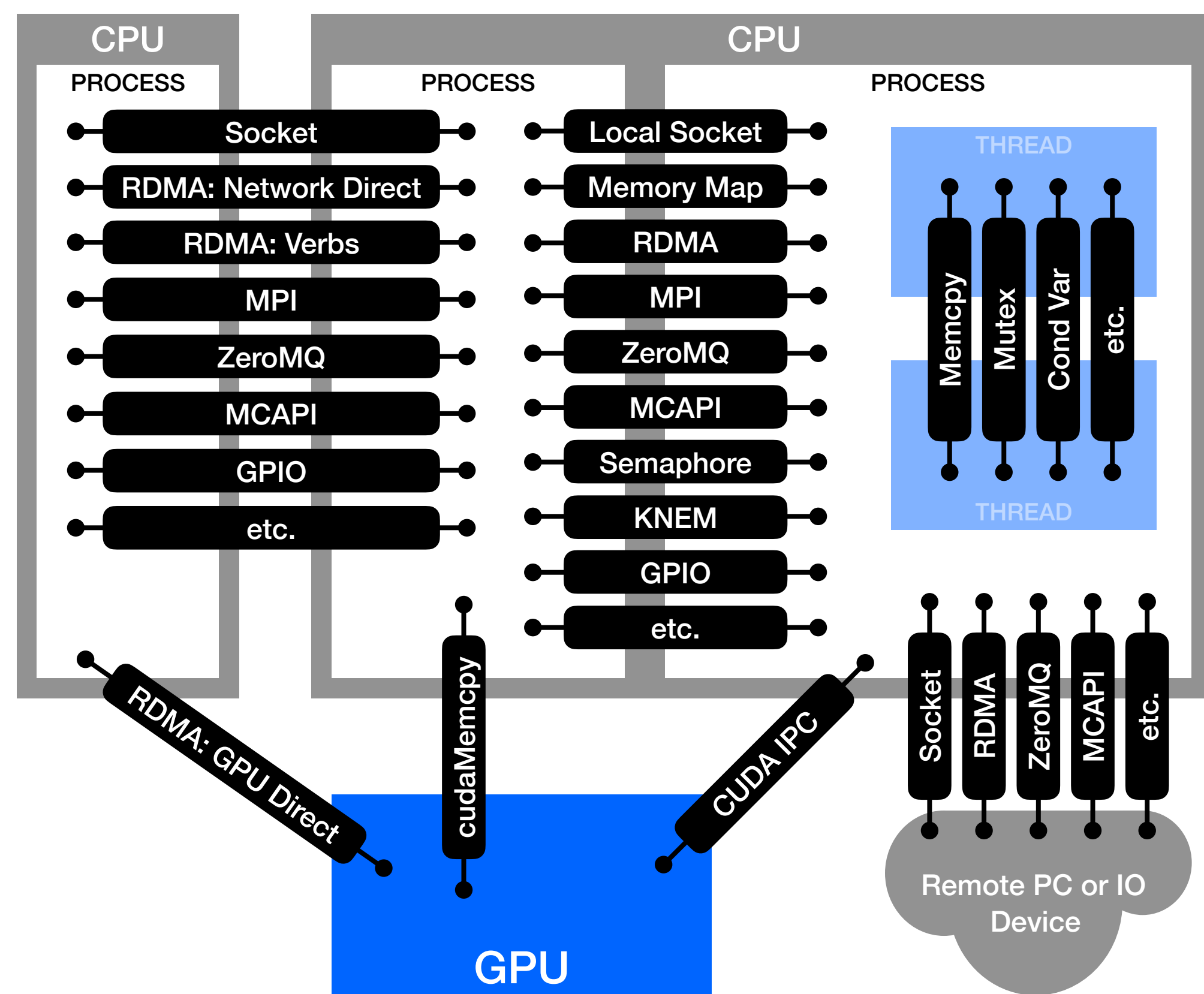
# Why Create Another API?

Within eHPC, communication is generally NOT a primary focus

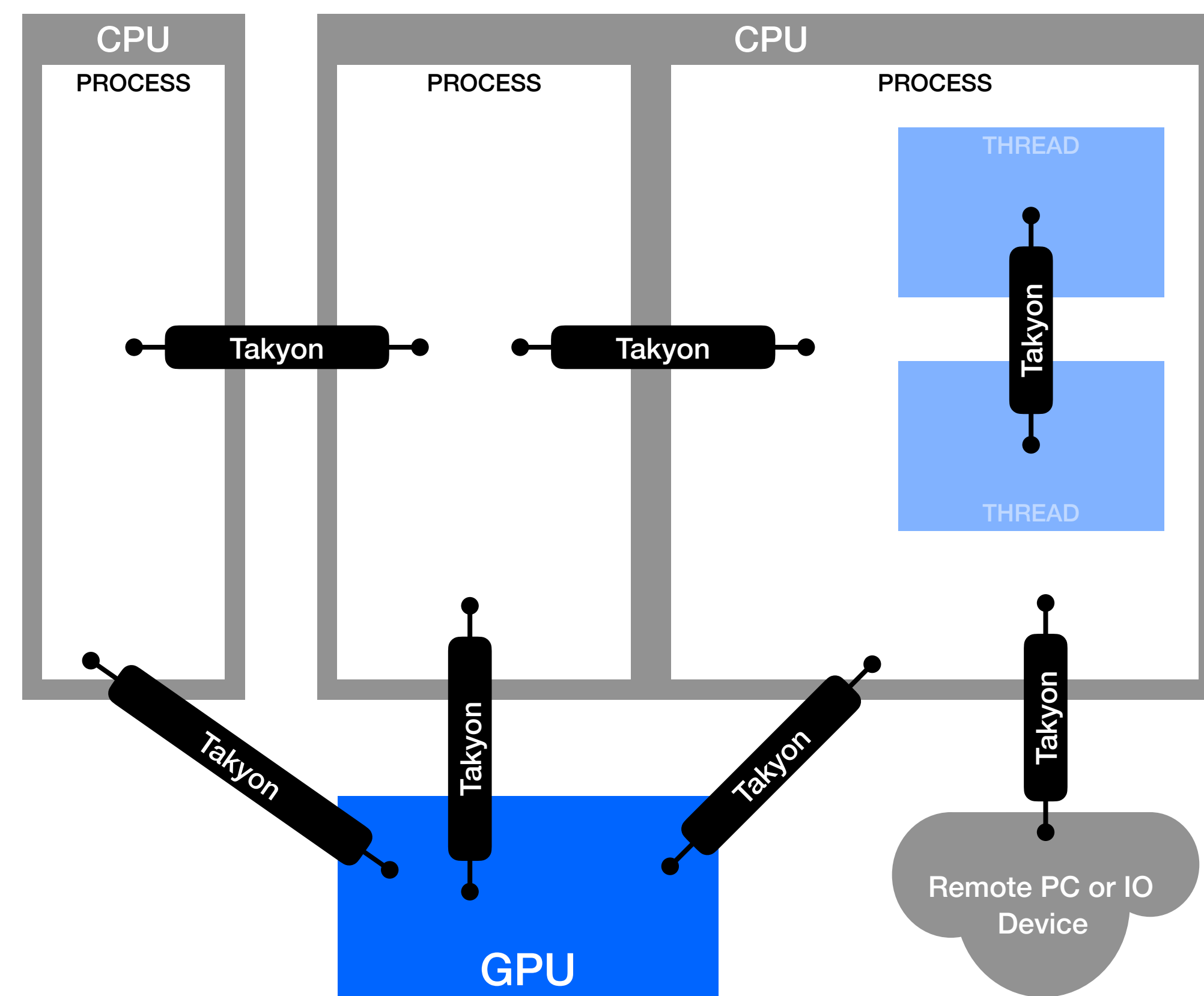
- Primary expertise is typically focused on a domain specific technology:
  - Signal Processing, AI & Deep Neural Networks, Gaming, etc.
- System may be heterogeneous, requiring more than one communication API:
  - May contain many variations of compute nodes and sensors
  - E.g. sub systems of a military jet
- A lack of a good heterogeneous communication API commonly leads to various issues:
  - Insufficient time to become an expert with a complex standard
  - Choosing a less capable but easier API
  - Poor performing software that is difficult to maintain
  - Increased hardware and cost to compensate for reduced performance

# Problem #1 Solved: A One-Stop Shop

## Before Takyon



## With Takyon



No single solution to fit all scenarios



# Problem #2 Solved: No Compromised Features

## Comparing with common open standard APIs

Feature	Sockets and similar: MCAPI, ZeroMQ	RDMA (OFA Verbs, Network Direct)	OFA's libFabrics	MPI	Takyon
Reliable and Unreliable	Yes	Yes	Yes		Yes
Communication to external apps, sensors, and other IO devices	Yes	Yes	?		Yes
Explicit fault tolerant hooks (timeouts, disconnect detection, create/destroy paths on the fly)	Yes	Yes	?		Yes
Deterministic: avoids implicit communication and allocations	Yes	Yes	Yes		Yes
Includes Inter-thread communication			Yes		Yes
Non blocking transfers		Yes	Yes	Yes	Yes
One way read/write: no involvement from remote endpoint		Yes	Yes	Yes	Yes
GPU support		Yes	Yes	Yes	Yes
Multiple memory blocks per message		Yes	?	Yes	Yes
Memory pre-registered before transfer		Yes	Yes	Partial	Yes
Zero copy and one-way (i.e. no implicit round trip)		Yes	?		Yes
32bit piggy back message with main message		Yes	?		Yes

 All features are common with eHPC

# Problem #3 Solved: Keep it Simple and Intuitive

Comparing with common open standard APIs

API	Function Count	Typical Drawbacks
Sockets	~20	<ul style="list-style-type: none"> <li>• Confusing naming and concepts</li> <li>• May not have required performance due to OS involvement</li> </ul>
RDMA (OFA Verbs, Network Direct)	~100	<ul style="list-style-type: none"> <li>• Very confusing naming and concepts</li> <li>• Experts are very rare</li> <li>• Development will be painfully slow or not time feasible</li> </ul>
OFA's libFabrics	~100	<ul style="list-style-type: none"> <li>• Confusing naming and concepts</li> <li>• Overwhelming learning curve for eHPC developers</li> </ul>
MPI	~300	<ul style="list-style-type: none"> <li>• Very feature limited (see previous slide) and may not be feasible</li> <li>• Difficult to understand appropriate transfer model</li> <li>• 'mpirun' is not portable and very confusing to tune</li> </ul>
Takyon	8	

**SHOUT-OUT:** MPI and libFabrics are great APIs, just not practical for eHPC

## Problem #4 Solved: Enable Best Performance

- Pre-register transport memory when the path is created
  - This is expensive and should not be done at transfer time
  - If the path is connected, then endpoints share memory registrations to allow for one-sided
- Pre-post receive requests
  - This makes sure there is no delay or implicit buffer needed when sending
- The above bullets allow for: zero copy, and one-way
  - No round trip needed to coordinate transfers (one-sided or two-sided)
- Non-Blocking
  - Offload transfer from the CPU and OS to allow for efficient concurrent processing and IO

**NOTE:** Not all interconnects support the above, but Takyon's abstraction does not inhibit the interconnects that do support the above.



# Takyon API

Only 8 Functions!!!

Function	Description
takyonCreate()	Create one endpoint of a communication path
takyonDestroy()	Destroy the endpoint
takyonSend()	Start sending a message If the communication does not support non-blocking then this will block
takyonIsSent()	Complete a non-blocking send
takyonPostRecvs()	If supported, pre-post a list of recv requests before the sender starts sending
takyonIsRecved()	Block until a message arrives
takyonOneSided()	Start a one sided message transfer (read or write)
takyonIsOneSidedDone()	Complete a non-blocking one-sided transfer

All interconnects (RDMA, sockets, etc.) use the same functions

# Defining the Interconnect via a Takyon Provider

All providers are defined in a text string passed to takyonCreate()

Locality	Example Takyon Providers
Inter-Thread	"InterThread -pathID=<non_negative_integer>"
Inter-Process	"InterProcess -pathID=<non_negative_integer>" "SocketTcp -local -pathID=<non_negative_integer>"
Inter-Processor	"SocketTcp -client -remoteIP=<ip_addr> -port=<number>" "SocketTcp -server -localIP=<ip_addr> Any -port=<number> [-reuse]"  "SocketUdpSend -multicast -localIP=<ip_addr> -groupIP=<multicast_ip> -port=<number> [-noLoopback] [-TTL=<time_to_live>]" "SocketUdpRecv -multicast -localIP=<ip_addr> -groupIP=<multicast_ip> -port=<number> [-reuse] [-rcvbuf=<bytes>]"  "RdmaRC -client -remoteIP=<ip_addr> -port=<number>" "RdmaRC -server -localIP=<ip_addr> Any -port=<number> [-reuse]"  "RdmaUC -client -remoteIP=<ip_addr> -port=<number> -IBport=<number>" "RdmaUC -server -localIP=<ip_addr> Any -port=<number> [-reuse] -IBport=<number>"  "RdmaUDSend -multicast -localIP=<ip_addr> -groupIP=<multicast_ip>" "RdmaUDRecv -multicast -localIP=<ip_addr> -groupIP=<multicast_ip>"

No limit to the Provider possibilities: GPIO, sensors, FPGAs, etc.

# Order of Messages with Send and Recv

The sender does not choose which destination buffer the message is sent to, but instead it's based on one of:

- If the Provider does not support `takyonPostRecv()`, then the order of received messages is defined by the order of calling `takyonIsRecv()`
- If the Provider supports `takyonPostRecv()`, then the order of received messages is defined by the order of posting receives regardless of the order of `takyonIsRecv()`

The order of the sent message is based on one of:

- If the Provider is 'reliable' then the messages will arrive in the same order as sent
- If the Provider is 'unreliable' then the messages may:
  - Arrive in a different order than sent
  - Be dropped and lost forever
  - Be duplicate of a previously arrived message

# Blocking versus Non-Blocking Transfers

## Blocking

Starting a transfer and waiting for the transfer to complete is a single function call

Examples:

`send()` - blocks until complete

`recv()` - blocks until complete

Note: `recv()` can be called before `send()`

## Non-Blocking

Starting a transfer and waiting for the transfer to complete are separate function calls

Examples:

`sendStart()` - starts transfer

`isSent()` - block until complete

`postRecv()` - provides a place to recv data ahead of time

`isRecv()` - blocks until complete

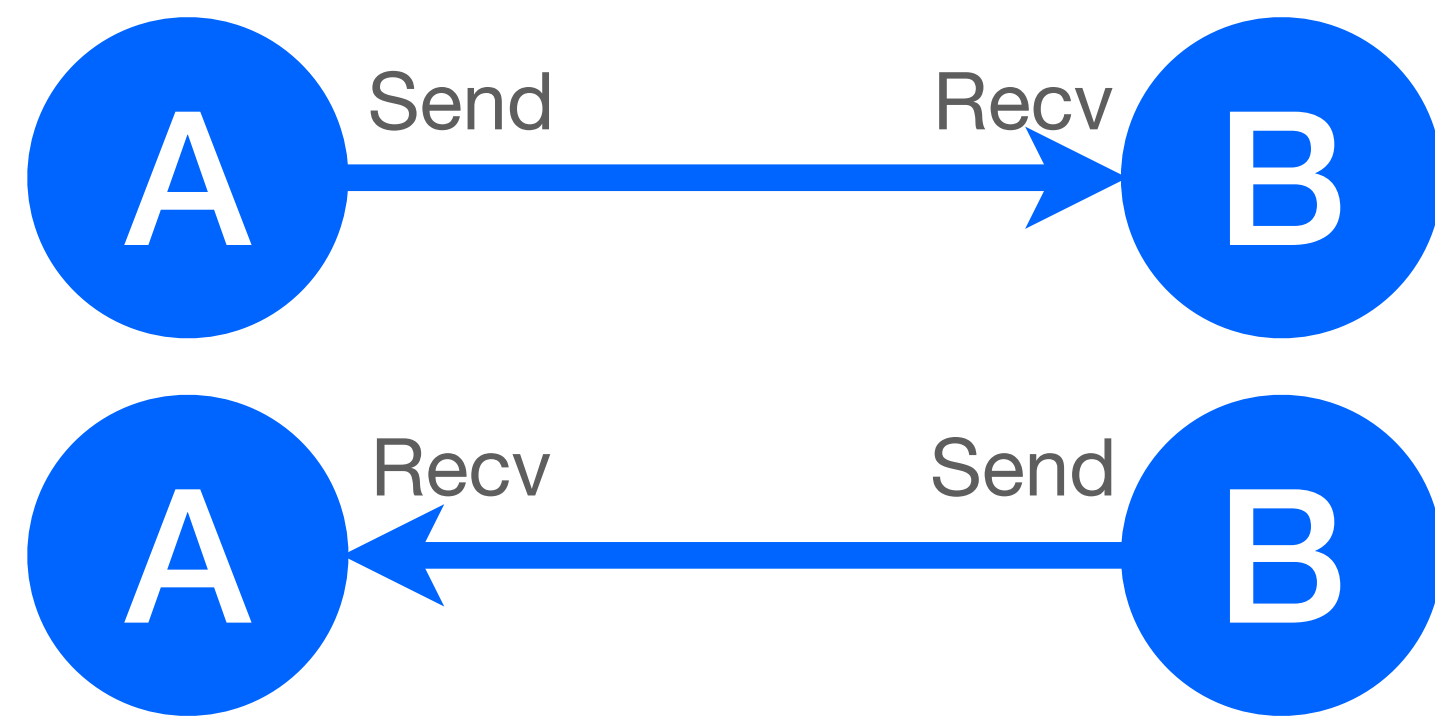
Critical:

- Recv must be posted before send starts
- Don't modify send side data while transfer is in progress
- Don't re-post receive until receive data is processed

Some interconnects only allow blocking

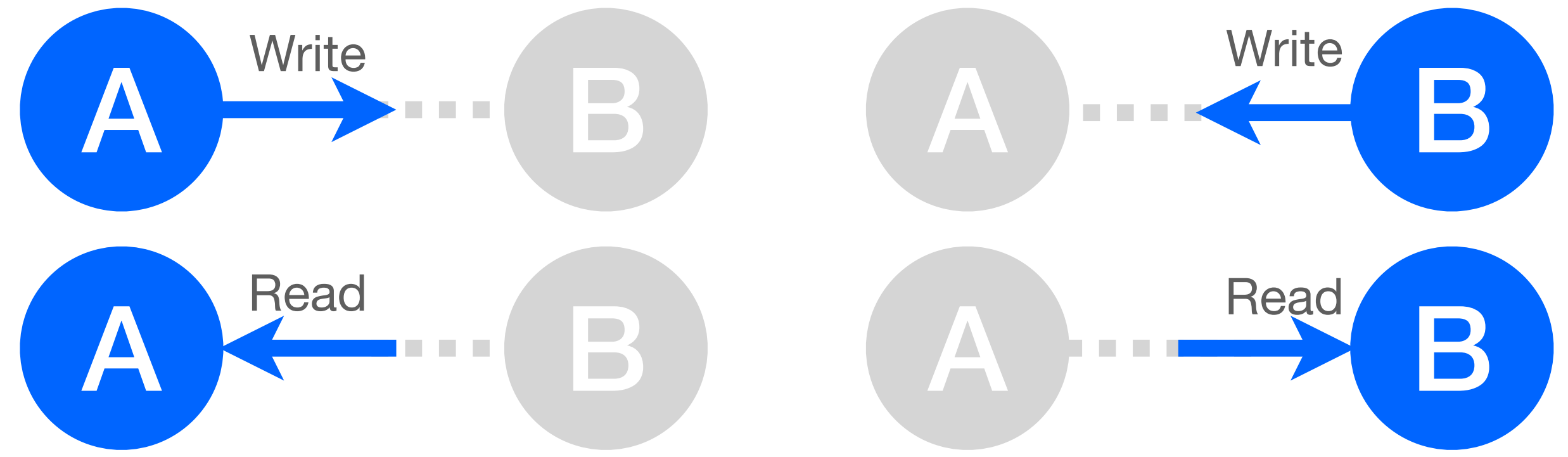
# Two-Sided versus One-Sided Transfers

## Two-Sided



- Both endpoints are involved with a coordinated send and recv

## One-Sided



- Only one endpoint is involved
- Must be a connected interconnect so the remote endpoint's transport memory is known

Some interconnects only allow one or the other

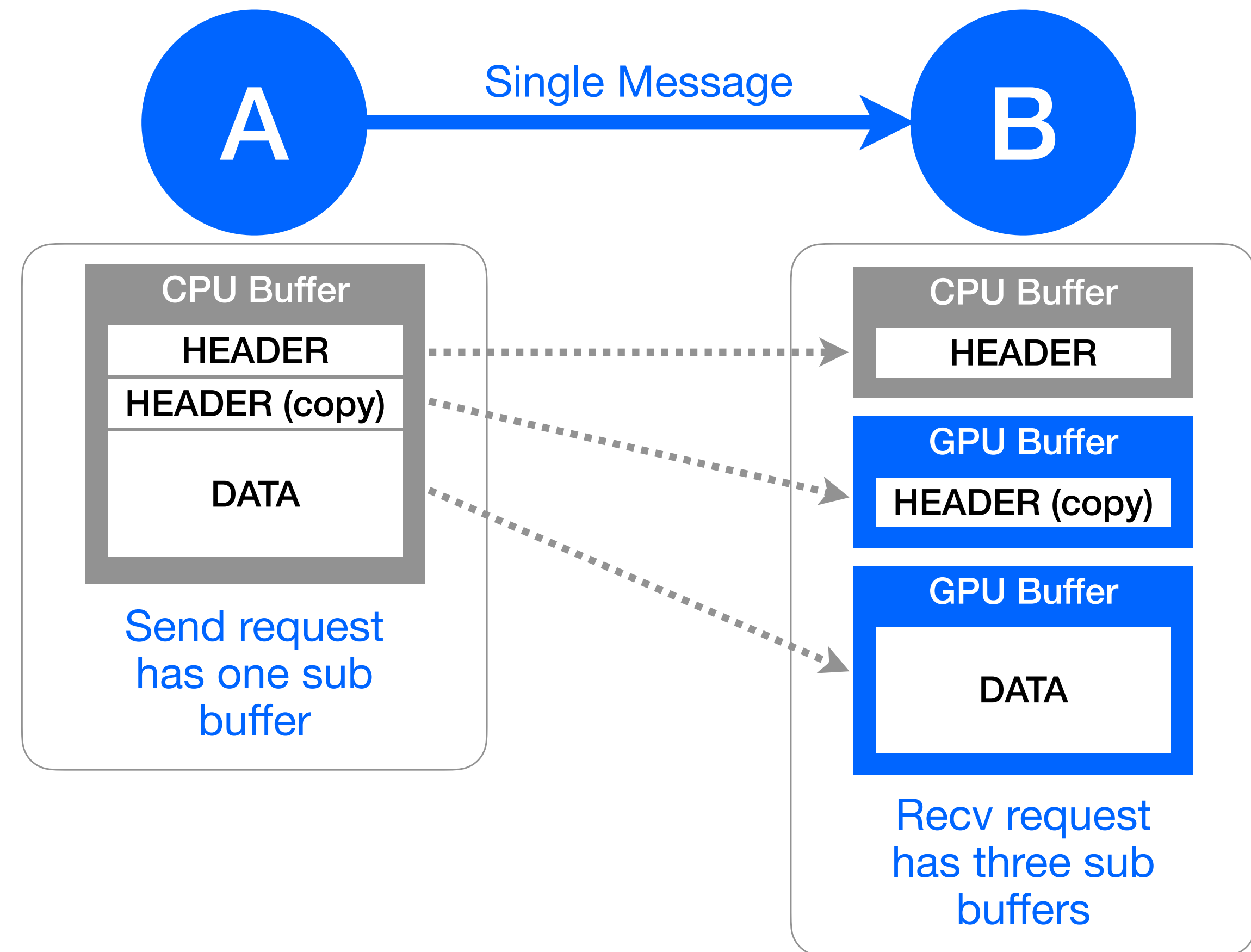


# Single Message, Multiple Buffers

Multiple buffers may allow for highly optimized processing

Hypothetical Example:

- It's common for GPUs to do heavy processing and CPU does light book keeping, but both need to know the attributes of the data



Some interconnects only allow one buffer per message

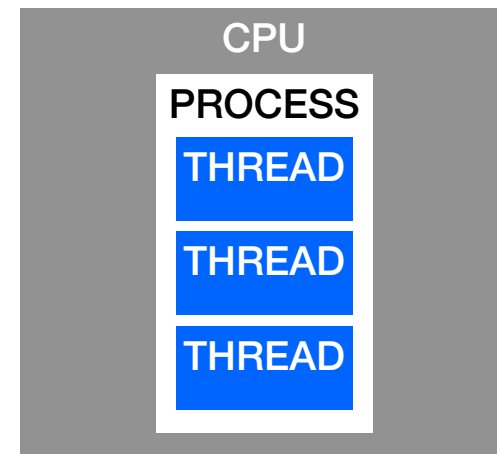
# Fault Tolerant Communication

## Detecting and handling degraded communication paths

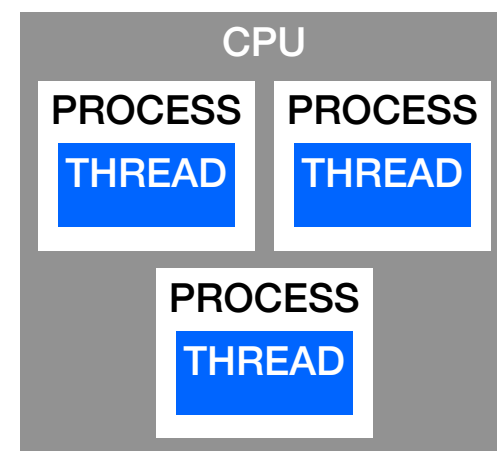
- Detecting degraded communication
  - Disconnect detection
  - Timeouts
- Handling degraded communication
  - Destroy and re-create path using the same Provider
  - Use an alternative Provider and path
- Communication API Requirements
  - Communication API should provided the hooks for fault tolerance
  - Only the app can know what to do when communication degrades
  - Communication paths should be independent of each other (“One light goes out they all go out”)

**DISTINCTION:** Takyon is not fault tolerant (and it shouldn't be), but does provide fault tolerant hooks

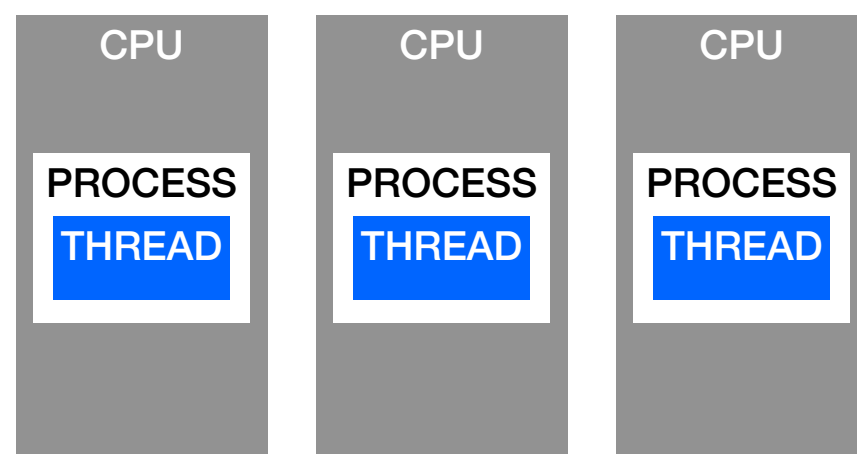
# Takyon Enables Accelerating Development by Staging



1. Start with one process and multiple threads
  - While dataflow is being developed, only need to run a single executable
  - Easier to debug crashes or validate memory leaks/overwrites (e.g. valgrind)



2. Move to multiple processes on one CPU
  - Simple way to validate the migration of dataflow, without jumping to multi-processors



3. Move to multiple processors
  - Migration should be simple
  - Can now test for performance

Could also have a blend of inter-thread, inter-process, and inter-processor

# Looking to the Future

## Possible Enhancements

- Atomics
- Strided Transfers
  - Currently avoiding this since common interconnects don't support this
- Publish/Subscribe
  - A potential replacement for the overly complex DDS
  - Could have simplified participants, publishers, subscribers, and QoS
  - Make messages opaque and private (removes need for DDS's intermediate language)
- Collectives: barrier, scatter, gather, all-to-all, reduce, etc.
  - Already done as a separate API with Takyon 1.x, and I will convert it to Takyon 2.x
  - Create a complimenting GUI to build and maintain the collective groups visually

**CHALLENGE:** Is Takyon missing a key feature?



# Ultimate Goal: It's Almost an Open Standard



The creation of Takyon inspired a Khronos exploratory group that determined the industry is in need of a better Heterogeneous Communication API:

<https://www.khronos.org/exploratory/heterogeneous-communication/>

Takyon is currently the only proposal and is waiting for an industry quorum (4 or 5 unique companies) to move forward with the working group that will formalize the specification. **It's very difficult finding the expertise who can contribute.**

**NEED HELP:** Join the Khronos Working Group to help get the specification moving forward