Heterogeneous Communication API For eHPC (embedded High Performance Computing)

https://github.com/michaelboth/Takyon

Michael Both: Oct 6 2022

Takyon is a Message Passing API

Based on almost 30 years of experience…

Focused on eHPC & SWaP (size, weight & power)…

Unification of RDMA, Sockets, and More…

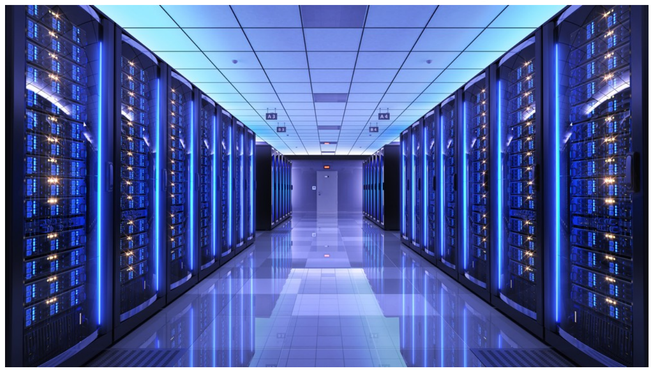
Without Compromising Performance…

Made Easy!

Takyon's Intended Audience

HPC is better suited by MPI or libFabric

- 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 farm, physics simulation



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

Why Introduce Another Communication API?

Within eHPC, communication is generally NOT a primary focus

- Primary expertise is typically focused on a domain specific technology; signal processing, autonomous algorithms, AI & deep neural networks, gaming, etc.
- Developers don't want to be bogged down in learning or maintaining complex communication software

Lack of expertise results in four major issues:

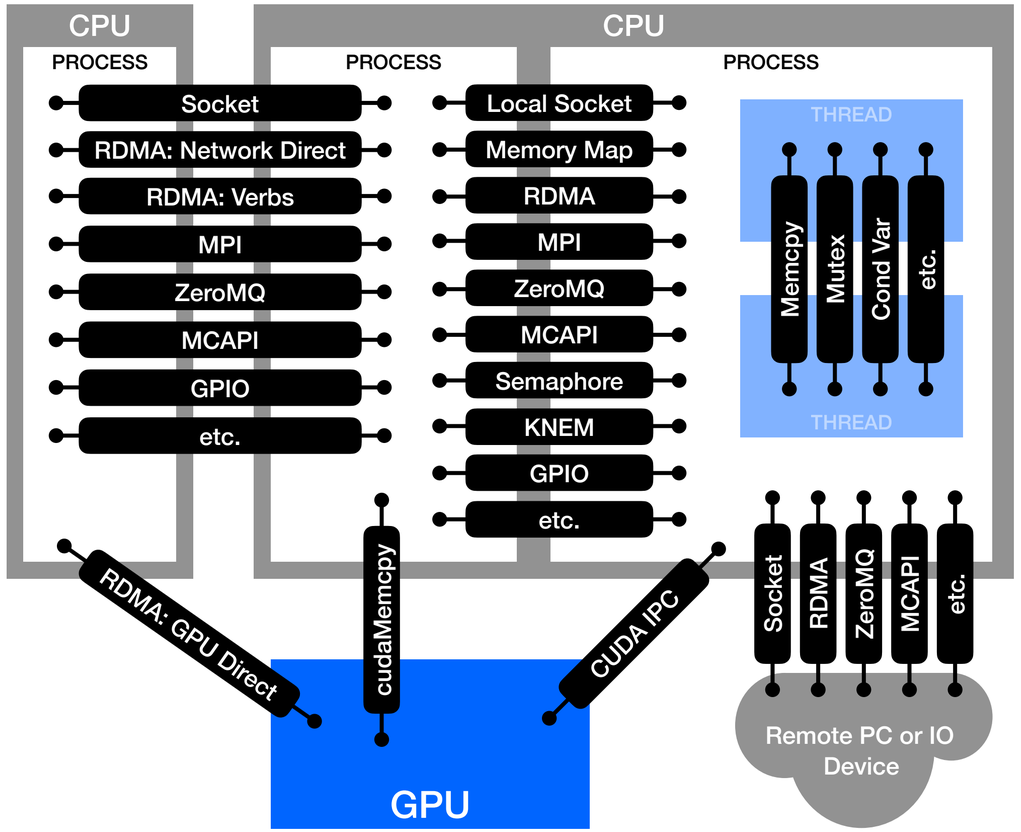
1. No single API works across all devices and localities
2. Easy-to-use APIs are not fully featured
3. Easy-to-use API reduce performance (latency, throughput, and determinism)
4. Fully featured APIs have an unrealistic learning curve

The Result:

- Poor performing software that is difficult to maintain
- Increased hardware, development, and maintenance costs

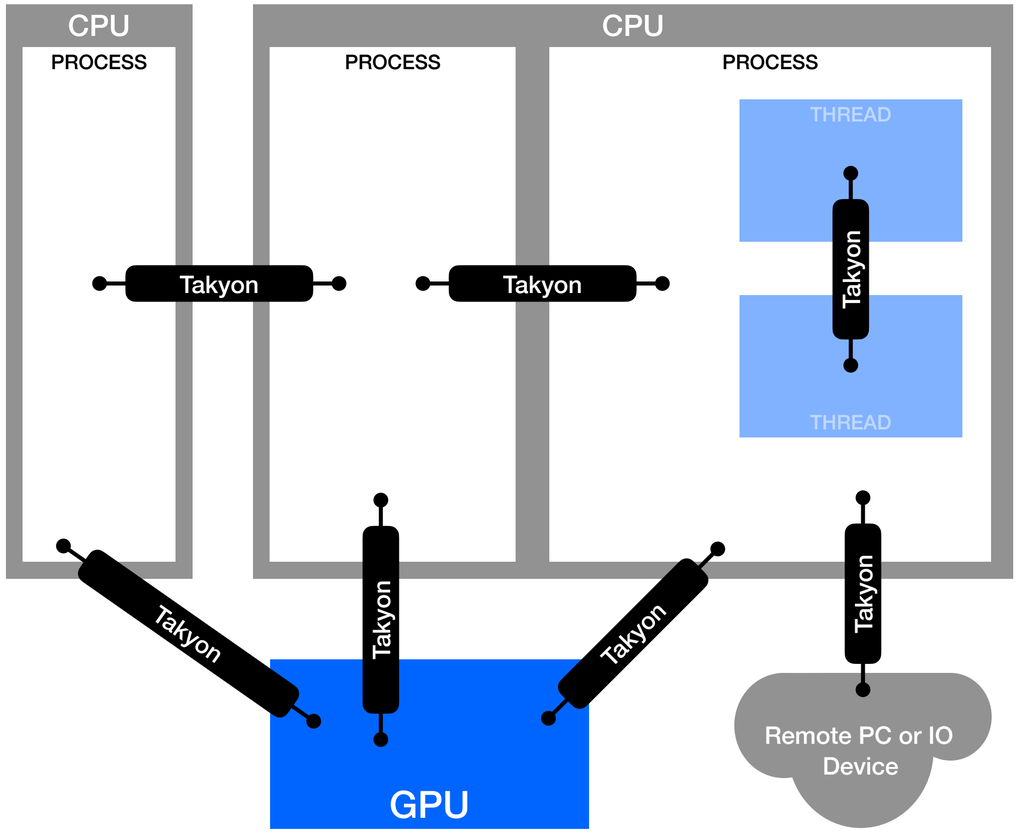
Problem #1 Solved: A One-Stop Shop

Before Takyon



No single solution to fit all devices and localities

With Takyon



Takyon

Problem #2 Solved: No Compromised Features

Comparing with common 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

API	Function Count	Typical Drawbacks
Sockets	~20	• Confusing naming and concepts • May not have required performance due to OS involvement
RDMA (OFA Verbs, Network Direct)	~100	• Very confusing naming and concepts • Experts are very rare • Development will be painfully slow or not time feasible
OFA's libFabrics	~100	• Confusing naming and concepts • Overwhelming learning curve for eHPC developers
MPI	~300	• Very feature limited (see previous slide) and may not be feasible • Difficult to understand appropriate transfer model • 'mpirun' is not portable and very confusing to tune
Takyon	8	

An eHPC development and maintenance nightmare

SHOUT-OUT: Sockets, Verbs, libFabrics, MPI, and many other comm APIs are fantastic for various industries, just not practical for eHPC

Problem #4 Solved: Enable Best Performance

Latency, Throughput, and Determinism

- Zero-copy, and one-way (no round trips!)
  - Achieved by:
    - Pre-registering (is time consuming) transport memory when the path is created
    - Pre-post receive requests
      - Creates a holding place for data to arrive later asynchronously in the background
      - This makes sure there is no delay or implicit buffering needed when sending
- Non-Blocking
  - Offload transfers from the CPU to allow for efficient concurrent processing and IO

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

Takyon API

Only 8 Functions!!!

Two-sided functions

One-sided 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> -rdmaDevice=<name> -rdmaPort=<number>” “**RdmaRC** -server -localIP=<ip_addr>|Any -port=<number> [-reuse] -rdmaDevice=<name> -rdmaPort=<number>” “**RdmaUC** -client -remoteIP=<ip_addr> -port=<number> -rdmaDevice=<name> -rdmaPort=<number>” “**RdmaUC** -server -localIP=<ip_addr>|Any -port=<number> [-reuse] -rdmaDevice=<name> -rdmaPort=<number>” “**RdmaUDMulticastSend** -localIP=<ip_addr> -groupIP=<multicast_ip>” “**RdmaUDMulticastRecv** -localIP=<ip_addr> -groupIP=<multicast_ip>”

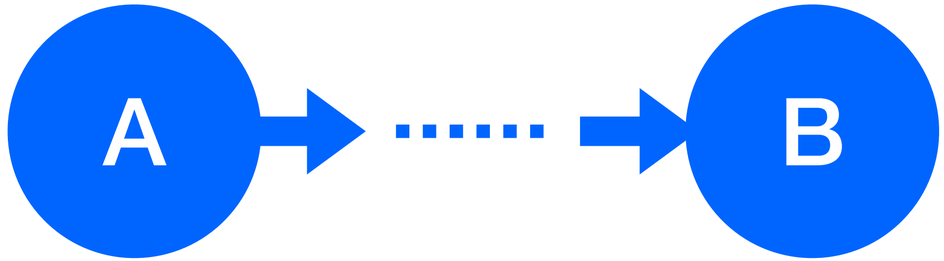
No limit to the Provider possibilities: GPIO, sensors, FPGAs, etc.

Takyon Supports Reliable and Unreliable Transfers

Reliable

- **Every byte matters**
- Can support very large messages
- E.g. Distributed computation

Unreliable



- **Messages may drop, come out of order, or be duplicated**
- Unicast and multicast
- Usually small messages
- E.g. audio, video, lidar, radar

Transport Memory

Takyon does NOT allocate transport memory (by design)

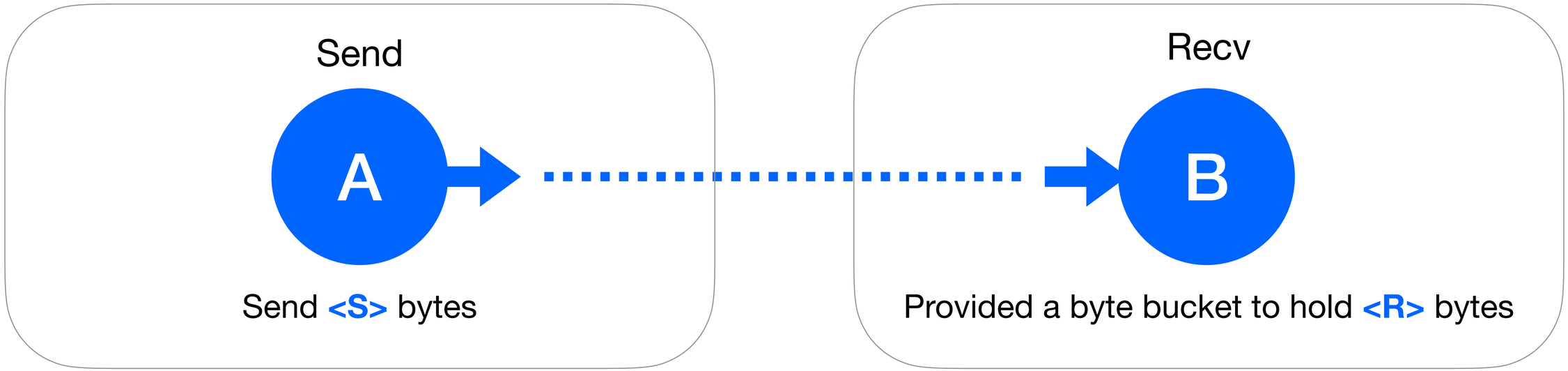
Transport memory may need to be shared between communication paths (Takyon or 3rd party) and other processing APIs and IO devices

Therefore it is logical to have the application organize all transport memory

- CPU (local or memory map)
- GPU (CUDA)
- Sensor/FPGA/etc. memory

And then provide pointers to the appropriate APIs as needed

Send -> Recv Message Bytes



<S> <= **<R>**

I.e. sent messages can be smaller than what
takyon's recv request provided as a byte bucket

Send -> Recv Message Order

Rule 1: The order of arriving messages 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 a duplicate of a previously arrived message

Rule 2: The arrived message is put into the recv buffer based on:

- takyonPostRecvs() IS supported: the order of received messages is defined by the order of posting recv requests regardless of the order of takyonIsRecved()
- takyonPostRecvs() NOT supported: the order of received messages is defined by the order of calling takyonIsRecved()
- FYI: Sender does not decide on recv order

Blocking versus Non-Blocking Transfers

Providers only support one or the other

Blocking

CPU is usually involved with full transfer

takyonSend() - blocks until complete

takyonIsRecved() - blocks until complete

Note: send() can be called before recv(), and send() will block waiting for recv() to be called

Non-Blocking

CPU is usually NOT involved with transfer

takyonSend() - starts transfer

takyonIsSent() - block until complete

Don't modify message at this point

Recv requests must be posted before send is called, or messages might be dropped (unreliable), or cause a failure (reliable)

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

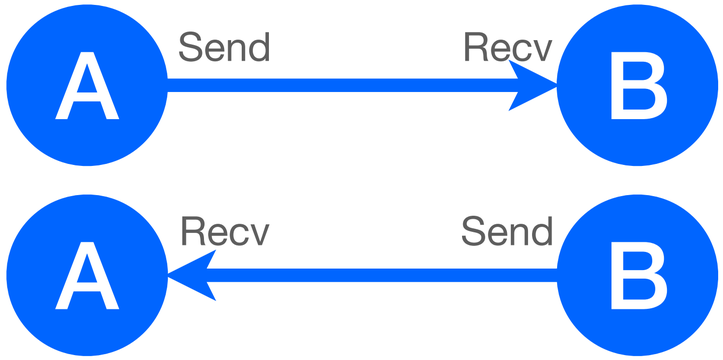
takyonIsRecved() - blocks until complete

Process the message before re-posting to avoid overwriting data from a newly arriving message

Two-Sided versus One-Sided Transfers

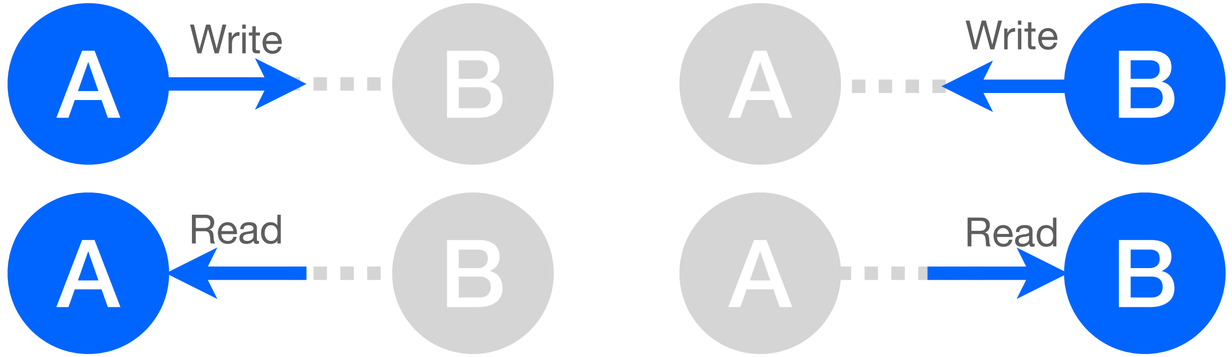
Some Providers only allow one or the other

Two-Sided



- Both endpoints are involved with a coordinated send() and recv()
- Unicast and Multicast are two-sided

One-Sided



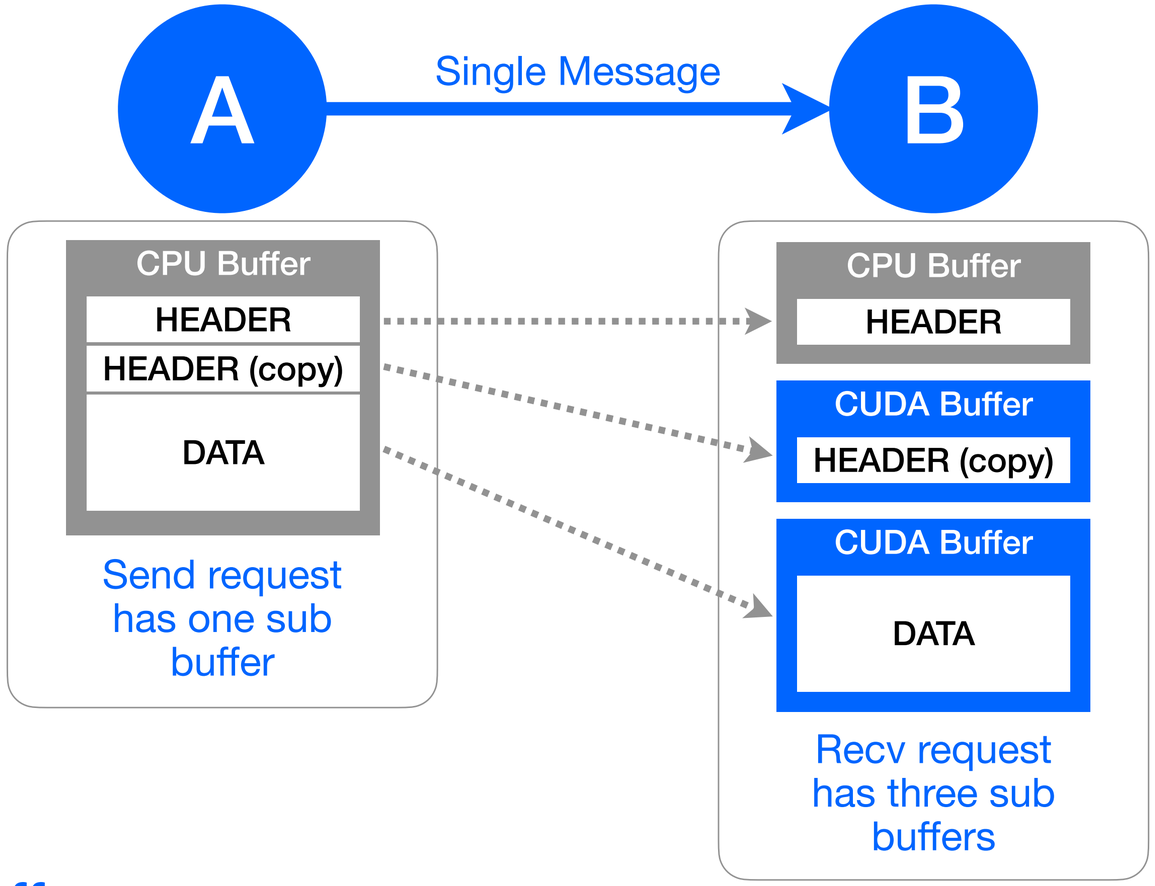
- Only one endpoint is involved with the transfer
- Can support both read and write

Single Message, Multiple Buffers

Multiple buffers may allow for highly organized and 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



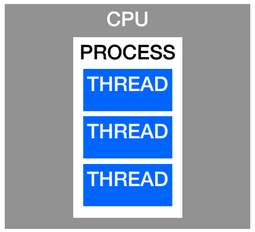
Some providers only allow one buffer per message

Fault Tolerant Communication

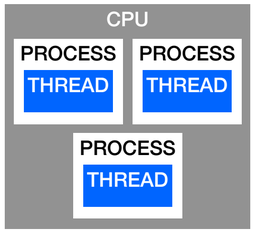
- Degraded communication is detected via:
  - Disconnect detection (e.g. network is down)
  - Timeouts (the transfer is not occurring in a reasonable amount of time)
- Handle degraded communication via:
  - Dynamic path destruction/creation, without effecting other existing paths
  - Some other application defined alternative
- Notes about being fault tolerant
  - 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

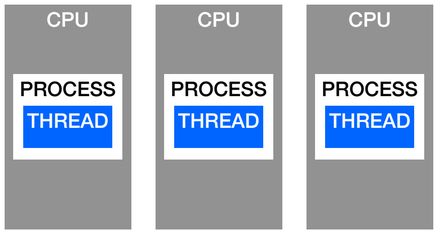
Accelerate Development by Locality Staging



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 deployment performance

All Takyon examples support this now; **even with CUDA memory**

Looking to the Future

Posible 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 may 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

KHRONOS® GROUP

CONNECTING SOFTWARE TO SILICON

The creation of Takyon inspired a Khronos exploratory group that determined the industry is in need of an eHPC 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.

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