



Simple, flexible, and high performance communication

Takyon is a point to point communication API for C/C++ applications. It's designed for software engineers, in the embedded HPC (High Performance Computing) field, developing complex, multi-threaded applications for heterogeneous compute architectures.

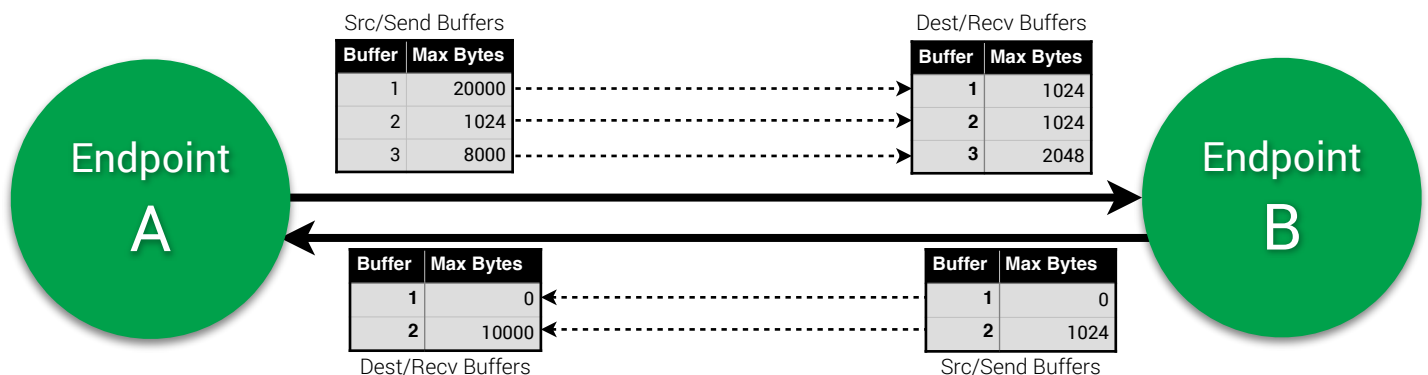
Takyon is designed to be:

- **Efficient:** A thin wrapper over lower level communication APIs that still maintains critical features.
- **High Performance:** one-way (single underlying transfer), zero-copy (no extra buffering), two-sided transfers (recv gets notification when data arrives).
- **Portable:** Can work with most any modern interconnect, OS, bit width, and endianness.
- **Unified:** Same API for any locality (inter-thread/process/processor/application) and point to point model (reliable, connectionless, IO devices).
- **Dynamic & Fault Tolerant:** Create and destroy paths as needed. Timeouts can be used at any stage. Broken connections can be detected by application and restarted by the application without effecting other connections.
- **Scalable:** No inherent limit to the number of connections, connection patterns (mesh, tree, etc.) or physical distance between connections.
- **Simple to Use:** While low level communications APIs may require weeks or months to learn, Takyon can be understood in a few days.

An application using the core Takyon APIs needs to include "takyon.h"

Takyon Path Details

Connected interconnects: Support reliable, bi-directional, and multi-buffering. Good for distributed algorithms where every byte matters.



Connectionless interconnects: Support unreliable datagram transfers: Good for live-streaming, multicasting, and IO devices where data could be dropped.



Transfer Details

The following must be specified when sending data (all are in bytes): size, source offset, and destination offset. The data size can be less than or equal to the max size of the buffer. The source and destination offset do not need to be the same. The destination offset must be zero for connectionless interconnects.





Create a Path

First fill in the fill in the following fields of the Takyon attributes structure to define the endpoint of the path.

TakyonPathAttributes Fields	Description
bool is_endpointA	true = Endpoint A. false = Endpoint B.
bool is_polling	true = Polling. false = Event driven.
bool abort_on_failure	true = Abort on error. false = Return status.
uint64_t verbosity	Or' the bits of the Takyon verbosity flag values.
TakyonCompletionMethod send_completion_method	One of: TAKYON_BLOCKING TAKYON_USE_SEND_TEST (i.e. send is non blocking)
TakyonCompletionMethod recv_completion_method	Must be TAKYON_BLOCKING
double path_create_timeout	Timeout to wait for takyonCreate() to complete
double send_start_timeout	Timeout to wait for takyonSend() to start transfer
double send_finished_timeout	Timeout to wait for send to complete transferring
double recv_start_timeout	Timeout to wait for takyonRecv() to start getting message
double recv_finished_timeout	Timeout to wait for takyonRecv() to get complete message
double path_destroy_timeout	Timeout to wait for takyonDestroy() to disconnect
int nbufs_AtoB	Number of buffers from endpoint A to B.
int nbufs_BtoA	Number of buffers from endpoint B to A.
uint64_t *sender_max_bytes_list	List of buffer sizes for the endpoint. Zero or greater.
uint64_t *receiver_max_bytes_list	List of buffer sizes for the endpoint. Zero or greater.
size_t *sender_addr_list	List of pre-allocated send buffers. Set each entry to NULL to auto-allocate.
size_t *receiver_addr_list	List of pre-allocated receive buffers. Set each entry to NULL to auto-allocate.
char interconnect[TAKYON_MAX_INTERCONNECT_CHARS]	Defines the interconnect and its properties to be used for transferring. The list of supported interconnects depends on the implementation

Verbosity flags	Value
TAKYON_VERBOSITY_NONE	0x00
TAKYON_VERBOSITY_ERRORS	0x01
TAKYON_VERBOSITY_CREATE_DESTROY	0x02
TAKYON_VERBOSITY_CREATE_DESTROY_MORE	0x04
TAKYON_VERBOSITY_SEND_RECV	0x08
TAKYON_VERBOSITY_SEND_RECV_MORE	0x10

Timeout values	Value
TAKYON_NO_WAIT	0
TAKYON_WAIT_FOREVER	-1
A double value, representing seconds: 1.25 - One and one quarter seconds 0.001 - One millisecond 0.00002 - 20 microseconds	0 or greater

Standard Interconnects	
InterThreadMemcpy	Socket
InterThreadPointer	OneSidedSocket
InterProcessMemcpy	UnicastSendSocket
InterProcessPointer	UnicastRecvSocket
InterProcessSocket	MulticastSendSocket MulticastRecvSocket

Then create the path's endpoint. If the other endpoint is also a Takyon endpoint, then takyonCreate() will also need to be called for that endpoint.

```
TakyonPath *takyonCreate(TakyonPathAttributes *attributes)
```

Send Message

Start sending a contiguous message (blocking and non-blocking):

```
bool takyonSend(TakyonPath *path, int buffer_index, uint64_t bytes, uint64_t src_offset, uint64_t dest_offset, bool *timed_out_ret)
```

Test if a non-blocking send is complete (only call this if path's send_completion_method is TAKYON_USE_SEND_TEST):

```
bool takyonIsSendFinished(TakyonPath *path, int buffer_index, bool *timed_out_ret)
```

Receive Message

Receive a message:

```
bool takyonRecv(TakyonPath *path, int buffer_index, uint64_t *bytes_ret, uint64_t *offset_ret, bool *timed_out_ret)
```

Destroy a Path

Destroy the path's endpoint.

```
char *takyonDestroy(TakyonPath **path_ret)
```



Hello World (multi-threaded, bi-directional transfers)

```
#include "takyon.h"
#include <stdio.h>
#include <string.h>
#include <pthread.h>

static const char *interconnect = NULL;

static void *hello_thread(void *user_data) {
    bool is_endpointA = (user_data != NULL);

    TakyonPathAttributes attrs;
    attrs.is_endpointA = is_endpointA;
    attrs.is_polling = false;
    attrs.abort_on_failure = true;
    attrs.verbosity = TAKYON_VERBOSITY_ERRORS;
    strncpy(attrs.interconnect, interconnect, TAKYON_MAX_INTERCONNECT_CHARS);
    attrs.path_create_timeout = TAKYON_WAIT_FOREVER;
    attrs.send_start_timeout = TAKYON_WAIT_FOREVER;
    attrs.send_finish_timeout = TAKYON_WAIT_FOREVER;
    attrs.recv_start_timeout = TAKYON_WAIT_FOREVER;
    attrs.recv_finish_timeout = TAKYON_WAIT_FOREVER;
    attrs.path_destroy_timeout = TAKYON_WAIT_FOREVER;
    attrs.send_completion_method = TAKYON_BLOCKING;
    attrs.recv_completion_method = TAKYON_BLOCKING;
    attrs.nbufs_AtoB = 1;
    attrs.nbufs_BtoA = 1;
    uint64_t sender_max_bytes_list[1] = { 1024 };
    attrs.sender_max_bytes_list = sender_max_bytes_list;
    uint64_t recver_max_bytes_list[1] = { 1024 };
    attrs.recver_max_bytes_list = recver_max_bytes_list;
    size_t sender_addr_list[1] = { 0 };
    attrs.sender_addr_list = sender_addr_list;
    size_t recver_addr_list[1] = { 0 };
    attrs.recver_addr_list = recver_addr_list;

    TakyonPath *path = takyonCreate(&attrs);

    const char *message = is_endpointA ? "Hello from endpoint A" : "Hello from endpoint B";
    for (int i=0; i<5; i++) {
        if (is_endpointA) {
            strncpy((char *)path->attrs.sender_addr_list[0], message, path->attrs.sender_max_bytes_list[0]);
            takyonSend(path, 0, strlen(message)+1, 0, 0, NULL);
            takyonRecv(path, 0, NULL, NULL, NULL);
            printf("Endpoint A received message %d: %s\n", i, (char *)path->attrs.recver_addr_list[0]);
        } else {
            takyonRecv(path, 0, NULL, NULL, NULL);
            printf("Endpoint B received message %d: %s\n", i, (char *)path->attrs.recver_addr_list[0]);
            strncpy((char *)path->attrs.sender_addr_list[0], message, path->attrs.sender_max_bytes_list[0]);
            takyonSend(path, 0, strlen(message)+1, 0, 0, NULL);
        }
    }

    takyonDestroy(&path);
    return NULL;
}

int main(int argc, char **argv) {
    if (argc != 2) { printf("usage: hello <interconnect>\n"); return 1; }
    interconnect = argv[1];
    pthread_t endpointA_thread_id;
    pthread_t endpointB_thread_id;
    pthread_create(&endpointA_thread_id, NULL, hello_thread, (void *)1LL);
    pthread_create(&endpointB_thread_id, NULL, hello_thread, NULL);
    pthread_join(endpointA_thread_id, NULL);
    pthread_join(endpointB_thread_id, NULL);
    return 0;
}
```

Takyon Header

Setup Path Attributes

This could be offloaded to a configuration file: either hand coded or generated from a data-flow design tool

Path Creation

Endpoint A Transfers

Endpoint B Transfers

Path Destruction

Run Endpoints A & B in Separate Threads

Notes:

- Endpoint A and endpoint B each do 5 sends and 5 receives, but notice how A sends first then B receives, then B is allowed to send then A receives. This is how application induced synchronization removes the need for Takyon to have implicit synchronization to manage correct use of the message buffers.
- No data addresses are passed to the takyonSend() or takyonRecv(). The message buffers are pre-allocated and registered via pathCreate() enabling Takyon to do one-way, zero-copy, two-side transfers. You just can't get faster than that.
- The calls to takyonRecv() use NULL to ignore 'bytes_received', 'dest_offset', and 'timed_out' since they are not needed for this application.



The following Takyon extension functions are provided as open source to be linked into the application as needed. These convenience APIs are provided as source code instead of libraries in order to let the developers duplicate and modify the source to best fit the application needs. It also has the added benefit of simplifying certification.

An application using the Takyon extension functions needs to:

- Include **"takyon_extensions.h"**, located in "Takyon/extensions/". All of the extension data structures are this header file.
- Compile and link the appropriate extension C files, located in "Takyon/extensions/"

Endian (takyon_endian.c)

Helpful functions if the endian of all threads are not the same. Swapping is in bytes.

```
bool takyonEndianIsBig();
void takyonEndianSwapUInt16(uint16_t *data, uint64_t num_elements)
void takyonEndianSwapUInt32(uint32_t *data, uint64_t num_elements)
void takyonEndianSwapUInt64(uint64_t *data, uint64_t num_elements)
```

Time (takyon_time.c)

Put the thread to sleep for the specified amount of time.

```
void takyonSleep(double seconds)
```

Returns current system wall clock time in seconds.

```
double takyonTime()
```

Named Memory Allocation (takyon_mmap.c)

Allocate named memory that can be accessed by remote processes in the same OS. This may be helpful with the collective gather operation when the 'Mmap' interconnect is used by one or more paths.

```
void takyonMmapAlloc(const char *map_name, uint64_t bytes, void **addr_ret, TakyonMmapHandle *mmap_handle_ret)
void takyonMmapFree(TakyonMmapHandle mmap_handle)
```

Path Attributes (takyon_attributes.c)

Simplify setting up the attributes for a Takyon path. Fields in the attribute structure can still be modified after initially calling takyonAllocAttributes()..

```
TakyonPathAttributes takyonAllocAttributes(bool is_endpointA, bool is_polling, int nbufs_AtoB, int nbufs_BtoA, uint64_t bytes, double
timeout, const char *interconnect)
void takyonFreeAttributes(TakyonPathAttributes attrs)
```

Collective Groups (takyon_collective.c)

Organize previously created Takyon paths into collective groups.

Barrier: (tree based, where each tree node can have any number of children)

```
extern TakyonCollectiveBarrier *takyonBarrierInit(int nchildren, TakyonPath *parent_path, TakyonPath **child_path_list)
extern void takyonBarrierRun(TakyonCollectiveBarrier *collective, int buffer)
extern void takyonBarrierFinalize(TakyonCollectiveBarrier *collective)
```

Reduce: (tree based, where each tree node can have any number of children)

```
TakyonCollectiveReduce *takyonReduceInit(int nchildren, TakyonPath *parent_path, TakyonPath **child_path_list)
void takyonReduceRoot(TakyonCollectiveReduce *collective, int buffer, uint64_t nelements, uint64_t bytes_per_elem,
void(*reduce_function)(uint64_t nelements,void *a,void *b), void *data, bool scatter_result)
void takyonReduceChild(TakyonCollectiveReduce *collective, int buffer, uint64_t nelements, uint64_t bytes_per_elem,
void(*reduce_function)(uint64_t nelements,void *a,void *b), bool scatter_result)
void takyonReduceFinalize(TakyonCollectiveReduce *collective)
```

Scatter: One source, multiple destinations

```
TakyonScatterSrc *takyonScatterSrcInit(int npaths, TakyonPath **path_list)
TakyonScatterDest *takyonScatterDestInit(int npaths, int path_index, TakyonPath *path)
void takyonScatterSend(TakyonScatterSrc *collective, int buffer, uint64_t *nbytes_list, uint64_t *soffset_list, uint64_t
*doffset_list)
void takyonScatterRecv(TakyonScatterDest *collective, int buffer, uint64_t *nbytes_ret, uint64_t *offset_ret)
void takyonScatterSrcFinalize(TakyonScatterSrc *collective)
void takyonScatterDestFinalize(TakyonScatterDest *collective)
```

Gather: Multiple sources, one destination

```
TakyonGatherSrc *takyonGatherSrcInit(int npaths, int path_index, TakyonPath *path)
TakyonGatherDest *takyonGatherDestInit(int npaths, TakyonPath **path_list)
void takyonGatherSend(TakyonGatherSrc *collective, int buffer, uint64_t nbytes, uint64_t soffset, uint64_t doffset)
void takyonGatherRecv(TakyonGatherDest *collective, int buffer, uint64_t *nbytes_list_ret, uint64_t *offset_list_ret)
void takyonGatherSrcFinalize(TakyonGatherSrc *collective)
void takyonGatherDestFinalize(TakyonGatherDest *collective)
```

One2One: a set of paths organized in a useful way, such as a pipeline, mesh, etc.

```
TakyonCollectiveOne2One *takyonOne2OneInit(int npaths, int num_src_paths, int num_dest_paths,
TakyonPath **src_path_list, TakyonPath **dest_path_list);
void takyonOne2OneFinalize(TakyonCollectiveOne2One *collective);
```

All2All: (coming soon)



Graph Description (takyon_graph.c)

This feature allows Takyon application data-flow to be described from a configuration file instead of hard-coding in the source code. This allows Takyon to be similar to MPI, but provide fully explicit control over the communication paths. This allows a very efficient use of paths in the application, for example:

- Use the same paths for different collective groups (scatter and gather), or use different paths for each collective group.
- Allow multiple paths between the same endpoints but with different performance properties (fast path for high priority commands, slow path for data)
- Use a different interconnect for each path (RDMA, TCP sockets, UDP socket, etc).
- Use a different number of buffers for each path (single buffer, double buffer, etc).

To see examples of how to use the graph functionality, review the Takyon examples: hello_world_graph, barrier, reduce, pipeline, and scatter_gather.

The Takyon graph description file format:

```
Groups
Group: <name>
    Instances: <integer >= 1>

Processes
Process: <integer_ID>
    GroupIDs: <group_name>[<index>] ...

Buffers
Buffer: <name><buffer_index>
    ProcessID: <ID>
    Where: <app_defined_name>
    Bytes: <integer >= 1>

Paths
Defaults
    IsPolling: {true | false}, {true | false}
    AbortOnFailure: {true | false}, {true | false}
    Verbosity: <or'ing of verbosity flags>, <or'ing of verbosity flags>
    PathCreateTimeout: <takyon timeout value>, <takyon timeout value>
    SendStartTimeout: <takyon timeout value>, <takyon timeout value>
    SendFinishTimeout: <takyon timeout value>, <takyon timeout value>
    RecvStartTimeout: <takyon timeout value>, <takyon timeout value>
    RecvFinishTimeout: <takyon timeout value>, <takyon timeout value>
    PathDestroyTimeout: <takyon timeout value>, <takyon timeout value>
    SendCompletionMethod: {TAKYON_BLOCKING | TAKYON_IS_SEND_FINISHED}, {TAKYON_BLOCKING | TAKYON_IS_SEND_FINISHED}
    RecvCompletionMethod: {TAKYON_BLOCKING, TAKYON_BLOCKING
    NBufsAtoB: <integer >= 0>, <integer >= 0>
    NBufsBtoA: <integer >= 0>, <integer >= 0>
    SenderMaxBytesList: <space separated list of integers >= 0>, <space separated list of integers >= 0>
    RecverMaxBytesList: <space separated list of integers >= 0>, <space separated list of integers >= 0>
    SenderAddrList: <space separated list of NULL and/or <mem_name>:<offset>>, <list of NULL and/or <mem_name>:<offset>>
    RecverAddrList: <space separated list of NULL and/or <mem_name>:<offset>>, <list of NULL and/or <mem_name>:<offset>>
Path: <integer ID>
    Endpoints: <group_name>[<index>], <group_name>[<index>]
    InterconnectA: <interconnect specification>
    InterconnectB: <interconnect specification>
    <other attributes to override defaults>

Collectives
Collective: <name>
    Type: { BARRIER | REDUCE | ONE2ONE | SCATTER | GATHER }
    PathSrcIds: <space separated list of <pathID>:{A|B}>
```

All section headers must be defined.

The following sections must include one or more items: Groups, Processes, Paths

The 'Defaults' section is a convenience to avoid duplicate information across multiple paths. This section does not need to be specified, but if it is, it does not need to specify all path defaults. Each item in 'Defaults' can be in any order. Each time an item is specified in 'Defaults', it replaces what the previous default was. There can be multiple 'Defaults' specified, which is helpful if groups of paths have different defaults.

Load and access a Takyon graph description from a file

```
TakyonGraph *takyonLoadGraphDescription(int process_id, const char *filename)
void takyonFreeGraphDescription(TakyonGraph *graph, int process_id)
```

Create Takyon paths and collective groups; called by the group instances (threads)

```
void takyonCreateGroupPaths(TakyonGraph *graph, int group_id)
void takyonDestroyGroupPaths(TakyonGraph *graph, int group_id)
```

Graph helper functions

```
void takyonPrintGraph(TakyonGraph *graph);
TakyonGroup *takyonGetGroup(TakyonGraph *graph, int group_id);
int takyonGetGroupInstance(TakyonGraph *graph, int group_id);
TakyonCollectiveBarrier *takyonGetBarrier(TakyonGraph *graph, const char *name, int group_id);
TakyonCollectiveReduce *takyonGetReduce(TakyonGraph *graph, const char *name, int group_id);
TakyonCollectiveOne2One *takyonGetOne2One(TakyonGraph *graph, const char *name, int group_id);
TakyonScatterSrc *takyonGetScatterSrc(TakyonGraph *graph, const char *name, int group_id);
TakyonScatterDest *takyonGetScatterDest(TakyonGraph *graph, const char *name, int group_id);
TakyonGatherSrc *takyonGetGatherSrc(TakyonGraph *graph, const char *name, int group_id);
TakyonGatherDest *takyonGetGatherDest(TakyonGraph *graph, const char *name, int group_id);
```

When Takyon graph files are used, the application has to define the following 2 functions to handle memory allocations even if no memory allocations are needed. These functions manage the buffers defined by name in the BUFFERS section; e.g. CPU, MMAP, GPU, etc.

```
void *appAllocateMemory(const char *name, const char *where, uint64_t bytes, void **user_data_ret)
void appFreeMemory(const char *where, void *user_data, void *addr)
```



Strided Message Transfers

IMPORTANT: Striding is not supported by most modern interconnects. If implicit software tricks were used to implement striding for interconnects that did not support it, performance would be significantly impacted. Due to this, Takyon does not support strided functions in the core API set.

For the interconnects that do support striding, the following proposed functions could be used as extensions to the core implementation.

Start sending a strided message (blocking and non-blocking, callable only if the interconnect natively supports striding):

```
bool takyonSendStrided(TakyonPath *path, int buffer_index, uint64_t num_blocks, uint64_t bytes_per_block, uint64_t src_offset,
uint64_t src_stride, uint64_t dest_offset, uint64_t dest_stride, bool *timed_out_ret)
```

Receive a strided message (callable only if the interconnect natively supports striding):

```
bool takyonRecvStrided(TakyonPath *path, int buffer_index, uint64_t *num_blocks_ret, uint64_t *bytes_per_block_ret, uint64_t
*offset_ret, uint64_t *stride_ret, bool *timed_out_ret)
```

Secure Communication

Still needs investigation if security can be fully specified within the Takyon attributes->interconnect[] specification. e.g. add "-SSL" to "Socket" interconnect?