Takyon Core API Reference

Takyon™

The flexibility and performance of low level, and simplicity of high level

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. They high performance, determinism, and fault tolerance, but also abstraction from the details of the underlying communication protocol.

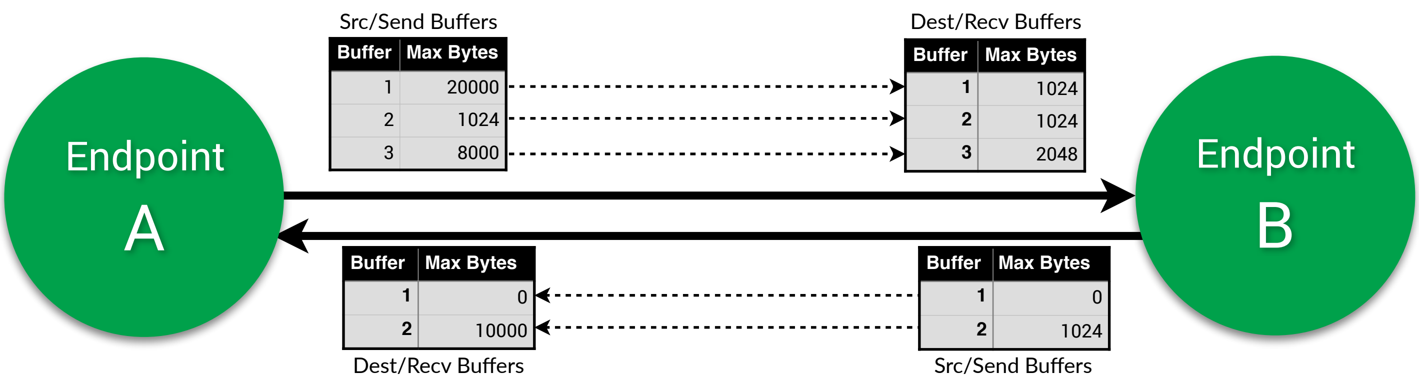
Takyon is designed to be:

- Efficient: A thin wrapper over low 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 and restarted 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.

Takyon Core API Reference

Create a Path

Create a path to a remote endpoint. If it's a connected interconnect then both end points call this to complete the connection. If it's a connectionless interconnect then the remote endpoint is not required to exist. Attributes are stored in the returned path->attrs for easy reference.

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

Returns NULL on error, and the error_message field in the attributes structure will be filled in (use TAKYON_VERBOSITY_ERRORS to print errors automatically).

TakyonPathAttributes Fields	Description
bool is_endpointA	true - Endpoint A of path. false - Endpoint B of path.
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 or TAKYON_USE_SEND_TEST (i.e. non blocking)
TakyonCompletionMethod recv_completion_method	Must be TAKYON_BLOCKING
double create_timeout	Timeout to wait for takyonCreate() to complete.
double send_start_timeout	Timeout to wait for takyonSend() to start transferring.
double send_complete_timeout	Timeout to wait for send to complete transferring.
double recv_complete_timeout	Timeout to wait for takyonRecv() to complete receiving.
double 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 *recver_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 *recver_addr_list	List of pre-allocated receive buffers. Set each entry to NULL to auto-allocate.
char interconnect[MAX_TAKYON_INTERCONNECT_CHARS]	Defines the interconnect and it's properties to be used for transferring. The list of supported interconnects are defined in the implementation's README.txt

Verbosity flags	Value
TAKYON_VERBOSITY_NONE	0x00
TAKYON_VERBOSITY_ERRORS	0x01
TAKYON_VERBOSITY_INIT	0x02
TAKYON_VERBOSITY_INIT_DETAILS	0x04
TAKYON_VERBOSITY_RUNTIME	0x08
TAKYON_VERBOSITY_RUNTIME_DETAILS	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

Locality	Interconnect Specification (only the ones portable to all implementations are listed here)
Inter-thread	Memcpy -ID <ID> [-share]
Inter-process	Mmap -ID <ID> [-share] [-app_alloced_recv_mem] [-remote_mmap_prefix <name>] Socket -local -ID <ID>
Inter-processor	Socket -remoteIP <IP> -port <port> Socket -localIP {<IP> | Any} -port <port> [-reuse]
Connectionless Inter-process/processor	SocketDatagram -unicastSend -remoteIP <IP> -port <port> SocketDatagram -unicastRecv -localIP {<IP> | Any} -port <port> [-reuse] SocketDatagram -multicastSend -localIP <IP> -group <IP> -port <port> [-disable_loopback] [-TTL <n>] SocketDatagram -multicastRecv -localIP <IP> -group <IP> -port <port> [-reuse]

Transferring

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

Receive a message:

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

Notes
Returns *true* if successful or *false* if failed. If failed, error text is stored in path->attrs.error_message, and takyonDestroy() must be called. If a timeout occurs after data starts transferring, then is it considered an unrecoverable error, and *false* will be returned.
If the function returns *true* but *timed_out_ret* is *true*, then no bytes were transferred and it is safe to try again.
timed_out_ret can be NULL if the appropriate timeouts are set to TAKYON_WAIT_FOREVER.
When using takyonRecv(), *bytes_ret* and *offset_ret* can be NULL if there is no need for that information.

Destroy a Path

Destroy the path. For connected interconnects, there is a coordination between both endpoints to make sure data in transit is flushed. The path handle will be set to NULL by this function call. Returns NULL on success, otherwise an error message is returned (and this message must be freed by the application).

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

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

Takyon Header

```
#include "takyon.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, MAX_TAKYON_INTERCONNECT_CHARS);
  attrs.create_timeout              = TAKYON_WAIT_FOREVER;
  attrs.send_start_timeout          = TAKYON_WAIT_FOREVER;
  attrs.send_complete_timeout       = TAKYON_WAIT_FOREVER;
  attrs.recv_complete_timeout       = TAKYON_WAIT_FOREVER;
  attrs.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;
```

Setup Path Attributes

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

Path Creation

```
  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;
}
```

Endpoint A Transfers

Endpoint B Transfers

Path Destruction

Run Endpoint A & B 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 with the transport 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 were not needed.

Takyon Open Source Utility API Reference

The following Takyon utility 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 core Takyon utility APIs needs to:

- include “**takyon_utils.h**”, located in "Takyon/utils/"
- compile and link the appropriate utility C files, located in "Takyon/utils/"

Endian

These will be helpful if the end points are not guaranteed to have the same endianess. 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

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

Allocate named memory than 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.

```
TBD
```

Load Dataflow Configuration File

Since applications can have a large number of communication paths and collective groups, it will be a great convenience to define the dataflow in a configuration file and have the application load the file, create the appropriate threads, create the paths, and create the collective groups. This significantly reduces the complexity of setting up the application framework at startup.

```
TBD
```

Collective Grouping

Allow grouping of a set of paths to be used in an intelligent way (grid, mesh, fan-in, fan-out, all to all, etc) and referenced in an intuitive way.

```
TBD
```

Barrier

O(log2(N)) tree-based barrier:

```
TBD
```

O(log2(N)) dissemination barrier:

```
TBD
```

Scatter

One source and multiple destinations:

```
TBD
```

Gather

Multiple sources and a single destination:

```
TBD
```

Reduce

Combine values from multiple locations into a single value using an application defined function:

```
TBD
```

All-to-All

All nodes in a group send to all nodes in a second group:

```
TBD
```

Potential Takyon Extensions

Strided Message Transfers

IMPORTANT: Striding is not supported by most modern interconnects. If 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 could be used as extensions to the 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 attributes->interconnect[] specification. e.g. add "-SSL" to "Socket" interconnect.