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

Takyon

Takyon is a reliable point to point message passing communication API for C/C++ applications. It's designed for software engineers developing complex, multi-threaded applications for heterogeneous compute architectures. They need low-level features, high performance, determinism, and fault tolerance, but also abstraction from the details of the underlying communication protocol. These needs are common in the embedded HPC (High Performance Computing) field.

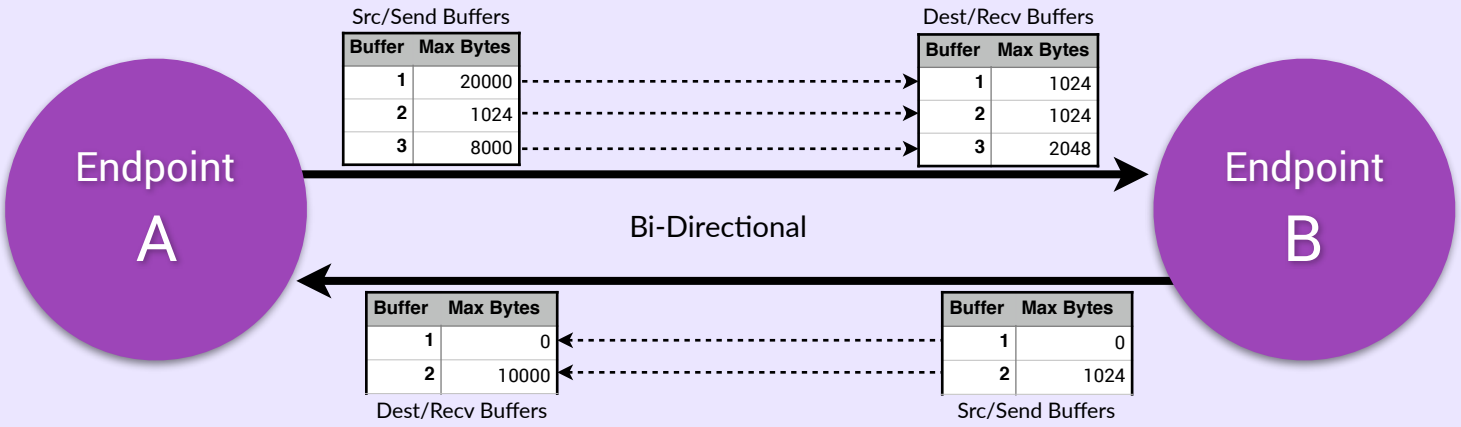
Takyon is designed to be:

- Efficient: A thin wrapper over low level communication APIs that still maintains critical features.
- High Performance: Provides one-way (single underlying transfer), zero-copy (no extra buffering), two-sided transfers (recv gets notification when send completes).
- Portable: Can work with most any modern interconnect, OS, bit width, and endianness.
- Unified: Same API for inter-thread, inter-process, and inter-processor. Use it within a single application or between multiple applications.
- Dynamic: 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

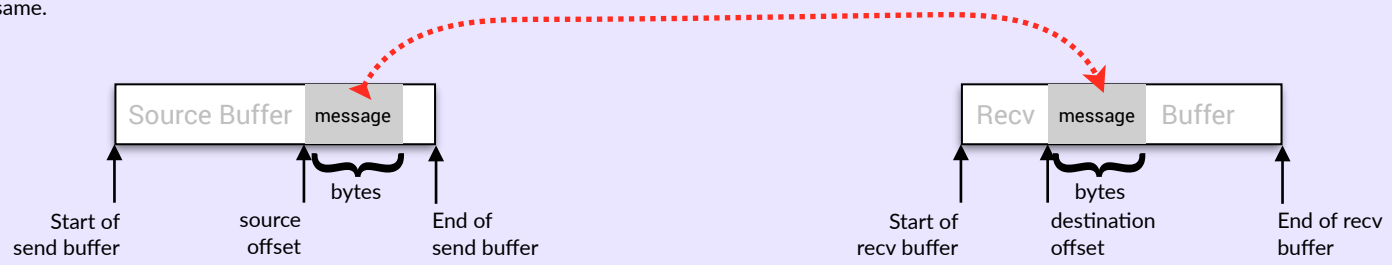
- Reliable message passing: the block of data that is sent is the same that is received.
- Bi-Directional: send in either direction.
- Multi-buffering built in: double and triple buffering is no longer a hassle. A to B and B to A can have a different number of buffers. Each buffer in the buffer list can have a different number of max bytes. The memory for each buffer is defined at time of path creation (can be application defined or allocated by Takyon).
- Can transfer zero-byte messages (good for synchronization).
- Properties of a path are defined at time of path creation and cannot be changed after. Create multiple paths if different properties are needed.
- Transfer notifications (send done, recv done) can be polling or event driven.
- The only implicit synchronization is when a receiver receives a message. There is no synchronization to tell a sender that the receiver is ready for more data. That needs to be done explicitly (e.g. a periodic round trip using a zero byte message). Extra care needs to be taken if using shared memory.



Transfer Details

The following must be specified when sending a message: size, source offset, and destination offset (all are in bytes).

Typically the offsets are set to zero. The message size does not need to be the same size as the buffer, and the source and destination offset do not need to be the same.



Create a Path

Create a path between two endpoints. Both end points call this to complete the connection. 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_start_timeout		Timeout to wait for takyonRecv() to start receiving.
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. One or greater.
int nbufs_BtoA	*	Number of buffers from endpoint B to A. One or greater.
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

* Means both endpoint must use the same value with only minor exceptions.

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

Portable to all implementations

Locality	Interconnect Specification
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> -port <port> Socket -localIP Any -port <port>

Transferring Messages

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. There is a coordination between both endpoints to make sure data in transit is flushed. The path will be set to NULL by this function call. Returns NULL on success, otherwise and 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);
```

Setup Path Attributes

(this could be offloaded to a configuration file, and generated from a dataflow design tool)

```
  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_start_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;
```

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) {
```

Side A Transfers

```
      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 {
```

Side B Transfers

```
      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);
```

```
    }
  }
```

Path Destruction

```
  takyonDestroy(&path);
```

```
  return NULL;
}

int main(int argc, char **argv) {
  if (argc != 2) { printf("usage: hello <interconnect>\n"); return 1; }
```

Run Side A & Side B Threads

```
  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;
}
```

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 a one-way, zero-copy, two-side transfer. 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 Utility API Reference

The following Takyon utility functions are provided as source code 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 endian. Swapping is in bytes.

```
int 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. If seconds is zero, then the thread yields to let other threads run.

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

Returns current system wall clock time (in nanoseconds):

```
uint64_t 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
```

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
```

Takyon Potential Optional Extensions

Path Creation Additions

Add `TAKYON_NO_NOTIFICATION` to the enumeration `TakyonCompletionMethod`. This is an advanced feature that is not directly supported by most interconnects, and is non trivial to use properly. This could be used to allow a non-blocking send to avoid the need to call TakyonSendTest(), and remove the interconnect's underlying notification, thus improving performance slightly by removing a context switch or a call to an interrupt handler.

Strided Message Transfers

The following are advanced features that are not directly supported by most interconnects, and can easily be misused if implemented via software tricks.

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:

```
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)
```

The call should return an error if the interconnect does not support it. This is to ensure a good application design. The alternative would be to:

- Allocate a temporary block of memory, copy the message to the block such that the message is contiguous, register the memory for transport, transfer, and then free the memory.
- Send each contiguous part of the message as a different underlying data transfer.

Both of the above options will have a significant and seemingly mysterious impact on performance.

IO Devices

IO Devices are typically one sided with software access; i.e. the device is not running a full OS that allows for fully featured executables. An application would typically receive from an input device (but not send to it), and send to an output device (but not receive from it). For example, this could facilitate communication to and from an FPGA.

Takyon could potentially allow for an input or output device, where:

- attribute->interconnect[] defines an IO device instead of an interconnect.
- takyonCreate() only needs to be called on one side, so no need to coordinate with a remote call to takyonCreate().
- If it's an input device, then:
  - only attributes->nbufs_AtoB, attributes->recver_max_bytes_list, and attributes->recver_addr_list need to be used.
  - only takyonRecv() is needed, but not takyonSend() or takyonSendTest().
- If it's an output device, then:
  - only attributes->nbufs_BtoA, attributes->sender_max_bytes_list, and attributes->sender_addr_list need to be used.
  - only takyonSend() and takyonSendTest() are needed, but not takyonRecv().

Connectionless and Multicast

The core APIs could potentially be used for (UDP) connectionless and multicast, with the following modifications:

- The attribute->interconnect[] specification needs to indicate that it is UDP and optionally provide a multicast group if it is multicast.
- Endpoint A is always a sender, and endpoint B is always a receiver. i.e. communication is unidirectional.
- Restrict attributes->nbufs_AtoB to 1, and attributes->nbufs_BtoA to 0.
- When sending, restrict 'dest_offset' to 0.
- takyonCreate() is no longer a two sided coordination. It's called once independently for the sender, and once independently for the receiver. If it's a multicast, then it can be called multiple times for the receiver.
- If a failure occurs, only the endpoint that failed needs to call takyonDestroy(), the remote side can still be valid.

Secure Communication

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