Takyon

User's Guide

Edition 1

abaco SYSTEMS

Introduction

Takyon is a high speed, portable, reliable, dynamic, fully scalable, point to point, message passing communication API. Its creation is based on the lessons learned over many years understanding the difficulty application developers have applying existing communication standards (MPI, sockets, verbs, etc.) to existing heterogeneous computer architectures used in embedded HPC applications. Takyon's design goals are:

- **Best posible performance**: latency and throughput
- **High application availability**: dynamic creation, timeouts, and disconnect detection
- **Simplicity**: easy to learn and maintain, and enforces good habits

Software engineers developing applications for the embedded HPC market want to focus their effort on the application and underlying algorithms. The embedded algorithms can be developed for a single processing core, but usually require multiple cores/processors working together to complete the computation in a reasonable amount of time. This requires communication between the cores/processors. Since developers are pushed to minimize time-to-market, they will need a communication package that is trivial to learn, and can provide performance that the underlying hardware is capable of.

Takyon's Prime Directives

In order to meet the above design goals, there are directives that should be enforced by the Takyon implementation (but not necessarily by the underlying interconnect).

Directive 1: One Way, Zero Copy, Two Sided

To get the best performance, as advertised by the interconnect, Takyon will need the following:

- **One Way**: the message is transferred from the sender to the receiver in one underlying communication. Some communication packages don't know where to send the message until the receiver provides the address at the time recv() is called, which requires a round trip to coordinate with the sender. Takyon registers the transport memory at the time the path is created, not when the send() or recv() calls are made. This means the sender always knows where to send the data even if the recv() has not yet been called, removing the need for a round trip. Some communication packages offer a pull method, but this generally requires a round trip with the data source, so Takyon does not offer a one way pull method.
- **Zero Copy**: the message is not copied to some intermediate buffer. Some communication packages don't know where to send the message until the receiver provides the address at the time recv() is called, but to get around this the message can be copied to a pre-registered buffer, and then sent to the receiver without waiting for the recv() call to be made, then the message is copied to the final destination once the recv() call is made. Takyon registers the transport memory at the time the path is created, not when the send() or recv() calls are made. This means the sender never needs to copy the message to an intermediate buffer, which allows for best performance.
- **Two Sided**: the receiver is notified when a message has arrived and is ready to be processed. Some communication packages provide one sided communication which requires either an extra communication used for "received" notification or worse yet a dangerous use of polling on a memory location in the message, waiting for a change. Almost all modern interconnects have a way to send a notification to the receiver by piggybacking a signal with the message. Takyon will make use of this piggybacked signal to avoid a secondary implicit synchronization message.

Directive 2: No Implicit Synchronization

Although Takyon does have implicit synchronization to notify the receiver when a message has arrived, Takyon should not add any implicit synchronization to know when it is safe to:

- Start a send.
- Fill a send buffer in the case where the sender and receive are sharing a memory buffer.

Some communication APIs use implicit synchronization, but this may perturb determinism by sending an implicit communication during a critical time in the life cycle of the application. Although this type of implicit synchronization is a nice convenience, Takyon does not do this. This allows Takyon to be very deterministic, and promotes good application development by explicitly exposing synchronization. Many applications already have explicit synchronization just due to round trip messages. Takyon's "hello world" example shows this type of explicit synchronization.

Directive 3: Dynamic Connections, Timeouts, Disconnect Detection

Some applications require HAA (high application availability). This means that if something goes wrong with communication, it can be detected and either fixed or use an alternative path or method. Some communication APIs are considered static or mostly static, which makes it difficult to achieve HAA. To have true HAA, three features are needed:

- **Dynamic Connections**: Can create and destroy paths at any time during the applications life cycle. Takyon can create and destroy paths at any time without affecting any other established paths. Takyon can even have multiple paths between the same two endpoints using the same or different interconnects.
- **Timeouts**: When sending and/or receiving, there may be some amount of time that passes where the path is no longer considered responsive. Timeouts will reveal this, and the application can use alternative methods to complete its functionality, such as using a secondary path to a backup processor. Takyon has timeouts for all stages of communication from create to destroy.
- **Disconnect Detection**: While timeouts can imply bad communication paths, most modern interconnects can detect when a path has disconnected due to remote failures, network failures, etc. Takyon is designed to detect and report failures from all of its APIs. If a disconnect is detected, then a new path can be created or the application can use an alternative path.

Directive 4: Follow the Intuition, Not the Hardware

Fundamentally, communication is about getting data from A to B. When using a high level communication package, there should be no need to have different APIs for each type of interconnect or for different localities (thread, process, processor). This intuition consists of:

- **Create**: create a two-way communication path between two endpoints
- **Send**: send a message to the remote endpoint (blocking and non-blocking)
- **Send Completion Test**: test if a previously started non blocking send is complete
- **Receive**: receive a message
- **Destroy**: destroy the path

Takyon's API is based on these five communication concepts, and therefore Takyon only has five core functions.

Message Passing 101

This chapter will cover the fundamentals of Takyon and the decisions to keep it simple and maintain best possible performance without any significant tradeoffs.

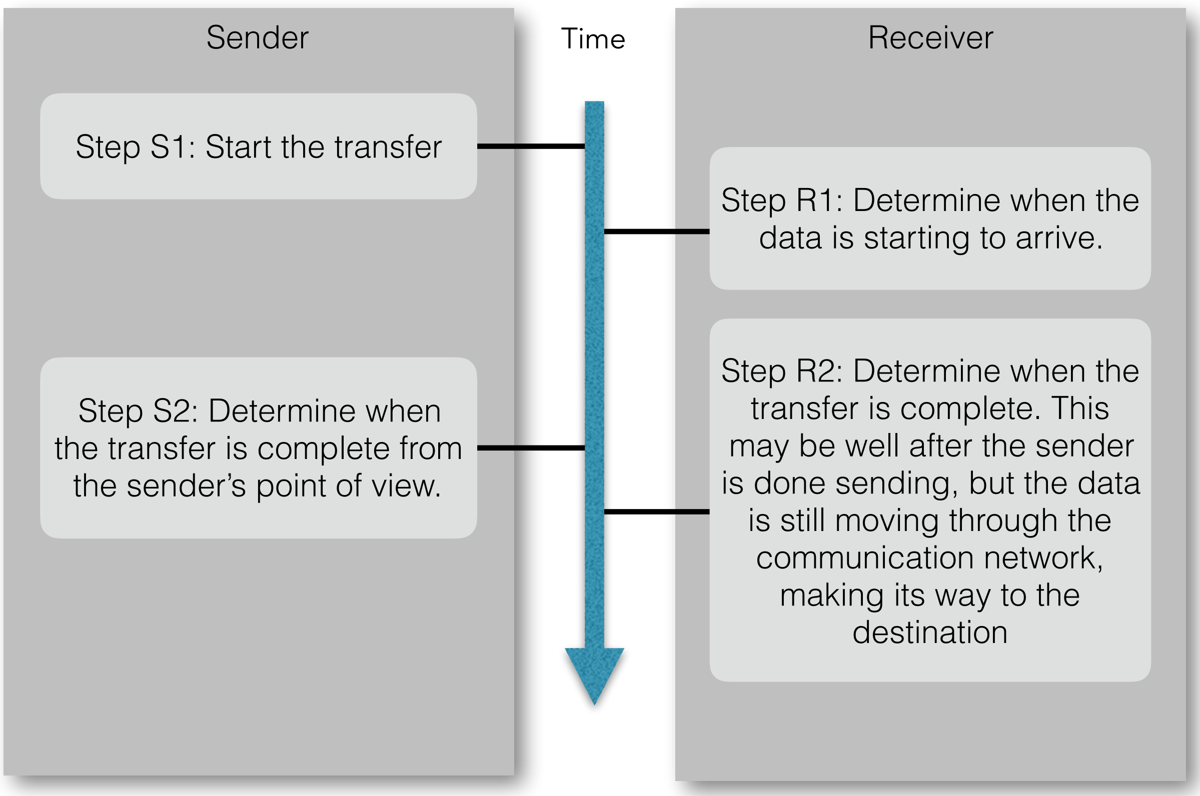
In this chapter, pseudo code will be used to describe the concepts instead of actual Takyon functions.

Messages

Takyon is designed to pass messages: i.e. what is sent by a single call to send() is the same as what is received by a single call to recv().

Data Transferring

Intuitively communication is about moving data from point A to point B. This usually involves sending and receiving. To break it down a little further, there's two phases of sending and two phases of receiving.



That's all there is to it.

Note that S1, S2, R1, and R2 (from the diagram above) will be referenced throughout this chapter.

Pre-Registered Memory

Most communication packages define the source and destination memory at the time of the transfer.

Sender

```
send(data, bytes)
```

Receiver

```
recv(data, &bytes_received)
```

One of the most fundamental aspects of Takyon, which is very different from many communication packages, is that Takyon requires the sender and receiver memory to be defined when the communication path is established. This means the send and receive functions will not need an address for the transfer memory.

Sender

```
send(bytes)
```

Receiver

```
recv(&bytes_received)
```

This may seem odd at first, but the improved performance is potentially a big benefit.

Blocking Transfers

The easiest form of transferring is to do steps S1 and S2 together in a single send function, and do steps R1 and R2 together in a single receive function.

Sender

```
send(bytes)
```

Receiver

```
recv(&bytes_received)
```

This is the common use case of transferring because it's simple to use and understand.

Non Blocking Transfers

This allows the sender to do step S1 with a "send start" function, and with a separate "send test" function it can do step S2.

Sender

```
send_start(bytes)
// do something useful here
send_test(WAIT_UNTIL_DONE)
```

Receiver

```
recv(&bytes_received)
```

Non blocking transfers are useful if the interconnect allows background transfers (no CPU interaction is needed while the data is being sent). This allows extra processing to occur at the same time the transfer is in progress.

The receiver could also be broken up into two separate functions, but it's typically not useful to know when data is starting to arrive, but instead only when it has completely arrived. Therefore Takyon only needs one receive function.

Application Controlled Synchronization

Another characteristic of Takyon achieving best performance, is the transfer needs to be one way. This means that the sender does not know when the receiver is ready for more data. This puts the responsibility of the synchronization on the application, but this is a good responsibility, because it allows the application to choose the best time to do the synchronization, and not waste time doing un-needed synchronization. For example, some applications are self synchronizing due to round trip communications. If the application does not naturally have round trip synchronization, then it's easy to use a zero byte transfer to explicitly do the synchronization, or even use some other method such as GPIO signals.

INCORRECT METHOD: The following represents sending without synchronization:

Sender

```
while (1) {
  prepare_data(data, bytes)
  send(bytes)
}
```

Receiver

```
while (1) {
  recv(&bytes)
  process_data(data, bytes)
}
```

In this case the receiver's data could be overwritten while in the middle of processing, or worse, cause a crash because the interconnect is not designed for this.

CORRECT METHOD: The proper way to guarantee correctness for all interconnects, is to add explicit synchronization.

Sender

```
while (1) {
  prepare_data(data, bytes)
  send(bytes)
  recv(NULL)
}
```

Receiver

```
while (1) {
  recv(&bytes)
  process_data(data, bytes)
  send(0)
}
```

The sender will block after sending a message. The receiver sends the synchronization signal after processing the data.

BETTER METHOD: The above is correct, but not fully efficient. It would be better to block before sending instead of after sending, allowing data to be generated without waiting.

Sender

```
while (1) {
  prepare_data(data, bytes)
  recv(NULL)
  send(bytes)
}
```

Receiver

```
while (1) {
  send(0)
  recv(&bytes)
  process_data(data, bytes)
}
```

This is more efficient since the send will not have to spend much time waiting.

If the application is using multiple transfer buffers, synchronization might only be needed after all of the buffers have been used up.

As you can see, there are many choices of when to synchronization. This shows why it may be detrimental if Takyon used implicit synchronization, since it may reduce performance, and perturb determinism.

Shared Memory

Some Takyon interconnects support shared memory. This is where the sender and receiver actually point to the same memory locations. When the send() is called, it only needs to send a signal to the receiver no matter how many bytes the message is. It's really just sharing a pointer to a memory address.

Synchronization and Shared Memory

If the interconnect is defined to use shared memory, then that means the sender and receiver are actually pointing to the same buffer (they don't have their own independent buffers). If this is the case, the application will need to do more sophisticated synchronization to know when it is safe to start filling the send side buffer. This means that the receiver side needs to have completed processing on any data it previously had.

Multi Buffering

Double and triple buffering is a common practice to overlap processing and transfers. A well developed multi-buffered application will minimize idle time, achieving great overall throughput.

Takyon builds multi-buffering right into the core APIs, making it easy for application developers to use multi-buffering in the application. Buffers are independent of each other, so when a transfer is busy on one buffer, the sender side can start filling data on another buffer and the receiver side can be processing the previous message it received. This is the idea of keeping the transport and the processing cores on both ends busy.

Here's an example of multi buffering and how synchronization is used:

Sender

```
nbufs = 3
buf = 0
while (1) {
  prepare_data(data[buf], bytes)
  if (buf==0) recv(buf, NULL)
  send(buf, bytes)
  buf = (buf+1) % nbufs
}
```

Receiver

```
nbufs = 3
buf = 0
while (1) {
  if (buf==0) send(buf, 0)
  recv(buf, &bytes)
  process_data(data[buf], bytes)
  but = (buf+1) % nbufs
}
```

In this example, synchronization is only done once for all the buffers. This minimizes the overhead for synchronization without compromising correctness.

This example is a great way to show that if Takyon had implicit synchronization for each transfer, it would degrade performance. This is why Takyon does not do any implicit synchronization and leaves it to the application.

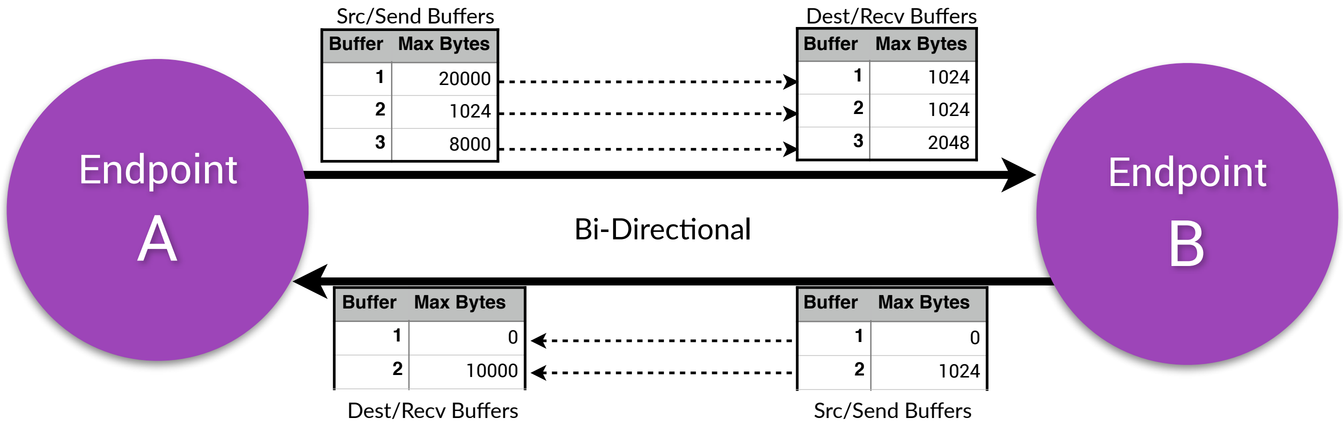
Polling and Event Driven

Takyon performance is also defined by the method of checking for transfer progress. When a send or receive has started, there needs to be a way to determine when it's finished. There are only two ways to do this:

- **Polling**: This uses CPU-based looping to keep checking if the transfer is complete. There are no context switches, so this is very responsive to the exact moment the transfer is complete. The drawback is that the compute core will not be available to do any other processing while Takyon is constantly checking to see if the transfer is complete. Plus a side effect is that the compute core will draw extra power and increase its temperature.
- **Event Driven**: This uses an OS-based mechanism to put the thread to sleep while waiting for the underlying communication interconnect to do the transfer. While the thread is sleeping, the compute core can be busy doing other work. When the transfer is complete, a signal will be sent to the OS to let it know it can wake up the sleeping thread. Knowing when the transfer completed will not be as responsive as polling due to the context switches, but it will free up the compute core to allow other threads and processes to use it while the transfer is in progress.

Bi-Directional Communication and Buffer Sizes

Once a single Takyon path is established, transfers can be done in either direction. This allows the number of created paths to be minimized. Since the goal is to pre-register memory when the path is created, each side of the path must know the largest message size it can transfer. A decision may be to allow one direction to send large data blocks, and the other direction have no message space at all but be able to send zero byte messages to synchronize transfers. Another decision is to have the same buffer size for both directions. The send/recv buffer pairs do not need to be the same size. This is useful for collective communication like scatter or gather, where the one side of the communication may have a lot more contiguous data than the other side.



Locality & Interconnects

Takyon is designed to work between any two endpoints with almost any low level interconnect.

- Thread to thread in the same process
- Process to process on the same OS
- Process to process between different CPUs

This makes Takyon a truly portable communication package. When a path is created, the application can choose which underlying interconnect is used base on the locality of the end points. In most cases, there will be more than one interconnect to choose from. This gives the application the control to create multiple paths to use different interconnects in different ways.

See the release notes for the set of specific interconnects that are supported.

Communicating Between Different Applications

Communication is about moving data regardless of the end points. Takyon does not try to distinguish if it's communicating within a single application or between two applications. I.e. is the same set of APIs whether within an application or between two applications.

IO Device Memory and GPU Memory

Takyon buffers are not restricted to CPU ram, but can also use IO device or GPU memory (e.g. CUDA). It's up to the implementer to add the support for any particular interconnect.

Scalability & Dataflow

Communication should be scalable for any size system. It does mean there needs to be a discipline for how to design the scalability. Developers need to spend time mapping an application to a system, with a specific number of processors, cores, interconnects, memory sizes, etc. If the developer hardcodes the application to a particular system, then it will make it difficult to migrate to a future system with a different number of resources. Designing an application should be done at a logical level regardless of the final deployed system. In this way the application can be tested with minimal resources. When scaled out, it should avoid the need to modify core application code.

If you have an extremely large system, hundreds of processors, then it would probably not be feasible to have direct communication connections between all processor pairs. For example, if you have a 2D grid of processors, each processor may only need to connect to adjacent processors. If the application needs to send a message to a far away processor, it could use multiple hops to make it to its destination.

Scalability should not be enforced by a communication package, but instead should be enforced by good dataflow design. Takyon's design allows for any scalability and dataflow that is needed.

Strided Messages

Very few modern interconnects support strided transfers, which means strided messages need to be reorganized before transporting, or each block of the message needs to be sent separately. To get best performance, the application needs to control how to reorganize the data into a contiguous block before sending.

Where Are the Callbacks?

The concept of callbacks are that the application waits for completion through a function that gets called by an event handler in a separate thread. Callbacks may seem like they fit well with communication, but if the application does not have a specific event loop, callbacks will be confusing and difficult to manage in a way that is safe for communication.

Unlike graphical user interfaces (GUIs) that thrive on event loops with callbacks, most low level communication packages don't require event loops to work. Forcing all communication in a single thread through an event loop may degrade performance. Without an event loop, a callback would need to be handled in a separate thread from the thread that started the communication. This is the fundamental issue, in that one thread starts a communication and another thread would need to complete it, which means there needs to be coordination at the application level between these threads in order to have correct functionality. This is complex enough that it would likely be a feature that is rarely used or understood correctly.

The simpler way is to just have functions that are callable in the same thread to start and test for completion. If something needs to be done while communication is going on, just do it between the send start and send test if in the same thread, or use a different thread if the processing is independent of the current transfer.

The same is true for the receive side. If some processing needs to occur while a receive is ongoing, then either do the processing before calling the recv function, or do the processing in a separate thread if it is independent of the data being received.

Utility Functions

Takyon includes some helpful utility functions that are common to many applications using communication. This includes functions for getting current time, sleeping, endian testing, and endian swapping. Theses functions are provided as source code to make it easy to modify as needed. See the 'Takyon/utils/' folder for the set of available functions.

Collective Communication

Collective communication is about coordinating a set of transfers over a group of communication paths in an organized way: e.g. barrier, scatter, gather, all-to-all, reduce. The core Takyon APIs can be wrapped into higher level convenience functions to handle more complex communication groups. Collective function reference source code will eventually be packaged with Takyon. This will give developers a great starting point to customizing their own collective functions.

Profiling Communication

Profiling communication is the concept of determining usage statistics for all the Takyon functions:

- When a function is called
- When it finishes
- When the send side of the transfer actually occurs, which may be a background transfer that occurs after the send() function returns (non blocking send).
- When the receiver side of the transfer occurs, which may take place before the recv() call has even been made.

Third party event profiling wrappers could be used, which can track when the functions are called, but they won't be able to know when the transferring actually occurs. Deeply integrated profiling, implemented by the Takyon implementor, can do a deeper profiling, to know both when the functions are called, and when the transfers occur.

API Reference

takyonCreate

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

This creates a reliable communication path between two end points. Both end points need to call this in order for the connection to be made. The properties of the path are defined by the attributes, and must be set at time of creation. They cannot be changed once the path is created. If different behavior is needed over time, then create multiple paths to handle the different behaviors.

The attributes are passed in from a pointer to a TakyonPathAttributes structure, which contains the following fields:

Field	Description
char interconnect[MAX_TAKYON_INTERCONNECT_CHARS]	This defines the interconnect to use for this path, a unique identifier to distinguish this path from all other paths using the same interconnect, and any optional parameters supported by the interconnect. See the implementation's README.txt for a list of supported interconnects, the format of the string, and the set of supported parameters. Both end points must use the same interconnect name but the interconnect parameters are implementation specific. The size, in bytes, of the *interconnect* field is MAX_TAKYON_INTERCONNECT_CHARS, which needs to include the null terminating character.
bool is_endpointA	One of the ends must be designated as endpoint A by setting this field to 'true'. If set to 'false', then it's endpoint B. After the connection is made, there is no functional difference, and either end point can do transfers.
bool is_polling	This sets if the communication path is event driven or polling when determining if a transfer is complete. Set to 'true' to use polling, or set to 'false' to use event driven. Some interconnects may require both end points to have the same value.
bool abort_on_failure	This determines if the application should abort if a failure occurs. • Set to 'true' if the application will call abort() if a failure is detected. Before aborting, a helpful error message will be printed to stderr. Using this mode allows the application to avoid the need for error checking the return values of the Takyon APIs, resulting in cleaner source code. • If set to 'false', then the Takyon functions will return even if a failure occurs. All returned values need to be checked to have a reliable application.

Field	Description
uint64_t verbosity	There may be a need to see what's going on within the Takyon functions. This will allow the application to print different types of information to stdout/stderr. If set to 0, the takyon functions do not print any information. Otherwise, do a bitwise or-ing of the following values: • TAKYON_VERBOSITY_NONE - A convenience to show there is no verbosity, this value is 0, but if masked with other values, then this will be ignored. • TAKYON_VERBOSITY_ERRORS - Print (to stderr) details of what caused an error. If this is not used then errors will not get printed out unless 'abort_on_failure' is set to 'true'. • TAKYON_VERBOSITY_INIT - Print (to stdout) basic high level information during the create and destroy functions. • TAKYON_VERBOSITY_INIT_DETAILS - Print (to stdout) extra details during the create and destroy functions. • TAKYON_VERBOSITY_RUNTIME - Print (to stdout) basic high level details during the send and receive functions. • TAKYON_VERBOSITY_RUNTIME_DETAILS - Print (to stdout) extra details during the send and receive functions.
double create_timeout	The timeout period, in seconds, to wait for takyonCreate() to complete. If the connection is not made within the timeout period, then the create function returns NULL.
double send_start_timeout	The timeout period, in seconds, to wait for takyonSend() to start transferring data. Some interconnects are guaranteed to get started without ever waiting and will ignore this timeout, but some interconnects may have to wait for the resource to become available. If takyonSend() times out using this timeout, this is not an error, and sending can be attempted again at a later time.
double send_complete_timeout	The timeout period, in seconds, to wait for a send, that already started transferring bytes, to complete. If the send is blocking, then it will be used by takyonSend(). If the send is non blocking then this timeout will be used by takyonSendTest(). If the timeout occurs, this is considered an error, and the path must be destroyed. Only interconnects that can detect partial transfers will support this timeout.
double recv_start_timeout	The timeout period, in seconds, to wait for takyonRecv() to start receiving data. If takyonRecv() times out with this timeout, this is not an error, and can be attempted again at a later time. Some interconnects only notify when a transfer is complete, and in that case, this timeout is used and *recv_complete_timeout* will be ignored.
double recv_complete_timeout	The timeout period, in seconds, to wait for takyonRecv() to complete after data has started arriving. If the timeout occurs, this is considered an error, and the path must be destroyed. This timeout is only supported with interconnects that can detect partial transfers, such as sockets.
double destroy_timeout	The timeout period, in seconds, to wait for takyonDestroy() to gracefully close the connection, which is a coordination between both end points. If the connection cannot be properly shutdown in the timeout period, then the connection is force closed and this is considered an error, and in this case takyonDestroy() will return an error string.

Field	Description
TakyonCompletionMethod send_completion_method	Determines if takyonSend() is blocking or non blocking. Select one of the following values: • TAKYON_BLOCKING - Wait for takyonSend() to complete the send. • TAKYON_USE_SEND_TEST - Start the transfer using takyonSend(), then use takyonSendTest() to know when the send is complete. Interconnects that don't support non blocking will complete the transfer before takyonSend() returns, but takyonSendTest() will still need to be called to complete the transaction.
TakyonCompletionMethod recv_completion_method	Currently, this must be set to TAKYON_BLOCKING, which means takyonRecv() will block waiting for the transfer to complete.
int nbufs_AtoB	Defines the number of transfer buffers from A (the source) to B (the destination). This value must be 1 or greater. Both endpoints must use the same value.
int nbufs_BtoA	Defines the number of transfer buffers from B (the source) to A (the destination). This value must be 1 or greater. Both endpoints must use the same value.
uint64_t *sender_max_bytes_list	This is a pointer to the endpoint's list of send buffer byte sizes where the size of the list must be *nbufs_AtoB* items if it's endpoint A or *nbufs_BtoA* items if it's endpoint B. This defines the max bytes allowable, per buffer, to send, as long as it does not exceed the buffer size of the receiver, which may be different than the sender. An item in the list can be zero, allowing for zero byte messages to be sent.
uint64_t *recver_max_bytes_list	This is a pointer to the endpoint's list of receive buffer byte sizes where the size of the list must be *nbufs_BtoA* items if it's endpoint A or *nbufs_AtoB* items if it's endpoint B. This defines the max bytes allowable, per buffer, to receive, and does not need to be the same size as the sender. An item in the list can be zero, allowing for zero byte messages to be sent.
size_t *sender_addr_list	This is a pointer to a list of memory addresses where the size of the list must be *nbufs_AtoB* if this is endpoint A or *nbufs_BtoA* if this is endpoint B. If an item in the list is set to NULL, then Takyon will allocate the appropriate amount of memory aligned to a page boundary. If the corresponding byte size is 0, then no memory is allocated, but this will still allow Takyon to do zero byte transfers. If memory was allocated by Takyon, it will automatically free this memory when the path is destroyed. If an item in the list is not NULL, then it must be a valid memory address (casted to a *size_t*) pointing to the appropriate memory address containing the number of bytes defined by *sender_max_bytes_list*. Some interconnects may have additional restrictions. Read the details of the implementation to know the restrictions.

Field	Description
size_t *recver_addr_list	This is a pointer to a list of memory addresses where the size of the list must be *nbufs_BtoA* if this is endpoint A or *nbufs_AtoB* if this is endpoint B. If an item in the list is set to NULL, then Takyon will allocate the appropriate amount of memory aligned to a page boundary. If the corresponding byte size is 0, then no memory is allocated, but this will still allow Takyon to do zero byte transfers. If memory was allocated by Takyon, it automatically free this memory when the path is destroyed. If an item in the list is not NULL, then it must be a valid memory address (casted to a *size_t*) pointing to the appropriate memory address containing the number of bytes defined by *recver_max_bytes_list*. Some interconnects may have additional restrictions. Read the details of the implementation to know the restrictions.
char *error_message	This field should not be set by the application. This field is used to report errors if they occur with takyonCreate(), takyonSend(), takyonSendTest(), or takyonRecv(). If an error occurs with takyonDestroy() the string is returned directly.

If the path was successfully created, then a pointer to a TakyonPath structure is returned. This structure is opaque except for the *path->attrs* field. The attrs field allows the path to know all of it properties, including any memory buffers that Takyon allocated, which are located in the lists *sender_addr_list[]* and *recver_addr_list[]*.

If the path was not created due to an error or was not created within the timeout period *attributes->create_timeout*, then one of the following will occur:

- If *attributes->abort_on_failure* is *false*, then NULL is returned, and an error message is stored in *attributes->error_message*.
- If *attributes->abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonSend

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

Starts a message transfer.

The transfer will be blocking if the path was created using TAKYON_BLOCKING for *send_completion_method*.

If the path was created using TAKYON_USE_SEND_TEST for *send_completion_method* and the interconnect supports non blocking, then the transfer will be done in the background (offloaded from the CPU), otherwise this call will block until the transfer is complete. When in this mode, *takyonSendTest*() will need to be called to complete the transfer, even if the underlying transfer was blocking.

NOTE: Even if the sender completes the transfer, the receiver may still not have all the data, as some of it may still be in the network on its way to the receiver.

buffer_index (starting at 0) represents which memory buffer address in *path->attrs.sender_addr_list[]* that will be used as the source of the transfer.

bytes is the number of bytes that will be transferred. This must be between 0 and the max size of the buffer bytes.

src_offset is the offset from the start of the source's memory block where the transfer will start from. This must be between 0 and the max size of the buffer.

dest_offset is the offset from the start of the destination's memory block where the transferred data will start from. This must be between 0 and the max size of the buffer.

timed_out_ret can be a pointer to a bool (supplied by the application) or NULL if *path->attrs.send_start_timeout* and *path->attrs.send_complete_timeout* are both set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret, if not NULL*, will be set to false.

If the function times out, then it will return true, and *timed_out_ret will be set to true. In this case, it is safe to try the call again.

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonSendTest

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

This blocks until a previously started non blocking send complete's the transfer.

This function should only be called if the path was set up using TAKYON_USE_SEND_TEST for *send_completion_method*, even if the underlying interconnect does not support non blocking transfers.

buffer_index needs to match the buffer index set in the *takyonSend*() call that started the transfer.

timed_out_ret can be a pointer to a bool (supplied by the application) or NULL if *path->attrs.send_complete_timeout* is set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret, if not NULL*, will be set to false.

If the function times out, then it will return true, and *timed_out_ret will be set to true. In this case it is safe to try the call again.

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonRecv

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

This blocks until the message arrives from the other endpoint's call to *takyonSend*().

This call does not need to occur before the call to takyonSend(). This is because the sender already has a handle to the destination memory, and can complete the transfer without any explicit interaction from the receiver. If the data has already arrived, then this call will not block at all.

NOTE: With stream interconnects, like sockets, the transfer may not start until this function is called, which will block the call to sendStart().

buffer_index represents which memory buffer address in *path->attrs.recver_addr_list[]* that will be used to hold the received data. This must match the buffer index set by the call to takyonSend().

bytes_ret is the number of bytes that was received if the transfer completes successfully. If the send side set bytes to 0, then this will also be zero, which likely means it was just used for synchronization/notification. This can be set to NULL if the receiver already knows how many bytes it will receive.

offset_ret is the offset of the received data from the start of the destination's memory buffer address. This can be set to NULL if the receiver already knows the offset.

timed_out_ret can be a pointer to a bool (supplied by the application) or NULL if *path->attrs.recv_start_timeout* and *path->attrs.recv_complete_timeout* are set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret, if not NULL*, will be set to false.

If the function times out, then it will return true, and *timed_out_ret will be set to true. In this case it is safe to try the call again.

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonDestroy

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

If the path is still in a good state, this will block until both end points coordinate a proper disconnect or a timeout occurs as defined by *path->attrs.destroy_timeout* value. This coordinated shutdown allows any pending data on the transport to get flushed to the destination.

If the path is in a bad state due to a previous error, then the connection may be closed without any coordination with the remote end point.

If Takyon allocated any data buffers for this path when *takyonCreate*() was called, then those buffers will be freed.

Once this connection is closed, a new path with the same unique identification can be created again.

If the function fails, then one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then an error message is returned. When this error string is no longer needed by the application, then it needs to be freed by the application.
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

Helpful Programming Tips

Filling in the Path Attributes

There's a lot of attributes to fill in to create a path. Since some application may have a lot of communication paths, it may be easier to have some dataflow tool generate the path creation source code, which would also organize the unique IDs, IP addresses and port numbers needed to create all the paths.

Setting the Interconnects and its Optional Properties

Each path needs to have a unique ID for each type of interconnect used. For IP address based interconnects, this will be an IP address and port number. For most other interconnects, it will just be an ID. Here are some examples:

```
strncpy(attrs.interconnect, "Memcpy -ID 7",                             MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Memcpy -ID 15 -share",                     MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Mmap -ID 26",                              MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Mmap -ID 54 -share",                       MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Socket -local -ID 62",                     MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Socket -clientIP 127.0.0.1 -port 15323", MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Socket -serverIP 127.0.0.1 -port 16232", MAX_TAKYON_INTERCONNECT_CHARS);
strncpy(attrs.interconnect, "Socket -serverIP Any -port 12523",        MAX_TAKYON_INTERCONNECT_CHARS);
```

Since Takyon uses a text string to define the interconnect and it's properties, it makes it very flexible, and can work with many different variations. For example, CUDA memory could be supported via parameters set in the text.

Buffer Allocations

Unless there are some specific needs for buffer memory, just let Takyon create the memory. This reduces the complexity of the application source code. This is achieved by using zero in all the elements of the send and recv addresses passed into the create() call. For example:

```
size_t sender_addr_list[3] = { 0, 0, 0 };
attrs.sender_addr_list     = sender_addr_list;
size_t recver_addr_list[2] = { 0, 0 };
attrs.recver_addr_list     = recver_addr_list;
```

Getting Helpful Print Messages

During development, make sure error reporting is turned on:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS;
```

If errors or odd behavior are occurring, then turn on messages for each call to Takyon:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS | TAKYON_VERBOSITY_INIT | TAKYON_VERBOSITY_RUNTIME;
```

If tracking down difficult errors, it may be useful to turn on more details:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS | TAKYON_VERBOSITY_INIT | TAKYON_VERBOSITY_RUNTIME |
TAKYON_VERBOSITY_INIT_DETAILS | TAKYON_VERBOSITY_RUNTIME_DETAILS;
```

When development is complete, but your still want error messages, use:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS;
```

If your application is designed to handle disconnects, you may want to turn off error messages, so in that case use:

```
attributes.verbosity = TAKYON_VERBOSITY_NONE;
```

Handling Errors

If the application is not designed for handling disconnects, and you want to avoid error checking, then use the following:

```
attributes.abort_on_failure = true;
```

In this case, the application will just abort if it detects an error, and a helpful error message will be printed so you know what happened.

If you need to handle disconnects without aborting, then you need to set the following:

```
attributes.abort_on_failure = false;
```

All return values from the Takyon APIs will need to be checked.

If exit() is preferred over aborting, then the application can get the get status of each call and if an error occurs call exit(<n>) with the appropriate value for <n>.

Interconnect Porting Kit

Adding a new interconnect is relatively easy.

1. Copy the file `Takyon/API/src/takyon_template.c` to `Takyon/API/src/takyon_<interconnect>.c`
2. Fill in `takyon_<interconnect>.c` with the proper implementation. Use the existing `takyon_<name>.c` files as a reference for the implementation.
3. If any utilities need to be added to keep the implementation broken up into manageable chunks, just add a utility file `Takyon/API/src/utils_<funtionality>.c`, and add the global functions to `Takyon/API/inc/takyon_private.h`
4. Add the interconnect and any new utility files to the appropriate makefiles in `Takyon/API/builds/*`.
5. Run the examples to validate the new implementation.

WE INNOVATE. WE DELIVER. YOU SUCCEED.

Americas: 866-OK-ABACO or +1-866-652-2226 **Asia & Oceania:** +81-3-5544-3973
Europe, Africa, & Middle East: +44 (0) 1327-359444
Locate an Abaco Systems Sales Representative visit: abaco.com/products/sales

abaco.com @AbacoSys