tchannel v2

Design goals:

- Easy to implement in multiple languages, especially JS and Python
- High performance forwarding path. Intermediaries can make a forwarding decision quickly.
- Request / response model with out of order responses. Slow requests will not block subsequent faster requests at head of line.
- Large requests/responses may/must be broken into fragments to be sent progressively.
- Optional checksums
- Can be used to transport multiple protocols between endpoints, eg. HTTP+JSON and Thrift.

Why not Finagle?

The Finagle system developed by Twitter was a big inspiration for this protocol, specifically finagle-mux. However, there are several additional features we want out of this system that will require changes. Also, there is only one implementation of finagle-mux, which is in Scala. At Uber we'll need to implement at least a node and python version of this protocol, and a Go and JVM version are likely to follow after.

Field Length Conventions

In this document, all numeric values are unsigned and in big-endian byte order. All strings are UTF-8. All keys and values used in headers are strings.

The schema for describing field lengths and counts is the same as finagle-mux. The schema `size:4 body:10` defines the field "size" to be 4 bytes, followed by 10 bytes of the field "body". The schema `key~4` defines the field "key" to be 4 bytes of a size, followed by that many bytes of data. This is shorthand for `keysize:4 key:keysize`

Groups are denoted by parentheses. A group's repetition count is either `{*}` for 0 or more repetitions, or `{n}` for exactly n repetitions.

Message Flow

Tchannel is a bi-directional request/response protocol. Each connection between peers is considered equivalent, regardless of which side initiated it. It's possible, perhaps even desirable, to have multiple connections between the same pair of peers. Message ids are scoped to a connection. It is not valid to send a request down one connection and a response down another. It is certainly possible to send one request down one connection and then a subsequent request down another, for whatever reason you might have.

Initiating a new connection works like this:

1. node A initiates TCP connection to node B
2. B accepts TCP, must not send any data until receiving "init req"
3. A sends "init req" with desired version, must not send any data until receiving "init res"
4. B sends "init res", with selected version V. B may now send requests with version V.
5. A receives "init res". A may now send requests with version V.

Each message is encapsulated in a frame with some additional information that is common across all message types. Part of that framing information is an id. This id is chosen by the requestor when sending a request message. When responding to a request, the responding node uses the message id in the request frame for the response.

Each frame has a type which describes the format of the frame's body. Depending on the frame type, some bodies are 0 bytes.

The "call req" frame type has a "ttl" field which is used to specify the deadline for this request. The transmitter manages the ttl to account for time in transit or time to failover, and send it to receivers only for deadline propagation purposes. This allows receivers to know how much time they have remaining to complete this portion of the request. Intermediaries are free to make retries with backoff as necessary, as long as they are still within the ttl.

Here are some details of the message flow from a service A instance A1, through a service router R1, into a service B instance B1, and then back. These are not the full details of every single field, but they are intended to illustrate some less obvious behaviors.

The flow in this example is: `A1 -> R1 -> B1 -> R1 -> A1`

A1 sends "call req" (type 0x03) to R1:

1. select a message id M1 for this frame, in a predictable sequence for easier debugging. This will be used to match up the response to this request. The scope of M1 exists only between A1 and R1 and only for this connection.
2. generate a unique traceid which will be propagated to any dependent requests. This should only be done if A1 is the edge-most service in the call chain. The point of this unique traceid is to build a span tree that covers all RPCs supporting a given client-visible transaction, not just tracing of a single logical request through multiple intermediaries.
3. generate a unique spanid which is used to refer to this specific step in the network
4. set the ttl to the maximum allowable time for this request. If a response cannot be sent within this time, downstream nodes will cancel the request.
5. A1 doesn't need any headers, so defaults are used
6. a "call req" body is generated with service_name "service B", application payload args, and checksum

R1 receives "call req" from A1, sends "call req" (0x03) to B1:

1. select message id M2 for this frame
2. copy traceid from incoming message to traceid of new message
3. copy spanid from incoming message to parentid of new message
4. generate unique spanid
5. copy ttl from incoming message to ttl of new message
6. copy service name, args, and csum data to new message

B1 receives "call req" from R1, sends "call res" (0x04) to R1:

1. use message id M2 for this frame
2. send args from application response and compute csum

R1 receives "call res" from B1, send "call res" (0x03) to A1:

1. match incoming message id M2 with existing "call req"
2. use message id M1 for new message
3. copy args/csum from incoming message

A1 receives "call res" from R1:

1. match incoming message M1 with existing “call req”
2. deliver args to application

Framing

All frames of all types use this structure:

0-7	size:2	type:1	reserved:1	id:4
8-15	reserved:8			
16+	payload - based on type			

size:2	Total length of this frame in bytes, including the framing header and body. Note that this limits the total size of a frame to 64K, even though some of the other fields are also specified with 16 bit sizes. Implementations must take care to not exceed the total frame size of 64K.

type:1 Each type has different contents. Valid types are:

0x01	init req	First message on every connection must be init. No requests may be sent until init res is received.
0x02	init res	Remote response to init. Includes the version number that this connection will use.
0x03	call req	RPC method request.
0x04	call res	RPC method response.
0x13	call req continue	RPC request continuation fragment
0x14	call res continue	RPC response continuation fragment
0xc0	cancel	Cancel an outstanding call req / forward req (no body)
0xc1	claim	Claim / cancel a redundant request
0xd0	ping req	protocol level ping req (no body)
0xd1	ping res	ping res (no body)
0xff	error	Protocol level error. May come back from any request (details below)

id:4 An identifier for this message that is chosen by the sender of a request. This id is only valid for this sender on this connection. This is similar to how TCP has a sequence number in each direction. Each side of a connection may happen to select overlapping message ids, which is fine because they are directional.

“id” represents the top level message id. Both request frames and response frames use the same id, which is how they are matched up. A single id may be used across multiple request or response frames after being broken up into a sequence of fragments, as described below.

Valid values for “id” are from 0 to 0xFFFFFFFE. The value 0xFFFFFFFF is reserved for protocol error

	responses.
reserved	unused space for future protocol revisions, padded for read alignment, which may or may not help on modern Intel processors.
payload	0 or more bytes whose contents are determined by the frame type

The framing layer is common to all payloads. It intentionally limits the frame size to 64K to allow better interleaving of frames across a shared TCP connection. In order to handle most operations, implementations will need to decode some part of the payload.

Payload: init req (0x01)

```
version:2 (key~2 value~2){2}
```

This must be the first message sent on a new connection. It is used to negotiate a common protocol version and describe the service names on both ends. In the future, we will likely use this to negotiate authentication and authorization between services.

version is a 16 bit number. The currently specified protocol version is 2. If new versions are required, this is where a common version can be negotiated.

There are a variable number of key/value pairs. For version 2, the following are required:

host_port	address:port where this process can be reached
process_name	additional identifier for this instance, used for logging

Payload: init res (0x02)

```
version:2 (key~2 value~2){2}
```

The initiator requests a version number, and the server responds with the actual version that will be used for the rest of this connection. The name/values are the same, but identify the server.

Payload: call req (0x03)

```
flags:1 ttl:4 tracing:24 traceflags:1 service~2 nh:1 (hk~1 hv~1){nh} csumtype:1 (csum:4){0,1}
arg1~2 arg2~2 arg3~2
```

This is the primary RPC mechanism. The triple of (arg1, arg2, arg3) is sent to “service” via the remote end of this connection. Whether connecting directly to a service or through a service router, the service name is always specified. This supports an explicit router model as well as peers electing to delegate some requests to another service. A forwarding intermediary can relay payloads without understanding the contents of the args triple. A “call req” may be fragmented across multiple frames. If so, the first frame is a “call req”, and all subsequent frames are “call req continue” frames.

Field description:

flags:1 Used to control fragmentation. Valid flags:

0x01	more fragments follow. If this flag is not set, then this is the only frame for this message id.

ttl:4 Time To Live in milliseconds. Intermediaries should decrement this as appropriate when making dependent requests. Since all numbers are unsigned, the ttl can never be less than 0. Care should be taken when decrementing ttl to make sure it doesn't go below 0. Requests should never be sent with ttl of 0. If the ttl expires, an error response should be generated.

tracing:24 tracing payload in zipkin format. The contents are:

spanid:8	int64 that identifies the current "span", a logic operation like call or forward.
parentid:8	int64 of the previous span
traceid:8	int64 assigned by the original requestor. This id does not change as it propagates through various services.

When a request enters our system, the edge-most requester selects a traceid which will remain unchanged as this message and any dependent messages move through the system. Each time a new request is generated, a new spanid is generated. if it was started from a previous request, that id is moved to the parentid.

traceflags:1

0x01	tracing enabled for this request. When this flag is not set, the tracing data is still required to be present in the frame, but the tracing data will not necessarily be sent to zipkin.

service~2 UTF-8 string identifying the destination service to which this request should be routed

headers described below in the "headers" section

arg1~2
arg2~2
arg3~2 The meaning of the three args depends on the systems on each end. The format of arg1, arg2, and arg3 is unspecified at this the transport level. These are opaque binary blobs as far as tchannel is concerned.

checksum described below in the "checksums" section

Future versions will likely allow callers to specify specific service instances on which to run this request, or a mechanism to route a certain percentage of all traffic to a subset of instances for canary analysis.

Headers:

tchannel headers are 0 or more key/value pairs. The keys and values are UTF-8 strings. The number of headers is sent with the first byte, "nh" in the above example. If nh is 0, then the next two bytes would be arg1's length. If nh is 1, then the one key/value pair will follow. The length of the key and value are sent as the first byte. These headers are intended to control things the transport and routing layer, so they are expected to be small and inexpensive to process. Application level headers can be expressed at a higher level in the protocol.

Duplicate header keys are not allowed. Parsers should send back an error if they encounter this.

For example, consider this header sequence:

0x01	0x03	0x63	0x69	0x64	0x02	0x68	0x69
1 - there is exactly 1 key/val pair here	key 1 is 3 bytes long	ASCII c	ASCII i	ASCII d	value 1 is 2 bytes long	ASCII h	ASCII i

All of these headers are optional. While the protocol does support the natural addition of arbitrary keys and values, the only allowable keys in this protocol version for this frame type are these:

Key	**Value**	**Description**
cas	host_port	“claim on start” - this request has also been sent to another instance running at host_port. Send a claim message when work is started.
caf	host_port	“claim on finish” - Send claim message to host_port when response is being sent.
re	retry flags	Flags are encoded as a UTF-8 string, as are all header values. Each flag is represented by a single character. 0x6e “n” | no retry, expose all errors immediately - cancels 0x6e and 0x63 0x63 “c” | retry on connection error. The specific mechanism of retry is not specified by tchannel. Actual retries are handled by the routing layer. 0x74 “t” | retry on timeout If unspecified, re is assumed to be 0x63. Valid values for re are: “n”, “c”, “t”, “ct”, and “tc”.
se	count	length is always 1. Count is the number of nodes on which to run this request, encoded as a UTF-8 string, as are all header values. The only valid value for se in protocol version 2 is 0x32 (ASCII 2). In the future we may extend this speculative execution system to more than 2 nodes. In the interests of simplicity and minimizing confusion, we are intentionally limiting the se factor to 2.
fd	str	“failure domain” - A string describing a group of related requests to the same service that are likely to fail in the same way, if they were to fail. For example, some requests might have a downstream dependency on another service, while others might be handled entirely within the requested service. This is used to implement Hystrix-like “circuit breaker” behavior.

A note on host_port

While these host_port fields are indeed strings, the intention is to provide the address where other entities not directly connected can reach back to this exact entity. In the current world of IPv4, these host_port fields should be an IPv4 address as “dotted quad” ASCII, followed by a colon, and then the listening port number, in ASCII. Example:

```
10.0.0.1:12345
```

This field should not use hostnames or DNS names, unless we find a compelling reason to use them.

For IPv6 addresses, assuming we ever use them, the format should be:

```
[1fff:0:a88:85a3::ac1f]:8001
```

Note that since IPv6 addresses have colons in the address part, implementations should use the last colon in the string to separate the address from the port.

Deadlines and the TTL

The TTL is the amount of time the calling service is willing to wait for the called portion of the call graph to complete. An instance may retry as often as they like so long as they continue to meet that deadline. To propagate the deadline downstream, all new calls that you make should be done with a TTL that equals the TTL assigned to you minus the amount of time you've already spent since receiving the request. It may help to add some fuzz for network latency.

TTLs are not sent back with the response. It is up to the calling service to track the time spent with dependent requests and validate the incoming deadline before dispatching every new dependent request.

Checksums:

Checksums are optional in order to ease implementations in different languages and platforms. While TCP provides a checksum within the scope of a connection, tchannel payloads are potentially forwarded through multiple TCP connections. By adding a checksum at the source, intermediaries and the destination can validate that the payload hasn’t been corrupted. Validating these checksums and rejecting frames with invalid checksums is desirable, but not required.

Checksums are computed across all three args like this:

```
csum = func(arg1, 0);
csum = func(arg2, csum);
csum = func(arg3, csum);
```

This behavior changes slightly when messages are fragmented. See “fragmentation” below.

Checksum types:

0x00	no checksum, 0 bytes	The easy way out. If the csumtype is 0x00, then the csum field is omitted.
0x01	crc32, 4 bytes	CRC32 is intended as a checksum, and many implementations exist, including SSE4.2 instructions on supported platforms.
0x02	farmhash hash32 x86, 4 bytes	Farmhash is a hash function with excellent distribution. It is surprisingly faster than CRC32 on modern CPUs, depending on how it is compiled.

Fragmentation

There are two important and potentially overlapping cases to support with fragmentation:

1. large messages, ones whose args do not fit into a single frame
2. messages whose total size is not known when the request is initiated, similar to HTTP's chunked encoding

The maximum size of an individual frame is intentionally limited to 64K to avoid head of line blocking on a shared TCP connection. This size restriction allows frames from other messages to be interleaved with frames of the larger message. Limiting the size also reduces the buffering requirements on any intermediary nodes and may support the use of fixed allocations in some implementations.

Both "call req" and "call res" frames start with this common set of fields:

```
flags:1 ttl:4 tracing:24 traceflags:1 service~2 nh:1 (hk~1 hv~1){nh}
```

These fields must all fit into a single frame. After sending these fields, the checksum args are sent, or as much of the args as will fit into this frame. If the combined size of the frame is less than 64K, the frame is complete, and the "more fragments follow" flag bit is not set. However, if there is not enough space in the frame to fit the checksum and all of the args, then this message is continued with a "call req continue" frame. On the last "call req continue" frame, the "more fragments follow" flag is not set, and the message is complete.

Since the full size of a given arg may not be known in advance, the the size fields in the frames represent how much of that arg is present in this frame. An arg is considered complete when there is more data in a frame past the end of that arg. If the arg happens to end on an exact frame boundary, the next continuation frame will start with a size of 0 bytes for that arg.

Checksums are computed for the contents of the frame, with all of the args data that is present in that frame. The checksum for each frame is seeded with the cumulative checksum from the previous frames. This allows corruption to be detected before delivering arg payloads to the application.

Example:

Sending a "call req" with a 4 byte arg1, a 2 byte arg2, and an 8 byte arg3. The args are streamed in tiny fragments as an easier to comprehend example of fragmentation. This might happen in the real world if a producer is streaming updates as they arrive. If the total sending size is known in advance, then sending the largest frames possible is preferable.

Frame 1 is a "call req" 0x3 with the "more fragments remain" flag set. The ttl, tracing, traceflags, service, and headers are all set to some reasonable values. We have 2 data bytes of arg1 available, but arg1 is incomplete. We compute the checksum over arg1's 2 bytes with a seed of 0. When sending arg1, we specify a length of 2 bytes, the data bytes of which are the last 2 bytes in the frame. The receiving parser notes that arg1 is incomplete.

Frame 2 is a "call req continue" 0x13 with the "more fragments remain" flag set. 2 more bytes of arg1 are now available, as are 2 bytes of arg2. The checksum is computed over this frame's contents for arg1 and arg2 using the seed from the previous frame of 0xBEEF. The receiving parser knows that it is continuing arg1. Arg1 in this frame has 2 bytes, but there are still more bytes in this frame, so the receiving parser knows that arg1 is now complete. There are 2 bytes of arg2 that end this frame. The receiving parser notes that arg2 is incomplete, even though we know that no more bytes will follow from arg2. This is done to illustrate what happens when an arg ends on an exact frame boundary.

Frame 3 is a “call req continue” 0x13 with the “more fragments remain” flag cleared. There are 0 bytes from arg2 and 8 bytes from arg3. The checksum is generated with 8 bytes from arg3 and the seed from the previous frame of 0xDEAD. The receiving parser knows that it is continuing arg2, and the 0 length indicates that arg2 is finished. The 8 bytes of arg3 are read, which fall on the frame boundary. The receiving parser knows that arg3 is complete because the “more fragments remain” flag was not set.

type	**id**	**payload**	**state after parsing**
0x03	1	flags:1=0x1 ttl:4=0x2328, tracing:24=0x1,0x2,0x3, traceflags:1=0x1, service~2=0x5”svc A”, nh:1=0x1, hk~1=0x1”k”, hv~1=0xA”abcdefghij”, csumtype:1=0x2 csum:4=0xBEEF arg1~2=0x2<2 bytes>	sending arg1
0x13	1	flags:1=0x1, csumtype:1=0x2 csum:4=0xDEAD arg1~2=0x2<2 bytes> arg2~2=0x2<2 bytes>	sending arg2
0x13	1	flags:1=0x0 csumtype:1=0x2 csum:4=0xF00F arg2~2=0x0<0 bytes> arg3~2=0x8<8 bytes>	complete

Payload: call req continue (0x13)

```
flags:1 csumtype:1 (csum:4){0,1} {continuation}
```

This frame continues a “call req” as described in “fragmentation” above.

“flags” has the same definition as in “call req”: to control fragmentation.

Payload: call res (0x04)

```
flags:1 code:1 nh:1 (hk~1 hv~1){nh} csumtype:1 (csum:4){0,1} arg1~2 arg2~2 arg3~2
```

This is the response to a “call req”. It contains the triple of (arg1, arg2, arg3) and optional checksum as generated by the remote application.

“flags” has the same definition as with “call req”: to control fragmentation.

“code” is a response code from the following table:

0x00	OK - everything is great and we value your contribution.
0x01	Not OK - application error, details are in the args. This does not imply anything about whether this request should be retried.

Headers:

Headers for “call res” are optional. The only supported response header in v2 of the protocol is the “fd” header, which is used for the same thing as “fd” in the request message.

“fd”	str	“failure domain” - A string describing a group of related requests to the same service that are likely to fail in the same way, if they were to fail. For example, some requests might have a downstream dependency on another service, while others might be handled entirely within the requested service. This is used to implement Hystrix-like “circuit breaker” behavior.

The encoding of args and checksum follow the same rules for “call res” as they do for “call req”, including fragmentation behavior. A “call res” message is continued using a “call res continue” (0x14) message.

Payload: call res continue (0x14)

```
flags:1 csumtype:1 (csum:4){0,1} {continuation}
```

This frame continues a “call res” as described in “fragmentation” above.

“flags” has the same definition as in “call req”: to control fragmentation.

Payload: cancel (0xC0)

```
why~2
```

This message forces the original response to either a “call req” with an error type of 0x02 (cancelled). The id in the frame of the cancel message should match the id of the req frame intended to be cancelled. Note that since message ids are scoped to a connection, cancelling a message might trigger the cancellation of one or more dependent messages.

“why” is a string describing why the cancel was initiated. It is used only for logging.

It should be noted that the response and cancellation message could pass each other in flight, so we need to be able to handle a response after it was cancelled. We also need to be able to handle duplicated responses for similar reasons. The edges of this network need to implement their own de-duping strategy if necessary.

Payload: claim (0xC1)

```
tracing:24
```

This message is used to claim or cancel a redundant request. When a request is sent to multiple nodes, as they start or finish work, depending on the option, they will tell the other nodes about this to minimize extra work. This claim message is sent from worker to to worker. The claimed request is referred to by its full zipkin tracing data, which was chosen by the originator of the first request.

When a worker B receives a claim message from worker A for a tracing T that B doesn't know about, B will note this for a short time. If B later receives a request for T, B will silently ignore T. This is expected to be the common case, because the forwarding router will add some delay after sending to A and before sending to B.

This is an implementation of Google's "The Tail at Scale" paper where they describe "backup requests with cross-server cancellation". Slides from a presentation about this paper are here:

http://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/Berkeley-Latency-Mar2012.pdf

The relevant section starts on page 39. A video of this talk is here:

http://youtu.be/C_PxVdQmfpk?t=26m45s

Payload: ping req (0xD0)

Used to verify that the protocol is functioning correctly over a connection. The receiving side will send back a "ping res" message, but this is not expected to be visible to the application. If more detailed health checking and validation checks are desired, these can be implemented at a higher level with "call req" and "call res" messages.

This message type has no body.

Payload: ping res (0xD1)

Always sent in response to a "ping req". Sending this does not necessarily mean the service is "healthy." It only validates connectivity and basic protocol functionality.

This message type has no body.

Payload: error (0xFF)

```
code:1 id:4 message~2
```

Response message to a failure at the protocol level or when the system was unable to invoke the requested RPC for some reason. Application errors do not go here. Application errors are sent with "call res" messages and application specific exception data in the args.

code:1	0x00	not a valid value for "code". Do not use.
	0x01	timeout - no nodes responded successfully within the deadline
	0x02	request was cancelled with a cancel message
	0x03	peer is too busy, this request is safe to retry elsewhere if desired, prefer other peers for future requests where possible

0x04	peer declined request for reasons other than load, the request is safe to retry elsewhere if desired, do not change peer preferencing for future requests
0x05	request resulted in an unexpected error. The request may have been completed before the error, retry only if the request is idempotent or duplicate execution can be handled.
0x06	request args do not match expectations, request will never be satisfied, do not retry
0xFF	fatal protocol error - connection will close after this frame. message ID of this frame is 0xFFFFFFFF.

id:4	message id of the original request that triggered this error, or 0xFFFFFFFF if no message id is available.
message~2	a human readable string not intended to be shown to the user. This is something that goes into error logs to help engineers in the future debug code they've never seen before. We will likely need to adopt some kind of convention around the contents of this field, particularly so that they can be parseable for proper search indexing and aggregation. However, for the purposes of the protocol, the contents of "message" are intentionally not specified.

Future Work

not part of the v2 spec unless it gets cleaned up quickly.

Body: publish (call me maybe)

```
nh:1 (hk~1 hv~1){nh} arg1~2 arg2~2 arg3~2 csumtype:1 csum:4
```

Sends a message without waiting for or expecting a reply.

This needs some work to make sure we can handle the Hystrix monitoring use case

Comments:

- name “publish” is confusing
- what about subscribe?
- Probably want a way to specify fanout based on some descriptor
- In general people have asked for a way to send a request without requiring a response. The specific use cases for this request are not clear.

Suggestion to add a QoS header is interesting. When we get more experience operating a system like this, we should consider how to best implement this.