Package 'rgrpc'

September 17, 2026

Type Package

Title Asynchronous 'gRPC' Client and Server Runtime

Version 0.1.1

Date 2026-09-09

Description A first-class asynchronous 'gRPC' <https://grpc.io/> runtime built on the generic asynchronous 'C++' API ('GenericStub', 'AsyncGenericService') <https://grpc.github.io/grpc/cpp/>. Requests and responses cross the native boundary as method names plus opaque byte buffers; 'RProtoBuf' supplies and consumes the bytes, so schemas are loaded at runtime and no generated service stubs are required. Native completion threads never call the R API: completions are queued natively and delivered in batches on the R main thread. Complements 'RProtoBuf' rather than replacing it. Links against the system 'gRPC' library for 'C++'.

License Apache License (>= 2)

URL https://github.com/cornball-ai/rgrpc

BugReports https://github.com/cornball-ai/rgrpc/issues

Depends R (>= 4.3.0)

Imports methods

Suggests RProtoBuf, tinytest

SystemRequirements gRPC C++ and protobuf libraries with development headers, found via 'pkg-config grpc++ protobuf' (Ubuntu/Debian: 'libgrpc++-dev', 'libprotobuf-dev'; on Windows both ship with Rtools 4.3 and later).

Encoding UTF-8

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Troy Hernandez [aut, cre] (ORCID: <https://orcid.org/0009-0005-4248-604X>), cornball.ai [cph]

Maintainer Troy Hernandez <troy@cornball.ai>

Repository CRAN

Date/Publication 2026-09-17 14:40:21 UTC

Contents

`rgrpc-package` *Asynchronous 'gRPC' Client and Server Runtime*

Description

A first-class asynchronous 'gRPC' <https://grpc.io/> runtime built on the generic asynchronous 'C++' API ('GenericStub', 'AsyncGenericService') <https://grpc.github.io/grpc/cpp/>. Requests and responses cross the native boundary as method names plus opaque byte buffers; 'RProtoBuf' supplies and consumes the bytes, so schemas are loaded at runtime and no generated service stubs are required. Native completion threads never call the R API: completions are queued natively and delivered in batches on the R main thread. Complements 'RProtoBuf' rather than replacing it. Links against the system 'gRPC' library for 'C++'.

Package Content

Index of help topics:

grpc_await	Wait for events belonging to one call
grpc_call	Start a unary call
grpc_cancel	Cancel an in-flight call or stream
grpc_client	Create an asynchronous gRPC client
grpc_close	Shut down a client or server
grpc_decode	Decode protocol buffer bytes to a message
grpc_fd	Completion wakeup file descriptor
grpc_finish	End a server stream
grpc_method	Look up a method in a resolved service
grpc_pending	Number of pending operations
grpc_poll	Receive completed events
grpc_read	Pull the next inbound message on a server stream
grpc_reply	Answer an incoming request
grpc_send	Send a message on a stream
grpc_server	Create an asynchronous generic gRPC server
grpc_server_port	Bound TCP port of a server
grpc_service	Resolve a gRPC service from the RProtoBuf descriptor pool
grpc_state	Channel connectivity state
grpc_status_codes	gRPC status codes
grpc_stream	Open a streaming call
grpc_tls	TLS credentials
grpc_version	Version of the linked gRPC C++ library
grpc_writes_done	Half-close a client stream
rgrpc-package	Asynchronous 'gRPC' Client and Server Runtime

Maintainer

Troy Hernandez <troy@cornball.ai>

Author(s)

Troy Hernandez [aut, cre] (ORCID: <https://orcid.org/0009-0005-4248-604X>), cornball.ai [cph]

`grpc_await` *Wait for events belonging to one call*

Description

Like `grpc_poll`, but scoped to a single call: only that call's events are returned, and the wait ends when one of them arrives rather than when anything at all does. Events for other calls stay queued in arrival order and are delivered by a later `grpc_poll` or `grpc_await` on their own call.

Usage

```
grpc_await(x, timeout_ms, max_events = 64L)
```

Arguments

x	A "grpc_call", "grpc_stream", or "grpc_request" object.
timeout_ms	How long to wait for an event belonging to x: 0 returns immediately, a positive value waits up to that many milliseconds, -1 waits indefinitely. Required, so that a stalled peer cannot silently become an unbounded wait; the call's own deadline is the other half of that guarantee.
max_events	Maximum events to return in this batch.

Details

This is the sequential-mode API, on both sides of the wire. It removes the demultiplexing a shared client or server otherwise pushes onto the caller: there is no way for a grpc_await loop to splice another call's messages into this one's payload, or to mistake another call's completion for this one's. The cost is that awaiting one call means not looking at the others, so a second call's deadline can pass unnoticed while this one is waited on. Drive genuinely concurrent work with grpc_poll and dispatch on id.

A call ends at its terminal event, so loop until that arrives: a "unary" event for grpc_call, a "stream_status" for grpc_stream. On a server "grpc_request" the awaited events are the ones that follow the request itself – "stream_msg", "client_done", "stream_writable", "cancelled" – since the request arrives from grpc_poll in the first place. A server call has no terminal event of its own: it ends when the handler ends it with grpc_reply or grpc_finish, so "client_done" is what a client-streaming handler loops to.

Two different clocks are in play, and the words for them are not self-distinguishing. deadline_ms on grpc_call or grpc_stream bounds the RPC: when it expires the call really is over, and the peer is told. timeout_ms here bounds only this wait. Setting the wait shorter than the deadline is normal and harmless; setting no deadline at all is what makes timeout_ms = -1 an unbounded wait. A server sees the client's deadline as deadline_ms on the request event.

An empty result means the wait ended with nothing for this call. It is not a failure and not an answer: an empty await leaves the call exactly as it was, so await it again to keep waiting, and the worst it costs is another trip round the loop. Because an empty result is possible, index the batch only after checking it – grpc_await(call, timeout_ms = 1000)[[1]] raises subscript out of bounds on a slow peer, which reads like a bug in the caller rather than the timeout it is.

Usually the wait ended because timeout_ms expired, but it also returns early when a signal interrupts it, so an empty result is not proof that timeout_ms of wall time passed. That early return is deliberate — it is what lets R process a Ctrl-C instead of ignoring it until the timeout runs out. It matters only if you are deriving elapsed time from the number of empty awaits; use a clock for deadlines, not a count.

Value

A list of events for x (possibly empty), in arrival order. Each event is a list with the same fields grpc_poll delivers for that kind of object.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## unary: keep waiting until the completion arrives; the call's own
## deadline_ms is what guarantees this loop ends
call <- grpc_call(cl, "/demo.Echo/Say", charToRaw("hi"), deadline_ms = 5000)
req <- next_request(srv)
grpc_reply(req, req$request)
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break                # empty just means "not yet"
}
evs[[1]]$status_name

## server handler: drain one client-streaming call without seeing any
## other call's messages
s <- grpc_stream(cl, "/demo.Echo/Collect", deadline_ms = 5000)
for (i in 1:3) grpc_send(s, as.raw(i))
grpc_writes_done(s)
req <- next_request(srv)
got <- list(req$request)
repeat {
  grpc_read(req)
  evs <- grpc_await(req, timeout_ms = 1000L)
  for (ev in evs) if (ev$type == "stream_msg") got <- c(got, list(ev$request))
  if (length(Filter(function(e) e$type %in% c("client_done", "cancelled"), evs))) break
}
grpc_reply(req, as.raw(length(got)))
length(got)

grpc_close(cl)
grpc_close(srv)
```

`grpc_call` *Start a unary call*

Description

Starts an asynchronous unary RPC. The request is opaque bytes (a serialized protocol buffer, e.g. from `RProtoBuf`'s `serialize()`). The call completes via `grpc_poll`.

Usage

```
grpc_call(
  client,
  method,
  request,
  deadline_ms = NULL,
  metadata = NULL,
  wait_for_ready = FALSE
)
```

Arguments

client	A "grpc_client" object.
method	Full method path, e.g. "/runtime.v1.RuntimeService/Version", or a "grpc_method" object for a typed call.
request	Raw vector with the serialized request message, or an RProtoBuf Message to serialize.
deadline_ms	Optional deadline in milliseconds; on expiry the call completes with status DEADLINE_EXCEEDED.
metadata	Optional named character vector of request metadata.
wait_for_ready	If TRUE, queue the call until the channel connects instead of failing fast with UNAVAILABLE.

Details

Typed calls: when method is a "grpc_method" (from grpc_method) and request is an RProtoBuf Message, the request type is validated against the method's input_type before sending, and the completion delivered by grpc_poll carries the decoded response as response_message. Streaming methods are refused here; open them with grpc_stream.

Value

An object of class "grpc_call": a list with the client, the call id that its completion event will carry, and the method path. Wait for the completion with grpc_await or grpc_poll.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
```

```
}

call <- grpc_call(cl, "/demo.Echo/Say", charToRaw("ping"), deadline_ms = 5000)
class(call)

## echo it back from the server half of the same process
req <- next_request(srv)
grpc_reply(req, req$request)

repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name
rawToChar(evs[[1]]$response)

## a call nobody answers ends at its deadline
call <- grpc_call(cl, "/demo.Echo/Say", raw(0), deadline_ms = 200)
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name

grpc_close(cl)
grpc_close(srv)
```

`grpc_cancel` *Cancel an in-flight call or stream*

Description

Requests cancellation. The call or stream still completes through `grpc_poll`, normally with status `CANCELLED`. Cancelling something already completed is a no-op. On a server, a `"grpc_request"` can be cancelled as the hard escalation when even an abortive `grpc_finish` cannot get its status past a peer that has stopped reading; the peer sees `CANCELLED`.

Usage

```
grpc_cancel(x)
```

Arguments

`x`	A `"grpc_call"`, `"grpc_stream"`, or `"grpc_request"` object.

Details

For a `"grpc_request"` the (invisible) return is `TRUE` if cancellation was requested on a live call and `FALSE` if the call was already terminal. `FALSE` guarantees no further messages can be delivered on the stream; it is *not* a receipt that any terminal status reached the peer.

Value

Invisibly. For a client "grpc_call" or "grpc_stream", NULL: the function is called for its side effect, and the outcome arrives as the call's CANCELLED completion. For a server "grpc_request", a logical scalar: TRUE if cancellation was requested on a live call, FALSE if the call was already terminal.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## a call the server holds without answering: cancel it instead of
## waiting for its deadline
call <- grpc_call(cl, "/demo.Echo/Say", raw(0), deadline_ms = 60000)
req <- next_request(srv)
grpc_cancel(call)
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name

## the server hears of it as a "cancelled" event, and the request can no
## longer be answered
repeat {
  evs <- grpc_await(req, timeout_ms = 1000L)
  if (length(Filter(function(e) e$type == "cancelled", evs))) break
}
(grpc_reply(req, raw(0)))               # FALSE
(grpc_cancel(req))                      # FALSE: already terminal

grpc_close(cl)
grpc_close(srv)
```

---

grpc_client *Create an asynchronous gRPC client*

Description

Opens a channel to `target` and starts the client's completion machinery: a background thread that drains the gRPC completion queue and signals a wake descriptor. The background thread never calls the R API; completions are received on the R main thread via `grpc_poll`.

Usage

```
grpc_client(
  target,
  credentials = NULL,
  keepalive_ms = NULL,
  keepalive_timeout_ms = NULL
)
```

Arguments

`target`	Server address, e.g. `"localhost:50051"` or `"unix:///run/containerd/containerd.sock"`.
`credentials`	`NULL` for a plaintext channel, or a `grpc_tls` object.
`keepalive_ms`	Interval of transport inactivity after which an HTTP/2 keepalive ping is sent. `NULL` (default) disables keepalive.
`keepalive_timeout_ms`	Time to wait for a ping answer before the connection is declared dead.

Details

Keepalive: with `keepalive_ms` set, the client pings the peer every `keepalive_ms` of transport inactivity and declares the connection dead `keepalive_timeout_ms` after an unanswered ping (gRPC's default timeout is 20000). Pings are enabled without payload data and without active calls, so a quiet connection is genuinely watched. The server must tolerate the cadence: see `min_ping_interval_ms` in `grpc_server` — gRPC's server default kills clients that ping more often than every 5 minutes.

Value

An object of class `"grpc_client"`: a list holding the native channel handle (`ptr`) and the `target`. Pass it to `grpc_call`, `grpc_stream`, `grpc_poll`, and `grpc_close`.

Forking

A client must not be used across a `fork()`. The background completion thread does not survive forking, so in the child there is nothing left to drain the completion queue: calls are posted and never complete. Worse, using an inherited client in the child also breaks it in the *parent* — the parent's subsequent calls fail with `DEADLINE_EXCEEDED`. Setting `GRPC_ENABLE_FORK_SUPPORT=1` does not change any of this.

This matters because `parallel::mclapply` and much of the mirai ecosystem fork. Measured behaviour (`tools/fork-probe.sh`):

- Child uses the parent's client: the call never completes. A wait with a timeout returns empty; a wait without one does not return. Set `deadline_ms` and you get `DEADLINE_EXCEEDED` instead of a hang.
- Parent afterwards: broken, `DEADLINE_EXCEEDED`.
- Forking alone, with the child never touching the client: harmless, the parent keeps working.
- Child opens its *own* client after the fork: works normally.

So the rule is to open the client after forking, once per process, and never to let a forked worker inherit one. Fork safety is a non-goal; this is documented behaviour, not a defect to be fixed.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
grpc_state(cl)                          # "IDLE": nothing has connected yet

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

call <- grpc_call(cl, "/demo.Echo/Say", as.raw(1:4), deadline_ms = 5000)

## answer it from the server half of the same process
req <- next_request(srv)
grpc_reply(req, req$request)

## completions arrive through grpc_poll(), or per call through grpc_await()
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name
evs[[1]]$response

grpc_close(cl)
grpc_close(srv)
```

`grpc_close` *Shut down a client or server*

Description

Cancels outstanding work, shuts down the completion queue, joins the completion thread, and releases the transport. Undelivered events are discarded. Closing twice is a no-op; the finalizer performs the same shutdown if the object is garbage collected unclosed.

Usage

```
grpc_close(x)
```

Arguments

x	A "grpc_client" or "grpc_server" object.

Value

No return value (NULL, invisibly); called for its side effect of shutting the object down.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
grpc_close(cl)
grpc_close(srv)
grpc_close(srv)                        # closing twice is a no-op
```

---

grpc_decode *Decode protocol buffer bytes to a message*

Description

Thin wrapper over RProtoBuf::read() for decoding a request or response payload against a type from the runtime descriptor pool.

Usage

```
grpc_decode(bytes, type)
```

Arguments

bytes	Raw vector, e.g. the request field of a server event or the response field of a client completion.
type	Fully qualified message type name, e.g. an input_type from a "grpc_method".

Value

An RProtoBuf Message of type type, with the fields decoded from bytes.

Examples

```
if (requireNamespace("RProtoBuf", quietly = TRUE)) {
  RProtoBuf::readProtoFiles2("health.proto",
      protoPath = system.file("proto", "health", package = "rgrpc"))
  ## bytes as they arrive in a request or response event
  msg <- RProtoBuf::P("grpc.health.v1.HealthCheckRequest")$new(service = "demo")
  bytes <- RProtoBuf::serialize(msg, NULL)
  decoded <- grpc_decode(bytes, "grpc.health.v1.HealthCheckRequest")
  print(decoded$service)
}
```

---

grpc_fd *Completion wakeup file descriptor*

Description

Returns the object's wake descriptor: a pipe file descriptor on Unix, a socket descriptor on Windows. It is readable exactly while events are queued, so an event loop can wake on it instead of polling, e.g. with `later::later_fd()`. Do not read from this descriptor; `grpc_poll` drains it and leaves it readable if it returned a partial batch.

Usage

```
grpc_fd(x)
```

Arguments

x	A "grpc_client" or "grpc_server" object.

Value

Integer scalar: the wake descriptor, valid until the object is closed.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
grpc_fd(cl)
grpc_fd(srv)
## hand the descriptor to an event loop instead of polling, e.g.
##   later::later_fd(function(ready) grpc_poll(cl), readfds = grpc_fd(cl))
grpc_close(cl)
grpc_close(srv)
```

grpc_finish *End a server stream*

Description

Sends the terminal status for a streaming call after any messages queued with grpc_send have drained. For unary replies use grpc_reply, which sends a payload and the status in one step. Returns (invisibly) TRUE if accepted, FALSE if the call is no longer answerable.

Usage

```
grpc_finish(request, status = 0L, message = "", metadata = NULL, drain = TRUE)
```

Arguments

request	A "grpc_request" event from grpc_poll.
status	Integer status code or name from grpc_status_codes.
message	Optional error detail string for non-OK status.
metadata	Optional named character vector sent as trailing metadata.
drain	If TRUE (default), queued messages are delivered before the status; if FALSE, they are discarded and the status is prioritized.

Details

With drain = FALSE the close is abortive: queued messages are discarded and the terminal status goes out first. This is for fencing — e.g. ABORTED on session replacement — where delivering queued-but-stale messages to the peer would be wrong and waiting behind them (potentially forever, if the peer has stopped reading) delays the fence. One already-posted message cannot be recalled, so the status can still wait for that single in-flight write; if the peer's flow-control window is exhausted even that may not complete, and grpc_cancel on the request is the hard escalation (the peer then sees CANCELLED rather than this status).

Value

Invisibly, a logical scalar: TRUE if the status was accepted for sending, FALSE if the call is no longer answerable.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
```

```
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## server streaming: the client sends one request and half-closes
s <- grpc_stream(cl, "/demo.Echo/Watch", deadline_ms = 5000)
grpc_send(s, as.raw(7))
grpc_writes_done(s)

## server: three messages, then the status with trailing metadata
req <- next_request(srv)
for (i in 1:3) grpc_send(req, as.raw(i))
grpc_finish(req, metadata = c("x-count" = "3"))

out <- list()
repeat {
  evs <- grpc_await(s, timeout_ms = 1000L)
  for (ev in evs) if (ev$kind == "stream_msg") out <- c(out, list(ev$response))
  st <- Filter(function(e) e$kind == "stream_status", evs)
  if (length(st)) break
}
length(out)
st[[1]]$status_name
st[[1]]$trailing_metadata

## an error status ends a stream without any payload
s <- grpc_stream(cl, "/demo.Echo/Watch", deadline_ms = 5000)
grpc_send(s, raw(1))
req <- next_request(srv)
grpc_finish(req, status = "NOT_FOUND", message = "no such stream")
repeat {
  evs <- grpc_await(s, timeout_ms = 1000L)
  st <- Filter(function(e) e$kind == "stream_status", evs)
  if (length(st)) break
}
st[[1]]$status_name
st[[1]]$message

grpc_close(cl)
grpc_close(srv)
```

`grpc_method` *Look up a method in a resolved service*

Description

Look up a method in a resolved service

Usage

```
grpc_method(service, name)
```

Arguments

service	A "grpc_service" from grpc_service.
name	Method name, e.g. "Version".

Value

A "grpc_method" object: name, path, input_type, output_type, client_streaming, server_streaming. Pass it as the method argument of grpc_call for typed calls.

Examples

```
if (requireNamespace("RProtoBuf", quietly = TRUE)) {
  RProtoBuf::readProtoFiles2("health.proto",
      protoPath = system.file("proto", "health", package = "rgrpc"))
  svc <- grpc_service("grpc.health.v1.HealthCheckRequest", "Health")
  m <- grpc_method(svc, "Check")
  print(m$path)
  print(m$input_type)
  print(grpc_method(svc, "Watch")$server_streaming)
}
```

---

grpc_pending *Number of pending operations*

Description

For a client, calls started but not yet delivered by grpc_poll. For a server, accepted calls not yet completed.

Usage

```
grpc_pending(x)
```

Arguments

x	A "grpc_client" or "grpc_server" object.

Value

Integer scalar: the number of in-flight operations, zero when nothing is outstanding.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

grpc_pending(cl)                        # 0
call <- grpc_call(cl, "/demo.Echo/Say", raw(0), deadline_ms = 5000)
grpc_pending(cl)                        # 1: started, not yet completed

req <- next_request(srv)
grpc_pending(srv)                       # 1: accepted, not yet answered
grpc_reply(req, raw(0))
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
grpc_pending(cl)                        # 0 again

grpc_close(cl)
grpc_close(srv)
```

---

grpc_poll *Receive completed events*

Description

Drains up to `max_events` pending events from a client or server.

Usage

```
grpc_poll(x, max_events = 64L, timeout_ms = 0L)
```

Arguments

`x`	A `"grpc_client"` or `"grpc_server"` object.
`max_events`	Maximum events to return in this batch.
`timeout_ms`	How long to wait if no event is ready: `0` returns immediately, a positive value waits up to that many milliseconds, `-1` waits indefinitely.

Details

For a "grpc_client", each event is a completed call: a list with id, status (integer, see grpc_status_codes), status_name, message, response (raw vector, NULL unless the status is OK), initial_metadata, and trailing_metadata.

For a "grpc_server", each event is either an incoming request (class "grpc_request": type = "request", id, method, request raw vector, metadata, deadline_ms, answered via grpc_reply) or a cancellation notice (type = "cancelled", id) for a request the peer abandoned before it was answered.

An empty result means the wait ended with nothing queued at that moment; it says nothing about outstanding work, which ends only when its terminal event is delivered ("unary" for a call, "stream_status" for a stream). Loop until that event rather than treating an empty batch as the end of a call; grpc_pending reports what is still in flight.

An empty result does *not* prove timeout_ms elapsed. The wait also returns early when a signal interrupts it, which is deliberate: handing control back to R is what lets an interrupt be processed, where restarting the wait would swallow a Ctrl-C for the rest of the timeout. Nothing changes for the ordinary caller — empty still means "nothing yet, go round again" — but do not build a deadline by counting empty returns and multiplying by timeout_ms, because that arithmetic silently undercounts on an interrupted wait. Read a clock instead.

One queue serves the whole client or server, so a batch can mix events from every call in flight, and they arrive in completion order rather than the order the calls were started. Dispatch on id, in both directions: the first event in a batch need not belong to the call you just started, and not every "stream_msg" in it belongs to the stream you are reading. Taking events[[1]] as the answer to a unary call, and accumulating every "stream_msg" into one stream's payload, are the same assumption – one queue per call – and it does not hold. This bites hardest after a stream is abandoned unread, since its queued messages keep arriving; grpc_cancel bounds how many more are produced but cannot recall events already queued.

Value

A list of events (possibly empty), each a list as described above; server request events additionally carry class "grpc_request".

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## three calls in flight at once
calls <- lapply(1:3, function(i) {
  grpc_call(cl, "/demo.Echo/Say", as.raw(i), deadline_ms = 5000)
})

## server: one queue for every call; answer requests as they arrive
## (5 s of silence is an error rather than an endless wait)
answered <- 0L
while (answered < 3L) {
  evs <- grpc_poll(srv, timeout_ms = 5000L)
  if (!length(evs)) stop("no request within 5 s")
```

```
  for (ev in evs) {
    if (ev$type == "request") {
      grpc_reply(ev, ev$request)
      answered <- answered + 1L
    }
  }
}

## client: completions come back in completion order, so match on id
ids <- vapply(calls, function(x) x$id, numeric(1))
got <- vector("list", length(calls))
while (any(vapply(got, is.null, logical(1)))) {
  for (ev in grpc_poll(cl, timeout_ms = 100L)) {
    if (ev$kind == "unary") got[[match(ev$id, ids)]] <- ev$response
  }
}
unlist(got)

grpc_close(cl)
grpc_close(srv)
```

---

grpc_read *Pull the next inbound message on a server stream*

Description

Posts one read on a client- or bidirectionally-streaming call. The result arrives through grpc_poll as a "stream_msg" event (with request bytes), or "client_done" when the peer has half-closed. One read at a time: returns (invisibly) FALSE if a read is already in flight or the call is over.

Usage

```
grpc_read(request)
```

Arguments

request	A "grpc_request" event from grpc_poll.

Value

Invisibly, a logical scalar: TRUE if a read was posted, FALSE if one is already in flight or the call is over.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
```

```
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

s <- grpc_stream(cl, "/demo.Echo/Collect", deadline_ms = 5000)
for (i in 1:4) grpc_send(s, as.raw(i))
grpc_writes_done(s)

req <- next_request(srv)
got <- as.integer(req$request)          # the first message rides on the request
(grpc_read(req))                        # TRUE: one read is now in flight
(grpc_read(req))                        # FALSE: one read at a time
## each read yields one "stream_msg", or "client_done" at the end
repeat {
  evs <- grpc_await(req, timeout_ms = 1000L)
  for (ev in evs) if (ev$type == "stream_msg") got <- c(got, as.integer(ev$request))
  if (length(Filter(function(e) e$type %in% c("client_done", "cancelled"), evs))) break
  grpc_read(req)                        # post the next read
}
got
grpc_reply(req, as.raw(length(got)))

grpc_close(cl)
grpc_close(srv)
```

`grpc_reply` *Answer an incoming request*

Description

Completes a request received from `grpc_poll` on a server, either with a response payload (status `OK`) or with an error status. Returns (invisibly) `TRUE` if the reply was accepted, or `FALSE` if the call is no longer answerable (the peer cancelled, timed out, or the request was already answered).

Usage

```
grpc_reply(
  request,
  response = NULL,
  status = 0L,
  message = "",
  metadata = NULL
)
```

Arguments

request	A "grpc_request" event from grpc_poll.
response	Raw vector with the serialized response message, or an RProtoBuf Message to serialize. Required when status is OK; ignored otherwise.
status	Integer status code or name from grpc_status_codes, e.g. "NOT_FOUND".
message	Optional error detail string for non-OK status.
metadata	Optional named character vector sent as trailing metadata.

Value

Invisibly, a logical scalar: TRUE if the reply was accepted for sending, FALSE if the call is no longer answerable.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## a payload reply (status OK), with trailing metadata
call <- grpc_call(cl, "/demo.Echo/Say", charToRaw("hi"), deadline_ms = 5000)
req <- next_request(srv)
(grpc_reply(req, req$request, metadata = c("x-served-by" = "example")))
(grpc_reply(req, req$request))          # FALSE: already answered
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name
evs[[1]]$trailing_metadata

## an error status needs no payload
call <- grpc_call(cl, "/demo.Echo/Say", raw(0), deadline_ms = 5000)
req <- next_request(srv)
grpc_reply(req, status = "NOT_FOUND", message = "no such thing")
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
evs[[1]]$status_name
evs[[1]]$message
```

```
is.null(evs[[1]]$response)

grpc_close(cl)
grpc_close(srv)
```

---

grpc_send *Send a message on a stream*

Description

Enqueues one outbound message on a bounded write queue. For a client "grpc_stream", this is the request direction; for a server "grpc_request", the response direction. Returns (invisibly) TRUE if the message was queued, or FALSE if the queue is full (backpressure: wait for the "stream_writable" event and retry) or the stream can no longer accept writes.

Usage

```
grpc_send(x, msg, ...)
```

Arguments

x	A "grpc_stream" (client) or "grpc_request" (server) object.
msg	Raw vector, or an RProtoBuf Message to serialize (validated against the method's input_type on a typed client stream).
...	Reserved.

Value

Invisibly, a logical scalar: TRUE if the message was queued for sending, FALSE if the write queue is full or the stream no longer accepts writes (half-closed, finished, cancelled, or past its deadline).

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## client streaming: queue five messages, then half-close
s <- grpc_stream(cl, "/demo.Echo/Collect", deadline_ms = 5000)
for (i in 1:5) grpc_send(s, as.raw(i))
```

```
grpc_writes_done(s)
(grpc_send(s, as.raw(6)))              # FALSE: no writes after the half-close

## server: count what arrives (the first message rides on the request
## event), then answer once, unary-style
req <- next_request(srv)
n <- 1L
repeat {
  grpc_read(req)
  evs <- grpc_await(req, timeout_ms = 1000L)
  n <- n + length(Filter(function(e) e$type == "stream_msg", evs))
  if (length(Filter(function(e) e$type %in% c("client_done", "cancelled"), evs))) break
}
grpc_reply(req, as.raw(n))

## client: the reply is one "stream_msg", followed by the "stream_status"
reply <- NULL
repeat {
  evs <- grpc_await(s, timeout_ms = 1000L)
  for (ev in evs) if (ev$kind == "stream_msg") reply <- ev$response
  if (length(Filter(function(e) e$kind == "stream_status", evs))) break
}
as.integer(reply)

## a full write queue makes grpc_send() return FALSE; wait for the
## "stream_writable" event (grpc_await() on the stream) and retry
grpc_close(cl)
grpc_close(srv)
```

`grpc_server` *Create an asynchronous generic gRPC server*

Description

Binds `address`, registers the generic asynchronous service (any method name is accepted; there are no generated stubs), and starts the server's completion machinery: a background thread that drains the completion queue and signals a wake descriptor. Incoming requests are received on the R main thread via `grpc_poll` and answered with `grpc_reply`.

Usage

```
grpc_server(
  address = "127.0.0.1:0",
  credentials = NULL,
  accept_window = 8L,
  max_active = 256L,
  keepalive_ms = NULL,
  keepalive_timeout_ms = NULL,
  min_ping_interval_ms = NULL
)
```

Arguments

address	Bind address, e.g. "127.0.0.1:0" for an ephemeral TCP port (see grpc_server_port) or "unix:/path/to.sock" for a unix-domain socket.
credentials	NULL for a plaintext listener, or a grpc_tls object (needs cert_file and key_file; require_client_cert = TRUE for mTLS).
accept_window	Outstanding accept slots (burst capacity).
max_active	Bound on concurrently active calls.
keepalive_ms	Interval of transport inactivity after which the server pings a client. NULL (default) disables keepalive.
keepalive_timeout_ms	Time to wait for a ping answer before the connection is declared dead.
min_ping_interval_ms	Minimum interval between client pings the server tolerates without counting a ping strike.

Details

Backpressure: at most accept_window accept slots are kept outstanding, and no new slot is posted while active calls plus outstanding slots would exceed max_active. Excess incoming calls queue in the transport until capacity frees up.

Request events carry the transport peer address and, on a TLS listener with require_client_cert, the verified peer_identity (the client certificate's identity values).

Keepalive: keepalive_ms/keepalive_timeout_ms make the server ping quiet clients, mirroring grpc_client. min_ping_interval_ms is the tolerance for *client* pings: gRPC's server default allows one unsolicited ping per 5 minutes and kills faster clients with a too_many_pings GOAWAY, so a deployment where clients keep 10-second heartbeats must lower this to at most the client ping interval.

Value

An object of class "grpc_server": a list holding the native server handle (ptr) and the bind address. Pass it to grpc_poll, grpc_server_port, and grpc_close.

Forking

A server must not be used across a fork(), for the same reason a client must not: the completion thread does not survive forking. See the Forking section of grpc_client, which documents the measured behaviour. Create the server in the process that will serve it.

Examples

```
srv <- grpc_server("127.0.0.1:0")
grpc_server_port(srv)

## a client in the same process makes one call
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
call <- grpc_call(cl, "/demo.Echo/Say", charToRaw("hello"), deadline_ms = 5000)
```

```
## the server sees it as a request event; there are no handlers to
## register, the method name arrives with the event. One queue serves
## every call, so step over other events; 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}
req <- next_request(srv)
req$method
grpc_reply(req, req$request)                        # echo it back

## the client collects the completion
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break                            # empty means "not yet"
}
evs[[1]]$status_name
rawToChar(evs[[1]]$response)

grpc_close(cl)
grpc_close(srv)
```

`grpc_server_port` *Bound TCP port of a server*

Description

The port actually bound, useful with an ephemeral `"host:0"` bind address. Meaningless for unix-domain sockets.

Usage

```
grpc_server_port(server)
```

Arguments

`server`	A `"grpc_server"` object.

Value

Integer scalar: the TCP port the server listens on.

Examples

```
srv <- grpc_server("127.0.0.1:0")
grpc_server_port(srv)
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
grpc_close(cl)
grpc_close(srv)
```

---

grpc_service *Resolve a gRPC service from the RProtoBuf descriptor pool*

Description

Looks up the service definitions in the `.proto` file that defines `anchor`, a message type from that file (schemas are loaded at runtime with `RProtoBuf::readProtoFiles2()`; there is no generated code). Returns the service with its methods resolved: full method paths, input and output message types, and streaming flags.

Usage

```
grpc_service(anchor, service = NULL)
```

Arguments

`anchor`	A message type from the `.proto` file that defines the service: a type name like `"runtime.v1.VersionRequest"`, an `RProtoBuf` `Descriptor`, or a `Message`.
`service`	Service to select, by short or fully qualified name. May be omitted when the file defines exactly one service.

Value

An object of class `"grpc_service"`: `name`, `package`, and a named list `methods` of `"grpc_method"` objects.

Examples

```
if (requireNamespace("RProtoBuf", quietly = TRUE)) {
  ## the gRPC health-checking schema ships with the package
  RProtoBuf::readProtoFiles2("health.proto",
      protoPath = system.file("proto", "health", package = "rgrpc"))
  svc <- grpc_service("grpc.health.v1.HealthCheckRequest", "Health")
  print(svc$name)
  print(names(svc$methods))
}
```

grpc_state *Channel connectivity state*

Description

Observes (without provoking a connection attempt) the client channel's connectivity state.

Usage

```
grpc_state(client)
```

Arguments

client	A "grpc_client" object.

Details

For deeper transport diagnostics, gRPC's built-in tracing applies to this package unchanged: set the GRPC_TRACE and GRPC_VERBOSITY environment variables before the package loads, e.g. GRPC_TRACE=http,connectivity_s GRPC_VERBOSITY=debug.

Value

A character scalar, one of "IDLE", "CONNECTING", "READY", "TRANSIENT_FAILURE", "SHUTDOWN": the channel's connectivity state at the moment of the call.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))
grpc_state(cl)                          # "IDLE": channels connect on first use

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

call <- grpc_call(cl, "/demo.Echo/Say", raw(0), deadline_ms = 5000)
req <- next_request(srv)
grpc_reply(req, raw(0))
repeat {
  evs <- grpc_await(call, timeout_ms = 1000L)
  if (length(evs)) break
}
```

```
grpc_state(cl)                          # "READY" once a call has gone through

grpc_close(cl)
grpc_close(srv)
```

`grpc_status_codes` *gRPC status codes*

Description

Named integer vector mapping gRPC status names to their wire codes.

Usage

```
grpc_status_codes
```

Format

An object of class `integer` of length 17.

`grpc_stream` *Open a streaming call*

Description

Opens a client-, server-, or bidirectionally-streaming RPC. Messages are sent with `grpc_send`, the request direction is half-closed with `grpc_writes_done`, and everything inbound arrives through `grpc_poll` on the client: `"stream_msg"` events per message (with `response` bytes, plus `response_message` decoded on a typed stream), `"stream_writable"` when the send queue drains, and a final `"stream_status"` with status and trailing metadata.

Usage

```
grpc_stream(
  client,
  method,
  deadline_ms = NULL,
  metadata = NULL,
  wait_for_ready = FALSE,
  read_buffer = 16L,
  write_buffer = 16L
)
```

Arguments

client	A "grpc_client" object.
method	Full method path, or a "grpc_method" object for a typed stream (any streaming shape).
deadline_ms	Optional deadline in milliseconds for the whole stream.
metadata	Optional named character vector of request metadata.
wait_for_ready	If TRUE, wait for the channel to connect instead of failing fast.
read_buffer	Bound on undelivered inbound messages.
write_buffer	Bound on queued outbound messages.

Details

Inbound flow control is automatic and bounded: at most read_buffer undelivered messages are held; beyond that the stream stops reading until grpc_poll drains, and HTTP/2 backpressure propagates to the peer.

A stream runs until its "stream_status", not until the caller loses interest: dropping the returned object stops nothing, and the stream's queued messages keep surfacing in grpc_poll alongside later calls on the same client. Read every stream to its status, or grpc_cancel the ones you are done with, and dispatch events on id either way.

Value

An object of class "grpc_stream": a list with the client, the stream id that its events will carry, and the method path. Send on it with grpc_send, half-close with grpc_writes_done, and receive with grpc_await or grpc_poll.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

## client: three messages out, then half-close the request direction
s <- grpc_stream(cl, "/demo.Echo/Chat", deadline_ms = 5000)
for (i in 1:3) grpc_send(s, as.raw(i))
grpc_writes_done(s)

## server: the first message rides on the request event; read the rest,
## then echo each one back doubled and end the stream
```

```
req <- next_request(srv)
got <- list(req$request)
repeat {
  grpc_read(req)
  evs <- grpc_await(req, timeout_ms = 1000L)
  for (ev in evs) if (ev$type == "stream_msg") got <- c(got, list(ev$request))
  if (length(Filter(function(e) e$type %in% c("client_done", "cancelled"), evs))) break
}
for (m in got) grpc_send(req, as.raw(as.integer(m) * 2L))
grpc_finish(req)

## client: responses arrive as "stream_msg" events up to the "stream_status"
out <- list()
repeat {
  evs <- grpc_await(s, timeout_ms = 1000L)
  for (ev in evs) if (ev$kind == "stream_msg") out <- c(out, list(ev$response))
  st <- Filter(function(e) e$kind == "stream_status", evs)
  if (length(st)) break
}
unlist(out)
st[[1]]$status_name

grpc_close(cl)
grpc_close(srv)
```

`grpc_tls` *TLS credentials*

Description

Builds a credentials object for `grpc_client` or `grpc_server`. PEM files are read at construction time.

Usage

```
grpc_tls(
  ca_file = NULL,
  cert_file = NULL,
  key_file = NULL,
  require_client_cert = FALSE,
  target_name_override = NULL
)
```

Arguments

`ca_file`	Path to a PEM CA bundle (trust anchor).
`cert_file`	Path to a PEM certificate chain (own identity).
`key_file`	Path to the PEM private key for `cert_file`.

require_client_cert
: Server side: require and verify a client certificate (mTLS). Needs ca_file.

target_name_override
: Client side: hostname to verify the server certificate against instead of the dialed target. For testing with certificates whose name does not match the address; do not use in production.

Details

For a client: ca_file pins the CA that must have signed the server certificate; cert_file/key_file present a client identity (mTLS). For a server: cert_file/key_file are its identity; with require_client_cert = TRUE the server demands a client certificate signed by ca_file, and the verified identity appears on request events as peer_identity.

Value

An object of class "grpc_tls": a list with the PEM contents read from the files (ca, cert, key; NULL where no file was given), require_client_cert, and target_name_override. Pass it as the credentials argument of grpc_client or grpc_server.

Examples

```
## needs certificate files, which only exist in a real deployment (the
## package's TLS tests generate throwaway ones with the openssl
## command-line tool), so this example is not run
## Not run:
## client pinning a CA and presenting its own identity (mTLS)
creds <- grpc_tls(ca_file = "ca.pem",
                  cert_file = "client.pem", key_file = "client.key")
cl <- grpc_client("node1:41900", credentials = creds)

## server demanding client certificates signed by that CA
srv <- grpc_server("0.0.0.0:41900",
                   credentials = grpc_tls(ca_file = "ca.pem",
                                          cert_file = "server.pem",
                                          key_file = "server.key",
                                          require_client_cert = TRUE))

## End(Not run)
```

---

grpc_version *Version of the linked gRPC C++ library*

Description

Returns the version string of the system gRPC C++ library this package was built against.

Usage

```
grpc_version()
```

Value

A character string, e.g. "1.51.1".

Examples

```
grpc_version()
```

---

grpc_writes_done *Half-close a client stream*

Description

Signals that no further messages will be sent. Queued messages are flushed first. Returns (invisibly) FALSE if already half-closed.

Usage

```
grpc_writes_done(stream)
```

Arguments

stream	A "grpc_stream" object.

Value

Invisibly, a logical scalar: TRUE if the half-close was posted, FALSE if the stream was already half-closed or is over.

Examples

```
srv <- grpc_server("127.0.0.1:0")
cl <- grpc_client(sprintf("127.0.0.1:%d", grpc_server_port(srv)))

## the next request event; the server has one queue for every call, so
## other events are stepped over, and 5 s of silence is an error
next_request <- function(srv) {
  repeat {
    evs <- grpc_poll(srv, timeout_ms = 5000L)
    if (!length(evs)) stop("no request within 5 s")
    for (ev in evs) if (ev$type == "request") return(ev)
  }
}

s <- grpc_stream(cl, "/demo.Echo/Collect", deadline_ms = 5000)
grpc_send(s, as.raw(1))
(grpc_writes_done(s))                  # TRUE: request direction closed
(grpc_writes_done(s))                  # FALSE: already half-closed
(grpc_send(s, as.raw(2)))              # FALSE: no writes after the half-close
```

```
## the server reads past the first message and sees "client_done"
req <- next_request(srv)
grpc_read(req)
repeat {
  evs <- grpc_await(req, timeout_ms = 1000L)
  done <- Filter(function(e) e$type %in% c("client_done", "cancelled"), evs)
  if (length(done)) break
}
done[[1]]$type
grpc_reply(req, raw(0))

grpc_close(cl)
grpc_close(srv)
```

Index