Pitaya Documentation

TFGCo

May 07, 2020

Contents:

CHAPTER 1

Overview

Pitaya is an easy to use, fast and lightweight game server framework inspired by starx and pomelo and built on top of nano's networking library.

The goal of pitaya is to provide a basic, robust development framework for distributed multiplayer games and server-side applications.

1.1 Features

- **User sessions** - Pitaya has support for user sessions, allowing binding sessions to user ids, setting custom data and retrieving it in other places while the session is active
- **Cluster support** - Pitaya comes with support to default service discovery and RPC modules, allowing communication between different types of servers with ease
- **WS and TCP listeners** - Pitaya has support for TCP and Websocket acceptors, which are abstracted from the application receiving the requests
- **Handlers and remotes** - Pitaya allows the application to specify its handlers, which receive and process client messages, and its remotes, which receive and process RPC server messages. They can both specify custom init, afterinit and shutdown methods
- **Message forwarding** - When a server receives a handler message it forwards the message to the server of the correct type
- **Client library SDK** - libpitaya is the official client library SDK for Pitaya
- **Monitoring** - Pitaya has support for Prometheus and statsd by default and accepts other custom reporters that implement the Reporter interface
- **Open tracing compatible** - Pitaya is compatible with open tracing, so using Jaeger or any other compatible tracing framework is simple
- **Custom modules** - Pitaya already has some default modules and supports custom modules as well

- **Custom serializers** - Pitaya natively supports JSON and Protobuf messages and it is possible to add other custom serializers as needed
- **Write compatible servers in other languages** - Using libpitaya-cluster its possible to write pitaya-compatible servers in other languages that are able to register in the cluster and handle RPCs, there's already a csharp library that's compatible with unity and a WIP of a python library in the repo.
- **REPL Client for development/debugging** - Pitaya-cli is a REPL client that can be used for making development and debugging of pitaya servers easier.
- **Bots for integration/stress tests** - Pitaya-bot is a server test framework that can easily copy users behaviour to test corner case scenarios, which can validate the responses received, or make massive accesses into pitaya servers.

1.2 Architecture

Pitaya was developed considering modularity and extendability at its core, while providing solid basic functionalities to abstract client interactions to well defined interfaces. The full API documentation is available in Godoc format at godoc.

1.3 Who's Using it

Well, right now, only us at TFG Co, are using it, but it would be great to get a community around the project. Hope to hear from you guys soon!

1.4 How To Contribute?

Just the usual: Fork, Hack, Pull Request. Rinse and Repeat. Also don't forget to include tests and docs (we are very fond of both).

CHAPTER 2

Features

Pitaya has a modular and configurable architecture which helps to hide the complexity of scaling the application and managing clients' sessions and communications.

Some of its core features are described below.

2.1 Frontend and backend servers

In cluster mode servers can either be a frontend or backend server.

Frontend servers must specify listeners for receiving incoming client connections. They are capable of forwarding received messages to the appropriate servers according to the routing logic.

Backend servers don't listen for connections, they only receive RPCs, either forwarded client messages (sys rpc) or RPCs from other servers (user rpc).

2.2 Groups

Groups are structures which store information about target users and allows sending broadcast messages to all users in the group and also multicast messages to a subset of the users according to some criteria.

They are useful for creating game rooms for example, you just put all the players from a game room into the same group and then you'll be able to broadcast the room's state to all of them.

2.3 Listeners

Frontend servers must specify one or more acceptors to handle incoming client connections, Pitaya comes with TCP and Websocket acceptors already implemented, and other acceptors can be added to the application by implementing the acceptor interface.

2.4 Acceptor Wrappers

Wrappers can be used on acceptors, like TCP and Websocket, to read and change incoming data before performing the message forwarding. To create a new wrapper just implement the Wrapper interface (or inherit the struct from BaseWrapper) and add it into your acceptor by using the WithWrappers method. Next there are some examples of acceptor wrappers.

2.4.1 Rate limiting

Read the incoming data on each player's connection to limit requests troughput. After the limit is exceeded, requests are dropped until slots are available again. The requests count and management is done on player's connection, therefore it happens even before session bind. The used algorithm is the Leaky Bucket. This algorithm represents a leaky bucket that has its output flow slower than its input flow. It saves each request timestamp in a `slot` (of a total of `limit` slots) and this slot is freed again after `interval`. For example: if `limit` of 1 request in an `interval` of 1 second, when a request happens at 0.2s the next request will only be handled by pitaya after 1s (at 1.2s).

```
0     request
|--------|
   0.2s
0                  available again
|-------------------------|
|- 0.2s -|----- 1s ------|
```

2.5 Message forwarding

When a server instance receives a client message, it checks the target server type by looking at the route. If the target server type is different from the receiving server type, the instance forwards the message to an appropriate server instance of the correct type. The client doesn't need to take any action to forward the message, this process is done automatically by Pitaya.

By default the routing function chooses one instance of the target server type at random. Custom functions can be defined to change this behavior.

2.6 Message push

Messages can be pushed to users without previous information about either session or connection status. These push messages have a route (so that the client can identify the source and treat properly), the message, the target ids and the server type the client is expected to be connected to.

2.7 Modules

Modules are entities that can be registered to the Pitaya application and must implement the defined interface. Pitaya is responsible for calling the appropriate lifecycle methods as needed, the registered modules can be retrieved by name.

Pitaya comes with a few already implemented modules, and more modules can be implemented as needed. The modules Pitaya has currently are:

2.7.1 Binary

This module starts a binary as a child process and pipes its stdout and stderr to info and error log messages, respectively.

2.7.2 Unique session

This module adds a callback for `OnSessionBind` that checks if the id being bound has already been bound in one of the other frontend servers.

2.7.3 Binding storage

This module implements functionality needed by the gRPC RPC implementation to enable the functionality of broadcasting session binds and pushes to users without knowledge of the servers the users are connected to.

2.8 Monitoring

Pitaya has support for metrics reporting, it comes with Prometheus and Statsd support already implemented and has support for custom reporters that implement the `Reporter` interface. Pitaya also comes with support for open tracing compatible frameworks, allowing the easy integration of Jaeger and others.

Some of the reported metrics reported by the `Reporter` include: number of connected clients, request duration and dropped messages.

2.9 Custom Metrics

Besides pitaya default monitoring, it is possible to create new metrics. If using only Statsd reporter, no configuration is needed. If using Prometheus, it is necessary do add a configuration specifying the metrics parameters. More details on doc and this example.

2.10 Pipelines

Pipelines are middlewares which allow methods to be executed before and after handler requests, they receive the request's context and request data and return the request data, which is passed to the next method in the pipeline.

2.11 RPCs

Pitaya has support for RPC calls when in cluster mode, there are two components to enable this, RPC client and RPC server. There are currently two options for using RPCs implemented for Pitaya, NATS and gRPC, the default is NATS.

There are two types of RPCs, *Sys* and *User*.

2.11.1 Sys RPCs

These are the RPCs done by the servers when forwarding handler messages to the appropriate server type.

2.11.2 User RPCs

User RPCs are done when the application actively calls a remote method in another server. The call can specify the ID of the target server or let Pitaya choose one according to the routing logic.

2.11.3 User Reliable RPCs

These are done when the application calls a remote using workers, that is, Pitaya retries the RPC if any error occurrs.

Important: the remote that is being called must be idempotent; also the ReliableRPC will not return the remote's reply since it is asynchronous, it only returns the job id (jid) if success.

2.12 Server operation mode

Pitaya has two types of operation: standalone and cluster mode.

2.12.1 Standalone mode

In standalone mode the servers don't interact with one another, don't use service discovery and don't have support to RPCs. This is a limited version of the framework which can be used when the application doesn't need to have different types of servers or communicate among them.

2.12.2 Cluster mode

Cluster mode is a more complete mode, using service discovery, RPC client and server and remote communication among servers of the application. This mode is useful for more complex applications, which might benefit from splitting the responsabilities among different specialized types of servers. This mode already comes with default services for RPC calls and service discovery.

2.13 Serializers

Pitaya has support for different types of message serializers for the messages sent to and from the client, the default serializer is the JSON serializer and Pitaya comes with native support for the Protobuf serializer as well. New serializers can be implemented by implementing the `serialize.Serializer` interface.

The desired serializer can be set by the application by calling the `SetSerializer` method from the `pitaya` package.

2.14 Service discovery

Servers operating in cluster mode must have a service discovery client to be able to work. Pitaya comes with a default client using etcd, which is used if no other client is defined. The service discovery client is responsible for registering the server and keeping the list of valid servers updated, as well as providing information about requested servers as needed.

2.15 Sessions

Every connection established by the clients has an associated session instance, which is ephemeral and destroyed when the connection closes. Sessions are part of the core functionality of Pitaya, because they allow asynchronous communication with the clients and storage of data between requests. The main features of sessions are:

- **ID binding** - Sessions can be bound to an user ID, allowing other parts of the application to send messages to the user without needing to know which server or connection the user is connected to
- **Data storage** - Sessions can be used for data storage, storing and retrieving data between requests
- **Message passing** - Messages can be sent to connected users through their sessions, without needing to have knowledge about the underlying connection protocol
- **Accessible on requests** - Sessions are accessible on handler requests in the context instance
- **Kick** - Users can be kicked from the server through the session's `Kick` method

Even though sessions are accessible on handler requests both on frontend and backend servers, their behavior is a bit different if they are a frontend or backend session. This is mostly due to the fact that the session actually lives in the frontend servers, and just a representation of its state is sent to the backend server.

A session is considered a frontend session if it is being accessed from a frontend server, and a backend session is accessed from a backend server. Each kind of session is better described below.

2.15.1 Frontend sessions

Sessions are associated to a connection in the frontend server, and can be retrieved by session ID or bound user ID in the server the connection was established, but cannot be retrieved from a different server.

Callbacks can be added to some session lifecycle changes, such as closing and binding. The callbacks can be on a per-session basis (with `s.OnClose`) or for every session (with `OnSessionClose`, `OnSessionBind` and `OnAfterSessionBind`).

2.15.2 Backend sessions

Backend sessions have access to the sessions through the handler's methods, but they have some limitations and special characteristics. Changes to session variables must be pushed to the frontend server by calling `s.PushToFront` (this is not needed for `s.Bind` operations), setting callbacks to session lifecycle operations is also not allowed. One can also not retrieve a session by user ID from a backend server.

CHAPTER 3

Communication

In this section we will describe in detail the communication process between the client and the server. From establishing the connection, sending a request and receiving a response. The example is going to assume the application is running in cluster mode and that the target server is not the same as the one the client is connected to.

3.1 Establishing the connection

The overview of what happens when a client connects and makes a request is:

- Establish low level connection with acceptor
- Pass the connection to the handler service
- Handler service creates a new agent for the connection
- Handler service reads message from the connection
- Message is decoded with configured decoder
- Decoded packet from the message is processed
- First packet must be a handshake request, to which the server returns a handshake response with the serializer, route dictionary and heartbeat timeout
- Client must then reply with a handshake ack, connection is then established
- Data messages are processed by the handler and the target server type is extracted from the message route, the message is deserialized using the specified method
- If the target server type is different from the current server, the server makes a remote call to the right type of server, selecting one server according to the routing function logic. The remote call includes the current representation of the client's session
- The receiving remote server receives the request and handles it as a *Sys* RPC call, creating a new remote agent to handle the request, this agent receives the session's representation
- The before pipeline functions are called and the handler message is deserialized

- The appropriate handler is then called by the remote server, which returns the response that is then serialized and the after pipeline functions are executed
- If the backend server wants to modify the session it needs to modify and push the modifications to the frontend server explicitly
- Once the frontend server receives the response it forwards the message to the session specifying the request message ID
- The agent receives the requests, encodes it and sends to the low-level connection

3.1.1 Acceptors

The first thing the client must do is establish a connection with the Pitaya server. And for that to happen, the server must have specified one or more acceptors.

Acceptors are the entities responsible for listening for connections, establishing them, abstracting and forwarding them to the handler service. Pitaya comes with support for TCP and websocket acceptors. Custom acceptors can be implemented and added to Pitaya applications, they just need to implement the proper interface.

3.1.2 Handler service

After the low level connection is established it is passed to the handler service to handle. The handler service is responsible for handling the lifecycle of the clients' connections. It reads from the low-level connection, decodes the received packets and handles them properly, calling the local server's handler if the target server type is the same as the local one or forwarding the message to the remote service otherwise.

Pitaya has a configuration to define the number of concurrent messages being processed at the same time, both local and remote messages count for the concurrency, so if the server expects to deal with slow routes this configuration might need to be tweaked a bit. The configuration is `pitaya.concurrency.handler.dispatch`.

3.1.3 Agent

The agent entity is responsible for storing information about the client's connection, it stores the session, encoder, serializer, state, connection, among others. It is used to communicate with the client to send messages and also ensure the connection is kept alive.

3.1.4 Route compression

The application can define a dictionary of compressed routes before starting, these routes are sent to the clients on the handshake. Compressing the routes might be useful for the routes that are used a lot to reduce the communication overhead.

3.1.5 Handshake

The first operation that happens when a client connects is the handshake. The handshake is initiated by the client, who sends informations about the client, such as platform, version of the client library, and others, and can also send user data in this step. This data is stored in the client's session and can be accessed later. The server replies with heartbeat interval, name of the serializer and the dictionary of compressed routes.

3.1.6 Remote service

The remote service is responsible both for making RPCs and for receiving and handling them. In the case of a forwarded client request the RPC is of type *Sys*.

In the calling side the service is responsible for identifying the proper server to be called, both by server type and by routing logic.

In the receiving side the service identifies it is a *Sys* RPC and creates a remote agent to handle the request. This remote agent is short-lived, living only while the request is alive, changes to the backend session do not automatically reflect in the associated frontend session, they need to be explicitly committed by pushing them. The message is then forwarded to the appropriate handler to be processed.

3.1.7 Pipeline

The pipeline in Pitaya is a set of functions that can be defined to be run before or after every handler request. The functions receive the context and the raw message and should return the request object and error, they are allowed to modify the context and return a modified request. If the before function returns an error the request fails and the process is aborted.

3.1.8 Serializer

The handler must first deserialize the message before processing it. So the function responsible for calling the handler method first deserializes the message, calls the method and then serializes the response returned by the method and returns it back to the remote service.

3.1.9 Handler

Each Pitaya server can register multiple handler structures, as long as they have different names. Each structure can have multiple methods and Pitaya will choose the right structure and methods based on the called route.

CHAPTER 4

Configuration

Pitaya uses Viper to control its configuration. Below we describe the configuration variables split by topic. We judge the default values are good for most cases, but might need to be changed for some use cases.

4.1 Service Discovery

These configuration values configure service discovery for the default etcd service discovery module. They only need to be set if the application runs in cluster mode.

Configuration	Default value	Type	Description
pitaya.cluster.sd.etcd.dialtimeout	5s	time.Time	Dial timeout value passed to the service discovery etcd client
pitaya.cluster.sd.etcd.endpoints	localhost:2379	string	List of comma separated etcd endpoints
pitaya.cluster.sd.etcd.user		string	Username to connect to etcd
pitaya.cluster.sd.etcd.pass		string	Password to connect to etcd
pitaya.cluster.sd.etcd.heartbeat.ttl	60s	time.Time	Hearbeat interval for the etcd lease
pitaya.cluster.sd.etcd.grantlease.timeout	60s	time.Duration	Timeout for etcd lease
pitaya.cluster.sd.etcd.grantlease.maxretries	15	int	Maximum number of attempts to etcd grant lease
pitaya.cluster.sd.etcd.grantlease.retryinterval	5s	time.Duration	Interval between each grant lease attempt
pitaya.cluster.sd.etcd.revoke.timeout	5s	time.Duration	Timeout for etcd's revoke function
pitaya.cluster.sd.etcd.heartbeat.log	false	bool	Whether etcd heartbeats should be logged in debug mode
pitaya.cluster.sd.etcd.prefix	pitaya/	string	Prefix used to avoid collisions with different pitaya applications, servers must have the same prefix to be able to see each other
pitaya.cluster.sd.etcd.syncservers.interval	120s	time.Duration	Interval between server syncs performed by the service discovery module
pitaya.cluster.sd.etcd.shutdown.delay	10s	time.Duration	Time to wait to shutdown after deregistering from service discovery
pitaya.cluster.sd.etcd.servertypeblacklist	[]	[]string	A list of server types that should be ignored by the service discovery
pitaya.cluster.sd.etcd.syncservers.parallelism	10	int	The number of goroutines that should be used while getting server information on etcd initialization

4.2 RPC Service

The configurations only need to be set if the RPC Service is enabled with the given type.

Configuration	Default value	Type	Description
pitaya.buffer.cluster.rpc.server.nats.messages	75	int	Size of the buffer that for the nats RPC server accepts before starting to drop incoming messages
pitaya.buffer.cluster.rpc.server.nats.push	100	int	Size of the buffer that the nats RPC server creates for push messages
pitaya.cluster.rpc.client.grpc.dialtimeout	5s	time.Duration	Timeout for the gRPC client to establish the connection
pitaya.cluster.rpc.client.grpc.lazyconnection	true	bool	Whether the gRPC client should use a lazy connection, that is, connect only when a request is made to that server
pitaya.cluster.rpc.client.grpc.requesttimeout	5s	time.Duration	Request timeout for RPC calls with the gRPC client
pitaya.cluster.rpc.client.nats.connect	nats://localhost:4222	string	Nats address for the client
pitaya.cluster.rpc.client.nats.connectiontimeout	5s	time.Duration	Timeout for the nats client to establish the connection
pitaya.cluster.rpc.client.nats.requesttimeout	5s	time.Duration	Request timeout for RPC calls with the nats client
pitaya.cluster.rpc.client.nats.maxreconnectionretries	15	int	Maximum number of retries to reconnect to nats for the client
pitaya.cluster.rpc.server.nats.connect	nats://localhost:4222	string	Nats address for the server
pitaya.cluster.rpc.server.nats.connectiontimeout	5s	time.Duration	Timeout for the nats server to establish the connection
pitaya.cluster.rpc.server.nats.maxreconnectionretries	15	int	Maximum number of retries to reconnect to nats for the server
pitaya.cluster.rpc.server.grpc.port	3434	int	The port that the gRPC server listens to
pitaya.concurrency.remote.service	30	int	Number of goroutines processing messages at the remote service for the nats RPC service
pitaya.worker.redis.url	localhost:6379	string	Redis url pitaya workers use to register jobs
pitaya.worker.redis.pool	10	string	Number of connections to keep with Redis
pitaya.worker.redis.password	""	string	Redis password to connect to pitaya workers redis
pitaya.worker.concurrency	1	int	Number of workers to execute job
pitaya.worker.namespace	""	string	Worker namespace, can be used to differ stacks in a blue-green deployment
pitaya.worker.retry.enabled	true	bool	If true, retry job if errored for max times
pitaya.worker.retry.max	5	int	Max number of job retries
pitaya.worker.retry.exponential	2	int	Retry job after backoff of nRetry**2
pitaya.worker.retry.minDelay	0	int	Min time to wait on backoff to retry job
pitaya.worker.retry.maxDelay	10	int	Max time to wait on backoff to retry job
pitaya.worker.retry.maxRandom	10	int	Random time to wait during backoff

4.3 Connection

Configuration	Default value	Type	Description
pitaya.handler.messages.compression	true	bool	Whether messages between client and server should be compressed
pitaya.heartbeat.interval	30s	time.Time	Keepalive heartbeat interval for the client connection
pitaya.conn.ratelimiting.interval	1s	time.Duration	Window of time to count requests
pitaya.conn.ratelimiting.limit	20	int	Max number of requests allowed in a interval
pitaya.conn.ratelimiting.forcedisable	false	bool	If true, ignores rate limiting even when added with WithWrappers

4.4 Metrics Reporting

Configuration	Default value	Type	Description
pitaya.metrics.statsd.enabled	false	bool	Whether statsd reporting should be enabled
pitaya.metrics.statsd.host	localhost:9125	string	Address of the statsd server to send the metrics to
pitaya.metrics.statsd.prefix	pitaya.	string	Prefix of the metrics reported to statsd
pitaya.metrics.statsd.rate	1	int	Statsd metrics rate
pitaya.metrics.prometheus.enabled	false	bool	Whether prometheus reporting should be enabled
pitaya.metrics.prometheus.port	9090	int	Port to expose prometheus metrics
pitaya.metrics.constTags	map[string]string	map[]string]string	Constant tags to be added to reported metrics
pitaya.metrics.additionalTags	map[string]string	map[]string]string	Additional tags to reported metrics, the map is from tag to default value
pitaya.metrics.periodicMetrics.period	15s	string	Period that system metrics will be reported
pitaya.metrics.custom.counters	[]	[]interface{}	Custom metrics counter
pitaya.metrics.custom.counters[].Subsystem	""	string	Custom counter subsystem name
pitaya.metrics.custom.counters[].Name	""	string	Custom counter name, must not be empty
pitaya.metrics.custom.counters[].Help	""	string	Custom counter help which explain what is the metric, must not be empty
pitaya.metrics.custom.counters[].Labels	[]	[]string	Custom counter labels the metric will carry
pitaya.metrics.custom.gauges	[]	[]interface{}	Custom metrics gauge
pitaya.metrics.custom.gauges[].Subsystem	""	string	Custom gauge subsystem name
pitaya.metrics.custom.gauges[].Name	""	string	Custom gauge name, must not be empty
pitaya.metrics.custom.gauges[].Help	""	string	Custom gauge help which explain what is the metric, must not be empty
pitaya.metrics.custom.gauges[].Labels	[]	[]string	Custom gauge labels the metric will carry
pitaya.metrics.custom.summaries	[]	[]interface{}	Custom metrics summary
pitaya.metrics.custom.summaries[].Subsystem	""	string	Custom summary subsystem name
pitaya.metrics.custom.summaries[].Name	""	string	Custom summary name, must not be empty
pitaya.metrics.custom.summaries[].Help	""	string	Custom summary help which explain what is the metric, must not be empty
pitaya.metrics.custom.summaries[].Labels	[]	[]string	Custom summary labels the metric will carry
pitaya.metrics.custom.summaries[].Quantiles	map[float64]float64{0.5: 0.05, 0.9: 0.01, 0.99: 0.001}	map[float64]float64	Custom summary objectives with quantiles

4.5 Concurrency

Configuration	Default value	Type	Description
pitaya.buffer.agent.messages	100	int	Buffer size for received client messages for each agent
pitaya.buffer.handler.localprocess	20	int	Buffer size for messages received by the handler and processed locally
pitaya.buffer.handler.remoteprocess	20	int	Buffer size for messages received by the handler and forwarded to remote servers
pitaya.concurrency.handler.dispatch	25	int	Number of goroutines processing messages at the handler service

4.6 Modules

These configurations are only used if the modules are created. It is recommended to use Binding Storage module with gRPC RPC service to be able to use all RPC service features.

Configuration	Default value	Type	Description
pitaya.session.unique	true	bool	Whether Pitaya should enforce unique sessions for the clients, enabling the unique sessions module
pitaya.modules.bindingstorage.etcd.endpoints	localhost:2379	string	Comma separated list of etcd endpoints to be used by the binding storage module, should be the same as the service discovery etcd
pitaya.modules.bindingstorage.etcd.prefix	pitaya/	string	Prefix used for etcd, should be the same as the service discovery
pitaya.modules.bindingstorage.etcd.dialtimeout	5s	time.Time	Timeout to establish the etcd connection
pitaya.modules.bindingstorage.etcd.leasettl	1h	time.Duration	Duration of the etcd lease before automatic renewal

4.7 Default Pipelines

These configurations control if the default pipelines should be enabled or not

Configuration	Default value	Type	Description
pitaya.defaultpipelines.structvalidation.enabled	false	bool	Whether Pitaya should enable the default struct validator for handler arguments

4.8 Groups

These configurations are used for group services implementations.

Configuration	Default value	Type	Description
pitaya.groups.etcd.endpoints	localhost:2379	string	Comma separated list of etcd endpoints to be used by the groups etcd service
pitaya.groups.etcd.prefix	pitaya/	string	Prefix used for every group key in etcd
pitaya.groups.etcd.dialtimeout	5s	time.Time	Timeout to establish the etcd group connection
pitaya.groups.etcd.transactiontimeout	1s	time.Duration	Timeout to finish group request to Etcd
pitaya.groups.memory.tickduration	30s	time.Duration	Duration time of tick that will check if should delete group or not

CHAPTER 5

Pitaya API

5.1 Handlers

Handlers are one of the core features of Pitaya, they are the entities responsible for receiving the requests from the clients and handling them, returning the response if the method is a request handler, or nothing, if the method is a notify handler.

5.1.1 Signature

Handlers must be public methods of the struct and have a signature following:

Arguments

- `context.Context`: the context of the request, which contains the client's session.
- `pointer or []byte`: the payload of the request (*optional*).

Notify handlers return nothing, while request handlers must return:

- `pointer or []byte`: the response payload
- `error`: an error variable

5.1.2 Registering handlers

Handlers must be explicitly registered by the application by calling `pitaya.Register` with a instance of the handler component. The handler's name can be defined by calling `pitaya/component`.WithName(`"handlerName"`) and the methods can be renamed by using `pitaya/component`.WithNameFunc(`func(string) string`).

The clients can call the handler by calling `serverType.handlerName.methodName`.

5.1.3 Routing messages

Messages are forwarded by pitaya to the appropriate server type, and custom routers can be added to the application by calling `pitaya.AddRoute`, it expects two arguments:

- `serverType`: the server type of the target requests to be routed
- `routingFunction`: the routing function with the signature `func(*session.Session, *route.Route, []byte, map[string]*cluster.Server) (*cluster.Server, error)`, it receives the user's session, the route being requested, the message and the map of valid servers of the given type, the key being the servers' ids

The server will then use the routing function when routing requests to the given server type.

5.1.4 Lifecycle Methods

Handlers can optionally implement the following lifecycle methods:

- `Init()` - Called by Pitaya when initializing the application
- `AfterInit()` - Called by Pitaya after initializing the application
- `BeforeShutdown()` - Called by Pitaya when shutting down components, but before calling shutdown
- `Shutdown()` - Called by Pitaya after the start of shutdown

5.1.5 Handler example

Below is a very barebones example of a handler definition, for a complete working example, check the cluster demo.

```
import (
  "github.com/topfreegames/pitaya"
  "github.com/topfreegames/pitaya/component"
)

type Handler struct {
  component.Base
}

type UserRequestMessage struct {
  Name    string `json:"name"`
  Content string `json:"content"`
}

type UserResponseMessage {
}

type UserPushMessage{
  Command string `json:"cmd"`
}

// Init runs on service initialization (not required to be defined)
func (h *Handler) Init() {}

// AfterInit runs after initialization (not required to be defined)
func (h *Handler) AfterInit() {}
```

(continues on next page)

(continued from previous page)

```
// TestRequest can be called by the client by calling <servertype>.testhandler.
↪testrequest
func (h *Handler) TestRequest(ctx context.Context, msg *UserRequestMessage)␣
↪(*UserResponseMessage, error) {
  return &UserResponseMessage{}, nil
}

func (h *Handler) TestPush(ctx context.Context, msg *UserPushMessage) {
}

func main() {
  pitaya.Register(
    &Handler{}, // struct to register as handler
    component.WithName("testhandler"), // name of the handler, used by the clients
    component.WithNameFunc(strings.ToLower), // naming conversion scheme to be used␣
↪by the clients
  )

  ...
}
```

5.2 Remotes

Remotes are one of the core features of Pitaya, they are the entities responsible for receiving the RPCs from other Pitaya servers.

5.2.1 Signature

Remotes must be public methods of the struct and have a signature following:

Arguments

- `context.Context`: the context of the request.
- `proto.Message`: the payload of the request (*optional*).

Remote methods must return:

- `proto.Message`: the response payload in protobuf format
- `error`: an error variable

5.2.2 Registering remotes

Remotes must be explicitly registered by the application by calling `pitaya.RegisterRemote` with a instance of the remote component. The remote's name can be defined by calling `pitaya/component`.WithName(`"remoteName"`) and the methods can be renamed by using `pitaya/component`.WithNameFunc(`func(string) string`).

The servers can call the remote by calling `serverType.remoteName.methodName`.

5.2.3 RPC calls

There are two options when sending RPCs between servers:

- **Specify only server type**: In this case Pitaya will select one of the available servers at random
- **Specify server type and ID**: In this scenario Pitaya will send the RPC to the specified server

5.2.4 Lifecycle Methods

Remotes can optionally implement the following lifecycle methods:

- `Init()` - Called by Pitaya when initializing the application
- `AfterInit()` - Called by Pitaya after initializing the application
- `BeforeShutdown()` - Called by Pitaya when shutting down components, but before calling shutdown
- `Shutdown()` - Called by Pitaya after the start of shutdown

5.2.5 Remote example

For a complete working example, check the cluster demo.

CHAPTER 6

Examples

Example projects can be found here

CHAPTER 7

Indices and tables

- genindex
- modindex
- search