Pavel Císař

Saturnin/Saturnin SDK - lead developer
IBPhoenix & Firebird Project

Firebird Conference 2019, Berlin

Firebird Conference 2019

Berlin, 17-19 October

YOUR PREMIER SOURCE OF FIREBIRD SUPPORT

IBSurgeon

MOSCOW
EXCHANGE

Fast Reports
Reporting must be fast!

Part I.
Saturnin SDK 101

What is Saturnin SDK

Reference implementation of ***Firebird Butler Development Platform*** in Python 3

- ✔ demonstration of functional implementation of specifications for other implementers
- ✔ package for creating and using Firebird Butler services in Python
- ✔ engineering toolset to build ***efficient***, ***robust*** and ***scalable*** solutions
- ✔ the base over which the ***Saturnin*** project is being built

What is a Butler Service

- A Butler Service is basically a piece of software that uses ***ZeroMQ socket*** and ***Firebird Butler Service Protocol*** (***FBSP***) for communication over this ZeroMQ channel
- A service could use multiple ZeroMQ sockets for various purposes, but only one primary socket is required to support the Butler Service protocol
- They ***could do anything***, but a well designed service does only one task, or a small set of closely related tasks within single category
- We also introduced a category of ***microservices*** that do not communicate via *FBSP* but use ***some other standard protocol*** to communicate with other components *(eg Firebird Butler Data Pipe Protocol -* ***FBDP****)*

What is ZeroMQ

- ZeroMQ (also known as ØMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework
- It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast
- You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply
- It's fast enough to be the fabric for clustered products
- Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks
- It has a score of language APIs and runs on most operating systems
- ZeroMQ is from iMatix and is LGPLv3 open source

SDK Factsheet

- Requirements
  - Python 3.6+, will move to Python 3.7 next year
  - PyZMQ >=18.0.0
  - Protobuf >=3.9.0
- Uses
  - type annotations
  - docstrings
  - indent 4 spaces
  - package **saturnin.sdk** distributed as **saturnin-sdk** on PyPI

Part II.
SDK Structure

SDK content & structure, part I.

- namespace packages ***firebird*** and ***saturnin***
  - package ***firebird.butler*** contains ***protobuf*** classses used by various Firebird Butler protocols (eg. *firebird.butler.fbsp_pb2*)
  - package ***saturnin.sdk*** is the SDK
  - services in ***saturnin.services*** package and microservices in ***saturnin.micro*** package
- distribution package ***saturnin-sdk*** should contain only essentials
  - examples in separate distribution package ***saturnin-sdk-examples***
  - service execution and test environment is for now part of SDK, but will be separated into its own distribution package

SDK content & structure, part II.

- ***saturnin.sdk*** – only <u>platform</u> OID, UID and VERSION, and <u>vendor</u> OID and UID
- ***saturnin.sdk.types*** – Type definitions (exceptions, enums, flags, dataclasses etc.)
- ***saturnin.sdk.collections*** – Object ***Registry*** and ***ObjectList***
- ***saturnin.sdk.config*** – Classes for configuration definitions
- ***saturnin.sdk.base*** – ZeroMQ messaging
- ***saturnin.sdk.service*** – Firebird Butler Services
- ***saturnin.sdk.client*** – Firebird Butler Service Clients
- ***saturnin.sdk.classic*** – Classic Service (sync I/O in separate thread or process)
- ***saturnin.sdk.protocol.fbsp*** – implementation of Firebird Butler Service Protocol
- ***saturnin.sdk.protocol.fbdp*** – implementation of Firebird Butler Data Pipe Protocol

- ***saturnin.sdk.tools.**** – SDK tools (*svc_run* & *svc_test*)
- ***saturnin.sdk.test.**** – SDK service test environment (currently only for *fbsp*)
- ***saturnin.services.**** – Saturnin services
- ***saturnin.micro.**** – Saturnin microservices

ZeroMQ and Saturnin messaging

The following slides show only a preview of key elements of the ZeroMQ

For complete information visit
https://zeromq.org/
and read excellent
http://zguide.zeromq.org
PyZMQ at https://pyzmq.rtfd.io

ZeroMQ Essentials, part I.
The Context

- ZeroMQ applications always start by creating a ***context***, and then using that for creating ***sockets***
- it's the container for all sockets in a single process
- if at runtime a process has two contexts, these are like separate ZeroMQ instances
- context acts as the transport for ***inproc*** sockets
  which are the fastest way to connect threads in one process
- PyZMQ provides ***Context*** class
  - ***Context()*** returns new instance
  - ***Context.instance()*** returns global (for process) instance

ZeroMQ Essentials, part II.
Sockets

- the interface is based on the Berkeley sockets API
- but socket is **NOT** single connection to single peer
- it’s a doorway to fast little background comm. engine that manages a whole set of connections, that are private and invisible
- all I/O is done in background threads (in *Context*)
- ZMQ is not a neutral carrier: it imposes a framing on the transport protocols it uses
- underneath socket API is a messaging pattern, and the pattern used determines the type of socket
- patterns are (mostly) implemented as pairs of matching types
- they have 1-to-N routing behavior built-in, according to the type

ZeroMQ Essentials, part III.
Messages

- sockets carry messages
  like UDP, rather than a stream of bytes as TCP
- they are transmitted in a background thread
  they arrive in local input queues and are sent from local output queues
- theys consists from parts called ***Frames***, which are length-specified block of binary data (>=0b)
  - normal message has one frame
  - multipart messages let you send or receive a list of frames as a single on-the-wire message
- you may send zero-length messages
  e.g., for sending a signal from one thread to another
- it is guaranteed that you will receive all parts, or none of them
- whole message (all frames) must fit in memory

ZeroMQ Essentials, part IV.
Socket types

Socket types (in brief):

- REQ(est) + REP(ly)
- PUSH + PULL
- PUB + SUB and XPUB + XSUB
- ***DEALER***
- ***ROUTER***
- PAIR
- STREAM

ZeroMQ Essentials, part V.
DEALER socket

- ***Bidirectional***
- ***Unrestricted*** send/receive pattern
- Outgoing routing strategy: ***Round-robin***
- Incoming routing strategy: ***Fair-queued***
- Action in mute state: ***Block***
- Compatible peer sockets:
  - DEALER
  - ROUTER
  - REP

ZeroMQ Essentials, part VI.
ROUTER socket

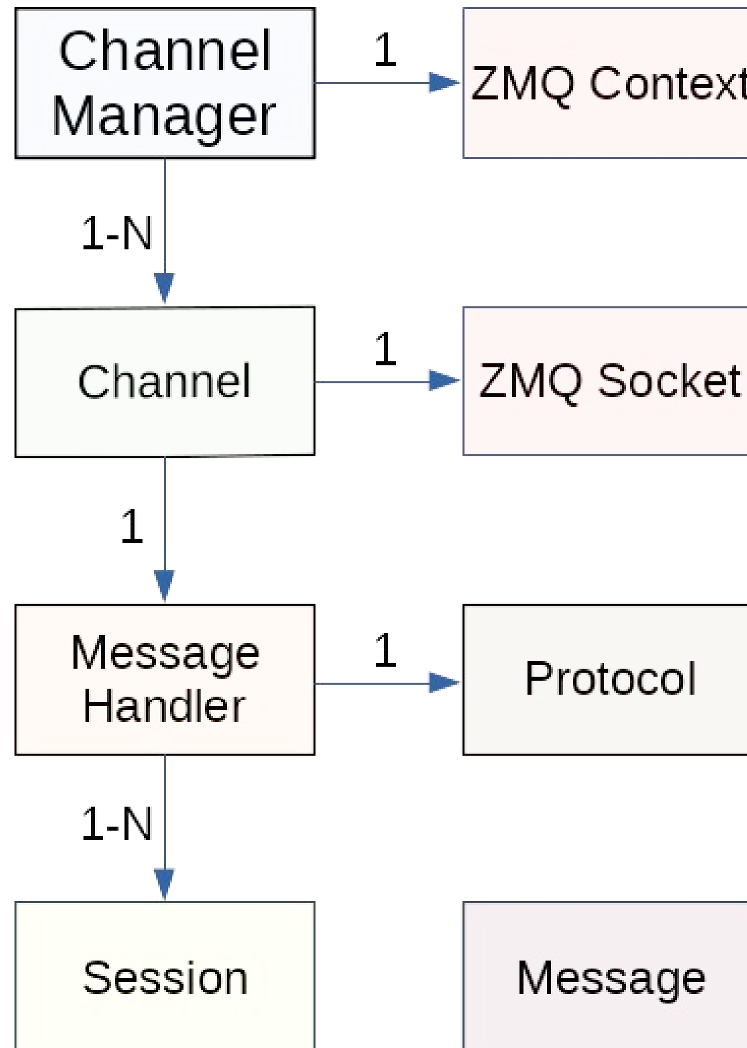
- ***Bidirectional***
- ***Unrestricted*** send/receive pattern
- Outgoing routing strategy: ***1 frame is routing ID***
- Incoming routing strategy: ***Fair-queued***
- Action in mute state: ***Drop***
- Compatible peer sockets:
  - DEALER
  - ROUTER
  - REQ

Messaging in SDK, part I.
Factsheet

- module ***saturnin.sdk.base***
- a base layer above PyZMQ to simplify the creation of ZeroMQ applications in general
- base abstraction for messages, ZMQ sockets as communication channels, communication protocols, message handlers and sessions
- uses module ***saturnin.sdk.types***

Messaging in SDK, part II.

Architecture



Messaging in SDK, part III.

Specific properties

- Multipart messages
- Non-blocking ***send()*** and ***receive()*** with *100ms* default timeout
- Control of output message queue through deferred ***send()*** and session suspension

Messaging in SDK, part IV.
Important types

- Enums:
  - ***Origin*** – Origin of received message in protocol context
  - ***SocketMode*** – ZMQ socket mode (bind, connect)
  - ***AddressDomain*** – ZMQ address domain (local, node, network)
  - ***TransportProtocol*** – ZMQ transport protocol (inproc, ipc, tcp etc.)
  - ***SocketType*** – ZMQ socket type (pub, sub, push, pull, dealer, router etc.)
  - ***SocketUse*** – Socket use (producer, consumer, exchange)
- Flags:
  - ***Direction*** – ZMQ socket direction of transmission (in, out, both)
- Other:
  - ***ZMQAddress*** – ZMQ endpoint address. Descendant from builtin ***str*** type with additional R/O properties ***protocol***, ***address*** and ***domain***

Messaging in SDK, part V.
Message

- base class that simply holds ZMQ multipart message in its ***data*** attribute
- child classes can override ***from_zmsg()*** and ***as_zmsg()*** methods to pack/unpack some or all parts of ZMQ message into their own attributes
- methods ***clear()*** and ***copy()***, functions ***has_data()*** and ***has_zmq_frames()***
- Abstract methods:
  - class method ***validate_zmsg()*** – verifies that sequence of ZMQ data frames is a valid message
- Attributes:
  - ***data*** – sequence of data frames

Messaging in SDK, part VI.

Protocol

- main purpose of protocol classes is to validate ZMQ messages and create protocol messages
- base class defines common interface for parsing and validation: ***validate()***, ***parse()***, ***is_valid()*** and ***has_greeting()***
- descendant classes typically add methods for creation of specific protocol messages
- class attributes: ***OID***, ***UID*** (must be set in child class) and ***REVISION*** (default 1)
- Uses a ***factory pattern*** to create *Messages*

Messaging in SDK, part VII.
Channel

- base class for ZeroMQ communication channel (socket)
- Attributes:
  - ***socket_type*** – ZMQ socket type
  - ***direction*** – Direction of transmission
  - ***socket*** – ZMQ socket for transmission of messages
  - ***flags*** – ZMQ flags used for send() and receive()
  - ***sock_opts*** – Dictionary with socket options that should be set after socket creation
  - ***routed*** – True if channel uses internal routing
  - ***endpoints*** – List of binded/connected endpoints (addresses)
  - ***snd_timeout*** – Timeout for send operations
  - ***rcv_timeout –*** Timeout for receive operations
  - ***handler*** – Message handler used to process messages received from peer(s)
  - ***uid*** – Unique channel ID used by channel manager
  - ***mngr_poll*** – True if channel should register its socket into manager Poller
- R/O attributes:
  - ***mode*** – BIND/CONNECT mode for socket
  - ***identity*** – Identity value for ZMQ socket
  - ***manager*** – The channel manager to which this channel belongs
- Methods: ***bind()***, ***unbind()***, ***connect()***, ***disconnect()***, ***close()***, ***send()***, ***receive()***, ***set_handler(), is_active()*** and ***configure()***

Messaging in SDK, part VIII.
Channel Manager

- A superstructure over the ZMQ context ensuring control over communication channels, and a pillar of the I/O loop
- Attributes:
  - ***ctx*** – ZMQ Context instance
  - ***channels*** – Channels associated with manager
  - ***deferred*** – List with deferred work, contains tuples with *(Callable,List)*
- Methods:
  - ***add()*** & ***remove()*** channel
  - ***register()***, ***unregister()*** and ***is_registered()***
  - ***defer()***, ***is_deferred()*** and ***process_deferred()***
  - ***wait()*** – Wait for I/O events on registered channnels
  - ***shutdown()*** – Terminate all managed channels

Messaging in SDK, part IX.
Message Handler

- the semantic communication layer
  everything essential (in application) is done through the descendants of this class
- Attributes:
  - ***chn*** – Handled I/O channel
  - ***role*** – Peer role
  - ***protocol*** – Protocol used
  - ***sessions*** – Dictionary of active sessions, key=routing_id
  - ***resume_timeout*** – Time limit in seconds for how long session could be suspended before it's cancelled
  - ***on_cancel_session*** – Callback executed before session is cancelled
- Abstract methods:
  - ***dispatch*** - Process message received from peer
- Methods:
  - ***send()*** & ***receive()***
  - ***create***|***get***|***discard***|***suspend***|***resume***|***cancel_session***
  - ***handle_*()*** for handling of invalid messages and dispatch errors

Messaging in SDK, part X.
Session

- contextual information for communication with a particular peer
- base session holds only essentials, child classes may add their own information and functionality
- Attributes:
  - ***routing_id*** – Channel routing ID
  - ***endpoint*** – Connected endpoint address, if any
  - ***discarded*** – True if sessions was discarded
  - ***messages*** – List of deferred messages
  - ***pending_since*** – Value is either None or *monotonic()* time of first unsuccessful send operation (i.e. notes time of suspension and start of *resume_timeout* period)
- Methods:
  - ***send_later()***, ***get_next_message()*** and ***message_sent()***
  - ***is_suspended()***

Part IV.

Firebird Butler Service Protocol (FBSP)

The following slides show only a preview of key elements of the *FBSP* specification

The complete specification is at
https://firebird-butler.rtfd.io/en/latest/rfc/4/FBSP.html

FBSP specification, part I.
Overall Behavior

Exchange of messages over the ***Transport Channel*** is implemented as a ***Connection*** between the ***Service*** and the ***Client*** where the connection has the following stages:

1. The *Client* **MUST** initiate the connection by sending the ***HELLO*** message.
2. The *Service* **MUST** reply to the ***HELLO*** message by sending a ***WELCOME*** message to confirm the connection. If the *Service* cannot accept the connection, it **MUST** send the ***ERROR*** message instead.
3. After confirming a successful *Connection*, the *Client* can start sending messages to which the *Service* responds by sending one or more messages of its own.
4. The *Client* or *Service* can terminate the *Connection* at any time by sending a ***CLOSE*** message, or by closing the *Transport Channel*. However, the peer initiating the connection termination **SHOULD** send the ***CLOSE*** message before it closes the *Transport Channel* to the other peer.

FBSP specification, part II.

Client and Service Identity

Both the *Client* and the *Service* must be uniquely identified

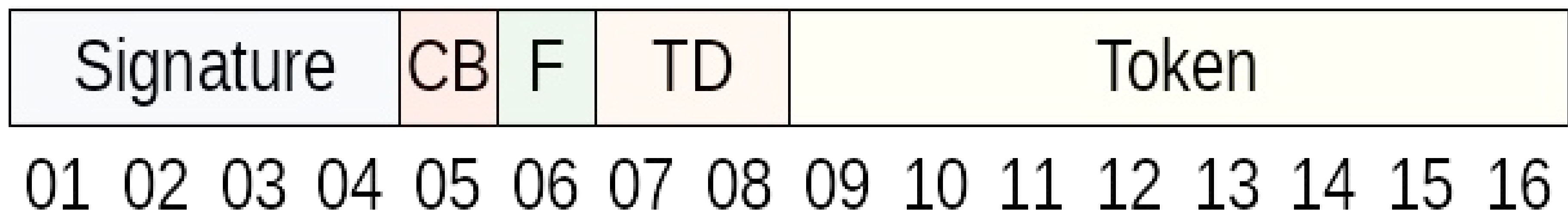
- The content of the *Identity* **MAY** be arbitrary
- There **MUST** be a canonical string *Identity* representation
- Both the *Client Identity* and the *Service Identity* **MUST** be unique in the same namespace
- Both the *Client* and the *Service* **MUST** use the *Identity* for all identification purposes
- If *Service* acts as a *Client* to another *Service*, then **MUST** use its own *Service Identity* as the *Client Identity* to the another *Service*
- It is **RECOMMENDED** that both the *Client* and the *Service* use the *Identity* for routing purposes
- It is **RECOMMENDED** to use UUID as *Identity*

FBSP specification, part III.
Message structure

- Mesage consists from a ***control frame***, and zero or more ***data frames***

Control frame:

- Signature – “FBSP”
- CB (control byte) – encodes a message type, and protocol version. Both are decimal numbers. ***msg-type*** on upper (leftmost) 5 bits, ***version*** on lower (rightmost) 3 bits
- Flags – various control flags as individual bits
- TD (type data) – Message-type specific data



FBSP specification, part IV.
Message types

Message type	*Who can send*	*Description*
HELLO	Client	Initial message from Client
WELCOME	Service	Initial response from Service
NOOP	Client, Service	No operation, used for keep-alive & ping
REQUEST	Client	Client request
REPLY	Service	Service response to client request
DATA	Client, Service	Separate data sent by either client or service
CANCEL	Client	Cancel request
STATE	Service	Operation state information
CLOSE	Client, Service	Sent by peer that is going to close the conn.
ERROR	Service	Error reported by service

FBSP specification, part V.

Message Token

Message Token is a tool to pair *Client* requests with multiple messages sent by *Service* in response

Processing of the token is governed by the following rules:

1. The content of the *Token* **MAY** be arbitrary
2. The content of the *Token* **SHALL** be specified by *Client* only
3. The *Token* **MUST** be returned without change in any message sent by the *Service*, which is a logical response to the original message sent by the *Client* containing that token
4. Messages sent by a *Service* that can not be uniquely identified as a logical response to a previous message sent by a *Client* (such as unexpected general ***ERROR***, ***CLOSE***, or ***NOOP*** sent to check the client's availability) **MUST** contain the *Token* passed by the *Client* in the ***HELLO*** message

FBSP specification, part VI.
Message Flags

Flags are encoded as individual bits in flags field of the ***control-frame***

ACK-REQUEST (bit: 0, mask: 1)

- intended for verification and synchronization purposes
- any received ***control-frame*** of message-type ***NOOP***, ***REQUEST***, ***REPLY***, ***DATA***, ***STATE*** or ***CANCEL*** that have ***ACK-REQUEST*** flag set **SHALL** be sent back to the sender as confirmation of accepted message, unless the receiver is a *Service* and an error condition occurs. In such a case the ***ERROR*** message **SHALL** be sent by *Service* instead confirmation message

ACK-REPLY (bit: 1, mask: 2)

- indicates that message is a confirmation of the message previously sent by receiver

MORE (bit: 2, mask: 4)

- The MORE flag **SHALL** be set for all messages that are a part of logical message stream, and are not the terminal message of this stream

FBSP specification, part VII.

Service API

The ***Service API*** consists from ***Interfaces*** (API contracts) that consists from individual operations (functions)

1. An ***Interface*** **SHALL** have a globally unique identification (GUID). It's **RECOMMENDED** to use uuid version 5 - SHA1, namespace OID
2. An ***Interface*** **MUST** provide at least one ***Operation*** (function), and **MAY** provide up to 255 individual ***Operations***
3. An ***Operation*** **MUST** have numeric identification unique within the ***Interface***, and with value in range 1..255. This identification is called ***Interface operation code***
4. The *Service* **MUST** assign an unique ***Interface identification number*** in range 1..255 to each ***Interface*** it provides, and announce the ***Interface*** identification along with assigned number in the ***data-frame*** of the ***WELCOME*** message
5. The *Service* **MUST** provide at least one ***Interface***, and **MAY** provide up to 255 individual ***Interfaces***
6. The set of ***Interfaces*** that *Service* provides **MUST** be stable, which means that all *Service* instances with the same ***Agent Identification*** **MUST** provide the same set of ***Interfaces*** to all *Clients*

FBSP specification, part VIII.
Request Codes

The ***Request Code*** uniquely identifies the *Service* functionality (an API call)

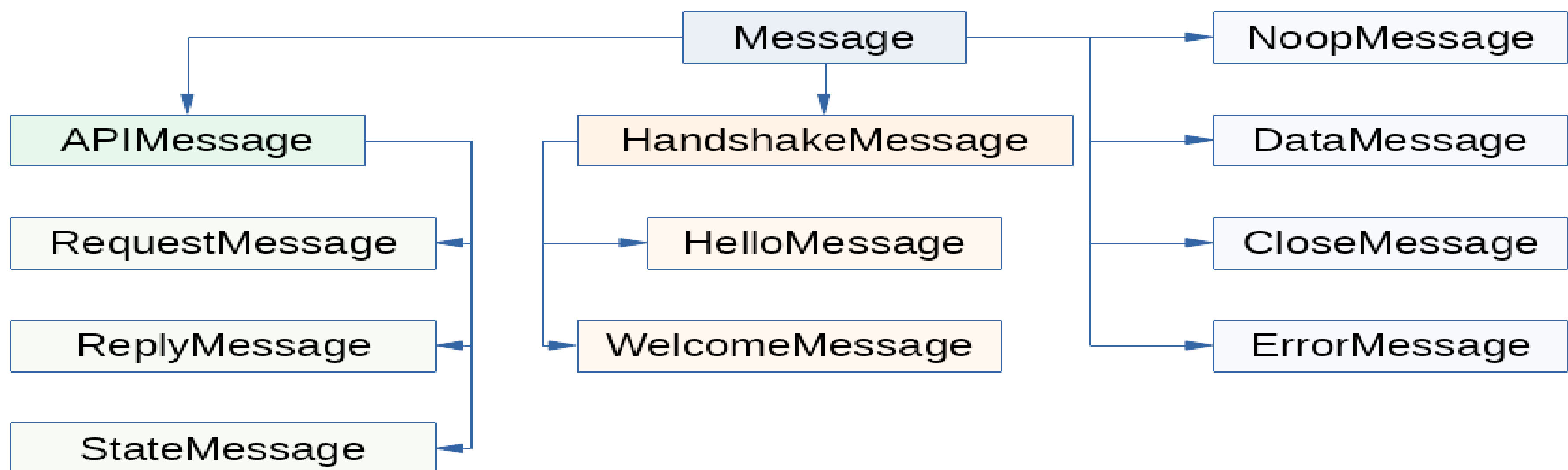
1. The first (more significant) byte of ***type-data*** field **SHALL** contain the ***Interface identification number*** assigned by *Service* to particular ***Interface*** it supports (see *Data frames - WELCOME*)
2. The second (less significant) byte of ***type-data*** field **SHALL** contain the ***Interface operation code***

FBSP implementation, part I.
Factsheet

- module ***saturnin.sdk.protocol.fbsp***
- protobuf classes in ***firebird.butler.fbsp_pb2***
  also available as *saturnin.sdk.protocol.fbsp.fbsp_proto*
- redefines ***Message***, ***Protocol*** and ***Session*** classes
- provides *FBSP* ***Message Handlers*** for *Services* and *Clients*
- provides helper functions for conversion between ***ERROR*** message and ***exception***
- provides global ***Protocol*** instance

FBSP implementation, part II.
Types

- Enums: ***MsgType*** and ***ErrorCode***
- Flags: ***MsgFlag***
- Classes ***Protocol*** and ***Session***
- Handler classes: ***BaseFBSPHandler***, ***ServiceMessagelHandler*** and ***ClientMessageHandler***
- Message classes:



FBSP implementation, part III.
Protocol class

- Class attributes:
  - ***OID, UID, REVISION***
  - ***VALID_ACK***
  - ***ORIGIN_MESSAGES***
  - ***MESSAGE_MAP***
- Class methods: ***instance()***
- Methods:
  - ***create_message_for()*** - new msg instance for particular FBSP message type
  - ***create_ack_reply()*** - new msg that is an *ACK-REPLY* response message
  - ***create_welcome_reply()*** - new *WelcomeMessage* that is a reply to HELLO
  - ***create_error_for()*** - new *ErrorMessage* that relates to specific message
  - ***create_reply_for()*** - new *ReplyMessage* for specific *RequestMessage*
  - ***create_state_for()*** - new *StateMessage* that relates to *RequestMessage*
  - ***create_data_for()*** - new *DataMessage* for reply to specific *RequestMessage*
  - ***create_request_for()*** - new *RequestMessage* for specific API call

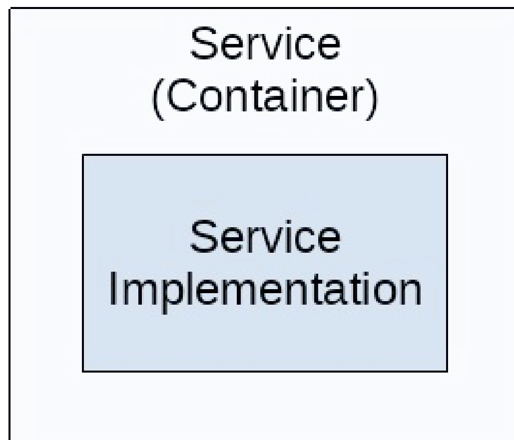
FBSP implementation, part IV.
FBSP Message Handlers

- Implements ***dispatch()*** that uses ***handlers*** dictionary to route received messages to appropriate handlers
  - Child classes may update this table with their own handlers in ***__init__()***
  - Dictionary key could be either a ***<message_type>*** or ***tuple(<message_type>,<type_data>)***
- Exceptions in handlers are captured by ***dispatch()***, and processed via ***handle_exception()*** method
- ***ServiceMessagelHandler*** and ***ClientMessageHandler*** classes implement required and typical message processing patterns for *Services* a *Clients* using various ***handle_*()*** methods
- There is only several abstract methods left:
  - ServiceHandler: ***handle_cancel()***
  - ClientHandler: ***handle_reply()***, ***handle_data()***, ***handle_state()***, ***handle_error()***

Part V.
Butler services in Saturnin

Services, part I.
Architecture, part I.

- service implementation is separated into two class hierarchies defined in ***saturnin.sdk.service*** module
  - ***(Base)Service*** defines structure of the service composed from abstract structural parts
  - ***(Base)ServiceImpl*** provides implementation of structural parts
- Service implementation involves creating a child of a class in the ***BaseServiceImpl*** hierarchy, and a child of the ***FBSPServiceHandler*** class

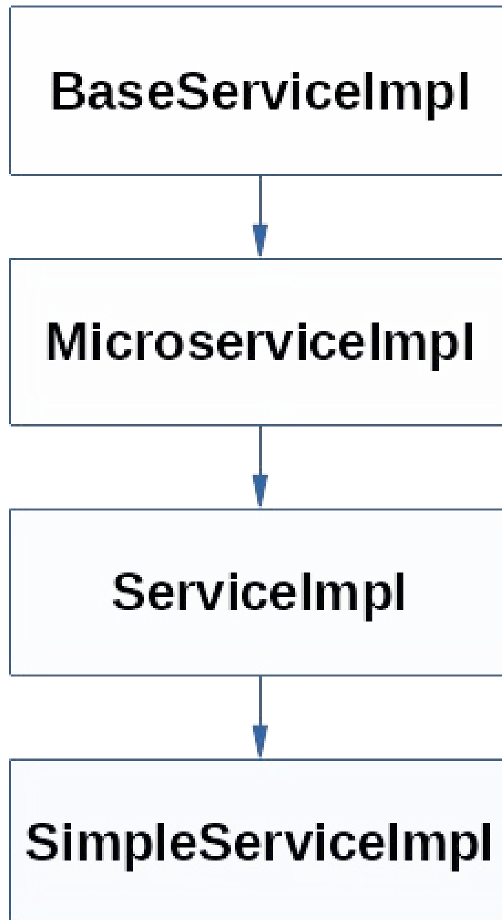


Service source code layout

Directory ***saturnin.service.<service-name>***

- ***api.py*** – descriptors and configuration
- ***service.py*** – service impl + msg handler
- ***client.py*** – svc client (msg handler)
- ***test.py*** – svc tester class

Services, part II.
Architecture, part II.



BaseServiceImpl

- Attributes: ***mngr***, ***stop_event***
- Cfg. options: ***shutdown_linger***
- Methods: ***get()***, ***initialize()***, ***configure()***, ***validate()***, ***finalize()***, ***idle()***

MicroserviceImpl

- Attributes: ***agent***, ***peer***, ***instance_id***

ServiceImpl

- Attributes: ***endpoints***, ***welcome_df***, ***api***

SimpleServiceImpl

- Attributes: ***svc_chn***
- Cfg. options: ***sock_opts***

Services, part III.
Architecture, part III.

- module ***saturnin.sdk.classic*** provides ***Service*** *(container)* that runs its own message loop and do not use any async programming techniques or Python libraries
- Running a service effectively blocks the main thread, so if an application needs to perform other tasks or if you need to run multiple services in parallel in one application, each service must run in a separate ***thread*** or ***subprocess***
- Module provides ***ServiceExecutor*** class to run such classic (micro)services in their own separate thread or subprocess
  - ✔ could be used in application that is built from (micro)services
  - ✔ or in services to distribute work across pool of worker microservices

Services, part IV.

Services, Agents, Peers and Interfaces, part I.

- ***Agent*** = particular ***service***, ***client*** or ***application***
  for example *backup-service*, *my-server-console* etc.
- ***Peer*** = a running instance of particular ***Agent***
- Both the ***Client*** and the ***Service*** exchange their ***Agent*** and ***Peer*** information during the **HELLO/WELCOME** handshake
- ***Service*** also sends information about its API (interfaces)
- The format, content and structure is defined by ***3/FBSD*** and ***4/FBSP*** specifications
  - Peer info:
    uid (bytes), ***pid*** (uint32), ***host*** (string) and ***supplement*** (repeated Any)
  - Agent info:
    uid (bytes), ***name*** (string), ***version*** (string), ***vendor*** (VendorId), ***platform*** (PlatforId), ***classification*** (string) and ***supplement*** (repeated Any)

Services, part V.

Services, Agents, Peers and Interfaces, part II.

The Saturnin SDK builds on this information its own ***Service Description*** structure – dataclasses defined in ***saturnin.sdk.types***

AgentDescriptor	
uid	Agent ID (UUID)
name	Agent name
version	Agent version string
vendor_uid	Vendor ID (UUID)
classification	Agent classification string
platform_uid	Butler platform ID (UUID)
platform_version	Butler platform version string
supplement	Optional list of supplemental information

InterfaceDescriptor	
uid	Interface ID (UUID)
name	Interface name
revision	Interface revision number
number	Interface Identification Number assigned by Service
requests	Enum for interface operation codes

PeerDescriptor	
uid	Peer ID (UUID)
pid	Peer process ID
host	Host name
supplement	Optional list of supplemental information

Services, part VI.

Services, Agents, Peers and Interfaces, part III.

ServiceDescriptor	
agent	Service agent descriptor
api	Service FBSP API description or None (for microservice)
dependencies	List of (DependencyType, UUID) tuples
execution_mode	Preferred execution mode
service_type	Type of service
facilities	Service facilities
description	Text describing the service
implementation	Locator string for service implementation class
container	Locator string for service container class
config	Locator string for service configuration callable
client	Locator string for service client class
tests	Locator string for service test class

Part VI.
Service configuration

Service configuration

- Set of classes in ***saturnin.sdk.config***
- Hierarchy of of configuration ***Option*** classes that supports:
  - typed values
  - validation
  - serialization to/from ***protobuf Struct*** message
  - loading from standard Python ***configparser***
  - printing
  - creation of default/sample configuration files
  - enough metadata for interactive value specification
- Supported option data types: ***str***, ***int***, ***float***, ***bool***, ***strList***, ***ZMQAddress***, ***ZMQAddressList***, ***Enum***, ***UUID***, ***MIME*** type, ***Config*** (collection of options) and ***ConfigList***
- ***Config*** class is a collection of ***Options***
- Config descendants with predefined options for ***Microservices*** and ***Services***

Questions?

Thanks for your attention

Contacts:

- Email: pcisar@ibphoenix.cz
- www.ibphoenix.com

Saturnin SDK:

- git: https://github.com/FirebirdSQL/saturnin-sdk
- Documentation: https://saturnin-sdk.rtfd.io/
- Mailing list: https://groups.google.com/d/forum/saturnin-sdk