xbus.broker Documentation

Release 0.1.6-dev

XCG Consulting

April 14, 2016

1 Getting started 3
1.1 With docker 3
1.2 Using the source code 3

2 General Presentation 5
2.1 High coherence 5
2.2 Low coupling 5
2.3 Xbus 6

3 Architecture overview 7
3.1 Frontend 7
3.2 Backend 7

4 Semantics 9

5 Source code documentation 11
5.1 Front-End 11
5.2 Back-End 11

6 Xbus recipient API 13
6.1 Workflows 13
6.2 get_metadata 14
6.3 ping 14
6.4 has_clearing 15
6.5 has_immediate_reply 15
6.6 start_event 15
6.7 send_item 15
6.8 end_event 16
6.9 end_envelope 16
6.10 stop_envelope 16

7 Supervision 17
7.1 Configuration 17
7.2 Polling 17

8 Immediate reply Xbus feature 19
8.1 Description 19

9 Data clearing Xbus feature 21

9.1 Feature dependencies 21
9.2 Database schema 21
9.3 Monitor-consumer communication 21

10 TODO **23**

11 Indices and tables **25**

Contents:

CHAPTER 1

Getting started

1.1 With docker

Xbus is packaged using docker to help deploy it. The docker images are available on the docker hub.

1.2 Using the source code

Copy the sample config file and edit it to suit your needs:

```
$ cp config.ini.sample config.ini
```

Create your own virtualenv:

```
$ virtualenv env-xbus
$ source env-xbus/bin/activate
(env-xbus)$ setup_xbusbroker -c config.ini
```

This should create your data tables into the database you chose in the configuration file. You should now start the server:

```
$ start_xbusbroker -c config.ini
```

Once this is done, the broker is running.

CHAPTER 2

General Presentation

Xbus is an Enterprise service bus. As such it aims to help IT departments achieve a better application infrastructure layout by providing a way to urbanize the IT systems.

The goals of urbanization are:

- high coherence
- low coupling

2.1 High coherence

This is important because everyone wants applications to behave in a coherent way and not fall appart with bizare errors in case of data failure. Think of your accounting system or your CRM. You want the accounting to achieve data consistency event if an incoming invoices batch fails to load properly.

2.2 Low coupling

But this is not because you want coherence that you are happy with every application in your infrastructure talking directly to the API of every other application. This may seem appealing when you have 3 or 4 applications, but quite rapidly you'll get tens or even hundreds of different interface flows between many applications. All these flows must be maintained:

- at the emitter side, because the emitter needs to implement the receiver API, or at least flat file layout
- at the receiver side, because one day the receiver will want to change the file schema or the API it provides
- at the operator side, (yup!), because when you begin have tens or more of nightly (or on demand) interface flows you will need to monitor them and make sure they get delivered. And you will also want to know when they fail and what to do in those cases (ie: is this because the receiver failed and we should retry or is this an emitter problem...)

When you don't use a Bus or Broker you are in a situation of high coupling. Changing one component of your IT infrastructure may prove a big challenge, just because it received interfaces from 3 different systems and provided nightly data to 2 other.

2.3 Xbus

With the Xbus broker we try to address those issues registering all emitters, all consumers and what kind of events they can emit or receive.

Since the emitter is not technically aware of who will consume its event data it will not be necessary to change its side of the interface when replacing a consumer by another in your infrastructure.

CHAPTER 3

Architecture overview

Xbus is split into two distinct core components:

- the frontend `xbus.broker.core.front.rpc.XbusBrokerFront`: responsible to handle all emitters, talk to them, get their events and acknowledge their data as soon as it is safely stored
- the backend `xbus.broker.core.back.rpc.XbusBrokerBack`: responsible to forward event data to the network of workers and eventually consumers

Those two parts speak together transparently and the end-user does not necessarily sees a difference between the two.

Front and back both have their separate 0mq sockets. Emitters use the front socket while workers and consumers use the backend socket.

When you start an xbus server it will generally start one frontend and one backend attached to it.

3.1 Frontend

`xbus.broker.core.front.rpc.XbusBrokerFront` is the component that handles all emitter connexions and provides the emitter API. It operates on its own socket (by default listening on TCP/1984).

3.2 Backend

`xbus.broker.core.back.rpc.XbusBrokerBack` is the component that handles all worker and consumer connexions. It operates on its own socket (by default listening on TCP/4891).

When a backend instance starts it tries to register itself to a given frontend by connecting to an internal 0mq socket. The frontend will acknowlege this and then use the normal backend socket to send all the events it needs to send.

CHAPTER 4

Semantics

Before you are able to connect an emitter to the *Frontend* or a worker to the *Backend*, you'll need to understand the xbus broker semantics.

The most important terms are:

- *event*
- *event type*
- *event node*
- *emitter*
- *emitter profile*
- *item*
- *worker*
- *consumer*
- *service*
- *role*
- *envelope*
- *immediate reply*

Event In the Xbus semantics the core of what we transfer between actors of the IT infrastructure are considered as events. Events are arbitrary collections of data structures. An event will be received by the *Frontend* and propagated to all the consumers that have registered to receive it.

Event Type Each and every event in xbus needs only one thing: an event type. This is only a name, but this means a lot: Xbus does not try by itself to assert anything about the datastructure it transports and forwards. But at the same time it is necessary for the *consumers* (receivers) to know what kind of data they will receive and how to treat it. The event type is that contract. IE: if I say to the bus that I emit *new_clients* the consumers may rely on the bus to make sure that the same kind of datastructure will be served to them each time.

Envelope Any *event* sent into the bus must be enclosed into an envelope. This is important because the evelope is a transactional unit that permits to rollback operations at the envelope level if your consumers are transaction aware.

This really depends on your application layout but you can easily imagine a simple network that handles *invoices* and a final *consumer* that will write accounting lines into accounting books. If for *any* reason the envelope failed in the middle you would want that NO single line was written in your books. Putting all the *invoice lines* in a single envelope you ensure that everything in the same envelope will be part of the same transaction.

Event Node Internally we use the term *event node* to describe a node in our graph that will handle an event. This is specifically used in the *Backend* part of the broker and refers to eitheir a *worker* or a *consumer*.

Emitter An emitter is an independant program in your IT infrastructure that needs to send information about a change, a new item or whatever. In the internal Xbus database each emitter is assigned an emitter row that contains its login / password pair. An emitter is just that, it does not directly declare what it wants to emit.

This is declared by the Xbus administrator using *emitter profiles*

The emitter however declares what profile it is using.

Emitter Profile A profile is used to link one or more emitters to a list of allowed *event types*

An *emitter* can only emit the type of events that are linked to its profile. Xbus will refuse any other event type.

Item An *event* contain one or more items.

When sending, an event_id and data (JSON or msgpack) are needed.

Worker A worker is an independant program that connects to the xbus *Backend* and declares itself ready to handle events as they are emitted in the network.

It is important to understand that a worker is not intended to be used as a final node of a graph but instead as an intermediate node that will process data, transform or enrich it and then return it back to the broker.

The contract between a worker and the *xbus backend* is that the bus will send all items of an event down to a worker and that the worker must send back a list of items.

Consumer A consumer as a *worker node* is still an independant program that connects to the *xbus backend*, but it is considered as a final node that will not return data for each item received.

On the contrary it will wait for the end of an envelope to give some kind of answer to the *Backend*.

Recipient Nodes that form the actual execution path for an ongoing event.

Service An abstract representation of one or more *event nodes* be it a *worker* or *consumer*. The service is the link between an event node and one or more concrete workers.

Attached to the service we will find a role, which is the concrete distinct instance of a *worker* or *consumer*.

Role The individual *worker* or *consumer*. There is a separation between *service* and role because you can connect many different roles to your bus that will provide the same service.

In effect, once you have described your work graph using a tree of *event nodes*, each one attached to a distinct service, you'll be able to spawn as many real workers (programs that provide a service) that will attach to one service.

The bus will automatically distribute the work between all the roles that provide the same *service*.

Immediate reply *Emitters* of *events* marked as wanting an "immediate reply" will wait for a reply from the consumer once they have called the "end_event" RPC call.

The reply will be sent via the return value of the "end_event" call.

The "immediate reply" flag is an attribute of *event types*.

Restrictions:

- Immediate replies are disallowed when more than one consumer is available to the emitter wishing to send events with that flag.
- The consumer MUST announce support for the "Immediate reply" feature (see the documentation about the Xbus recipient API for details).

CHAPTER 5

Source code documentation

5.1 Front-End

5.1.1 Front-end RPC

5.2 Back-End

5.2.1 Back-end RPC

5.2.2 Envelope

5.2.3 Event

5.2.4 Node

CHAPTER 6

Xbus recipient API

Description of the API Xbus *recipients* must implement in order to register into an Xbus network.

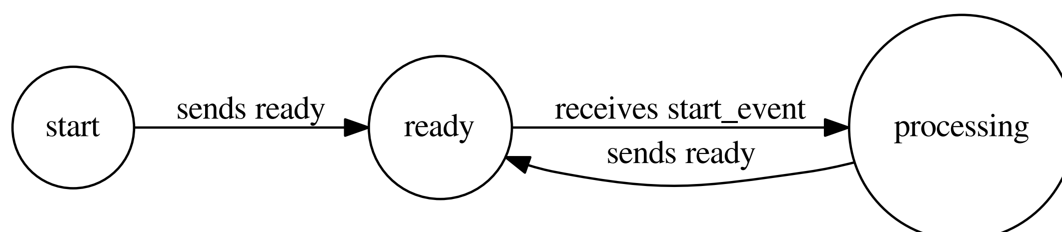
Version of this document: 0.2.

Methods described in this document:

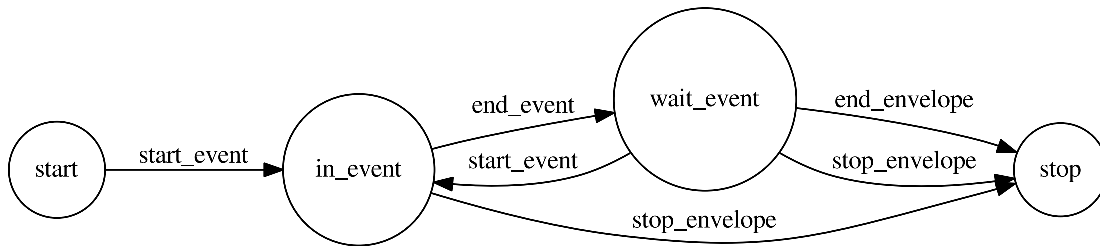
- *get_metadata()*
- ping()
- *has_clearing()*
- *has_immediate_reply()*
- *start_event()*
- *send_item()*
- *end_event()*
- *end_envelope()*
- *stop_envelope()*

6.1 Workflows

Recipients can receive multiple events concurently, the only constraint is that the broker cannot send `start_event` to a recipient until it is "ready".



Events corresponding to a particular envelope are sequential. There is no `start_envelope` event, the recipient is expected to infer the begining of envelopes with the envelope IDs passed in each event.



6.2 get_metadata

get_metadata()

Required.

Called to ask for information about the recipient.

Return type dict

Returns

A dictionary.

Required return dictionary keys:

- name (string): Name of the recipient.
- version (float): Version of the recipient.
- api_version (float): Version of the Xbus recipient API.
- host (string): Host name of the server hosting the recipient.
- start_date (string): ISO 8601 date-time of When the recipient was started.
- local_time (string): ISO 8601 time on the recipient server.

Optional return dictionary keys:

- locale (string): Locale code, in a format specified by BCP 47 <http://tools.ietf.org/html/bcp47>.

6.3 ping

ping(*token*)

Required.

Called when Xbus wants to check whether the recipient is up.

Parameters **token** (*str*) – String that must be sent back.

Returns The token string sent as parameter.

6.4 has_clearing

has_clearing()
Required.

Called to determine whether the recipient supports the "data clearing" feature; and if it does, to get more information about that process.

Returns

2-element tuple:

- Boolean indicating whether the feature is supported.
- URL of the data clearing database (or nothing if the feature is not supported). The database must respect the schema described in the "Data clearing" section of the Xbus documentation.

6.5 has_immediate_reply

has_immediate_reply()
Required.

Called to determine whether the recipient supports the "immediate reply" feature.

Returns

2-element tuple:

- Boolean indicating whether the feature is supported.
- List of event type names the recipient declares immediate reply support for.

6.6 start_event

start_event(*envelope_id*, *event_id*, *type_name*)
Required.

Called when a new event is available.

Parameters

- **envelope_id** (*str*) – [TODO] String.
- **event_id** (*str*) – [TODO] String.
- **type_name** (*str*) – [TODO] String.

Returns [TODO] tuple.

6.7 send_item

send_item(*envelope_id*, *event_id*, *indices*, *data*)
Required.

Called to send the recipient an item.

Parameters

- **envelope_id** (*str*) – [TODO] String.
- **event_id** (*str*) – [TODO] String.
- **index** (*int*) – index of the item in the event.
- **data** – [TODO] Byte array.

Returns [TODO] tuple.

6.8 end_event

end_event (*envelope_id*, *event_id*)

Required.

Called at the end of an event.

Parameters

- **envelope_id** (*str*) – [TODO] String.
- **event_id** (*str*) – [TODO] String.

Returns [TODO] tuple.

6.9 end_envelope

end_envelope (*envelope_id*)

Required.

Called once an envelope (and its individual events) has been sent.

Parameters **envelope_id** (*str*) – [TODO] String.

Returns [TODO] tuple.

6.10 stop_envelope

stop_envelope (*envelope_id*)

Required.

Called to signal an early envelope exit.

Parameters **envelope_id** (*str*) – [TODO] String.

Returns [TODO] boolean.

CHAPTER 7

Supervision

The supervision service monitors the state of recipients logged into the backend.

7.1 Configuration

Supervision is disabled by default To enable it, set the `enabled` option in the `supervision` section of broker's the configuration file.

Options:

enabled Whether to enable supervision, defaults to false

node.polling.interval How often recipients should be checked, the value is the time interval in seconds

node.polling.timeout The timeout for checking recipients, the value is the duration in seconds

node.misses.threshold The number of consecutive polling timeouts to wait before the recipient can be considered lost, the value is the number of timeouts to allow

Example configuration:

```
[supervision]
; supervision is disabled by default for backwards compatibility
enabled = true
; polling time interval in seconds, determines how often event nodes are checked
node.polling.interval = 5
; time between pings
node.polling.timeout = 1
; number of timed out pings to allow before considering the recipient as lost
node.misses.threshold = 2

backsocket = inproc://#back2sup
```

7.2 Polling

Supervision works by checking recipient regularly to make sure that they are still available. The supervisor issues an RPC call to each recipient using the *Xbus recipient API*.

If the RPC call fails, the recipient is considered to have 'missed' the call. After a number of misses (see `node.misses.threshold` above) the recipient is considered as lost.

When a recipient is deemed lost, the broker terminates that recipient's session (logout). The broker backend keeps a list of inactive recipients.

CHAPTER 8

Immediate reply Xbus feature

This document describes the “immediate reply” feature Xbus recipient nodes may implement.

If they do, they must appropriately answer the “has_immediate_reply” API call (see the section of the Xbus documentation describing Xbus recipient API calls for details).

8.1 Description

See the *Immediate reply*” section of the Xbus documentation for details about this feature.

CHAPTER 9

Data clearing Xbus feature

This document describes the “data clearing” feature Xbus recipient nodes may implement.

If they do, they must appropriately answer the “has_clearing” API call (see the section of the Xbus documentation describing Xbus recipient API calls for details).

9.1 Feature dependencies

Xbus recipient nodes providing data clearing MUST also support:

- Immediate replies.

9.2 Database schema

A database must be available and initialized with the schema described below.

TODO

9.3 Monitor-consumer communication

This section describes how the Xbus monitor and Xbus recipient nodes providing data clearing communicate.

For general duties, the Xbus monitor directly accesses the database announced by Xbus recipient node providing data clearing.

Certain specific operations, however, happen via requests emitted through the Xbus broker.

- Theses requests are enclosed into regular Xbus envelopes / events / items.
- The requests use the “immediate reply” feature (so a specific event type).
- The Xbus monitor is the emitter of these requests.
- Xbus recipient nodes providing data clearing are consumers of these requests.

Each data blob sent in “send_item” calls is a dictionary with an “action” key, referencing one of the following actions:

- “get_item_details”: Provide details about a data clearing item.

  Compulsory dictionary keys:

  – item_id: ID of the data clearing item to clear.

- “clear_items”: “Clear” a data clearing item. Compulsory dictionary keys:
  - item_id: ID of the data clearing item to clear.
  - values.

CHAPTER 10

TODO

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

C

Consumer, **10**

E

Emitter, **10**
Emitter Profile, **10**
end_envelope() (built-in function), 16
end_event() (built-in function), 16
Envelope, **9**
Event, **9**
Event Node, **10**
Event Type, **9**

G

get_metadata() (built-in function), 14

H

has_clearing() (built-in function), 15
has_immediate_reply() (built-in function), 15

I

Immediate reply, **10**
Item, **10**

P

ping() (built-in function), 14

R

Recipient, **10**
Role, **10**

S

send_item() (built-in function), 15
Service, **10**
start_event() (built-in function), 15
stop_envelope() (built-in function), 16

W

Worker, **10**