xbus.broker Documentation

Release 0.0.0

Florent Aide, Jérémie Gavrel

March 25, 2016

Contents

Contents:

CHAPTER 1

Getting started

1.1 With docker

Xbus is packaged using docker to help deploy it. The docker images will soon be ready 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 bizzare 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.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.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.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.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 importants terms are:

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

4.1 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.

4.2 Event Type

Each and every event in xbus needs only on 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.

4.3 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.

4.4 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

4.5 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.

4.6 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.

4.7 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.

4.8 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*.

4.9 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*.

4.10 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*.

4.11 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

Indices and tables

- genindex
- modindex
- search