EVE Market Data Relay Documentation

Release 0.1

Greg Taylor

May 01, 2017

Contents

With the introduction of native-to-EVE market APIs, the EMDR project has ran its course. As of May 1, 2017, we have shuttered the network. This repo will remain in an archived state. Thanks to all who helped make EMDR a success!

EVE Market Data Relay (EMDR) is a super scalable, highly available firehose of real-time market data. For those that wish to record price and history data as it comes in, EMDR will help you do so as efficiently and reliably as possible. EMDR's data feed is open to the public, and is developed as an open source project.

EMDR may appeal to you if:

- You need real-time access to market data, as soon as possible. Perhaps for sending out price/inventory level alerts, notification of lucrative trade routes, or real-time charts and graphs.
- You want to record prices over time.
- You want the absolutely most complete set of data that you can get.
- The effort and overhead of getting large amounts of direct player uploads to your site is too much to bear.

EMDR's primary goals are:

- Ensuring that all market sites have access to player-uploaded market data.
- Extremely high reliability.
- Minimize expense to those running EMDR (shared burden).
- Minimize expense to those consuming the feed (bandwidth).

For a more complete run-down, see *A High-Level Overview*.

License: EVE Market Data Relay is licensed under the BSD License.

CHAPTER 1

Assorted Info

- Mailing list - If you are consuming the feed, make sure to subscribe to this for important announcements. This is also one of the best places to ask questions or discuss EMDR stuff.
- Slack Channel - join #emdr on tweetfleet.slack.com, an excellent place for getting quick help, or hanging out with other developers and consumers. Get your account here if you don't have one!
- Issue tracker - Report bugs here.
- GitHub project - Source code and issue tracking.
- EMDR monitor - EMDR relay/announcer monitor web app.
- EMDR map - See the solar systems light up as market data arrives.
- @gctaylor Twitter - Tweets from the maintainer.

CHAPTER 2

General Topics

The following topics are higher-level overviews, and general documentation. If you are just curious, or wondering how to upload data to EMDR, this section is all you need.

A High-Level Overview

Project Motivation

For the application developer, there are quite a few barriers to entry for those wishing to write market-data-driven applications. A developer would probably need to:

- Either write their own uploader client, or re-purpose an existing uploader.
- Write a service/API to accept the data in whatever format(s) the uploader(s) they'd like to support use.
- Actually get players to point their uploader at said service/API (This is the hardest part!)
- Probably pull data from other market sites to flesh out their data set.
- Then, and only then, start writing the *fun* part of their application as the amount of data coming in slowly grows. By this point, they are probably a ways down the road to burnout.

None of these tasks are fun, they all involve re-inventing the wheel. By the time the developer gets through all of this (if they do), burnout is a distinct possibility. All before getting to the fun stuff!

EVE Market Data Relay (EMDR) allows you to forgo all of this drudgery, and instead, connect to a firehose of data in the standardized Unified Uploader Data Interchange Format format. EMDR's ZeroMQ underpinnings also make it easier, and exponentially more efficient than accepting HTTP uploads directly.

Core Principles

During the early design and development of EMDR, these were the main pillars we built on:

- There should be no single point of failure. Every component of the architecture should be simple to make redundant using trusted volunteered machines.
- The application must be able to accept an extremely large number of incoming market orders without performance issues.
- The cost for people hosting parts of EMDR's network should be kept to an absolute minimum. This means being stingey with CPU, RAM, and bandwidth. Likewise, consuming the feed shouldn't break the bank, either.
- It must be very easy to scale the system without restarts/reconfigs on the primary setup.
- The broadcasting of the market data needs to happen in a "fan out" manner. In this way, we can keep adding additional subscribers without running into scalability issues.

How it all fits together

For any given submitted market order, here is the flow said order goes through:

```
(Gateway) -> (Announcer) -> (Relays) -> (Applications)
```

First, the order hits the **Gateway**, which is a simple HTTP application that parses the message. Incoming messages are in Unified Uploader Data Interchange Format.

The Gateway interprets the message, validates it, normalizes anything weird, then pipes it to all of the root-level **Announcers** in the network.

The **Announcer** is the first tier of our market data distribution. Announcers relay any data they receive to **Relays** that are connected to the Announcer. There are only a few Announcers, and these only accept connections from approved Relays. Most relays connect to multiple announcers for added redundancy.

The **Relay**, like the Announcer, is a dumb repeater of everything it receives. Relays receive data from their Announcers, then pipe it out to any subscribers that are connected to them. Subscribers can be other **Relays**, or actual user sites/applications.

By using our system of Relays, we keep bandwidth usage and costs lower on the top-level Announcers. We are also able to keep "fanning out" to improve redundancy and serve greater numbers of consumers without large increases in bandwidth utilization.

We are left with a very efficient, very sturdy data relay network. The next section goes into detail about fault-tolerance.

High Availability through shared burden

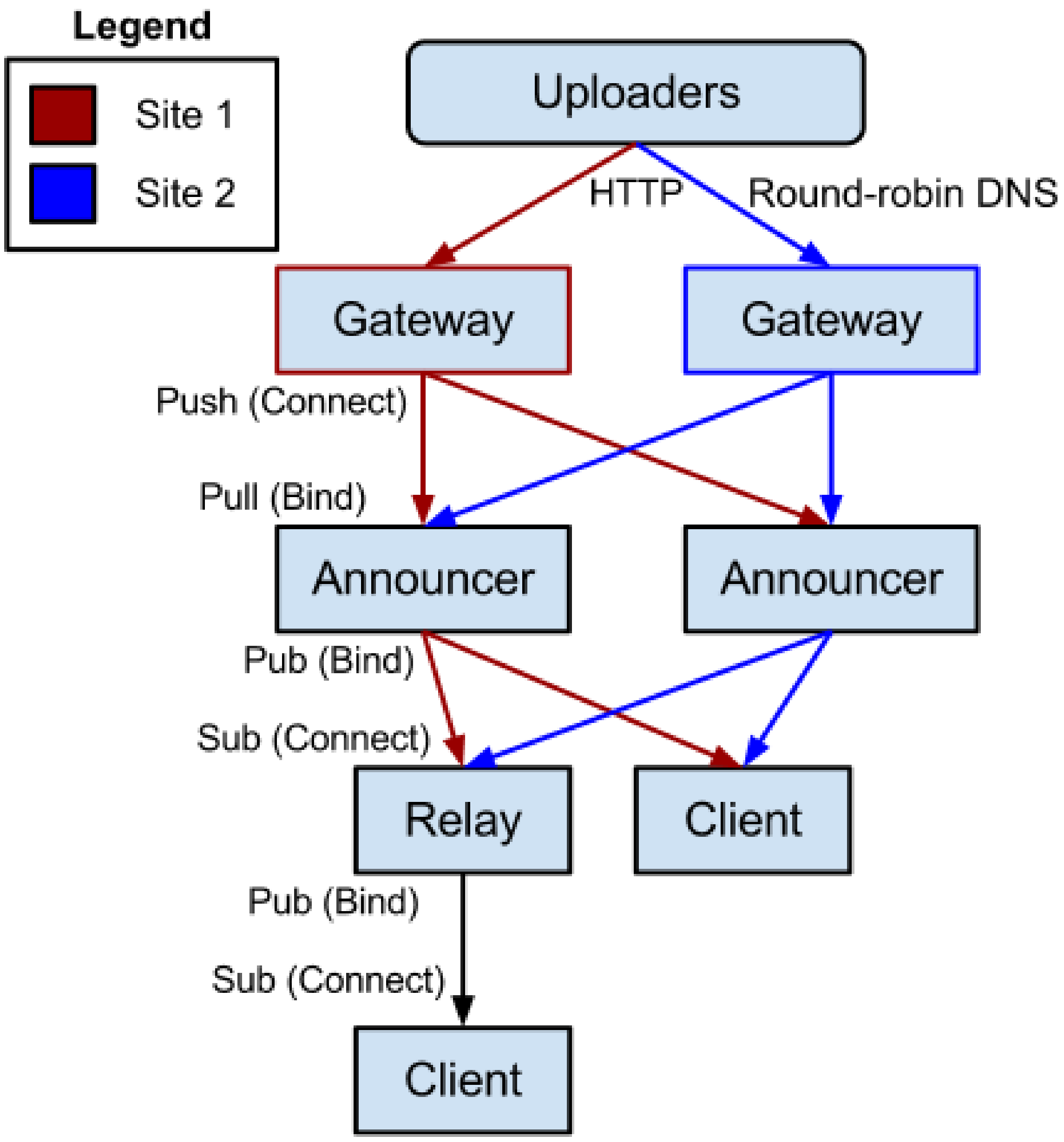
EMDR is architected in a way that allows every single component to be replicated. We can easily add additional daemons at each level of the stack in order to improve availability, or to spread costs.

HTTP Uploads are dispersed to Gateways via Round-Robin DNS, which is a simple way to distribute the traffic across multiple machines. For each additional Gateway added to DNS rotation, incoming bandwidth consumption drops for the whole pool as the load is divided. If at any time one of the gateways becomes unreachable, it is automatically removed from the DNS rotation.

In the diagram below, we see a rough representation of our current deployment. Site 1 is comprised of EMDR running on Greg Taylor's (the project maintainer) machines, and Site 2 is a separate copy running in another data center. The relays are all ran by different volunteers.

Note: We are not limited to just two instances of EMDR, there is no hard limit. Additionally, we'll mostly scale by adding more Gateways, since additional Announcers are only for redundancy.

At every step of the entire flow, we can afford to lose one of the two daemons without a service interruption. The infrastructure can be scaled well out past the initial two sites, if need be.



Security

Security is something we take seriously, but let's consider the current reality of market data with EVE sites: *Players upload market data directly to market sites.* We are no less secure than that. Uploads can be faked, and malicious payloads can be sent, though EMDR will do its best to catch anything harmful.

Note: As a consumer, you may wish to cross-reference incoming data. In many cases, you will get the same data point multiple times, as several players upload the same thing. This can be used to your advantage.

Technology Used

This is the least interesting part of the overview, so it goes towards the ends.

- EMDR is written in Python.
- All network-related stuff is handled by ZeroMQ, which is an incredibly simple and performant networking library.
- gevent is used for their excellent greenlet-based Queue, Workers, and async network I/O.
- The gateway HTTP servers run bottle.

The entire stack is super low overhead, and very fast.

Volunteering

If you would like to volunteer computing resources to the EMDR network, see *Volunteering computing resources* for more details.

Sites using EMDR

EMDR delivers nearly a million messages a day to a number of different projects around the world. Some of these projects are listed below. If we're missing a project, please file an issue in the issue tracker with the name and a URL to the project. Bonus points for describing what you're doing.

Market sites

- Element 43
- EVE Addicts
- EVE Marketdata
- Eveonomics

Uploading Market data to EMDR

Uploading to EMDR contributes data to for public use. Feeding the firehose benefits us all, so please do consider pointing your uploader at the network.

With EVEMon

To upload data with EVEMon, you need only have it installed and running. Market data is uploaded to EMDR by default.

With EMDU (Mac, Linux, and Windows)

EMDU (EVE Market Data Uploader) is a cross-platform, console-based market uploader client. It uploads directly to EMDR. For those who are running Mac or Linux, this client should run beautifully for you. It runs on Windows, as well, but it's probably easier to install EVEMon.

See the the install instructions for how to get started.

With other clients

While we prefer EVEMon, you can use any client that supports the Unified Uploader Data Interchange Format. An up to date list is maintained here: Clients supporting UUDIF.

Steps vary from client to client, but here is the typical process:

- Open the dialog that lets you specify where to send market data.
- Create a new endpoint. Select Unified format if it asks.
- Set the URL to: http://upload.eve-emdr.com/upload/
- Enter your upload key, if you feel like it. Otherwise, just make something up or leave it blank.
- Hit save, and start uploading.

You can then use any market service's auto-uploader pages.

CHAPTER 3

Consumer Documentation

The following topics are useful to those wishing to connect to and use EMDR's data feed.

Data Sources

EMDR is 'fed' data by player uploads. There are a number of clients that can be used to chip in, outlined in more detail on *Uploading Market data to EMDR*.

Uploader applications monitor the EVE Online cache files on your machine, which are populated with market data as you browse around the in-game market dialogs.

Individual market orders are uploaded, along with market history (if you switch to history tabs for items).

Most well-designed uploader applications use small enough amounts of CPU and bandwidth as to run unnoticed.

Getting access to the EMDR network

In order to get access to the EMDR network, you will merely need to connect to a relay. For your convenience, we have listed the relays below that have available capacity. If you'd like to verify the health of a relay before using it, check out our EMDR monitor.

URI	ISP	Location	Access	Notes
tcp://relay-us-west-1.eve-emdr.com:8050	Cogent	Sacramento, CA	Open	West coast US realy. Volunteered by Yann of EVE Central.
tcp://relay-us-central-1.eve-emdr.com:8050	Ubuquity Hosting	Chicago, IL	Open	Central US relay. Volunteered by udsaxman.
tcp://relay-eu-germany-1.eve-emdr.com:8050	Hetzner	Germany	Open	German relay. Volunteered by FuzzySteve.
tcp://relay-eu-germany-2.eve-emdr.com:8050	Hetzner	Germany	Open	German relay. Volunteered by Agedon Group, Inc.
tcp://relay-eu-germany-3.eve-emdr.com:8050	Intergenia	Germany	Open	German relay. Volunteered by EVE-HQ.com.
tcp://relay-eu-germany-4.eve-emdr.com:8050	Hetzner	Germany	Open	German relay. Volunteered by Karbowiak.
tcp://relay-eu-denmark-1.eve-emdr.com:8050	ComX	Denmark	Open	Volunteered by Karbowiak.

Once you have chosen a relay, simply adapt the sample in *Connecting to the EMDR network* to use the hostname/port of the relay of your choice.

Tip: If reliability is a concern, you'll want to connect to multiple relays and handle the duplicate messages. This will keep you running even if one of the relays you're using dies.

Note: Some relays, marked as *Restricted*, require that you request access to use them. You will need to get in touch with the relay via the contact info listed to work it out with the admin.

Connecting to the EMDR network

In order to connect to the EVE Market Data Relay feed, you'll need to use the ZeroMQ bindings for the language of your choice. This involves installing ZeroMQ, then installing the bindings. See your language's section below for a link to its bindings.

Once you have ZeroMQ plus bindings installed, you'll need to choose a Relay to connect to. See *Getting access to the EMDR network* for a list, and any potential special case instructions for each relay. After that, you'll be set to connect and begin receiving data.

Data Format

All data coming out of EMDR is in Unified Uploader Data Interchange Format, which is a JSON-based standard for market orders and history. See the spec for more details.

An important note on keeping up

Our relays use PUB/SUB to broadcast market data to the consumers. Within each relay is a buffer for each consumer. If the consumer is having a hard time keeping up with the flow of data from the relay, the relay's send buffer will gradually fill, since data is being produced faster than the consumer can receive it. Once the buffer reaches a certain size, we start discarding messages, which can be a very bad thing for your application. Here is the relevant quote from the ZeroMQ documentation:

```
When a ZMQ_PUB socket enters an exceptional state due to having reached
the high water mark for a subscriber, then any messages that would be
sent to the subscriber in question shall instead be dropped until the
exceptional state ends. The zmq_send() function shall never block for
this socket type.
```

The important take-away is that when designing your consumers, you'll want to either do as little processing in your consumer as possible, or make sure that your network IO remains async or non-blocking so that you don't lose any messages.

Tip: While the examples below are a good way to get you started, you will probably need to adapt them with this IO concern in mind as you add more.

Examples for various languages

Below are a few examples of how to connect to the data feed. If you see anything wrong with the examples below, please let us know on the issue tracker. The original author of this documentation is only familiar with Python.

Python

The following example uses the pyzmq module (available off of PyPi) and simplejson. For a more complete list of examples, see the Python examples dir on github.:

```python
"""
Example Python EMDR client.
"""
import zlib
import zmq
# You can substitute the stdlib's json module, if that suits your fancy
import simplejson

def main():
    context = zmq.Context()
    subscriber = context.socket(zmq.SUB)

    # Connect to the first publicly available relay.
    subscriber.connect('tcp://relay-us-central-1.eve-emdr.com:8050')
    # Disable filtering.
    subscriber.setsockopt(zmq.SUBSCRIBE, "")

    while True:
        # Receive raw market JSON strings.
        market_json = zlib.decompress(subscriber.recv())
        # Un-serialize the JSON data to a Python dict.
        market_data = simplejson.loads(market_json)
        # Dump the market data to stdout. Or, you know, do more fun
        # things here.
        print market_data

if __name__ == '__main__':
    main()
```

PHP

PHP accesses EMDR via ZeroMQ's php-zmq PHP bindings:

```php
<?php
/*
 * Example PHP EMDR client.
 */

$context = new ZMQContext();
$subscriber = $context->getSocket(ZMQ::SOCKET_SUB);

// Connect to the first publicly available relay.
$subscriber->connect("tcp://relay-us-central-1.eve-emdr.com:8050");
// Disable filtering.
$subscriber->setSockOpt(ZMQ::SOCKOPT_SUBSCRIBE, "");

while (true) {
    // Receive raw market JSON strings.
    $market_json = gzuncompress($subscriber->recv());
    // Un-serialize the JSON data to a named array.
    $market_data = json_decode($market_json);
    // Dump the market data to stdout. Or, you know, do more fun things here.
    var_dump($market_data);
}
```

Ruby

Ruby accesses EMDR via ZeroMQ's zmq Ruby bindings:

```ruby
#
# Synchronized subscriber
#

require 'rubygems'
require 'ffi-rzmq'
require 'json'
require 'zlib'

context = ZMQ::Context.new
subscriber = context.socket(ZMQ::SUB)

# Connect to the first publicly available relay.
subscriber.connect("tcp://relay-us-central-1.eve-emdr.com:8050")
subscriber.setsockopt(ZMQ::SUBSCRIBE,"")

loop do
  # Receive raw market JSON strings.
  subscriber.recv_string(string = '')
  # Un-compress the stream.
  market_json = Zlib::Inflate.new(Zlib::MAX_WBITS).inflate(string)
  # Un-serialize the JSON data.
  market_data = JSON.parse(market_json)
  # Dump the market data to stdout. Or, you know, do more fun things here.
  puts market_data
end
```

Go

```go
package main

import (
	"log"
	"bytes"
	"io/ioutil"
	"compress/zlib"
	zmq "github.com/pebbe/zmq2" // or zmq3/zmq4
)

// go run emdr_client_example.go
func main() {
	client, err := zmq.NewSocket(zmq.SUB)
	if err != nil {
		log.Fatal(err)
	}

	// Connect
	err = client.Connect("tcp://relay-us-central-1.eve-emdr.com:8050")
	client.SetSubscribe("")
	if err != nil {
		log.Fatal(err)
	}

	// Endless loop.
	for {
		// Receive message from ZeroMQ.
		msg, err := client.Recv(0)
		if err != nil {
			log.Fatal(err)
		}

		// Prepare to decode.
		decoded, err := ZlibDecode(msg)
		if err != nil {
			log.Fatal(err)
		}

		// Output as json string, should do something useful at this point ...
		log.Printf("%s", decoded)
	}
}

func ZlibDecode(encoded string) (decoded []byte, err error) {
	b := bytes.NewBufferString(encoded)
	pipeline, err := zlib.NewReader(b)

	if err == nil {
		defer pipeline.Close()
		decoded, err = ioutil.ReadAll(pipeline)
	}

	return
}
```

C#

C# accesses EMDR via ZeroMQ's clrzmq binding:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using System.Linq;
using System.Text;
using System.Web.Script.Serialization; // Needs reference to 'System.Web.Extensions.
↪dll'
using ZMQ; // Needs reference to 'clrzmq.dll' and adding 'libzmq.dll' to project
           // 'clrzmq' can be found at: https://github.com/zeromq/clrzmq/downloads

namespace EMDR_Client
{
    public class Program
    {
        private static void Main()
        {
            using (var context = new Context())
            {
                using (var subscriber = context.Socket(SocketType.SUB))
                {
                    //Connect to the first publicly available relay.
                    subscriber.Connect("tcp://relay-us-central-1.eve-emdr.com:8050");

                    // Disable filtering.
                    subscriber.SetSockOpt(SocketOpt.SUBSCRIBE, Encoding.UTF8.GetBytes(
↪""));

                    // Alternatively 'Subscribe' can be used
                    //subscriber.Subscribe("", Encoding.UTF8);

                    while (true)
                    {
                        try
                        {
                            // Receive compressed raw market data.
                            var receivedData = subscriber.Recv();

                            // The following code lines remove the need of 'zlib'␣
↪usage;
                            // 'zlib' actually uses the same algorith as
↪'DeflateStream'.
                            // To make the data compatible for 'DeflateStream', we␣
↪only have to remove
                            // the four last bytes which are the adler32 checksum and
                            // the two first bytes which are the 'zlib' header.
                            byte[] decompressed;
                            byte[] choppedRawData = new byte[(receivedData.Length -␣
↪4)];
                            Array.Copy(receivedData, choppedRawData, choppedRawData.
↪Length);
                            choppedRawData = choppedRawData.Skip(2).ToArray();

                            // Decompress the raw market data.
                            using (MemoryStream inStream = new␣
↪MemoryStream(choppedRawData))
                            using (MemoryStream outStream = new MemoryStream())
                            {
                                DeflateStream outZStream = new DeflateStream(inStream,
↪ CompressionMode.Decompress);
                                outZStream.CopyTo(outStream);
                                decompressed = outStream.ToArray();
                            }

                            // Transform data into JSON strings.
                            string marketJson = Encoding.UTF8.GetString(decompressed);

                            // Un-serialize the JSON data to a dictionary.
                            var serializer = new JavaScriptSerializer();
                            var dictionary = serializer.Deserialize<Dictionary<string,
↪ object>>(marketJson);

                            // Dump the market data to console or, you know, do more␣
↪fun things here.
                            foreach (KeyValuePair<string, object> pair in dictionary)
                            {
                                Console.WriteLine("{0}: {1}", pair.Key, pair.Value);
                            }
                            Console.WriteLine();
                        }
                        catch (ZMQ.Exception ex)
                        {
                            Console.WriteLine("ZMQ Exception occurred : {0}", ex.
↪Message);
                        }
                    }
                }
            }
        }
    }
}
```

Visual Basic

Visual Basic, like C#, accesses EMDR via ZeroMQ's clrzmq binding:

```
Imports System.Text
Imports System.IO
Imports System.IO.Compression
Imports System.Web.Script.Serialization ' Needs reference to 'System.Web.Extensions.
↪dll'
Imports ZMQ ' Needs reference to 'clrzmq.dll' and adding 'libzmq.dll' to project
            ' 'clrzmq' can be found at: https://github.com/zeromq/clrzmq/downloads

Module MainModule

    Sub Main()
        Using context = New Context()
            Using subscriber = context.Socket(SocketType.SUB)

                'Connect to the first publicly available relay.
                subscriber.Connect("tcp://relay-us-central-1.eve-emdr.com:8050")

                ' Disable filtering.
                subscriber.SetSockOpt(SocketOpt.SUBSCRIBE, Encoding.UTF8.GetBytes(""))

                ' Alternatively 'Subscribe' can be used.
                'subscriber.Subscribe("", Encoding.UTF8)

                While True
                    Try
                        ' Receive compressed raw market data.
                        Dim receivedData() = subscriber.Recv()

                        ' The following code lines remove the need of 'zlib' usage;
                        ' 'zlib' actually uses the same algorith as 'DeflateStream'.
                        ' To make the data compatible for 'DeflateStream', we only␣
↪have to remove
                        ' the four last bytes which are the adler32 checksum and
                        ' the two first bytes which are the 'zlib' header.
                        Dim decompressed() As Byte
                        Dim choppedRawData(receivedData.Length - 4) As Byte
                        Array.Copy(receivedData, choppedRawData, choppedRawData.
↪Length)
                        choppedRawData = choppedRawData.Skip(2).ToArray()

                        ' Decompress the raw market data.
                        Using inStream = New MemoryStream(choppedRawData)
                            Using outStream = New MemoryStream()
                                Dim outZStream = New DeflateStream(inStream,␣
↪CompressionMode.Decompress)
                                outZStream.CopyTo(outStream)
                                decompressed = outStream.ToArray
                            End Using
                        End Using

                        ' Transform data into JSON strings.
                        Dim marketJson = Encoding.UTF8.GetString(decompressed)

                        ' Un-serialize the JSON data to a dictionary.
                        Dim serializer = New JavaScriptSerializer()
                        Dim dictionary = serializer.Deserialize(Of Dictionary(Of␣
↪String, Object))(marketJson)

                        ' Dump the market data to console or, you know, do more fun␣
↪things here.
                        For Each pair In dictionary
                            Console.WriteLine("{0}: {1}", pair.Key, pair.Value)
                        Next
                        Console.WriteLine()
                    Catch ex As Exception
                        Console.WriteLine("ZMQ Exception occurred : {0}", ex.Message)
                    End Try
                End While
            End Using
        End Using
    End Sub
End Module
```

Perl

Perl uses the ZeroMQ-Perl binding for Perl:

```perl
#!/usr/bin/perl
use warnings;
use strict;
$|=1;

use ZeroMQ qw/:all/;

my $cxt = ZeroMQ::Context->new;
my $sock = $cxt->socket(ZMQ_SUB);
$sock->connect('tcp://relay-us-central-1.eve-emdr.com:8050');
$sock->setsockopt(ZMQ_SUBSCRIBE, "");

while (1) {
    my $msg = $sock->recv();
    last unless defined $msg;

    use Compress::Zlib;
    my $json = uncompress($msg->data);

    use JSON;
    my $data = decode_json($json);

    use Data::Dumper;
    print Dumper($data),"\n\n";
}
```

Java

Java uses jzmq binding:

```java
/*
 * Example Java EMDR client.
 */

import org.zeromq.*; // https://github.com/zeromq/jzmq
import org.json.simple.*; // http://code.google.com/p/json-simple/downloads/list
import org.json.simple.parser.*;
import java.util.zip.*;

public class EMDR_Client {

    public static void main(String[] args) throws Exception {

        ZMQ.Context context = ZMQ.context(1);
        ZMQ.Socket subscriber = context.socket(ZMQ.SUB);

        // Connect to the first publicly available relay.
        subscriber.connect("tcp://relay-us-central-1.eve-emdr.com:8050");

        // Disable filtering.
        subscriber.subscribe(new byte[0]);

        while (true) {
            try {
                // Receive compressed raw market data.
                byte[] receivedData = subscriber.recv(0);

                // We build a large enough buffer to contain the decompressed data.
                byte[] decompressed = new byte[receivedData.length * 16];

                // Decompress the raw market data.
                Inflater inflater = new Inflater();
                inflater.setInput(receivedData);
                int decompressedLength = inflater.inflate(decompressed);
                inflater.end();

                byte[] output = new byte[decompressedLength];
                System.arraycopy(decompressed, 0, output, 0, decompressedLength);

                // Transform data into JSON strings.
                String market_json = new String(output, "UTF-8");

                // Un-serialize the JSON data.
                JSONParser parser = new JSONParser();
                JSONObject market_data = (JSONObject)parser.parse(market_json);

                // Dump the market data to console or, you know, do more fun things␣
↪here.
                System.out.println(market_data);
            } catch (ZMQException ex) {
                System.out.println("ZMQ Exception occurred : " + ex.getMessage());
            }
        }
    }
}
```

Erlang

Erlang uses erlzmq2 binding:

```
#!/usr/bin/env escript

% you will need the ZeroMQ Erlang library: https://github.com/zeromq/erlzmq2
% I also use jiffy for Json: https://github.com/davisp/jiffy

main(_Args) ->
  {ok, Context} = erlzmq:context(),
  {ok, Subscriber} = erlzmq:socket(Context, sub),
  ok = erlzmq:connect(Subscriber,"tcp://relay-us-central-1.eve-emdr.com:8050"),
  ok = erlzmq:setsockopt(Subscriber, subscribe, <<>>),
  msgcheck(Subscriber).

msgcheck(Subscriber) ->
  {ok,Msg} = erlzmq:recv(Subscriber),
  io:format("~p\n",[jiffy:decode(zlib:uncompress(Msg))]),
  msgcheck(Subscriber).
```

Node.js

Node.js uses the zeromq.node binding:

```
/*
 *  Example node.js EMDR client
 */

var zmq = require('zmq');
var zlib = require('zlib');

var sock = zmq.socket('sub');

// Connect to the first publicly available relay.
sock.connect('tcp://relay-us-central-1.eve-emdr.com:8050');
// Disable filtering
sock.subscribe('');

sock.on('message', function(msg){
    // Receive raw market JSON strings.
    zlib.inflate(msg, function(err, market_json) {
        // Un-serialize the JSON data.
        var market_data = JSON.parse(market_json);

        // Do something useful
        console.log(market_data);
    });
});
```

Design considerations for consumers

This document outlines some useful tips in designing your consumer applications. If you have anything to add, post an entry on our issue tracker.

Keeping up with market data

As EMDR grows, the volume of market data you see over a given span of time will continue to increase. This means that you'll need to design with concurrency in mind.

Ideally, you have a dedicated process that is just enough to connect to EMDR and save the data to your DB backend. We suggest doing any aggregation or additional processing in another process, to make sure you don't lose any data due to blocking or bugs introduced in your processing/aggregation code. If your consumer can't process the incoming data fast enough to keep up with the relay, we end up discarding pending messages on the relay to prevent buffer overflow. The end result is that you will lose messages.

For an idea of what this looks like, see our greenlet_consumer code example. This is written in Python, using gevent to perform the DB saves using greenlets, which are micro-threads. Most languages have something similar available, so don't let the fact that this is in Python psyche you out if you're using another language.

Deal with duplicate data

You will see some duplicate data coming down through EMDR. There are a few different kinds of duplication:

- Multiple players are sitting in Jita, looking at Module X at about the same time. You'll see two individual messages, containing the same (or very similar) data.
- Another market service uploads a message that has already been through EMDR. This is a duplicate in its purest sense. We will do our best to hunt this down and take care of it for you, but do design with it in mind.

Some elect to store every individual data point for each item. This is a viable approach, and not extremely expensive. Your aggregator process can go through data as it's coming in to look for suspicious patterns. Duplicate data can be a valuable means of cross-checking incoming data, in this case.

Others only store the current price for items, using the generatedAt values to determine whether the message contains newer data than they have.

Don't be too trusting

The reality of player-upload-driven market data sites is that we are at the mercy of said players as far as the data goes. The vast majority of uploaders are going to send good data. However, there is a minority that does not play so nicely.

In many cases, multiple players will upload the details for the same orders multiple times. This can be used to your advantage, in that you can cross-check things as they come in. If one message says Large Shield Extender I is going for 5 billion isk in Jita, but another three are saying much lower than that, your outlier is probably fraudulent and is best ignored.

You also have the option of cross-referencing the APIs of other sites who do not consume EMDR data. While this can defeat some of the purpose of using EMDR, the option is there to complement the feed.

Drop the banhammer on vandals

Uploaders may be uniquely identified via the `EMDR` key/value pair in each message's `uploadKeys` list. The value of the `EMDR` key is a salted, hashed string unique to the uploader's IP address. While this may be spoofed, it will offer some ability to blacklist obviously malicious users.

Use EMDR's redundancy to your advantage

EMDR is built with high availability in mind. Our only single point of failure is Amazon's Route 53 DNS service, which has an excellent reliability track record.

While you can connect to only one relay, you may wish to connect your consumer to two. This will allow your consumer to keep functioning, even if one of the relays it is subscribed to dies a fiery death. The only complication is that you will need to de-dupe the data coming in, as you'll be receiving two copies of each message (one from each relay).

Optionally, fire up a private EMDR relay within your infrastructure and consume from that. It'll do the de-duplication for you.

CHAPTER 4

EMDR Developer Documentation

The following topics will be useful to you if you would like to help improve EMDR, or volunteer additional computing resources to the network.

Installation

- pip install -r requirements.txt
- ???
- profit

Volunteering computing resources

EMDR is ran by volunteers who foot the cost in order for everyone to get access to market data. While running pieces of EMDR is not something that will get you fame or notoriety, it is crucial to the continued survival and success of the network.

If you have idle computing resources, or would like to share capacity on a machine, we'd love to have it. `gtaylor` is available to assist with setup and configuration.

Current status

We currently have everything that we need. Special thanks to everyone who volunteered to help make this happen. Should our needs change, this page will be updated, and announcement will land on the mailing list.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search