Ichnaea

Release 2.3

unknown

Aug 04, 2023

CONTENTS

1 Table of contents **3**
- 1.1 User documentation . . . 3
- 1.2 Development/Deployment documentation . . . 21
- 1.3 Algorithms . . . 72
- 1.4 Changelog . . . 83
- 1.5 Glossary . . . 111

2 Indices **113**

3 Source code and license **115**

4 About the name **117**

Index **119**

Ichnaea is a service to provide geolocation coordinates from other sources of data (Bluetooth, cell or WiFi networks, GeoIP, etc.). It uses both *Cell-ID* and Wi-Fi based positioning (*WPS*) approaches.

Mozilla hosts an instance of this service, called the Mozilla Location Service (*MLS*).

You can interact with the service in two ways:

- If you know where you are, submit information about the radio environment to the service to increase its quality.
- or locate yourself, based on the radio environment around you.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 User documentation

This section covers information for using the APIs directly as well as through applications and libraries.

1.1.1 Services API

The service APIs accept data submission for geolocation stumbling as well as reporting a location based on IP addresses, cell, or WiFi networks.

New client developments should use the *Region: /v1/country*, *Geolocate: /v1/geolocate*, or *Geosubmit Version 2: /v2/geosubmit* APIs.

Requesting an API Key

The api key has a set daily usage limit of about 100,000 requests. As we aren't offering a commercial service, please note that we do not make any guarantees about the accuracy of the results or the availability of the service.

Please make sure that you actually need the raw API access to perform geolocation lookups. If you just need to get location data from your web application, you can directly use the HTML5 API.

To apply for an API key, please fill out this form. When filling out the form, please make sure to describe your use-case and intended use of the service. Our Developer Terms of Service govern the use of MLS API keys.

We'll try to get back to you within a few days, but depending on vacation times it might take longer.

API Access Keys

Note: Mozilla is currently evaluating its MLS service and terms and is not currently distributing API keys.

You can anonymously submit data to the service without an API key via any of the submission APIs.

You must identify your client to the service using an API key when using one of the *Region: /v1/country* or *Geolocate: /v1/geolocate* APIs.

If you want or need to specify an API key, you need to be provide it as a query argument in the request URI in the form:

```
https://location.services.mozilla.com/<API>?key=<API_KEY>
```

Each API key can be rate limited per calendar day, but the default is to allow an unlimited number of requests per day.

Errors

Each of the supported APIs can return specific error responses. In addition there are some general error responses.

Invalid API Key

If an API key was required but either no key was provided or the provided key was invalid, the service responds with a `keyInvalid` message and HTTP 400 error code:

```
{
    "error": {
        "errors": [{
            "domain": "usageLimits",
            "reason": "keyInvalid",
            "message": "Missing or invalid API key."
        }],
        "code": 400,
        "message": "Invalid API key"
    }
}
```

API Key Limit

API keys can be rate limited. If the limit for a specific API key is exceeded, the service responds with a `dailyLimitExceeded` message and a HTTP 403 error code:

```
{
    "error": {
        "errors": [{
            "domain": "usageLimits",
            "reason": "dailyLimitExceeded",
            "message": "You have exceeded your daily limit."
        }],
        "code": 403,
        "message": "You have exceeded your daily limit."
    }
}
```

Parse Error

If the client sends a malformed request, typically sending malformed or invalid JSON, the service will respond with a `parseError` message and a HTTP 400 error code:

```
{
    "error": {
        "errors": [{
            "domain": "global",
            "reason": "parseError",
            "message": "Parse Error"
        }],
```

(continues on next page)

(continued from previous page)

```
        "code": 400,
        "message": "Parse Error"
        "details": {
            "decode": "JSONDecodeError('Expecting value: line 1 column 1 (char 0)')"
        }
    }
}
```

The `details` item will be a mapping with the key `"decode"` or `"validation"`. If the key is `"decode"`, the value will be a string describing a fundamental decoding issue, such as failing to decompress gzip content, to convert to unicode from the declared charset, or to parse as JSON. If the key is `"validation"`, the value will describe validation errors in the JSON payload.

Service Error

If there is a transient service side problem, the service might respond with HTTP 5xx error codes with unspecified HTTP bodies.

This might happen if part of the service is down or unreachable. If you encounter any 5xx responses, you should retry the request at a later time. As a service side problem is unlikely to be resolved immediately, you should wait a couple of minutes before retrying the request for the first time and a couple of hours later if there's still a problem.

APIs

Geolocate: /v1/geolocate

Purpose: Determine the current location based on data provided about nearby Bluetooth, cell, or WiFi networks and the IP address used to access the service.

- *Request*
- *Field Definition*
  - *Global Fields*
  - *Bluetooth Beacon Fields*
  - *Cell Tower Fields*
  - *WiFi Access Point Fields*
  - *Fallback Fields*
  - *Deviations From GLS API*
- *Response*

Request

Geolocate requests are submitted using a POST request to the URL:

```
https://location.services.mozilla.com/v1/geolocate?key=<API_KEY>
```

This implements a similar interface as the Google Maps Geolocation API endpoint also known as *GLS* or Google Location Service API. Our service implements all of the standard GLS API with a couple of additions.

Geolocate requests are submitted using an HTTP POST request with a JSON body.

Here is a minimal example body using only WiFi networks:

```
{
    "wifiAccessPoints": [{
        "macAddress": "01:23:45:67:89:ab",
        "signalStrength": -51
    }, {
        "macAddress": "01:23:45:67:89:cd"
    }]
}
```

A minimal example using a cell network:

```
{
    "cellTowers": [{
        "radioType": "wcdma",
        "mobileCountryCode": 208,
        "mobileNetworkCode": 1,
        "locationAreaCode": 2,
        "cellId": 1234567,
        "signalStrength": -60
    }]
}
```

A complete example including all possible fields:

```
{
    "carrier": "Telecom",
    "considerIp": true,
    "homeMobileCountryCode": 208,
    "homeMobileNetworkCode": 1,
    "bluetoothBeacons": [{
        "macAddress": "ff:23:45:67:89:ab",
        "age": 2000,
        "name": "beacon",
        "signalStrength": -110
    }],
    "cellTowers": [{
        "radioType": "wcdma",
        "mobileCountryCode": 208,
        "mobileNetworkCode": 1,
        "locationAreaCode": 2,
        "cellId": 1234567,
        "age": 1,
```

(continues on next page)

(continued from previous page)

```
        "psc": 3,
        "signalStrength": -60,
        "timingAdvance": 1
    }],
    "wifiAccessPoints": [{
        "macAddress": "01:23:45:67:89:ab",
        "age": 3,
        "channel": 11,
        "frequency": 2412,
        "signalStrength": -51,
        "signalToNoiseRatio": 13
    }, {
        "macAddress": "01:23:45:67:89:cd"
    }],
    "fallbacks": {
        "lacf": true,
        "ipf": true
    }
}
```

Field Definition

All of the fields are optional, but in order to get a Bluetooth or WiFi based position estimate at least two networks need to be provided and include a `macAddress`. The two networks minimum is a mandatory privacy restriction for Bluetooth and WiFi based location services.

Cell based position estimates require each cell record to contain at least the five `radioType`, `mobileCountryCode`, `mobileNetworkCode`, `locationAreaCode`, and `cellId` values.

Position estimates do get a lot more precise if in addition to these unique identifiers at least `signalStrength` data can be provided for each entry.

Note that all the cell JSON keys use the same names for all radio types, generally using the official GSM name to denote similar concepts even though the actual client side API's might use different names for each radio type and thus must be mapped to the JSON keys.

Global Fields

carrier The clear text name of the cell carrier / operator.

considerIp Should the clients IP address be used to locate it; defaults to true.

homeMobileCountryCode The mobile country code stored on the SIM card.

homeMobileNetworkCode The mobile network code stored on the SIM card.

radioType Same as the `radioType` entry in each cell record. If all the cell entries have the same `radioType`, it can be provided at the top level instead.

Bluetooth Beacon Fields

macAddress The address of the Bluetooth Low Energy (BLE) beacon.

name The name of the BLE beacon.

age The number of milliseconds since this BLE beacon was last seen.

signalStrength The measured signal strength of the BLE beacon in dBm.

Cell Tower Fields

radioType The type of radio network. One of `gsm`, `wcdma`, or `lte`. This is a custom extension to the GLS API which only defines the top-level radioType field.

mobileCountryCode The mobile country code.

mobileNetworkCode The mobile network code.

locationAreaCode The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

cellId The cell id or cell identity.

age The number of milliseconds since this networks was last detected.

psc The primary scrambling code for WCDMA and physical cell id for LTE. This is an addition to the GLS API.

signalStrength The signal strength for this cell network, either the RSSI or RSCP.

timingAdvance The timing advance value for this cell network.

WiFi Access Point Fields

Note: Hidden WiFi networks and those whose SSID (clear text name) ends with the string `_nomap` must NOT be used for privacy reasons.

It is the responsibility of the client code to filter these out.

---

macAddress The BSSID of the WiFi network.

age The number of milliseconds since this network was last detected.

channel The WiFi channel for networks in the 2.4GHz range. This often ranges from 1 to 13.

frequency The frequency in MHz of the channel over which the client is communicating with the access point. This is an addition to the GLS API and can be used instead of the channel field.

signalStrength The received signal strength (RSSI) in dBm.

signalToNoiseRatio The current signal to noise ratio measured in dB.

ssid The SSID of the Wifi network.

Wifi networks with a SSID ending in `_nomap` must not be collected.

Fallback Fields

The fallback section is a custom addition to the GLS API.

By default, both a GeoIP based position fallback and a fallback based on cell location areas (lac's) are enabled. Omit the `fallbacks` section if you want to use the defaults. Change the values to `false` if you want to disable either of the fallbacks.

lacf If no exact cell match can be found, fall back from exact cell position estimates to more coarse grained cell location area estimates rather than going directly to an even worse GeoIP based estimate.

ipf If no position can be estimated based on any of the provided data points, fall back to an estimate based on a GeoIP database based on the senders IP address at the time of the query.

Deviations From GLS API

Our API differs from the GLS API in these ways:

- The entire Bluetooth section is a custom addition–the GLS API does not have this.
- Cell entries allow you to specify the `radioType` per cell network instead of globally. This allows for queries with data from multiple active SIM cards. For example, this allows for queries where one of SIM card is on a GSM connection while the other uses a WCDMA connection.
- Cell entries take an extra `psc` field.
- The WiFi `macAddress` field takes both upper- and lower-case characters. It also tolerates `:`, `-`, or no separator and internally strips them.
- WiFi entries take an extra `frequency` field.
- The `fallbacks` section allows some control over the more coarse grained position sources. If no exact match can be found, these can be used to return a "404 Not Found" rather than a coarse grained estimate with a large accuracy value.
- If either the GeoIP or location area fallbacks were used to determine the response, an additional fallback key will be returned in the response.
- The `considerIp` field has the same purpose as the fallbacks/ipf field. It was introduced into the GLS API later on and we continue to support both, with the fallbacks section taking precedence.

Response

A successful response returns a position estimate and an accuracy field. Combined these two describe the center and radius of a circle. The user's true position should be inside the circle with a 95th percentile confidence value. The accuracy is measured in meters.

If the position is to be shown on a map and the returned accuracy is large, it may be advisable to zoom out the map, so that all of the circle can be seen, even if the circle itself is not shown graphically. That way a user should still see his true position on the map and can zoom in further.

If instead the returned position is shown highly zoomed in, the user may just see an arbitrary location that they don't recognize at all. This typically happens when GeoIP based results are returned and the returned position is the center of a city or the center of a region.

An example of a successful response:

```
{
    "location": {
        "lat": -22.7539192,
        "lng": -43.4371081
    },
    "accuracy": 100.0
}
```

An example of a successful response based on a GeoIP estimate:

```
{
    "location": {
        "lat": 51.0,
        "lng": -0.1
    },
    "accuracy": 600000.0,
    "fallback": "ipf"
}
```

Alternatively the fallback field can also state `lacf` for an estimate based on a cell location area.

If no position information could be determined, an HTTP status code 404 will be returned:

```
{
    "error": {
        "errors": [{
            "domain": "geolocation",
            "reason": "notFound",
            "message": "Not found",
        }],
        "code": 404,
        "message": "Not found",
    }
}
```

Region: /v1/country

Purpose: Determine the current region based on data provided about nearby Bluetooth, cell, or WiFi networks the IP address used to access the service.

The responses use region codes and names from the GENC dataset, which is mostly compatible with the ISO 3166 standard.

Note: While the API endpoint and JSON payload refers to *country*, no claim about the political status of any region is made by this service.

- *Request*
- *Response*

Request

Requests are submitted using an HTTP POST request to the URL:

```
https://location.services.mozilla.com/v1/country?key=<API_KEY>
```

This implements the same interface as the *Geolocate: /v1/geolocate* API.

The simplest request contains no extra information and simply relies on the IP address to provide a response.

Response

Here's an example successful response:

```
{
    "country_code": "US",
    "country_name": "United States"
}
```

Should the response be based on a GeoIP estimate:

```
{
    "country_code": "US",
    "country_name": "United States",
    "fallback": "ipf"
}
```

If no region could be determined, a HTTP status code 404 will be returned:

```
{
    "error": {
        "errors": [{
            "domain": "geolocation",
            "reason": "notFound",
            "message": "Not found",
        }],
        "code": 404,
        "message": "Not found",
    }
}
```

Geosubmit Version 2: /v2/geosubmit

Purpose: Submit data about nearby Bluetooth beacons, cell or WiFi networks.

- *Request*
- *Field Definition*
  - *Global Fields*
  - *Position Fields*

  - *Bluetooth Beacon Fields*
  - *Cell Tower Fields*
  - *Wifi Access Point Fields*
- *Response*

Request

Geosubmit requests are submitted using an HTTP POST request to the URL:

```
https://location.services.mozilla.com/v2/geosubmit?key=<API_KEY>
```

Note: Omit the optional `?key=<API_KEY>` parameter to anonymously submit data

There is an earlier *Geosubmit: /v1/geosubmit (DEPRECATED)* v1 API, with a slightly different and less extensive field list.

Geosubmit requests are submitted using an HTTP POST request with a JSON body.

Here is an example body:

```
{"items": [{
    "timestamp": 1405602028568,
    "position": {
        "latitude": -22.7539192,
        "longitude": -43.4371081,
        "accuracy": 10.0,
        "age": 1000,
        "altitude": 100.0,
        "altitudeAccuracy": 50.0,
        "heading": 45.0,
        "pressure": 1013.25,
        "speed": 3.6,
        "source": "gps"
    },
    "bluetoothBeacons": [
        {
            "macAddress": "ff:23:45:67:89:ab",
            "age": 2000,
            "name": "beacon",
            "signalStrength": -110
        }
    ],
    "cellTowers": [
        {
            "radioType": "lte",
            "mobileCountryCode": 208,
            "mobileNetworkCode": 1,
            "locationAreaCode": 2,
            "cellId": 12345,
```

(continues on next page)

(continued from previous page)

```
            "age": 3000,
            "asu": 31,
            "primaryScramblingCode": 5,
            "serving": 1,
            "signalStrength": -51,
            "timingAdvance": 1
        }
    ],
    "wifiAccessPoints": [
        {
            "macAddress": "01:23:45:67:89:ab",
            "age": 5000,
            "channel": 6,
            "frequency": 2437,
            "radioType": "802.11n",
            "signalToNoiseRatio": 13,
            "signalStrength": -77
        },
        {
            "macAddress": "23:45:67:89:ab:cd"
        }
    ]
}]}
```

Field Definition

Requests always need to contain a batch of *reports*. Each *report* must contain at least one entry in the `bluetoothBeacons` or `cellTowers` array or two entries in the `wifiAccessPoints` array.

Almost all of the fields are optional. For Bluetooth and WiFi records the `macAddress` field is required.

Global Fields

timestamp The time of observation of the data, measured in milliseconds since the UNIX epoch. Can be omitted if the observation time is very recent. The age values in each section are relative to this timestamp.

Position Fields

The position block contains information about where and when the data was observed.

latitude The latitude of the observation (WSG 84).

longitude The longitude of the observation (WSG 84).

accuracy The accuracy of the observed position in meters.

altitude The altitude at which the data was observed in meters above sea-level.

altitudeAccuracy The accuracy of the altitude estimate in meters.

age The age of the position data (in milliseconds).

heading The heading field denotes the direction of travel of the device and is specified in degrees, where 0° heading < 360°, counting clockwise relative to the true north.

pressure The air pressure in hPa (millibar).

speed The speed field denotes the magnitude of the horizontal component of the device's current velocity and is specified in meters per second.

source The source of the position information. If the field is omitted, "gps" is assumed. The term `gps` is used to cover all types of satellite based positioning systems including Galileo and Glonass. Other possible values are `manual` for a position entered manually into the system and `fused` for a position obtained from a combination of other sensors or outside service queries.

Bluetooth Beacon Fields

macAddress The address of the Bluetooth Low Energy (BLE) beacon.

name The name of the BLE beacon.

age The number of milliseconds since this BLE beacon was last seen.

signalStrength The measured signal strength of the BLE beacon in dBm.

Cell Tower Fields

radioType The type of radio network; one of `gsm`, `wcdma` or `lte`.

mobileCountryCode The mobile country code.

mobileNetworkCode The mobile network code.

locationAreaCode The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

cellId The cell id or cell identity.

age The number of milliseconds since this cell was last seen.

asu The arbitrary strength unit indicating the signal strength if a direct signal strength reading is not available.

primaryScramblingCode The primary scrambling code for WCDMA and physical cell id for LTE.

serving A value of `1` indicates this as the serving cell, a value of `0` indicates a neighboring cell.

signalStrength The signal strength for this cell network, either the RSSI or RSCP.

timingAdvance The timing advance value for this cell tower when available.

Wifi Access Point Fields

macAddress The BSSID of the Wifi network.

Hidden Wifi networks must not be collected.

radioType The Wifi radio type; one of `802.11a`, `802.11b`, `802.11g`, `802.11n`, `802.11ac`.

age The number of milliseconds since this Wifi network was detected.

channel The channel is a number specified by the IEEE which represents a small band of frequencies.

frequency The frequency in MHz of the channel over which the client is communicating with the access point.

signalStrength The received signal strength (RSSI) in dBm.

signalToNoiseRatio The current signal to noise ratio measured in dB.

ssid The SSID of the Wifi network.

Wifi networks with a SSID ending in `_nomap` must not be collected.

Response

Successful requests return a HTTP 200 response with a body of an empty JSON object.

Geosubmit: /v1/geosubmit (DEPRECATED)

Deprecated since version 1.2: (2015-07-15) Please use the *Geosubmit Version 2: /v2/geosubmit* API instead.

Purpose: Submit data about nearby Bluetooth, cell and WiFi networks.

- *Request*
- *Field Definition*
- *Response*

Request

Geosubmit requests are submitted using an HTTP POST request to the URL:

```
https://location.services.mozilla.com/v1/geosubmit?key=<API_KEY>
```

Geosubmit requests are submitted using an HTTP POST request with a JSON body.

Here is an example body:

```
{"items": [
   {
    "latitude": -22.7539192,
    "longitude": -43.4371081,
    "accuracy": 10.0,
    "altitude": 100.0,
    "altitudeAccuracy": 50.0,
    "timestamp": 1405602028568,
    "heading": 45.0,
    "speed": 3.6,
    "bluetoothBeacons": [
        {
            "macAddress": "ff:74:27:89:5a:77",
            "age": 2000,
            "name": "beacon",
            "signalStrength": -110
        }
    ],
    "cellTowers": [
        {
```

(continues on next page)

(continued from previous page)

```
            "radioType": "gsm",
            "cellId": 12345,
            "locationAreaCode": 2,
            "mobileCountryCode": 208,
            "mobileNetworkCode": 1,
            "age": 3,
            "asu": 31,
            "signalStrength": -51,
            "timingAdvance": 1
        }
    ],
    "wifiAccessPoints": [
        {
            "macAddress": "01:23:45:67:89:ab",
            "age": 3,
            "channel": 6,
            "frequency": 2437,
            "signalToNoiseRatio": 13,
            "signalStrength": -77
        },
        {
            "macAddress": "23:45:67:89:ab:cd"
        }
    ]
   }
]}
```

Field Definition

Requests always need to contain a batch of *reports*. Each *report* must contain at least one entry in the *cellTowers* array or two entries in the *wifiAccessPoints* array.

Most of the fields are optional. For Bluetooth and WiFi records only the *macAddress* field is required. For cell records the *radioType*, *mobileCountryCode*, *mobileNetworkCode*, *locationAreaCode* and *cellId* fields are required.

The Bluetooth array is an extension and can contain four different fields for each Bluetooth network:

macAddress The address of the Bluetooth Low Energy (BLE) beacon.

name The name of the BLE beacon.

age The number of milliseconds since this Bluetooth beacon was last seen.

signalStrength The measured signal strength of the BLE beacon in dBm.

The cell record has been extended over the geolocate schema to include three more optional fields:

age The number of milliseconds since this cell was primary. If age is 0, the cell id represents a current observation.

asu The arbitrary strength unit. An integer in the range of 0 to 95 (optional).

psc The physical cell id as an integer in the range of 0 to 503 (optional).

The WiFi record has been extended with one extra optional field *frequency*. Either *frequency* or *channel* may be submitted to the geosubmit API as they are functionally equivalent.

frequency The frequency in MHz of the channel over which the client is communicating with the access point.

The top level schema is identical to the geolocate schema with the following additional fields:

latitude The latitude of the observation (WSG 84).

longitude The longitude of the observation (WSG 84).

timestamp The time of observation of the data, measured in milliseconds since the UNIX epoch. Should be omitted if the observation time is very recent.

accuracy The accuracy of the observed position in meters.

altitude The altitude at which the data was observed in meters above sea-level.

altitudeAccuracy The accuracy of the altitude estimate in meters.

heading The heading field denotes the direction of travel of the device and is specified in degrees, where 0° heading < 360°, counting clockwise relative to the true north. If the device cannot provide heading information or the device is stationary, the field should be omitted.

speed The speed field denotes the magnitude of the horizontal component of the device's current velocity and is specified in meters per second. If the device cannot provide speed information, the field should be omitted.

Response

Successful requests return a HTTP 200 response with a body of an empty JSON object.

Submit: /v1/submit (DEPRECATED)

Deprecated since version 1.2: (2015-07-15) Please use the *Geosubmit Version 2: /v2/geosubmit* API instead.

Purpose: Submit data about nearby cell and WiFi networks.

- *Request*
- *Field Definition*
  - *Bluetooth Fields*
  - *Cell Fields*
  - *WiFi Fields*
- *Response*

Request

Submit requests are submitted using an HTTP POST request to one of:

```
https://location.services.mozilla.com/v1/submit
https://location.services.mozilla.com/v1/submit?key=<API_KEY>
```

with a JSON body containing a position report.

Here is an example position report:

```
{"items": [
   {
    "lat": -22.7539192,
    "lon": -43.4371081,
    "time": "2012-03-01T00:00:00.000Z",
    "accuracy": 10.0,
    "altitude": 100.0,
    "altitude_accuracy": 1.0,
    "heading": 45.0,
    "speed": 13.88,
    "radio": "gsm",
    "blue": [
        {
            "key": "ff:74:27:89:5a:77",
            "age": 2000,
            "name": "beacon",
            "signal": -110
        }
    ],
    "cell": [
        {
            "radio": "umts",
            "mcc": 123,
            "mnc": 123,
            "lac": 12345,
            "cid": 12345,
            "age": 3000,
            "signal": -60
        }
    ],
    "wifi": [
        {
            "key": "01:23:45:67:89:ab",
            "age": 2000,
            "channel": 11,
            "frequency": 2412,
            "signal": -51
        }
    ]
   }
   ]
}
```

Field Definition

The only required fields are `lat` and `lon` and at least one Bluetooth, cell, or WiFi entry. If neither `lat` nor `lon` are included, the record will be discarded.

The `altitude`, `accuracy`, and `altitude_accuracy` fields are all measured in meters. Altitude measures the height above or below the mean sea level, as defined by WGS84.

The `heading` field specifies the direction of travel in 0 <= heading <= 360 degrees, counting clockwise relative to the true north.

The `speed` field specifies the current horizontal velocity and is measured in meters per second.

The `heading` and `speed` fields should be omitted from the report, if the speed and heading cannot be determined or the device was stationary while observing the environment.

The `time` has to be in UTC time, encoded in ISO 8601. If not provided, the server time will be used.

Bluetooth Fields

For `blue` entries, the `key` field is required.

key (required) The `key` is the mac address of the Bluetooth network. For example, a valid key would look similar to `ff:23:45:67:89:ab`.

age The number of milliseconds since this BLE beacon was last seen.

signal The received signal strength (RSSI) in dBm, typically in the range of -10 to -127.

name The name of the Bluetooth network.

Cell Fields

radio The type of radio network. One of `gsm`, `umts` or `lte`.

mcc The mobile country code.

mnc The mobile network code.

lac The location area code for GSM and WCDMA networks. The tracking area code for LTE networks.

cid The cell id or cell identity.

age The number of milliseconds since this networks was last detected.

psc The primary scrambling code for WCDMA and physical cell id for LTE.

signal The signal strength for this cell network, either the RSSI or RSCP.

ta The timing advance value for this cell network.

WiFi Fields

For `wifi` entries, the `key` field is required. The client must check the Wifi SSID for a `_nomap` suffix. Wifi networks with this suffix must not be submitted to the server.

Most devices will only report the WiFi frequency or the WiFi channel, but not both. The service will accept both if they are provided, but you can include only one or omit both fields.

key (required) The `key` is the BSSID of the WiFi network. So for example a valid key would look similar to `01:23:45:67:89:ab`.

The client must check the WiFi SSID for a `_nomap` suffix. WiFi networks with this suffix must not be submitted to the server.

WiFi networks with a hidden SSID should not be submitted to the server either.

age The number of milliseconds since this network was last detected.

frequency The frequency in MHz of the channel over which the client is communicating with the access point.

channel The channel is a number specified by the IEEE which represents a small band of frequencies.

signal The received signal strength (RSSI) in dBm, typically in the range of -51 to -113.

signalToNoiseRatio The current signal to noise ratio measured in dB.

ssid The SSID of the Wifi network. Wifi networks with a SSID ending in `_nomap` must not be collected.

Here's an example of a valid WiFi record:

```
{
    "key": "01:23:45:67:89:ab",
    "age": 1500,
    "channel": 11,
    "frequency": 2412,
    "signal": -51,
    "signalToNoiseRatio": 37
}
```

Response

On successful submission, you will get a 204 status code back without any data in the body.

History

The service launched in 2013, and first offered custom `/v1/search/` and */v1/submit/* APIs, used by the MozStumbler app. Later that year the */v1/geolocate* API was implemented, to reduce the burden on clients that already used the Google Maps Geolocation API. This was followed by the */v1/geosubmit* API, to make contribution more consistant.

In 2014, the `/v1/geosubmit` and `/v1/geolocate` API were recommended for client development, and the `/v1/submit` and `/v1/search` APIs were marked as deprecated.

In 2015, the */v2/geosubmit* API was added, to expand the submitted data fields and accept data from partners. The */v1/country* API was also added, to provide region rather than position lookups.

In 2016, a `/v1/transfer` API was added, for bulk transfers of data from one instance of Ichnaea to another. This was also when work started on the 2.0 implementation of Ichnaea, and MozStumbler switched to `/v1/geolocate` and `/v1/geosubmit`.

In 2017, the `/v1/transfer` API was removed from the 2.0 branch, and Mozilla stopped active development of Ichnaea, leaving 1.5 as the production deployment.

In 2019, Ichnaea development started up again, to prepare the 2.0 codebase for production. The deprecated `/v1/search` API was removed.

1.1.2 Applications / Libraries

A number of applications and libraries with different capabilities exist which allow you to interact with the service. Please check the Software listing on the Mozilla Wiki to learn more.

1.2 Development/Deployment documentation

This section covers information for people who are developing or deploying Ichnaea.

1.2.1 Local development environment

This chapter covers getting started with Ichnaea using Docker for a local development environment.

Contents

- *Local development environment*
  - *Setup Quickstart*
  - *Updating the Dev Environment*
    - *Updating code*
  - *Specifying configuration*
    - *Setting configuration specific to your local dev environment*
    - *Overriding configuration*
  - *Alembic and Database Migrations*
  - *Building Static Assets (CSS/JS)*
  - *Running Tests*
  - *Building Docs*
  - *Updating Test GeoIP Data and Libraries*
  - *Building Datamap Tiles*
  - *Running Tasks*

Setup Quickstart

1. Install required software: Docker, docker-compose (1.27+), make, and git.

   Linux:

   Use your package manager.

   macOS:

   Install Docker for Mac which will install Docker and docker-compose.

   Use homebrew to install make and git:

   ```
   $ brew install make git
   ```

   Other:

   Install Docker.

   Install docker-compose. You need 1.27.4 or higher.

   Install make.

   Install git.

2. Clone the repository so you have a copy on your host machine.

   Instructions for cloning are on the Ichnaea page in GitHub.

3. Set environment options.

   To create the environment options file `my.env`

   ```
   $ make my.env
   ```

   If you're on **Linux**, you will need to set the UID/GID of the app user that runs in the Docker containers to match your UID/GID. Run `id` to get your UID/GID. Edit `my.env` and set the `ICHNAEA_UID` and `ICHNAEA_GID` variables. These will get used when creating the app user in the base image.

   If you are using **macOS** on a computer with Apple Silicon (such as M1 Macs released in 2020 or later), you'll need to select a different database engine, since Docker images are not available for the arm64 platform. Edit `my.env` and set `ICHNAEA_DOCKER_DB_ENGINE` to `mariadb_10_5`. This will be used when creating and running the database.

   If you ever want different values, change them in `my.env` and re-run the setup steps below (`make setup`, `make runservices`, etc.).

4. Build Docker images for Ichnaea services.

   From the root of this repository, run:

   ```
   $ make build
   ```

   That will build the app Docker image required for development.

5. Initialize Redis and MySQL.

   Then you need to set up services. To do that, run:

   ```
   $ make runservices
   ```

   This starts service containers. Then run:

```
# docker-compose ps
```

If the state of the database container is `Up (health: starting)`, wait a minute and try `docker-compose ps`. When the state is `Up (healthy)`, then the database is initialized and ready for connections.

If the state is just `Up`, then the container doesn't provide health checks. Wait a couple of minutes before trying the next step.

Then run:

```
$ make setup
```

This creates the MySQL database and sets up tables and things.

You can run `make setup` any time you want to wipe any data and start fresh.

At this point, you should have a basic functional Ichnaea development environment that has no geo data in it.

To see what else you can do, run:

```
$ make
```

Updating the Dev Environment

Updating code

Any time you want to update the code in the repostory, run something like this from the main branch:

```
$ git pull
```

The actual command depends on what you're working on and the state of your copy of the repository.

After you have the latest code, you'll need to update other things.

If there were changes to the requirements files or setup scripts, you'll need to build new images:

```
$ make build
```

If there were changes to the database tables, you'll need to wipe the MySQL database and Redis:

```
$ make setup
```

Specifying configuration

Configuration is pulled from these sources:

1. The `my.env` file.
2. ENV files located in `/app/docker/config/`. See `docker-compose.yml` for which ENV files are used in which containers, and their precedence.
3. Configuration defaults defined in the code.

The sources above are ordered by precedence, i.e. configuration values defined in the `my.env` file will override values in the ENV files or defaults.

The following ENV files can be found in `/app/docker/config/`:

local_dev.env This holds *secrets* and *environment-specific configuration* required to get services to work in a Docker-based local development environment.

This should **NOT** be used for server environments, but you could base configuration for a server environment on this file.

test.env This holds configuration specific to running the tests. It has some configuration value overrides because the tests are "interesting".

my.env This file lets you override any environment variables set in other ENV files as well as set variables that are specific to your instance.

It is your personal file for your specific development environment–it doesn't get checked into version control.

The template for this is in `docker/config/my.env.dist`.

In this way:

1. environmental configuration which covers secrets, hosts, ports, and infrastructure-specific things can be set up for every environment
2. behavioral configuration which covers how the code behaves and which classes it uses is versioned alongside the code making it easy to deploy and revert behavioral changes with the code depending on them
3. `my.env` lets you set configuration specific to your development environment as well as override any configuration and is not checked into version control

See also:

See *Configuration* for configuration settings.

Setting configuration specific to your local dev environment

There are some variables you need to set that are specific to your local dev environment. Put them in `my.env`.

Overriding configuration

If you want to override configuration temporarily for your local development environment, put it in `my.env`.

Alembic and Database Migrations

Ichnaea uses Alembic.

To create a new database migration, do this:

```
$ make shell
app@blahblahblah:/app$ alembic revision -m "SHORT DESCRIPTION"
```

Then you can edit the file.

Building Static Assets (CSS/JS)

To build changed assets:

```
$ make assets
```

To rebuild asset files from scratch:

```
$ make clean-assets assets
```

To recreate the node container, applying changes in `package.json`:

```
$ make build clean-assets assets
```

Running Tests

You can run the test suite like this:

```
$ make test
```

If you want to pass different arguments to pytest or specify specific tests to run, open up a test shell first:

```
$ make testshell
app@blahblahblah:/app$ pytest [ARGS]
```

Building Docs

You can build the docs like this:

```
$ make docs
```

This will create an application container with a volume mount to the local `docs/build/html` directory and update the documentation so it is available in that local directory.

To view the documentation open `file://docs/build/html/index.html` with a web brower.

Updating Test GeoIP Data and Libraries

The development environment uses a test MaxMind GeoIP database, and the Ichnaea test suite will fail if this is more than 1000 days old. To update this database and confirm tests pass, run:

```
$ make update-vendored test
```

Commit the refreshed files.

This command can also be used to updated `libmaxmindb` and the `datamaps` source. Update `docker.make` for the desired versions, and run:

```
$ make update-vendored build test
```

Commit the updated source tarballs.

Building Datamap Tiles

To build datamap tiles for the local development environment, run:

```
$ make local-map
```

If you have data in the `datamap` tables, this will create many files under `ichnaea/content/static/datamap`. This uses `ichnaea/scripts/datamap.py`, which can also be run directly.

To see the map locally, you will need to configure *Mapbox*. A free developer account should be sufficient.

To use an S3 bucket for tiles, you'll need to set `ASSET_BUCKET` and `ASSET_URL` (see *Map tile and download assets*). To upload tiles to an S3 bucket, you'll also need AWS credentials that can read, write, and delete objects in the `ASSET_BUCKET`. Here are two ways, neither of which is ideal since it adds your AWS credentials in plain text:

1. Add credentials as environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` in `my.env`.
2. Add credentials to a file `my.awscreds` in the project folder, and add `AWS_SHARED_CREDENTIALS_FILE=/app/my.awscreds` to `my.env`.

You can then generate and upload tiles with:

```
$ docker-compose run --rm app map
```

This will generate a fresh set of tiles in a temporary directory and sync the S3 bucket with the changes.

Running Tasks

To run worker tasks in the development environment, run:

```
$ make runcelery
```

This will run the `scheduler`, which will schedule periodic tasks, as well as the `worker`, which runs the tasks. If you see this error:

```
scheduler_1  | ERROR: Pidfile (/var/run/location/celerybeat.pid) already exists.
```

then stop the `make runcelery` process (Ctrl-C) and re-create the `scheduler`:

```
$ docker rm -f scheduler
$ make runcelery
```

To manually run a task, call it from a shell:

```
$ make shell
$ celery -A ichnaea.taskapp.app:celery_app call ichnaea.data.tasks.update_statregion
```

This will add the task `update_statregion` to the Redis queue. The `worker` task will read the queue and execute it.

The commands will also run if you start a shell with `make testshell`, but the task will not execute. A different Redis URI is setup for the test environment, and the worker running with `make runcelery` will not read that Redis queue, and will not see the request.

There are other commands available, such as this one to display registered tasks:

```
$ celery -A ichnaea.taskapp.app:celery_app inspect registered
```

1.2.2 Python Requirements

`requirements.in` files contain the first-order requirements. It is split into requirements required for production, and those used in development. If you want to change requirements, change this file.

`requirements.txt` includes the exact versions and package hashes for the first-order requirements as well as the requirements-of-requirements. This is the file used with `pip install` to install packages in the Docker image. `pip-compile`, from pip-tools, generates and maintains this file using `requirements.in`.

`requirements-docs.txt` has the requirements used on ReadTheDocs to build the Ichnaea documentation on each merge to main. Since this is a non-production environment, we neither pin nor hash the requirements.

Compiling requirements.txt

After making changes, the `.in` files can be compiled to `.txt` output files by running:

```
make update-reqs
```

This will start a Docker container and run `pip-compile` with the proper options. Running in the container ensures that the correct dependencies are chosen for the Docker environment, rather than your host environment.

There will be a warnings at the end of the process:

> The generated requirements file may be rejected by pip install. See # WARNING lines for details.

This is expected. `pip` and `setuptools` are provided by the container, and should not be pinned.

To apply the new requirements, rebuild your Docker image:

```
make build
```

Automated Updates

Dependabot opens PRs for updates around the first of the month. It also opens PRs for security updates when they are available. It seems to have some support for `pip-tools`, but it may be necessary to manually run `make update-reqs` to correctly regenerate the requirements.

paul-mclendahand is useful for packaging several PRs into a single PR, and avoiding the rebase / rebuild / test cycle when merging one Dependabot PR at a time.

Manually upgrading requirements.txt

To upgrade all the requirements, run `make shell` to enter the Docker environment, and run

```
CUSTOM_COMPILE_COMMAND="make update-reqs" pip-compile --generate-hashes --upgrade
```

To upgrade a single package, run this instead:

```
CUSTOM_COMPILE_COMMAND="make update-reqs" pip-compile --generate-hashes --upgrade-
↪package <package-name>
```

You'll need to exit the Docker environment and run `make build` to recreate the Docker image with your changes.

Other tools

pipdeptree displays the requirements tree, which can be useful to determine which package required an unknown package.

1.2.3 Debugging

MySQL Config

First, let's check the database configuration.

In a local development environment, you can run the mysql client like this:

```
make mysql
```

In a server environment, use the mysql client to connect to the database.

Next, check if alembic migrations have been run:

```
select * from alembic_version;
+--------------+
| version_num  |
+--------------+
| d2d9ecb12edc |
+--------------+
1 row in set (0.00 sec)
```

This needs to produce a single row and some `version_num` in it. If it isn't there, check the *Database Setup* part of *the deploy docs*.

Now check the API keys:

```
select * from api_key\G
*************************** 1. row ***************************
                  valid_key: test
                     maxreq: NULL
             allow_fallback: 0
               allow_locate: 1
               allow_region: 1
             allow_transfer: 0
              fallback_name: NULL
               fallback_url: NULL
         fallback_ratelimit: NULL
fallback_ratelimit_interval: NULL
      fallback_cache_expire: NULL
        store_sample_locate: 100
        store_sample_submit: 100
1 row in set (0.00 sec)
```

And the export config:

```
select * from export_config;
+----------+-------+----------+------+-----------+---------------+
| name     | batch | schema   | url  | skip_keys | skip_sources |
```

(continues on next page)

(continued from previous page)

```
+----------+-------+----------+------+-----------+--------------+
| internal |   100 | internal | NULL | NULL      | NULL         |
+----------+-------+----------+------+-----------+--------------+
1 row in set (0.00 sec)
```

If you are missing either of these entries, then it's likely you need to set up API keys and export configuration.

Connections

Check if the web service and celery containers can connect to the MySQL database and Redis datastore.

Follow the instructions in the *Runtime Checks* part of the *the deploy docs*. Make sure to call the `/__heartbeat__` HTTP endpoint on the web application.

Another way to check connections is to start a container and try to connect to the two external connections from inside it.

In a local development environment, you can do this:

```
make shell
```

In a server environment, you need to run the container with configuration in the environment.

Once inside the container, you can do this:

```
$ redis-cli -u $REDIS_URI
172.17.0.2:6379> keys *
1) "_kombu.binding.celery"
2) "unacked_mutex"
3) "_kombu.binding.celery.pidbox"
```

If the task worker containers are running or have been run at least once, you should see keys listed.

Similarly, we can connect to the MySQL database from inside the container. Using the same shell, you can run the mysql client:

```
$ mysql -h DBHOST -uUSERNAME --password=PASSWORD DBNAME
...
Welcome to the MySQL monitor.  Commands end with ; or \g.
...
mysql>
```

Substitute `DBHOST`, `USERNAME`, `PASSWORD`, and `DBNAME` according to your database setup.

Task Worker

The asynchronous task worker uses a Python framework called Celery. You can use the Celery monitoring guide for more detailed information.

A basic test is to call the `inspect stats` commands. Open a shell container and inside it run:

```
$ celery -A ichnaea.taskapp.app:celery_app inspect stats
-> celery@388ec81273ba: OK
{
```

(continues on next page)

(continued from previous page)

```
    ...
    "total": {
        "ichnaea.data.tasks.monitor_api_key_limits": 1,
        "ichnaea.data.tasks.monitor_api_users": 1,
        "ichnaea.data.tasks.update_blue": 304,
        "ichnaea.data.tasks.update_cell": 66,
        "ichnaea.data.tasks.update_cellarea": 21,
        "ichnaea.data.tasks.update_incoming": 29,
        "ichnaea.data.tasks.update_wifi": 368
    }
}
```

If you get `Error: no nodes replied within time constraint.`, then Celery isn't running.

If this section continues to be empty, something is wrong with the scheduler and it isn't adding tasks to the worker queues.

Otherwise, the output is pretty long. Look at the "total" section. If you have your worker and scheduler container running for some minutes, this section should fill up with various tasks.

Data Pipeline

Now that all the building blocks are in place, let's try to send real data to the service and see how it processes it.

Assuming containers for all three roles are running, we'll use the HTTP geosubmit v2 API endpoint to send some new data to the service:

```
$ curl -H 'Content-Type: application/json' http://127.0.0.1:8000/v2/geosubmit?key=test -
↪d \
'{"items": [{"wifiAccessPoints": [{"macAddress": "94B40F010D01"}, {"macAddress":
↪"94B40F010D00"}, {"macAddress": "94B40F010D03"}], "position": {"latitude": 51.0,
↪"longitude": 10.0}}]}'
```

We can find this data again in Redis, open a Redis client and do:

```
lrange "queue_export_internal" 0 10
1) "{\"api_key\": \"test\", \"report\": {\"timestamp\": 1499267286717, \
↪"bluetoothBeacons\": [], \"wifiAccessPoints\": [{\"macAddress\": \"94B40F010D01\"}, {\
↪"macAddress\": \"94B40F010D00\"}, {\"macAddress\": \"94B40F010D03\"}], \"cellTowers\":␣
↪[], \"position\": {\"latitude\": 51.0, \"longitude\": 10.0}}}"
```

The data pipeline is optimized for production use and processes data in batches or if data sits too long in a queue. We can use the later feature to trick the pipeline into processing data sooner.

In the same Redis client use:

```
expire "queue_export_internal" 300
```

This tells the queue to get deleted in 300 seconds. The scheduler runs a task to check this queue about once per minute and checks both its length and its remaining time-to-live.

If we check the available Redis keys again, we might see something like:

```
keys *
1) "_kombu.binding.celery"
2) "apiuser:submit:test:2017-07-05"
3) "update_wifi_0"
4) "unacked_mutex"
5) "statcounter_unique_wifi_20170705"
6) "_kombu.binding.celery.pidbox"
```

If we wait a bit longer, the `update_wifi_0` entry should vanish.

Once that happened, we can check the database directly. On a MySQL client prompt do:

```
select hex(`mac`), lat, lon from wifi_shard_0;
+--------------+------+------+
| hex(`mac`)   | lat  | lon  |
+--------------+------+------+
| 94B40F010D00 |   51 |   10 |
| 94B40F010D01 |   51 |   10 |
| 94B40F010D03 |   51 |   10 |
+--------------+------+------+
3 rows in set (0.00 sec)
```

Once the data has been processed, we can try the public HTTP API again and see if we can locate us. To do that we can use both the geolocate and region APIs:

```
curl -H 'Content-Type: application/json' http://127.0.0.1:8000/v1/geolocate?key=test -d \
'{"wifiAccessPoints": [{"macAddress": "94B40F010D01"}, {"macAddress": "94B40F010D00"}, {
↪"macAddress": "94B40F010D03"}]}'
```

This should produce a response:

```
{"location": {"lat": 51.0, "lng": 10.0}, "accuracy": 10.0}
```

And again using the region API:

```
curl -H 'Content-Type: application/json' http://127.0.0.1:8000/v1/country?key=test -d \
'{"wifiAccessPoints": [{"macAddress": "94B40F010D01"}, {"macAddress": "94B40F010D00"}, {
↪"macAddress": "94B40F010D03"}]}'
```

```
{"country_code": "DE", "country_name": "Germany"}
```

If you check Redis queues again, there's a new entry in there for the geolocate query we just submitted:

```
172.17.0.2:6379> lrange "queue_export_internal" 0 10
1) "{\"api_key\": \"test\", \"report\": {\"wifiAccessPoints\": [{\"macAddress\": \
↪"94b40f010d01\"}, {\"macAddress\": \"94b40f010d00\"}, {\"macAddress\": \"94b40f010d03\
↪"}], \"fallbacks\": {\"ipf\": true, \"lacf\": true}, \"position\": {\"latitude\": 51.0,
↪ \"longitude\": 10.0, \"accuracy\": 10.0, \"source\": \"query\"}}}"
```

Note the `"source": "query"` part at the end, which tells the pipeline the position data does not represent a GPS verified position, but was the result of a query.

You can use the same `expire` trick as above again, to get the data processed faster.

In the mysql client, you can see the result:

```
select hex(`mac`), last_seen from wifi_shard_0;
+--------------+------------+
| hex(`mac`)   | last_seen  |
+--------------+------------+
| 94B40F010D00 | 2017-07-05 |
| 94B40F010D01 | 2017-07-05 |
| 94B40F010D03 | 2017-07-05 |
+--------------+------------+
3 rows in set (0.00 sec)
```

Since all the WiFi networks were already known, their position just got confirmed. This gets stored in the `last_seen` column, which tracks when the network was last confirmed in a query.

1.2.4 Configuration

The application takes a number of different settings and reads them from environment variables. There are also a small number of settings inside database tables.

- *Environment variables*
  - *Database*
  - *GeoIP*
  - *Redis*
  - *Sentry*
  - *StatsD*
  - *Map tile and download assets*
  - *Mapbox*
- *Configuration in the database*
  - *API Keys*
    - *Fallback*
  - *Export Configuration*
    - *S3 Bucket Export (s3)*
    - *Internal Export (internal)*
    - *HTTP Export (geosubmit)*

Environment variables

component ichnaea.conf.**AppComponent**
Configuration summary:

Setting	Parser	Required?
LOCAL_DEV_ENV	*bool*	
TESTING	*bool*	
LOGGING_LEVEL	*ichnaea.conf.logging_level_parser*	
ASSET_BUCKET	*str*	
ASSET_URL	*str*	
DB_READONLY_URI	*str*	Yes
DB_READWRITE_URI	*str*	Yes
SENTRY_DSN	*str*	
SENTRY_ENVIRONMENT	*str*	
STATSD_HOST	*str*	
STATSD_PORT	*int*	
REDIS_URI	*str*	Yes
CELERY_WORKER_CONCURRENCY	*int*	Yes
MAPBOX_TOKEN	*str*	
GEOIP_PATH	*str*	
SECRET_KEY	*str*	

Configuration options:

LOCAL_DEV_ENV

Parser *bool*

Default "false"

Required No

Whether we are (True) or are not (False) in a local dev environment. There are some things that get configured one way in a developer's environment and another way in a server environment.

TESTING

Parser *bool*

Default "false"

Required No

Whether or not we are running tests.

LOGGING_LEVEL

Parser *ichnaea.conf.logging_level_parser*

Default "INFO"

Required No

Logging level to use. One of CRITICAL, ERROR, WARNING, INFO, or DEBUG.

ASSET_BUCKET

Parser *str*

Default “”

Required No

name of AWS S3 bucket to store map tile image assets and export downloads

ASSET_URL

Parser *str*

Default “”

Required No

url for map tile image assets and export downloads

DB_READONLY_URI

Parser *str*

Required Yes

uri for the readonly database; `mysql+pymysql://USER:PASSWORD@HOST:PORT/NAME`

DB_READWRITE_URI

Parser *str*

Required Yes

uri for the read-write database; `mysql+pymysql://USER:PASSWORD@HOST:PORT/NAME`

SENTRY_DSN

Parser *str*

Default “”

Required No

Sentry DSN; leave blank to disable Sentry error reporting

SENTRY_ENVIRONMENT

Parser *str*

Default “”

Required No

Sentry environment

STATSD_HOST

Parser *str*

Default “”

Required No

StatsD host; blank to disable StatsD

STATSD_PORT

Parser *int*

Default "8125"

Required No

StatsD port

REDIS_URI

Parser *str*

Required Yes

uri for Redis; `redis://HOST:PORT/DB`

CELERY_WORKER_CONCURRENCY

Parser *int*

Required Yes

the number of concurrent Celery worker processes executing tasks

MAPBOX_TOKEN

Parser *str*

Default ""

Required No

Mapbox API key; if you do not provide this, then parts of the site showing maps will be disabled

GEOIP_PATH

Parser *str*

Default "/home/docs/checkouts/readthedocs.org/user_builds/ichnaea/checkouts/latest/ichnaea/tests/data/GeoIP2-City-Test.mmdb"

Required No

absolute path to mmdb file for GeoIP lookups

SECRET_KEY

Parser *str*

Default "default for development, change in production"

Required No

a unique passphrase used for cryptographic signing

Alembic requires an additional item in the environment:

```
# URI for user with ddl access
SQLALCHEMY_URL=mysql+pymysql://USER:PASSWORD@HOST:PORT/DBNAME
```

The webapp uses gunicorn which also has configuration.

```
# Port for gunicorn to listen on
GUNICORN_PORT=${GUNICORN_PORT:-"8000"}

# Number of gunicorn workers to spin off--should be one per cpu
GUNICORN_WORKERS=${GUNICORN_WORKERS:-"1"}

# Gunicorn worker class--use our gevent worker
GUNICORN_WORKER_CLASS=${GUNICORN_WORKER_CLASS:-"ichnaea.webapp.worker.
↪LocationGeventWorker"}

# Number of simultaneous greenlets per worker
GUNICORN_WORKER_CONNECTIONS=${GUNICORN_WORKER_CONNECTIONS:-"4"}

# Number of requests to handle before retiring worker
GUNICORN_MAX_REQUESTS=${GUNICORN_MAX_REQUESTS:-"10000"}

# Jitter to add/subtract from number of requests to prevent stampede
# of retiring
GUNICORN_MAX_REQUESTS_JITTER=${GUNICORN_MAX_REQUESTS_JITTER:-"1000"}

# Timeout for handling a request
GUNICORN_TIMEOUT=${GUNICORN_TIMEOUT:-"60"}

# Python log level for gunicorn logging output: debug, info, warning,
# error, critical
GUNICORN_LOGLEVEL=${GUNICORN_LOGLEVEL:-"info"}
```

Database

The MySQL compatible database is used for storing configuration and application data.

The webapp service requires a read-only connection.

The celery worker service requires a read-write connection.

Both of them can be restricted to only DML (data-manipulation) permissions as neither need DDL (data-definition) rights.

DDL changes are done using the alembic database migration system.

GeoIP

The web and worker roles need access to a maxmind GeoIP City database in version 2 format. Both GeoLite and commercial databases will work.

Redis

The Redis cache is used as a:

- classic cache by the web role
- backend to store rate-limiting counters
- custom and a worker queuing backend

Sentry

All roles and command line scripts use an optional Sentry server to log application exception data. Set this to a Sentry DSN to enable Sentry or `''` to disable it.

StatsD

All roles and command line scripts use an optional StatsD service to log application specific metrics. The StatsD service needs to support metric tags.

The project uses a lot of metrics as further detailed in *the metrics documentation*.

All metrics are prefixed with a *location* namespace.

Map tile and download assets

The application can optionally generate image tiles for a data map and public export files available via the downloads section of the website.

These assets are stored in a static file repository (Amazon S3) and made available via a HTTPS frontend (Amazon CloudFront).

Set `ASSET_BUCKET` and `ASSET_URL` accordingly.

To access the `ASSET_BUCKET`, authorized AWS credentials are needed inside the Docker image. See the Boto3 credentials documentation for details.

The development environment defaults to serving map tiles from the web server, and not serving public export files for download.

Mapbox

The web site content uses Mapbox to display a world map. In order to do this, it requires a Mapbox API token. Without a token, the map is not displayed.

You can create an account on their site: https://www.mapbox.com

After you have an account, you can create an API token at: https://account.mapbox.com

Set the `MAP_TOKEN` configuration value to your API token.

Configuration in the database

API Keys

The project requires API keys to access the locate APIs.

API keys can be any string of up to 40 characters, though random UUID4s in hex representation are commonly used, for example `329694ac-a337-4856-af30-66162bc8187a`.

Fallback

You can also enable a fallback location provider on a per API key basis. This allows you to send queries from this API key to an external service if Ichnaea can't provide a good-enough result.

In order to configure this fallback mode, you need to set the `fallback_*` columns. For example:

```
fallback_name: mozilla
fallback_schema: ichnaea/v1
fallback_url: https://location.services.mozilla.com/v1/geolocate?key=some_key
fallback_ratelimit: 10
fallback_ratelimit_interval: 60
fallback_cache_expire: 86400
```

The name can be shared between multiple API keys and acts as a partition key for the cache and rate limit tracking.

The schema can be one of `NULL`, `ichnaea/v1`, `combain/v1`, `googlemaps/v1` or `unwiredlabs/v1`.

`NULL` and `ichnaea/v1` are currently synonymous. Setting the schema to one of those means the external service uses the same API as the geolocate v1 API used in Ichnaea.

If you set the url to one of the unwiredlabs endpoints, add your API token as an anchor fragment to the end of it, so instead of:

```
https://us1.unwiredlabs.com/v2/process.php
```

you would instead use:

```
https://us1.unwiredlabs.com/v2/process.php#my_secret_token
```

The code will read the token from here and put it into the request body.

Note that external services will have different terms regarding caching, data collection, and rate limiting.

If the external service allows caching their responses on an intermediate service, the `cache_expire` setting can be used to specify the number of seconds the responses should be cached. This can avoid repeated calls to the external service for the same queries.

The rate limit settings are a combination of how many requests are allowed to be send to the external service. It's a "number" per "time interval" combination. In the above example, 10 requests per 60 seconds.

Export Configuration

Ichnaea supports exporting position data that it gets via the APIs to different export targets. This configuration lives in the `export_config` database table.

Currently three different kinds of backends are supported:

- `s3`: Amazon S3 buckets
- `internal`: Ichnaea's internal data processing pipeline which creates/ updates position data using new position information
- `geosubmit`: submitting position information to an HTTP POST endpoint in geosubmit v2 format

The type of the target is determined by the `schema` column of each entry.

All export targets can be configured with a `batch` setting that determines how many reports have to be available before data is submitted to the backend.

All exports have an additional `skip_keys` setting as a set of API keys. Data submitted using one of these API keys will not be exported to the target.

There can be multiple instances of the bucket and HTTP POST export targets in `export_config`, but only one instance of the internal export.

Here's the SQL for setting up an "internal" export target:

```
INSERT INTO export_config
(`name`, `batch`, `schema`) VALUES ("internal test", 1, "internal");
```

For a production setup you want to set the batch column to something like `100` or `1000` to get more efficiency. For initial testing its easier to set it to `1` so you immediately process any incoming data.

S3 Bucket Export (s3)

The schema column must be set to `s3`.

The S3 bucket export target combines reports into a gzipped JSON file and uploads them to the specified bucket `url`, for example:

```
s3://amazon_s3_bucket_name/directory/{source}{api_key}/{year}/{month}/{day}
```

The url can contain any level of additional static directories under the bucket root. The `{api_key}/{year}/{month}/{day}` parts will be dynamically replaced by the *api_key* used to upload the data, the source of the report (e.g. gnss) and the date when the backup took place. The files use a random UUID4 as the filename.

An example filename might be:

```
/directory/test/2015/07/15/554d8d3c-5b28-48bb-9aa8-196543235cf2.json.gz
```

Internal Export (internal)

The schema column must be set to `internal`.

The internal export target forwards the incoming data into the internal data pipeline.

HTTP Export (geosubmit)

The schema column must be set to `geosubmit`.

The HTTP export target buffers incoming data into batches of `batch` size and then submits them using the *Geosubmit Version 2: /v2/geosubmit* API to the specified `url` endpoint.

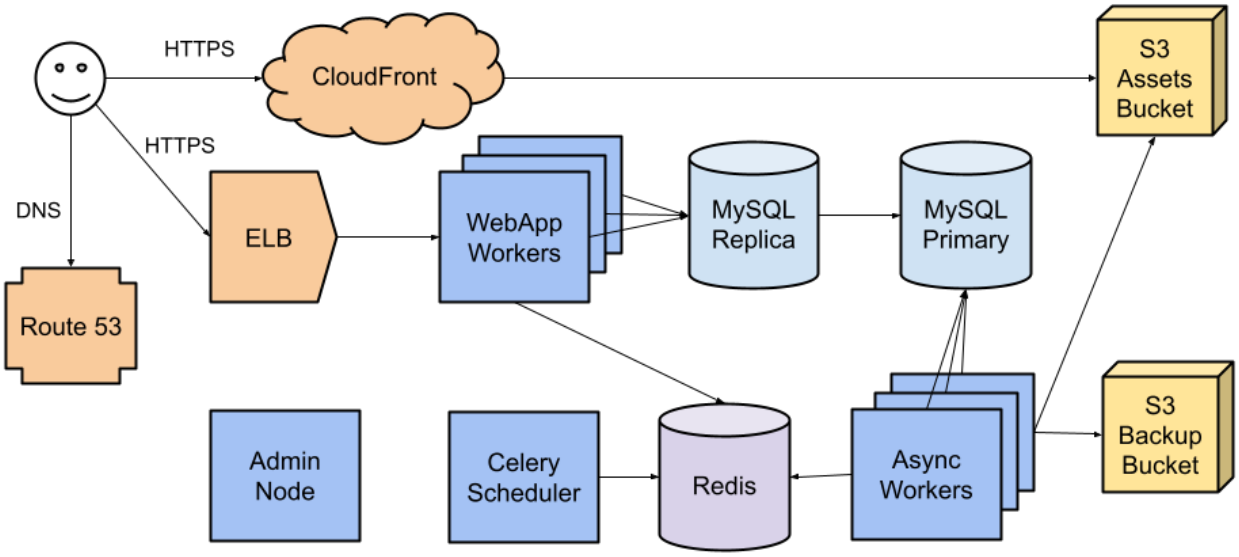
If the project is taking in data from a partner in a data exchange, the `skip_keys` setting can be used to prevent data being round tripped and sent back to the same partner that it came from.

1.2.5 Deployment

Mozilla deploys Ichnaea in an Amazon AWS environment, and there are some optional dependencies on specific AWS services like Amazon S3. The documentation assumes you are also using a AWS environment, but Ichnaea can be run in non-AWS environments as well.

Mozilla's Production Deployment

Mozilla's deployment of Ichnaea looks something like this:



The required parts are:

- One or more **WebApp workers** running the user-facing web page and APIs. Mozilla uses 20 EC2 instances in an Auto Scaling Group (ASG), behind an Elastic Load Balancer (ELB).
- One or more **Async workers** that run Celery tasks that process observations, update the station database, create map tiles, export data, and other tasks. Mozilla uses 5 EC2 instances in an ASG.
- A **Celery scheduler** to schedule periodic tasks. Mozilla uses an EC2 instance.

- A **MySQL or compatible database**, to store station data. Mozilla uses Amazon's Relational Database Service (RDS), MySQL 5.7, in Multi-AZ mode. The user-facing website does not write to a database, and reads from a read-only replica.
- A **Redis cache server**, for cached data, Celery tasks queues, and observation data pipelines. Mozilla uses Amazon's ElastiCache Redis, in Multi-AZ mode.

The optional parts are:

- An **S3 asset bucket** to store map tiles and public data like cell exports. Mozilla uses Cloudfront as a CDN in front of the asset bucket.
- An **S3 backup bucket** to store observation samples.
- An **Admin node**, to provide interactive access to the cluster and to run database migrations. Mozilla uses an EC2 instance.
- **DNS entries** to publish on the Internet. Mozilla uses AWS's Route 53.

Optional parts not shown on the diagram:

- A **statsd-compatible metrics server**. Mozilla uses InfluxDB Cloud.
- A **log aggregator**. Mozilla uses Google Cloud Logging.
- **Sentry**, for aggregating captured exceptions. Mozilla uses a self-hosted instance.

MySQL / Amazon RDS

The application is written and tested against MySQL 5.7.x or Amazon RDS of the same versions. MariaDB 10.5 has also been tested in the development environment.

The default configuration works for the most part, but ensure you are using UTF-8 to store strings. For example in `my.cnf`:

```
[mysqld]
character-set-server = utf8
collation-server = utf8_general_ci
init-connect='SET NAMES utf8'
```

The WebApp frontend role only needs access to a read-only version of the database, for example a read-replica. The Async Worker backend role needs access to the read-write primary database.

You need to create a database called `location` and a user with DDL privileges for that database.

Mozilla's deployment processes 500 to 1000 million observations a day. We have had issues in the past with replica lag, related binary log sizes, and transaction log sizes. The replica lag and disk usage should be monitored, for example with AWS RDS metrics. The transaction history length can be monitored via the metric *trx_history.length*. Mozilla reduced replica lag to 2 seconds or less by increasing `innodb_log_file_size` from 125 MB to 2 GB, and control observation throughput dynamically. See *Processing Backlogs and Rate Control* for more information.

Redis / Amazon ElastiCache

The application uses Redis:

- As the Celery backend and broker for the Async backend workers,
- As queues for observation data,
- As an API key cache, and
- As storage for API key rate limits and usage.

You can install a standard Redis or use Amazon ElastiCache (Redis). The application is tested against Redis 3.2.

Amazon S3

The application uses Amazon S3 for various tasks, including backup of *observations*, export of the aggregated cell table and hosting of the data map image tiles.

All of these are triggered by asynchronous jobs and you can disable them if you are not hosted in an AWS environment.

If you use Amazon S3 you might want to configure a lifecycle policy to delete old export files after a couple of days and *observation* data after one year.

Statsd / Sentry

The application uses Statsd to aggregate metrics and Sentry to log exception messages.

To use Statsd and Sentry, you need to configure them via environment variables as detailed in *the config section*.

Installation of Statsd and Sentry are outside the scope of this documentation.

Logging

The application logs to `stdout` by default. The WebApp logs using the MozLog format, while the Async workers have more traditional logs. If you want to view logs across a deployment, a logging aggregation system is needed. This is outside the scope of this documentation.

Image Tiles

The code includes functionality to render out image tiles for a data map of places where observations have been made. These can be stored in an S3 bucket, allowing them to be viewed on the website.

You can trigger this functionality periodically via a cron job, by calling the application container with the `map` argument.

Docker Config

The *development section* describes how to set up an environment used for working on and developing Ichnaea itself. For a production install, you should use pre-packaged docker images, instead of installing and setting up the code from Git.

Docker images are published to https://hub.docker.com/r/mozilla/location. When the `main` branch is updated (such as when a pull request is merged), an image is uploaded with a label matching the commit hash, such as `082156a5a8714a0db0b78f7b405ced2153184c1b`, as well as the `latest` tag. This is deployed to the stage deployment, and the deployed commit can be viewed at /__version__ on stage. When it is ready for production, it is tagged

with the date, such as `2021.08.16`, and is deployed to production. The deployed tag and commit can be viewed at /__version__ on prod, and the available tags at https://github.com/mozilla/ichnaea/tags.

Pull the desired docker image:

```
docker pull mozilla/location:2021.11.23
```

To test if the image was downloaded successfully, you can create a container and open a shell inside of it:

```
docker run -it --rm mozilla/location:2021.11.23 shell
```

Close the container again, either via `exit` or `Ctrl-D`.

Next create the application config as a docker environment file, for example called *env.txt*:

```
DB_READONLY_URI=mysql+pymysql://USER:PASSWORD@HOSTNAME:3306/location
DB_READWRITE_URI=mysql+pymysql://USER:PASSWORD@HOSTNAME:3306/location
SQLALCHEMY_URL=mysql+pymysql://USER:PASSWORD@HOSTNAME:3306/location
GEOIP_PATH=/mnt/geoip/GeoLite2-City.mmdb
REDIS_URI=redis://HOST:PORT/0
SECRET_KEY=change_this_value_or_it_will_not_be_secret
```

You can use either a single database user with DDL/DML privileges, or separate users for DDL (`SQLALCHEMY_URL`), read-write (`DB_READWRITE_URI`), and read-only (`DB_READONLY_URI`) privileges.

See *Environment variables* for additional options.

Database Setup

The user with DDL privileges and a database called `location` need to be created manually. If multiple users are used, the initial database setup will create the read-only / read-write users. Something like this should work in a `mysql` shell:

```
CREATE DATABASE location;

CREATE USER 'read'@'%' IDENTIFIED BY 'read-password';
GRANT SELECT ON location.* TO 'read'@'%';

CREATE USER 'write'@'%' IDENTIFIED BY 'write-password';
GRANT SELECT, INSERT, UPDATE, DELETE ON location.* TO 'write'@'%';

CREATE USER 'admin'@'%' IDENTIFIED BY 'admin-password';
GRANT ALL PRIVILEGES ON * TO 'admin'@'%';

quit
```

These usernames and passwords need to match the database connection URLs in the `env.txt` file. Next up, run the initial database setup:

```
docker run -it --rm --env-file env.txt \
    mozilla/location:2021.11.23 shell alembic stamp base
```

And update the database schema to the latest version:

```
docker run -it --rm --env-file env.txt \
    mozilla/location:2021.11.23 shell alembic upgrade head
```

The last command needs to be run whenever you upgrade to a new version of Ichnaea. You can inspect available database schema changes via alembic with the `history` and `current` sub-commands.

An API key will be needed to use the service, and a testing one can be created now that the database is available. This can be used to create one called `test`:

```
docker run -it --rm --env-file env.txt \
    mozilla/location:2021.11.23 shell /app/ichnaea/scripts/apikey.py create test
```

GeoIP

The application uses a Maxmind GeoIP City database for various tasks. It works both with the commerically available and Open-Source GeoLite databases in binary format.

You can download the GeoLite database for free from MaxMind after signing up for a GeoLite2 account.

Download and untar the downloaded file. Put the `GeoLite2-City.mmdb` into a directory accessible to docker (for example `/opt/geoip`). The directory or file can be mounted into the running docker containers.

You can update this file on a regular basis. Typically once a month is enough for the GeoLite database. Make sure to stop any containers accessing the file before updating it and start them again afterwards. The application code doesn't tolerate having the file being changed underneath it.

Docker Runtime

Finally you are ready to start containers for the three different application roles.

There is a web frontend, a task worker and a task scheduler role. The scheduler role is limited to a single running container. You need to make sure to never have two containers for the scheduler running at the same time. If you use multiple physical machines, the scheduler must only run on one of them.

The web app and task worker roles both scale out and you can run as many of them as you want. You can tune the web instance with the variables `GUNICORN_WORKERS` and similar variables - see docker/run_web.sh for details. You can run a single docker container per physical/virtual machine, or multiple with a system like Kubernetes.

All roles communicate via the database and Redis only, so can be run on different virtual or physical machines. The task workers load balance their work internally via data structures in Redis.

If you run multiple web frontend roles, you need to put a load balancer in front of them. The application does not use any sessions or cookies, so the load balancer can simply route traffic via round-robin.

You can configure the load balancer to use the `/__lbheartbeat__` HTTP endpoint to check for application health.

If you want to use docker as your daemon manager run:

```
docker run -d --env-file env.txt \
    --volume /opt/geoip:/mnt/geoip
    mozilla/location:2021.11.23 scheduler
```

The `/opt/geoip` directory is the directory on the docker host, with the `GeoLite2-City.mmdb` file inside it. The `/mnt/geoip/` directory corresponds to the `GEOIP_PATH` config section in the `env.txt` file.

The two other roles are started in the same way:

```
docker run -d --env-file env.txt \
    --volume /opt/geoip:/mnt/geoip
    mozilla/location:2021.11.23 worker
```

(continues on next page)

(continued from previous page)

```
docker run -d --env-file env.txt \
    --volume /opt/geoip:/mnt/geoip
    -p 8000:8000/tcp
    mozilla/location:2021.11.23 web
```

The web role can take an additional argument to map the port 8000 from inside the container to port 8000 of the docker host machine.

You can put a web server (e.g. Nginx) in front of the web role and proxy pass traffic to the docker container running the web frontend.

Runtime Checks

To check whether or not the application is running, you can check the web role, via:

```
curl -i http://localhost:8000/__heartbeat__
```

This should produce output like:

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 10 Jan 2022 23:34:25 GMT
Content-Length: 193
Connection: keep-alive

{"database": {"up": true, "time": 2, "alembic_version": "3be4004781bc"},
 "geoip": {"up": true, "time": 0, "age_in_days": 4, "version": "2022-01-06T16:20:21Z"},
 "redis": {"up": true, "time": 1}}
```

The `__lbheartbeat__` endpoint has simpler output and doesn't check the database / Redis backend connections. The application is designed to degrade gracefully and continue to work with limited capabilities without working database and Redis backends.

The `__version__` endpoint shows what version of the software is currently running.

To test one of the HTTP API endpoints, you can use:

```
curl -H "X-Forwarded-For: 81.2.69.192" \
    http://localhost:8000/v1/geolocate?key=test
```

Change the command if you used a name other that `test` for a first API key in the *Database Setup*.

This should produce output like:

```
{"location": {"lat": 51.5142, "lng": -0.0931}
```

Test this with different IP addresses like `8.8.8.8` to make sure the database file was picked up correctly.

Upgrade

In order to upgrade a running installation of Ichnaea to a new version, first check and get the docker image for the new version, for example:

```
docker pull mozilla/location:2021-11-23
```

Next up stop all containers running the scheduler and task worker roles. If you use docker's own daemon support, the `ps`, `stop` and `rm` commands can be used to accomplish this.

Now run the database migrations found in the new image:

```
docker run -it --rm --env-file env.txt \
    mozilla/location:2.2.0 alembic upgrade head
```

The web app role can work with both the old database and new database schemas. The worker role might require the new database schema right away.

Start containers for the scheduler, worker and web roles based on the new image.

Depending on how you run your web tier, swich over the traffic from the old web containers to the new ones. Once all traffic is going to the new web containers, stop the old web containers.

1.2.6 Processing Backlogs and Rate Control

Ideally, all *observations* would be used to update the Ichnaea database. This would require backend resources (the worker servers, the cache, and the database) that can easily handle peak traffic. Such a system will be overprovisioned most of the time, as traffic ebbs and flows over days and weeks. A backend that can easily handle peak traffic may be prohibitively expensive or impractical, and require quickly adding more resources if traffic is consistently growing. There is a complex relationship between API traffic and observations, so it is not possible to predict the required backend resources for future traffic.

When the backend is unable to process observations, a backlog builds up. This is acceptable for traffic spikes or daily peaks, and the backend works through the backlog when the traffic drops below the peaks.

When traffic consistently exceeds the capacity of the backend, a steadily increasing backlog is generated, causing problems. A large backlog tends to slow down the cache and database, which can increase the rate of backlog growth. The backlog is stored in Redis, which eventually runs out of memory. Redis slows down dramatically as it determines what data it can throw away, resulting in slow API requests and timeouts. Eventually, Redis will discard entire backlogs, "fixing" the problem but losing data. An administrator can monitor the backlog by looking at Redis memory usage and at the *queue Metric*.

The chart below shows what a steadily increasing backlog looks like. The backlog would take less than an hour to clear, but new observations continue to be added from API usage. Around 3 PM, API usage generates fewer observations, allowing the backend to make progress on reducing the backlog. Around 6 PM, after an hour of lower traffic, the API usage exceeds the backend capabilities again, and the backlog begins increasing.

If traffic is steadily increasing over weeks and months, eventually the backend will be unable to recover in a full 24 hour cycle, leading to slower service and eventually data loss.

It is possible to have a steady backlog that does not cause issues for Redis, but that strains the database resources, by increasing replica lag or accumulating old transaction logs.

The primary database can have a large number of write operations, and the replica database, used for read-only operations, can fall behind. This causes a build-up of binary logs on the primary database, filling up the disk. It also delays

changes to API keys, such as adding a key or changing requests limits, from being applied to the API. The primary database disk usage and the replica lag should be monitored to detect and prevent this problem. Mozilla was able to decrease replica lag to seconds by setting a larger `innodb_log_file_size` (125 MB to 2 GB).

Updates from observations can cause the transaction history / undo logs to grow faster than the purge process can delete them. This causes the system log (`ibdata1`) to grow, and it can't be shrunk without recreating the database. The primary database disk usage and transaction history length should be monitored to prevent this problem.

Ichnaea has rate controls that can be used to sample incoming data, and reduce the observations that need to be processed by the backend. It can also turn off processing if the InnoDB transaction history becomes too large. There are currently no automated controls for replica lag.

Rate Control by API Key

There are two rate controls that are applied by API key, in the `api_key` database table:

- `store_sample_locate` (0 - 100) - The percent of locate API calls that are turned into observations.
- `store_sample_submit` (0 - 100) - The percent of submission API calls that are turned into observations

An administrator can use these to limit the observations from large API users, or to ignore traffic from questionable API users. The default is 100% (all locate and submission observations) for new API keys.

Global Rate Control

The Redis key `global_locate_sample_rate` is a number (stored as a string) between 0 and 100.0 that controls a sample rate for all locate calls. This is applied as a multiple on any API key controls, so if an API has `store_sample_locate` set to 60, and `global_locate_sample_rate` is 50.0, the effective sample rate for that API key is 30%.

An administrator can use this control to globally limit observations from *geolocate* calls. A temporary rate of 0% is an effective way to allow the backend to process a large backlog of observations. If unset, the default global rate is 100%.

There is no global rate control for submissions.

Automated Rate Control

Optionally, an automated rate controller can set the global locate sample rate. The rate controller is given a target of the maximum data queue backlog, and periodically compares this to the backlog. It lowers the rate while the backlog is near or above the target, and raises it to 100% again when below the target.

To enable the rate controller:

1. Set the Redis key `rate_controller_target` to the desired maximum queue size, such as `1000000` for 1 million observations. A suggested value is 5-10 minutes of maximum observation processing, as seen by summing the *data.observation.insert metric* during peak periods with a backlog.
2. Set the Redis key `rate_controller_enabled` to `1` to enable or `0` to disable the rate controller. If the rate controller is enabled without a target, it will be automatically disabled.

The rate controller runs once a minute, at the same time that *queue metrics* are emitted. The rate is adjusted during the peak traffic to keep the backlog near the target rate, and the backlog is more quickly processed when the peak ends.

In our simulation, the controller picked a sample rate between 90% and 100% during peak traffic, which was sufficient to keep the queue sizes slightly above the target. This means that most observations will be processed, even during busy periods. It quickly responded to traffic spikes during peak periods by dropping the sample rate to 60% or lower.

The rate controller is a general proportional-integral-derivative controller (PID controller), provided by simple-pid. By default, only the proportional gain K_p is enabled, making it a P controller. The input is the queue size in observations, and the output is divided by the target, so the output is between 0.0 and 1.0 when the data queues exceed the target, and greater than 1.0 when below the target. This is limited to imited to the range 0.0 to 1.0, and then multiplied by 100 to derive the new sample rate.

The gain parameters are stored in Redis keys, and can be adjusted:

- K_p (Redis key `rate_controller_kp`, default 8.0) - The proportional gain. Values between 1.0 and 10.0 work well in simulation. This controls how aggressively the controller drops the rate when the targer is exceeded. For example, for the same queue size, K_p=2.0 may lower the rate to 95% while K_p=8.0 may lower it to 80%.
- K_i (Redis key `rate_controller_ki`, default 0.0) - The integral gain. The integral collects the accumulated "error" from the target. It tends to cause the queue size to overshoot the target, then sets the rate to 0% to recover. 0.0 is recommended, and start at low values like 0.0001 only if there is a steady backlog due to an underprovisioned backend.
- K_d (Redis key `rate_controller-kd`, default 0.0) - The derivative gain. The derivative measures the change since the last reading. In simulation, this had little noticable effect, and may require a value of 50.0 or higher to see any changes.

The rate controller emits several *metrics*. An administrator can use these metrics to monitor the rate controller, and to determine if backend resources should be increased or decreased based on long-term traffic trends.

Transaction History Monitoring

Observation processing can cause the InnoDB transaction history or undo logs to grow faster than the purge process can delete them. This can cause disk usage to grow in a way that can't be reduced without recreating the database.

The size of the transaction history can be monitored, if the celery worker database connection ("read-write") has the PROCESS privilege. To turn on transaction history monitoring, ensure this connection can execute the SQL:

```
SELECT count FROM information_schema.innodb_metrics WHERE name = 'trx_rseg_history_len';
```

The transaction history length will be checked at the same time as the queue sizes, and will emit a metric *trx_history.length*. An administrator can use these metrics to monitor the transaction history length, and tune aspects of the MySQL transaction system or backend processing.

Additionally, the automated rate controller can be used to pause processing of locate samples and reduce the creation of new transactions. When the transaction history length exceeds a maximum size, the global locate sample rate is set to 0%. When the MySQL purge process reduces the transaction history to a safe level, the rate is allowed to rise again. *Additional metrics* are emitted to track the process.

To use the rate controller to pause processing:

- Tune the Redis key `rate_controller_trx_max` to the maximum history length, such as `1000000`. This should be set to a number that takes less than a day to clear.
- Tune the Redis key `rate_controller_trx_min` to the minimum history length, such as `1000`. This should be set to a number that is between 0 and the maximum rate.
- Set the Redis key `rate_controller_target` and `rate_controller_enabled` as described in *Automated Rate Control*.

1.2.7 Metrics and Structured Logs

Ichnaea provides two classes of runtime data:

- Statsd-style **Metrics**, for real-time monitoring and easy visual analysis of trends
- **Structured Logs**, for offline analysis of data and targeted custom reports

Structured logs were added in 2020, and the migration of data from metrics to logs is not complete. For more information, see the *Implementation* section.

Metrics are emitted by the web / API application, the backend task application, and the datamaps script:

Metric Name	App	Type	Tags
api.limit	task	gauge	key, path
data.batch.upload	web	counter	key
data.export.batch	task	counter	key
data.export.upload	task	counter	key, status
data.export.upload.timing	task	timer	key
data.observation.drop	task	counter	type, key
data.observation.insert	task	counter	type
data.observation.upload	task	counter	type, key
data.report.drop	task	counter	key
data.report.upload	task	counter	key
data.station.blocklist	task	counter	type
data.station.confirm	task	counter	type
data.station.dberror	task	counter	type, errno
data.station.new	task	counter	type
datamaps.dberror	task	counter	errno
locate.fallback.cache	web	counter	fallback_name, status
locate.fallback.lookup	web	counter	fallback_name, status
locate.fallback.lookup.timing	web	timer	fallback_name, status
locate.query	web	counter	key, geoip, blue, cell, wifi
locate.request	web	counter	key, path
locate.result	web	counter	key, accuracy, status, source, fallback_allowed
locate.source	web	counter	key, accuracy, status, source
locate.user	task	gauge	key, interval
queue	task	gauge	data_type, queue, queue_type
rate_control.locate	task	gauge	
rate_control.locate.dterm	task	gauge	
rate_control.locate.iterm	task	gauge	
rate_control.locate.kd	task	gauge	
rate_control.locate.ki	task	gauge	
rate_control.locate.kp	task	gauge	
rate_control.locate.pterm	task	gauge	
region.query	web	counter	key, geoip, blue, cell, wifi
region.request	web	counter	key, path
region.result	web	counter	key, accuracy, status, source, fallback_allowed
region.user	task	gauge	key, interval
request	web	counter	path, method, status
request.timing	web	timer	path, method
submit.request	web	counter	key, path
submit.user	task	gauge	key, interval

continues on next page

Table 1 – continued from previous page

Metric Name	App	Type	Tags
task	task	timer	task
trx_history.length	task	gauge	
trx_history.max	task	gauge	
trx_history.min	task	gauge	
trx_history.purging	task	gauge	

Web Application Metrics

The website handles HTTP requests, which may be page requests or API calls.

Request Metrics

Each HTTP request, including API calls, emits metrics and a structured log entry.

request

`request` is a counter for almost all HTTP requests, including API calls. The exceptions are static assets, like CSS, images, Javascript, and fonts, as well as some static content like `robots.txt`.

Additionally, invalid requests (HTTP status in the `4xx` range) do not omit this metric, unless they are API endpoints.

The `path` tag is the request path, like `/stats/regions`, but normalized to tag-safe characters. The initial slash is dropped, and remaining slashes are replaced with periods, so `/stats/regions` becomes `stats.regions`. The homepage, `/` is normalized as `.homepage`, to avoid an empty tag value.

Tags:

- `path`: The metrics-normalized HTTP path, like `stats.regions`, `v1.geolocate`, and `.homepage`
- `method`: The HTTP method in lowercase, like `post`, `get`, `head`, and `options`
- `status`: The returned HTTP status, like `200` for success and `400` for client errors

Related structured log data:

- *http_method*: The non-normalized HTTP method
- *http_path*: The non-normalized request path
- *http_status*: The HTTP status code

request.timing

`request.timing` is a timer for how long the HTTP request took to complete in milliseconds.

Tags:

The tags `path` and `method` are the same as *request*. The tag `status` is omitted.

Related structured log data:

- *duration_s*: The time the request took in seconds, rounded to the millisecond

API Metrics

These metrics are emitted when the API is called.

data.batch.upload

`data.batch.upload` is a counter that is incremented when a submit API, like */v2/geosubmit*, is called with any valid data. A submission batch could contain a single report or multiple reports, but both would increment `data.batch.upload` by one. A batch with no (valid) reports does not increment this metric.

Tags:

- `key`: The API key, often a UUID, or omitted if the API key is not valid.

Related structured log data:

- *api_key*: The same value as tag `key` for valid keys

locate.query

`locate.query` is a counter, incremented each time the *Geolocate API* is used with a valid API key that is not rate limited. It is used to segment queries by the station data contained in the request body.

Tags:

- `key`: The API key, often a UUID
- `geoip`: `false` if there was no GeoIOP data, and omitted when there is GeoIP data for the client IP (the common case)
- `blue`: Count of valid Bluetooth *stations* in the request, `none`, `one` or `many`
- `cell`: Count of valid cell *stations* in the request, `none`, `one`, or `many`
- `wifi`: Count of valid WiFi *stations* in the request, `none`, `one`, or `many`

Changed in version 2020.04.16: Removed the `region` tag

Related structured log data:

- *api_key*: The same value as tag `key`
- *has_geoip*: Always set, `False` when `geoip` is `false`
- *blue*: Count of Bluetooth stations, as a number instead of text like `many`
- *cell*: Count of Cell stations
- *wifi*: Count of WiFi stations

locate.request

`locate.request` is a counter, incremented for each call to the *Geolocate API*.

Tags:

- `key`: The API key, often a UUID, or `invalid` for a known key that can not call the API, or `none` for an omitted key.
- `path`: `v1.geolocate`, the standardized API path

Related structured log data:

- *api_key*: The same value as tag `key`, except that instead of `invalid`, the request key is used, and `api_key_allowed=False`
- *api_key_allowed*: `False` when the key is not allowed to use the API
- *api_path*: The same value as tag `path`
- *api_type*: The value `locate`

locate.result

`locate.result` is a counter, incremented for each call to the *Geolocate API* with a valid API key that is not rate limited.

If there are no Bluetooth, Cell, or WiFi networks provided, and GeoIP data is not available (for example, the IP fallback is explicitly disabled), then this metric is not emitted.

Tags:

- `key`: The API key, often a UUID
- `accuracy`: The expected accuracy, based on the sources provided:
  - `high`: At least two Bluetooth or WiFi networks
  - `medium`: No Bluetooth or WiFi networks, at least one cell network
  - `low`: No networks, only GeoIP data
- `status`: Could we provide a location estimate?
  - `hit` if we can provide a location with the expected accuracy,
  - `miss` if we can not provide a location with the expected accuracy. For cell networks (`accuracy=medium`), a `hit` includes the case where there is not an exact cell match, but the cell area (the area covered by related cells) is small enough (smaller than tens of kilometers across) for an estimate.
- `source`: The source that provided the hit:
  - `internal`: Our crowd-sourced network data
  - `geoip`: The MaxMind GeoIP database
  - `fallback`: An optional external fallback provider
  - Omitted when `status=miss`
- `fallback_allowed`:
  - `true` if the external fallback provider was allowed
  - Omitted if the external fallback provider was not allowed

Changed in version 2020.04.16: Removed the `region` tag

Related structured log data:

- *accuracy*: The accuracy level of the result, `high`, `medium`, or `low`
- *accuracy_min*: The same value as tag `accuracy`
- *api_key*: The same value as tag `key`
- *result_status*: The same value as tag `status`

locate.source

`locate.source` is a counter, incremented for each processed source in a location query. If *station* data (Bluetooth, WiFi, and Cell data) is provided, this usually two metrics for one request, one for the `internal` source and one for the `geoip` source.

The required accuracy for a `hit` is set by the kind of station data in the request. For example, a request with no station data requires a `low` accuracy, while one with multiple WiFi networks requires a `high` accuracy. The `high` accuracy is at least 500 meters, and the minimum current MaxMind accuracy is 1000 meters, so the `geoip` source is expected to have a `miss` status when accuracy is `high`.

Tags (similar to *locate.result*) :

- `key`: The API key, often a UUID
- `accuracy`: The expected accuracy, based on the sources provided:
  - `high`: At least two Bluetooth or WiFi networks
  - `medium`: No Bluetooth or WiFi networks, at least one cell network
  - `low`: No networks, only GeoIP data
- `status`: Could we provide a location estimate?
  - `hit`: We can provide a location with the expected accuracy,
  - `miss`: We can not provide a location with the expected accuracy
- `source`: The source that was processed:
  - `internal`: Our crowd-sourced network data
  - `geoip`: The MaxMind GeoIP database
  - `fallback`: An optional external fallback provider
- `fallback_allowed`:
  - `true` if the external fallback provider was allowed
  - Omitted if the external fallback provider was not allowed

Changed in version 2020.04.16: Removed the `region` tag

Related structured log data:

- *api_key*: The same value as tag `key`
- *source_internal_accuracy*: The accuracy level of the internal source
- *source_internal_accuracy_min*: The required accuracy level of the internal source, same value as tag `accuracy` when `source=internal`
- *source_internal_status*: The same value as tag `status` when `source=internal`

- *source_geoip_accuracy*: The accuracy level of the GeoIP source
- *source_geoip_accuracy_min*: The required accuracy level of the GeoIP source, same value as tag `accuracy` when `source=geoip`
- *source_geoip_status*: The same value as tag `status` when `source=geoip`
- *source_fallback_accuracy*: The accuracy level of the external fallback source
- *source_fallback_accuracy_min*: The required accuracy level of the fallback source, same value as tag `accuracy` when `source=fallback`
- *source_fallback_status*: The same value as tag `status` when `source=fallback`

region.query

`region.query` is a counter, incremented each time the *Region API* is used with a valid API key. It is used to segment queries by the station data contained in the request body.

It has the same tags (`key`, `geoip`, `blue`, `cell`, and `wifi`) as *locate.query*.

region.request

`region.request` is a counter, incremented for each call to the *Region API*.

It has the same tags (`key` and `path`) as *locate.request*, except the `path` tag is `v1.country`, the standardized API path.

region.result

`region.result` is a counter, incremented for each call to the *Region API* with a valid API key that is not rate limited.

If there are no Bluetooth, Cell, or WiFi networks provided, and GeoIP data is not available (for example, the IP fallback is explicitly disabled), then this metric is not emitted.

It has the same tags (`key`, `accuracy`, `status`, `source`, and `fallback_allowed`) as *locate.result*.

region.source

`region.source` is a counter, incremented for each processed source in a region query. If *station* data (Bluetooth, WiFi, and Cell data) is provided, this usually two metrics for one request, one for the `internal` source and one for the `geoip` source. In practice, most users provide no station data, and only the `geoip` source is emitted.

It has the same tags (`key`, `accuracy`, `status`, `source`, and `fallback_allowed`) as *locate.source*.

submit.request

`submit.request` is a counter, incremented for each call to a Submit API:

- *Geosubmit Version 2: /v2/geosubmit*
- *Submit: /v1/submit (DEPRECATED)*
- *Geosubmit: /v1/geosubmit (DEPRECATED)*

This counter can be used to determine when the deprecated APIs can be removed.

It has the same tags (`key` and `path`) as *locate.request*, except the `path` tag is `v2.geosubmit`, `v1.submit`, or `v1.geosubmit`, the standardized API path.

API Fallback Metrics

These metrics were emitted when the fallback location provider was called. MLS stopped using this feature in 2019, so these metrics are not emitted, but the code remains as of 2020.

These metrics have not been converted to structured logs.

locate.fallback.cache

`locate.fallback.cache` is a counter for the performance of the fallback cache.

Tags:

- `fallback_name`: The name of the external fallback provider, from the API key table
- `status`: The status of the fallback cache:
  - `hit`: The cache had a previous result for the query
  - `miss`: The cache did not have a previous result for the query
  - `bypassed`: The cache was not used, due to mixed *stations* in the query, or the high number of individual stations
  - `inconsistent`: The cached results were for multiple inconsistent locations
  - `failure`: The cache was unreachable

locate.fallback.lookup

`locate.fallback.lookup` is a counter for the HTTP response codes returned from the fallback server.

Tags:

- `fallback_name`: The name of the external fallback provider, from the API key table
- `status`: The HTTP status code, such as `200`

locate.fallback.lookup.timing

`locate.fallback.lookup.timing` is a timer for the call to the fallback location server.

Tags:

- `fallback_name`: The name of the external fallback provider, from the API key table
- `status`: The HTTP status code, such as `200`

Web Application Structured Logs

There is one structured log emitted for each request, which may be an API request. The structured log data includes data that was emitted as one or more metrics.

Request Metrics

All requests, with the exception of static assets and static views (see *request*), include this data:

- `duration_s`: The time in seconds, rounded to the millisecond, to serve the request.
- `http_method`: The HTTP method, like `POST` or `GET`.
- `http_path`: The request path, like `/` for the homepage, or `/v1/geolocate` for the API.
- `http_status`: The response status, like `200` or `400`.

This data is duplicated in metrics:

- *request*
- *request.timing*

API Metrics

If a request is an API call, additional data can be added to the log:

- `accuracy`: The accuracy of the result, `high`, `medium`, or `low`.
- `accuracy_min`: The minimum required accuracy of the result for a hit, `high`, `medium`, or `low`.
- `api_key`: An API key that has an entry in the API key table, often a UUID, or `none` if omitted. Same as statsd tag `key`, except that known but disallowed API keys are the key value, rather than `invalid`.
- `api_key_allowed`: `False` if a known API key is not allowed to call the API, omitted otherwise.
- `api_key_db_fail`: `True` when a database error prevented checking the API key. Omitted when the check is successful.
- `api_path`: The normalized API path, like `v1.geolocate` and `v2.geosubmit`. Same as statsd tag `path` when an API is called.
- `api_response_sig`: A hash to identify repeated geolocate requests getting the same response without identifying the client.
- `api_type`: The API type, `locate`, `submit`, or `region`.
- `blue`: The count of Bluetooth radios in the request.
- `blue_valid`: The count of valid Bluetooth radios in the request.
- `cell`: The count of cell tower radios in the request.
- `cell_valid`: The count of valid cell tower radios in the request.
- `fallback_allowed`: `True` if the optional fallback location provider can be used by this API key, `False` if not.
- `has_geoip`: `True` if there is GeoIP data for the client IP, otherwise `False`.
- `has_ip`: `True` if the client IP was available, otherwise `False`.
- `invalid_api_key`: The invalid API key not found in API table, omitted if known or empty.
- `rate_allowed`: `True` if allowed, `False` if not allowed due to rate limit, or omitted if the API is not rate-limited.

- `rate_quota`: The daily rate limit, or omitted if API is not rate-limited.
- `rate_remaining`: The remaining API calls to hit limit, 0 if none remaining, or omitted if the API is not rate-limited.
- `region`: The ISO region code for the IP address, `null` if none.
- `result_status`: `hit` if an accurate estimate could be made, `miss` if it could not.
- `source_fallback_accuracy`: The accuracy level of the external fallback source, `high`, `medium`, or `low`.
- `source_fallback_accuracy_min`: The required accuracy level of the fallback source.
- `source_fallback_status`: `hit` if the fallback source provided an accurate estimate, `miss` if it did not.
- `source_internal_accuracy`: The accuracy level of the internal source (Bluetooth, WiFi, and cell data compared against the database), `high`, `medium`, or `low`.
- `source_internal_accuracy_min`: The required accuracy level of the internal source.
- `source_internal_status`: `hit` if the internal check provided an accurate estimate, `miss` if it did not.
- `source_geoip_accuracy`: The accuracy level of the GeoIP source, `high`, `medium`, or `low`.
- `source_geoip_accuracy_min`: The required accuracy level of the GeoIP source.
- `source_geoip_status`: `hit` if the GeoIP database provided an accurate estimate, `miss` if it did not.
- `wifi`: The count of WiFi radios in the request.
- `wifi_valid`: The count of valid WiFi radios in the request.

Some of this data is duplicated in metrics:

- *api.limit*
- *locate.query*
- *locate.request*
- *locate.result*
- *locate.source*
- *region.query*
- *region.request*
- *region.result*
- *region.source*
- *submit.request*

Task Application Metrics

The task application, running on celery in the backend, implements the data pipeline and other periodic tasks. These emit metrics, but have not been converted to structured logging.

API Monitoring Metrics

These metrics are emitted periodically to monitor API usage. A Redis key is incremented or updated during API requests, and the current value is reported via these metrics:

api.limit

`api.limit` is a gauge of the API requests, segmented by API key and API path, for keys with daily limits. It is updated every 10 minutes.

Tags:

- `key`: The API key, often a UUID
- `path`: The normalized API path, such as `v1.geolocate` or `v2.geosubmit`

Related structured log data is added during the request when an API key has rate limits:

- *rate_allowed*: `True` if the request was allowed, `False` if not allowed due to the rate limit
- *rate_quota*: The daily rate limit
- *rate_remaining*: The remaining API calls to hit limit, 0 if none remaining

locate.user

`locate.user` is a gauge of the estimated number of daily and weekly users of the *Geolocate API* by API key. It is updated every 10 minutes.

The estimate is based on the client's IP address. At request time, the IP is added via PFADD to a HyperLogLog structure. This structure can be used to estimate the cardinality (number of unique IP addresses) to within about 1%. See PFCOUNT for details on the HyperLogLog implementation.

Tags:

- `key`: The API key, often a UUID
- `interval`: `1d` for the daily estimate, `7d` for the weekly estimate.

region.user

`region.user` is a gauge of the estimated number of daily and weekly users of the *Region API* by API key. It is updated every 10 minutes.

It has the same tags (`key` and `interval`) as *locate.user*.

submit.user

`submit.user` is a gauge of the estimated number of daily and weekly users of the submit APIs (*/v2/geosubmit* and the deprecated submit APIs) by API key. It is updated every 10 minutes.

It has the same tags (`key` and `interval`) as *locate.user*.

Data Pipeline Metrics - Gather and Export

The data pipeline processes data from two sources:

- **Submission reports**, from the submission APIs, which include a position from an external source like GPS, along with the Wifi, Cell, and Bluetooth *stations* that were seen.
- **Location queries**, from the geolocate and region APIs, which include an estimated position, along with the stations.

Multiple reports can be submitted in one call to the submission APIs. Each batch of reports increment the *data.batch.upload* metric when the API is called. A single report is created for each location query, and there is no corresponding metric.

The APIs feed these *reports* into a Redis queue `update_incoming`, processed by the backend task of the same name. This task copies reports to "export" queues. Four types are supported:

- `dummy`: Does nothing, for pipeline testing
- `geosubmit`: POST reports to a service supporting the *Geosubmit API*.
- `internal`: Divide *reports* into *observations*, for further processing to update the internal database.
- `s3`: Store report JSON in S3.

Ichnaea supports multiple export targets for a type. In production, there are three export targets, identified by an export key:

- `backup`: An `s3` export, to a Mozilla-private S3 bucket
- `tostage`: A `geosubmit` export, to send a sample of reports to stage for integration testing.
- `internal`: An `internal` export, to update the database

The data pipeline has not been converted to structured logging. As data moves through this part of the data pipeline, these metrics are emitted:

data.export.batch

`data.export.batch` is a counter of the report batches exported to external and internal targets.

Tags:

- `key`: The export key, from the export table. Keys used in Mozilla production:
  - `backup`: Reports archived in S3
  - `tostage`: Reports sent from production to stage, as a form of integration testing
  - `internal`: Reports queued for processing to update the internal station database

data.export.upload

`data.export.upload` is a counter that tracks the status of export jobs.

Tags:

- `key`: The export key, from the export table. Keys used in Mozilla production are `backup` and `tostage`, with the same meaning as *data.export.batch*. Unlike that metric, `internal` is not used.
- `status`: The status of the export, which varies by type of export:
  - `backup`: `success` or `failure` storing the report to S3
  - `tostage`: HTTP code returned by the submission API, usually `200` for success or `400` for failure.

data.export.upload.timing

`data.export.upload.timing` is a timer for the report batch export process.

Tags:

- `key`: The export key, from the export table. See *data.export.batch* for the values used in Mozilla production.

data.observation.drop

`data.observation.drop` is a counter of the Bluetooth, cell, or WiFi *observations* that were discarded before integration due to some internal consistency, range or validity-condition error encountered while attempting to normalize the observation.

Tags:

- `key`: The API key, often a UUID. Omitted if unknown or not available
- `type`: The *station* type, one of `blue`, `cell`, or `wifi`

data.observation.upload

`data.observation.upload` is a counter of the number of Bluetooth, cell or WiFi *observations* entering the data processing pipeline, before normalization and blocked station processing. This count is taken after a batch of *reports* are decomposed into observations.

The tags (`key` and `type`) are the same as *data.observation.drop*.

data.report.drop

`data.report.drop` is a counter of the *reports* discarded due to some internal consistency, range, or validity-condition error.

Tags:

- `key`: The API key, often a UUID. Omitted if unknown or not available

data.report.upload

`data.report.upload` is a counter of the *reports* accepted into the data processing pipeline.

It has the same tag (`key`) as *data.report.drop*.

Data Pipeline Metrics - Update Internal Database

The internal export process decomposes *reports* into *observations*, pairing one position with one *station*. Each observation works its way through a process of normalization, consistency-checking, and (possibly) integration into the database, to improve future location estimates.

The data pipeline has not been converted to structured logging. As data moves through the pipeline, these metrics are emitted:

data.observation.insert

`data.observation.insert` is a counter of the Bluetooth, cell, or WiFi *observations* that were successfully validated, normalized, integrated.

Tags:

- `type`: The *station* type, one of `blue`, `cell`, or `wifi`

data.station.blocklist

`data.station.blocklist` is a counter of the Bluetooth, cell, or WiFi *stations* that are blocked from being used to estimate positions. These are added because there are multiple valid *observations* at sufficiently different locations, supporting the theory that it is a mobile station (such as a picocell or a mobile hotspot on public transit), or was recently moved (such as a WiFi base station that moved with the owner to a new home).

Tags:

- `type`: The *station* type, one of `blue`, `cell`, or `wifi`

data.station.confirm

`data.station.confirm` is a counter of the Bluetooth, cell or WiFi *stations* that were confirmed to still be active. An *observation* from a location query can be used to confirm a station with a position based on submission reports.

It has the same tag (`type`) as *data.station.blocklist*

data.station.dberror

`data.station.dberror` is a counter of retryable database errors, which are encountered as multiple task threads attempt to update the internal database.

Retryable database errors, like a lock timeout (`1205`) or deadlock (`1213`) cause the station updating task to sleep and start over. Other database errors are not counted, but instead halt the task and are recorded in Sentry.

Tags:

- `errno`: The error number, which can be found in the MySQL Server Error Reference

- `type`: The *station*, one of `blue`, `cell`, or `wifi`, or the aggregate station type `cellarea`

data.station.new

`data.station.new` is a counter of the Bluetooth, cell or WiFi *stations* that were discovered for the first time.

Tags:

- `type`: The *station* type, one of `blue`, `cell`, or `wifi`

datamaps.dberror

`datamaps.dberror` is a counter of the number of retryable database errors when updating the `datamaps` tables.

Tags:

- `errno`: The error number, same as *data.station.dberror*

Backend Monitoring Metrics

queue

`queue` is a gauge that reports the current size of task and data queues. Queues are implemented as Redis lists, with a length returned by LLEN.

Task queues hold the backlog of celery async tasks. The names of the task queues are:

- `celery_blue`, `celery_cell`, `celery_wifi` - A task to process a chunk of *observation* data
- `celery_content` - Tasks that update website content, like the datamaps and statistics
- `celery_default` - A generic task queue
- `celery_export` - Tasks exporting data, either public cell data or the Data Pipeline
- `celery_monitor` - Tasks updating metrics gauges for this metric and *API Monitoring Metrics*
- `celery_reports` - Tasks handling batches of submission reports or location queries

Data queues are the backlog of *observations* and other data items to be processed. Data queues have names that mirror the shared database tables:

- `update_blue_0` through `update_blue_f` (16 total) - Observations of Bluetooth stations
- `update_cell_gsm`, `update_cell_lte`, and `update_cell_wcdma` - Observations of cell stations
- `update_cell_area` - Aggregated observations of cell towers `data_type: cellarea`
- `update_datamap_ne`, `update_datamap_nw`, `update_datamap_se`, and `update_datamap_sw` - Approximate locations for the contribution map
- `update_incoming` - Incoming reports from geolocate and submission APIs
- `update_wifi_0` through `update_wifi_f` (16 total) - Observations of WiFi stations

Tags:

- `queue`: The name of the task or data queue
- `queue_type`: `task` or `data`

- `data_type`: For data queues, `bluetooth`, `cell`, `cellarea`, `datamap`, `report` (queue `update_incoming`), or `wifi`. Omitted for task queues.

task

`task` is a timer that measures how long each Celery task takes. Celery tasks are used to implement the data pipeline and monitoring tasks.

Tags:

- `task`: The task name, such as `data.export_reports` or `data.update_statcounter`

Rate Control Metrics

The optional *rate controller* can be used to dynamically set the global locate sample rate and prevent the data queues from growing without bounds. There are several metrics emitted to monitor the rate controller.

rate_control.locate

`rate_control.locate` is a gauge that reports the current setting of the *global locate sample rate*, which may be unset (100.0), manually set, set by the rate controller, or set to 0 by the *transaction history monitor*.

rate_control.locate.target

`rate_control.locate.target` is a gauge that reports the current target queue size of the rate controller. It is emitted when the rate controller is enabled.

rate_control.locate.kp

`rate_control.locate.kp` is a gauge that reports the current value of K_p, the proportional gain. It is emitted when the rate controller is enabled.

rate_control.locate.ki

`rate_control.locate.ki` is a gauge that reports the current value of K_i, the integral gain. It is emitted when the rate controller is enabled.

rate_control.locate.kd

`rate_control.locate.kd` is a gauge that reports the current value of K_d, the derivative gain. It is emitted when the rate controller is enabled.

rate_control.locate.pterm

`rate_control.locate.pterm` is a gauge that reports the current value of of the proportional term of the rate controller. It is emitted when the rate controller is enabled.

rate_control.locate.iterm

`rate_control.locate.pterm` is a gauge that reports the current value of of the integral term of the rate controller. It is emitted when the rate controller is enabled.

rate_control.locate.dterm

`rate_control.locate.dterm` is a gauge that reports the current value of of the derivative term of the rate controller. It is emitted when the rate controller is enabled.

Transaction History Metrics

Processing the data queues can cause the MySQL InnoDB transaction history to grow faster than it can be purged. The transaction history length can be monitored, and when it exceeds a maximum, it can turn off observation processing until it is reduced. See *transaction history monitoring* for details.

Monitoring the transaction history length requires that the celery worker database connection ("read-write") has the `PROCESS` privilege. If the connection does not have this privilege, then no related metrics are emitted. If the connection has this privilege, then one or more metrics are emitted to monitor this process:

trx_history.length

`trx_history.length` is a gauge that reports the current length of the InnoDB transaction history.

trx_history.purging

If the rate controller is enabled, then `trx_history.purging` is a gauge that becomes 1 when the transaction history exceeds the maximum level. Observation processing is paused by setting the *global locate sample rate* to 0%, which allow the MySQL purge process to reduce the transaction history. When it drops below a safe minimum level, the rate is allowed to rise again.

If the rate controller is not enabled, then purging mode is not used, and this metric is not emitted.

trx_history.max

`trx_history.max` is a gauge that report the current maximum value for the transaction history length before the system switches to purging mode.

If the rate controller is not enabled, then purging mode is not used, and this metric is not emitted.

trx_history.min

`trx_history.min` is a gauge that report the current minimum value for the transaction history length before the system switches out of purging mode.

If the rate controller is not enabled, then purging mode is not used, and this metric is not emitted.

Datamaps Structured Log

The datamap script generates a data map from the gathered observations. It does not emit metrics.

The final `canonical-log-line` log entry has this data:

- `bucket_name`: The name of the S3 bucket
- `concurrency`: The number of concurrent threads used
- `create`: True if `--create` was set to generate tiles
- `csv_converted_count`: How many CSV files were converted to quadtrees
- `csv_count`: How many CSV files were exported from the database
- `duration_s`: How long in seconds to run the script
- `export_duration_s`: How long in seconds to export from tables to CSV
- `intermediate_quadtree_count`: How many partial quadtrees were created (due to multiple CSVs exported from large tables) and merged into one per-table quadtree
- `merge_duration_s`: How long in seconds to merge the per-table quadtrees
- `quadtree_count`: How many per-table quadtrees were generated
- `quadtree_duration_s`: How long in seconds to convert CSV to quadtrees
- `render_duration_s`: How long in seconds to render the merged quadtree to tiles
- `row_count`: The number of rows across datamap tables
- `script_name`: The name of the script (`ichnaea.scripts.datamap`)
- `success`: True if the script completed without errors
- `sync_duration_s`: How long in seconds it took to upload tiles to S3
- `tile_changed`: How many existing S3 tiles were updated
- `tile_count`: The total number of tiles generated
- `tile_deleted`: How many existing S3 tiles were deleted
- `tile_failed`: How many upload or deletion failures
- `tile_new`: How many new tiles were uploaded to S3
- `tile_unchanged`: How many tiles were the same as the S3 tiles
- `upload`: True if `--upload` was set to upload / sync tiles

Much of this data is also found in the file `tiles/data.json` in the S3 bucket for the most recent run.

Implementation

Ichnaea emits statsd-compatible metrics using markus, if the `STATSD_HOST` is configured (see *the config section*). Metrics use the the tags extension, which add queryable dimensions to the metrics. In development, the metrics are displayed with the logs. In production, the metrics are stored in an InfluxDB database, and can be displayed as graphs with Grafana.

Ichnaea also emits structured logs using structlog. In development, these are displayed in a human-friendly format. In production, they use the MozLog JSON format, and the data is stored in BigQuery.

In the past, metrics were the main source of runtime data, and tags were used to segment the metrics and provide insights. However, metric tags and their values were limited to avoid performance issues. InfluxDB and other time-series databases store metrics by the indexed series of tag values. This performs well when tags have a small number of unique values, and the combinations of tags are limited. When tags have many unique values and are combined, the number of possible series can explode and cause storage and performance issues (the "high cardinality" problem).

Metric tag values are limited to avoid high cardinality issues. For example, rather than storing the number of WiFi stations, the `wifi` tag of the *locate.query* metric has the values `none`, `one`, and `many`. The region, such as `US` or `DE`, was once stored as a tag, but this can have almost 250 values, causing MLS to have the highest processing load across Mozilla projects.

BigQuery easily handles high-cardinality data, so structured logs can contain precise values, such as the actual number of WiFi stations provided, and more items, such as the region and unexpected keys. On the other hand, there isn't a friendly tool like Grafana to quickly explore the data.

As of 2020, we are in the process of duplicating data from metrics into structured logging, expanding the data collected, and creating dashboards. We'll also remove data from metrics, first to reduce the current issues around high-cardinality, then to focus metrics on operational data. Structured data will be used for service analysis and monitoring of long-term trends, and dashboards created for reference.

1.2.8 Architecture

Overview

The application consists of an HTTP web service implementing the APIs and web site, and a streaming data pipeline.

Web Service

The web service uses the Python Pyramid web framework.

The web service serves several content views (home page, downloads page, maps page, and so on), serves the locate and submit API endpoints, and serves several monitoring endpoints.

The web service uses MySQL, but should function and respond to requests even if MySQL is down or unavailable.

Redis is used to track API key usage and unique IP addresses making service requests.

All API endpoints require a valid API key to use. The web service caches keys to reduce MySQL lookups.

Requests to locate API endpoints that only contain an IP address are fulfilled just by looking at the Maxmind GeoIP database without any MySQL lookups.

Requests to locate API endpoints that contain additional network information are fulfilled by using *location providers*. These are responsible for matching the data against the MySQL tables and generate possible result values and corresponding data quality/trustworthiness scores.

Some API keys allow falling back to an external web service if the best internal result does not match the expected accuracy/precision of the incoming query. In those cases an additional HTTPS request is made to an external service and that result is considered as a possible result in addition to the internal ones.

The system only deals with probabilities, fuzzy matches, and has to consider multiple plausible results for each incoming query. The data in the database will always represent a view of the world which is outdated, compared to the changes in the real world.

Should the service be able to generate a good enough answer, this is sent back as a response. The incoming query and this answer are also added to a queue, to be picked up by the data pipeline later. This query based data is used to validate and invalidate the database contents and estimate the position of previously unknown networks as to be near the already known networks.

Data pipeline

The data pipeline uses the Python Celery framework, its Celery scheduler and custom logic based on Redis.

The Celery scheduler schedules recurring tasks that transform and move data through the pipeline. These tasks process data in batches stored in custom Redis Queues implemented as Redis lists. Celery tasks themselves don't contain any data payload, but instead act as triggers to process the seperate queues.

Things to note:

1. The pipeline makes no at-most or at-least once delivery guaruantees, but is based on a best-effort approach.
2. Most of the data is being sent to the service repeatedly and missing some small percentage of overall data doesn't negatively impact the data quality.
3. A small amount of duplicate data is processed which won't negatively impact the data qualtiy.

1.2.9 Data flows

- *Position data*
- *API keys*
- *Export config*
- *Stat data*
- *Datamap data*

Position data

Position data is stored in the database in shard tables:

- `blue_shard_*`: hex 0 through f
- `cell_*`: `area`, `gsm`, `lte`, and `wcdma`
- `wifi_shard_*`: hex 0 through f

This data is created and updated from incoming API requests.

Data flow:

1. User submits data to one of the submit API endpoints.

   If the user used an api key and the key is sampling submissions, then the submission might get dropped at this point.

   OR

   User submits query to one of the locate API endpoints.

   If the query is handled by `InternalPositionSource`, then the web frontend adds a submission.

2. If the submission is kept, then the web frontend adds an item to the `update_incoming` queue in Redis.

   The item looks like a:

   ```
   {"api_key": key, "report": report, "source": source}
   ```

   "source" can be one of:

   - `gnss`: Global Navigation Satellite System based data
   - `fused`: position data obtained from a combination of other sensors or outside service queries
   - `fixed`: outside knowledge about the true position of the station
   - `query`: position estimate based on query data

3. The Celery scheduler schedules the `update_incoming` task every X seconds–see task definition in ichnaea/data/tasks.py.

4. A Celery worker executes the `update_incoming` task.

   This task acts as a multiplexer and its behavior depends on the `export_config` database table table.

   The `update_incoming` task will store data in multiple Redis lists depending on the `export_config` table. For example, it could store it in `queue_export_internal` and one more `queue_export_*` for each export target. These targets all have different batch intervals, so data is duplicated at this point.

   The task checks the length and last processing time of each queue and schedules an `export_reports` task if the queue is ready for processing.

5. A Celery worker executes `export_reports` tasks:

   - `dummy`:

     The dummy export does nothing–it's a no-op.

   - `geosubmit`:

     The geosubmit export sends the data as JSON to some HTTP endpoint. This can be used to submit data to other systems.

   - `s3`:

     The s3 export generates a gzipped JSON file of the exports and uploads to a configured AWS S3 bucket.

   - `internal`:

     The internal processing job splits the reports into its parts, creating one observation per network and queues them into multiple queues, one queue corresponding to one database table shard.

     This data ends up in Redis queues like `update_cell_gsm`, `update_cell_wcdma`, `update_wifi_0`, `update_wifi_1`, etc to be processed by `station updater` tasks.

     The internal processing job also fills the `update_datamap` queue, to update the coverage data map on the web site.

6. The Celery worker executes `station updater` tasks.

   These tasks take in the new batch of observations and match them against known data. As a result, network positions can be modified, new networks can be added, and old networks be marked as blocked noting that they've recently moved from their old position. See *Observations* for details.

API keys

API keys are stored in the `api_key` table.

API keys are created, updated, and deleted by an admin.

Export config

Export configuration is stored in the `export_config` table.

Export configuration is created, updated, and deleted by an admin.

Stat data

Stat data is stored in the `stat` and `region_stat` tables.

FIXME: data flow for stat data?

Datamap data

Datamap data is stored in the `datamap_*` (`ne`, `nw`, `se`, `sw`) tables. The north / south split is at 36N latitude, and the east / west split at 5 longitude, to attempt to split the data into four equal shards. The rows divide the world into boxes that are one-thousandth of a degree on each side (about 112 meters at the equator), and record the first and latest time that an observation was seen in that box. Other details, including the exact position of the observation, are not recorded. These tables are updated during the `internal` processing job.

A periodic task runs the script `ichnaea/scripts/datamap.py` to convert this data into transparent tile images for the contribution map. Thread pools are used to distribute the work across available processors. The process is:

1. Export the datamap tables as CSV files.

   The latitude, longitude, and days since last observation are fed into a randomizer that creates 0 to 13 nearby points, more for the recently observed grid positions. This emulates the multiple observations that go into each grid position, and hides details of observations for increased privacy.

2. Convert the CSV files to a quadtree structure.

   The binary quadtree structure efficiently stores points when there are large areas with no points, and is faster for determining points within the bounding box of a tile.

3. Merge the per-table quadtrees to a single quadtree file.

   This includes removing duplicates at the boundaries of tables.

4. Generate and minimize tiles for the different zoom levels.

   Each zoom level potentially has four times the tiles of the previous zoom level, with 1 at zoom level 0, 4 at zoom level 1, 16 at zoom level 2, up to over 4 million at maximum zoom level 11. However, tiles with no observations are not rendered, so the actual number of generated tiles is less. The tiles are stored in a folder structure by zoom level, x position, and files at the y position, to match Mapbox tile standards and to avoid having too many files in a folder.

Tiles are further optimized for disk space by reducing the colorspace, without reducing quality below a target.

A double-resolution tile at zoom level 0 is created for the map overview on the front page on high-resolution displays.

5. Upload the tiles to an S3 bucket.

   There may be existing tiles in the S3 bucket from previous uploads. The script collects the size and MD5 hash of existing S3 tiles, and compares them to the newly generated tiles, to determine which are new, which are updated, which are the same an can be ignored, and which S3 tiles should be deleted.

   New and updated tiles are uploaded. Uploading is I/O bound, so the concurrency of uploads is doubled. The deleted tiles are deleted in batches, for speed.

   A file `tiles/data.json` is written to record when the upload completed and details of the tile generation process.

Quadtree and tile generation tools are provided by ericfischer/datamaps, and PNG size optimization by pngquant.

1.2.10 Data import and export

Data export

Ichnaea supports automatic, periodic CSV (comma separated values) export of aggregate cell data (position estimates). Mozilla's exports are available at https://location.services.mozilla.com/downloads

The data exchange format was created in collaboration with the *OpenCellID* project.

Records should be written one record to a line with CRLF (0x0D 0x0A) as line separator.

A value should be written as an empty field—two adjacent commas, for example—rather than being omitted.

The first five fields (radio to cell) jointly identify a unique logical cell network. The remaining fields contain information about this network.

The data format does not specify the means and exact algorithms by which the position estimate or range calculation was done. The algorithms might be unique and changing for each source of the data, though both Ichnaea and *OpenCellID* currently use similar and comparable techniques.

Cell Fields

The fields in the CSV file are as follows:

radio Network type. One of the strings `GSM`, `UMTS` (for WCMDA networks), or `LTE`.

mcc Mobile Country Code. This is an integer.

For example, `505` is the code for Australia.

net For GSM, UMTS, and LTE networks, this is the mobile network code (MNC). This is an integer.

For example, `4` is the MNC used by Vodaphone in the Netherlands.

area For GSM and UMTS networks, this is the location area code (LAC). For LTE networks, this is the tracking area code (TAC). This is an integer.

For example, `2035`.

cell For GSM and LTE networks, this is the cell id or cell identity (CID). For UMTS networks this is the UTRAN cell id, which is the concatenation of 2 bytes of radio network controller (RNC) code and 2 bytes of cell id. This is an integer.

For example, `32345`.

unit For UMTS networks, this is the primary scrambling code (PSC). For LTE networks, this is the physical cell id (PCI). For GSM networks, this is empty. This is an integer.

For example, `312`.

lon Longitude in degrees between -180.0 and 180.0 using the WGS 84 reference system. This is a floating point number.

For example, `52.3456789`.

lat Latitude in degrees between -90.0 and 90.0 using the WGS 84 reference system. This is a floating point number.

For example, `-10.034`.

range Estimate of radio range, in meters. This is an estimate on how large each cell area is, as a radius around the estimated position and is based on the *observations* or a knowledgeable source. This is an integer.

For example, `2500`.

samples Total number of *observations* used to calculate the estimated position, range and averageSignal. This is an integer.

For example, `1200`.

changeable Whether or not this cell is a position estimate based on *observations* subject to change in the future or an exact location entered from a knowledgeable source. This is a boolean value, encoded as either `1` (for "changeable") or `0` (for "exact").

created Timestamp of the time when this record was first created. This is an integer counting seconds since the UTC Unix Epoch of 1970-01-01T00:00:00Z.

For example, `1406204196` which is the timestamp for 2014-07-24T12:16:36Z.

updated Timestamp of the time when this record was most recently modified. This is an integer, counting seconds since the UTC Unix Epoch of 1970-01-01T00:00:00Z.

For example, `1406204196`, which is the timestamp for 2014-07-24T12:16:36Z.

averageSignal Average signal strength from all observations for the cell network. This is an integer value, in dBm.

For example, `-72`.

This field is only used by the *OpenCellID* project and has been used historically as a hint towards the quality of the position estimate.

Data import

Aggregate cell data can be imported into an Ichnaea instance. For the development environment:

1. Download a Differental Cell Export from Mozilla's Download page. Do not extract it, but keep it in the compressed `.csv.gz` format, in the root of the repository.
2. In a shell in the app container, import the data:

```
$ make shell
# Replace with the filename of the downloaded export file
app@blahblahblah:/app$ ichnaea/scripts/load_cell_data.py MLS-diff-cell-export-YYYY-
↪MM-DDTHH0000.csv.gz
```

This will import the cell data, then queue tasks to aggregrate cell areas and region statistics. It should take about a minute to process an 300kB export of 10,000 stations.

Importing a Full Cell Export is not recommended. This will fail due to unexpected data, and the development environment may require undocumented changes for the larger resource requirements of a full cell export.

1.3 Algorithms

The project uses a couple of different approaches and algorithms.

There's two general approaches to calculate positions from signal sources, without the cooperation of the signal sources or mobile networks.

1. Determine the location of signal sources from *observations* and then compare / trilaterate user locations.
2. Generate signal fingerprints for a fine-grained grid of the world. Find the best match for an observed fingerprint.

The second approach has much better accuracy, but relies on more available and constantly updated data. For most of the world this approach is not practical, so we currently focus on the first approach.

In theory one could use signal strength data to infer a distance measure: the further a device is away from the signal source, the weaker the signal should get.

Unfortunately the signal strength is more dependent on the device type, how a user holds a device, and changing environmental factors like trucks in the way. Even worse, modern networks adjust their signal strength to the number of devices inside their reception area. This makes this data highly unreliable while looking up a user's position via a single reading.

In aggregate, over many data points this information can still be valuable in determining the actual position of the signal source. While observing multiple signals at once, their relative strengths can also be used, as this keeps some of the changing factors constant like the device type.

One other approach is using time of flight data as a distance metric. While there are some reflection and multipath problems, it's a much more accurate distance predictor. Fine grained enough timing data is unfortunately almost never available to the application or operating system layer in client devices. Some LTE networks and really modern WiFi networks with support for 802.11v are the rare exception to this. These are so rare that we currently ignore timing data.

1.3.1 Accuracy

Depending on the signal standard, we can promise different levels of accuracy.

Underlying this is the assumption that we have enough data about the area. Without enough data, Ichnaea will fall back to less accurate data sources depending on the configuration.

Bluetooth is the most accurate, followed by WiFi, and than cell based estimation using single cells, multiple cells, or cell location areas. GeoIP serves as a general fallback.

Bluetooth / WiFi

Bluetooth and WiFi networks have a fairly limited range. Bluetooth low-energy beacons typically reach just a couple meters and WiFi networks reach up to 100 meters. With obstacles like walls and people in the way, these distances get even lower.

However, this data can be skewed when the device in question is moving. It takes some time to do a network scan and devices tend to cache this information heavily. There can be a time delta of tens of seconds between when a network was actually seen and when it is reported to the application layer. With a fast moving device this can lead to inaccuracies of a couple kilometers. WiFi networks tend to show up in scans long after they are out of reach, especially if the the device was actually connected to these networks.

This means position estimates based on WiFi networks are usually accurate to 100 meters. If a lot of networks are available in the area, accuracy tends to increase to about 10 or 20 meters. Bluetooth networks tend to be accurate to about 10 meters.

One difficult challenge with Bluetooth and WiFi networks are the constantly moving networks. For example, WiFi networks installed on buses or trains or in the form of hotspot-enabled mobile phones or tablets. Detecting movement and inconsistencies between observed data and the database world view are important.

GSM Cells

In GSM networks, one typically only has access to the unique cell id of the serving cell. In GSM networks, the phone does not know the full cell ids of any neighboring cells unless it associates with the new cell as part of a hand-over and forgets the cell id of the old cell.

So we're limited to a basic *Cell-ID* approach where we assume that the user is at the center of the current GSM cell area and we use the cell radius as the accuracy.

GSM cells are restricted to a maximum range of 35km, but there are rare exceptions using the GSM extended range of 120km.

In more populated places the cell sizes are typically much smaller, but accuracy will be in the range of tens of kilometers.

WCDMA Cells

In WCDMA networks, neighboring cell information can be available. However, limitations in chipset drivers, the radio interface layer, and the operating systems often hide this data from application code or only partially expose the cell identifiers. For example, they might only expose the carrier and primary scrambling code of the neighboring cells.

In most cases we are limited to the same approach as for GSM cells. In urban areas, the typical sizes of WCDMA cells are much smaller than GSM cells. This leads to improved accuracy in the range of 1 to 10 kilometers. However in rural areas, WCDMA cells can be larger than GSM cells, sometimes as large as 60 to 70 kilometers.

LTE Cells

LTE networks are similar to WCDMA networks and the same restrictions on neighboring cells applies. Instead of a primary scrambling code, LTE uses a physical cell id which for our purposes has similar characteristics.

LTE cells are often smaller than WCDMA cells which leads to better accuracies.

LTE networks also expose a time-based distance metric in the form of the timing advance. While we currently don't use this information, it has the potential to significantly improve position estimates based on multiple cells.

GeoIP

The accuracy of GeoIP depends on the region the user is in. In the US, GeoIP can be fairly accurate and often places the user in the right city or metropolitan area. In many other parts of the world, GeoIP is only accurate to the region level.

Typical GeoIP accuracies are either in the 25 km range for city based estimates or multiple hundred kilometers for region based estimates.

IPv6 has the chance to improve this situation, as the need for private carrier networks and network address translation decreases. So far this hasn't made any measurable impact and most traffic is still restricted to IPv4.

1.3.2 Observations

Ichnaea builds its database of Bluetooth, WiFi, and Cell stations based on observations of those stations. The goal is not to identify the position of the station, but to identify where the station is likely to be observed.

Depending on the observation quality, the station updating algorithm can confirm a station is active, adjust the position estimate, block it temporarily from location queries, or remove historic position data completely.

- *The Station Model*
- *Modeling WiFi and Cell Stations*
- *Sources and Batching*
- *Observation Weight*
- *Blocked Stations*
- *Updating Stations*
- *Weight Algorithm Details*

The Station Model

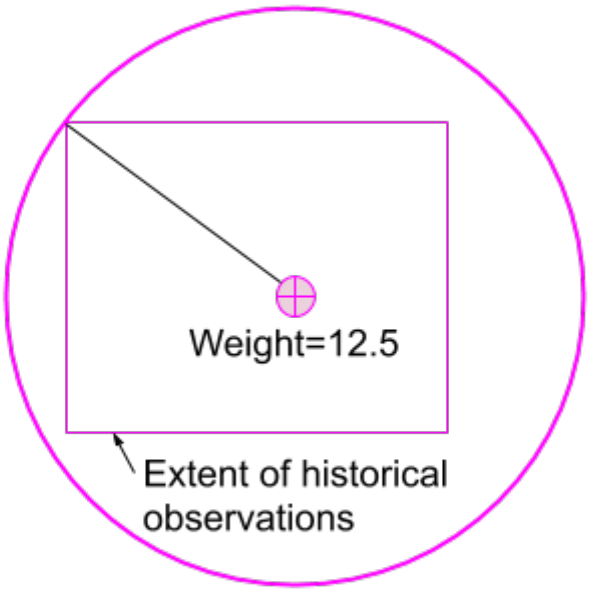
A station, regardless of type, is modeled as a circle, where the center is the weighted average position of observations, and the radius is large enough to contain historical observations.

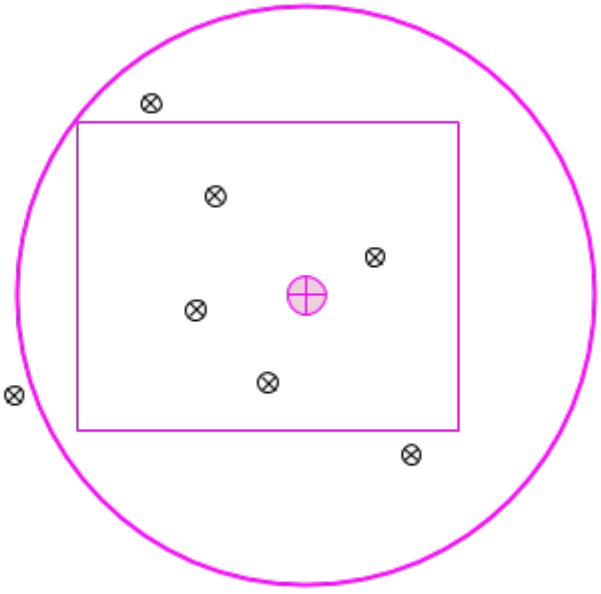
New observations are matched to the existing model. If the observations match the model type (for example, GPS submissions for a GPS-based model), then they can update the model's position and radius.

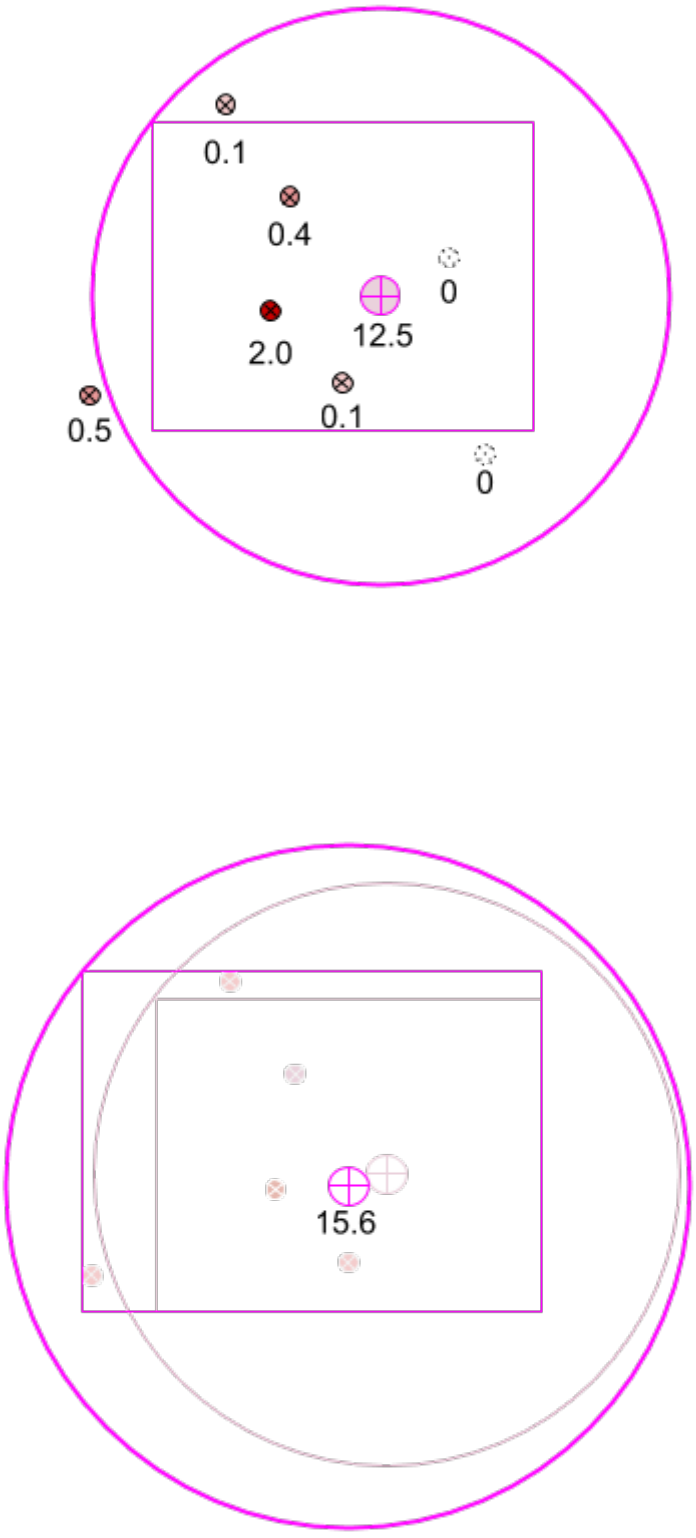
The new observations are weighted by features like accuracy, age, speed, and signal strength. Some observations have a weight of 0, and are discarded.

The station's position is adjusted by the new observations, and the station weight is increased. Stations with many observations have a large weight, so new observations have a diminishing impact on the station position.

To complete the update, the observation bounding box is expanded, if needed, to enclose the new observations. The radius is adjusted for the new center and bounding box.


Weight=12.5
Extent of historical
observations

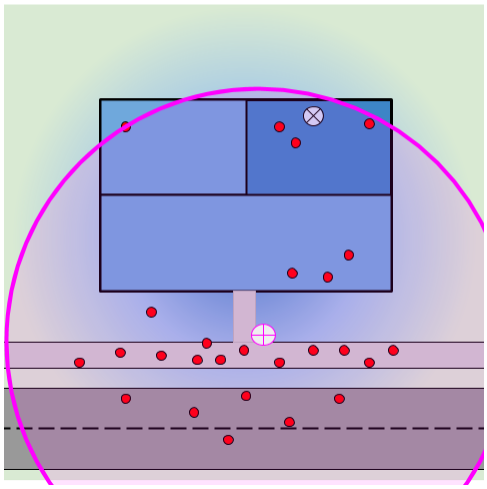



0.1
0.4
0
12.5
2.0
0.1
0.5
0
15.6

Modeling WiFi and Cell Stations

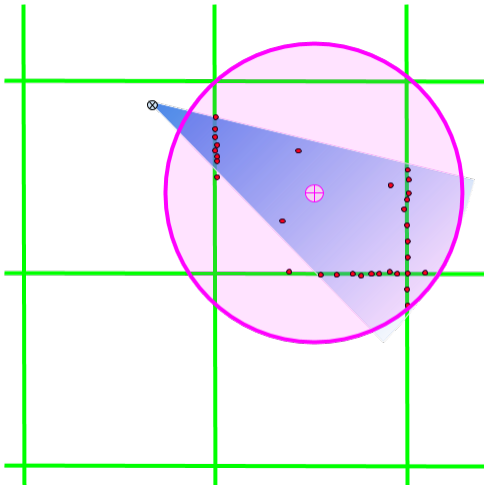
The station model tracks where the station is observed, and does not attempt to determine where the emitter is located.

For example, a WiFi router may physically be located inside a building, to maximize the signal for the people in the building. However, people on the sidewalk or road outside the building are more likely to observe the WiFi router at the same time they have a good GPS or other GNSS position lock. The WiFi station model will be weighted toward the outside observations, and may show a position outside of the building.

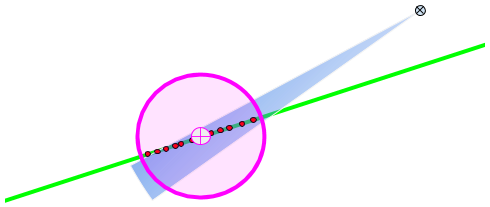


Cell signals are directional, transmitting in an arc or wedge rather than in all directions. They are often observed by phones in vehicles, such as when following directions from a map application.

In a city, the station model will encompass the service area, biased toward observations on roads. The cell emitter will often be outside of the model radius.



Outside of cities, a cell tower often covers a large area, and individual cell signals are broadcast in narrow wedges. The observations may be a large distance away from the emitter, along cross-country roads. The station model is often centered on these roads, and the cell signal source is well outside of the radius.



Sources and Batching

Observations come from two sources:

Location queries The device sends the detected radio sources, and Ichnaea returns a position estimate or region based on known stations and the requester's IP address. This data is used to discover new stations, and to confirm that known stations are still active.

Submission reports The device sends the detected radio stations, along with a position, which is usually derived from high-precision satellite data such as GPS. These reports are used to determine the position of newly discovered stations, or to refine the position estimates of known stations.

The *data flow process* creates observations by pairing the position data with each station, and then adds the observations to update queues based on the database sharding. Cell stations are split by radio type, and the observations are added to queues like `update_cell_gsm` and `update_cell_wcdma`. Bluetooth and WiFi stations are split into 16 groups by the first hexadecimal letter of the identifier, and the observations are added to queues like `update_wifi_0` and `update_blue_a`.

These per-shard queues are processed when a large enough batch is accumulated, or when the queue is about to expire. Batching increases the chances that there will be several observations for a station processed in the same chunk. It also increases the chance that two station updating threads will try to update the same station. This may cause timeouts or deadlocks due to lock contention, and is tracked with the metric `data.station.dberror`.

A station is either based on observations from location queries (with estimated positions from Ichnaea), or from observations from submission reports (with positions from GPS or similar sources). When a station built from location queries has a valid observation from a submission report, the station is upgraded by discarding the existing position estimate and using the submitted, satellite-backed position (see the *Replace* transition state in the *Updating Stations* section below).

Observation Weight

Each observation is assigned a weight, to determine how much it should contribute to the station position estimate, or if it should be discarded completely. The observation weight is based on four metrics:

Accuracy Expected distance from the reported position to the actual position, in meters.

Age The time from when the radio was seen until the position was recorded, in seconds. The age can be negative for observations after the position was recorded.

Speed The speed of the device when the position was recorded, in meters per second.

Signal The strength of the radio signal, in dBm (decibel milliwatts).

The observation weight is the product of four weights:

(accuracy weight) x (age weight) x (speed weight) x (signal weight)

The first three weights range from 0.0 to 1.0. If the accuracy radius is too large (200m for WiFi), the age is too long ago (20 seconds), or the device is moving too quickly (50m/s), the weight is 0.0 and the observation is discarded. If the accuracy distance is small (10m or less), the age is very recent (2s or less), and the device is moving slowly (5m/s or less), then the weight is 1.0.

The signal weight for cell and WiFi stations is 1.0 for the average signal strength (-80 dBm for WiFi, -105 dBm to -95 dBm for different cell generations), grows exponentially for stronger signals, and drops exponentially for weaker signals. It never reaches 0.0, so signal strength does not disqualify an observation in the same way as accuracy, age, or speed. For bluetooth stations, the signal weight is always 1.0.

When accuracy, age, speed, or signal strength is unknown, the weight for that factor is 1.0.

An observation weight of 0.0 disqualifies that observation. An average observation should have a weight of 1.0. Weights are used when averaging observation positions, and when adjusting the position of an existing station. Existing stations store the sum of weights of previous observations, so that new observations have a smaller influence on position over time.

For more information, see *Weight Algorithm Details*.

Blocked Stations

Only stationary cell, WiFi, and Bluetooth stations should be considered when estimating a position for a location query. Mobile stations are identified by observations that are well outside the expected range of the station type. Ichnaea keeps track of these as blocked stations, and uses observations to keep them blocked or move them back to regular stations.

When a station is blocked, it remains blocked for 48 hours. This temporary block is used to handle a usually stationary station that is moved, such as a WiFi access point that moves to a new location.

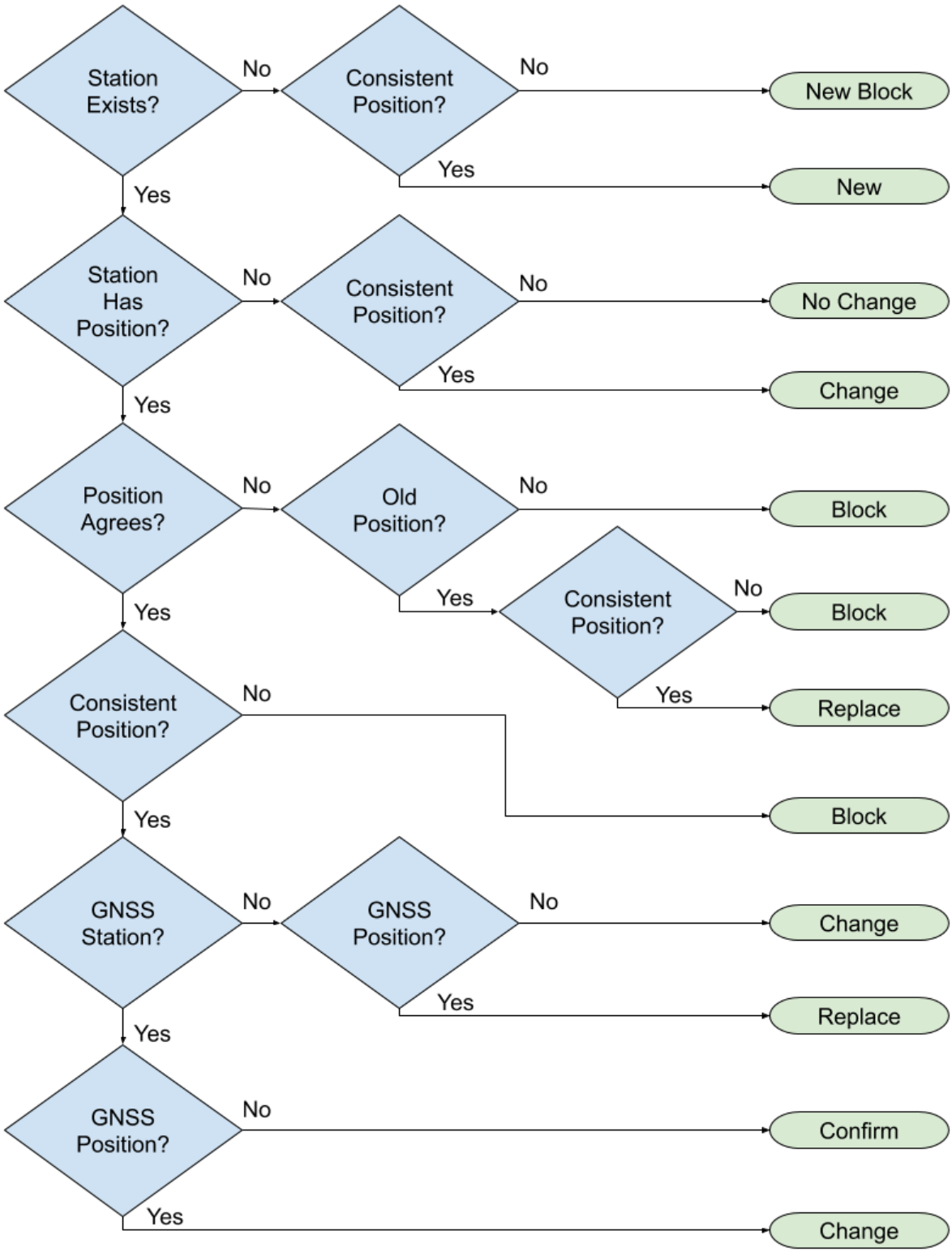
A station's block count is tracked, and compared to how long the station has been tracked. If a station has been blocked more times than its age in 30-day "months", then it is considered a mobile station and remains in a long-term block. For example, if a station tracked for a year has been blocked 12 times or more, it remains in a long-term block.

Observations for blocked stations are added to the daily observation count, but are not processed to update the station. Blocked stations do not store a position estimate, but retain a region if they once had a position estimate, and can still be used for region queries.

Updating Stations

The observations (with non-zero weights) for a station are processed as a group, to determine how the station should be updated. If there are valid GPS-based observations, only those are used, discarding any observations based on location queries.

If an existing station is still blocked, then it remains blocked. For unblocked stations, here is the decision process for determining what the "transition state", or update type, should be:



Several yes-or-no facts are used to determine the update type:

- *Station Exists?* - Is there a record for this station in the database?
- *Consistent Position?* - Are multiple observations close enough that they could be observing the same stationary

station, or are they spread out enough that they could be observing different stations or a moving station? The "close enough" radius changes based on the type of station (100m for Bluetooth, 5km for WiFi, and 100km for cell stations).

- *Station Has Position?* - Does the station have a position estimate in the database?
- *Position Agrees?* - Does the station position agree with the observations, or do the observations suggest the station has moved?
- *Old Position?* - Has the station's position not been confirmed for over a year?
- *GNSS Station?* - Is the station's position based on Global Navigation Satellite System data, such as GPS?
- *GNSS Position?* - Is the observation based on a GNSS position submission, rather than a location query?

These are used to determine a transition state:

- *No Change* - No change is made to the station
- *New* - A new station is added to the database.
- *New Block* - A new blocked station is added to the database.
- *Change* - An existing station's position is adjusted, based on the weighted average of the current position and the observations.
- *Confirm* - An existing station is confirmed to still be active today. Stations that were already confirmed today are unchanged.
- *Replace* - A station's position is replaced with the observation position
- *Block* - A station's position is removed, and it is blocked from being used for location queries

Related cell stations are grouped into a *cell area*. These can be used for location queries, when a particular cell station is unknown but others in the cell area group are known. If a cell station is created or has an updated position (all transition states but *No Change* or *Confirm*), then the cell area is added to a queue *update_cellarea*, and processed when enough cell areas are accumulated.

Metrics are collected based on the update type. There is a daily count of observations, and a count of newly tracked stations, both by radio type, stored in Redis. There are four statsd counters as well:

- `data.observation.insert` - Counts all observations with a non-zero weight, including those observing a blocked station
- `data.station.blocklist` - Counts new stations that start blocked (*New Block*) and stations converted to blocked (*Block*)
- `data.station.confirm` - Counts existing stations confirmed to still be active (*Confirm*)
- `data.station.new` - Counts new stations added, either as blocked stations (*New Block*), or non-blocked stations (*New*)

Weight Algorithm Details

The observation weight is the product of four weights:

(accuracy weight) x (age weight) x (speed weight) x (signal weight)

The accuracy, age, and speed weights use the same algorithm, with these features:

- The weight is 1.0 if the metric is small enough (at or below **MIN**), fully weighting the observation. If the metric is unknown, the weight is also 1.0.
- The weight is 0.0 if the metric is too large (at or above **MAX**), rejecting the observation.

- The weight drops logarithmically from 1.0 if the metric is between **MIN** and **MAX**.

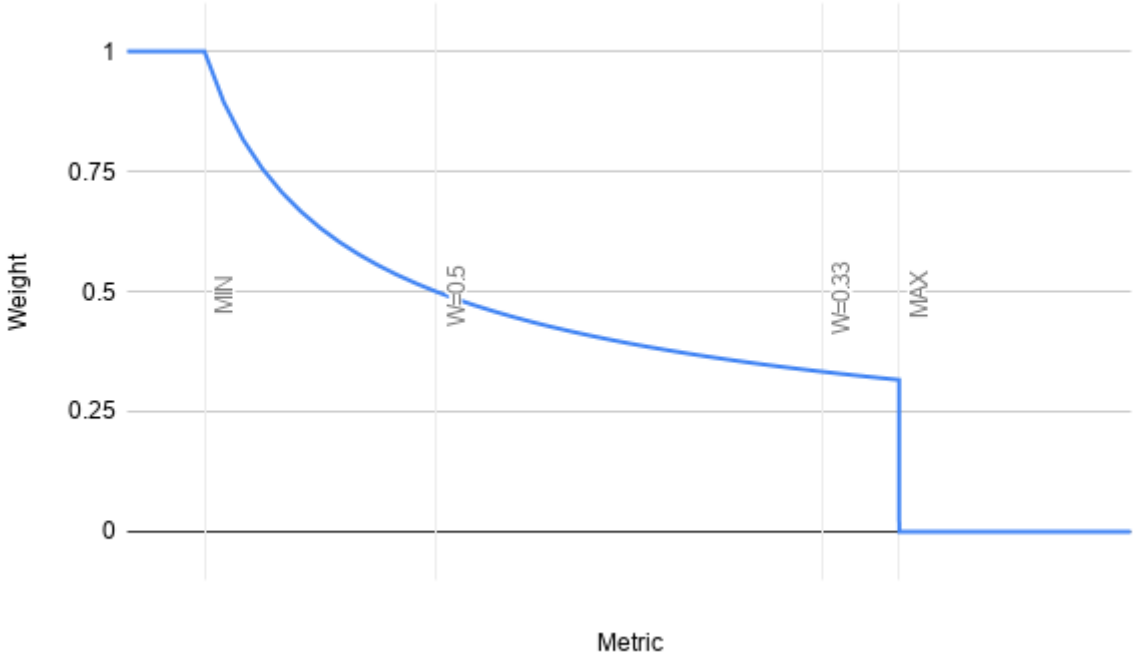


Fig. 1: The weight curve for qualifying metrics

Metric	MIN, Weight=1.0	Weight=0.5	Weight=0.33	MAX, Weight=0.0
Accuracy	10 m	40 m	90 m	100 m (Bluetooth) 200 m (WiFi) 1000 m (Cell)
Age	2 s	8 s	18 s	20 s
Speed	5 m/s	20 m/s	45 m/s	50 m/s

The signal weight algorithm varies by radio type. The signal weight is always 1.0 for Bluetooth. For WiFi and Cell radios, the weight is 1.0 for the average signal, and grows exponentially as the signal gets stronger.

Here are the signal strengths for interesting weights:

Radio	Weight=0.5	Weight=1.0 (Avg)	Weight=2.0	Weight=4.0
WiFi	-98.9 dBm	-80 dBm	-64.1 dBm	-50.7 dBm
GSM	-113.9 dBm	-95 dBm	-79.1 dBm	-65.7 dBm
WCDMA	-118.9 dBm	-100 dBm	-84.1 dBm	-70.7 dBm
LTE	-123.9 dBm	-105 dBm	-89.1 dBm	-75.7 dBm

If the signal strength is unknown, a signal weight of 1.0 is used.

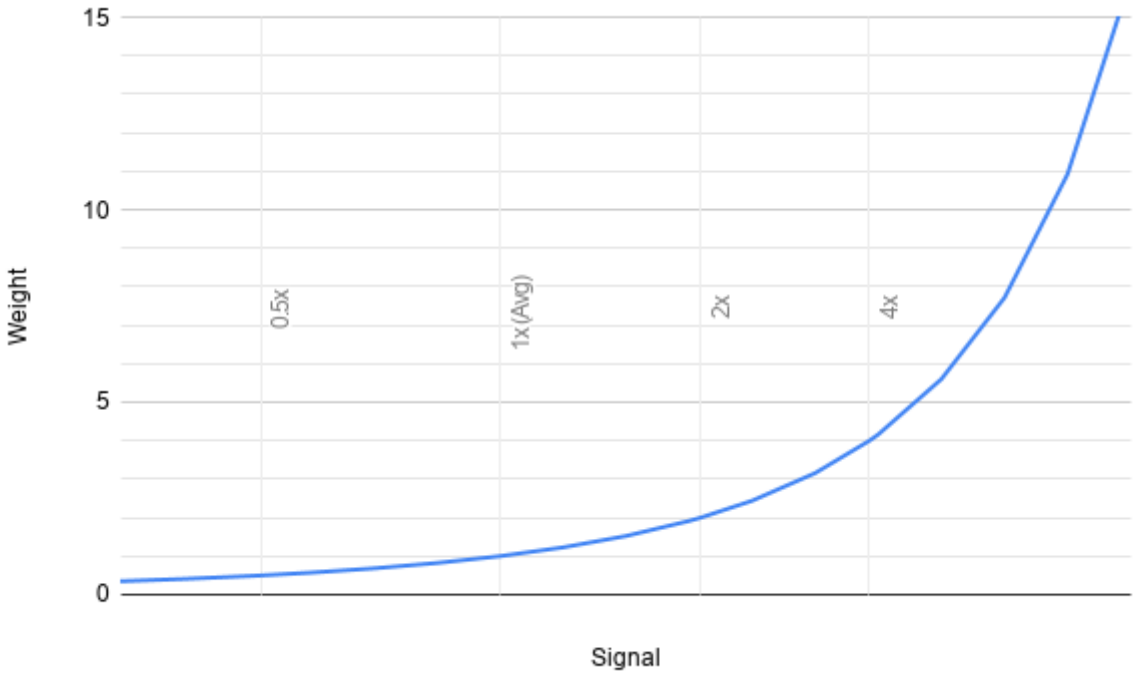


Fig. 2: The weight curve for signal strength

1.4 Changelog

1.4.1 Changelogs

Changelog

See the tags page for releases after November 2019.

See the releases page for releases from August 2017 to August 2019.

2.2.0 (2017-08-23)

Migrations

- a0ee5e10f44b: Remove allow_transfer column from API key table.
- 138cb0d71dfb: Drop cell OCID tables.
- 5797389a3842: Add fallback_schema column to API key table.
- 30a4df7eafd5: Add allow_region column to API key table.
- 73c5f5ae5b23: Drop shortname column from API key table.

Changes

- Remove unfinished transfer HTTP API.
- #508: Add support for unwiredlabs as a fallback provider.
- Combine rate limit and unique IP counting into one Redis call.
- Add new cleanup stat task.
- Remove the monitor queue size task.
- Check API keys for region requests.
- Avoid filling the datamap queues if the web content is disabled.
- Internal optmization in SQL export queries.

2.1.0 (2017-06-27)

Compatibility

- Move back to Celery 3.
- Drop support for Python 2.7, require Python 3.6.

Changes

- Rely on *cleanup_datamap* task to remove old datamap entries.
- Use mysql-connector for datamap and local dump script.
- Remove tabzilla, update web site style.
- Add Zilla Slab font files, remove non-woff fonts.
- Replace custom base map with *mapbox.dark*.
- Update CSS/JS dependencies.
- Replace bower in CSS/JS dev setup with npm.
- Install MySQL 5.7 and Redis command line utilities.
- Remove radio field workaround in cell locate API.
- Adjust the text on the download and stats pages.
- Use SQLAlchemy core instead of ORM in various places.

2.0 (2017-03-22)

Compatibility

- Application configuration moved to environment variables.
- Moved initial database schema creation into an alembic migration.
- Test against Redis 3.2 instead of 2.8.
- Test against MySQL 5.7 instead of 5.6.
- No longer create *lbcheck* database user in *location_initdb* script.
- Drop support for Python 2.6.

Migrations

- d2d9ecb12edc: Add modified index on *datamap_** tables.
- cad2875fd8cb: Add *store_sample_** columns to api_key table.
- Removed old migrations. The database needs to be at least at version *1bdf1028a085* or *385f842b2526* before upgrading to this version.

Changes

- #496: Don't store queries if all networks where seen today.
- #492: Add new datamap cleanup task to delete old datamap rows.
- Update to botocore/boto3.
- No longer use secondary cell tables during lookups.
- Remove continous cell import functionality.
- Relax GeoIP database check to allow *GeoLite2-City* databases.
- Update region specific statistics once per day.
- Add in-memory API key cache.
- Add */contribute.json* view.
- Update to Celery 4.
- Remove */leaders* HTTP redirects.
- Replace the */apps* page with a link to the Wiki.

Changelog 1.5

1.5 (unreleased)

Untagged

Migrations

Changes

- Add a workaround for a bug in the RTree library.

20160810110400

Migrations

- 385f842b2526: Add allow_transfer column to api_key table.

Changes

- Use a docker volume for the css/js build pipeline.

20160809152200

Changes

- Add a new *location_dump* command line script to dump/export networks.
- #491: Add an alternate homepage if the web content is disabled.
- Reduce datamap detail.
- No longer log sentry messages for client errors.
- Replace station data for old stations with new conflicting observations.
- Make statsd and sentry optional service dependencies.
- Disable the web site content by default.
- #490: Limit full cell export to cells which were modified in the last year.
- Use the accuracy_radius field from the GeoIP database.
- Remove ipf fallback from responses for queries based only on GeoIP data.

20160525130100

Migrations

- Remove the celery section from the config file, add the web section.

Changes

- Lower maximum accuracy values returned from locate queries.
- Accept observations with slightly worse accuracy values.
- The website content can be disabled via a setting in the config file.
- Make map related settings configurable via the config file.
- Use the first of the month, to display stats for the entire month.
- Calculate and display stats for today.
- Preserve or derive both WiFi channel and frequency values.

20160429092200

Changes

- Allow report submissions without any position data.
- #485: Fix development documentation.

20160427100000

Migrations

- 1bdf1028a085: Extend export config table.
- 6ec824122610: Add export config table.
- 4255b858a37e: Remove user/score tables.
- In service checks, change anything that checked the */__heartbeat__* view to check */__lbheartbeat__* instead. Change */__monitor__* to */__heartbeat__*.

Changes

- Be more explicit about closing socket connections.
- Use GNSS observations to replace purely query-based stations.
- Use query observations to confirm, blocklist and insert new stations.
- Configure release for raven/sentry client.
- Change hearbeat/monitor view to lbhearbeat/hearbeat views.
- Update last_seen column on each station update.

- Use Vincenty formula for lat/lon additions.
- Use Vincenty instead of Haversine formula for distance calculations.
- Take age into account during locate lookups.
- Filter out observations with too little weight.
- Take age and speed into account in observation weights.
- Pass queries into internal data pipeline.
- Allow stations to be blocklisted once per 30 days of their lifespan.
- Normalize age fields for internal observations to GPS time difference.
- Add stricter validation of radio, source and timestamp fields.
- Pass pressure and source data into internal data pipeline.
- Read export config from database instead of ini file.

20160412083700

Migrations

- 27400b0c8b42: Drop api_key log columns.
- 88d1704f1aef: Drop cell_ocid table.

Changes

- Remove intermediate schedule_export_reports task.
- #456: Retire old leaderboard.
- Remove intermediate upload_report task.

20160401185900

Changes

- Downgrade numpy to 1.10.4 due to build failures.

20160401110200

Migrations

- e23ba53ab89b: Add sharded OCID cell tables.
- fdd0b256cecc: Add fallback options to API key table.

Changes

- Tag location fallback metrics with the fallback name.
- #484: Allow per API key fallback configuration.
- Document and forward age argument through all layers of abstraction.
- Limit the columns loaded for API keys.
- Prevent errors when receiving invalid timestamps.

20160323102800

Changes

- #456: Deprecate weekly leaderboard.
- Remove the implied metadata setting from the config file.
- Enable extended metrics for all API keys.
- Speed up full cell export.
- Rename internal blue/wifi observation key to mac.
- Removed migrations before version 1.4.

Changelog 1.4

1.4 (2016-03-03)

20160303094100

Changes

- Deprecate hashkey and internaljson logic.
- Improve readability of downloads page.
- Restrict valid characters in API keys.
- Take signal strength into account for location queries.
- Decrease database session times in data tasks.
- Retry station updates on deadlocks and lock timeouts.
- Simplify and speed up InternalUploader.

20160218132400

Changes

- Display region specific BLE statistics.
- Remove geodude compatibility API again.
- Avoid intermediate Redis task round-trip.
- Queue data for up to 24 hours.
- Simplify colander schemata.
- Update dependencies.

20160202154300

Changes

- Avoid fancy syntax in build requirements.

20160202101600

Migrations

- 4b11500c9014: Add Bluetooth region stat.
- b247526b9501: Add sharded Bluetooth tables.
- 0987336d9d63: Add weight and last_seen columns to station tables.
- 44e1b53944ee: Remove old cell tables.

Changes

- #476: Emit basic request metrics for region API.
- Remove internal API key human readable metric names.
- Accept and use Bluetooth networks in public HTTP APIs.
- Weight observations by their accuracy and signal strength values.
- Add stricter validation of asu, signal and ta values.
- Restrict observations to maximum accepted accuracy values.
- Allow queries to the fallback source if the combined score is too low.
- #151: Choose best position result based on highest combined score.
- #481: Fix broken cell export.
- #151: Choose best region result based on highest combined score.
- #371: Extend region API to use wifi data.
- #371: Extend region API to use cell area data.

- #151: Choose best region result based on highest score.
- Remove migrations and tests for 1.2 to 1.3 upgrade.
- Enable *shapely.speedups* to speed up GeoJSON parsing.
- Ship buffered region file with the code.
- Stop forwarding client IP address to data pipeline.

20160112143200

Migrations

- 9743e7b8a17a: Add allow_locate column to API key table.
- 5d245a440c6f: Remove unused user email field.
- d350610e27e: Shard cell table.
- 40d609897296: Add sharded cell tables.
- The command line for starting gunicorn has changed. The *-c* option now needs a *python:* prefix and has to be *-c python:ichnaea.webapp.settings*.

Changes

- #478: Restrict some API keys from using the locate API.
- Register OCID import tasks based on the configuration file.
- #471: Remove sentence about Firefox OS.
- #477: Decrease cell maximum radius to 100 km.
- Use sharded cell tables.
- Keep separate rate limits per API version.
- Update to latest versions of dependencies.

20151118134500

Migrations

- 91fb41d12c5: Drop mapstat table.

Changes

- #469: Update to static tabzilla.
- #468: Add CORS headers and support OPTIONS requests.
- #467: Implement geodude compatibility API.
- #151: Choose best WiFi cluster based on a data quality score.
- Use up to top 10 WiFi networks in WiFi location.
- Use proper agglomerative clustering in WiFi clustering.
- Remove arithmetic/hamming distance analysis of BSSIDs.
- Accept and forward WiFi SSID's in public HTTP API's.

20151105120300

Migrations

- 78e6322b4d9: Copy mapstat data to sharded datamap tables.
- 4e8635b0f4cf: Add sharded datamap tables.

Changes

- Use new sharded datamap tables.
- Parallelize datamap CSV export, Quadtree generation and upload.
- Introduce upper bound for cell based accuracy numbers.
- Fix database lookup fallback in API key check.
- Switch randomness generator for data map, highlight more recent additions.
- Update to latest versions of lots of dependencies.

20151021143400

Migrations

- 450f02b5e1ca: Update cell_area regions.
- 582ef9419c6a: Add region stat table.
- 238aca86fe8d: Change cell_area primary key.
- 3fd11bfaca02: Drop api_key log column.
- 583a68296584: Drop old OCID cell/area tables.
- 2c709f81a660: Rename cell/area columns to radius/samples.

Changes

- Maintain *block_first* column.
- Introduce upper bound for Wifi based accuracy numbers.
- Provide better GeoIP accuracy numbers for cities and subdivisions.
- Fix cell queries containing invalid area codes but valid cids.
- #242: Add WiFi stats to region specific stats page.
- Add update_statregion task to maintain region_stat table.
- Update to latest versions of alembic, coverage, datadog, raven and requests.

20151013115000

Migrations

- 33d0f7fb4da0: Add api_type specific logging flags to api keys.
- 460ce3d4fe09: Rename columns to region.
- 339d19da63ee: Add new cell OCID tables.
- All OCID data has to be manually imported again into the new tables.

Changes

- Add new *fallback_allowed* tag to locate metrics.
- Calculate region radii based on precise shapefiles.
- Use subunits dataset to preserve smaller regions.
- Use GENC codes and names in GeoIP results.
- Consider more responses as high accuracy.
- Change internal names to refer to region.
- Change metric tag to region for region codes.
- Temporarily stop using cell/area range in locate logic.
- Discard too large cell networks during import.
- Use mcc in region determination for cells.
- Use new OCID tables in the entire code base.
- Use the intersection of region codes from GENC and our shapefile.
- Avoid base64/json overhead for simple queues containing byte values.
- Maintain a queue TTL value and process remaining data for inactive queues.
- Remove hashkey functionality from cell area models.
- Remove non-sharded update_wifi queue.
- Merge scan_areas/update_area tasks into a single new update_cellarea task.

- Remove backwards compatible tasks and area/mapstat task processing logic.
- Update to latest versions of bower, clean-css and uglify-js.
- Update to latest versions of cryptography, Cython, kombu, numpy, pyasn1, PyMySQL, requests, Shapely, six and WebOb.

20150928100200

Migrations

- 26c4b3a7bc51: Add new datamap table.
- 47ed7a40413b: Add cell area id columns.

Changes

- Improve locate accuracy by taking station circle radius into account.
- Split out OCID cell area updates to their own queue.
- Switch mapstat queue to compact binary queue values.
- Speed up update_area task by only loading required cell columns.
- Validate all incoming reports against the region areas.
- Add a precision reverse geocoder for region lookups.
- Add a finer grained region border file in GeoJSON format.
- Shard update_wifi queue/task by the underlying table shard id.
- Update datatables JS library and fix default column ordering.
- Switch to GENC dataset for region names.
- #372: Add geocoding / search control to map.
- Support the new *considerIp* field in the geolocate API.
- #389: Treat accuracy, altitude and altitudeAccuracy as floats.
- Speed up */stats/regions* by using cell area table.
- Use cell area ids in update_cellarea task queue.
- Enable country level result metrics.
- Removed migrations before version 1.2.
- Update to latest versions of numpy, pytz, raven, rtree and Shapely.

Changelog 1.3

1.3 (2015-09-16)

20150916130500

Migrations

- b24dbb9ccfe: Remove CDMA networks.
- 18d72822fe20: Remove wifi table.

Changes

- Stop importing and exporting CDMA networks.
- #222: Maintain a country/region code estimate for new wifi networks.
- Add new *location_load* script to load cell dumps into a local db.
- Remove obsolete *remove_wifi* task.
- Update to latest versions of certifi, cryptography, coverage and Cython.

20150903095100

Migrations

- Manually run the wifi migration script in *scripts/migrate.py*.

Changes

- Stop using the wifi table.
- Update to latest versions of datadog, greenlet, mako and raven.

20150828125400

Changes

- Fix bug in block_count station update routine.

20150827110300

Migrations

- c1efc747c9: Remove unused api_key email/description columns.
- 4f12bf0c0828: Remove standalone wifi blocklist table.

Changes

- Insert new wifi networks into sharded tables.
- Factor out more of the *aggregate position* logic.
- Optimize gzip compression levels.
- Add new celery queues (*celery_cell*, *celery_ocid*, *celery_wifi*).
- Remove extra internal_dumps call from insert task.
- Add a source tag for successful result metrics.
- Update data tables and stats/regions page.
- Setup Cython support and convert geocalc centroid and distance functions.
- Optimize OCID cell data export and import.
- Return multiple results from MCC based country source.
- Move best country result selection into searcher logic.
- Update to latest versions of alembic, cffi, cryptography, coverage, cython, hiredis, numpy, pip and scipy.

20150813105600

Changes

- Use data_accuracy as the criteria to decide if more locate sources should be consulted.
- Use both old and new wifi tables in locate logic.
- Add a new *__version__* route.
- Cache Wifi-only based fallback position results.
- Don't rate limit cache lookups for the fallback position source.
- Retry outbound connections once to counter expired keep alive connections.

20150806105100

Migrations

- 2127f9dd0ed7: Move wifi blocklist entries into wifi shard tables.
- 4860cb8e54f5: Add new sharded wifi tables.
- The structure of the application ini file changed and the *ichnaea* section was replaced by a number of new more specific sections.

Changes

- Enable SSL verification for outbound network requests.
- Add new metrics for counting unique IPs per API endpoint / API key.
- Enable locate source level metrics.
- #457: Fix cell export to again use UMTS as the radio type string.
- Optimize various tasks by doing batch queries and inserts.
- Avoid using a metric tag called *name* as it conflicts with a default tag.
- Deprecate *insert_measure_** tasks.
- Move new station score bookkeeping into insert_measures task.
- Update to latest version of datadog.

20150730143600

Changes

- Make report and observation drop metrics more consistent.

20150730111000

Migrations

- The statsd configuration moved from the *statsd_host* option in the application ini file into its own section called *statsd*.

Changes

- Move blocklist and station creation logic into update_station tasks.
- Add new *ratelimit_interval* option to *locate:fallback* section.
- Set up a HTTPS connection pool used by the fallback source.
- Disable statsd request metrics for static assets.
- Let all internal data pipeline metrics use tags.

- Let all public API and fallback source metrics use tags.
- Let task, datamaps, monitor and HTTP counter/timer metrics use tags.
- Add support for statsd metric tagging.
- Use colander to map external to internal names in submit schemata.
- Add dependencies pyopenssl, ndg-httpsclient and pyasn1.
- Switch to datadog statsd client library.
- Consider Wifi based query results accurate enough to satisfy queries.
- Stop maintaining separate Python dependency list in setup.py.
- #433: Move GeoIP lookup onto the query object.
- #433: Add new detailed query metrics.
- Use a colander schema to describe the outbound fallback provider query.
- Set up and configure locate searchers and providers once during startup.
- Move all per-query state onto the locate query instance.
- Split customjson into internal and external pretty float version.
- Update to latest versions of alembic, setproctitle, simplejson and SQLAlchemy.

Changelog 1.2

1.2 (2015-07-15)

20150716174000

Changes

- Add a database migration test from a fresh SQL structure dump.

Migrations

- 1a320a751cf: Remove observation tables.

Changes

- #395: Move *incomplete_observation* logic onto colander schema.
- #287: Replace observation models with non-db-model classes.
- #433: Move query data validation into Query class.
- #433: Introduce a new *api.locate.query.Query* class.
- Handle any RedisError, e.g. TimeoutError and not just ConnectionErrors.
- Update to latest raven release and update transport configuration.
- Explicitly limit the cell cache key to its unique id parts.
- Add *fallback* key to all locate responses.

- #451: Properly test and reject empty submit requests.
- #376: Document the added home mcc/mnc fields.
- #419: Update geolocate docs to mention all GLS fields.

20150707130400

Migrations

- 2e0e620ebc92: Remove id column from content models.

Changes

- Add workaround for andymccurdy/redis-py#633.
- Unify v1 and v2 parse error responses to v2 format.
- Batch key queries depending on a per-model batch size.
- #192: Suggest a observation data retention period.
- Optimize mapstat and station data tasks.
- Switch to using bower for CSS/JS dependency management.
- Update to latest versions of all CSS and JS dependencies.
- Update to latest versions of geoip2, SQLAlchemy and unittest2.

20150616104200

Migrations

- 55db289fa497: Add content model composite primary keys.
- 14dbafc4fec2: Remove new_measures indices.
- 19d6d9fbdb6b: Increase stat table value column to biginteger.

Changes

- Fix locate errors for incomplete cell keys.
- Remove backwards compatibility code.

20150610103900

Migrations

- 38fde2949750: Remove measure_block table.

Changes

- #287: Remove table based location_update tasks and old backup code.
- Adjust batch sizes for new update_station tasks.
- Bugzilla 1172833: Use apolitical names on country stats page.
- #443: Reorganize internal module/classes.
- Update to latest version of SQLAlchemy.

20150604164500

Changes

- #446: Filter out incomplete csv cell records.
- #287: Switch location_update tasks to new queue based system.
- #438: Add explicit fallback choices to geolocate API.
- Replace the last daily stats task with a queue based one.
- #440: Allow search/locate queries without a cell id.
- Update to latest versions of nose, simplejson and SQLAlchemy.

20150528085200

Changes

- #394: Replace magic schema values by *None*.
- #423: Add new public *v2/geosubmit* API.
- #242: Pass through submission IP address into the data pipeline.
- #242: Expose geoip database to async tasks.
- Make sure there are no unexpected raven messages left after each test.
- #434: Internal test only changes to test base classes.
- Update to latest versions of gevent and simplejson.

20150522094900

Changes

- #421: Pass through additional lookup data into the fallback query.
- #421: Cache cell-only lookups for fallback provider queries.
- #421: Add rate limiting to fallback provider.
- #421: Reordered data sources to prefer fallback over geoip responses.
- Fix api-key specific report upload counter.
- Add workaround for raven issue #608.
- Enable new stat counter tasks.
- #433: Remove the wifi specific query stats.
- Updated to latest version of alembic, celery, greenlet, kombu and pytz.

20150507103300

Changes

- Correct handling for requests without API keys.
- #421: Fix encoding of radioType in fallback queries.

20150505113200

Migrations

- e9c1224f6bb: Add allow_fallback column to api_key table.

Changes

- #287: Move mapstat and score processing to separate queues/tasks.
- #287: Keep track of uploaded data via Redis stat counters.
- #287: Add new backup to S3 export target.
- #421: Add fallback geolocation provider.
- Deal with nan/inf floating point numbers in data submissions.
- Fixed upload issues for cell entries without any radio field.
- Updated to latest versions of certifi, greenlet, pyramid, raven and requests.

20150423105800

Changes

- Allow anonymous data submission via the geosubmit API.
- #425: Refactor internal API key logic.
- Updated to latest raven version, requires a Sentry 7 server.
- Updated to latests versions of billiard, pyramid and WebOb.

20150416111700

Migrations

- The command line invocation for the services changed, please refer to the deploy docs for the new syntax.

Changes

- #423: Add a first version of an export job.
- Expose all config file settings to the runtime services.
- Move runtime related code into async/webapp sub-packages.
- #385: Configure Python's logging module.
- #423: Add a new queue system using the new geosubmit v2 internal format.
- Updated to latest versions of boto and maxminddb.

20150409120500

Changes

- Make radio an internally required field.
- Don't validate radio fields in request side schema.
- #418: Remove country API shortcut implementation.
- Removed BBB code for old tasks and pre-hashkey queued values.
- Updated to latest versions of alabaster, boto, factory_boy and pytz.

20150320100800

Changes

- Remove the circus docs and example ini file.
- Remove the vaurien/integration tests.
- #416: Accept radioType inside the cellTowers mapping in geolocate queries.
- Updated to latest version of Sphinx and its new dependencies.
- Updated to latest versions of pyramid, requests, SQLAlchemy and statsd.

20150309175500

- Fix unittest2 version pin.

20150305122500

Migrations

- 1d549c1d6cfe: Drop total_measures index on station tables.
- 230bbf3fe044: Add index on mapstat.time column.
- 6527bee5ac1: Remove auto-inc id columns from cell related tables.
- 3b8d52a9eac4: Change score, stat and measure_block enum columns to tinyint.

Changes

- Replace heka-py-raven with a direct raven client.
- #319: Remove the per station ingress filtering.
- Allow partial cell ids in geolocate/geosubmit APIs.
- Removed the mixed locate/submit mode from the geosubmit API.
- #402: Avoid multiple validation of common report data fields.
- Add a new CellCountryProvider to allow country searches based on cell data.
- #406: Allow access to the country API via empty GET requests.
- Massive internal code refactoring and cleanup.
- Updated to latest versions of iso3166, pyramid and requests.

20150211113000

Changes

- Reestablish database connections on connection failures.

20150209110000

Changes

- Backup/delete all observation data except for the current day.
- Updated to latest versions of boto, Chameleon, gunicorn, jaraco.util, Mako, psutil, Pygments, pyzmq and WebTest.

20150203093000

Changes

- Specify statsd prefix in application code instead of heka config.
- Fix geoip country lookup for entries without countries.
- #274: Extend monitor view to include geoip db status.

20150127130000

Migrations

- 10542c592089: Remove invalid lac values.
- fe2cfea89f5: Change cell/_blacklist tables primary keys.

Changes

- #367: Tighten lac filtering to exclude 65534 (gsm) and 65535 (all).
- Remove alembic migrations before the 1.0 PyPi release.
- #353: Remove auto-inc id column from cell/_blacklist tables.
- Add additional stats to judge quality of WiFi based queries.
- #390: Remove command line importer script.

20150122140000

Migrations

- 188e749e51ec: Change lac/cid columns to signed integers.

Changes

- #352: Switch to new maxmind v2 database format and libraries.
- #274: Add a new *__monitor__* endpoint.
- #291: Allow 32bit UMTS cell ids, tighten checks for CDMA and LTE.
- #311: On station creation optionally use previous blacklist time.
- #378: Use colander for internal data validation.
- Remove explicit queue declaration from celery base task.
- Updated to latest versions of alembic, boto, Chameleon, jaraco.util, mobile-codes, psutil, requests-mock, WSGIProxy2 and zope.deprecation.

20150105140000

Migrations

- 48ab8d41fb83: Move cell areas into separate table.

Changes

- Prevent non-countries from being returned by the country API.
- #368: Add per API key metrics for uploaded batches, reports and observations.
- Clarify metric names related to batches/reports/observations, add new *items.uploaded.batch_size* pseudo-timer and *items.uploaded.reports* counter.
- Introduce new internal *GeoIPWrapper.country_lookup* API.
- #343: Fall back to GeoIP for incomplete search requests.
- #349/#350: Move cell areas into new table.
- Give all celery queues a prefix to better distinguish them in Redis.
- #354: Remove scan_lacs fallback code looking at new_measures.
- Updated to latest versions of alembic, argparse, billiard, celery, colander, filechunkio, iso8601, kombu, PyMySQL, pytz, requests, six, WebTest and zope.interface.

20141218093500

- #371: Add new country API.

20141120130000

- Add api key specific stats to count best data lookup hits/misses.
- Validate WiFi data in location lookups earlier in the process.
- #312: Add email field to User model.
- #287: Move lac update scheduling to Redis based queue.
- #304: Auto-correct radio field of GSM cells with large cid values.
- Move responsibility for lac entry deletion into update_lac task.
- Accept more ASU values but tighten signal strength validation.
- #305: Stricter range check for mnc values for non-CDMA networks.
- Add a convenience *session.on_post_commit* helper method.
- #17: Remove the unused code for cell backfill.
- #41: Explicitly allow anonymous data submissions.
- #335: Omit incomplete cell records from exports.
- Delete measures in batches of 10k rows in backup tasks.
- Re-arrange backup tasks to avoid holding db session open for too long.
- Report errors for malformed data in submit call to sentry.
- Report errors during backup job to sentry.
- #332: Fix session handling in map tiles generation.
- Updated to latest versions of argparse, Chameleon, irc, Pygments, pyramid, translationstring and unittest2.

20141103125500

- #330: Expand api keys and download sections.
- Close database session early in map tiles generation.
- Close database session early in export task to avoid timeout errors while uploading data to S3.
- Optimize cell export task and avoid datetime/unixtime conversions.
- Add an index on cell.modified to speed up cell export task.
- Updated to latest versions of boto, irc, pygeoip, pytz, pyzmq, simplejson and unittest2.

20141030113700

- Add play store link for Mozilla Stumbler to apps page.
- Updated privacy notice style to match general Mozilla style.
- Switch gunicorn to use a gevent-based worker.
- Clean last database result from connections on pool checkin.
- Close the database connections even if exceptions occurred.

Changelog 1.1

1.1 (2014-10-27)

20141027122000

- Lower DB pool and overflow sizes.
- Update Mozilla Stumbler screenshot.
- Update to new privacy policy covering both Fennec and Mozilla Stumbler.

20141023094000

- Updated Fennec link to point to Aurora channel.
- Renamed MozStumbler to Mozilla Stumbler, added new screenshot.
- Increase batch size for *insert_measures_wifi* task.
- Extend queue maximum lifetime for incoming reports to six hours.
- Extend observation task batching logic to apply to cell observations.
- #328: Let gunicorn start without a valid geoip database file.
- Extend the *make release* step to deal with Python files with incompatible syntax.
- Update to latest versions of configparser, greenlet, irc and pyzmq.

20141016123300

- Log gunicorn errors to stderr.
- #327: Add an anchor to the leaderboard table.
- Move the measure tables gauges to an hourly task.
- Fix initdb script to explicitly import all models.

20141014161400

- #311: Filter out location areas from unique cell statistics.
- Introduce a 10 point minimum threshold to the leaderboard.
- Change download page to list files with kilobytes (kB) sizes.
- #326: Quantize data maps image tiles via pngquant.
- Optimize file size of static image assets.
- Remove task modules retained for backwards compatibility.
- Update to latest version of SQLAlchemy.

20141009121300

- Add a task to monitor the last import time of OCID cells.
- Change api_key rate limitation monitoring task to use shortnames.
- Small improvements to the manual importer script.
- #276: Fix bug in batch processing, when receiving more than 100 observations in one submission.
- Refactor some internals and move code around.
- Create a new *lbcheck* MySQL user in the *location_initdb* command.
- Fix *monitor_api_key_limits* task to work without api limit entries.
- #301: Schedule hourly differential imports of OCID cell data.
- Update to latest versions of boto, celery, iso3166, jaraco.util, requests and simplejson.

20141002103900

- #301: Add OCID cell data to statistics page.
- Allow a radio type of *lte* for the geolocate API. Relates to https://bugzilla.mozilla.org/show_bug.cgi?id=1010284.
- #315: Add a *show my location* control to the map.
- Reverse ordering of download files to display latest files first.
- Extend db ping to retry connections for *2003 connection refused* errors.
- Ignore more exception types in API key check, to continue degraded service in case of database downtimes.
- Switch from d3.js/rickshaw to flot.js and prepare graphs to plot multiple lines in one graph.
- Make country statistics table sortable.
- Remove auto-increment column from ocid_cell table and make the radio, mcc, mnc, lac, cid combination the primary key. Also optimize the column types of the lac and cid fields.
- Update to latest versions of alembic, amqp, celery, configparser, cornice, greenlet, jaraco.util, kombu, protobuf, psutil, pytz, requests, six, Sphinx and WebTest.
- #301: Add code to do continuous updates of the OpenCellID data and add license note for OCID data.

20140904094000

- #308: Fixed header row in cell export files.

20140901114000

- #283: Add manual logic to trigger OpenCellID imports.
- Add Redis based caching for SQL queries used in the website.
- #295: Add a downloads section to the website and enable cell export tasks.
- Clarify api usage policy.
- Monitor api key rate limits and graph them in graphite.
- Update to latest versions of nose and simplejson.
- #282: Add a header row to the exported CSV files.

20140821114200

- #296: Trust WiFi positions over GeoIP results.
- Optimized SQL types of mnc, psc, radio and ta columns in cell tables.
- Update to latest versions of country-bounding-boxes, gunicorn and redis.
- #282: Added code to do exports of cell data, both daily snapshots as well as hourly diffs. Currently the automatic schedule is still disabled. This also adds a new modified column to the cell and wifi tables.

20140812120000

- Include links to blog and new @MozGeo twitter account.
- Update to latest version of alembic, boto, redis, simplejson and statsd.
- Add a monitoring task to record Redis queue length.
- Make a Redis client available in Celery tasks.
- #285: Update favicon, add touch icon and tile image.
- Only retain two days of observation data inside the DB.
- Fixed image tiles generation to generate up to zoom level 13 again.
- #279: Offer degraded service if Redis is unavailable.
- #72: Always log sentry messages for exceptions inside tasks.
- #53: Document testing approaches.
- #130: Add a test for syntactic correctness of the beat schedule.
- #27: Require sufficiently different BSSIDs in WiFi lookups. This reduces the chance of being able to look up a single device with multiple logical networks.

20140730133000

- Avoid using *on_duplicate* for common update tasks of tables.
- Remove GeoIP country submission filter, as GeoIP has shown to be too inaccurate.
- #280: Relax the GeoIP country restriction and also trust the mcc derived country codes.
- #269: Improve search logic when dealing with multiple location areas.
- Correctly deal with multiple country codes per mcc value and don't restrict lookups to one arbitrary of those countries.
- Fix requirement in WiFi lookups to really only require two networks.
- Added basic setup for documenting internal code API's and use the geocalc and service.locate modules as first examples.
- Initialize the application and outbound connections as part of the gunicorn worker startup process, instead of waiting for the first request and slowing it down.
- Switch pygeoip module to use memory caching, to prevent errors from changing the datafile from underneath the running process.
- Introduce 10% jitter into gunicorn's max_requests setting, to prevent all worker processes from being recycled at once.
- Update gunicorn to 19.1.0 and use the new support for config settings based on a Python module. The gunicorn invocation needs to include *-c ichnaea.gunicorn_config* now and can drop various of the other arguments.
- Updated production Python dependencies to latest versions.
- Updated supporting Python libraries to latest versions.
- Update clean-css to 2.2.9 and uglify-js to 2.4.15.
- Update d3.js to 3.4.11 and jQuery 1.11.1.
- Changed graphs on the stats page to show a monthly count for the past year, closes https://bugzilla.mozilla.org/show_bug.cgi?id=1043386.
- Update rickshaw.js to 1.5.0 and tweak stats page layout.
- Add MLS logo and use higher resolution images where available.
- Always load cdn.mozilla.net resources over https.
- Updated deployment docs to more clearly mention the Redis dependency and clean up Heka / logging related docs.
- Split out circus and its dependencies into a separate requirements file.
- Remove non-local debug logging from map tiles generation script.
- Test all additional fields in geosubmit API and correctly retain new *signalToNoiseRatio* field for WiFi observations.
- Improve geosubmit API docs and make them independent of the submit docs.
- Update and tweak metrics docs.
- Adjust Fennec link to point to Fennec Nightly install instructions. https://bugzilla.mozilla.org/show_bug.cgi?id=1039787

20140715114000

- Adjust beat schedule to update more rows during each location update task.
- Let the backup tasks retain three full days of measures in the DB.
- Remove the database connectivity test from the heartbeat view.

Changelog 1.0

1.0 (2014-07-14)

- Initial production release.

0.1 (2013-11-22)

- Initial prototype.

1.5 Glossary

Cell-ID

Cell ID

Cell IDs

CGI Cell ID describes both a globally unique identifier for any logical cell network, as well as an approach to locate devices based on prior knowledge of the positions of these cell networks. Sometimes the term cell global identity (CGI) can also be found. See also *WPS*.

CID A term describing the GSM or WCDMA cell id, LTE cell identity, or CDMA base station id. All these form the last part of the globally unique *Cell ID*.

decimal degree

decimal degrees Decimal degrees express latitude and longitude coordinates as decimal fractions. For example, `10.1234567`.

MLS The Mozilla Location Service is an instance of the Ichnaea software hosted by Mozilla.

observation

observations An observation describes the data collected about a single *station* identified by its unique global identifier, additional data about the *station* like signal strength readings, and *report* data about the position of the *station*.

OpenCellID

OCID OpenCellID is a collaborative project to create a free worldwide database of *Cell IDs*.

report

reports A report describes the data collected in a single reading consisting of data about the position and movement at the time of taking the reading and data about multiple *stations* observable at the time. For example, one report could contain information about one cell network and 10 WiFi networks.

station

stations A term referring to any radio signal emitting stationary device or the radio network it emits. Examples of what we call stations are WiFi access points / WiFi network, cell towers / cell networks, and Bluetooth Beacons / Bluetooth LE networks.

Web Mercator

WSG84 WSG 84 Web Mercator refers to the geographic map projection used throughout this project. The latitude and longitude values use WSG 84 as the coordinate reference system.

WiPS

WPS Wi-Fi based positioning system describes a system of using prior knowledge about the location of WiFi networks, identified by their globally unique BSSID/MAC address, to position devices. See also *Cell ID*.

CHAPTER

TWO

INDICES

- genindex
- *Glossary*

CHAPTER

THREE

SOURCE CODE AND LICENSE

All source code is available on GitHub under ichnaea.

The `Ichnaea` source code is offered under the Apache License 2.0.

CHAPTER

FOUR

ABOUT THE NAME

In Greek mythology, Ichnaea () means “the tracker”.

INDEX

C

Cell ID, **111**
Cell IDs, **111**
Cell-ID, **111**
CGI, **111**
CID, **111**

D

decimal degree, **111**
decimal degrees, **111**

I

ichnaea.conf.AppComponent (*component*), 33
ichnaea.conf.AppComponent.ASSET_BUCKET (*component*), 33
ichnaea.conf.AppComponent.ASSET_URL (*component*), 34
ichnaea.conf.AppComponent.CELERY_WORKER_CONCURRENCY (*component*), 35
ichnaea.conf.AppComponent.DB_READONLY_URI (*component*), 34
ichnaea.conf.AppComponent.DB_READWRITE_URI (*component*), 34
ichnaea.conf.AppComponent.GEOIP_PATH (*component*), 35
ichnaea.conf.AppComponent.LOCAL_DEV_ENV (*component*), 33
ichnaea.conf.AppComponent.LOGGING_LEVEL (*component*), 33
ichnaea.conf.AppComponent.MAPBOX_TOKEN (*component*), 35
ichnaea.conf.AppComponent.REDIS_URI (*component*), 35
ichnaea.conf.AppComponent.SECRET_KEY (*component*), 35
ichnaea.conf.AppComponent.SENTRY_DSN (*component*), 34
ichnaea.conf.AppComponent.SENTRY_ENVIRONMENT (*component*), 34
ichnaea.conf.AppComponent.STATSD_HOST (*component*), 34
ichnaea.conf.AppComponent.STATSD_PORT (*component*), 34
ichnaea.conf.AppComponent.TESTING (*component*), 33

M

MLS, **111**

O

observation, **111**
observations, **111**
OCID, **111**
OpenCellID, **111**

R

report, **111**
reports, **111**

S

station, **111**
stations, **112**

W

Web Mercator, **112**
WiPS, **112**
WPS, **112**
WSG84, **112**