Company Confidential

Feature Specification and Design Document

Usage Data Collection for SoundTouch Devices.

Document No.: none
Document Version: 0.3
Date: April, 15th, 2015

Abstract

This document describes requirements and design Usage Data Capture for SoundTouch Devices. This document describes the device side or client side design of the feature. The server side design is covered in another document.

Issued by:
Home Entertainment Division
Bose Corporation
100 Mountain Road
Framingham, MA,
USA

Document History

Version	Date	Author	Description
0.1	December, 5th,2014	Keith Martin, Amit Kuthiala, Raj Garikapati, David Datta, Young Gao, Trevor Lai, Yishai Sered	First Draft. Requirements. High Level Design.
0.2	March, 15th, 2015	Boris Tkachenko, Trevor Lai, Yishai Sered	Additions and Clarification Requirements and High Level Design. Pre Implementation Research, Design Elaboration, Implementation Details.
0.3	April, 14th, 2015	Trevor Lai, Sharad Mathur, John Cooper, Mike Cook, Boris Tkachenko	Additions Related to Network Information Capture Architecture and Design.

Table of Contents

1.

1.

1. Introduction

1.1 Purpose

Statistics collection is an important function of many consumer market applications. It is very important for SoundTouch line of devices.

Authoritative Opinion of Bose Marketing:
Phase 1 of the Usage Data Collection supports exploration of new SoundTouch features as outlined in the Bose Connected Music strategy. For more information contact Keith Martin or Eric Scheirer."

Somewhat more specifically, this data collection enables us to understand customer music preferences, which we intend to use to make contextual and preference-based recommendations and other novel features.

UDC is an introduced by one of the authors acronym for Usage Data Collection for SoundTouch devices. This document describes the requirements and design of UDC.

Section 1, Introduction, highlights the audience for which this document is intended, contains an overview of this new feature, lists the references used in creating the document, and includes a table of acronyms used in this document.

Section 2, Assumptions, describes all assumptions made in writing this document. Although this section is not decomposed by subject, the assumptions may address all or some of the following topics: hardware, operating system software, installation tools, configuration, tools, database, documentation, training, performance and capacity, and availability and reliability.

Section 3, Feature Requirements Specification and Architecture Description, This chapter contains feature requirements specification that was originally defined by project founders and stakeholders in the process of inter departmental collaboration.

Section 4, High Level Design, describes design approach, static and dynamic aspects of the design. This chapter was further formalized, clarified and finalized in the process of actual implementation of the feature. Is a product of design decisions and the design and code re view.

Section 5, Design Elaboration, describes practical aspects of the design, design choices and the trade-offs.

Section 6, Implementation Details, describes and documents exact implementation details of generic and procedural nature that are not obvious from code review alone.

Section 7, Design Validation and Testing, describes and documents exact implementation details of generic and procedural nature that are not obvious from code review alone.

1.2 Audience

The intended audience is Project Management, System Engineering, Software Development, Software Testing. Familiarity with Shelby ecosystem is assumed. Familiarity with Embedded Linux Operating System, Shell Scripting, Object Oriented Analysis and Development, Software Development Lifecycle and C/C++ programming language may be helpful for understanding of some chapters.

1.3 Feature Overview

1.4 References

[1] Proposed Data Capture Architecture, 12/05/2014, Author Keith Martin with input from Amit Kuthiala, Raj Garikapati, Young Gao, Trevor Lai, Yishai Sered.

[2] Email communication between Trevor Lai, Yishai Sered, Boris Tkachenko, Keith Martin, Rick Schnur, Kuthiala Amit regarding UDC project. December 2014 and January 2015.

[3] Usage Data Collection for SoundTouch Devices, December 2014, by Yishai Sered.

[4] SCM source code tree. https://svn.bose.com/hepd/Shelby

[5] Bose Engineering Wiki.
http://hepdwiki.bose.com/bin/view/Main/WebHome

[6] Bose Confluence. Capture Playback Event page, by Kuthiala, Amit,
http://ocgwiki.bose.com/display/OCGCW/Capture+Playback+Event

[7] Bose Confluence. Get Blacklist Filter for Data Collection page, by Kuthiala, Amit,http://ocgwiki.bose.com/display/OCGCW/Get+Blacklist+filter+for+data+collection

[8] Mike Lorince Data Collection Alignment weekly teleconferencing calls.

[9] Bose Shelby List of Acronyms and Abbreviations,
http://ocgwiki.bose.com/display/OCGCW/Shelby+Glossary

[10] http://ocgwiki.bose.com/display/OCGCW/Deep+ETag+Implementation

1.5 Acronyms and Abbreviations

EFE	Engineering Field Evaluation.
DCS	Data Capture Service.
DNSF	Do Not Send Filter.
PC	Personal Computer.
SCM	Shelby Common Module, that refers to both SM1 and SM2 throughout this document.
SSID	Service Set Identifier.
NRF	Negative Reply Failure.
CTF	Connection Timeout Failure.
RTF	Reply Timeout Failure.
PPTCI	Pre Provisioned Test Case Initialization.
NCADI	Network Card Address Device Identifier. This is a hexadecimal string of 12 characters long.
UDC	Usage Data Collection. Usage Data Collection Service may be used as a synonym for DCS. Statistics Data Capture may be used to refer to an end to end process of collecting statistics and sending them to DCS.

1.6 Definitions

UDC Subsystem	A collection of changes additions and enhancements that work together to implement a Usage Data Collection requirements.
MACAF	Map of All Collectable Able Fields.

CMCF	Current Map of Collected Fields.
CI	Context Information. Consists of Information that is to be posted with every UDC POST that includes Device Info and Track Info.
CE	Capture Element is a new code that is added to the SCM in a course of this project to capture one or many collectable events at the location in SCM source code where such capture is optimal.

2. Assumptions

- UDC is assumed to consist of set of enhancements to BoseApp.
- Hardware assumed to be TI ARM.
- OS Assumed ARM Linux 32 bit.
- GCC compiler version assumed is 4.7.2 or later version that is backward compatible.

3. Feature Requirements Specification and Architecture Description.

Server Side General Overview.

The server side technology is not maintained by SCM and provided here only for general background for the reader to have some idea of what is going on server side.
The Data Capture Service will be independent of Marge.
The Data Capture Service will run on a Tomcat Web Container, connected via AWS SQS (a managed Queue Service) to an Amazon S3 Data Repository. Data will be time-windowed (i.e., events older than, e.g., 10 days, will be archived to slower, cheaper storage, while active data will remain in the faster S3 repository). The database layer itself has not been determined yet, but it will explicitly support a Spark layer (and any layers that can be built on Spark, e.g., Shark for SQL queries).

SCM Client Side Requirements that are within the Initial Scope of the Project.

It is normal that more features may emerge in the minds of project initiators and customers while project is in a stage of implementation. That is a product of increased awareness of things that are now possible and new features that become within reach. Such new features and expanded requirements are not within the Initial Scope of the Project. Such new requirements and expanding scope may and need to be documented. They will be documented in separate chapter.

3.1 Data transport
The data transport between SCM and the Data Capture Service will be a POST URL sent over HTTPS, with the payload data encoded in JSON format (data fields specified below).

3.2 Only SCM is required to send data to the Data Capture Service for this initial set of requirements.

3.3 SCM will not send data unless the EULA has been accepted and network (that includes Wi Fi or Ethernet) is connected (i.e., we will not collect AirPlay/Bluetooth data unless the user has accepted our license terms, and we will not cache data while disconnected from the network). If the user has created a marge account that means that he accepted the account license agreement that includes the permission to Bose to collect Usage Data. We therefore assume that if user has marge account and is logged in on that device that means the device allowed to collect Usage Data.

3.4
The Data Capture Service will maintain a “blacklist” that allows Bose to specify, for particular regions and particular accounts, which data will not be transmitted from SCM to the Data Capture Service. Each time the SCM comes out of network standby, it requests the appropriate blacklist from the server. The blacklist specifies the data fields that the SCM should not transmit.

These data fields are hierarchical, so for example, if the blacklist specifies “track”, the SCM will not transmit any of the subfields for the track data structure (i.e., artist, album, trackTitle, etc.). The filter is a synonym for “blacklist”. The blacklist data is persisted in SCM. On power up, the old blacklist is used until a new blacklist is fetched.

3.5 Each event (button press, transport event), the SCM will immediately queue an asynchronous POST task.

3.6 In the event of a failure, the SCM will retry exactly once. In the event of failed retry, the SCM will throw away the data, but keep a flag that some amount of data has been lost. The data will be discarded after 2 failures, only the flag that “some data was lost” would be preserved. This flag would be attached to the subsequent transmission (triggered by the next event) and will be cleared once transmission has succeeded. If easy, we can persist the flag. If not easy, we can probably get permission to lose the flag when the system is rebooted.

3.7 Preset is pressed but is empty: Still send that event

3.8 Volume button pressed but nothing is playing: Still send that event

3.9 What do we send for Bluetooth? Make a separate source provider for Bluetooth. Track changes on Bluetooth are probably just "nowPlaying" or "started" events: anything is fine but started is preferred.

3.10 The Now Playing Track Information will be cached and immediately available for other events. Whenever NPTI changes from any reason the Start Playing event must be sent to UDC Server.

3.11 We also know that by virtue of its design and implementation the SCM produces repeated EVT_NOW_PLAYING_UPDATED when only the timestamp changes. The SCM UDC client software must compare the NPTI with the currently stored NPTI and whenever NPTI does not change no UDC post will be sent to marge. Repeated events that differ only by timestamp will not be posted over UDC.

3.12 The Now Playing Track Information (NPTI) that will be posted to Data Server will not include all information that has arrived in original protocol buffer event. Posts to data server will include fields required and permitted to collect for Usage Data Collection Purposes only.

4. Pre Implementation Research Results

The goal of pre implementation research is to figure out the whole system behavior that affect an integration of a subsystem under design and development that is described here.

4.1 Why there is a three Now Playing events arrive in a sequence when song changes.

It was discovered that when the song or radio station that is currently played is changing the behavior is as follows: three consecutive EVT_NOW_PLAYING_UPDATED messages are produced.

The event information comparator was written to verify the reason why this is occurring.

The following was discovered.

The First EVT_NOW_PLAYING_UPDATED message that arrives it does not have a track info, the track info is empty. The play state is BUFFERING.

The Second EVT_NOW_PLAYING_UPDATED message that arrives contains full track info describing the new song selected by user it notably the Art Image status is changing from INVALID to IMAGE_PRESENT.

The Third EVT_NOW_PLAYING_UPDATED contains all information that is the same as in the Second except the play state is changing from BUFFERING to PLAY.

4.2 How to formally through the code verify that account is logged in to marge.

A device is logged in to Marge if it has a Marge Bearer Token cached locally.

4.3 Where and how can I get Marge bearer token.

Bearer token obtained from Marge at the time device is paired with Marge account. The User interface to enter login information is not on the SCM device it is on the client whether it is SoundTouch App Homer or Gabbo or another. Therefore presence of not empty bearer token is a reliable indication that user has logged in to Marge. It is also an indication that user has allowed us to collect Usage Data at the time of Marge account creation.

Marge bearer token can be accessed using the following method:

Persistence::SystemConfigurationStore::GetAccountMargeCredentials

4.4 Produce Example of Usage of Asynchronous HTTP Post.

Http::StartAsyncSimpleURLFetchRequest function will be used to do an asynchronous post of collected data to Data Server. This is from **Core/Infrastructure/Http**

4.5 Produce Example of Read Write Persistence to NVRAM.

NVRAM persistence for UDC Blacklist Filter is implemented just like the rest of persistence using write and read to /mnt/nv file system in an XML format. UDC persistent data will be kept in **/mnt/nv/BoseApp-Persistence/1/blacklist.xml**

4.6 Event Protocol Compression Requirements for Key Press Events.

4.6.1 It is TBD whether we would like to collect all generated events for key press events. For example for one of the simplest key presses “like”, “dislike” two events are generated key press and key release. Do we really need to collect both events or just key release will be enough?

4.6.2 The TBD 4.6.1 was resolved to decision to collect only Key Release, discard the Key Press.

4.7 Collect Data Server API Specification from the Server Team.

Capture Playback Event

URI

```
/data/device/{deviceid}/event/{eventName}
```

Request Header

```
Content-Type: application/json
Accept: application/json
Authorization: Bearer 666331e56ecb09c349efc7c8623502b
```

Request Body

```
{
"event":{
    "transportEvent":"started",
        "buttonPress": "1",
        "eventOrigin": "device"
},
"track":{
        "artist": "Artist Name",
        "album": "Album title",
        "trackTitle": "Track title",
        "sourceProvider":"2",
        "sourceName": "name",
        "location":"location",
        "trackType": "song"
},
"device":{
       "deviceID": "1A345fg",
        "boseID": "12345",
        "playEverywhere":"master",
        "volume": "30"
}
}
```

Response Header

```
200 OK
```

POST https request will be used for Capture Playback Event API. The understanding was reached to rename Capture Playback Event to just Capture Event because it is used for every event posted.

Get Blacklist filter for data collection

Overview

This request will return the list of attributes/nodes that should not be sent back to the server when a diagnostic push is done. The device asks for it's blacklist filter and the server returns a filter back depending on the rules that have been setup. This could be account level or based on the region where the account belongs to.

The server will also maintain a Deep ETag implementation for this as this could be a heavy call as well.

URI

```
/streaming/device/{deviceId}/data/blacklist
```

Request Header

```
Accept: application/json
Authorization: Bearer 666331e56ecb09c349efc7c8623502b
If-None-Match: "06ad14fd9b1852db2295621516b15cf85"
```

Response Header

```
200 OK

ETag: "0428a8553238b3bb4dbb6a2a0f2e17689"
```

Response

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<blacklist>
  <item>track.song</item>
  <item>event</item>
</blacklist>
```

To indicate that nothing should be sent the server would response with the xml below.

All data blacklisted

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<blacklist>
  <item>*</item>
</blacklist>
```

GET https request will be used for Get Blacklist Filter API.

Other Events, Key Press Events and Transport Events Included.

The format of posted content of the remainder of events that will be posted to Data Server by SCM is very similar to content posted for Capture Playback Event. The JSON fields that are specified using Red Bold font below will vary by event.


```
{
  "device" : {
    "boseID" : "24387",
    "deviceID" : "D05FB8A9ACC1",
    "deviceType" : "SoundTouch 20",
    "serialNumber" : "Y43320035047200480000010"
  },
  "event" : {
    "buttonPress" : "none",
    "eventOrigin" : "device",
    "transportEvent" : "started"
  },
  "track" : {
    "album" : "",
    "artist" : "",
    "sourceName" : "BBC World Service English News Online",
    "sourceProvider" : 2,
    "trackTitle" : "",
    "trackType" : "song"
  }
}
```


The above format is applicable to collection of User Activity and Audio Transport events. Other types of information that handled by UDC can use formats that significantly differ from the format specified above.

The examples of actual JSON posted for “started” event end for one of the key press events follows. These JSON were produced during actual developer testing.

Song Started Event URL and JSON Example.

```
https://streamingefeint.bose.com/data/device/D05FB8A9ACC1/event/started
{
   "device" : {
      "boseID" : "24387",
      "deviceID" : "D05FB8A9ACC1",
      "deviceType" : "SoundTouch 20",
      "serialNumber" : "d05fb8a9acc1"
   },
   "event" : {
      "buttonPress" : "none",
      "eventOrigin" : "device",
      "transportEvent" : "started"
   },
   "track" : {
      "album" : "",
      "artist" : "",
      "sourceName" : "DiscoParty - Disco Polo",
      "sourceProvider" : "2",
      "trackTitle" : "",
      "trackType" : "song"
   }
}
```

Key Press Event URL and JSON Example.

```
https://streamingefeint.bose.com/data/device/D05FB8A9ACC1/event/volume-down-pressed
{
   "device" : {
      "boseID" : "24387",
      "deviceID" : "D05FB8A9ACC1",
      "deviceType" : "SoundTouch 20",
      "serialNumber" : "d05fb8a9acc1"
   },
   "event" : {
      "buttonPress" : "11",
      "eventOrigin" : "ir-remote",
      "transportEvent" : "none"
   },
   "track" : {
      "album" : "",
      "artist" : "",
      "sourceName" : "DiscoParty - Disco Polo",
      "sourceProvider" : "2",
      "trackTitle" : "",
      "trackType" : "song"
   }
}
```

The “buttonPress” JSON field contain a key code. The key codes are defined in Shelby Keys protocol buffer.

Bose Key Codes from Shelby Keys protocol buffer.

```
enum KEY_VALUE {
  KEY_VAL_PLAY = 0,
  KEY_VAL_PAUSE = 1,
  KEY_VAL_STOP = 2,
  KEY_VAL_PREV_TRACK = 3,
  KEY_VAL_NEXT_TRACK = 4,
  KEY_VAL_THUMBS_UP = 5,
  KEY_VAL_THUMBS_DOWN = 6,
  KEY_VAL_BOOKMARK = 7,
  KEY_VAL_POWER = 8,
  KEY_VAL_MUTE = 9,
  KEY_VAL_VOLUME_UP = 10,
  KEY_VAL_VOLUME_DOWN = 11,
  KEY_VAL_PRESET_1 = 12,
  KEY_VAL_PRESET_2 = 13,
  KEY_VAL_PRESET_3 = 14,
  KEY_VAL_PRESET_4 = 15,
  KEY_VAL_PRESET_5 = 16,
  KEY_VAL_PRESET_6 = 17,
  KEY_VAL_AUX_INPUT = 18,
  KEY_VAL_SHUFFLE_OFF = 19,
  KEY_VAL_SHUFFLE_ON = 20,
  KEY_VAL_REPEAT_OFF = 21,
  KEY_VAL_REPEAT_ONE = 22,
  KEY_VAL_REPEAT_ALL = 23,
  KEY_VAL_PLAY_PAUSE = 24,
  KEY_VAL_ADD_FAVORITE = 25,
  KEY_VAL_REMOVE_FAVORITE = 26,
  INVALID_KEY = 27
};
```

5. High Level Design.

Data will be sent to a cloud server (URL in Configuration-devicex-dist.xml) using HTTPS and in JSON format. Data will be sent for each supported user interaction immediately following the interaction. If the send transaction fails for some reason the interaction will be dropped. No queueing, no acknowledgement beyond the HTTPS mechanism. Data will be sent only after the user has consented to the EULA. The device will fetch (again using HTTPS and JSON) an interaction filter from the server when it completes OOB, and every time it is turned on from Network Standby or from Low Power Standby. The interaction filter will tell the device which interactions are to be sent. The JSON format for the filter will be modeled after the JSON format of the interaction instance and include true/false for each item. The JSON format will be hierarchical enough to support the expected filtering profile: top level false will stop collection from the device altogether. Other elements of the hierarchy will include user data, each content service provider, other TBD.

The data will be collected within BoseApp (no need for a separate process). The collector will be a Thread Agnostic Task Object or TO. TO will subscribe to events and interrogate PDOs to access data. The system already uses ProtoFromJson in production code. ProtoToJson was written as proof of concept and is known not to be production ready. ProtoToJson is based on Json-cpp library that is one layer down the software stack. The ProtoToJson generic tool will not be used because none of the current protocol buffers defined exactly maps on the data collection records that we plan to send and because some of the data that will be included will come from outside of any defined protocol buffer.

5.1 User Activity Capture.

Total four transactions are required.

1) Fetch Filter
2) Post Button Press
3) Post Transport Event
4) Post Now Playing Updated

Fetch Filter is GET over HTTPS and other three are POSTs over HTTPS

Track information:

"Now playing" data changes are received via `EVT_NOW_PLAYING_UPDATED`, can be handled by

```
bool ProcessNowPlayingUpdate( const CEvtSourcePb<WebInterface::nowPlaying>& msg )
{
    WebInterface::nowPlaying pb = msg.GetPb();
```

Most of the required fields under "track" can be fetched from `pb` using the following interfaces:

artist: `pb.artist()`
album: `pb.album()`

trackTitle: `pb.track()`
sourceProvider: `pb.source()`
sourceName: `pb.stationname()`
location: from PDO `CurrentContentItem` then `ci.location()`
trackType: `pb.isadvertisement()`

Device information:

deviceID: `SystemConfigurationStore::Get().GetDeviceId()`
boseID: `Persistence::SystemConfigurationStore::Get().GetAccountUUID()`
serialNumber: `SoftwareVersionRegistry::Get().GetSCMInfo().GetSerialNumber()`
deviceType: `SystemConfigurationStore::Get().GetPrettyModelDisplayName()`
playEverywhere: capture data from `CEvtZoneState` delivered by `EVT_SYSTEM_ZONE_MODE_STATE`
playEverywhereMaster: capture data from `CEvtZoneState` delivered by `EVT_SYSTEM_ZONE_MODE_STATE`
volume: current device volume level, 0..100. Some work will be needed to skip intermediate values.

Event information:

All keys are received in EVT_KEY, delivering

```
const CKeyEvent& keyEvent = static_cast<const CKeyEvent&>( msg );
CKeyData keyData = keyEvent.GetKeyData();
```

The buttonPress is available via `keyData.Key()` (values are `KEY_VAL_PLAY`, `KEY_VAL_PAUSE`, `KEY_VAL_STOP`, `KEY_VAL_PREV_TRACK`, `KEY_VAL_NEXT_TRACK`, `KEY_VAL_THUMBS_UP`, `KEY_VAL_THUMBS_DOWN`, `KEY_VAL_BOOKMARK`, `KEY_VAL_POWER`, `KEY_VAL_MUTE`, `KEY_VAL_VOLUME_UP`, `KEY_VAL_VOLUME_DOWN`, `KEY_VAL_PRESET_1`, `KEY_VAL_PRESET_2`, `KEY_VAL_PRESET_3`, `KEY_VAL_PRESET_4`, `KEY_VAL_PRESET_5`, `KEY_VAL_PRESET_6`, `KEY_VAL_AUX_INPUT`, `KEY_VAL_SHUFFLE_OFF`, `KEY_VAL_SHUFFLE_ON`, `KEY_VAL_REPEAT_OFF`, `KEY_VAL_REPEAT_ONE`, `KEY_VAL_REPEAT_ALL`, `KEY_VAL_PLAY_PAUSE`, `KEY_VAL_ADD_FAVORITE`, `KEY_VAL_REMOVE_FAVORITE`).
The eventOrigin is available via `keyData.Producer()` (values are `KEY_PRODUCER_GABBO`, `KEY_PRODUCER_CONSOLE`, `KEY_PRODUCER_IR_REMOTE`, `KEY_PRODUCER_LIGHTSWITCH_REMOTE`, `KEY_PRODUCER_ETAP`, `KEY_PRODUCER_BOSELINK_REMOTE`).

The AudioInterface sends `EVT_AUDIO_STATUS` delivering

```
const CAudioEvent& evt = static_cast<const CAudioEvent&>( msg );
```

The transportEvent is available via `CAudioEvent.GetAudioPipelineState()` (values are `STOPPED`, `BUFFERING`, `PLAYING`, `PAUSED`).

The URLs to POST to:

https://streamingint.bose.com/data/{device-id}/event/started

https://streamingint.bose.com/data/{device-id}/event/ended

https://streamingint.bose.com/data/{device-id}/event/skipforward

https://streamingint.bose.com/data/{device-id}/event/skipback

https://streamingint.bose.com/data/{device-id}/event/stopped

https://streamingint.bose.com/data/{device-id}/event/paused

https://streamingint.bose.com/data/{device-id}/event/unpaused

https://streamingint.bose.com/data/{device-id}/event/preset-pressed

https://streamingint.bose.com/data/{device-id}/event/volume-up-pressed

https://streamingint.bose.com/data/{device-id}/event/volume-down-pressed

https://streamingint.bose.com/data/{device-id}/event/aux-pressed

https://streamingint.bose.com/data/{device-id}/event/skip-forward-pressed

https://streamingint.bose.com/data/{device-id}/event/playback-pressed

https://streamingint.bose.com/data/{device-id}/event/like-pressed

https://streamingint.bose.com/data/{device-id}/event/dislike-pressed

https://streamingint.bose.com/data/{device-id}/event/playpause-pressed

https://streamingint.bose.com/data/{device-id}/event/power-pressed

It has been confirmed that Data Server uses the {device-id} in the format of URLs above and device id in Post Content as an opaque unique id. Therefore it does not matter what to send as device is as long as it can be tied to a particular user or system and grouped together. This Design specifies use of Network Card Address Device Identifier in the format of content and the URLs above.

The SCM software will use the following syntax to obtain this device id from system configuration:

Persistence::SystemConfigurationStore::Get().GetDeviceId()

The corresponding ***sandbox*** unified resource names for use during development start from the following prefix http://10.66.10.184:8082/data-collection/data/.

The corresponding ***EFE*** unified resource names for use during development start from the following prefix https://streamingefeint.bose.con/data/.

5.2 Network Information Capture.

The knowledge about actual environment where Bose devices are operating is very important.

First: we capture network information to learn about current network environments and feed this information back to further research and development to come up with advanced up to date products exceeding competition.

Second: we capture network information to be able to monitor and predict various trends and directions of change in global network environments where Bose devices operate.

This chapter was contributed by Network Manager Team and represents SCM vision of implementation architecture of network information capture. The Network Manager Team will be the main customer of this data in R&D.

Network Manager Team (Jon Cooper) recommends the following design:
Create a new class NetworkDataCollector (NDC).
The existing NetworkServicesController (NSC) will instantiate this class.
When the NSC receives events from the NetworkManager (NM), we'll have the NSC invoke the NDC.
Then NDC will cache information as needed and make decisions when to send events which will be consumed by the StatsDataCaptureTO (SDC).
The SDC won't register for any of the existing NM events. The SDC will register only for the new event(s) coming from the NDC. So, rather than try to leverage and extend the existing NM events, we'll isolate the data collection code more from the existing network code and minimize the likelihood we'll destabilize networking. Also, the NDC can collect any arbitrary information that we internally in networking code do not use for except in data collection (for example DHCP lease time).

6. Network Data Collection

The (new) Network Data Collection (NDC) object is instantiated by the Network Services Controller (NSC) in the BoseApp process. The NSC receives events from the NetManager (NM). The NSC invokes the NDC as necessary to pass network event information to the NDC. The NDC caches information as necessary and sends events whenever the Network Services Controller indicates the Shelby unit has connected to the network, but not more than once per hour.
StatsDataCaptureTO registers for events from the NDC and posts the protobufs to the specified URL.

6.1 EVT_NETMANAGER_DATA_COLLECTION

URL path: /data/device/{deviceid}/network
Fields:
bool wireless
    True if connected via Wi-Fi, false if connected via Ethernet.
string subnet
    In “slash” form with the DHCP-allocated IP address, e.g., “192.168.2.5/24”.
int32 leaseSeconds
    DHCP lease duration (total, not remaining).
repeated WiFiProfile profile
    All configured profiles. Uses the existing WiFiProfile message definition.
    string SSID
    int32 priority
    string security
    string passphrase
    string wepKey
    bool encrypted
    int32 lastConnected (POSIX time)
optional int32 activeProfile
    Index of the Wi-Fi profile in use, 0..N-1. Present only when on wireless.
optional AutoChannel
    AP mode automatic channel selection data points. Present only when AP mode auto channel selection has finished since boot.
    See WiFiManager::AccessPointAcsStats for more information.
    int32 channel (selected channel: 1, 6 or 11)
    int32 ccaBusy1 (unit-less indication of how busy channel 1 is RF-wise)
    int32 ccaBusy6
    int32 ccaBusy11
    repeated int32 numBssids (number of APs seen on each channel 1..13)
optional string scan
    Compressed (bz2) binary blob of iw scan’s output as collected early during boot.
    Sent no more than once per boot.
optional StationStats

See WiFiManager::StationStats for more details.
int32 rssi_dBm
int32 linkSpeed_Mbps
int32 noise_dBm
int32 frequency_kHz
int32 width_kHz
int32 averageRssi_dBm
int32 txGood_packets
int32 txBad_packets
int32 rxGood_packets
int32 tryAuthenticate
int32 tryAssociate
int32 connected
int32 disconnected
int32 handshakeFailed
int32 ssidTempDisabled

7. Design Elaboration

7.1 Object Oriented Analysis and Design.

Analysis of Objects that will be handled by UDC Subsystem:

- Target uniform resource name of the Data Server.
- Context information, includes track information and device information, capture on/off flag.
- Depending from chosen filtering architecture Map of All Collect Able Fields may be maintained or hardcoded. MACAF may be used to reinitialize the capture map.
- Depending from chosen filtering architecture Current Map of Collected Fields may be maintained or hardcoded. CMCF contain fields that are actually captured, this map is the same or a reduced version of the above map and is obtained by application of the filter on the above map. CMCF may be hardcoded when such solution is chosen every field must pass through filter at the time it is about to be collected.
- Default Blacklist Filter equivalent of Send Nothing.
- Active Blacklist Filter.
- Fetched Blacklist Filter. The FBF replaces ABF after successful filter fetch HTTPS transaction.
- Failure Counter. Keeps number of Usage Data posts that have failed.
- Post Content is a string representation of JSON Tree that is to be send with the usage data post request that is currently in progress.

Design approach chosen is to build a toolkit around the feature to produce a flexible toolset that can be customized to changing requirements and to requirements that are TBD. For Example CMCF may be created dynamically for every post

Data formats used by UDC Subsystem:

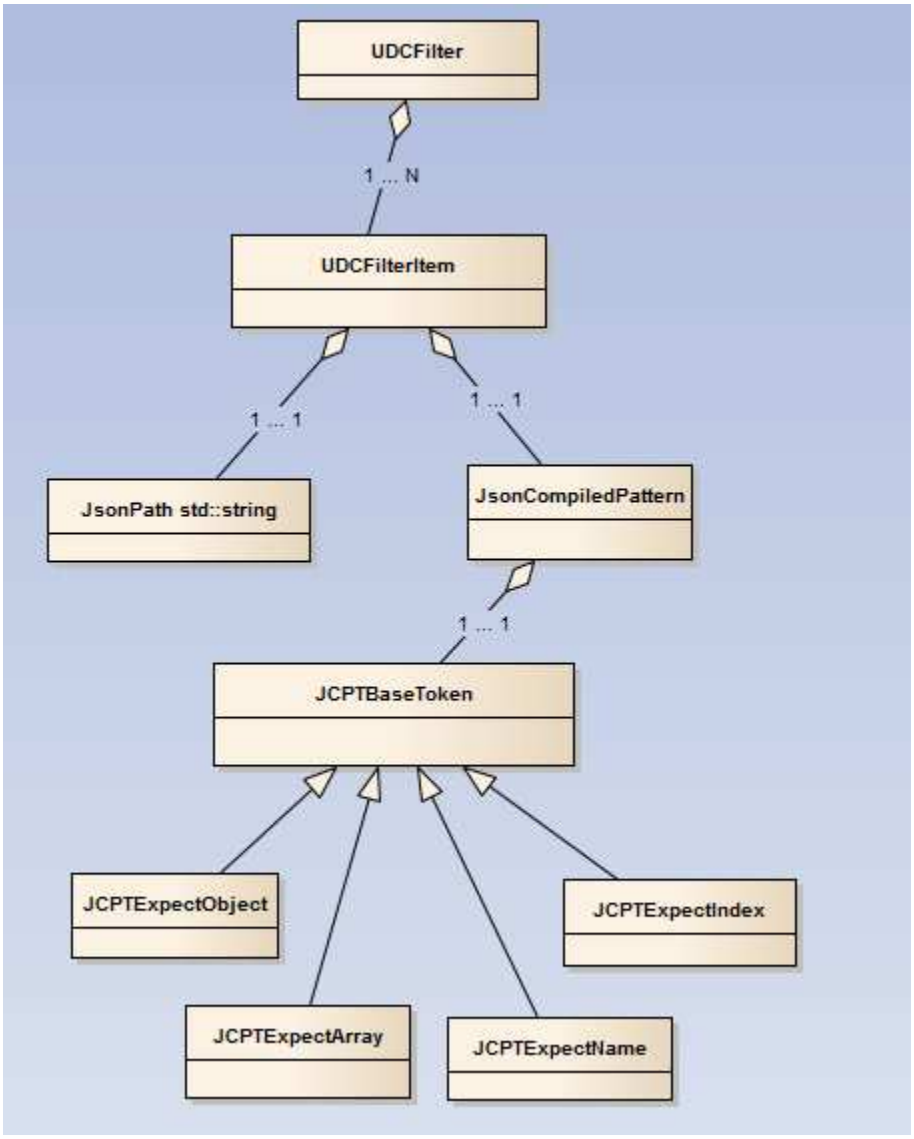
JSON Path is a string, examples of JSON Path: **event.eventOrigin**, **device.boseID** or just **device.** The set operation given the JSON Path and JSON value traverses a path creating new object or array nodes as necessary and attaches a value to a leaf node at the end of a path.

Compiled JSON Path is a chain of objects that instruct the processing program how to traverse a hierarchical JSON Object Tree. We can speak of setting a value at some JSON path. This will work similarly to *linux mkdir –p command*, the path in an object tree will be created to full depth of its specification and value will be set at a final leaf node specified.

JSON Filter Item is an element of the blacklist filter that blocks a send of single sub tree. The filter item may refer to a sub tree or to a leaf node.

JSON Do Not Send Filter is a set of strings that are matched against the JSON Path whenever one of the strings from the filter set matches a JSON Path in a value set request the set operation does not proceed to set Value any further and bails out.

Object Oriented Unified Modelling Language Diagram of JSON Do No Not Send Filter Implementation



UDCRequestBuilder a class that contains logic to build all components of a Usage Data Collection Post Request

UDCFilterItem a class that unites JSON Compiled Path to set fields and JSON String Path for filtering using set belongs/does not belong operations. The class insures that Compiled Path and String Path remain equivalent at all times. This class is of potential dual use. First possible use it can represent a field as a member of a send set to be formatted and posted. Second use it can represent a field as part of the filter.

UDCFilter a class that contain information of what fields are not allowed to capture.

UDCFilterFactory a class that builds the Fetched Blacklist Filter from raw content received in response to Get Blacklist Filter HTTPS transaction.

UDCBlackListXmlParser a class that parses Data Server Filter XML using TinyXML library. TinyXML library is used by many other parts of BoseApp and is already built in that is why there in no extra memory allocation to use it.

UDCInternalException programming style that relies of heavy use of exceptions is indispensable in certain application areas, the parsing and setting of a values in a JSON that is expected to be in a particular format may be one of them. Therefore one or more exception classes will be introduced that are local to Usage Data Collection. All the introduced in this design exception classes will be derived from this class. By design any exceptions thrown within StatsDataCaptureTO will be caught within the StatsDataCaptureTO and will not affect anything that is outside of this Task Object.

JsonCompiledPatter this class allows not to repeat a character by character parsing of a filter string after it has been done once. A compiled patter is a sequence of tokens. The JSON compiled pattern may be used directly to set fields on a JSON tree.

JCPTBaseToken base class for the other tokens of a compiled json pattern.

JCPTExpectObject, JCPTExpectArray, JCPTExpectName, JCPTExpectIndex are JSON compiled path tokens that are all derived from JCPTBaseToken.

Conceptual Details of JSON Black List Filter Design.

To illustrate proof of concept of JSON Path and JSON Compiled Path the following sample JSON will be used:


```
{
   "coord":{
      "lon":-0.13,
      "lat":51.51
   },
   "sys":{
      "type":3,
      "id":60992,
      "message":0.0308,
      "country":"GB",
      "sunrise":1420531481,
      "sunset":1420560492
    },
    "weather":[
         {
             "id":804,
             "main":"Clouds",
             "description": "overcast clouds",
             "icon":"04d"
         },
         {
             "id":805,
             "main":"Stars",
             "description": "upcast clouds",
             "icon":"666"
         }
    ],
    "base":"cmc stations",
    "main":{
        "temp":283.05,
        "humidity":93,
        "pressure":1017.4,
        "temp_min":283.05,
        "temp_max":283.05,
        "wind":{
            "speed":1,
            "gust":2.9,
            "deg":202
        },
        "rain":{
            "3h":0
        },
        "clouds":{
            "all":92
        }
    },
    "dt":1420548714,
    "id":2643743,
    "name":"London",
    "cod":200
}
```

For example: **main/wind/speed/** is a path to fetch or set a wind speed.
weather[1/icon/ is a path to fetch or set icon object in the second member of weather array
"main/wind/speed/"="1" , "weather[1/icon/"="666"

JSON Path concept offers uniform mathematically consistent approach to getting and setting values in a JSON object or tree. The character representation of JSON path requires character by character parsing of a path and performing actions on JSON tree simultaneously. The character by character parsing of a path is not the most efficient way. More efficient approach to compile the JSON path to its tokenized representation will be used in this design. The JSON path strings will be compiled to tokenized representation as soon as they used the first time and any subsequent time the JSON path string is used only the tokenized efficient compiled form will be used.

Some Proof of Concept Testing Results for the Above Approach to JSON Filtering.

A command line program was written that presumes that it's every argument starting from argument one is a filter line as defined in Get Blacklist Filter API, see chapter 4.6

Original JSON before any filter applied.


```
{
   "device" : {
      "boseID" : "24387",
      "deviceID" : "D05FB8A9ACC1",
      "deviceType" : "SoundTouch 20",
      "serialNumber" : "24387"
   },
   "event" : {
      "buttonPress" : "none",
      "eventOrigin" : "device",
      "transportEvent" : "started"
   },
   "track" : {
      "album" : "Cant Buy me Love",
      "artist" : "Paul McCarthney",
      "sourceName" : "Abacus.fm Beatles",
      "sourceProvider" : "2",
      "trackTitle" : "Michelle Ma Bell",
      "trackType" : "song"
   }
}
```


A filter that removes all device identifying information and all track now playing information:

```
[bt1007821@hepdsw68 libjson_2014]$ ./udc_filter_test_DOTTED device. track.

TEST RESULT=[
{
   "event" : {
      "buttonPress" : "none",
      "eventOrigin" : "device",
      "transportEvent" : "started"
   }
}

]
[bt1007821@hepdsw68 libjson_2014]$
```

A filter that removes all device information except the device id is applied:

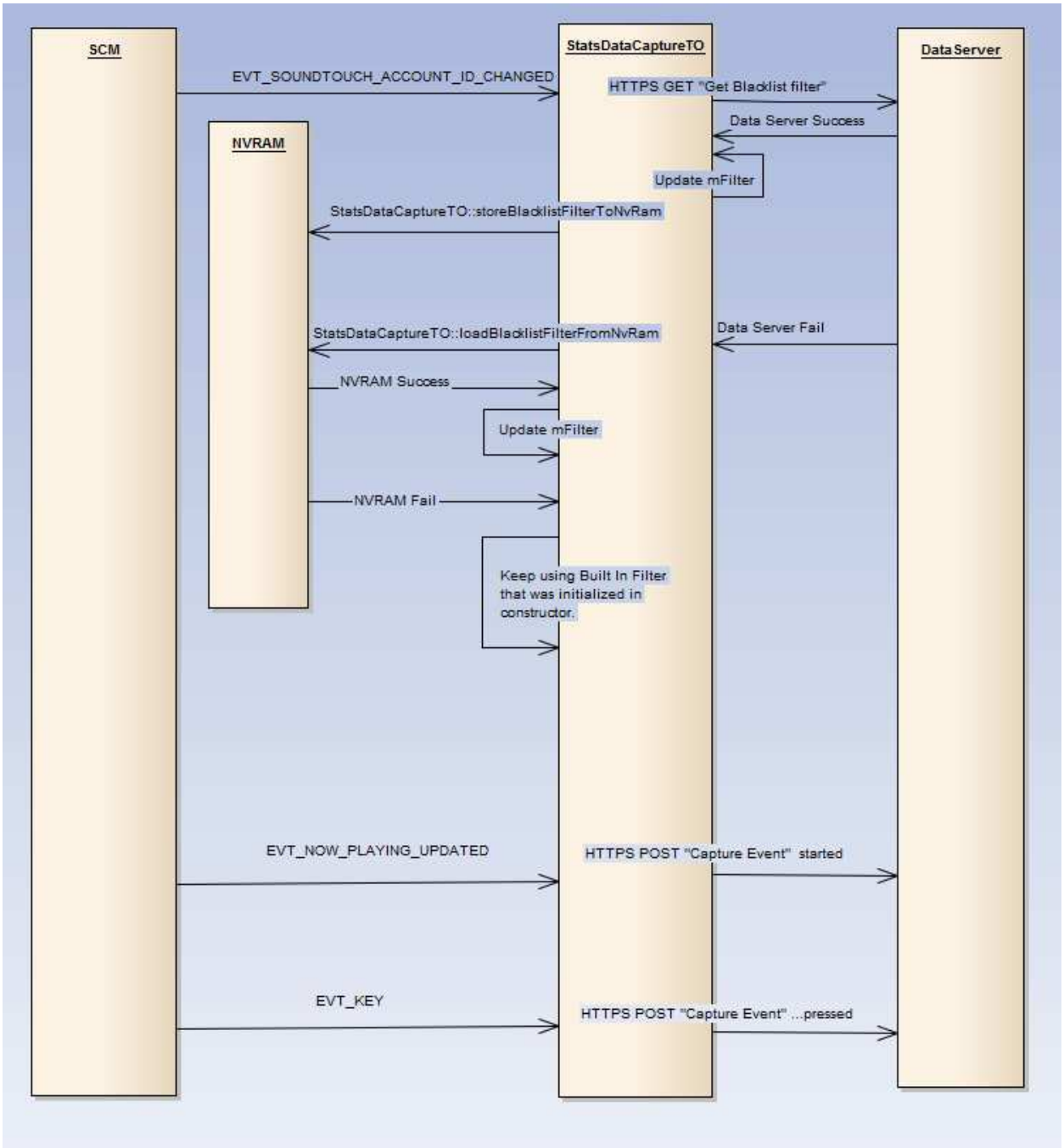
```
[bt1007821@hepdsw68 libjson_2014]$ ./udc_filter_test_DOTTED device.boseID device.deviceType device.serialNumber.

TEST RESULT=[
{
   "device" : {
      "deviceID" : "D05FB8A9ACC1"
   },
   "event" : {
      "buttonPress" : "none",
      "eventOrigin" : "device",
      "transportEvent" : "started"
   },
   "track" : {
      "album" : "Cant Buy me Love",
      "artist" : "Paul McCarthney",
      "sourceName" : "Abacus.fm Beatles",
      "sourceProvider" : "2",
      "trackTitle" : "Michelle Ma Bell",
      "trackType" : "song"
   }
}

]
[bt1007821@hepdsw68 libjson_2014]$
```

The data about the track in the above tests are made up data they might refer to a song that does not exist.

7.2 Process Flow Design.



7.2.1 Program Code Structure and its Place in the System.

The skeleton of Task Object and event capture infrastructure provides framework for implementing on the UDC feature, yet it does not finalize all structure and architecture of the code. Significant liberty of design choices remain that can result either in good or bad design.

Therefore code structure principles have to be finalized so it is understandable why the code of this feature was coded the way it was and not the other way. The following steps in the UDC code must be clearly separated: Event Capture, Context Maintenance, Event Validation, Marge URL Selection, Filter Application, JSON Generation, Posting to Stats Server.

Context Maintenance is keeping state information up to date maintaining shared data that should be available to all capture elements.

Event Validation is a determination whether the event in progress is eligible to be posted to Marge.

Marge URL Selection the internet uniform resource locator that we post to should be configurable and it changes depending on event that is posted and depending on device id. The URL is polymorphic by event type.

Filter Application. It is a requirement that before JSON field is set it will be validated against an efficient "Do not send" filter that fields masked by filter will be not included in outgoing JSON.

JSON Generation. The implementation of this is polymorphic by event type, the outcome of this is a text string representing properly formatted JSON that will be send as a content of POST request to UDCS. Whenever values are set to JSON tree to be sent to UDCS the full JSON path that leads to a value will be validated against similar paths that are stored in filter only values that are not blocked by filter will actually affect the JSON generated for POST request.

Posting to Data Server. This task includes serialization and send out of POST request to UDCS, that includes TCP socket connection, retries and synchronous vs. asynchronous. The existing infrastructure related to **Http::StartAsyncSimpleURLFetchRequest** function will be used.

Failure Count Maintenance and usage. Failure count may be used by a device as a reason to expedite configuration update or software update. Or it may be used to inform Bose that some devices unable to contact Bose for Data Collection through the alternative mechanism of communication. For example statistics on failures of data collection posts may be send to Bose together with software update request or at the end of calendar period like a month using separate transaction.

7.2.2 Logging Considerations.

The logging is an important feature of any embedded design, very often it is more useful in diagnostic of faults and in validations that system is working as expected. Logging can also be used by automated testing scripts to confirm success or failure of the tests that however has a drawback that in a future releases logging messages will not be able to change without breaking automated testing scripts.
The logging also imposes a performance penalty on the overall system.
The logging becomes almost useless when there is too much of it.
Therefore right middle way decision should be made on what to log and what not to, how to enable and disable parts of logging and possibly provide an alternative log that is active during development only and compiled off at Release build.

Properly placed log messages may be captured and used in Automatic Unit Tests. Log redirection over UDP Protocol or preferable piping logread –f to NFS file system may be used for capture log messages.
It is sometimes desired to use a custom log while program or system in a stage of active development and experimentation that includes developer testing and developer research testing. The above type log by name of developer log was used in the development of UDC subsystem. The advantages of developer log: ability to simplify logging and to see only the messages that are relevant to the task at hand without configuration efforts, independence from errors and bugs in logging facility itself. The developer log is used during development only and in production system can be either disabled by default, completely disabled, or totally excluded from being built in.

Developer log is a simple log to file system where log file is opened in append mode and closed after writing of every message therefore insuring that messages instantly appear in the log. The size of developer log in file system is limited by design to 50 kilobytes so there is no danger that file system is overflown.

7.2.3 Events, State and Concurrency Considerations.

Capture Subscription is a new event subscription that is introduced in Usage Data Collection Task Object in order to consume a significant for UDC purposes event and convert it into Data Server Post.

Capture Element is a new code that is added to the SCM in a course of this project to capture one collectable field at the location in SCM source code where such capture is possible or optimal.

The requirements of this project will be met by introducing multiple Capture Subscriptions and Capture Elements.

Context information by requirements must be accessible to every CE

Context information has device and track info yet only device information available in System Configuration and Persistent Store.

The track information must somehow be made available to every CE.

TO runs within a single thread there are minimal requirements for synchronization.

The objects discussed in chapter 6.1 will become a data members of StatsDataCaptureTO.

The task of maintaining instant up to date **NPTI** is delegated to **StatsDataCaptureTO** task object.

Map of all capture able fields should be hardcoded or initialized from hardcoded data before application gone multithreaded.

The Task Object StatsDataCaptureTO runs within a single thread. The only time this Task Object creates additional threads is when it is necessary to allocate completion callbacks for asynchronous posts.

HTTPS POST Request here is Asynchronous by requirements (see item 3.5) therefore the failure count must be able to be safely accessed from a completion handler that is run in another thread. The failure counter may be accessed by multiple completion handler threads whenever multiple completion handlers are outstanding. The failure counted should be protected and tread safe.

An Atomic integer data type will be used for the failure counter.

SCM sends all JSON values to Data Server as string values. The actual type of those values in Google Protocol Buffers definitions may include integers and enumerations (or enums). The enumerations will be send using their integer printed values.

7.2.4 Error processing considerations.

In client server system or in any distributed system the agreement has to be made regarding transition of responsibility between client and a server as far as awareness about errors and error handling. We have to keep in mind that even very important for Bose the data collection has limited value from user prospective that is why anything that is related to data collection has not get in the way of principal usage of the product by user. The user has bought Bose product because he/she wants to enjoy high quality audio and not because he/she wants to provide us with data of what he is doing on the device.

Steps in end to end processing of the message:

1. Message leaves the SCM.
2. Message received by Data Server.
3. Message validated by Data Server, it is confirmed that it is in proper format, URL and JSON wise.
4. Data Server confirms that Message that was received from SCM has a correct semantic. What does that mean? One example of semantically incorrect message would be a message that contains a device id of another device, like two different devices using the same device id is clearly a semantic error, or a device id from a range that is known not to be available in our domain. Or a device id that is shorter than it should be is a semantic error.
5. Data Server further processes the data collection message from SCM and persists it in their database.

It is opinion of the authors of this document that when the correctly according to Interface Control Document or API Specification message is posted to Data Server and received by Data Server and validated for correct JSON format and checked and confirmed to be semantically correct the 200 Ok response to SCM should be issued at that point.

The rationale for such design choice is that unlike many other information items that SCM uses Marge for where information travels in two directions from and to marge the Data Collection represents a strictly unidirectional flow of message from SCM to marge. After SCM has posted a properly formatted and correct message and it was received by Marge it should be of no concern to SCM what marge is further does with the message because SCM will not immediately rely on reading that information back from Marge in its operation. We understand that even SCM does not immediately rely on information kept in Data Server in the future additional service may be offered to user based on previously collected Data about users Bose device use.

7.2.5 Security Considerations.

The importance of protecting PPI (Person Identifying Information) is realized by Bose. Interdepartmental coordination is under way to design and implement such protections and safeguards. Current implementation of Shelby uses HTTPS for communication to Marge. Wireshark and Tcpdump in human readable format. Decision to use HTTPS or secure socket layer protocol has to be made considering the performance vs security requirements tradeoffs. Using encrypted socket communication for everything might be an overkill and whether it does make sense for Data Collection yet TBD. HTTPS or secure socket layer and Public Key Cryptography Infrastructure is provided by OpenSSL library on Linux. The authenticity of peers is verified during SSL connection establishment. Every encrypted POST request to Data Server will require separate connection to be established. SSL connection establishment process allows both side to verify whether the other side is what it claims to be all the way through the chain of certificates till the top lawfully authorized certificates. Need to keep in mind that SSL connection establishment may be programmed in such way that either side may chose to ignore certificate verification.

7.2.6 Transport Timeouts Considerations.

The current implementation of HTTP Post is provided in SCM by SimpleAsyncURLFetcher.cpp that internally uses SimpleURLFetcher.cpp that further uses CURL. The present state of affairs is such that on 02/17/2015 still can't handle a Reply Timeout Failure. RTF is an error when HTTP server on the other end of a connection consumes client request and stops (it may be caused by internal deadlock in the server). The server stops and newer sends a reply to the client nor does server closes a connection. This is an RTF (Reply Timeout Failure).
So somewhere there the select polling can be added to a socket for data available to read. If there is no data available to read in a specific time frame (like 10 seconds for example) the wait should be abandoned, socket shutdown and POST failed reported to upper layers of the software in embedded system.

7.2.7 Exception Processing Considerations.

It is recommended to catch all exceptions that generated by all and every component of Usage Data Collection Subsystem within UDCS itself. Therefore exceptions thrown by UDCS or any underlying layer of application software working on its behalf will be caught within the boundaries of the UDCS in every of the UDCS running threads.

7.2.8 Additional Design Considerations and Details.

Proposed location of capture on/off flag is in SystemConfiguration initially hardcoded, could be changed by tap. The default value is on.

Whenever the StatsServerURL XML element has text of zero length in its value that effectively disables Usage Data Collection.

While configuration XML is a source of some persistent values it is a source of read only persistent values. The UDC Do Not Send Filter has to be persisted in a read/write persistent storage. The WiFi configuration storage may be used as a prototype for coding the DNSF persistence. WiFi support is certainly need to use read write persistent storage to store discovered WiFi network SSIDs and login passwords.

It is belief of the author of this design that usage of full blown PDO is not necessary for persisting the filter. The NVRAM is mapped to Linux filesystem /mnt/nv. The Linux kernel provides tread safety as long as information is written to files that have different names so just picking a unique name for persisted UDC Blacklist filter will insure the isolation of reads and writes to it from other threads.

The UDC DNSF fetch from Data Server performed in two cases:
1) After Reboot.
2) After Network has been disconnected and connected again, there must be an event for that.

The number of times to retry the UDC DNSF fetch is TBD.

The UDC DNSF fetch failure handling. In case of failure we use previous filter. Therefor a filter must be stored not only in StatsDataCapture Task Object memory but also in ***Read Write Persistent Storage.***

The generic implementation of JSON field filtering can be viewed as operation on sets:

<Fields To Send> := <All Fields> - <Fields in Filter>

The implementation therefore can be coded as follows:

Case A: Per every POST request:
For every in <All Fields>

If Field is not in <Filter Fields>
Set(<Field>)

Case B:
Or may be the following way will be more efficient:
Per every filter change
<Fields To Send> = <All Fields> - <Filter Fields>
Per every event post:
Set(<Fields To Send>);

The only difference between the above design approaches is whether we would like to subtract set in advance or subtract every time we set. The seemingly large performance penalty for subtracting every time may be mitigated by cashing the decision made at the posting of first POST event until the filter changes.

Case A is more straightforward to code and it is preferred when main design criteria is "not to over engineer the solution.

7.2.9 Dynamics of Filter Maintenance.

In task object constructor: Create send noting Filter then read NVRAM. Whenever read succeeded replace Send Nothing Filter with filter from NVRAM.

On Power Up and on Network Up (that includes when IP address has changed): Fetch filter from Data Server using HTTP GET. Whenever here is a failure do not update filter in task object. Whenever here is success store the new filter to Task Object and to NVRAM.

The Get Blacklist HTTPS GET transaction by requirements will use the ETag mechanism to distinguish the page that is available yet has not changed. The fetch will not proceed if "Page Did Not Change" response was found. See list of references [10].

Every time the Blacklist filter is saved to NVRAM its ETag identifier must be saved as well.

7.2.10 Persistence Consideration.

There is couple of choices among those full blown NVPDO and a simplified shortcut type code that avoids PDO machinery in its entirety. The initial design choice is implement a simple synchronous access to persistence without PDO and associated with that event overhead. Justification for such simplification is that filter maintenance and event posting to Data Server occur within the same thread therefor access to Persistence for write and for read is only can be only synchronous and blocking. This is within our performance constraints because the actual

blocking time for persistence NVRAM write on current hardware is only 3 milliseconds. To understand that this is acceptable one need to realize that UDC Task Object is only one thread of many in BoseApp and that actual load on UDC Task Object in reality will be very low because events only happen as fast as human operator of Bose product can produce. The ProtoToMarkup will be used for serializing and de serializing data to NVRAM.

This is the format that will be used to persist blacklist, please note that in comparison to API we add e-tag:

```
<?xml version="1.0" encoding="UTF-8" ?>
<blacklist>
   <etag>1425992854236</etag>
   <item>device.boseID</item>
   <item>device.deviceType</item>
   <item>device.serialNumber</item>
   <item>track.album</item>
</blacklist>
```

7.3 Design Details Clarification.

7.3.1 What to do when send nothing or “*” filter is configured.

There are several alternatives either perform a post transaction with content that specifies “null” for a JSON , Perform POST transaction with no content “Content Size: 0”, Do not issue a POST request therefore posting nothing and not making even a connection to data server. Yet there is even one more choice , when the “*” filter is received to wipe out the Data Server URL from configuration, therefore doing an Early Disable of Data Collection in a BoseApp flow.

7.4 Testing considerations.

7.4.1 Manual Testing or Developer Testing.

A typical scenario for manual testing of the UDC is when user drives the SoundTouch PC Application such as Homer and monitors the usage information that is posted to Marge using Wireshark or simulated marge server.

7.4.2 Automatic Testing.

It is desirable to implement an automatic testing for Usage Data Collection feature instead.

Such testing could be implemented using API scripting.

Two additional features should help automated testing:

- External feed for key presses: ***the existing external key press feed should be identified or new developed.*** External feed for key presses should be able to receive a key press event from a socket in some format for example JSON or XML and simulate an actual key press programmatically using the lowest possible level in application layer.
- Simulated server for HTTPs requests that can produce logs or export some API to read failure counts.
- The application should be able to be reconfigured to post to simulated Marge server or to actual Marge server and preferable automatically as part of the test script.

The automated testing program should simulate user activity via external key press feed and monitor appearance of POST requests to simulated Marge Stats server. Alternative to Simulated Marge server would be to add a message about success of a response from Marge at the end of every UDC post as soon as response has been received. Such logs to be generated only in debug mode.

Depending from the complexity of simulated server it can provide several levels of test measurement:

- Simple OK server that responds HTTP 200 OK on every request regardless of request content.
- JSON format validation server that checks whether request Content contains a properly formed JSON.
- Stateless API format validation server that checks not only whether JSON fields properly formatted but also that their fields conforms to standard API to UDC Marge. For example that it can check that every post has device and track information.
- A constant response HTTP simulated server will be used for Testing Fetch Blacklist transaction.

Advantages of having simulated server that SCM should be able to test API separately from Marge and before Marge is ready.
The pass through of the code page characters (characters in range 127 to 255 of ASCII table) in the JSON fields carried by UDC should be tested and confirmed working. These characters used for other alphabets like European characters with accent marks, Cyrillic and others.

The pass through of UTF-8 in a case when they actually carry two byte long characters like Japanese should be tested.

7.4.3 Existing TAP Features that can be used for UDC Automated Testing.

6.3.3.1 There is a command in TAP that is called key that can be used to simulate keypresses:

key <key value> [hold time millisec]

the following key codes are supported

volume_up, volume_down, preset_1, preset_2, preset_3, preset_4, preset_5, preset_6, aux power.

Example:

```
root@spotty:~# tap
->key volume_up 2000
Key pressed and held Waiting for the release...
->Key Release
->Key sent
->OK
->key volume_down 3000
Key pressed and held Waiting for the release...
->Key Release
->Key sent
->OK
->
```

Entering the above commands in tap will cause the device to change the volume up and then down, the volume slider will synchronously move in a paired SoundTouch Homer application as well.

6.3.3.2 Suggested automated test:

Script that logs in to tap over SSH redirects UDC data server to integrated simulated server and produces all keys, the keys will generate key-press posts and corresponding “now playing changed” events whenever preset is pressed. This quite thorough automatic test already may be created using “tap key” and existing simulated server alone. It will be required to script pre provisioned test initialization that will include marge account and known in advance presets.

7.4.4 External Event Feed.

During the development and developer testing of StatsDataCaptureTO some binary dumps of Google Protocol Buffer message WebInterface::nowPlaying event were capture in binary files on ARM Device side. These binary dumps contain serialized to full depth protocol buffer messages. It is possible to write a separate External Event Feed Task Object and add it to SCM to be active in Debug build only. EEFTO will listen on TCP socket and receive a binary serialized event using wrapper protocol that is TBD. After such binary event is received EEFTO will unwrap event from a thin TBD protocol wrapper, de serialize event using google protocol buffers API and push event into an SCM system using means similar to those where the EVT_NOW_PLAYING_UPDATED event is produced. EEFTO can be used for automatic testing of StatsDataCaptureTO.

7.5 Test Plans Considerations.

This chapter represents a developer prospective on a test plan and things that need to be tested the Server Team and QA will probably add to this. Authors of this document do not claim that this list of things to be tested is complete, yet it is a good starting point.

Every transaction type has to be tested. That means every event that has a separate URL in Interface Control Document and/or appears in Data Server API Specification must be tested. The posting to Data Server of a particular event type is further referred as a transaction type.

Things to test for every transaction type: normal operation, all foreseeable types of failures.

Normal operation is understood as successful connection to the Data Server followed by successful POST operation that results in HTTP or HTTPS (whichever we use) 200 OK response that is received in timely manner.

The foreseen types of failures are: intermittent failure, persistent failure, negative reply, connection failure, connection timeout, reply timeout.

The intermittent failure for the purposes of this design assumed a single failure that when retried results in successfully completed transaction.

The persistent failure is a failure that reappears when retried and remains a failure when retried a configured number for times and results in ultimately failed transaction.

To test all of the above cases the capability to simulate an erroneous behavior should be present in the server. This should be taken into consideration by the Server Team.

Negative Reply Failure when server responds with anything else then 200 OK. For example if a script that handles an SCM request has core dumped in Data Server normally the "**500 Internal Server Error" response** will be produced. This response code is a good candidate to simulate failure.

Connection Timeout Failure when connection is not accepted in timely manner by Data Server.

Reply Timeout Failure when connection is accepted and request is consumed yet reply is not produced nor connection closed by server so the client remains in limbo on whether and when the server will respond to the request.

Pre Provisioned Test Case Initialization is a set of initial values that has to be setup in SCM in order for the subsequent test have a meaningful environment to run in. For example if we setting a handling of preset keys and start events that caused by those keys and posting of all of those to data server we need to have something assigned to presets first. That requires some sort of automatic initialization script written that very likely to be reused by multiple test cases.

Logical Testing of UDC Filter. Usage Data Collection Filter is a Do Not Send type filter. A branch of JSON Tree that is to be flattened to text file and posted to Data Server may be masked to Do Not Send by user choice or applicable to geographic location legislation. UDC Filter is tool to perform such Do Not Send masking. Any branch can be masked. A test must be developed for end to end or local testing of correct workings of the UDC Filter. The JSON Path strings that is “not to send” should be added to a filter and removed from a filter, in between those operations on filter buttons should be clicked and do EVENT_NOW_PLAYING_UPDATED processed and post requests to Data Server issued. At the time the automatic test should validate whether outgoing JSON indeed conforms to the filter that was provisioned during the PPTCI.

8. Implementation Details.

1. Configuration element StatsServerURL will be introduced into Configuration.XML.
2. CLI commands will be added for implementation correctness verification.
3. The Json-cpp library that is offered by json/json.h header and by library libjsoncpp.a

will be used instead of ProtoToJson. The Json-cpp library is next layer down below ProtoToJson. Json-cpp library is something that was used to implement ProtoToJson. ProtoToJson will not be used in first release.

8.1 Configuration.

The introduction of new server separate from Marge is required part of the requirements. This new server will accept JSON data from SCM to accumulate and warehouse usage data. The server URL will be stored in Configuration.xml in StatsServerURL element.

```
root@spotty:nv# cat /etc/opt/Bose/Configuration-spotty-dist.xml
<Configuration>
    <ABLPortIPAddress>127.0.0.1</ABLPortIPAddress>
    <ABLEnabled>false</ABLEnabled>
    <AirplaySupported>true</AirplaySupported>
    <APServerURL>http://127.0.0.1:40000</APServerURL>
    <BluetoothSupported>true</BluetoothSupported>
    <DefaultDPrintLogLevel>INFO</DefaultDPrintLogLevel>
    <DisplayAvailable>true</DisplayAvailable>
    <HasStatusLed>false</HasStatusLed>
    <HasWifiLed>true</HasWifiLed>
    <OutOfBoxEnabled>true</OutOfBoxEnabled>
    <KeyDriverAvailable>true</KeyDriverAvailable>
    <MargeServerURL>https://streamingint.bose.com</MargeServerURL>
    <StatsServerURL>http://10.60.14.66:44868/stats</StatsServerURL>
    <SWUpdateURL>http://filestore.bose.com/hed/shelby/trunk/index.xml</SWUpdateURL>
    <DemoAudioURL>aux://stream.pcm</DemoAudioURL>
    <UsePandoraProductionServer>true</UsePandoraProductionServer>
    <DemoNetworkEnabled>false</DemoNetworkEnabled>
    <UseBoseDeviceTypeForSsdp>false</UseBoseDeviceTypeForSsdp>
</Configuration>
```

BOSE URL for the UDC server was standardized at **https://usagestatsint.bose.com,** this server is globally accessible on the internet to Bose equipment installations worldwide.

There is a number of configuration items that may be either hardcoded or statically or dynamically configurable. Such things as: Total Request Timeout between Request to DCS and Reply from DCS, Number of Retries to when Request to DCS fails. These information items will be accessed from multiple locations in the source code where Usage Data are captured.

These configurable information items will locate in **/Core/Infrastructure/DataCaptureCfg** directory.

8.2 Command Line Interfaces for Test Automation Protocol Tool.

CLI Commands Proposed:

Udc status, prints internal state of Usage Data Collection Sybsystem.
udc on, allows to turn Usage Data Collection on
udc off allows to turn Usage Data Collection completely off,
udc fields prints list of all collectable filters some of which may be actually collected and others covered by filter.
udc filterclear allows to send to marge everything that can be sent voids filter.
udc filterexcludepath <json-path> prevents specified data objects from being sent to Marge, internally adds a string to black list filter.
udc filterprint prints entire stored black list filter. Prints all "do not send" strings.
udc dataserver <URL> sets Data Server URL to the specified URL.
udc storefilter saves do not send filter to NVRAM.
udc loadfilter reads do not send filter from NVRAM overriding the current filter.

Please note that normal fetch of DNSF from Data Server using Web API Transaction will update also NVRAM where filter is stored. UDC commands with an exception of **udc savefilter** do not update the image of the filter in NVRAM.

udc log debug retargets logging of all progress (not and error) messages to Debug log level of standard SCM logging facility.
udc log info retargets logging of all progress (not an error) messages to Info log level of standard SCM logging facility.
udc log devlog retargets logging of all progress (not an error) messages to developer log.

8.3 Capturing EVT_NOW_PLAYING_UPDATED.

8.3.1 Implementation Design.

This information will be captured by separate task object residing in SystemController.
Task object name is StatsDataCaptureTO. It is derived from CTMIUser class, like any Task Object should be. Task Object is a Thread Agnostic object that can be part of a Task. The task object subscribes to EVT_NOW_PLAYING_UPDATED.

8.3.2 C++ Sources Added and Modified.

Task object implementation required two new files and modified file.

New files:
App/SystemController/StatsDataCapture/StatsDataCaptureTO.h
App/SystemController/StatsDataCapture/StatsDataCaptureTO.cpp

Modified files to attach new TO to a running Task.
App/SystemController/SystemControllerTask.h
App/SystemController/SystemControllerTask.cpp

Modified Cmake configuration files to enable build of the above task object.
App/SystemController/lib.cmake
Core/BuildConfig/includes.cmake

The configuration of new server URL required changes in the following files:
Core/Infrastructure/Configuration/ConfigurationMgr.h
Core/Infrastructure/Configuration/ConfigurationMgr.cpp

8.4 Capturing Key Press Messages.

The key press capture will be done capturing EVT_KEY event and examining the CKeyEvent. A track information from the last posted EVT_NOW_PLAYING_UPDATED will accompany the the key event when posted to Data Server.

8.5 Capturing Transport Event.

TBD

8.6 Coding considerations.

One of the very important coding considerations that is not usually emphasized is to program in such a way that is not only easy to read but also easy to locate.

We use identifiers that is named in a descriptive way and code indentation to emphasize the structure of the code that makes program easier to read and support.

It is a fact that before code can be read the relevant code must be located.

The code is located using various search tools, like Unix grep command, etag, Eclipse and others.

The consideration is that when we program we also must consider that someone will in the future grep against the code that we write.

9. Expanded Project Scope and Feature to Implement in Future Versions of UDC.

This is separate chapter to document features that were not in the original scope of the project and were added later. These new features will be implemented in next or future version of Usage Data Collection.

9.1 Possible Future Requirements Related to Gabbo (an Android or iPhone SoundTouch Client).

When the event is originated from Gabbo we would like to know what exact instance or installation of Gabbo has originated such event.