OSS SYSTEM ARCHITECTURE

Version 0.1

Abstract

This document was submitted as a candidate Architecture specification, for the IoT problem posed by Ericsson

Venkata R Pagadala

venkat@apache.org

January 2016

Table of Contents

1 Introduction

1.1 Purpose

This document describes the system architecture using different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

This Software Architecture Document provides an architectural overview of the OSS system that is responsible for remote monitoring and managing the power meters deployed by ESB Charging at their consumer locations.

1.3 Definitions, Acronyms and Abbreviations

ESB Charging	A ESB division and the customer for the OSS system
OSS	Operations Support System – The Ericsson software to use components built using this architecture
Power Meter	A smart device that sends energy consumption data to ESB Charging
3G/LTE	Wireless technologies used for communicating with Power Meters
(e) RBS	A protocol for managing the meters through an RBS system
Metrics	Monitoring data received from meters (raw and aggregated)
Dashboards	UI components that display visuals for aggregate data and alerts

2 Architectural Goals and Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. They are:

2.1 Functionality Goals

1. Monitor status of the system of the power meters
2. Manage S/W and configuration of the power meters

2.2 Expected Processing Workload on the System

Meter interface

• Meters:	2 million (adding a buffer of 500K)
• Monitoring events:	6000/sec notifications (once every 5 minutes per meter)
• Configuration events:	25/sec (on daily basis)
• Maintenance actions:	200/min (S/W upgrade, error alerts - once every week)

ESB interface

• Meter readings count:	1/min (from ESB, once every minute for all meters)
• ESB error alerts:	200/min (one error alert for a meter in week)

User Interface (Admin, App Users)

• Dashboard requests:	10/sec (100 concurrent users, with 10 sec think-time)

Note: The system should be capable of handling twice the workload mentioned above with an acceptable performance, to account for variations and cyclic peak-time work loads.

2.3 Data storage sizing

Entity data (fixed-size):	200 GB (100K for each meter and related data * 2M meters)
Monitoring data (raw):	10 GB/day (meter ID, buffer fill %, meter health stats - 20 bytes per event)
Aggregate metrics:	1 GB/day (processed analytics/reports data - hourly and daily aggregates, anomaly data)
OSS Alerts data:	100 MB /day (created by OSS while processing the raw monitoring data)

2.4 Non-functional Requirements (Architectural Qualities)

1. Real-time processing of meter data by matching the velocity of incoming data
2. Sub-second response times for dashboard requests
3. Security for accessing the data reports and performing management actions
4. High availability for metrics processing, dashboards and alert systems
5. High maintainability through visibility into system components and runtime situations
6. Allow for faster feature development and feature change or bug-fixing
7. Allow for quick deployment of the system and upgrade of components
8. Compatibility with different types of management interfaces on meters

3 Assumptions

1. The requirements mention that meter sends readings every 6 hours but does not specify who receives the readings (ESB Charging or OSS?). The diagram seems to indicate that ESB Charging receives the readings. So it is assumed that OSS is responsible only for managing the devices (configuration, upgrades, alerts, status monitoring), but not for processing meter readings.
2. The requirements also mention that “ESB Charging will generate # of collected readings”. It is assumed that ESB Charging sends this data to OSS every minute (count of readings collected in every minute). This data is used by OSS as one of the monitoring metrics.
3. Meters are running an App that is capable of sending monitoring metrics, alarms to OSS over 3G/LTE.
4. The agent app running on the power meter supports receiving configuration changes, S/W upgrades and queries for the status and other information.
5. The ESB Charging supports integration using REST API for communicating with OSS.
6. It was not clear from requirements about the expected number of meters to manage, as it mentions 1 million in 3 years, but also mentions 1.5 million in the 3rd year. So it is assumed that the total number of meters to be managed by OSS in the 3rd year is expected to be 1.5 million, according to the pace of adding the meters.

4 User Roles

Following are the user roles envisaged to interact with the system

1. Admin User
    a. Configures deployment, scalability, users, security
    b. Provisioning of instances, storage and networking
    c. Configures app and data
    d. Provisions meters, grants API tokens

    e. Configures dashboard views, metrics rate, alert handling
    f. Automates meter configuration changes and S/W upgrades
2. App User (OSS)
    a. Creates reports
    b. Schedules report generation
    c. Creates and configures alerts

5 System Integrations

The following systems will interact with OSS for data communications.

5.1 ESB Charging

ESB Charging connects to OSS over REST/HTTP interface for performing the following interactions.

1. Sends alarms to OSS
2. Send number of meter readings to OSS
3. Send meter activation/de-activation data to OSS

5.2 Power Meter

Power Meter connects to OSS over 3G/LTES using various TCP data formats for the following communications.

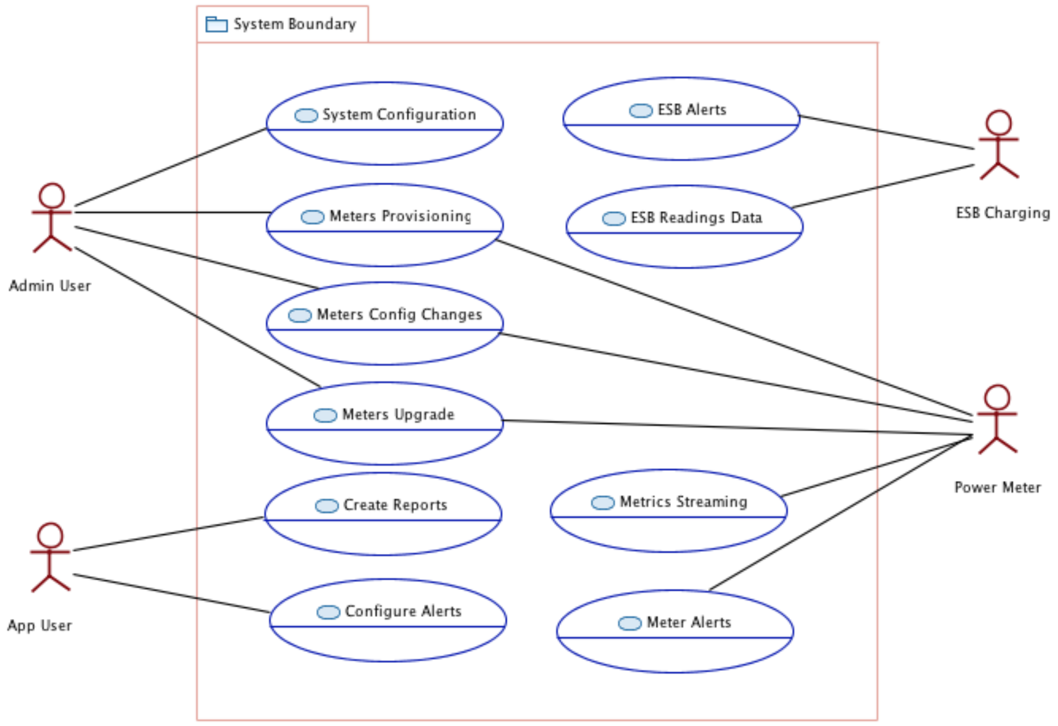
1. Streaming of Monitoring metrics to OSS
2. Send alarms (buffer % low, H/W or S/W errors) to OSS

6 Architectural Representation

This document presents the system architecture as a set of technical decisions and views (use-case view, technology choices, system architecture view) based on goals and constraints.

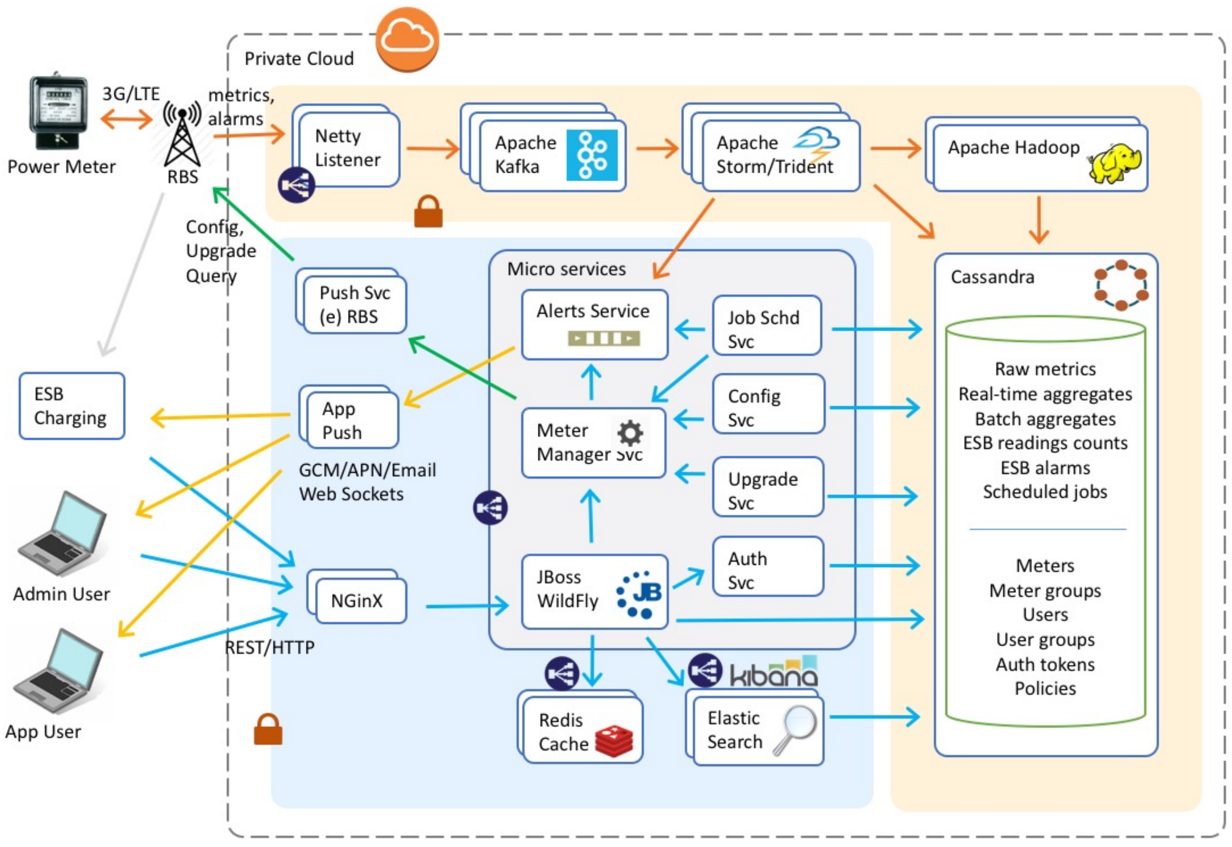
7 Use-case View

The Use-case View describes the set of scenarios or use-cases that have a substantial architectural coverage and represent some significant, central functionality. The following use-case diagram describes the significant use-cases in this system.



8 System Architecture View

The following view describes the System Architecture and its components.



9 Technology choices

9.1 Software Frameworks / Components

Netty	High throughput I/O system for receiving metrics from Meters
Apache Storm	The real-time metrics processing system
Hadoop	Batch processing system for monitoring data
Java	Programming language for implementing most of this architecture
Cassandra	A No-SQL Database system for storing monitoring data and entity data
WildFly (JBoss)	The Java EE container for hosting the application components
Elastic Search	Indexed text search into the meter data and metrics data
Kibana	A framework for data visualizations
AngularJS	UI Technology for composing views for the desktop web-app
iOS, Android	Mobile app platforms for developing the app clients to the OSS system

All backend nodes will run on Linux OS and some instances might use Docker container for virtualization (micro services).

9.2 Deployment Specifications

The following nodes (instances) are considered for deployment for various services.

Node type-1:	4 core 2.4GHz, 24 GB RAM
Node type-2:	16 core 2.4GHz, 128 GB RAM
Storage:	All nodes have 7.2K SATA drives or SSD
Network:	All nodes are connected to a rack switch with full duplex Gigabit Ethernet connection between nodes

Node role	Node type	Cluster size	Parallelism (cores)
Zookeeper	Type-1	3	12
Storm Nimbus	Type-1	2	8
Storm Supervisor	**Type-2**	**4**	**64**
Kafka	**Type-2**	**4**	**64**
Cassandra	**Type-2**	**5**	**80**
Hadoop	Type-1	4	16
Microservices	Type-1	6	24
Elastic Search	**Type-2**	**2**	**32**
Redis	Type-1	2	8
NGinX	Type-1	4	16
Netty	Type-1	4	16
Push Services	Type-1	4	16
Monitoring	Type-1	2	8

10 Data Architecture

The system will use a no-SQL data store (Apache Cassandra) for storage, update and retrieval of data. Cassandra is chosen due to its characteristics favoring time-series data types, high availability, high scalability and high performance.

The following are the significant data tables.

10.1 Event Data (high-growth / variable)

- Monitoring metrics (raw)
- Real-time aggregates
- Batch aggregates
- ESB readings counts
- ESB alarms
- Scheduled jobs
- Configuration changes
- S/W upgrade logs

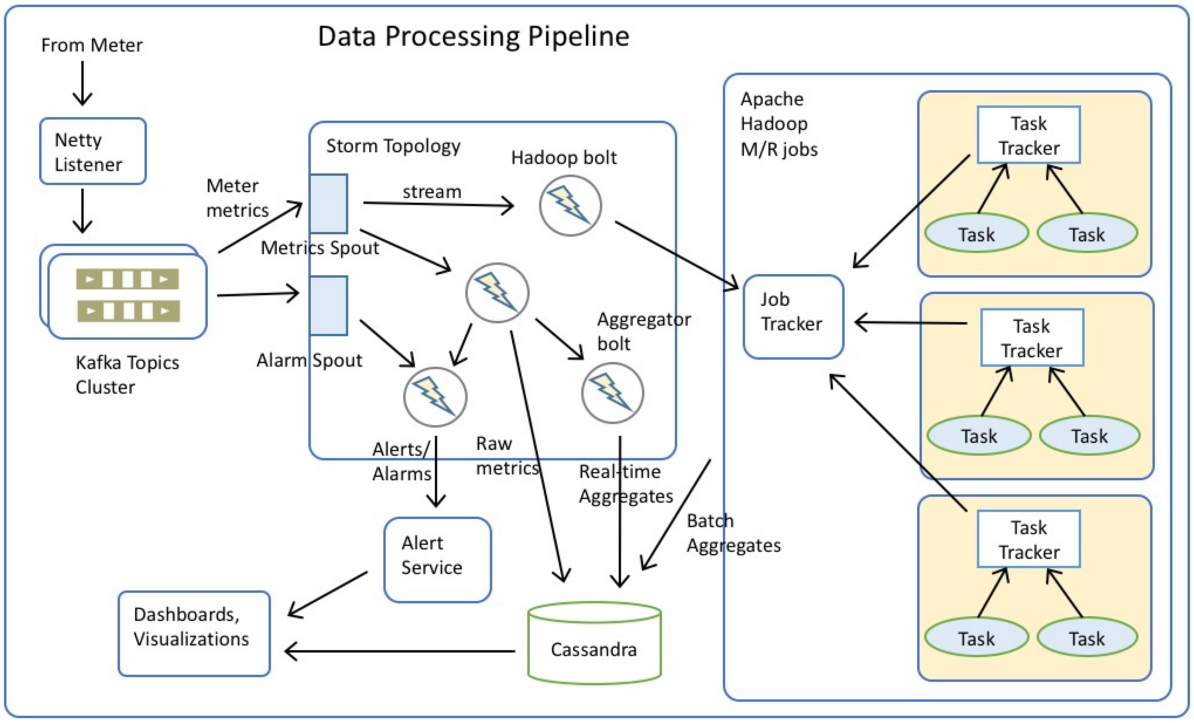
10.2 Entity data

- Meters
- Meter groups
- Users
- User groups
- Auth tokens
- Policies

Data sizing details are discussed in the section 2.3.

11 Data processing pipeline

The following diagram describes data processing pipeline for handling the monitoring metrics data flow.



The Storm topology provides the real-time aggregates such as number meters sending in last minute etc. and anomaly detection for buffer % threshold, or meters check-in count etc.

Hadoop map-reduce jobs provide various batch aggregates used by the dashboard views. The dashboard queries combine the batch aggregates from Hadoop with the real-time aggregates from storm topology to provide a continuous visibility into Meters status. Both Hadoop and Storm use Cassandra for storing the aggregate and raw data.

12 System Components

The following sections provide details about significant components that make up the architecture and the rational for the choices made.

12.1 Security Considerations

The architecture provides the following mechanisms to ensure security for accessing data and services.

1. Admin interface and App User interfaces are protected by the user authentication using password protection.
2. Metrics interface is protected by Authentication tokens provisioned to meters.
3. ESB communication interface is protected by REST API tokens.
4. Data at rest can be protected using encryption service (optional, as privacy related data is not involved).
5. All internal services are protected by the security groups (VPC) or private subnets
6. Firewall rules provide access at the network domain interfaces.

12.2 Metrics Streaming (Meter monitoring)

The Power Meters are expected to be Smart Devices running on an embedded OS (Linux kernel, Android etc) and paly a role of an IoT sensor. The device will have network connectivity over 3G/LTE and can be configured with a destination for sending data to.

Meter data types
A Meter handles the following kinds of data:

1. Outgoing
    a. Power usage readings (to ESB Charging)
    b. Meter monitoring metrics (to OSS)
2. Incoming (from OSS)
    a. Meter configuration
    b. S/W upgrade
    c. Adhoc queries (status, metrics)

Each monitoring metric will have the following data

1. Meter ID
2. Buffer %
3. S/W version
4. Status metrics (variable number)

The meter will also have a memory buffer (SRAM/SDRAM) that can be used to store the meter readings while awaiting the flush out to ESB Charging.

Meters connect to a Radio Base Station (RBS) over 3G/LTE which in turn transmits the data to OSS over IP network.

At the time of initial provisioning, Meters are configured to send monitoring data to the Load Balancer IP address that serves the Netty Listener. The configuration could also be changed later though a configuration push.

12.3 Netty Listener

Netty is a non-blocking I/O (NIO) framework. It is used in this architecture to build a high-throughput listener for receiving meter monitoring metrics without incurring the overhead of HTTP transport. HA Proxy is considered for use as a load balancer for Netty Listener. The Netty Listener will also act the client for Kafka, the next stage in the data processing pipeline.

12.4 Apache Kafka

Apache Kafka is a distributed publish-subscribe messaging system. It provides high throughput persistent messaging for parallel data loads. It uses compression to optimize IO performance and mirroring to improve availability, scalability.

Kafka writes the message immediately to page cache and disk queue rather than holding it in memory. Page cache is managed more efficiently than garbage collected memory. Kafka is shown to be faster than other messaging systems in performance benchmarks.

Our Kafka component will have separate topics for monitoring metrics and error alerts. The topics are subscribed by Storm spouts for consumption, in the next stage of data processing.

12.5 Apache Storm

Apache Storm is a general-purpose, event-processing system for real-time data stream processing. Storm uses a cluster of services for scalability and reliability. In Storm terminology, we create a topology that runs continuously over a stream of incoming data, which is analogous to a Hadoop job that runs as a batch process over a fixed data set and then terminates.

The data sources for the topology are called spouts and each processing node is called a bolt. Bolts can perform arbitrarily sophisticated computations on the data, including output to data stores and other services.

We use Storm for real-time processing of the data to generate alerts, aggregates and feed the data to Hadoop, the batch stage in out data processing.

12.6 Apache Hadoop

Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. This architecture uses Hadoop for batch processing of meter monitoring data with the goal of generating aggregate metrics, anomaly detection and data projections.

12.7 Apache Cassandra
Apache Cassandra is a distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra is proposed as the data store technology for all data needs in this architecture, because its characteristics match the requirements.

12.8 NGinX
Nginx is a web server with a high concurrency, performance and low memory usage. It can also act as a reverse proxy server for HTTP, HTTPS protocols, as well as a load balancer and an HTTP cache. This architecture uses NginX as a load balancer for the REST/HTTP connections.

12.9 JBoss (WildFly)
JBoss (version name: WildFly) is a Java EE container and web servier. JBoss is used in this architecture for the REST endpoints implementation and also as the container for Java-based micro services. RestEasy, a JBoss implementation of JAX-RS specification, is considered for the REST endpoint implementations, while JSON would be the data format choice.

12.10 Micro Services
Micro-services is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using APIs. These services are highly decoupled and focus on doing a small task, facilitating a modular approach to system-building.

All the request-response style backend processes in this architecture, such as Authentication service, will be implemented as micro services. A Linux container (LXC) technology such as Docker would be utilized for optimal deployment of the services on virtualized infrastructure.

12.11 Push services
The architecture uses two kinds of push services – the app push service and the RBS push service.

The app push service is an implementation for sending out alerts to mobile and desktop clients using Google Cloud Messaging (GCM) API, Apple Push Notification (APN) API, web sockets for desktop web clients and email.

The RBS push service is a device management client that can push configuration and S/W upgrades to Power Meters over the (e) RBS protocol. The RBS push service will support different Meter platforms which support different device management interfaces.

13 Monitoring the Deployment
The production deployments will use the following monitoring and instrumentation services or tools for visibility into system operations and issues

1. Nagios
2. Loggly
3. Rsyslog

14 System Maintenance

The system will need maintaining the data schema changes, data migration across version, an upgrade for software while keeping the system stable. The following sections discuss these concerns.

14.1 Data Migration and Schema Changes

Data migration and schema changes in Cassandra can be accomplished through CQL scripting. The scripts should be maintained in a versioning system.

14.2 S/W Upgrade

All software upgrades will follow a Versioning and Releases process to ensure that the all component are compatible with each other. A Release Notes document specifies the dependencies versions. The S/Wupgrade process should be automated to update the dependencies as well, as needed.

15 References

http://netty.io
http://www.confluent.io/blog/stream-data-platform-1/
http://storm.apache.org
https://github.com/apache/storm/tree/master/external/storm-kafka
http://hadoop.apache.org
http://cassandra.apache.org