How to Leverage GOSS: GridOPTICS Software System in a Research Environment

A Novel Software Framework for Integrating Power Grid Data Storage, Management and Analysis

3rd Workshop on Next-Generation Analytics for the Future Power Grid
July 16-18, 2014
Overview

- GOSS is a middleware architecture designed as a prototype future data analytics and integration platform
- [https://github.com/GridOPTICS](https://github.com/GridOPTICS)
- What does that mean?
  - **Supports heterogeneity** – ease of integration with new/existing power grid applications developed in different languages
  - **Data source abstraction** – separates data sources from applications and provides a unified application programming interface (API) for access
  - **Rapid development** – Quickly make new data/events available to other applications integrated with GOSS
  - **Real-time** – subscription to streaming data and events
  - **Reliability** – provides redundant data access for improved reliability
  - **Security** – role and data based access control
  - **Scalability & Performance**
Architecture

Power Grid Operation, Planning, or Visualization Applications

GOSS Client Adapter

Message Bus

Web Services

Data/Event Flow in Both Directions

Authentication and Authorization Security Layer

GridOPTICS Data Management Middleware and Event Processing Framework

Data Source Interface

PMU Data Source (Raw/Cleaned)

SCADA Data Source (Raw/Cleaned)

Event/Alerts Database

Power Grid Model Data

Forecasting Model Database

Cloud Data Source

Multiple Types Live Streaming Data
Sample GOSS applications: GCA

- **Graphical Contingency Analysis (GCA)** - a C# visual analysis application that aids power grid operators and planners to effectively manage potential network failures (N-1)

- GOSS simplified the application by allowing us to combine all input files (power system model, SCADA, power-flow) into a single data source instead of managing multiple files separately.

- Data source abstraction allowed GCA to work with time-windowed data.

- Application initiates a request for a topology and allows users to select the model to analyze.

- Access is restricted by roles. For each utility, access is granted to a set of roles and the user must be in one of these roles in order to access the data for that utility.
Sample GOSS applications: NIS

- Net Interchange Schedule (NIS) a MatLab application that displays the sum of the energy import and export transactions between an Independent System Operator (ISO) or a Balancing Authority and neighbors. NIS forecasting (NISF) application was developed to aid the ISOs in economically dispatching the generation resources.

- The original application used manually formulated files for the desired time series. With GOSS can use a light-weight client adapter and any time series.

- Now able to re-use the algorithm with different data types.

- The input is controlled the same as other PMU data sources, the application will only have access to PMU streams that the user has been granted access to.
Based on Proven Technologies

- Project Development
  - Java
  - Apache ActiveMQ

- Deployment
  - Apache Maven
  - OSGI via Apache Karaf

- Security
  - LDAP
  - JAAS
  - SSL
GOSS Security & Request Flow

Client API

JAAS Authentication

GOSS Data Management Layer

1. Access Control Lookup and Check

Request

JAAS Authentication

Credentials

2. Request Handler Lookup

Request

Response

Roles

Security Handlers

- Powergrid AC handler
- Forecasting AC handler
- PMU AC handler
- Event AC handler

Returns allowed roles

Request Handlers

- Powergrid model handler
- Forecasting model handler
- PMU request handler
- Events request handler

Processes request, Returns Data

July 18, 2014
Authentication – uses widely accepted tools already integrated into communication platform

- **Java Authentication and Authorization Service (JAAS)**
  - Easily substitute login modules

- **Lightweight Directory Access Protocol (LDAP)**
  - Open, industry standard application protocol for accessing and maintaining distributed directory information services

- **Transport Layer Security/Secure Sockets Layer (SSL)**
  - Cryptographic protocols to provide communication security
GOSS Access Control

- Access Control – customizable for each data source
  - Request Specific Security Handlers
- Security Handlers map request to list of allowed roles
- User verified for correct role access
- Multi-role Access
  - Request combining multiple sources
- Handler implementations for common data types
  - Time series data
Test 1: Comparison of *average* time taken by data store and GOSS individually in total READ request processing time
- Data size ~700 KB
- Number of requests = 4,000
- Number of Clients = 1
- Each client executed in separate thread.

Test 2: Request processing time with increasing number of concurrent READ data requests
- Each client sends 10 requests
- Data size ~700 KB
- Each client executed in separate thread
Synchronous Performance After Enhancements

- GOSS Overheads using same method as previous slide
  - Before enhancements, security adds almost 100% increase
  - After enhancements, reduced to only ~10%

**Overhead in ms**

- No Security: 9.8842 ms
- Security w/o Enhancements: 19.6122 ms
- Security w Enhancements: 10.0631 ms
Performance Benchmarking Analysis

- Per Client Request, processing time is stable even with increasing number of clients
- Scales well with increasing load
- Total time spent inside GOSS includes not only data access but also:
  - Data routing between data source and application
  - Query conversion. Generic query format to data store specific query (e.g., SQL)
  - Result conversion. Converting the results to format requested by the application (including object transformation). Eg., JSON, XML, Serialized Object, etc.
  - Security and access control
- Tests show results in “synchronous” access mode. Asynchronous access hides most of these latencies via pipelining.
- Real-time applications likely to use either event-based or asynchronous access.
Future Tasks

- **Synthetic Data Generation**
  - Modify the code as needed to perform research
  - Ability to interface with other applications with lower cost
  - Simulators will be tied to GOSS

- **Fine Grained Security**
  - Certificate based authentication
  - Improved multi-domain support

- **HPC Integration**
  - Access data and launch simulation

- **Fault tolerance**
GOSS Team

- Bora Akyol  
  bora@pnnl.gov
- Poorva Sharma  
  poorva.sharma@pnnl.gov
- Craig Allwardt  
  craig.allwardt@pnnl.gov
- Mark Rice  
  mark.rice@pnnl.gov
- Tara Gibson  
  tara@pnnl.gov
Part 2 - Tutorial
OSGI (Open Service Gateway initiative)

- Specification describing modular system and a service platform implementing dynamic component model

- Why?
  - Applications or components (called bundles) can be installed, started, stopped, updated, and uninstalled without requiring a reboot.
  - Application life cycle management
  - Service registry allows bundles to detect the addition and removal of services and adapt accordingly.
Apache Karaf is a platform providing features and services designed for creating OSGi-based servers.

- Enterprise Ready
- Easy maven integrated feature installation.
- SSH administration out of the box (even on windows).
- Web based administration with quick feature installation.
- JAAS Security Model (LDAP for GOSS implementation).
- Bundle and Feature constructs.
Bundles

- Java jar with some extra meta data
- Meta data states what is needed to let a specific jar do its job and what the jar will provide to the osgi environment.

```
Manifest-Version: 1.0
Bnd-LastModified: 1402552860709
Build-Jdk: 1.7.0_25
Built-By: D3M614
Bundle-Activator: pnnl.goss.core.client.GossClientActivator
Bundle-ManifestVersion: 2
Bundle-Name: GOSS Core Client Bundle
Bundle-SymbolicName: goss-core-client
Bundle-Version: 0.1.3.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package: pnnl.goss.core.client;uses:="com.google.inject,javax.inject,javax.jms,org.apache.http.auth,org.osgi.framework,org.osgi.service.cm,pnnl.goss.core";version="0.1.3.SNAPSHOT"
Import-Package: com.google.gson;version="[2.2,3)",com.google.inject;version="[1.3,2)",com.google.inject.binder;version="[1.3,2)",javax.inject;version="[1.0,2)",javax.jms,org.apache.activemq,org.apache.commons.logging;version="[1.1,2)",org.apache.http.auth;version="[4.3,5)",org.fusesource.hawtbuf;version="[1.9,2)",org.fusesource.stomp.jms,org.fusesource.stomp.jms.message,org.ops4j.peaberry;version="[1.1,2)",org.ops4j.peaberry.builders;version="[1.1,2)",org.osgi.framework;version="[1.5,2)",org.osgi.service.cm;version="[1.3,2)",org.slf4j,pnnl.goss.core;version="[0.1,1]")
Tool: Bnd-2.1.0.20130426-122213
```
Features are a Karaf construct that allows multiple bundles to be grouped together.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<features xmlns="http://karaf.apache.org/xmlns/features/v1.2.0">

  <feature name="goss-fusiondb-feature" version="${pom.version}"/>

  <!-- Required Features to be installed -->
  <feature>jdbc</feature>
  <feature>goss-core-feature</feature>

  <bundle start-level="88" dependency="true">mvn:pnnl.goss/goss-fusiondb-common/${pom.version}</bundle>
  <bundle start-level="89" dependency="true">mvn:pnnl.goss/goss-fusiondb-server/${pom.version}</bundle>

</feature>
</features>
```
Bundle and Feature Packaging

- tutorial-pmu-core [GOSS-Tutorial master]
  - src/main/java
    - pnnl.goss.datasource
    - pnnl.goss.handlers
    - pnnl.goss.request
    - pnnl.goss.tutorial
    - pnnl.goss.tutorial.datamodel
    - pnnl.goss.tutorial.impl
    - pnnl.goss.tutorial.launchers
    - pnnl.goss.utill
  - src/main/resources
  - src/test/java
  - JRE System Library [JavaSE-1.7]
  - Maven Dependencies
  - src
  - target
  - pom.xml
Building Bundles/Features

GOSS uses maven to layer the building of bundles and features.

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>${maven-bundle-plugin.version}</version>
      <extensions>true</extensions>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
          <Bundle-Version>${project.version}</Bundle-Version>
          <Export-Package>
            pnnl.goss.server.core
          </Export-Package>
          <Import-Package>
            org.apache.log4j; version="[1.2,1.3]";
            provider=paxlogging,
            org.apache.commons.logging;
            version="[1.0,1.1]"; provider=paxlogging,
            *
          </Import-Package>
          <!-- Allows import of packages that haven't been registered before
               this service to still gain reference. -->
          <DynamicImport-Package>
            *
          </DynamicImport-Package>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
```
Java Integration API

- To create a connection
  ```java
  new GossClient(new UsernamePasswordCredentials("username", "password"));
  ```

- To publish events
  ```java
  client.publishTo(topic, event)
  ```

- To subscribe to data or events
  ```java
  client.subscribeTo(topic, eventProcessor)
  ```

- To access data
  ```java
  request = new TopologyRequest(model);
  response = client.getResponse(request);
  ```

- To save data
  ```java
  request = UploadRequest(data, dataId)
  response = Client.getResponse(request)
  ```
Tutorial Example

- **Requirements**
  - As a utility we need a tool to calculate and publish/read the phase angle difference between two PMUs.

- **Testing**
  - We should verify that our implementation produces the correct value.

- **Simplifications for Tutorial**
  - We are using a random function to “generate” PMU values for each of the PMUs.
  - It is trivial to switch to a different streaming method (file, database, live stream)
Tutorial Example - Design

Web Client

Phase Difference Calculator

Desktop Client

PMU Phase Angle Difference Data

```
PMUPhaseAngleDiffData data = new PMUPhaseAngleDiffData();
data.setPhasor1(value1);
data.setPhasor2(value2);
data.setDifference(value);
data.setTimestamp(date);
Gson gson = new GsonBuilder().setDateFormat("PMUPhaseAngleDiffData.DATE_FORMAT.toPattern()")
    .create();

String json = gson.toJson(data);
System.out.println("Publishing " + json + " to " + outputTopic);
client.publish(outputTopic, json);
```

PMU Stream Generator

```
generator1 = new PMLGeneratorImpl(client, Arrays.asList(dataArr1));
generator1.start(pmu1Id, pmu1Topic, itemsPerInterval, intervalSeconds);
generator2 = new PMLGeneratorImpl(client, Arrays.asList(dataArr2));
generator2.start(pmu2Id, pmu2Topic, itemsPerInterval, intervalSeconds);
```
Stage 1 – Virtual machine

- Start virtual machine (auto-login, account: goss/goss)
Stage 2 – Start server

- Open Shellll (Shift+Alt+T)
- Completed tutorial is located at /opt/goss-tutorial-complete

```
goss@goss-vb:~/opt/goss-tutorial-complete$
goss@goss-vb:~/opt/goss-tutorial-complete$ cd /opt/goss-tutorial-complete/
goss@goss-vb:~/opt/goss-tutorial-complete$ bin/karaf

```

Apache Karaf (3.0.1)

Hit `<tab>` for a list of available commands and `[cmd] --help` for help on a specific command. Hit `<ctrl-d>` or type `system:shutdown` or `logout` to shutdown Karaf.

karaf@root()>
Stage 3 – Web client

Open browser to http://localhost:8181/pmu-tutorial/index.html

GridOPTICS Tutorial Workshop
GOSS: Web client Example

<table>
<thead>
<tr>
<th>Start PMU Stream</th>
<th>Stop PMU Stream</th>
<th>Start Aggregator</th>
<th>Stop Aggregator</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU 1 Phasor 1 Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMU 2 Phasor 1 Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase Difference</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stage 4 – Desktop Client

- Double click on GOSS_Desktop_Client folder
- Double click on TutorialClient.py
Stage 5 – Let’s see the code

- Server side components:
  - Tutorial-pmu-feature
    - tutorial-pmu-server
    - tutorial-pmu-common
    - tutorial-pmu-web-ui
  - tutorial-pmu-desktop-ui
  - Install server side components
  - Start GOSS karaf server