DRAFT

Final Report

AFRL SBIR Phase 3 Project

Indiana University
Sensor Cloud Project

April 23rd, 2012

TABLE OF CONTENTS

1. SENSORCLOUD ARCHITECTURE
    - 1.1 SensorCloud Overview
    - 1.2 Sensor Cloud Middleware
    - 1.3 Grid Builder
    - 1.4 Sensor Grid
    - 1.5 Sensor Service Abstration Layer
    - 1.6 Container Service
2. USER GUIDES
    - 2.1 SensorCloud components
    - 2.2 Deployment and User Guides
        - 2.2.1 Tutorial of sensor deployment
        - 2.2.2 User Guides for Sensor
    - 2.3 How to develop sensors and clients
    - 2.4 Advanced Guide to develop sensors and clients
    - 2.5 Sensor Cloud development: An overview
    - 2.6 NaradaBroker Distribution
    - 2.7 LDAP Security feature
    - 2.8 Rest Easy
        - 2.8.1 Guide for SensorCloud Client End users
        - 2.8.2 Guide for SensorCloud client developers
    - 2.9 Streaming Web Server
    - 2.10 Configuring dynamic deployment of domains
    - 2.11 OpenStack Compute: Deployment and Overview
3. PERFORMANCE METRICS
    - 3.1 This section will be completed by May 7th!
4. APPENDIX
    - 4.1 APPENDIX A Secure Cloud Computing with Brokered Trusted Sensor Networks
    - 4.2 APPENDIX B Overview of Status of Clouds

1. SENSORCLOUD ARCHITECTURE

1.1 Sensor Cloud Overview

Introduction

Anabas, Inc. and the Indiana University Pervasive Technology Institute have partnered to develop a Sensor-Centric Middleware System hereafter referred to as the Sensor Cloud.

The objective of the Sensor Cloud Project is to provide a general-purpose messaging system for sensor data called the *Sensor Grid Server*, and a robust *Application API* for developing new sensors and client applications. The key design objective of the Sensor Grid API is to create a simple integration interface for any third party application client or sensor to the Sensor Grid Server. This objective is accomplished by implementing the *publish/subscribe* design pattern which allows for loosely-coupled, reliable, scalable communication between distributed applications or systems.

Publish/Subscribe Architecture

The publish/subscribe (pub/sub) design pattern describes a loosely-coupled architecture based message-oriented communication between distributed applications. In such an arrangement applications may fire-and-forget messages to a broker that manages the details of message delivery. This is an especially powerful benefit in heterogeneous environments, allowing clients to be written using different languages and even possibly different wire protocols. The pub/sub provider acts as the middle-man, allowing heterogeneous integration and interaction in an asynchronous (non-blocking) manner.

The pub/sub architecture uses destinations known as *topics*. Publishers address messages to a topic and subscribers register to receive messages from the topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers. Figure 1 illustrates pub/sub messaging.

Message publication is inherently asynchronous in that no fundamental timing dependency exists between the production and the consumption of a message. Messages can be consumed in either of two ways:

- **Synchronously.** A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

- **Asynchronously.** A client can register a *message listener* with a consumer. A message listener is similar to an event listener.

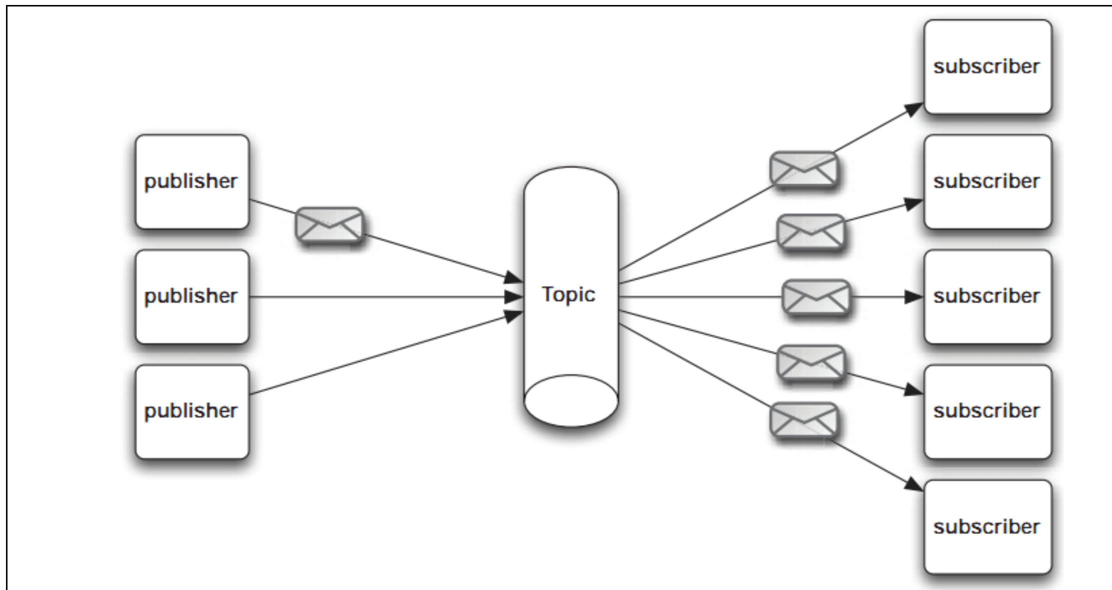


Figure 1 Elements of a Publisher/Subscribe System

A publish/subscribe system can be conveniently implemented using a Java Messaging Service (JMS) compliant Message-Oriented Middleware (MOM) such as NaradaBrokering, ActiveMQ, SonicMQ etc. to handle message mediation and delivery.

Sensor Cloud Overview

The Sensor Cloud implements the *publish/subscribe* design pattern to orchestrate communication between sensors and client applications which form an inherently distributed system.

- Sensor Cloud Server creates *Publisher-Subscribe Channels* (Represented as a JMS Topic)
- Sensors acting as publishers create *TopicPublishers* to send messages to a Topic
- Client applications acting as subscribers create *TopicSubscribers* to receive messages on a topic
- Apache ActiveMQ is used as the default underlying MOM and any other JMS style broker can be used as well.

Figure 2 shows a high-level overview of a typical deployment scenario for the Sensor Grid. Sensors are deployed by the Grid Builder into logical domains; the data streams from these sensors are *published* as topics in the sensor grid to which client applications may *subscribe*.

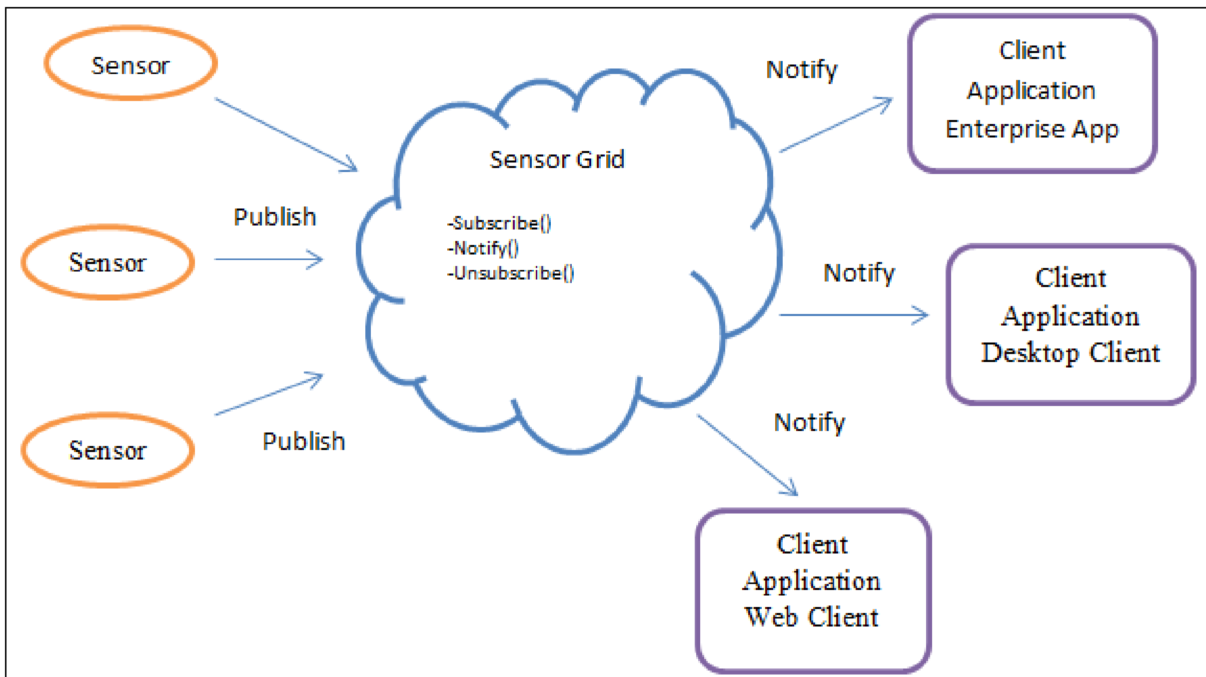


Figure 2 Schematic of the Sensor Cloud

Examples of physical devices already implemented include:

- Web/IP Cameras
- Wii Remotes
- Lego MindStorm NXT Robots
- Bluetooth GPS Devices
- RFID Readers

However Sensors can be made from chat clients, Power Point presentations, web pages virtually anything which produces data in a time-dependent stream can be implemented as a Sensor Grid sensor.

High-Level Sensor Cloud Architecture

The main objective of the Sensor Cloud Project is to design and develop an enabling framework to support easy development, deployment, management, real-time visualization and presentation of collaborative sensor-centric applications. The Sensor Grid framework is based on an event-driven model that utilizes a pub/sub communication paradigm over a distributed message-based transport network.

The Sensor Grid is carefully designed to provide a seamless, user-friendly, scalable and fault-tolerant environment for the development of different applications which utilize information provided by the sensors. Application developers can obtain properties, characteristics and data from the sensor pool through the Sensor Grid API, while the technical difficulties of deploying sensors are abstracted away. At the same time, sensor developers can add new types of sensors and expose their services to application developers through Sensor Grid's Sensor Service Abstraction Layer

(SSAL). Narada Broker (NB) is the transport-level messaging layer for the Sensor Grid. The overall architecture of the Sensor Grid is shown in Figure 3.

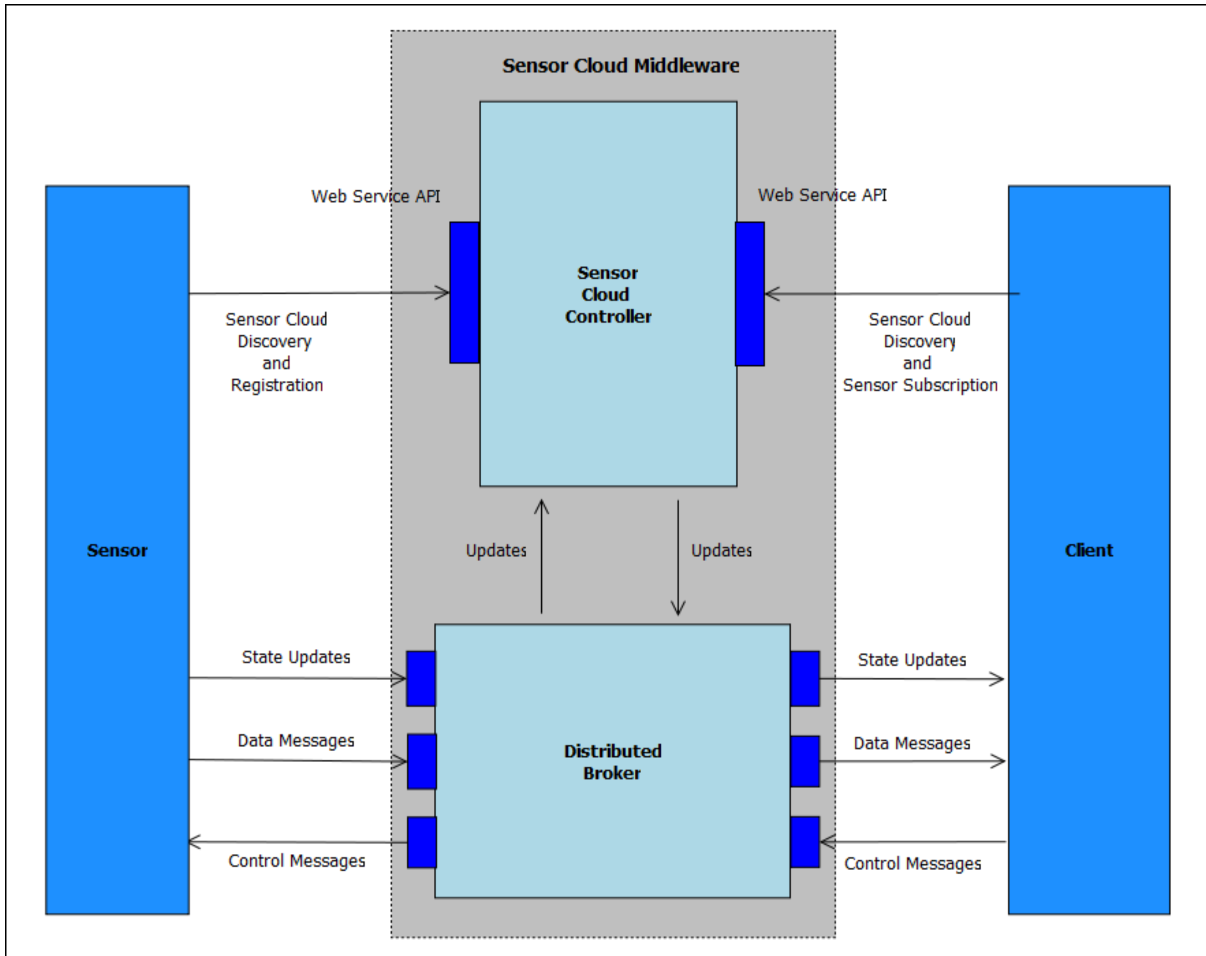


Figure 3 Sensor Grid Components

Sensor Grid Server (SG)

The SG mediates collaboration between sensors, applications and the GB. Primary function of SG is to manage and broker sensor message flows.

- Sensor/SG flow - The SG keeps track of the status of all sensors when they are deployed or disconnected so that all applications using the sensors will be notified of changes. Sensor data normally does not pass through SG.
- Application/SG flow - Applications communicate application API, which in turn communicates with SG internally. Applications can define their own filtering criteria, such as location, sensor id, and type to select which sensors they are interested in. These filters are sent to SG for discovering and linking appropriate sensors logically for that application and forwards messages among the relevant sensors and that application. SG must always check which sensors meet the selected filter criteria and update the list of relevant sensors accordingly. It then sends an update message to application if there are any changes of the relevant sensors.

- Sensors’ properties are defined by the sensors itself. Applications have to obtain this information through SG.
- Application/Sensor flow – The SG provides each application with information of sensors they need according to the filtering criteria. The application then communicates with sensors through the application API for receiving data and sending control messages.

Application API

The Sensor Grid aims at supporting a large amount of applications for users and service providers of different industries (e.g. financial, military, logistics, aerospace etc.). The Sensor Grid provides a common interface which allows any kind of application to retrieve information from the sensor pool managed by SCMW. The API also provides filtering mechanism which provides application with sensors matching their querying criteria only.

Sensor

The definition of sensor is a time-dependent stream of information with a geo-spatial location. A sensor can be a hardware device (e.g. GPS, RFID reader), a composite device (e.g. Robot carrying light, sound and ultrasonic sensor), Web services (e.g. RSS, Web page) or task-oriented Computational Service (e.g. video processing service).

Sensor Client Program

A sensor needs a Sensor Client Program (SCP) to connect to the Sensor Grid. The SCP is the bridge for communication between actual sensors and SCMW. On the sensor side SCP communicates with the sensor through device-specific components such as device drivers. On the Sensor Grid side SCP communicates with the Sensor Grid through the Sensor Service Abstraction Layer.

1.2 Sensor Cloud Middleware

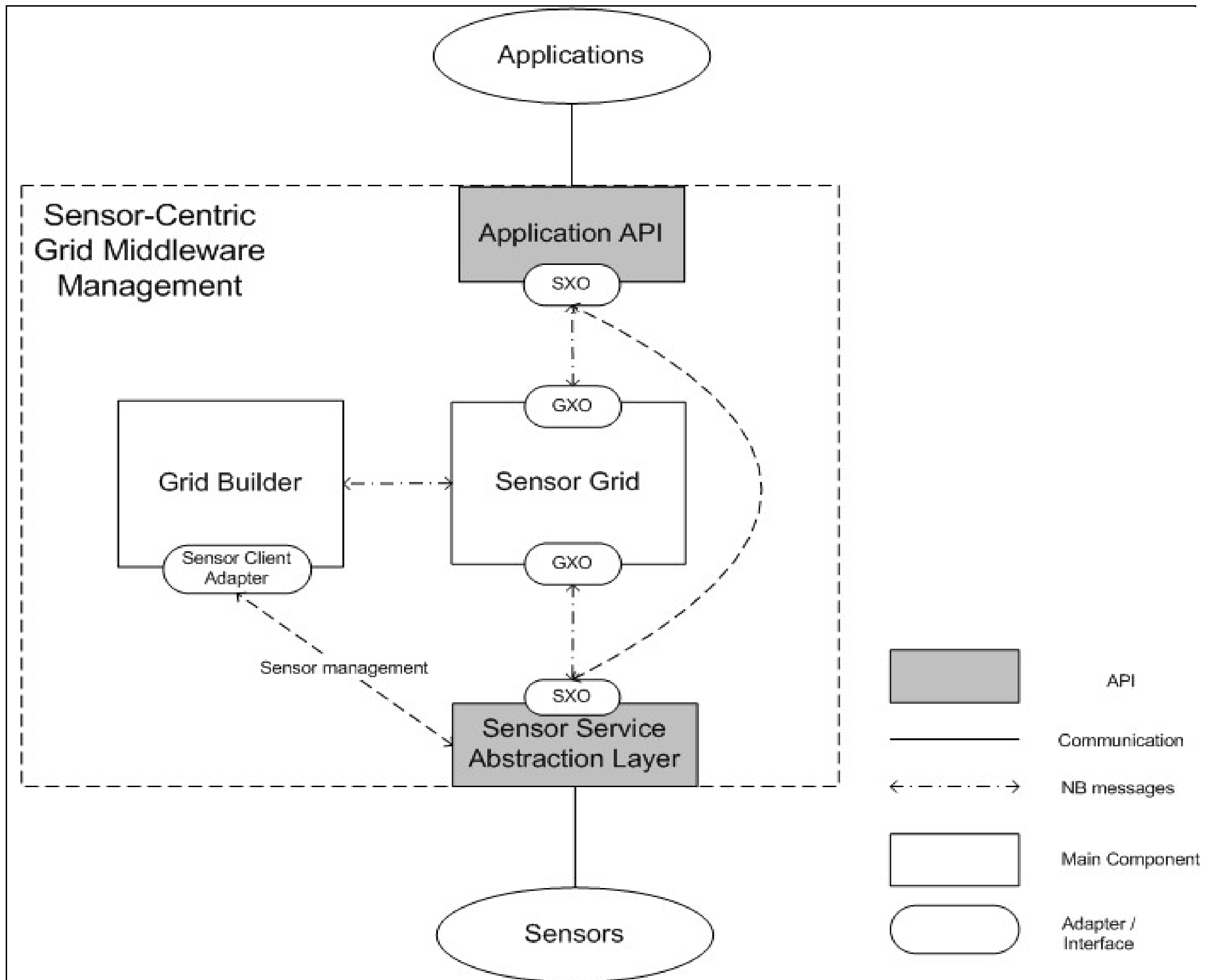


Figure 4 Sensor Cloud Middlware

Sensor-Centric Grid Middleware Management System (SCMW) is carefully designed to provide a seamless, user-friendly, scalable and fault-tolerant environment for the development of different applications which utilize information provided by the sensors. Application developers can obtain properties, characteristics and data from the sensor pool through the **Application API** (see Appendix B for details), while many of the technical difficulties of deploying sensors are abstracted away. At the same time, sensor developers can add new types of sensors and expose their services to application developers through SCMW's **Sensor Service Abstraction Layer (SSAL)** (see section 3.5 for details).

NaradaBrokering (NB) is the transport-level messaging layer for SCMW. It is a distributed message-based transport network based on the pub/sub messaging model.

By using NB as the transport different components of SCMW can be deployed and works collaboratively in a distributed manner.

The overall architecture of SCMW is shown in Figure 4. Internally SCMW is composed of 2 main modules – **Sensor Grid (SG)** and **Grid Builder (GB)** which serves different functions.

1.2.1 Grid Builder (GB)

Given the large amount of sensors, GB is a sensor management module which provides mechanism and services to do the following:

1. Define the properties of sensors
2. Deploy sensors according to defined properties
3. Monitor deployment status of sensors
4. Remote Management - Allow management irrespective of the location of the sensors
5. Distributed Management – Allow management irrespective of the location of the manager / user

GB itself posses the following characteristics:

1. Extensible – the use of Service Oriented Architecture (SOA) to provide extensibility and interoperability
2. Scalable - management architecture should be able to scale as number of managed sensors increases
3. Fault tolerant - failure of transports OR management components should not cause management architecture to fail

The details of GB are discussed in Section 3.3.

1.2.2 Sensor Grid (SG)

SG communicates with a) sensors b) applications c) Grid Builder to mediate the collaboration of the three parties. Primary functions of SG are to manage and broker sensor message flows.

1.2.2.1 Sensor/Sensor Grid flow

SG keeps track of the status of all sensors when they are deployed or disconnected so that all applications using the sensors will be notified for changes. Sensor data normally does not pass through SG except when it intentionally has to be recoded. In this case SG will subscribe to data of that particular sensor.

1.2.2.2 Application/Sensor Grid flow

Applications communicate with SCMW through the Application API, which in turn communicates with SG internally. Applications can define their own filtering criteria, such as location, sensor id, and type to select which sensors they are interested in. These

filters are sent to SG for discovering and linking appropriate sensors logically for that application and forwards messages among the relevant sensors and that application. SG must always check which sensors meet the selected filter criteria and update the list of relevant sensors accordingly. It then sends an update message to application if there are any changes of the relevant sensors.

1.2.2.3 Grid Builder/Sensor Grid flow

Sensors' properties are defined in GB. Applications have to obtain this information through SG. Moreover, filtering requests are periodically sent to GB for updating the lists of sensors needed for each application according to their defined filter parameters. Much of the information will be stored in a SG to minimize queries to Grid Builder.

1.2.2.4 Application/Sensor flow

SG provides each application with information of sensors they need according to the filtering criteria. The application then communicates with sensors through the Application API for receiving data and sending control messages.

The details of SG are discussed in Section 1.4.

1.2.3 SCMW API

The SCMW aims at supporting a large amount of applications for users and service providers of different industries (e.g. financial, military, logistics, aerospace etc.). SCMW provides an API which allows any kind of application to retrieve information from the sensor pool managed by SCMW. The API also provides filtering mechanism which provides application with sensors matching their querying criteria only.

Details of the SCMW API are discussed in Section 1.5.1

1.2.4 Sensor

The definition of sensor is any time-dependent stream of information with a geo-spatial location. A sensor can be a hardware device (e.g. GPS, RFID reader), a composite device (e.g. Robot carrying light, sound and ultrasonic sensor), Web services (e.g. RSS, Web page) or task-oriented Computational Service (e.g. video processing service).

1.2.4.1 Sensor Client Program

A sensor needs a **Sensor Client Program (SCP)** to connect to SCMW. The SCP is the bridge for communication between actual sensors and SCMW. On the sensor side SCP communicates with the sensor through device-specific components such as device drivers. On the SCMW side SCP communicates with SCMW through Sensor Service Abstraction Layer (refer to section 3.5 for details).

Figure 5 shows a physical sensor and the corresponding Sensor Client Program.

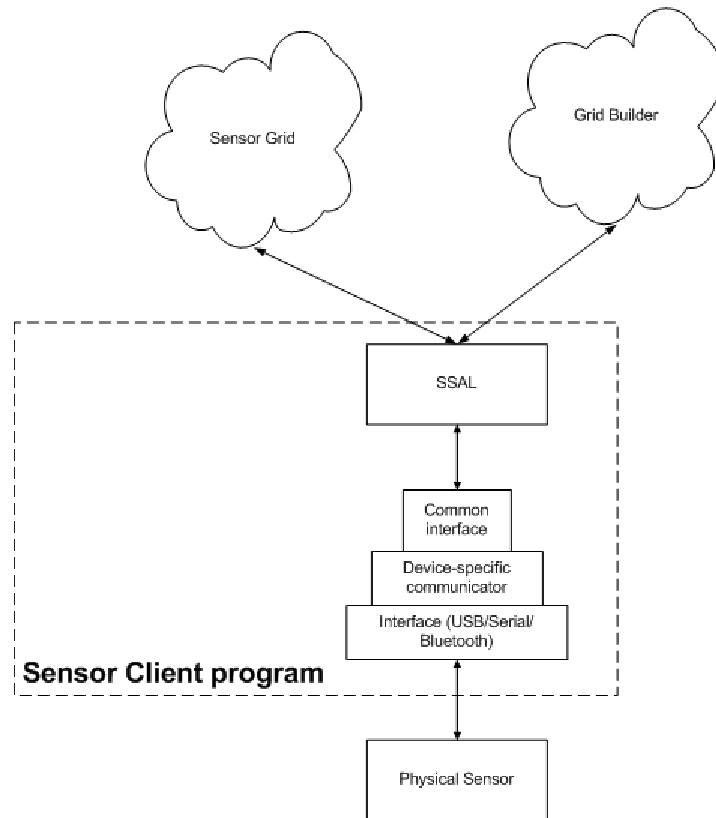


Figure 5 Structure of a Sensor Client Program

1.2.4.2 Computational Service

Computational Service is a special kind of sensor which does not take input from the environment. Instead, they take output of other sensors as their input, perform various computations on the data, and output the processed data finally. Since a Computational Service also produces a time-dependent stream of data it matches our definition of a sensor.

Figure 6 shows the data flow of how environmental data is transformed by processing data through a sensor and a Computational Service. The architecture of SCMW allows the data source to be assigned and reassigned dynamically.

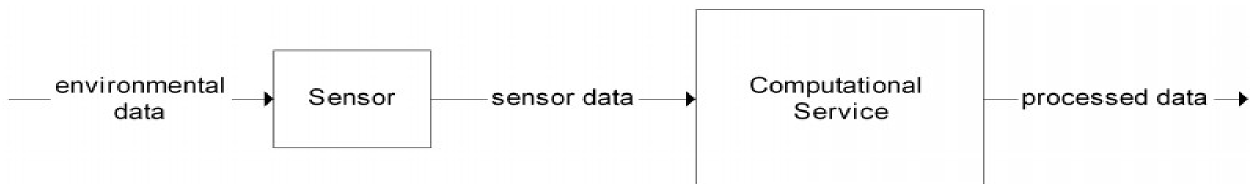


Figure 6 Computational Service

1.2.5 Sensor Service Abstraction Layer (SSAL)

SCMW can potentially support large amount of sensors of different kind. Ease of adding new sensors by different sensor developers without internal knowledge of SCMW is one of the most important requirements. SSAL provides a common interface for adding new sensors to the system easily. Sensor developers have to write simple programs utilizing SSAL libraries

for connecting sensors to SCMW. Afterwards the sensor will be available for all applications right away.

Details of the SSAL are discussed in Section 1.5.2

1.3 Grid Builder

1.3.1 An Overview of the Grid Builder Architecture

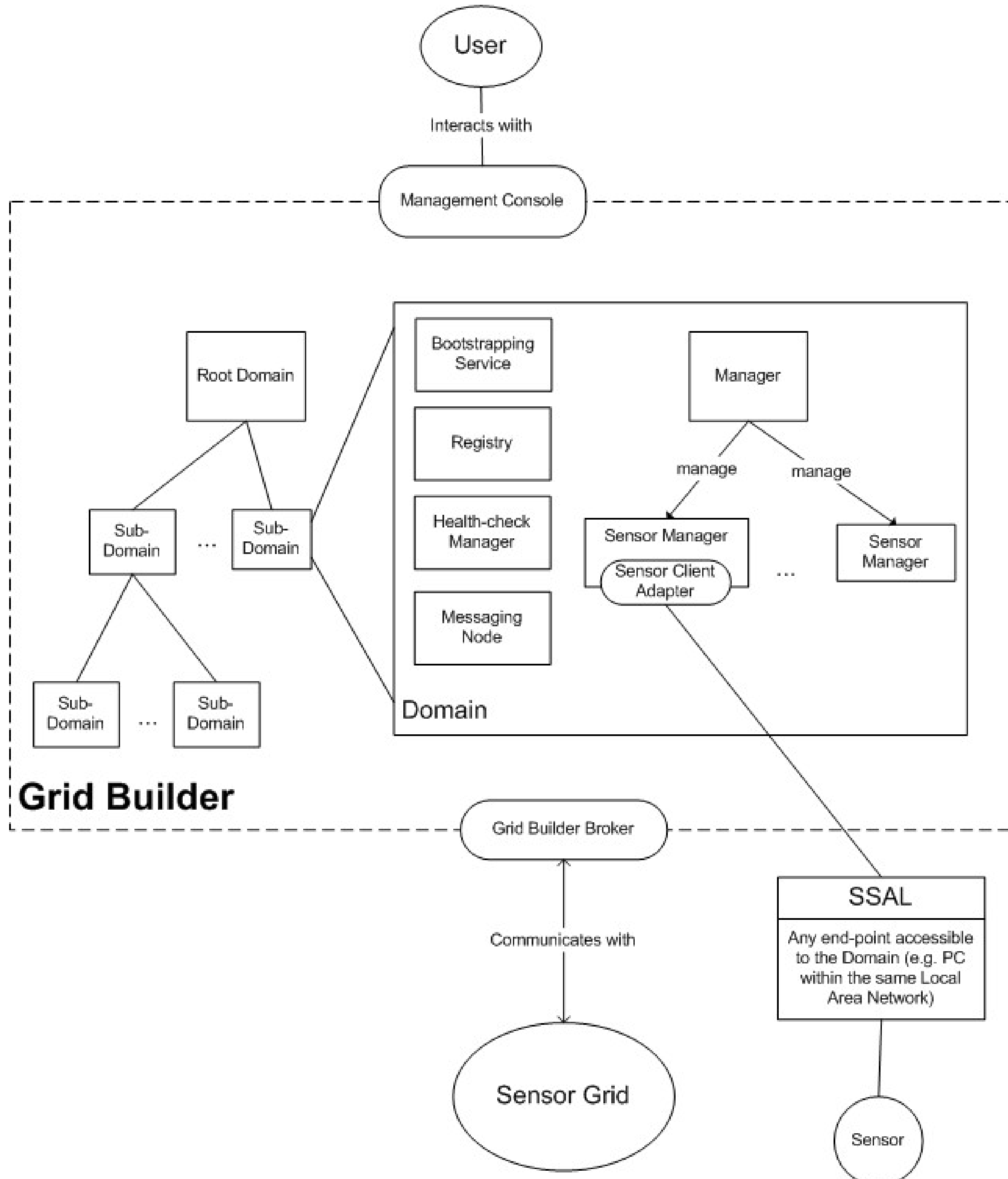


Figure 7 An overview of the Grid Builder architecture

Figure 7 depicts the overall Grid Builder (GB) architecture. GB is originally designed for managing Grid-of-Grids. For this project, GB is extended to include the management of a generalized sensor-centric grid of grids. Description of GB will focus on this specialized version. CGL-developed hpsearch is adopted and extended for this work [2].

The Grid which GB manages is arranged hierarchically into **Domains**. Each domain is typically, but not necessarily, a single PC which manages sensors which are closely related. Sensors can be deployed from any PC which is accessible from one of the domains. There can be only one root node in the grid known as the **Root Domain**. Each domain is started by its **Bootstrapping.**

Within each domain, there exist some basic components:

Managers and Resources

GB manages grids and resources through a manager-resource model. Each type of Resource which does not have a Web Service interface should be wrapped by a **Service Adapter (SA)**. Each kind of SA is managed by a corresponding **Manager**.

Since our grid contains sensors, a **Sensor Manager** is responsible for managing sensors through **Sensor Service Adapters (SSA).** Each SSA has its own set of defined **Sensor Policy**. This policy tells Sensor Manager how the SSA is to be managed, and defines the **properties** of the sensor bound to the SSA.

The **Health-check Manager** is responsible for checking the health of the whole system (ensures that the registry and messaging nodes are up and running and that there are enough managers for resources).

Bootstrapping Service

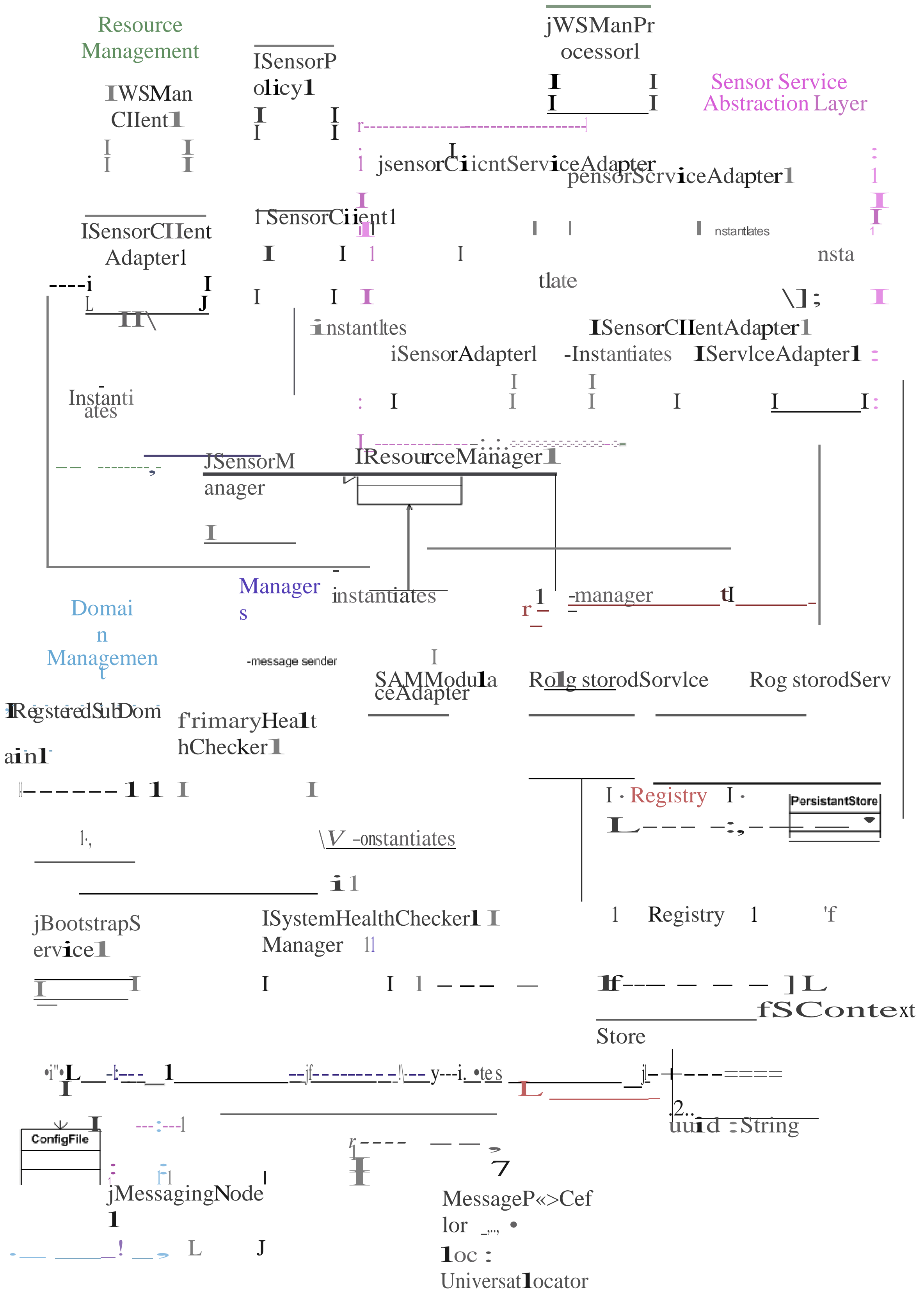
This service ensures that bootstrap processes of the current domain are always up and running. For example, it periodically spawns a health-check manager that checks the health of the system.

Registry

All data about registered services and service adapters are stored in memory called **Registry.** Registry is used to process messages so it can manage new SA, renew SA and update SA status.

1.3.2 Significant Classes

1.3.2.1 Class Diagram



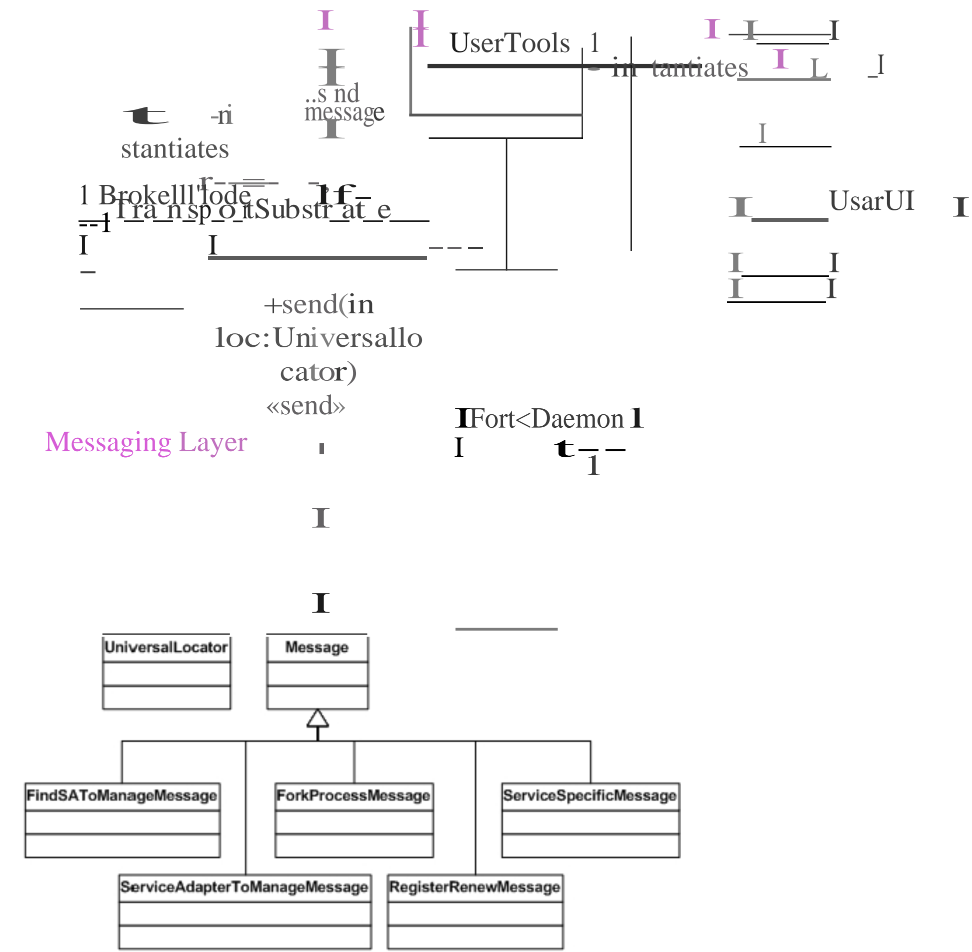


Figure 8 Class Diagram of Grid Builder

The diagram shows the class diagram of significant classes in GB. They are categorized into 5 main categories:

Messaging Layer

GB is built on top of a message-based architecture. All modules in GB such as BootstrapService, ForkDaemon, Managers, Registry and ServiceAdapters are standalone and communicate with one another by message passing. With this model, separate modules can be deployed as distributed services.

GB has a set of classes dedicated for message passing. Each module has a unique UUID and one or more UniversalLocator(s) (UL). UL provides all the information necessary to identify a module in the network, including transport type, host address, port and path. 4 transport types are supported: UDP, TCP, HTTP and NB. Each UL is responsible for message of one transport type.

TransportSubstrate is responsible for sending and receiving messages to and from a module. It automatically serializes the message content according to the transport type of destination. Once created, it spawns a thread which keeps waiting for incoming messages and notifies the associated MessageProcessor upon message arrival.

Modules which want to receive message should implement the MessageProcessor interface and associates itself with a TransportSubstrate. Important modules which implement this interface include BootstrapService, Registry, SystemHealthChecker, Manager, ServiceAdapter and UserTools.

Communications between SensorManager and SensorServiceAdapters use the Web Service (WS) interface. WS in GB is built on top of this messaging layer.

Domain Management

Domain management in GB is done by BootstrapService. Each domain has one BootstrapService which constantly communicates with the BootstrapServices of other domains. Each domain hierarchy contains one Root node. Each domain connects with at most one parent node and any number of child nodes. For now the hierarchy is defined using a configuration file (mgmtSystem.conf).

To keep the whole hierarchy up and running, each domain periodically sends a heart beat message to its parent domain. It also has to spawn the BootstrapService of all child domains if any of them is not sending heart beat for some time.

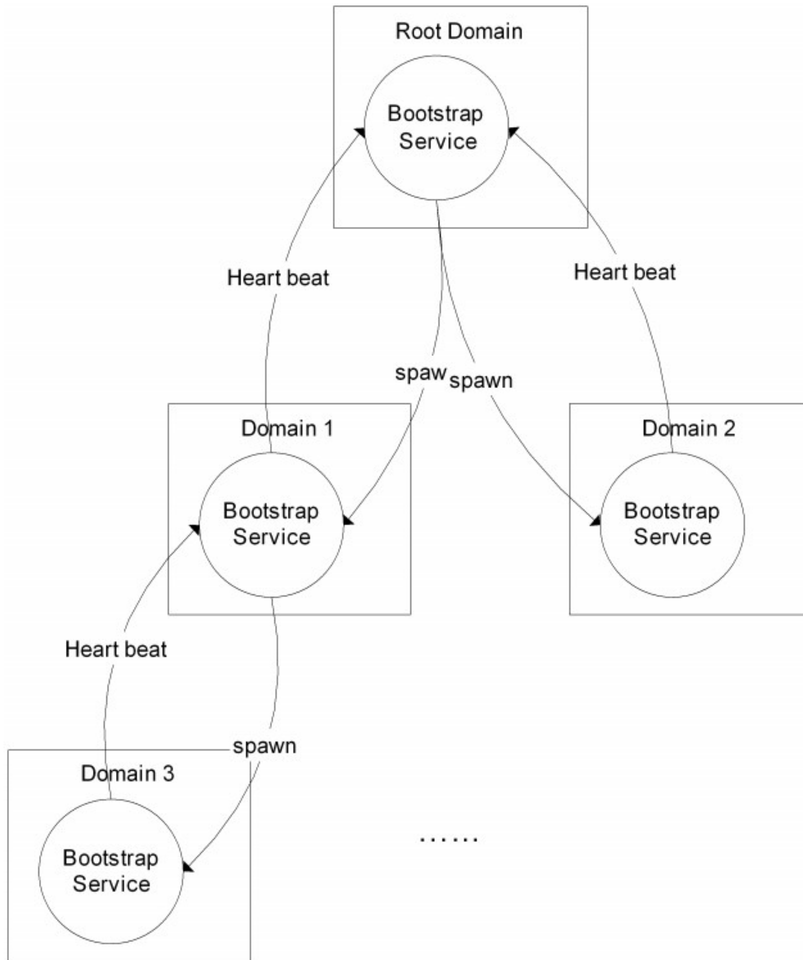


Figure 9 Domain Management

Managers

In GB there are two levels of managers. The lowest level is ResourceManager, which manages resource specific modules. For example, SensorManager is responsible for managing a SensorServiceAdapter through the Web Service interface and performs operation such as sending policies to the adapters.

The upper level is Manager, which manages ResourceManagers and ServiceAdapters. The Registry keeps checking whether there are ServiceAdapters which have been registered but do not have a Manager during the health check sequence. If there is one, the Manager is notified and create a SAMModule in turn creates a ResourceManager for the particular resource in the ServiceAdapter. SensorClientAdapter is an adapter inside SensorManager for communication with the associated SensorServiceAdapter inside the Service Adapter.

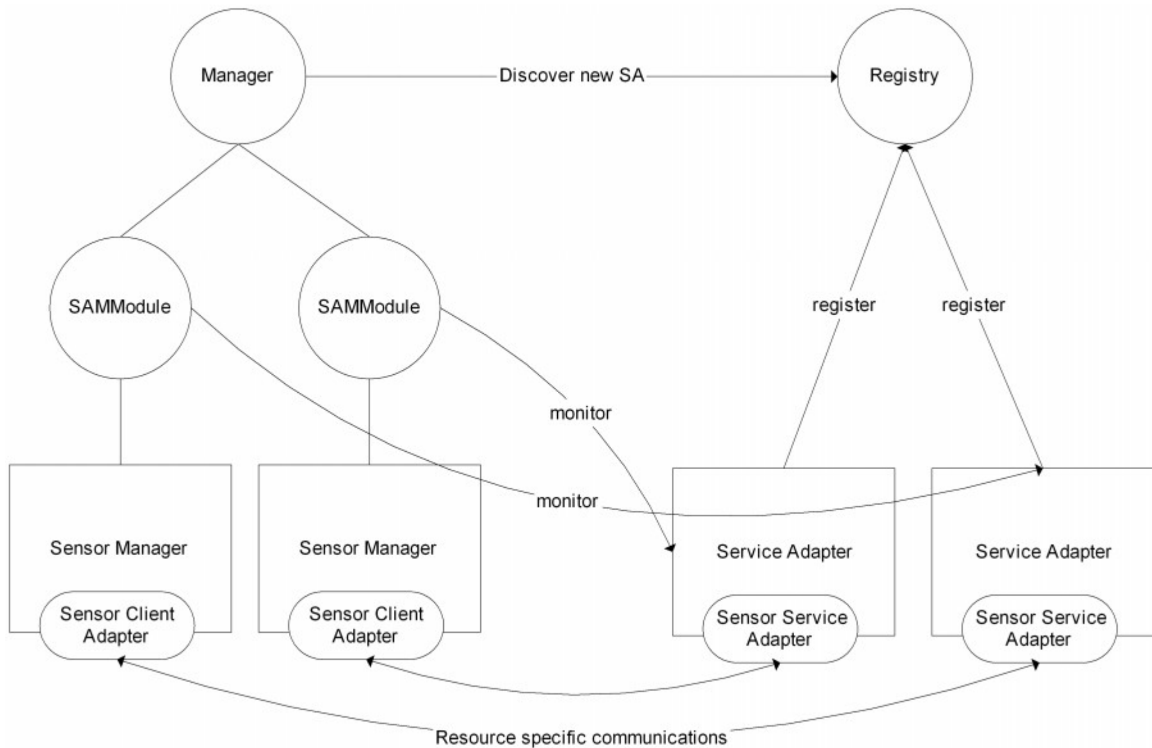


Figure 10 Manager and Service Adapters

Resource Management

These classes are at the resource level, where resource specific tasks are performed. Each sensor is treated as a resource in GB, and each sensor has a corresponding client program (represented by SensorClient) responsible for interfacing the sensor with SCMW.

Sensor Service Abstraction Layer (SSAL) is the interface for connecting all types of sensor client programs with GB. The class diagram only shows part of SSAL which resides in GB. The whole SSAL involves classes of SXO as well.

Communication between resource managers (i.e. SensorManager) and Resources (i.e. SensorServiceAdapter (SSA)) uses the Web Service (WS) interface for message passing. SSA therefore conforms to the WS "Put", "Get", "Delete" and "Create". "Get" is used for getting SensorPolicy of the sensor and initiates connection with SG. "Delete" is used for disconnecting connection with SG.

Registry

Each domain has a Registry which maintains the state of the entire domain, such as the Universal Locator of every module, how many Service Adapters have been registered, the status and policy of each sensor, which SA is assigned to which Manager etc.

RegisteredServiceAdapter is a class which contains information of ServiceAdapter such as UniversalLocator, SensorPolicy and current status. RegisteredService contains information of non-SA modules such as Managers and MessagingNodes.

Registry can work with or without persistent storage. By default all information is stored in memory using hash tables. The user has an option whether to write all information to persistent storage so that it can be retrieved later on even if the domain is restarted. The persistent storage used is compliant to WS-Context specification [3].

Figure 11 shows the overall architecture of the Domains, Registry and WS-Context modules in Grid Builder. To use WS-Context, an AXIS server and a MySQL server should be running in each domain for WS communication and storage. All domain related information in the Registry is stored in WS-Context and shared with other domains through NaradaBrokering's topic-based publish-subscribe messaging service.

Although the current implementation does not use WS-Context as a centralized database for service discovery, it can be easily enhanced to provide such service since the system is already WS compliant.

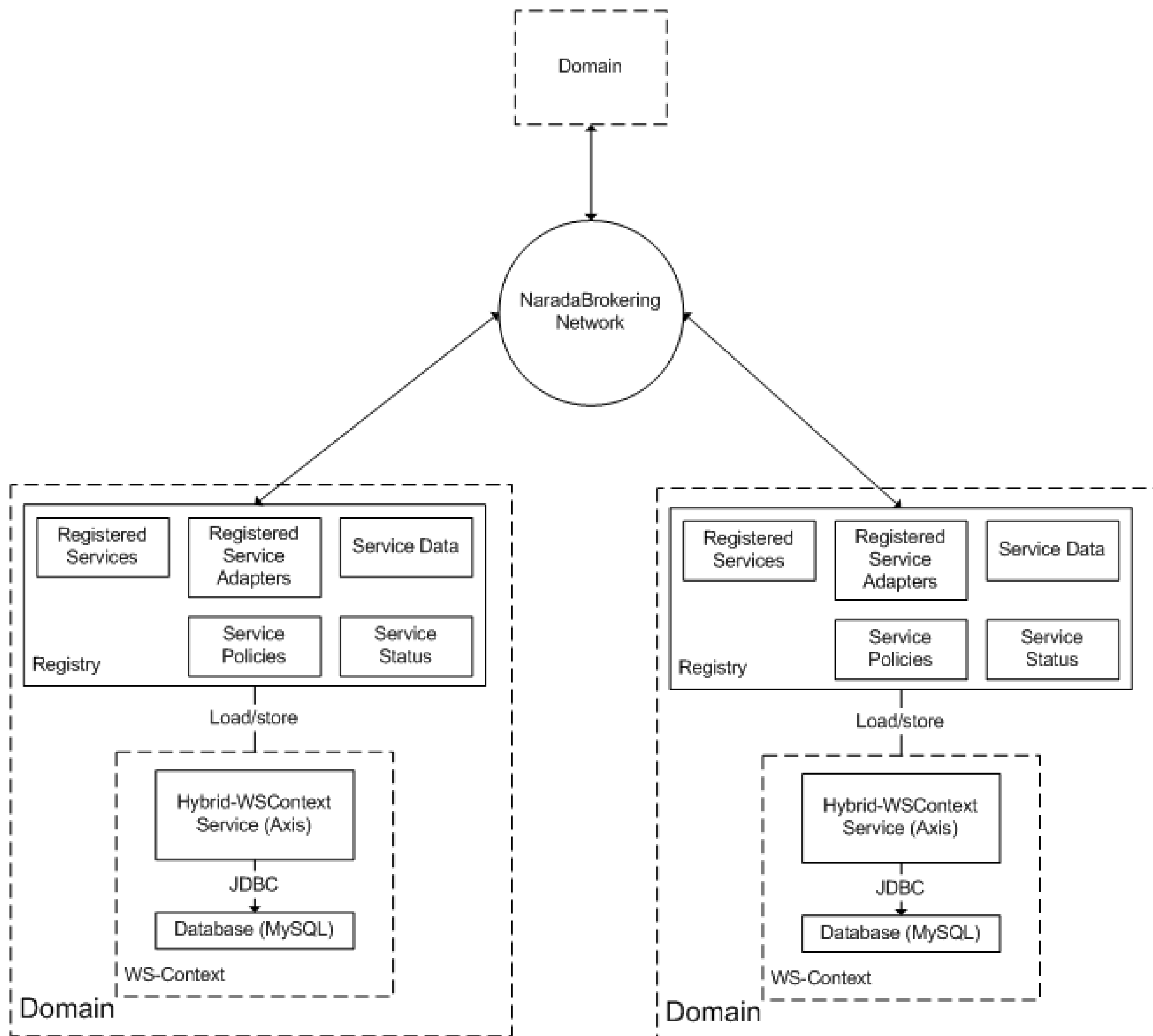


Figure 11 Registry and WS-Context

1.3.2.2 Class Description

This section provides brief description of each important class in GB.

Class name:	MessageProcessor
Package name:	cgl.hpsearch.core.transport
Description:	Interface for classes which use GB's messaging layer to receive messages
Important interface:	processMessage()

Class name:	MessagingNode
Package name:	cgl.hpsearch.core.services.messagingNode
Description:	Manages the GB's transport layer components (such as NB)
Important interface:	setBootstrapLocator(), startBrokerNode()

Class name:	TransportSubstrate
Package name:	cgl.hpsearch.core.transport
Description:	Responsible for receiving and sending messages to and from MessageProcessor using different transport protocols
Important interface:	register(), send(), getUniversalLocatorForTransport(), close()

Class name:	Message
Package name:	cgl.hpsearch.core.messages
Description:	Superclass of all types of messages in GB. Different types of message has different characteristics and serves different functions
Important interface:	getType(), getMessageId(), getTo(), getFrom(), getTimeStamp()

Class name:	UniversalLocator
Package name:	cgl.hpsearch.core.transport
Description:	A locator which lets different modules to identify one another for messaging passing. Records the host, port, and transport type of a module
Important interface:	getHost(), getPort(), getPath(), getTransportType()

Class name:	UserTools
Package name:	cgl.hpsearch.core.services.user
Description:	Responsible for forwarding different user operations (e.g. deploy sensors) to different modules in GB
Important interface:	getServiceData(), putServiceData(), retrieveStatus(), sendPolicyMessage(), sendRunMessage(), sendFilterMessage(), sendForkMessage()

Class name:	UserUI
Package name:	cgl.hpsearch.NaradaBrokering.usergui
Description:	Graphical user interface of GB's management console
Class name:	Manager
Package name:	cgl.hpsearch.core.services.manager
Description:	Manages all Resource Managers
Important interface:	processMessage(), startSAMManagementThread(), removeSAMManagementObject(), send()
Class name:	SystemHealthChecker
Package name:	cgl.hpsearch.core.services.manager
Description:	Responsible for checking whether all modules are up and running in a domain
Important interface:	processMessage()
Class name:	BootstrapService
Package name:	cgl.hpsearch.core.services.bootstrap
Description:	Responible for starting up all modules during domain initialization. Periodically spawns SystemHealthChecker and sending heart beat to parent domain
Class name:	ForkDaemon
Package name:	cgl.hpsearch.core.services.fork
Description:	Responsible for creating different modules locally as processes
Important interface:	process()
Class name:	SAMModule
Package name:	cgl.hpsearch.core.services.manager
Description:	Manages resources (sensors). Has one to one mapping to each Service Adapter and the corresponding Resource Manager.
Important interface:	send(), checkIfOwner(), getServiceData(), putServiceData(), spawnProcess(), sendMessage()
Class name:	SensorManager
Package name:	cgl.hpsearch.sensor
Description:	Resource manager for managing SensorServiceAdapter
Important interface:	processMessage(), getServicePolicy(), putServicePolicy(), runService()
Class name:	SensorClientAdapter
Package name:	cgl.hpsearch.sensor
Description:	The adapter of SensorManager for communication with SensorServiceAdapters using Web Service
Important	getServicePolicy, putServicePolicy(), runService()

Class name:	ServiceAdapter
Package name:	cgl.hpsearch.core.services.sa
Description:	Associated with a Resource Manager to manage the corresponding resource
Important interface:	start(), close(), publishData()

Class name:	SensorServiceAdapter
Package name:	cgl.hpsearch.sensor
Description:	Responsible for brokering the communication between a Resource Manager and sensor client program using Web Service
Important interface:	start(), close(), publishData(), handleSensorGridConnectionLoss(), setSensorProp(), processWxMGMT_Rename(), processWxfDelete(), processWxfPut(), processWxfCreate(), processWxfGet()

Class name:	SensorClientServiceAdapter
Package name:	cgl.hpsearch.sensor
Description:	Responsible for brokering the communication between a Resource Manager and service sensor client program using Web Service
Important interface:	start(), close(), publishData(), handleSensorGridConnectionLoss(), setSensorProp(), sendControl(), setFilter(), subscribeSensorData(), unsubscribeSensorData(), processWxMGMT_Rename(), processWxfDelete(), processWxfPut(), processWxfCreate(), processWxfGet()

Class name:	SensorPolicy
Package name:	cgl.hpsearch.core.policies
Description:	Holds resouce specific policy, that is the property of a sensor
Important interface:	getType(), getSensorProperty()

Class name:	WSManClient
Package name:	cgl.hpsearch.wsmgmt
Description:	Client interface for communicating with WSManProcessors (end points) using Web Service messaging
Important interface:	getMyEndPoint(), getServiceEndPoint(), setServiceEndPoint(), setWsEventingClient(), processMessage(), executeOneWay(), executeRequestReply(), sendOut(), CreateAndMarshallMessage()

Class name:	WSManProcessor
Package name:	cgl.hpsearch.wsmgmt
Description:	End point for receiving Web Service Message
Important interface:	setMessageSender(), setMyEndPoint(), processSOAPMessage(), processWxMGMT_Rename(), processWxfDelete(), processWxfPut(), processWxfCreate(), processWxfGet()

1.3.3 Important Features

1.3.3.1 System Health Check

Every module in GB are deployed in a distributed manager and linked together by different network protocols. A health check system is therefore fundamental to ensure every modules are indeed deployed and working properly. GB performs periodic **System Health Check (SHC)** to ensure that every thing is up and running.

SHC can be divided into three stages:

Initialization

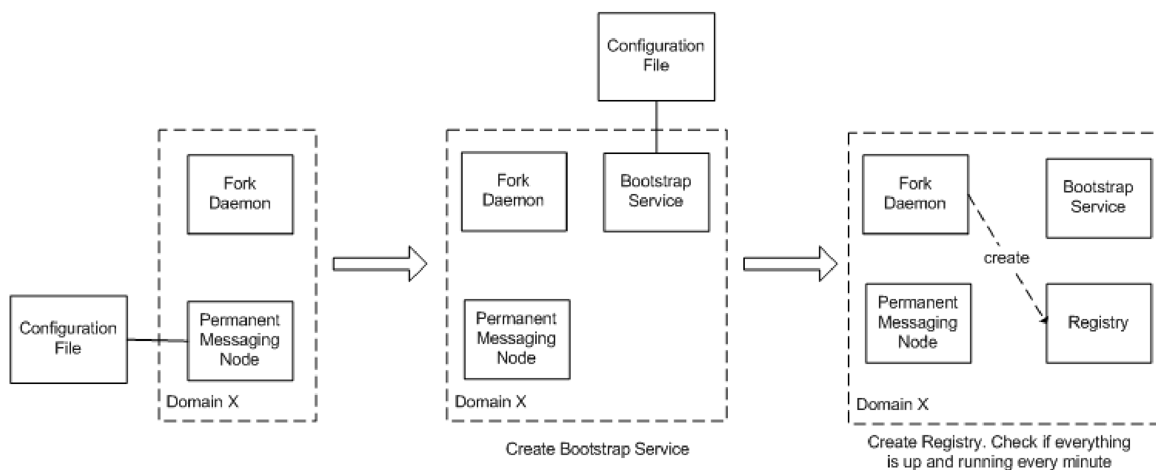


Figure 12 System Health Check (SHC) Initialization

To start a new Domain X, a user has to execute a script to perform a Primary Health Check Sequence. This action creates a Permanent Messaging Node, which is responsible for communication between all modules within a domain, and communication with other domains. After that, a Fork Daemon is created. Every module of Grid Builder (e.g. Registry, Service Adapters, Sensor Service Adapters etc.) is executed as a separate process in the operating platform. Fork Daemon is responsible for creating modules as separate processes.

After primary health check, the domain is now capable of receiving messages from other domains. The Bootstrap Service is launched when a message is received from the root domain. The Bootstrap Service is responsible for making sure that every module is up and running in a domain. It periodically spawns a System Health Checker to check the health of the system.

After Bootstrap Service has been initialized, it creates the Registry. The system then checks if all modules are up and running for every minute. If not, create the module that is missing (for details please refer to section 3.3.4.3).

Detect Changes

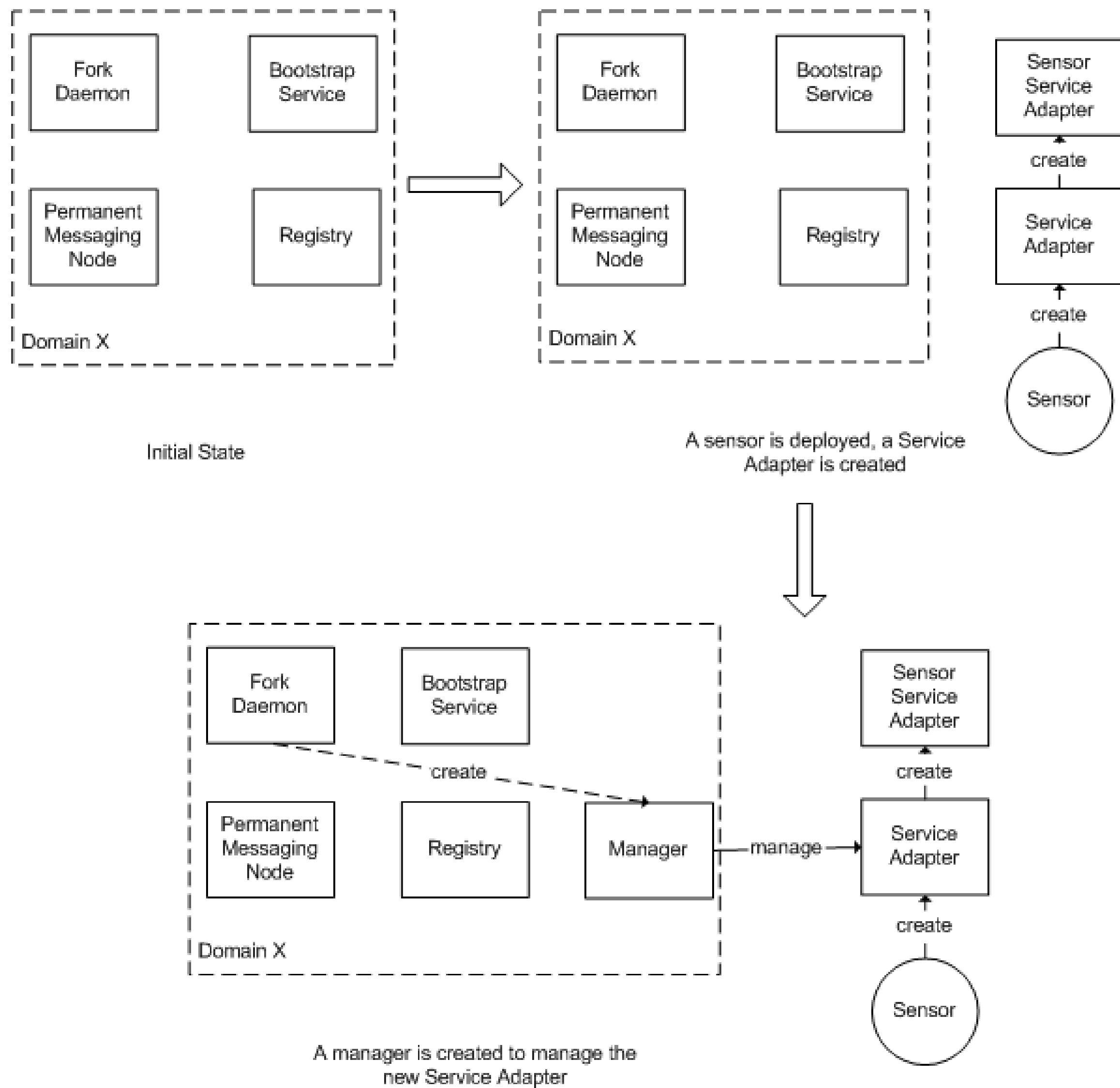


Figure 13 Adding Service Adapter

When we introduce changes to the system, such as deploying a sensor, SHC automatically detects and reacts to the change. For example, a user deploys a sensor by starting the corresponding sensor client program. The program automatically creates a new Service Adapter for the sensor which in turn creates a Sensor Service Adapter. If no Manager is present in the domain, a Manager process is created by ForkDaemon to manage the sensor through Service Adapter.

Maintain System State

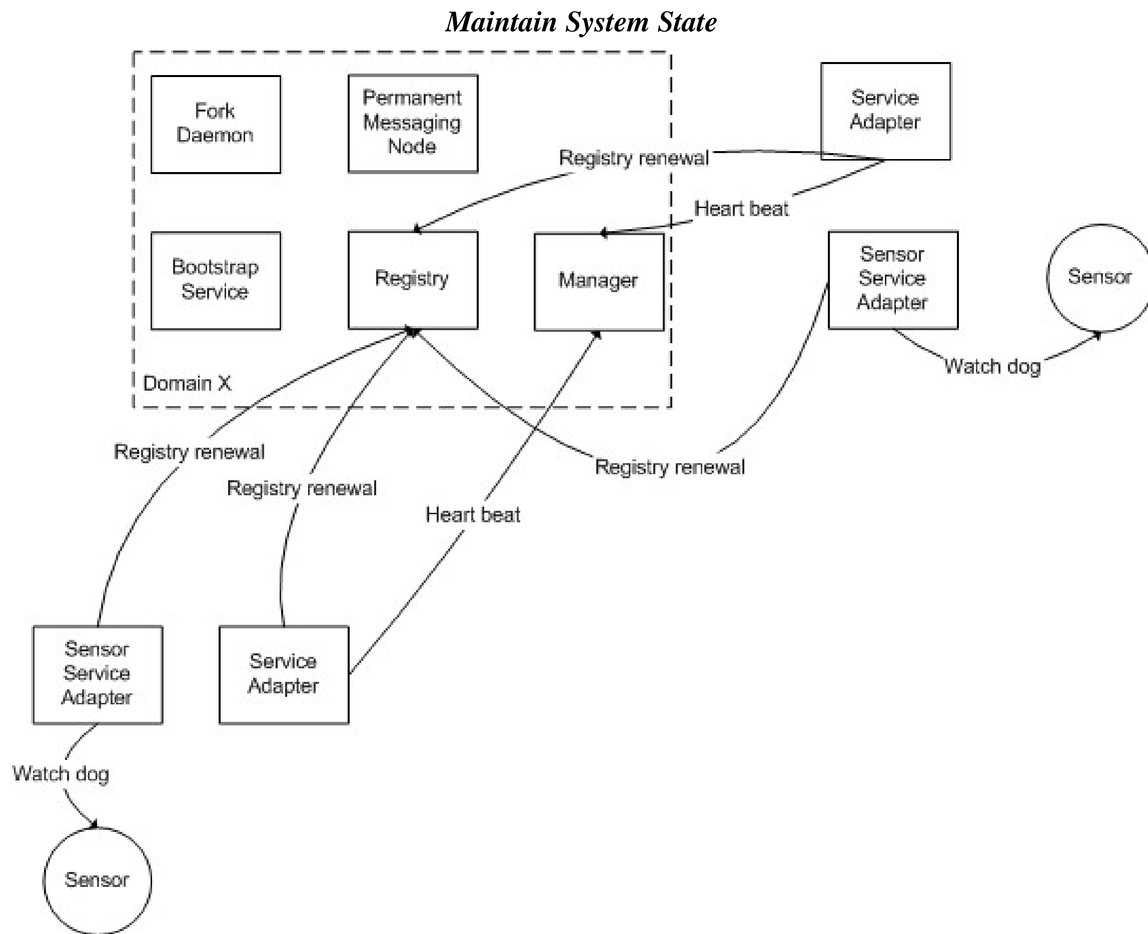


Figure 14 System Health Check (SHC) Maintaining System State

To make sure that every resource is up and running, each module periodically notifies its manager and the registry of its presence.

1.3.3.2 Classification Scheme

Classification defines all properties which are shared by all sensors supported by SCMW. Classification serves the following functions:

1. Allows GB to differentiate among different sensors for visualizing sensor's policies
2. Defines what can be filtered
3. Allows meaningful visualization of sensor data at application side
4. Allows application to differentiate different sensors

Figure 8 shows the class diagram of classification. It can be divided into 3 categories:

Sensor Property

In order to introduce a new sensor to SCMW, the following properties have to be defined in class SensorProperty:

Table 3-1 Fields of Sensor Property

Property	Description

sensorId	A Human readable ID for identification which does not have to be unique
groupId	Sensors can be assigned to different logical groups for easier management. GroupId identifies the group
sensorType	Textual description of the type of a sensor
sensorTypeId	An integer which helps identifying the sensor type. Application has to compare this together with field sensorType to uniquely identify the type of a sensor
location	Textual description of the location of a sensor, including street, city, state/province and country
historical	Defines whether to archive collected sensor data in SG. Currently this feature is not implemented
sensorControl	An array of integers which uniquely identifies each control message
controlDescription	A string array of textual description of control messages. Should align with sensorControl array
userDefinedProperty	A class which defines any user-defined properties specific for each type of sensor

SCMW comes with a set of predefined types. Class PredefineType contains information for generating predefined SensorProperty. UserDefinedProperty contains properties which are essential for the sensor but may not be common for all sensors (e.g. for deploying a RFID reader it needs the COM port for hardware interfacing). A set of user-defined properties for predefined sensors are implemented as subclasses of UserDefinedProperty.

For location, class PredefinedLocation contains a list of predefined mapping of city names and GPS latitude-longitude for easy visualization on a map.

Sensor Data

For each type of sensor, its data format is usually quite different from other sensors. In SCMW a class which extends SensorData should be created which defines how to decode and use data from a sensor.

Message Serialization

Each time before the property of a sensor is sent among modules (e.g. passing from GPSManager to SensorServiceAdapter and Registry), it is serialized into xml format. Class SensorClassificationUtil provides operation for message serialization and deserialization.

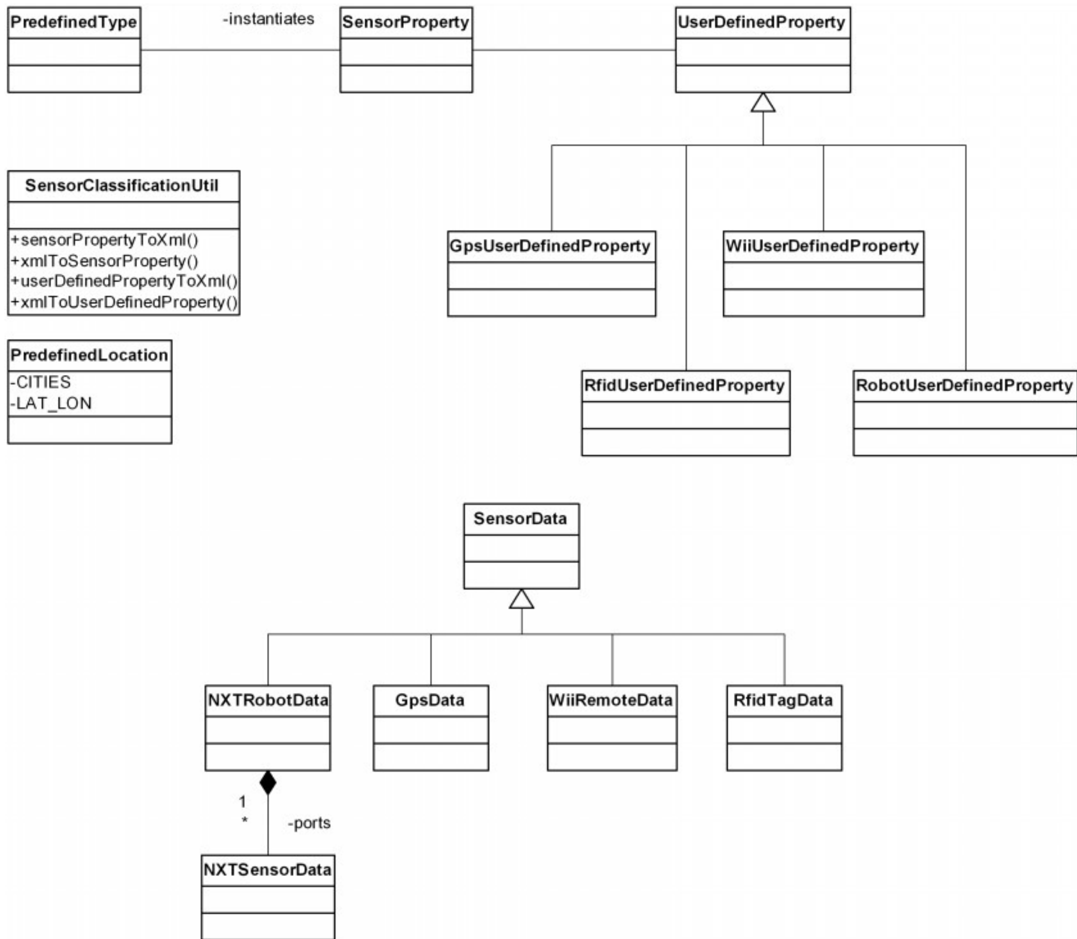
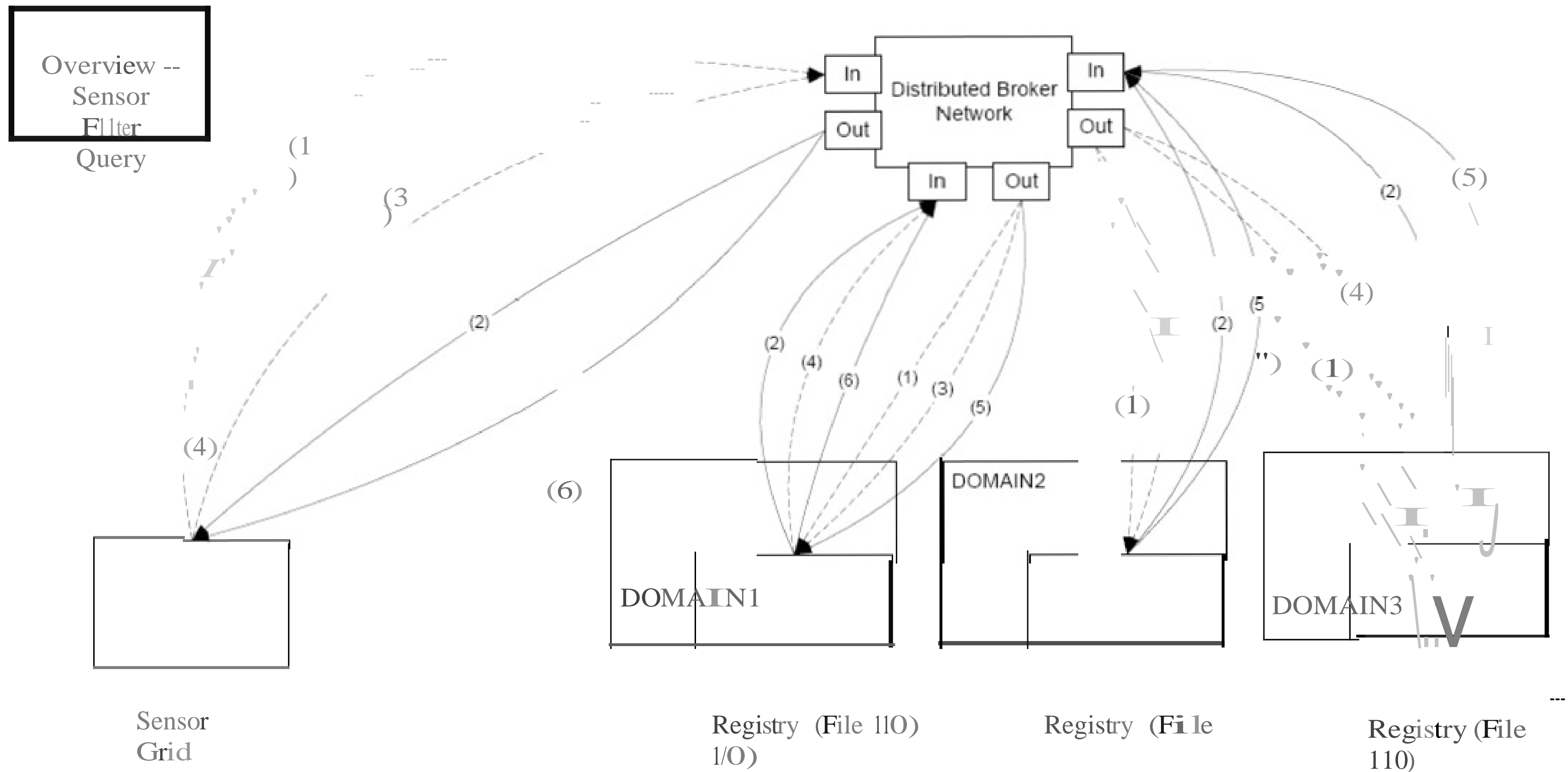


Figure 15 Class diagram of classification scheme in SCMW

1.3.3.3 Filtering Mechanism



Step 1. Sensor Grid looks for a registry for filler query.Publish a request message on a common topic or all registries. the registry which replies the eMiest will be chosen.
(1) GET_GRID_BUILDER_SERVICE_POLICY
(2) GET_GRID_BUILDER_SERVICE_POLICY_RESPONSE

Step 2. Sensor Grid sends a Sensor Filter Query to the chosen registry (say registry in Domain1) (3) FILTER_SERVICE_POLICY_QUERY (/oca/=false) (Note:localmeans localsearch only]

Step 3. Reg1stry publishes a filter query to all other registries
(4) FILTER_SERVICE_POLICY_QUERY (loca/=tnue, with timestamp)
[Note: timestamp is a unique Identifier for the query]

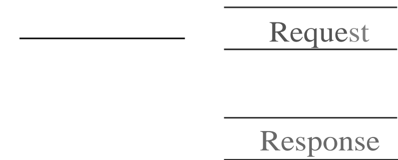
Step 4. Reg1stry receives a filter query with local=true. it will response with the result

immediately through the umque topic *of* the request registry.
(5) FILTER_SERVICE_POLICY_QUERY_RESPONSE (d ca/=true, with timestamp)

step 5. Regrstry

aggregates the responses (identified by timestamp) and send back to Sensor Grid
(6) FILTER_SERVICE_POLICY_QUERY_RESPONSE (local=false)

Figure 3-14 SCGMMS sensor filtering mechanism in a distributed architecture



At the application standpoint filtering is essential for retrieving only the required sensors from a possibly huge sensor pool. Filtering is done based on the SensorProperty of each sensor, which is defined according to based on rules in classification.

Defining a Filter

Applications have to define filtering criteria according to their UDOP requirements. The criteria are encapsulated in a SensorFilter object. A SensorFilter is composed of a set of properties defined in SensorProperty connected with Boolean "and" or "or" operators. Please refer to section 3.3.3.2 for the definition of SensorProperty. Given that a list of sensor properties in a sensor filter are connected together with the "and" operator, only sensors which have properties with exact match in string comparison with ALL the properties defined in the filter should get through. Similarly sensors which have
properties with exact match in string comparison with ANY of the properties defined in a sensor filter with sensor properties connected together with the "or" operator should get through.

The list of "and" and "or" sensor properties are represented as a 2D string array in SensorFilter. For example, if someone wants to get a list of SAID which have policy ((sensorType=GPS and location="Hong Kong") or (sensorType=RFID and location="New York" and historical=true)), set the filter like this:

```
SensorFilter filter=new SensorFilter(); String[][]
comp=new String[2][]; comp[0]=new String[2];
comp[1]=new String[3];
comp[0][0]="sensorType=GPS";
comp[0][1]="location=Hong Kong";
comp[1][0]="sensorType=RFID";
comp[1][1]="location=New York";
comp[1][2]="historical=true";
filter.setOrComparison(comp);
```

Data Flow

Filtering is done in three stages:

Application to SG

A filter query request is initiated from the application. For each filter query, fields which exist in SensorProperty can be combined using the "and" or "or" operator to form a query string. This string is then sent to SG.

SG to GB

SG forwards the request to GB. At this stage, GB searches through the registry of all domains and aggregates the unique id of sensors which match the query in a response message. The response message is then sent back to SG. SG periodically checks if the filter request from application changes. If it does, the application is notified in the same manner.

SG to **application**

SG releases the resources (e.g. unsubscribe sensor's NB topic) used by sensors which are no longer in the list, and initiates resources for new sensors. Then SG notifies the client for all changes made.

1.3.4 Detailed Description

In this section, message flow of various operation of SG will be discussed at Class level using UML collaboration diagrams.

1.3.4.1 Starting a Domain

The following diagram shows the events happening when a domain is started.

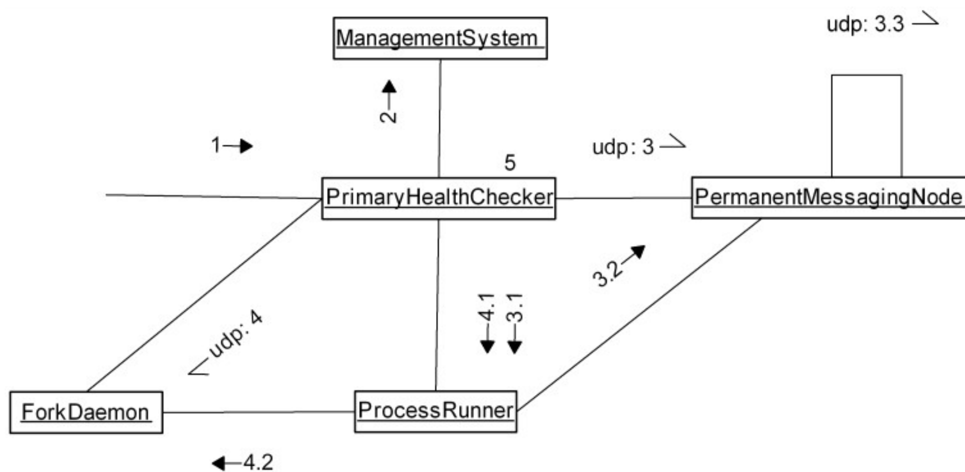


Figure 3-15 Event flow when starting a sensor grid domain

1. A user starts the domain by executing "runPrimaryHealthCheck.bat"
2. ManagementSystem.BootStrap() is called to initialize all system properties, environment variables and various user-defined properties from configuration files
3. Send a PingRequestMessage to the expected locator(s) of messaging node(s) registered in configuration files. If any messaging node does not respond with PingResponseMessage within 5 seconds, go to 3.1. Otherwise go to 4
   - 3.1. For each messaging node not responding, send a request to ProcessRunner to start a PermanentMessagingNode process
   - 3.2. ProcessRunner starts the messaging node process
   - 3.3. Spawns a thread which continuously monitors the presence of itself by using udp messages (ping request and response). Starts a BrokerNode (NB) according the configuration provided by configuration file (defaultMessagingNode.conf)
4. Send a PingRequestMessage to the expected locator(s) of ForkDaemon(s) registered in configuration files. If any ForkDaemon does not respond with PingResponseMessage within 5 seconds, go to 4.1. Otherwise go to 5
   - 4.1. For each ForkDaemon not responding, send a request to ProcessRunner to start a ForkDaemon process
   - 4.2. ProcessRunner starts the ForkDaemon process
5. PrimaryHealthChecker sleeps for 10 seconds to allow any pending processes to instantiate. Then it checks whether all messaging nodes and ForkDaemons are up and running. If yes, it sleeps for 30 seconds. Afterwards, it goes to step 3 and checks everything again

1.3.4.2 Starting BootstrapService of a Domain

When a domain is start, it undergoes the following Bootstrap sequence.

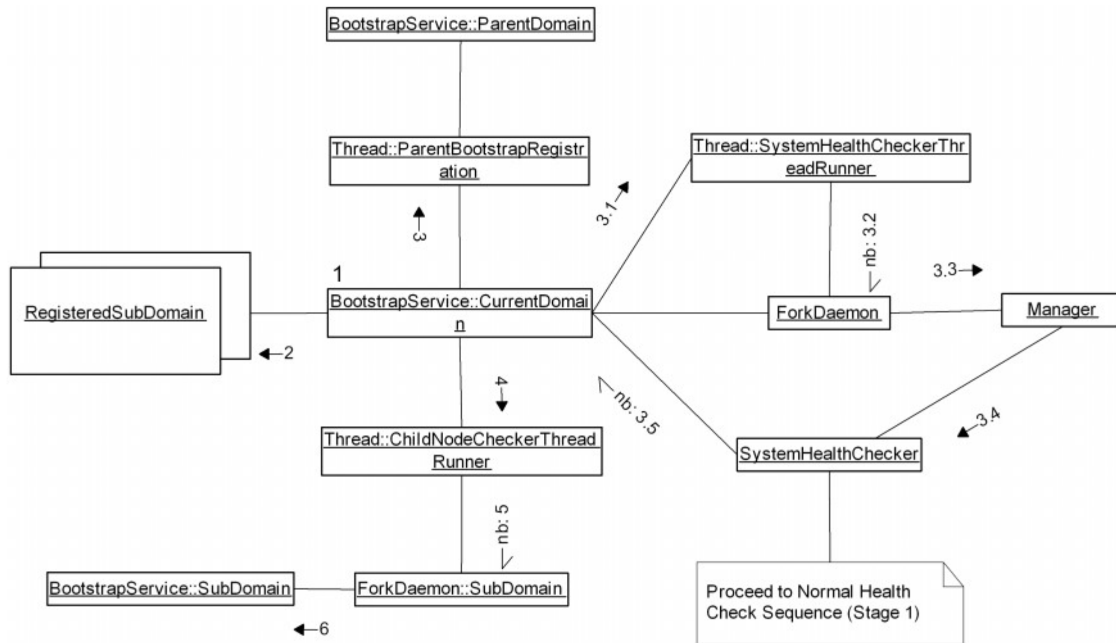


Figure 3-16 Starting BootstrapService of a Domain

1. Initialize the Bootstrap node from config file, including domain hierarchy and locators of ForkDaemons, RegistryForkDaemon, MessagingNodeDaemons. NB transport is initialized for NB communications with other domains
2. If the current domain is not a leaf node, register all sub-domains locally
3. If the current domain is not the root node, runs a thread that periodically sends a RegisterRenewMessage to the BootstrapService of its parent telling this domain's BootstrapService is running. If the domain is a leaf node, go to 3.1. Else go to 4
   - 3.1. Starts a thread that periodically spawns a SystemHealthCheck process for each registered ForkDaemon.
   - 3.2. Spawns a SystemHealthChecker process by sending a ForkProcessMessage to ForkDaemon with the "healthcheck" parameter
   - 3.3. ForkDaemon spawns the Manager process with the "healthcheck" parameter.
   - 3.4. Manager starts the SystemHealthChecker thread. System undergoes Normal Health Check Sequence (Please refer to section 3.3.4.3 for details). BootstrapService waits 10 seconds for the reply from SystemHealthChecker
   - 3.5. The replied status from SystemHealthChecker is either COMPLETE, UNKNOWN or RUNNING. Repeat 3.1 after some sleep
4. If the node is not a leaf node, spawns a thread that periodically checks the status of ALL RegisteredSubDomains (RSD). Under the Health Check mechanism, all RegisteredSubDomains are supposed to send a RegisterRenewMessage to its parent.
5. If no RegisteredRenewMessage is received from a SubDomain within a specified amount of time, the thread spawns a BootstrapService of the SubDomain remotely by sending a ForkProcessMessage to its ForkDaemon
6. ForkDaemon creates the BootstrapService of the SubDomain

1.3.4.3 Normal Health Check Sequence (Stage 1)

System Health Check has a number of stages. During the first state, Bootstrap Service checks if the Registry is present. If not, creates a Registry process using the Fork Daemon.

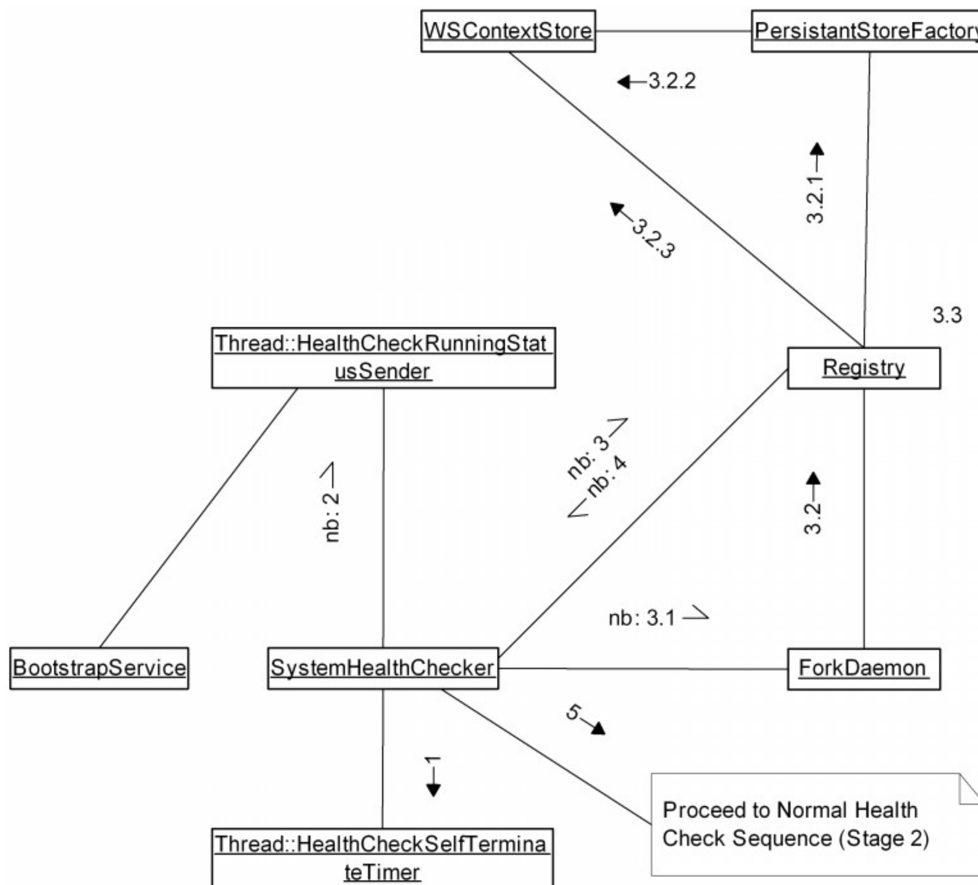


Figure 3-17 Normal Health Check Sequence (Stage 1)

1. After NB transport is initialized, a thread is started that automatically kills the the health checker if it is still running after 60 seconds
2. A thread is started that automatically notifies the BootstrapService at an interval of 2 seconds that the health checker is running
3. Checks if there is a Registry running in the domain by sending a RegistryQueryMessage to the defined Registry locator. If a RegistryQueryResponse message is received, go to 4. If no, go to 3.1
   3.1. Try spawning a Registry process by sending a ForkProcessMessage to ForkDaemon. Max retries = 5. After each retry, repeat 3. If number of retries reached, health checker terminates with abnormal exit status
   3.2. ForkDaemon creates the Registry process. Registry checks if persistent storage is used in configuration file (mgmtSystem.conf). If yes, go to 3.2.1. Otherwise persistent storage won't be used and everything will be saved in memory. Please proceed to 3.3

3.2.1. Registry asks PersistantStoreFactory for an instance of WSContextStore, which is responsible for storing and retrieving settings from persistent storage (e.g. relational database)

3.2.2. WSContextStore is initialized by making connections to various components defined in WSContext and removing all previous entries (e.g. registered service adapters, service policy, service status etc.). If any errors occur during initialization, go to 3.3 and everything will be saved in memory

3.2.3. Registry loads all settings from WSContextStore to in memory hash tables

3.3. Registry initializes NB transport by subscribing to two topic – one common to all registries and one uniquely identify itself. Registry spawning process has been finished. Go back to 3

4. Registry responds to SystemHealthChecker with the number of managers and service adapters expected in the domain.
5. System now enters health check stage 2. Proceed to section 3.3.4.4 .

1.3.4.4 Normal Health Check Sequence (Stage 2)

System Health Check has a number of stages. During the second stage, Bootstrap Service checks if enough Managers are spawned as defined in the configuration file.

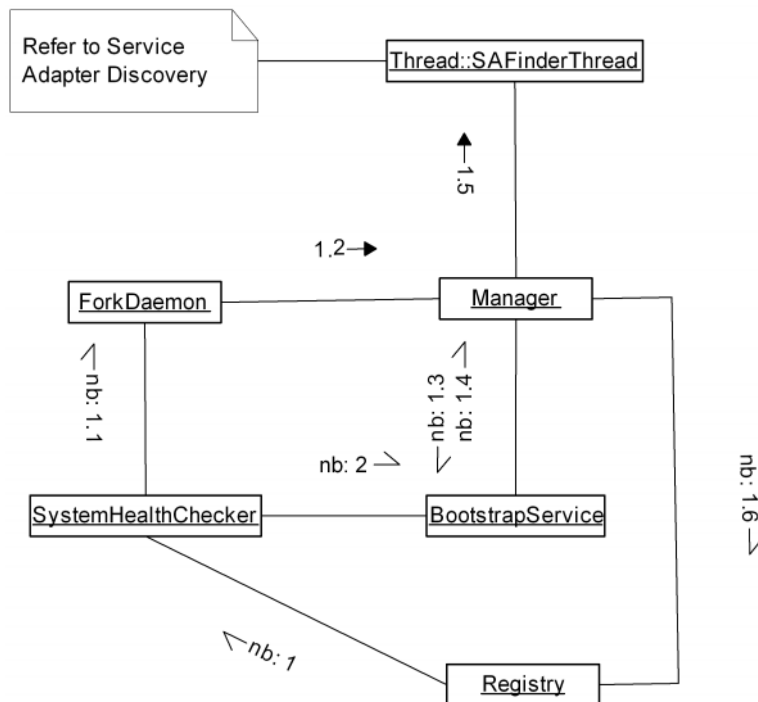


Figure 3-18 Normal Health Check Sequence (Stage 2)

1. The Registry responds to SystemHealthChecker with the number of managers and service adapters expected in the domain. If there are enough managers for all RegisteredServiceAdapters, go to 2. Otherwise go to 1.1

1.1. For each Manager lacking, create a Manager process without the "healthcheck" parameter sending a ForkProcessMessage to ForkDaemon
1.2. ForkDaemon creates the Manager process
1.3. Request system configuration from BootstrapService, including locator of Registry, ForkDaemon
1.4. BootstrapService replies with system configuration
1.5. Initialize NB transport support. Starts a SAFinderThread which keep sending FindSAToManageMessage to Registry requesting corresponding ServiceAdapters to manage. If no reply from Registry, the request is repeated periodically at 2 second interval. For details of this part, please refer to section 3.3.4.6 .
1.6. The Manager periodically sends a RegisterRenewMessage to the Registry to notify its presence
2. SystemHealthChecker sleeps for 10 seconds to allow any pending processes to instantiate. Then it checks whether all expected processes are up and running. If yes, it sends a SystemHealthCheck message to BootstrapService, notifying that System Health Check is completed and then terminates itself. Otherwise, it checks the system's health from stage one again (section 3.3.4.3) and tries spawning the process(s) missing

1.3.4.5 Registered Service Adapter Health Check Sequence

SAMModule notifies the Service Adapter which Manager it should send heart beat messages to

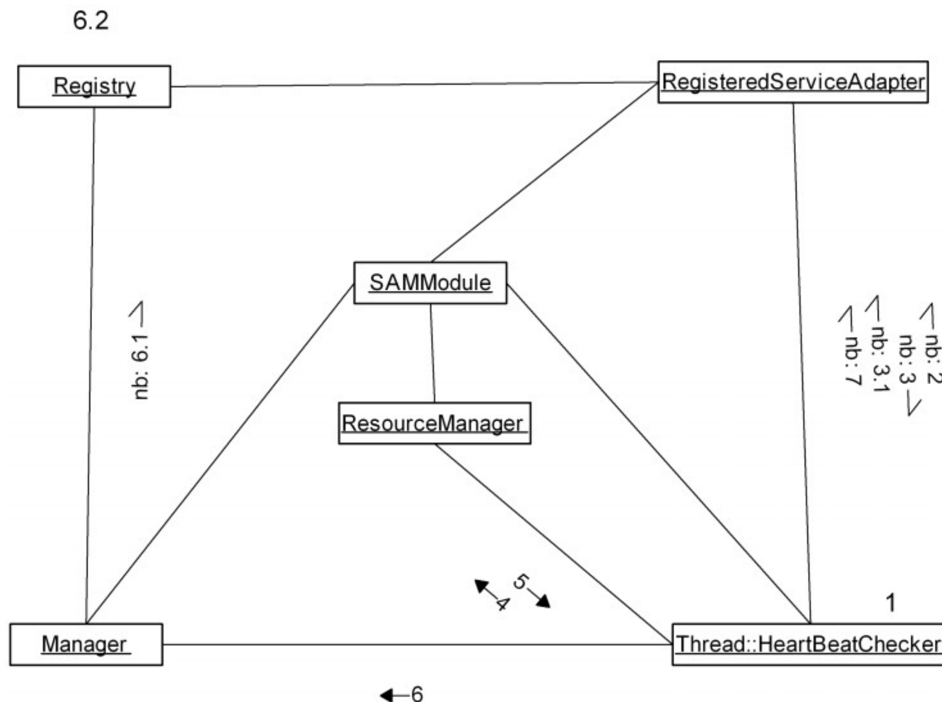


Figure 3-19 Registered Service Adapter (RSA) Health Check Sequence

1. Checks if the associated RSA has sent a HEARTBEAT within the specified interval. If yes, sleep for a while and do 1 again. Else go to 2
2. Sends a GetCurrentManager message to the associated RSA to check if it is the RSA's current owner. If RSA replies, go to 3. Else go to 4
3. If UUID of RSA's current owner matches with this SAMModule, go to 3.1. Else go to 4

    3.1. Sends a HEARTBEAT message to the RSA and wait. If RSA replies within a time limit, sleep for a while and do 1 again. Else go to 4
4. Ask ResourceManager(RM) whether to release the RSA.
5. If RM knows that the RSA is up and running, go to 7. Else go to 6
6. Notifies the Manager that the associated RSA is unreachable.
    6.1. Sends a UPDATE_SA_STATUS message to the Registry, saying that the RSA is UNREACHABLE
    6.2. Registry performs status update
7. Re-register with the RSA by sending a HEARTBEAT to it. Sleep for a while and do 1 again

1.3.4.6 Service Adapter Discovery

System Health Check checks if every Service Adapter is associated with its Manager.

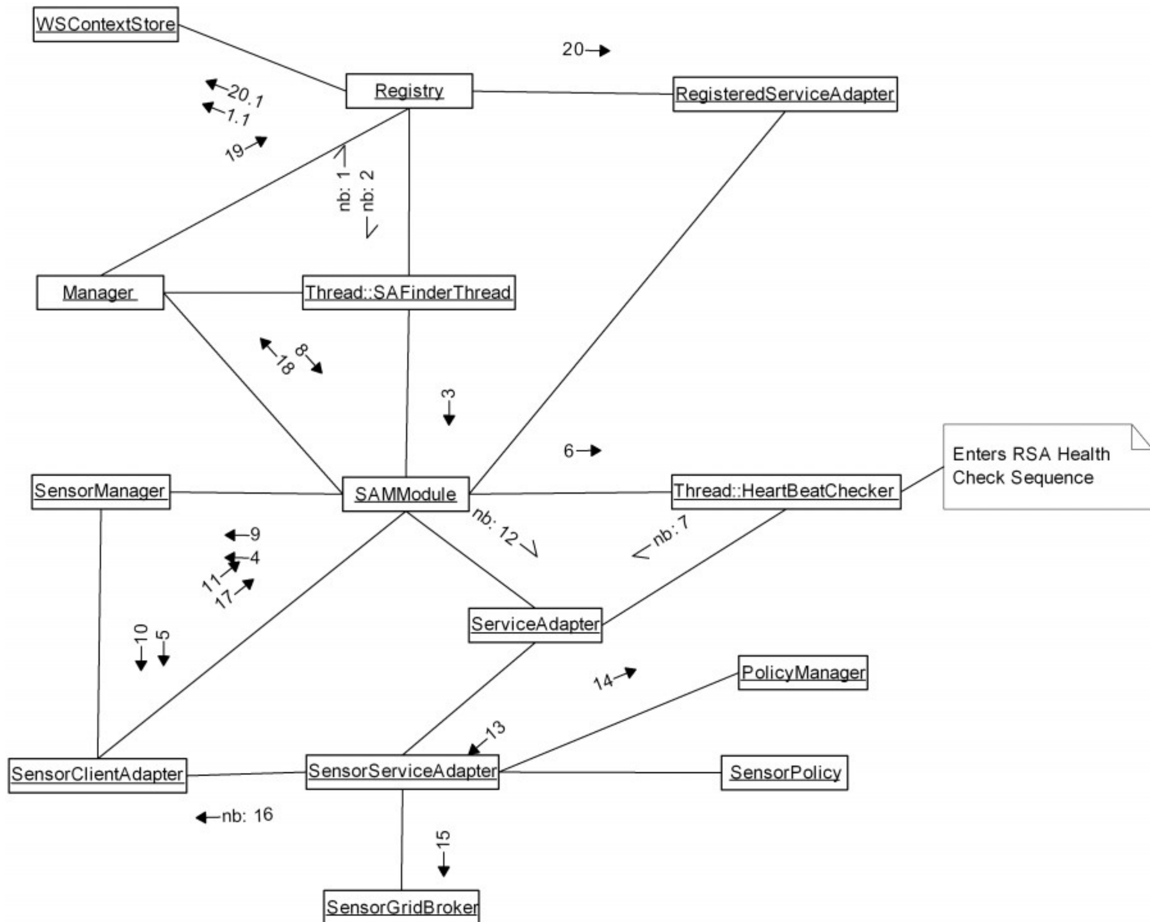


Figure 3-20 Message flow of service adapter discovery in a sensor grid

1. SAFinderThread sends a FindSAToManageMessage to Registry. If persistent storage is used in the Registry, go to 1.1. Otherwise go to 1.2.

1.1. Registry retrieves the information of a list of Registered Service Adapters from WSContextStore
1.2. Registry replies with ServiceAdapterToManageMessage to the Manager if there is at least one ServiceAdapter (SA) which does not have an associated SAMModule. Status of the SA is set to MANAGED. At most one SA will be replied for each request. If there are no SA to manage, the Manager shutdowns itself.
2. For each SA, the Manager creates a SAMModule which manages the SA.
3. SAMModule creates a specific type of ResourceManager specified in the SA (in ServiceAdapterInfo), and starts the ResourceManager in a new Thread. For sensors, a SensorManager (ResourceManager for sensors) is instantiated
4. A SensorClientAdapter is instantiated. The SAMModule of SensorManager is passed as message sender and the locator of the associated SA is set as message destination
5. SAMModule starts a HeartBeatCheckerThread that periodically checks 1) if SA is up and running 2) if SA is still associated with this SAMModule (possibly taken control by other Managers)
6. Sends a setHeartBeatLocator message to SA to associate the SA with this SAMModule and tells SA the locator of Manager which heart beat messages should be sent to. Afterwards, HeartBeatCheckerThread enters the loop of SA health check (please refer to section 3.4.5 - Registered Service Adapter Health Check Sequence)
7. Sends a GetServicePolicyMessage to SAMModule, request for the policy of the associated resource (i.e. sensor)
8. Forwards the request to SensorManager by calling getServicePolicy()
9. Invokes the associated SensorClientAdapter's getServicePolicy()
10. Sends a Wxf_Get message to the associated SensorServiceAdapter through SAMModule
11. Wraps the message with ServiceSpecificMessage and forwards it to the associated ServiceAdapter
12. Invokes processSOAPMessage of the associated SensorServiceAdapter (SSA)
13. If SensorPolicy has been defined, serialize it with PolicyManager. Otherwise, just create an empty message
14. If this is the first time SSA is assigned to a Manager, starts a SensorGridBroker which notifies SG of its presence
15. Sends back a response message with the serialized policy (if any)
16. Forwards the response to SAMModule
17. Forwards the response to Manager
18. Forwards the response to Registry
19. Updates the policy of the SA to the corresponding RSA in Registry. If persistent storage is used, go to 19.1; otherwise, go to 19.2
19.1. The RSA is stored in WSContextStore
19.2. The RSA is stored in memory

1.3.5 Deploying and Disconnecting sensors

1.3.5.1 Deploying a GPS Sensor

The message flow of deploying any sensors in a sensor grid is similar. For illustrative purposes, the message flow of deploying a GPS sensor is shown in Figure 3-21.

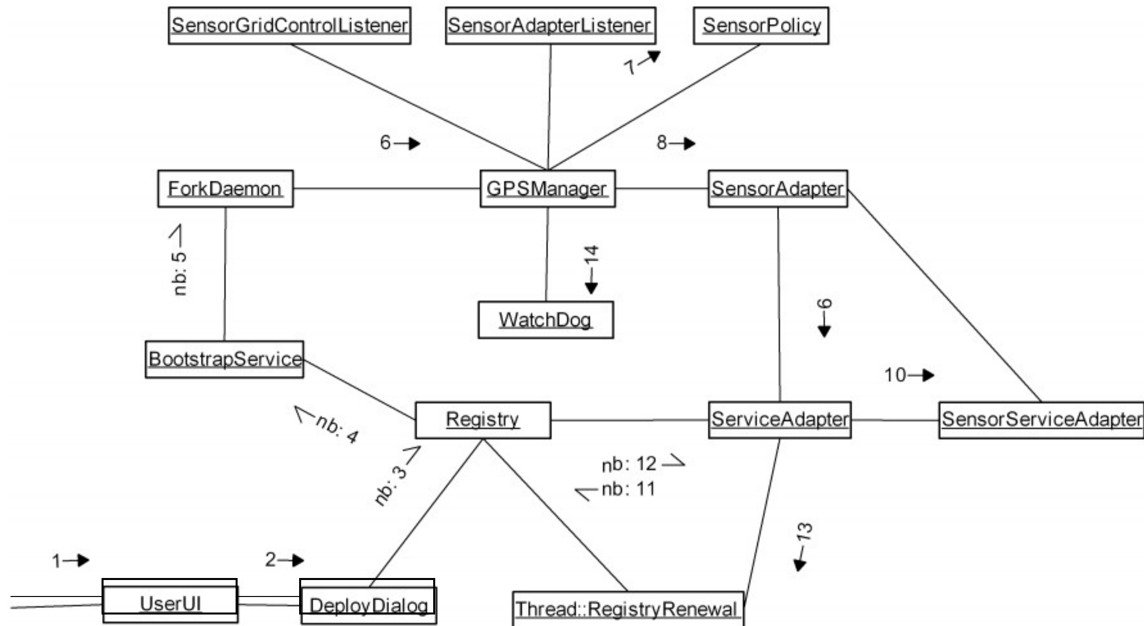


Figure 3-21 Deploying a GPS Sensor

1. User chooses a domain and clicks “deploy”
2. UserUI creates a DeployDialog
3. User defines the policies of the sensor and clicks “ok”. A ForkProcessMessage is sent to the Registry to spawn a sensor client program
4. The message is forwarded to BootstrapService
5. The message is forwarded to ForkDaemon
6. ForkDaemon starts the type of sensor client program according to policy defined. Suppose user needs a GPS sensor. ForkDaemon creates a GPSManager process
7. Creates an instance of SensorPolicy according to the type of sensor and classification.
8. Creates an instance of SensorAdapter, passing in a SensorAdapterListener, SensorGridControlListener and SensorPolicy
9. Creates an instance of ServiceAdapter (SA) with parameters “saType=cgl.hpsearch.sensor.SensorServiceAdapter” and “manType=cgl.hpsearch.sensor.SensorManager”
10. Subscribes to the SA's own NB topic. Instantiates a SensorServiceAdapter according to “saType”
11. Sends a RegisterRenewMessage to the Registry
12. If the SA is new to the Registry, it registers the SA, set SA's status to REGISTERED and replies SA with the new instanceId. If the SA is already registered, renew the status of SA according to its instanceId

13. Subscribes to a new NB topic according to the returned instanceId. Starts a new thread responsible for sending RegisterRenewMessage (heart beat) to the Registry. SA enters a state that keep tracking if NB connection is down. If yes, try to reconnect
14. GPSManager makes physical connection to the sensor, and starts a WatchDog which monitors the physical connection

After the new SA is registered in the registry, the Normal Health Check Sequence for Managers (Stage 2) will discover the new SA is not yet managed. A Manager will be assigned to it. For details please refer to session 3.3.4.4 .

1.3.5.2 Disconnecting a Sensor

There are two ways to disconnect a sensor. The first way is to terminate the Sensor Client Program explicitly. The second way is to do it through GB's management console. The diagram below shows the message flow of disconnecting a sensor through GB's management console.

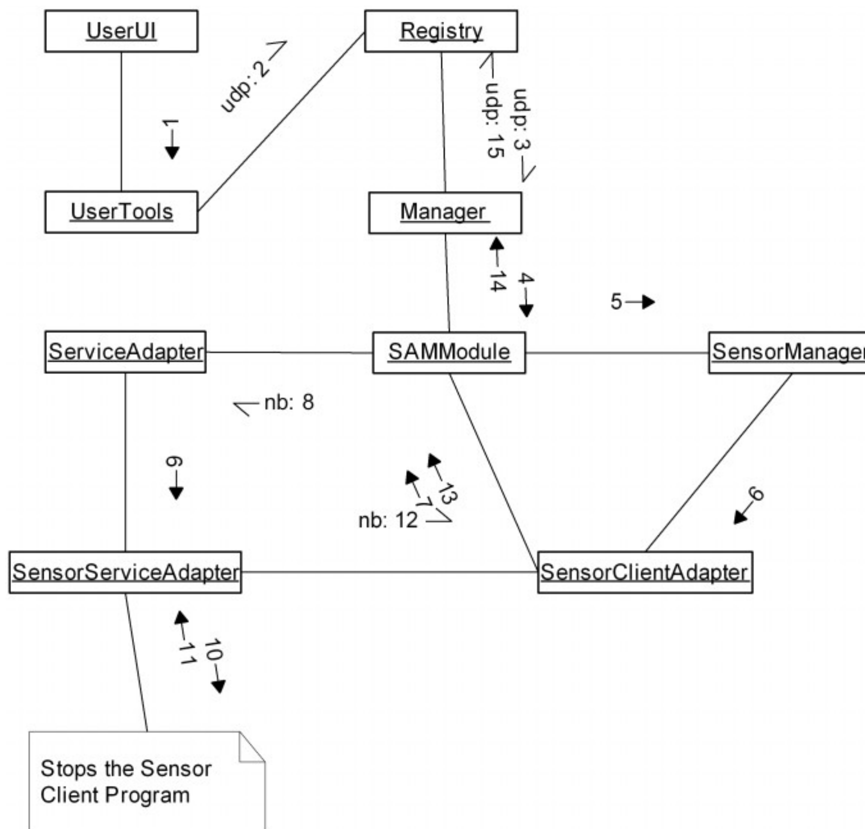


Figure 3-22 Disconnecting a sensor by using the Grid Builder management console

1 User selects a sensor in GB's management console and clicks "Stop". UserUI invokes sendRunMessage() of UserTools
2 UserTools creates a RunServiceMessage with parameters indicating the message is for disconnecting a sensor. The message is sent to Registry

3. Registry locates the Manager of the corresponding RegisteredServiceAdapter and forwards the message to it
4. Manager locates the corresponding SAMModule responsible for managing the ServiceAdapter and forwards the message to it
5. SAMModule forwards the message to the associated SensorManager
6. SensorManager forwards the message to the associated SensorClientAdapter
7. SensorClientAdapter sends a Wxf_Delete message to the associated SensorServiceAdapter through SAMModule
8. Wraps the message with ServiceSpecificMessage and forwards it to the associated ServiceAdapter
9. Invokes processSOAPMessage of the associated SensorServiceAdapter (SSA)
10. SensorServiceAdapter stops the sensor through SSAL. For details please refer to section 3.4.4.9
11. An error report message is replied indicating if any error exists
12. Forwards the reply to SensorClientAdapter
13. Wraps the reply with a RunServiceResponse message, and sends it back to Registry through SAMModule
14. Forwards the response to Manager
15. Forwards the response to Registry
16. Registry does not do anything to the response

1.4 Sensor Grid

1.4.1 Overall Architecture of Sensor Grid and Related Modules

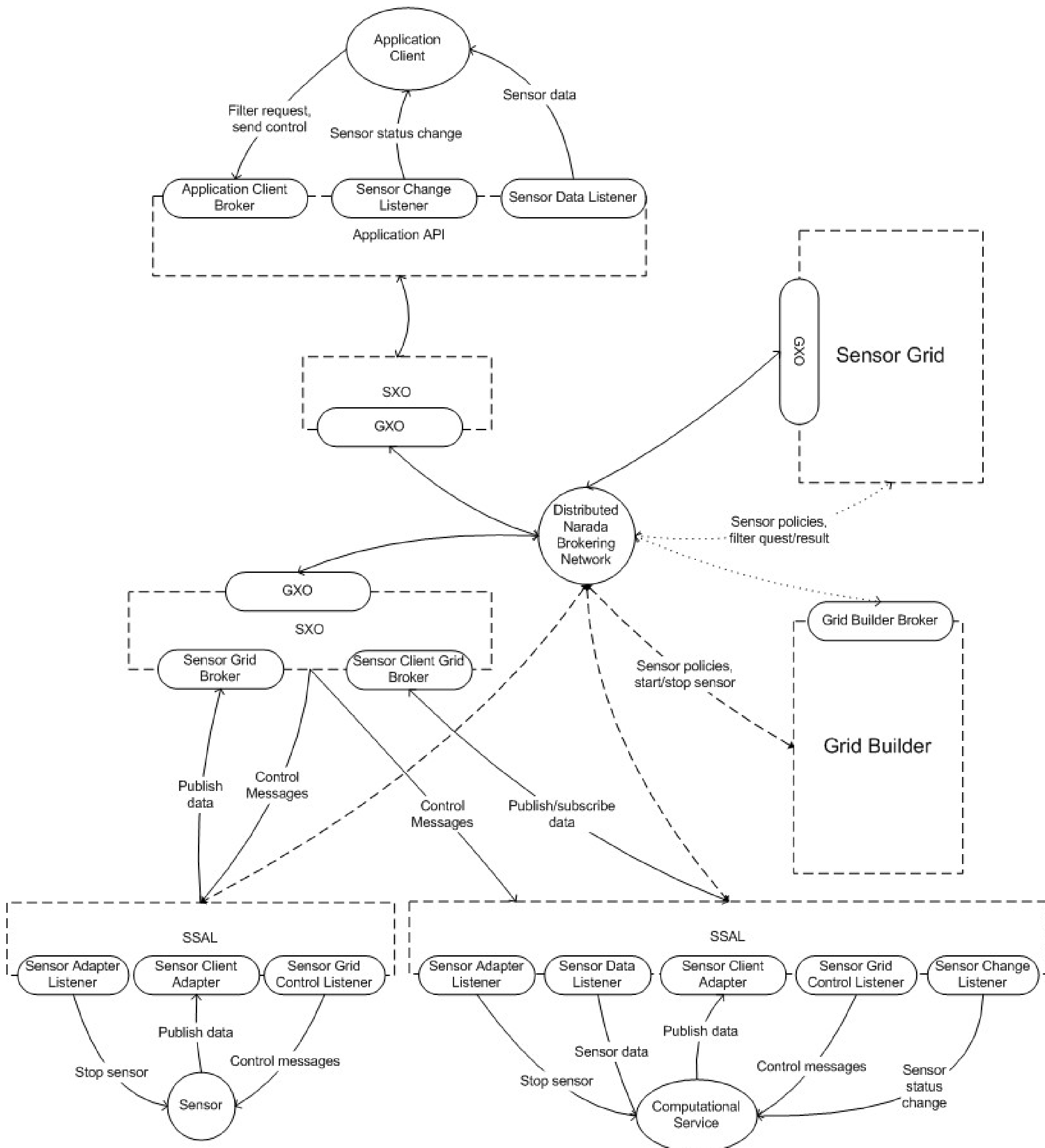


Figure 3-23 Overall Architecture of Sensor Grid and related Modules

Sensor Grid (SG) is the brokering module of SCMW connecting the sensors, application clients and Grid Builder. It serves two functions:

1.4.1.1 Message Brokering

It enables the flow of messages among all parties including:

1. sensor data
2. sensor control messages
3. filtering requests and results
4. changes of sensor status
5. sensor policies

The following modules are essential for communication among the parties.

GXO

GXO is a messaging layer which uses NaradaBrokering (NB) for message passing. It has the following characteristics:

1. supports a lot of transport layer protocols, including tcp, niotcp, udp, http, https and so on
2. abstracts messages into byte, text and object messages which performs automatic message serialization and de-serialization
3. uses a topic-based, publish and subscribe model which eliminates the need for identifying end points explicitly
4. allows flexible construction of brokering network

With the use of GXO, messages can propagate to the destination with minimum programming effort.

SXO

SXO is a layer built on top of GXO. It is the internal API which facilitates communications between sensors, application clients and SG. It handles the connection and disconnection of both sensors and application in a seamless and fault-tolerant manner. It contains logic and libraries for both Application API and SSAL to communicate with applications and sensors respectively.

Application API

All kinds of applications communicate with SCMW through the same API. The Application API provides libraries for applications to:

1. access data and metadata of sensors
2. send control messages to sensors
3. notified for change of sensor status
4. send filter requests to SCMW

These actions are done with the help of the following modules in the API:

Application Client Broker

Interface used by application clients to send requests to SG, such as sending filter requests to SG and control messages to sensors (through SSAL).

Sensor Change Listener
Interface used by application clients to receive messages from SG such as sensor status change.

Sensor Data Listener
Interface used by application clients to receive data from sensors.

To support different applications, Application API in turn communicates with SCMW through SXO. For more detailed description of Application API, pleased refer to section 3.5.

SSAL

All sensors communicate with SCMW through SSAL. Remember each sensor has a corresponding Sensor Client Program (SCP) to communicate with SCMW. SSAL provides libraries for sensors to do the following through SCP:

1. publish data
2. receive control messages
3. receive stop request from SCMW
4. subscribe to data of another sensor
5. listen to status change of subscribed sensor

Not all kind of sensors have to use all functionalities listed above. Remember sensors can be further classified into normal sensors and Computational Service. In fact these two categories utilize different subset of classes in SSAL. Some of the important modules of SSAL are listed below:

Sensor Client Adapter
An interface for publishing data

Sensor Data Listener
An interface for listening to data from subscribed sensors. Used by Computational Service

Sensor Adapter Listener
An interface for listening to stop requests from SCMW. The SCP should terminate upon receiving the request

Sensor Change Listener
An interface for being notified when the subscribed sensor has any status change. Used by Computational Service

Sensor Grid Control Listener
An interface which sensors listen to control messages

For more detailed description of SSAL please refer to section 3.5.2 .

1.4.1.2 Application Management

In SCMW, SG is responsible for maintaining the state of the whole system. For each deployed sensor and running application, SG caches down their presence and their relationships with one another. The figure below shows a scenario which 2 applications and 5 sensors are connected to SG. The four tables shows how SG maintains the state of the system, they include:

A list of online sensors (Table S)
SG maintains a list of online sensors which dynamically changes with the deployment status of the sensor

Application to sensor mapping (Table A_S)
Each application needs a different set of online sensors according to its filtering criteria. This is to make sure that sensors which are not concerned by the application do not hold unnecessary resources. A table is maintained to remember this mapping

Application to filter mapping (Table S_F)
Each application has its own filter, which are the criteria that define which sensors are needed by the application. The filter can be modified by the application at any time.

Sensor to sensor policy mapping (Table S_P)
Sensor Policies defines the characteristics of sensors. It is defined by Grid Builder before deployment. The sensor policy is obtained from GB and cached whenever a sensor is being deployed.

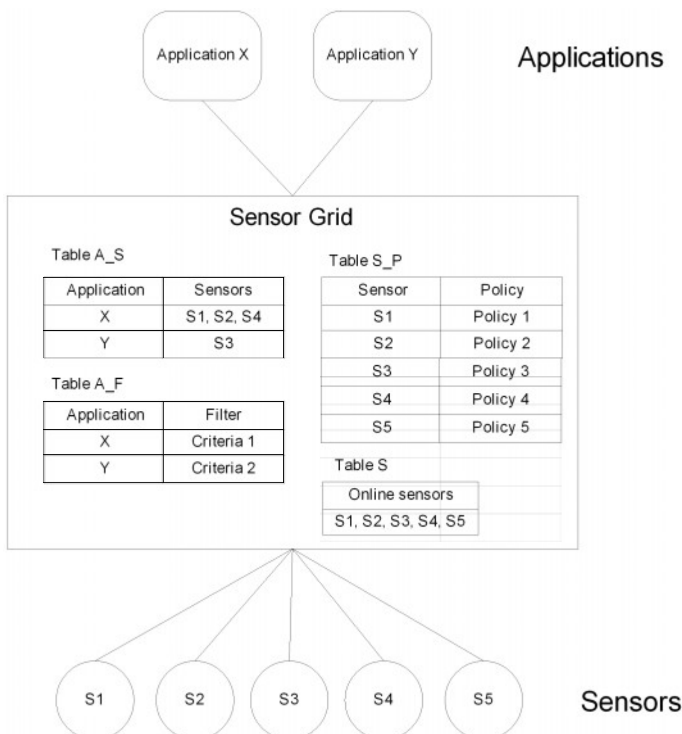
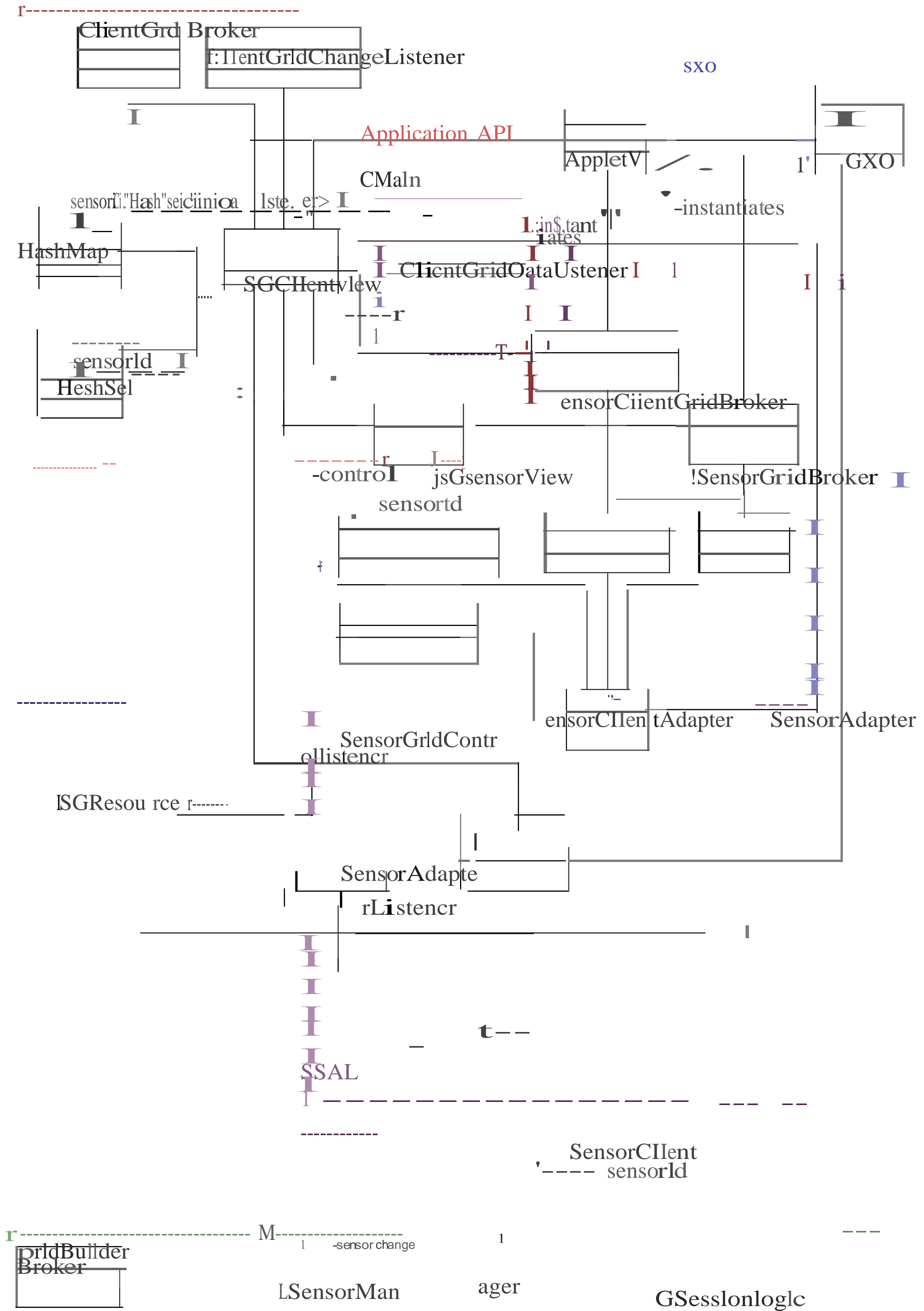


Figure 3-24 SG System Management

1.4.2 Significant Classes

1.4.2.1 Class Diagram



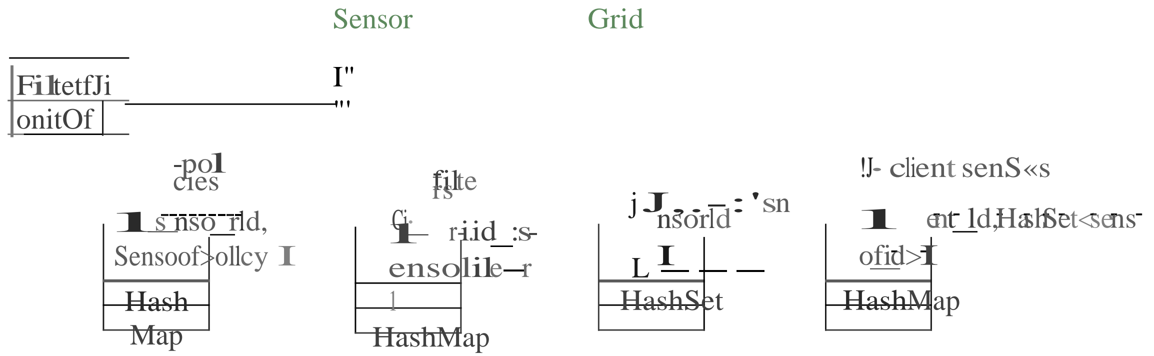


Figure 3-25 Class Diagram of SG, Sensor and Application Client

The figure above shows the class diagram of significant classes in SXO and SG. Within SXO, classes used by application clients and classes for sensors are also indicated respectively.

1.4.2.2 Class Description

This section provides brief description of important classes of SG and SSAL.

Class name:	ClientGridBroker
Package name:	com.anabas.sensorgrid.client
Description:	Part of the Application API. Provides the interface for external applications to communicate with SG and sensors. Notifies GXO for application joining
Important interface:	setFilter(), sendControl(), subscribeSensorData(), unsubscribeSensorData()

Class name:	ClientGridChangeListener
Package name:	com.anabas.sensorgrid.client
Description:	Part of the Application API. Provides the interface for receiving sensor status change due to sensor deployment, disconnection and filtering
Important interface:	handleSensorInit(), handleSensorChange()

Class name:	SGClientView
Package name:	com.anabas.sensorgrid.session.sharedlet
Description:	Part of SXO. Contains most of the application-client-side logic for the communication with SG and sensors, such as receiving sensor change, sending filter to SG and sending control messages to sensors. All NB topic and streams are handled here
Important interface:	setChangeListener(), startConnection(), subscribeSensorData, unsubscribeSensorDawta(), setFilter(), sendControl()

Class name:	ClientGridDataListener
Package name:	com.anabas.sensorgrid.client
Description:	Part of the Application API, responsible for notifying the application on sensor data arrival. If the application clients wants to receive data from a particular sensor, it has to create a ClientGridDataListener for that sensor. Afterwards, the listener will be notified for data arrival
Important interface:	handleSensorData()

Class name:	SGSensorView
Package name:	com.anabas.sensorgrid.session.sharedlet
Description:	Part of SXO. Contains most of the sensor-side logic for the communication with applications, such as publishing data and receiving control messages. All NB topics and streams are handled here
Important interface:	setControlListener(), publishData()

Class name:	SensorGridBroker
Package name:	com.anabas.sensorgrid.sensor
Description:	Part SXO. Brokers communication between SSAL, SG and sensors. Notifies GXO for sensor deployment and disconnection
Important interface:	publishData(), close()
Class name:	SensorClientGridBroker
Package name:	com.anabas.sensorgrid.sensorclient
Description:	Part of SXO. Brokers communication between SSAL, SG and service sensors. Notifies GXO for sensor deployment and disconnection
Important interface:	publishData(), sendControl(), setFilter(), subscribeSensorData(), unsubscribeSensorData()
Class name:	SensorGridControlListener
Package name:	com.anabas.sensorgrid.sensor
Description:	Part of the SSAL. Provides the interface for receiving control messages
Important interface:	handleSensorControl()
Class name:	SensorAdapter
Package name:	com.anabas.sensor.sensoradapter
Description:	Part of SSAL. Provides the interface for sensors to publish data to applications
Important interface:	publishData(), start(), close()
Class name:	SensorAdapterListener
Package name:	com.anabas.sensor.sensoradapter
Description:	Part of SSAL. Responsible for receiving termination commands from GB
Important interface:	handleSensorConnectionLoss(), handleSensorStopRequest()
Class name:	FilterMonitor
Package name:	com.anabas.sensorgrid.session.sharedlet
Description:	Actually this is an inner class of SensorManager responsible for periodic checkup to update the set of sensors for each application according to their corresponding filter
Important interface:	0
Class name:	SensorManager

Package name:	com.anabas.sensorgrid.session.sharedlet
Description:	Part of SG. Contains the logic for managing all connected applications and sensors. Maintained HashSets and HashMaps to cache sensor policies, applications' filters and sets of sensors mapped to each application.
Important interface:	addSensor(), removeSensor(), addClient(), startClient(), removeClient(), setFilter()

Class name:	SGSessionLogic
Package name:	com.anabas.sensorgrid.session.sharedlet
Description:	Part of SG. Responsible for handling communications with all applications and sensors through GXO. Performs state update

Important interface:	through SensorManager for every connections and disconnections of sensors and applications (notified by GXO) userJoined(), userLeft()

Class name:	AppletVCMain
Package name:	com.anabas.sharedlet.appletframework
Description:	Part of GXO. Resides at client side (applications and sensors) for allocating and releasing resources
Important interface:	allWindowsClosed()

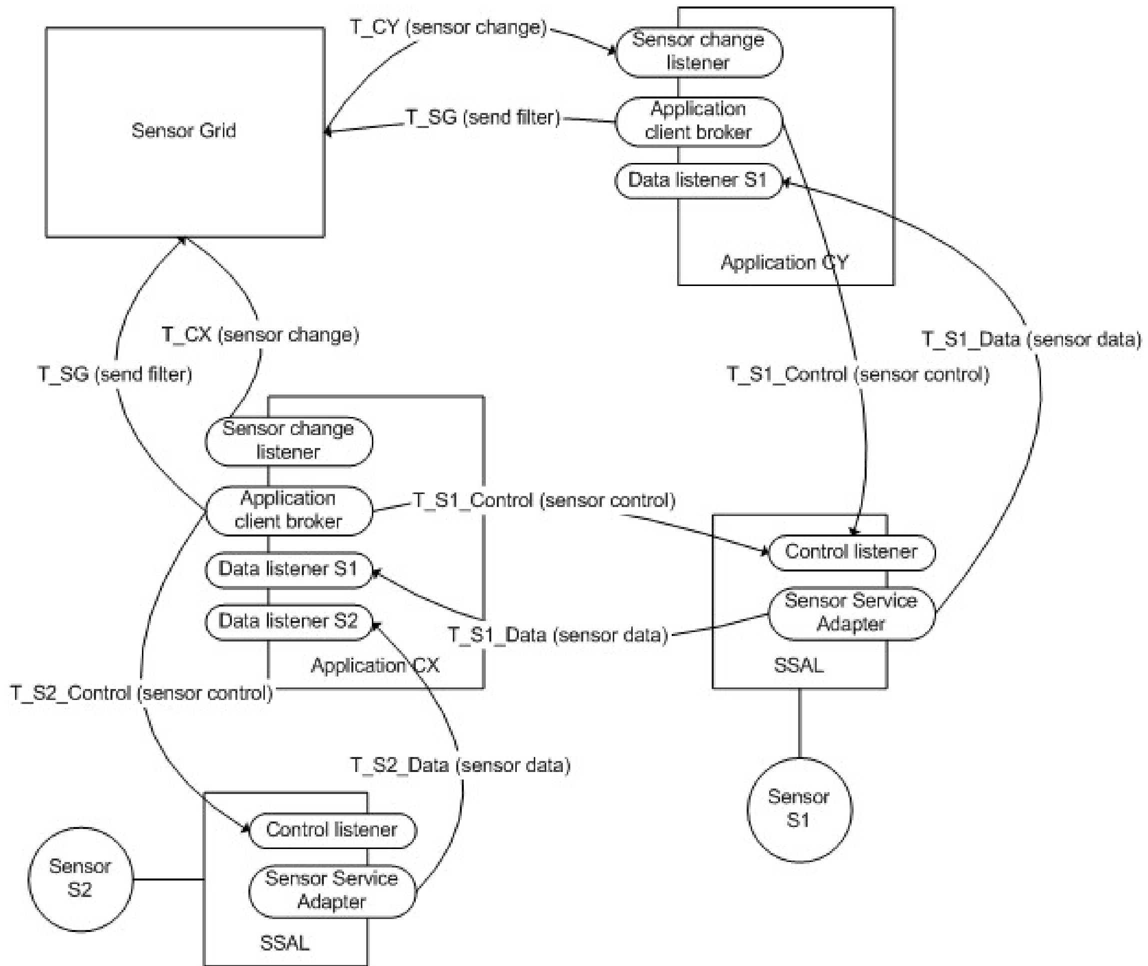
1.4.3 Important Features

1.4.3.1 NB Data Flow and Topic Management

Communication between applications, sensors and SG relies on NaradaBrokering (NB) for communication. This section provides a brief description of data flow between the three parties.

Each sensor creates a topic for publishing data and a topic for subscribing control messages. When an application is notified by SG for a new sensor, it subscribes to the two topics of the corresponding sensor directly for receiving data and publishing control messages.

For the communication between applications and SG, each application creates its own topic using its unique id for receiving sensor change notification. SG also creates a topic to receive filter requests from all applications.



Stream	NB Topic
T_SG	application/x-sharedlet-sensorgrid/private
T_CY	application/x-sharedlet-sensorgrid/client/CY
T_CX	application/x-sharedlet-sensorgrid/client/CX
T_S1_Data	application/x-sharedlet-sensorgrid/sensordata/S1
T_S1_Control	application/x-sharedlet-sensorgrid/sensorcontrol/S1
T_S2_Data	application/x-sharedlet-sensorgrid/sensordata/S2
T_S2_Control	application/x-sharedlet-sensorgrid/sensorcontrol/S2

Figure 3-26 Message flow between a Sensor Grid (SG), applications and sensors

1.4.4 Detailed Description

In this section, message flow of various operation of SG will be discussed at Class level using UML collaboration diagrams.

1.4.4.1 Sensor Grid Startup

Sensor Grid starts a perpetual session.

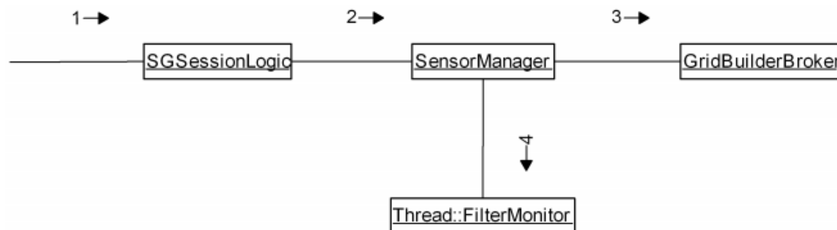


Figure 3-27 A Sensor Grid startup sequence

1 An instance of SGSessionLogic is created by the framework
2 An instance of SensorManager is created, which is responsible for handling sensor-application interaction
3 An instance of GridBuilderBroker is created, which is responsible for obtaining SensorPolicy from Grid Builder
4 A thread is created which do filtering for different application-clients for every 5 seconds

1.4.4.2 Deploying a Sensor

When deploying a sensor through the Grid Builder, sequencecs of messages are invoked to enable the management of deployed sensors as well as mechanisms to filter sensors based on sensor policies. Message flow when a sensor is deployed through Grid Builder is illustrated in Figure 3-28

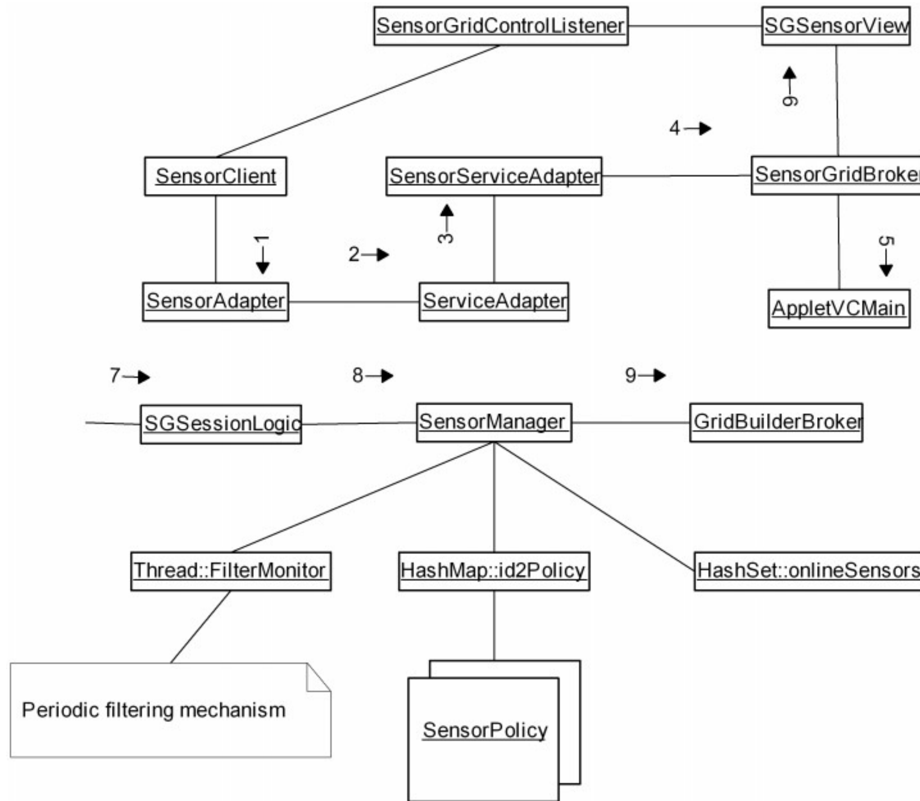


Figure 3-28 Message flow when depolying a sensor through the Grid Builder

1. The sensor client program instantiates SensorAdapter when it is started by Grid Builder
2. SensorAdapter instantiates ServiceAdapter, which is later on managed by Grid Builder
3. Service Adapter instantiates SensorServiceAdapter, which resides in SSAL for communication with SensorManager of Grid Builder
4. SensorServiceAdapter instantiates SensorGridBroker, which communicates with Sensor Grid
5. SensorGridBroker initializes all parameters needed for the sensor to join the Sensor Grid, including sensorId and system configuration, then instantiates AppletVCMain with all the parameters which tells the framework to prepare for a sensor client. Sleep for 5 seconds.
6. A SGSensorView is instantiated by the framework, which is responsible for message passing between application clients, sensors and Sensor Grid. A unique NB stream is created for publishing sensor data and another one created for subscribing control messages. SensorGridBroker obtains a reference to SGSensorView from the framework and registers the SensorGridControlListener
7. The framework notifies that a new sensor has joined through the SessionListener interface of SGSessionLogic (userJoined()).

8 Invokes addSensor() of SensorManager. SensorManager caches down the sensor in HashSet and its Policy in HashMap
9 Asks Grid Builder for SensorPolicy of the sensor through the GridBuilderBroker interface (getPolicy())
10 FilterMonitor Thread will notify all application-clients the presence of new sensor if it matches with the Filter. Please refer to section 3.4.4.3 for details

1.4.4.3 Periodic Filtering

SG periodically checks the status of sensors and whether there are changes for each filter defined by applications. Below shows the message flow.

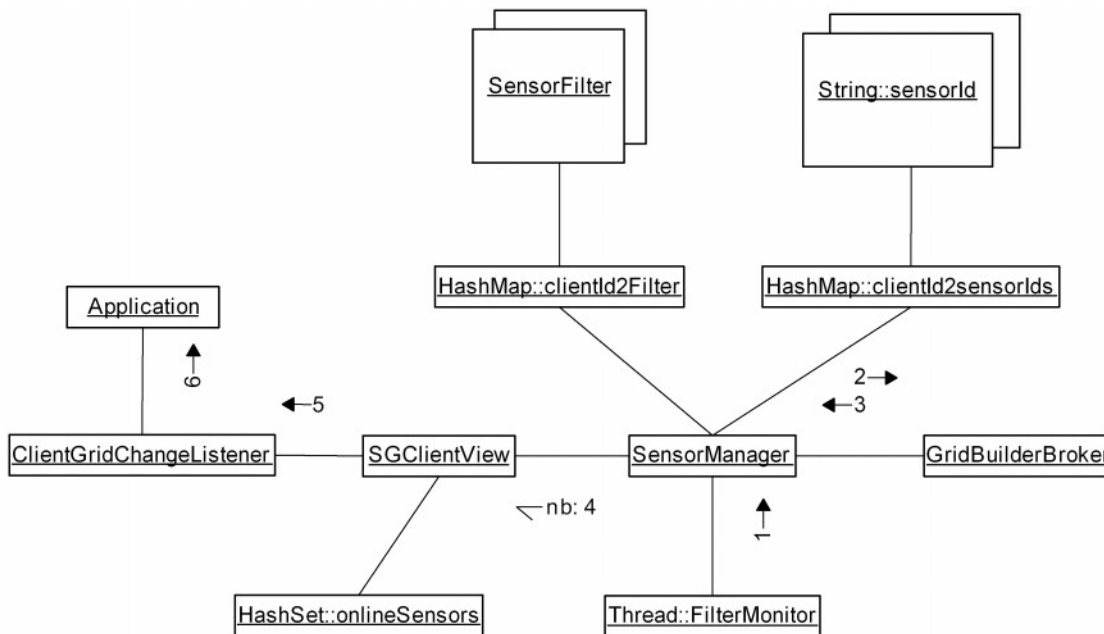


Figure 3-29 Sensor Grid message flow during periodic sensor filtering

1 Every 5 seconds, the FilterMonitor Thread performs a filtering sequence. For each registered application-clients, the corresponding Filter object is obtained from a HashMap. Invokes doFiltering() of SensorManager
2 Send a request to Grid Builder acquiring a list of sensors which matches the filtering criteria defined by the Filter
3 GridBuilderBroker returns a list of sensors fulfilling the criteria
4 Compare the list of returned sensors with the currently cached list of sensors for the application-client. Notifies the application-client all changes by sending a SENSOR_CHANGE message through a application-client specific NB stream
5 Updates the cached list of online sensors in HashSet. Invokes handleSensorChange() of the registered ClientGridChangeListener (Sensor Change Listener)
6 ClientGridChangeListener notifies application client of sensor change. Application client performs corresponding actions

1.4.4.4 Application Client Joining A Sensor Grid (SG)

When a sensor grid application client joins a sensor grid (SG), the message flow is illusgtrated as follows:

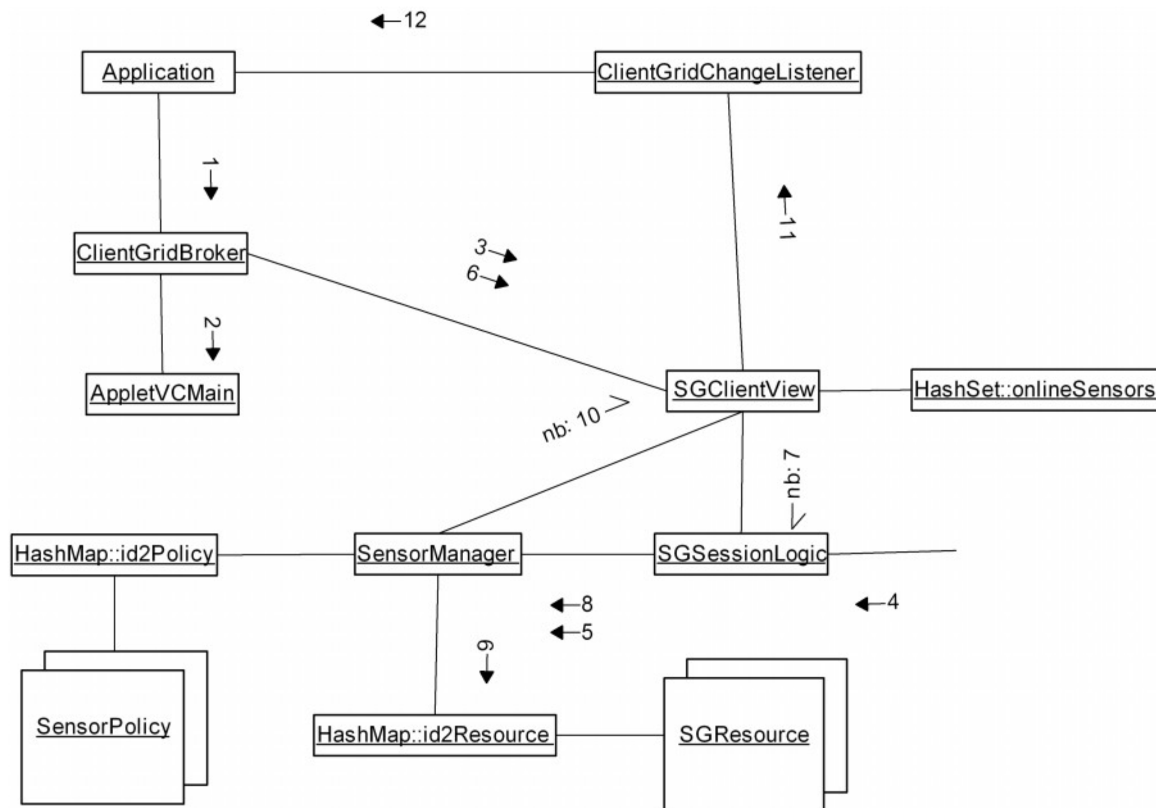


Figure 3-30 Message flow when an application joins a sensor grid

1 The application-client which implements the ClientGridChangeListener (Sensor Change Listener) interface, instantiates an instance of ClientGridBroker (Application Client Broker)
2 ClientGridBroker initializes all parameters needed for the application to join the Sensor Grid, including a generated client id which is unique to the system and client's system configuration, then instantiates AppletVCMain with all the parameters which tells the framework to prepare for an application client. Sleep for 5 seconds.
3 A SGClientView is instantiated by the framework, which is responsible for message passing between application clients, sensors and Sensor Grid. A unique NB stream is created for subscribing messages from Sensor Grid (e.g. sensor change information). ClientGridBroker obtains a reference to SGClientView from the framework and registers the ClientGridChangeListener
4 The framework notifies that a new application client has joined through the SessionListener interface of SGSessionLogic (userJoined()).
5 invokes addClient() of SensorManager. SensorManager initializes NB streams for communication with application client
6 Registers application client's ClientGridChangeListener. Invokes SGClientView's startConnection()
7 Sends a START_CLIENT message with its client id
8 Forwards the request to SensorManager

9 Creates a HashMap which maps the id of all online sensors to SGResource instances wrapping the policy and status of the sensors
10 Sends a INIT_SENSOR message to the client, containing the created HashMap
11 Updates the cached list of online sensors in HashSet. Invokes handleSensorInit() of the registered ClientGridChangeListener
12 ClientGridChangeListener notifies application client of sensor change. Application client performs corresponding actions

1.4.4.5 Sensor Publishing Data

After a sensor is deployed in a sensor grid, real-time stream of sensor data and metadata will be published to the sensor grid. The message flow of a sensor publishing data to the sensor grid in which application clients could subscribe to such live streams is illustrated in Figure 3-31.

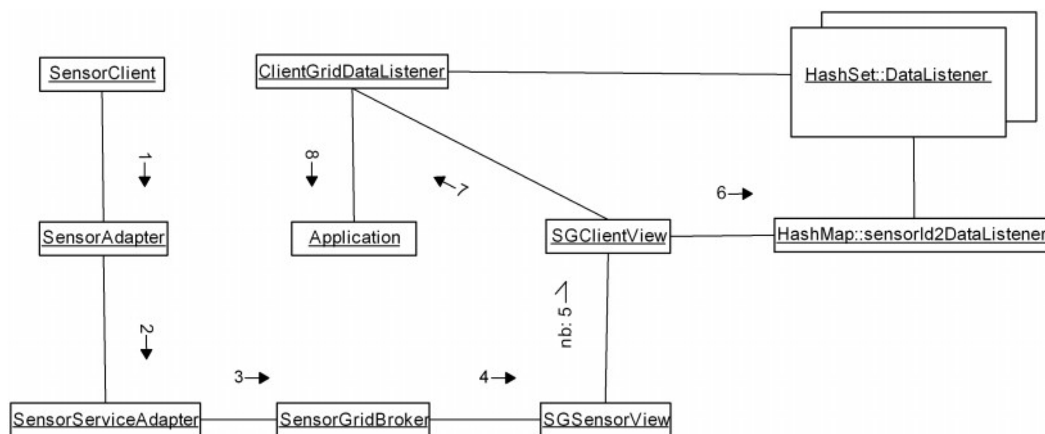


Figure 3-31 Message flow from deployed sensors to applications in a sensor grid

1 SensorClient publishes data by calling publishData() of SensorAdapter
2 SensorAdapter forwards the data to SensorServiceAdapter by calling publishData()
3 SensorServiceAdapter forwards the data to SensorGridBroker by calling publishData()
4 The data is forwarded to SGSensorView
5 Broadcast the data through the unique NB stream for the sensor
6 For ALL the SGClientViews which has subscribed to this NB stream, locates all registered ClientGridDataListeners (Sensor Data Listener) which has subscribed to data from this sensor
7 For each ClientGridDataListener found, notifies it for data arrival by invoking handleSensorData()
8 Notifies the application for data arrival

1.4.4.6 Subscribing Sensor Data

Applications that implement the SCMW API could receive relevant live sensor streams in the sensor grid by subscribing to them. The message flow of an application subscribes to live stream of a deployed sensor is shown below in Figure 3-32.

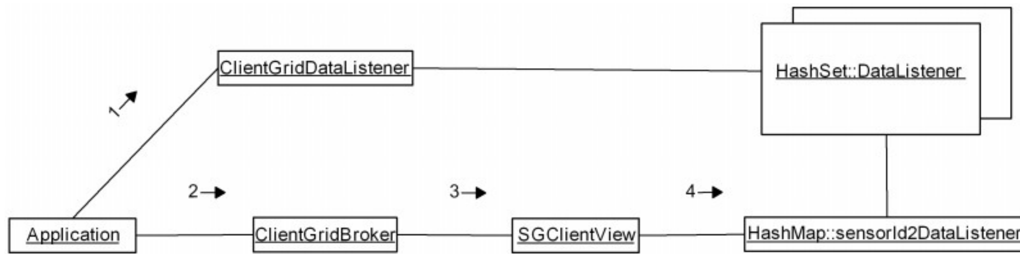


Figure 3-32 Message flow from a sensor grid to a subscribing application

1 After application client knows the presence of a sensor, it creates an instance of ClientGridDataListener (Sensor Data Listener) for the sensor
2 call subscribeSensorData() and provides the sensor id and ClientGridDataListener as parameter
3 Forwards the call to SGClientView
4 Register the ClientGridDataListener so that when sensor data arrives the listener will be notified. If this is the first request of subscribing data from this sensor, subscribes to the NB stream unique to the sensor

1.4.4.7 Setting a Filter

The design of SCMW suppots filtering of sensor streams in a sensor grid to facilitate construction of UDOP for situational awareness. The message flow of an application setting up a filter query is shown in

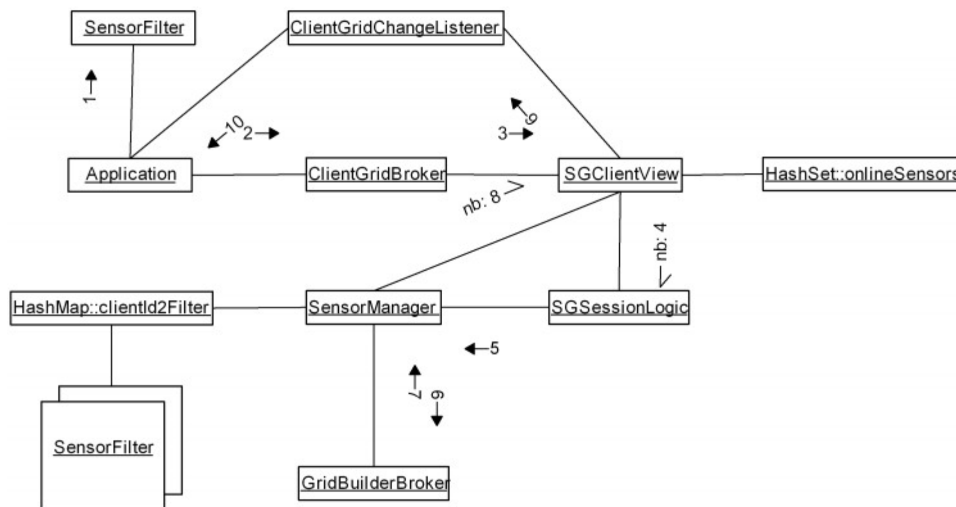


Figure 3-33 Message flow of filter setup in a sensor grid

1 Application client instantiates a SensorFilter object according to application-specific filter criteria

2 initiates a setFilter() request to ClientGridBroker, using the SensorFilter as parameter
3 Forwards the request to SGClientView
4 Sends a FILTER_MSG message to Sensor Grid through NB, together with the SensorFilter object
5 Pass the SensorFilter object to SensorManager
6 Send a request to Grid Builder acquiring a list of sensors which matches the filtering criteria defined by the Filter
7 GridBuilderBroker returns a list of sensors fulfilling the criteria
8 Compare the list of returned sensors with the currently cached list of sensors for the application-client. Notifies the application-client all changes by sending a SENSOR_CHANGE message through a application-client specific NB stream
9 Updates the cached list of online sensors in HashSet. Invokes handleSensorChange() of the registered ClientGridChangeListener (Sensor Change Listener)
10 ClientGridChangeListener notifies application client of sensor change. Application client performs corresponding actions

1.4.4.8 Sending Control to a Sensor

Some sensors do not only send live streams to a sensor grid. They could receive control information from users or applications and respond with sensor information that corresponds to received control information. The message flow of an application sending a control message to a sensor is illustrated in Figure 3-34.

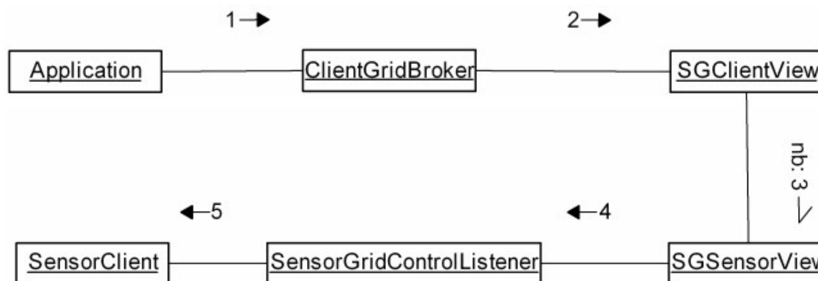


Figure 3-34 Message flow of control messages from applications to sensors in a sensor grid

1 Application client invokes sendControl() of ClientGridBroker with the specified sensor id and control message recognizable by the sensor
2 Forwards the request to SGClientView
3 Sends the SENSOR_CONTROL to the sensor through a unique NB stream for the sensor
4 Forwards the control message to the registered SensorGridControlListener by handleSensorControl()
5 Notifies SensorClient that a control message is received. The sensor client performs the corresponding actions

1.4.4.9 Disconnecting a Sensor

To disconnect a sensor, one of the ways is to stop the sensor client program through GB's management console. The diagram below shows the message flow of disconnecting a sensor this way.

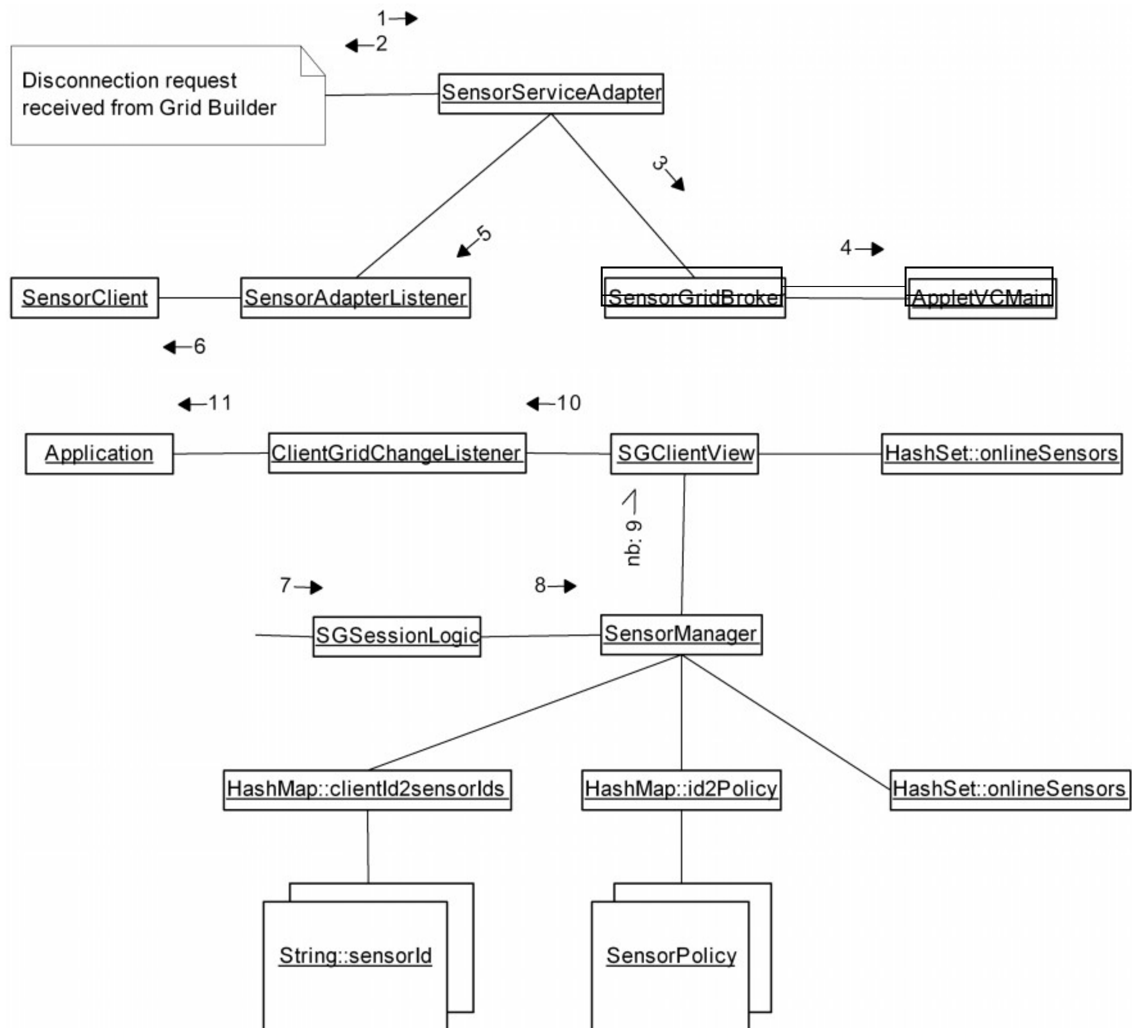


Figure 3-35 Message flow when disconnecting a deployed sensor from a sensor grid

1 A disconnection request is received from Grid Builder (please refer to session 3.3.5.2 for details). processWxfDelete() of SensorServiceAdapter is invoked
2 Reports the running status of the associated sensor client program by sending a Wxf_DeleteResponse message to SensorServiceAdapter. If the sensor client program is running, go to 3. Otherwise, does nothing and exits
3 Invokes close() of SensorGridBroker
4 Notifies the framework to dispose resource allocated to the sensor by calling allWindowsClosed() of AppletVCMain
5 Notifies the associated SensorAdapterListener to terminate the sensor client program by calling handleSensorStopRequest()
6 SensorClient disconnect all connections and exits
7 The framework notifies SGSessionLogic that the sensor has disconnected by invoking userLeft()

8 invokes removeSensor() of SensorManager
9 Removes the cached SensorPolicy and status for this sensor. For each application client, removes the sensor from the cached list of sensors associated with it, then notifies the application client by sending a SENSOR_CHANGE message through the unique NB stream for the client
10 Updates the cached list of online sensors in HashSet. Invokes handleSensorChange() of the registered ClientGridChangeListener (Sensor Change Listener)
11 ClientGridChangeListener notifies application client of sensor change. Application client performs corresponding actions

1.5 SCMW Application Program Interface (API) and Sensor Service Abstraction Layer (SSAL)

1.5.1Overview of the SCMW API

The SCCGMMS Application Program Interface (API) allows any third party application to connect and utilize functions provided by SCMW. An application can do the following through the SCMW API:

1. Obtains the policies and data of all sensors which are currently up and running
2. Selectively subscribes to sensors with their policies fulfilling filtering criteria defined by the application
3. Sends control messages to sensors
4. Dynamically notified for new sensors which fulfill the filtering criteria, and for sensors which have been disconnected

To use the SCMW API, an application has to instantiates an Application Client Broker (ClientGridBroker) and implements the Sensor Change Listener (ClientGridChangeListener) interface. Moreover, a Sensor Data Listener (ClientGridDataListener) has to be created for subscribing to data stream of each sensor.

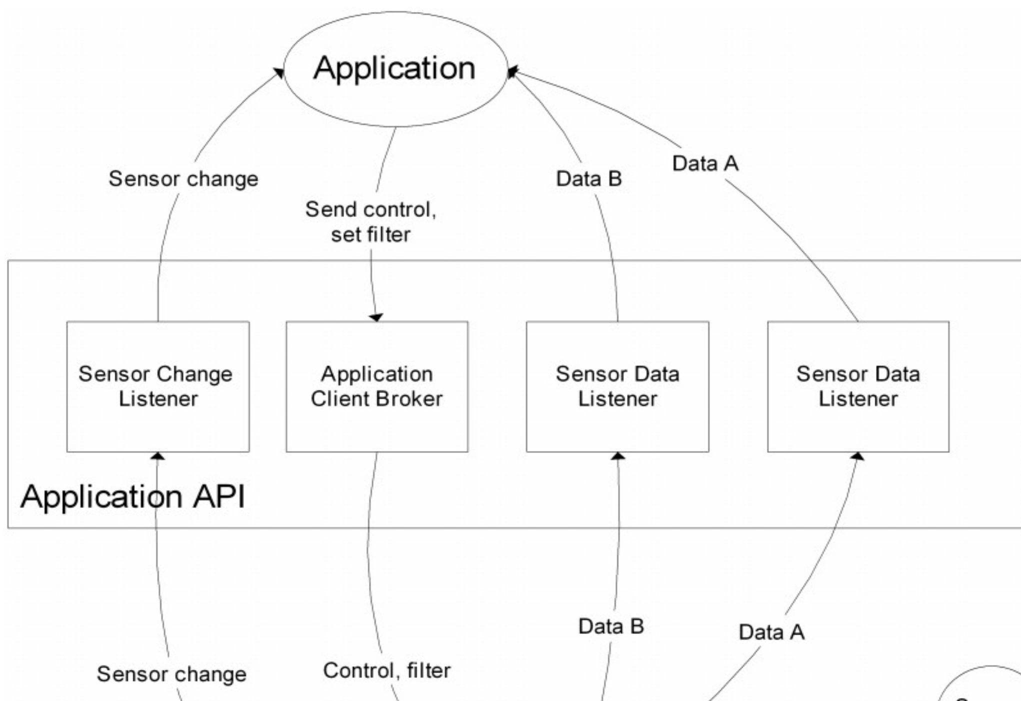


Figure 3-36 SCMW Application Programming Interface

Sensor Service Abstraction Layer (SSAL)

1.5.2 Overall Sensor Service Abstraction Layer Architecture

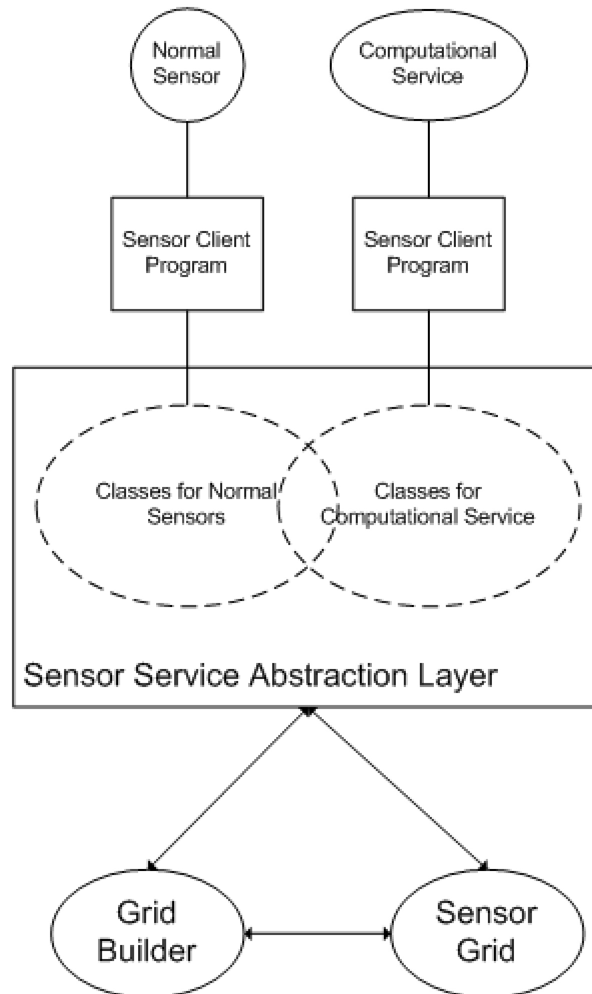


Figure 3-37 A high-level architecture of the Sensor Service Abstractioon Layer (SSAL)

Sensor Service Abstraction Layer (SSAL) provides a common interface for all kinds of sensors. Sensor developers add new sensors to SCMW by writing **Sensor Client Programs (SCP)** which connects to SCMW through libraries in SSAL.

Internally, SSAL communicates with GB for sensor management (e.g. creation, registration, definition) and SG for run-time management (e.g. data publishing, receiving control messages).

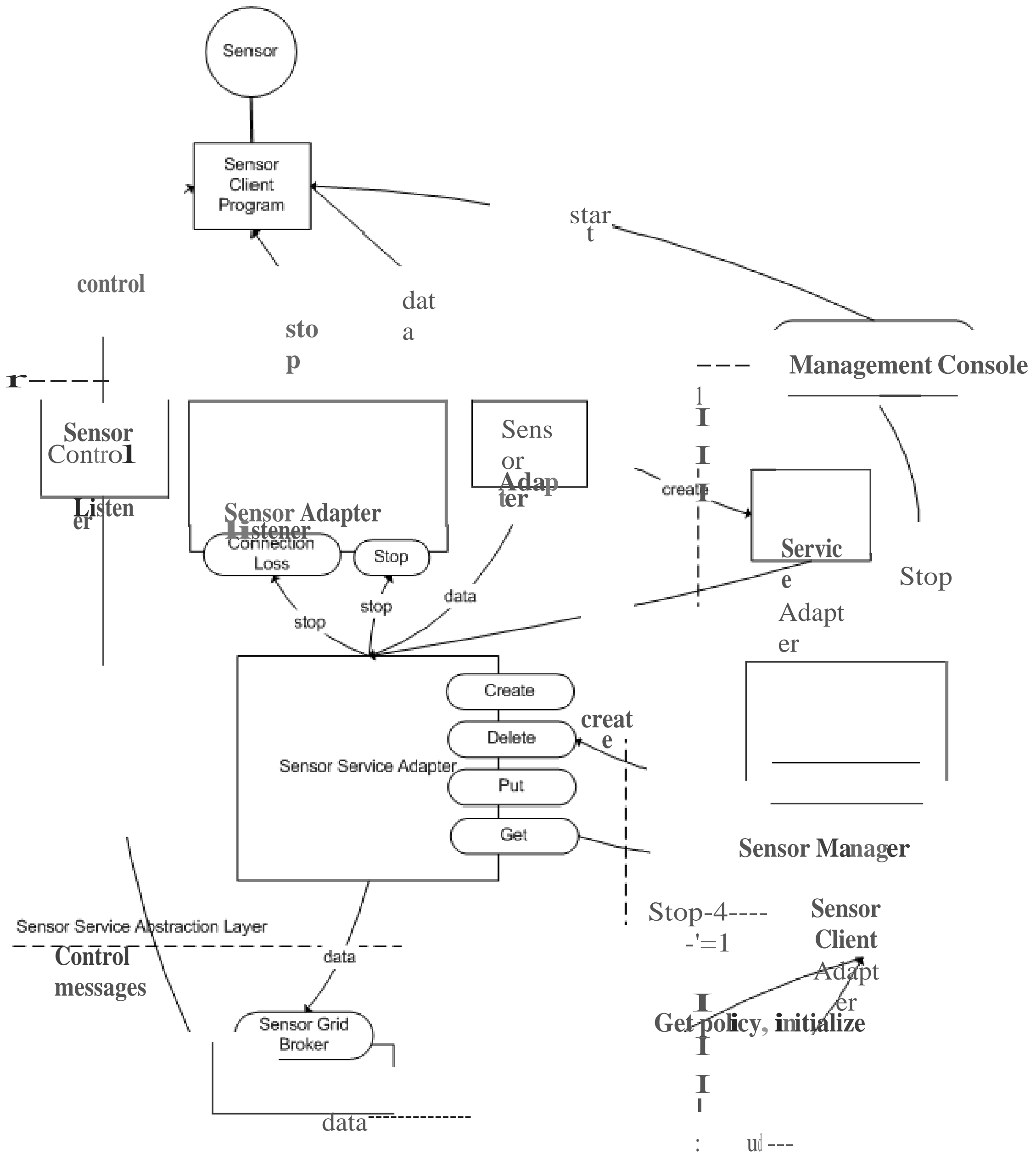
In SSAL, sensors are categorized into two categories:

Normal Sensors – Sensors which take input from external environment. The input data is external to SCMW.

Computational Service – Sensors which do not take input from the environment. Instead, they take output of other sensors as input, perform various computations on the data, and output the processed data finally

Functionalities of the two different categories of sensors are supported by two different sets of classes in SSAL. Some classes are shared between the two categories for common functionalities.

1.5.3 SSAL Architecture for General Sensor Services



sxo

Filter result sensor policies

Send filter

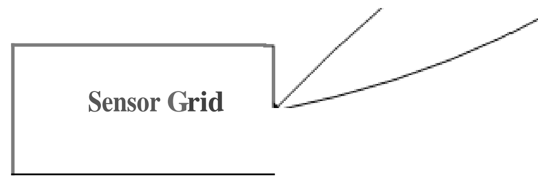


Figure 3-38 A detailed SSAL architecture for gmeral sensor sercvices

Figme 3-38 shows the architectme of SSAL for general sensors to be wrapped and deployed as sensor services. The following subsections explain the message flow for some basic operations.

1.5.3.1 Sensor Deployment

To deploy a sensor, the corresponding SCP has to instantiate a Sensor Adapter which notifies SCMW for its presence and data publishing. It also has to implement a Sensor Control Listener (for receiving control messages) and a Sensor Adapter Listener (for actions such as terminating SCP). The SCP can either be started by a way decided by the sensor developer (e.g. run a .bat script), or it can be embedded in SCMW so that it can be started by GB's Management Console. For a more detailed message flow, please refer to section 3.3.5 .

1.5.3.2 Data Publishing

SCP is responsible for collecting data from the sensor, and then publishes it through Sensor Adapter. Sensor Adapter in turn forwards the data to the corresponding Sensor Service Adapter, and finally to all applications that have subscribed to its data through SXO. For a more detailed message flow, please refer to section 3.4.4.5 .

1.5.3.3 Performing Actions on Sensor Client Program

Sometimes the user may want to perform some actions remotely on the SCP, such as pausing or terminating the SCP. SCP listens for these actions through Sensor Adapter Listener. Currently, there is only one action supported by SCMW – terminating the SCP.

1.5.4 SSAL Architecture for Computation as a Sensor Service

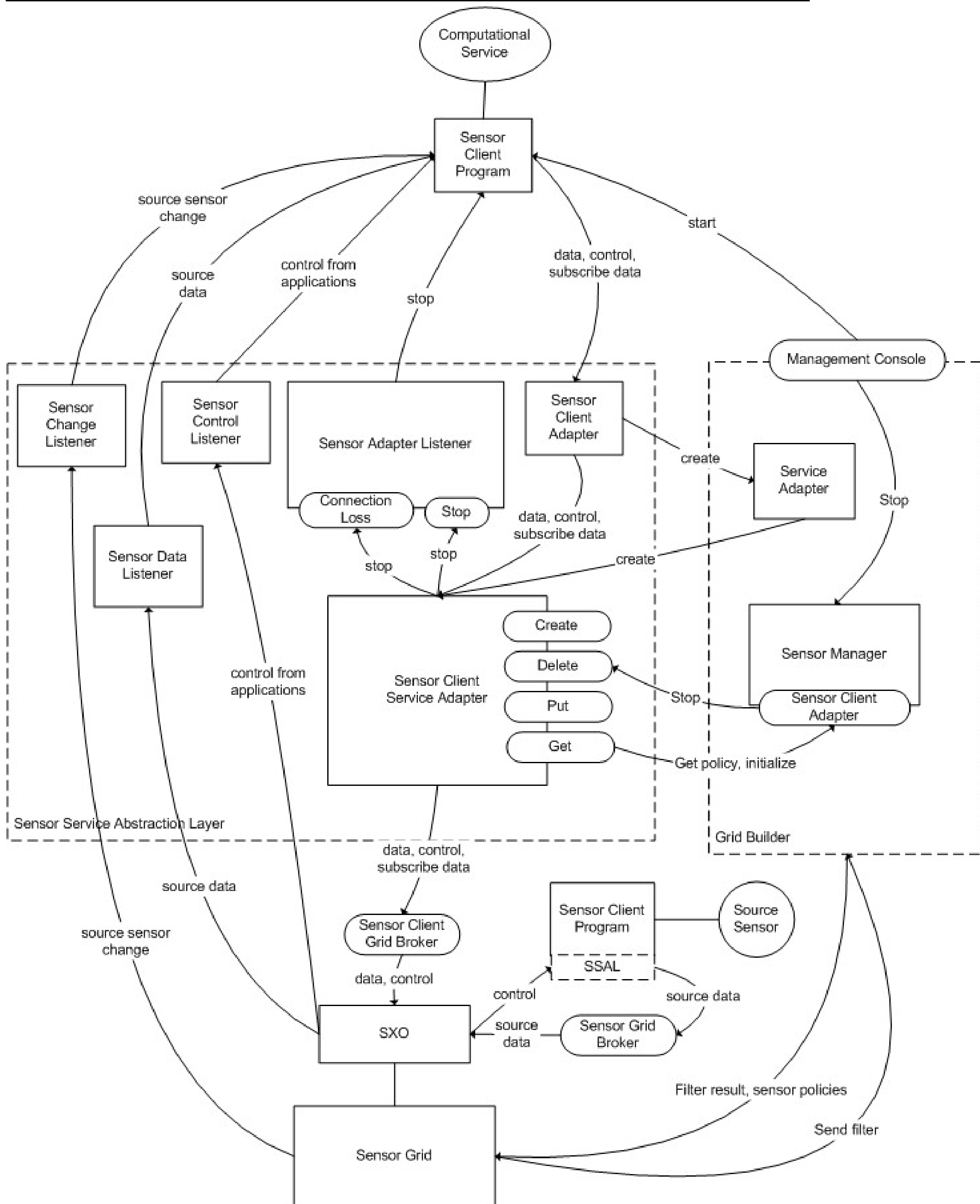


Figure 3-39 A detailed SSAL architecture for computation as a sensor service

Architecturally SSAL for Computational Service combines SSAL for normal sensors and SCMW API since it needs functionalities from both sides. Figure 3-39 shows SSAL for Computational Service. You can observe that components of the SCMW API are integrated with components of the original SSAL and some new modules to form the

SSAL for Computation as a Sensor Service. The extension of SSAL to cover computation as a sensor service significantly broadens the applicability of the Sensor- Centric Grid of Grids and eases the integration of new or legacy system of systems with sensor-centric applications.

The following subsections explain the message flow for some operations of Computational Services.

1.5.4.1 Sensor Deployment

To deploy a Computational Service, the corresponding SCP has to instantiate a Sensor Client Adapter which notifies SCMW for its presence and for various sensor related operations such as data publishing, subscribing data from source sensors and sending control messages to source sensors. It also has to implement a Sensor Control Listener (for receiving control messages) and a Sensor Adapter Listener (for actions such as terminating SCP) as what normal sensors do.

1.5.4.2 Subscribe Sensor Data

Since Computational Services take input from other sensors (source sensor), they have to subscribe data from other sensors in a similar way to applications. To subscribe data, the SCP of a Computational Service has to invoke functions of Sensor Client Adapter which in turn setup the connections through SXO. SCP has to implement the Sensor Change Listener and Sensor Data Listener interfaces. Whenever the state of source sensor changes (e.g. online to offline) the SCP will be notified through Sensor Change Listener. Similarly SCP will be notified for data arrival through Sensor Data Listener.

1.6 Container Service

Sensor Container (Manager & Services)

Motivation

Prior to the implementation of Sensor Container Manager & Services, every Sensor invoked used to live in its own JVM, hence there by consuming a lot of memory due to the overhead of individual 'Run Times', 'Garbage Collectors' etc. Due to limitation of system resources in a given Domain, the total number of Sensors that could be hosted/supported in a given Leaf Domain was limited by the degree of available system resources and not by the capacity of the underlined Broker. This was leading to Broker starvation, the Sensor Container Manager & Services implementation work towards eliminating this issue.

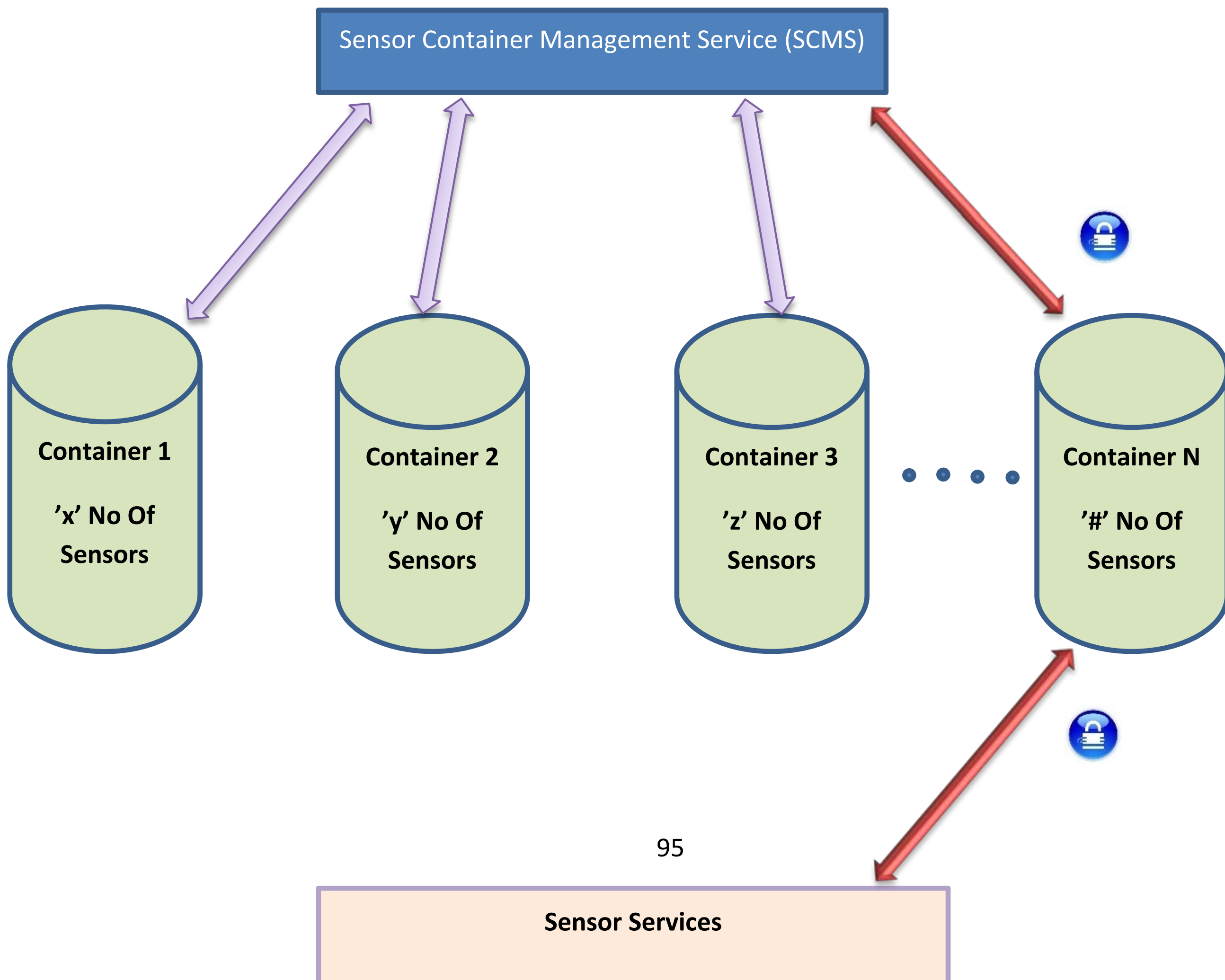
Implementation & Working

To support a huge number of Sensors in a single domain we looked forward towards possible use of Inter Process Communication (IPC) and hence thereby we came up with the Container managed services. Where a container provides space for multiple sensors to live and service within a single JVM process and thereby sharing a single JVM resources. This results in huge decrease in the consumption of system resources. The inner working of Sensor Container (Manager & Services) are mentioned below:

- Sensor Container Management Service (SCMS) is invoked using a script.
- SCMS brings up the first Container and provides the service Lock to the same.
- Once the container holding up the Service Lock is filled out, the same container shuts down its external services and releases the Lock.
- SCMS at that instance brings up another Container and provides the new container with the service lock.
- At any given instance if a few sensors have been shut down from a given container, the container registers to obtain the Service Lock and the same is handed over to the requesting Container by the SCMS once the current container holding the Service Lock is filled up and releases the Lock.

- SCMS also helps in cleaning up empty containers.

Architecture Diagram:

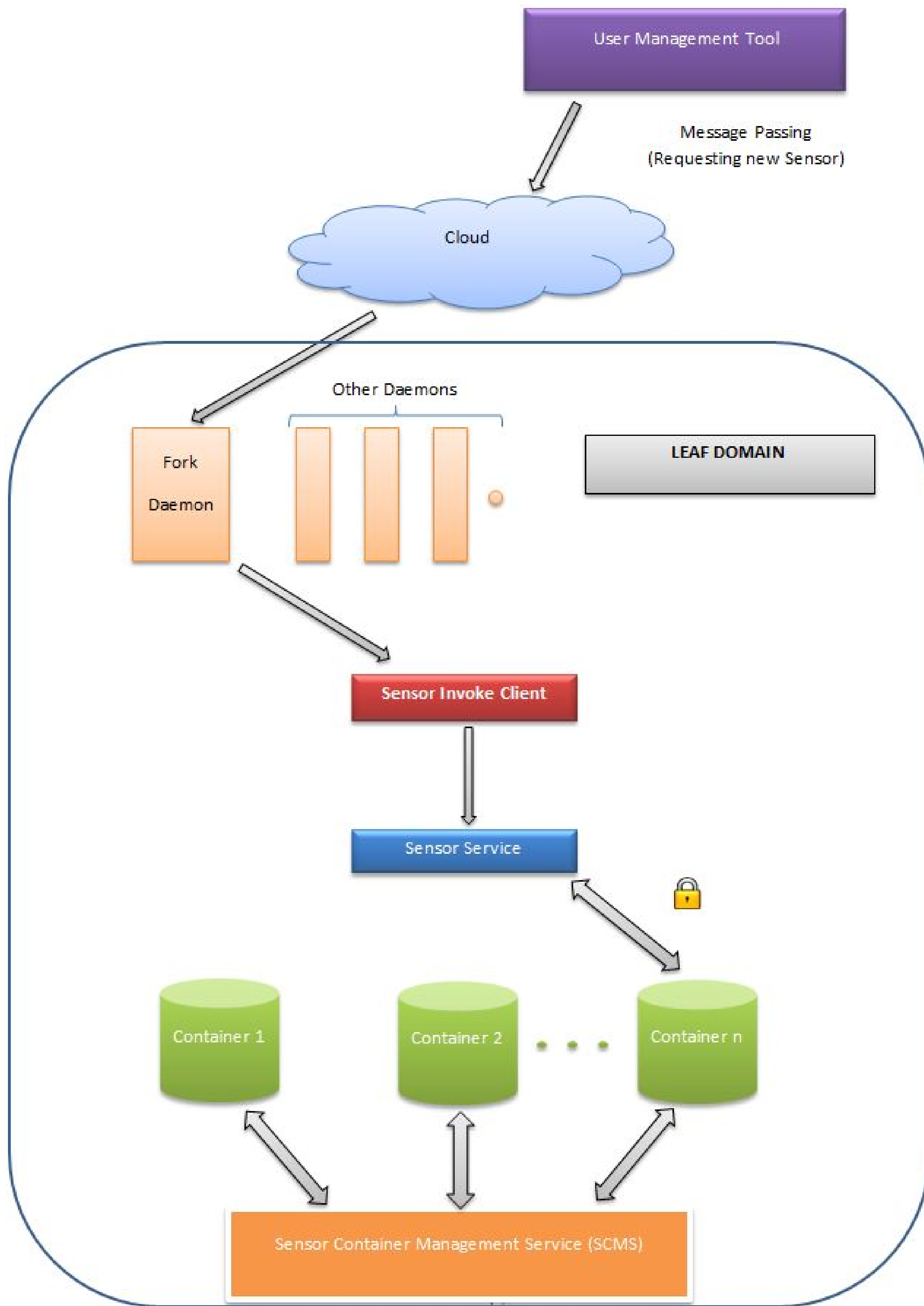


Architecture of Container Service

A sensor invoke request is generated either directly by a script or by the User Management tool via ForkDeamon for a given Domain. In either of the case it forks a new process containing the Sensor Invoke Client code. The Sensor Invoke Client repeatedly sends a request to the Sensor Service hosted by the Locked in Container until it receives a success response.

Once the Sensor Service receives the request it brings up a Sensor of a specific type requested with in the container which holds the lock at that instance. Hence there by multiple sensors could coexist with in a single Container.

Flow Diagram


User Management Tool
Message Passing
(Requesting new Sensor)
Cloud
Other Daemons
Fork
Daemon
LEAF DOMAIN
Sensor Invoke Client
Sensor Service
Container 1
Container 2
Container n
Sensor Container Management Service (SCMS)

2. USER GUIDES

2.6 Sensor Cloud components

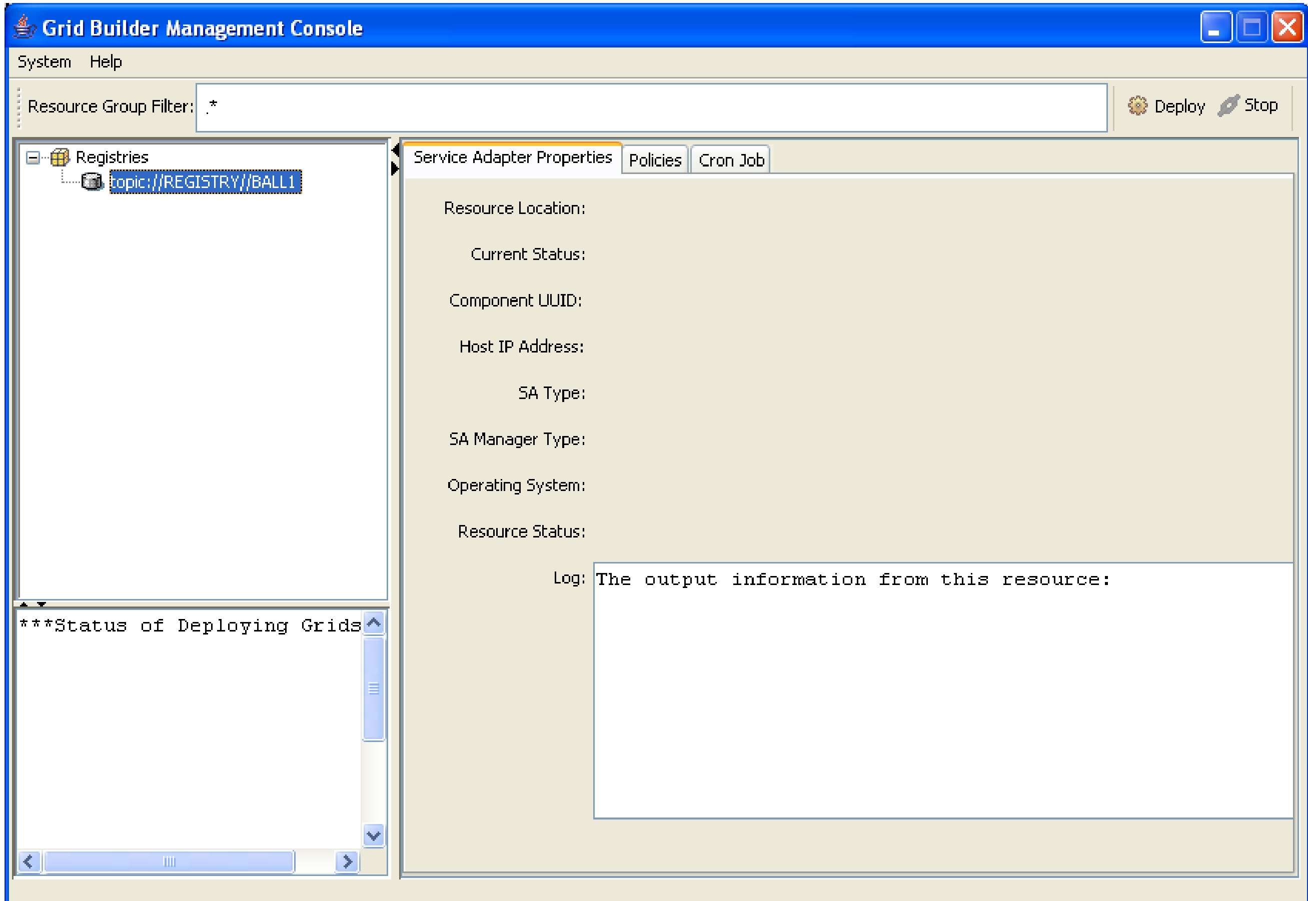
The main components of the Sensor Cloud project are:

1. **sensorcloud-middleware**: This is the core component which consists of the middleware code of the Sensor Cloud Framework which wraps around the original Narada Broker pub-sub architecture.
2. **sensorcloud-sensors**: The sensors can publish sensor data to the middleware. This maven module composes of the set of defined sensors and the users can add their custom sensors to this module.
3. **sensorcloud-clients**: The clients subscribe to the sensor data from the middleware published by the sensors. This maven module composes of code to display the data subscribed to the user.
4. **sensorcloud-managementui**: The Management User Interface is a swing application which is used to manage all the Grid Builder domains and sensors invoked.
5. **sensorcloud-managementwebui**: This is a Web Management interface developed with Google Web Toolkit which performs the same function as management ui.
6. **sensorcloud-restservlet**: The module makes use of JBoss rest easy technology to get and put sensor data at the web browser from the SGX.
7. **sensorcloud-streaming**: The module is used to make the streaming of sensor data from the SGX to the web browsers.
8. **sensorcloud-distribution**: The module distributes the jar packages of all the Maven modules to the Maven repository.

Sensor Cloud Framework can be started on the local machine by executing the following steps:

Launch Grid Builder

- Go to %SENSORCLOUD_HOME%\scripts
- Execute startLocal.bat by double-clicking it
- Several Console windows will startup
- Wait until the Java Grid Builder Management Console pops up (shown in the next figure)



Video Reference: http://screenr.com/DjTs

Once you have launched Grid Builder, you can use either or both of the following programs to interact with sensors that you chose to deploy with Grid Builder.

(1) Meeting Console Client

- Go to %SENSORCLOUD_HOME% \scripts\clients
- Execute prepareConsoleClient.bat by double clicking it
- Meeting Console Client program displays the sensors those are active in the Sensor Grid Framework and the messages exchanged between the sensor and the Sensor Cloud

(2) Swing Client

- Open Command Prompt
- Go to %SENSORCLOUD_HOME%\scripts\clients
- Execute prepareSwingClientLocal.bat
- Sensor Grid demo is a graphical display for the sensors that are active in the Sensor Grid framework.

The Sensor Grid processes can be shut down by executing the following commands in the command prompt

- Run taskkil.exe /f /IM *.java
- Run taskkill.exe /f /IM cmd.exe

OR by executing the script killer.bat at %SENSORCLOUD_HOME%\scripts\

2.7 Deployment and User Guides

2.7.1 Sensor Deployment Tutorial

We have developed a simple GPS Sensor for illustrative purposes which can display the GPS co-ordinates on the client console. In the next section of this guide, we shall describe how to deploy and use this sensor.

Tutorial: GPS Sensor

Go to %SENSORCLOUD_HOME%\scripts and execute the batch file GettingStartedDemo.bat. The contents of this batch file would look like:

```
ECHO Executing ...

cd GridBuilder

start runGPS "IU Innovation Center" GPS "2719 East 10th Street" Bloomington IN US yes yes
yes yes yes virtual_3910.38746,N_08630.12594,W

start runGPS "Ball" GPS "2875 Presidential Drive" Fairborn OH US yes yes yes yes yes
virtual_3946.6524,N_08403.56274,W
```

This batch file will deploy two instances of the GPS sensor with virtual geo co-ordinates which are given as the last two parameters for this sensor.

Viewing the output:

The sensor data generated by this sensor can be viewed either of the two ways mentioned below:

1. Console Client: Go to %SENSORCLOUD_HOME%\scripts\clients and execute prepareConsoleClient.bat which will open the client console in a command window where the GPS data received from the Sensor Grid can be viewed.

2. Sensor Grid Demo: Go to %SENSORCLOUD_HOME%\scripts\clients and execute prepareSwingClientLocal.bat which will open a graphical interface where the GPS data received from the Sensor Grid can be viewed in a graphical format.

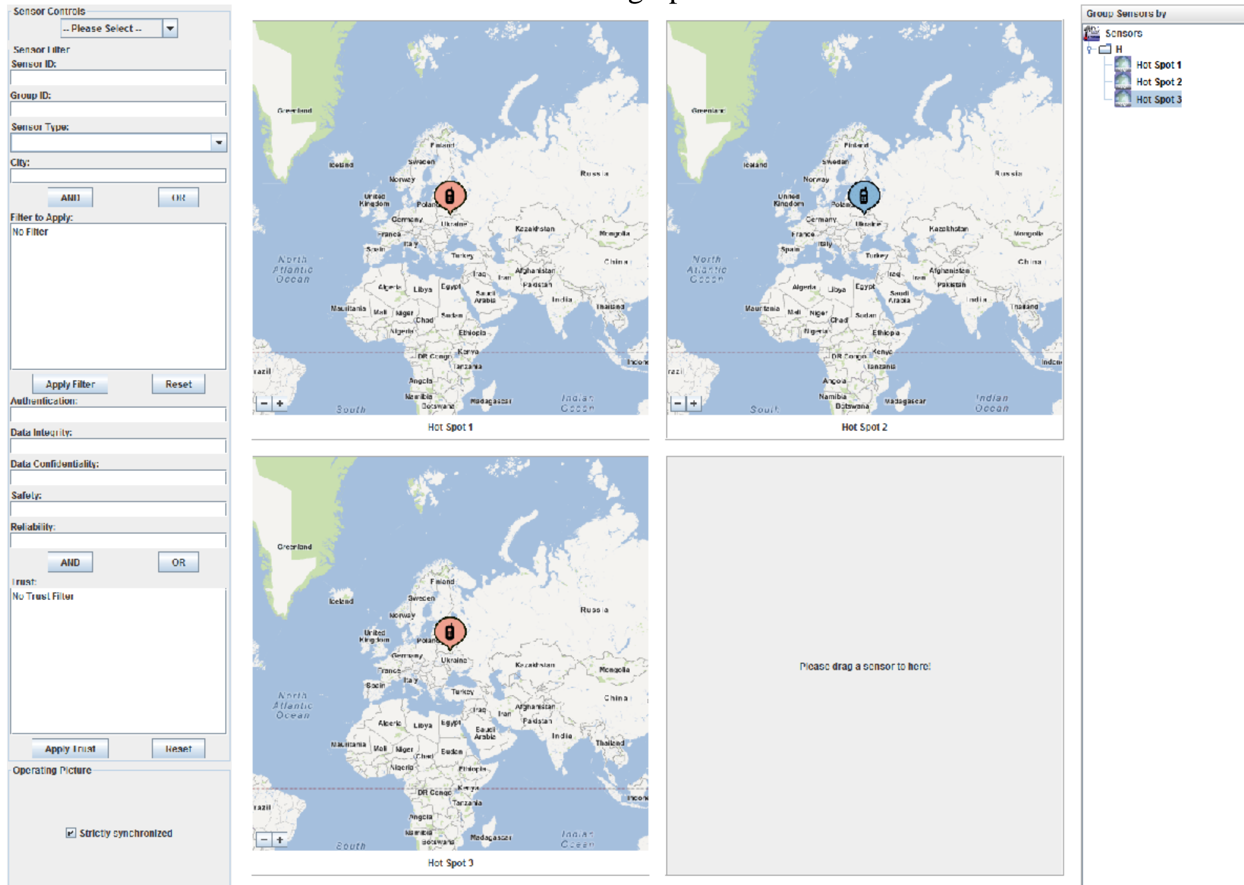


Fig: View Using Swing Client

Video Reference: http://screenr.com/bjTs

2.2.2 User Guide for Sensors

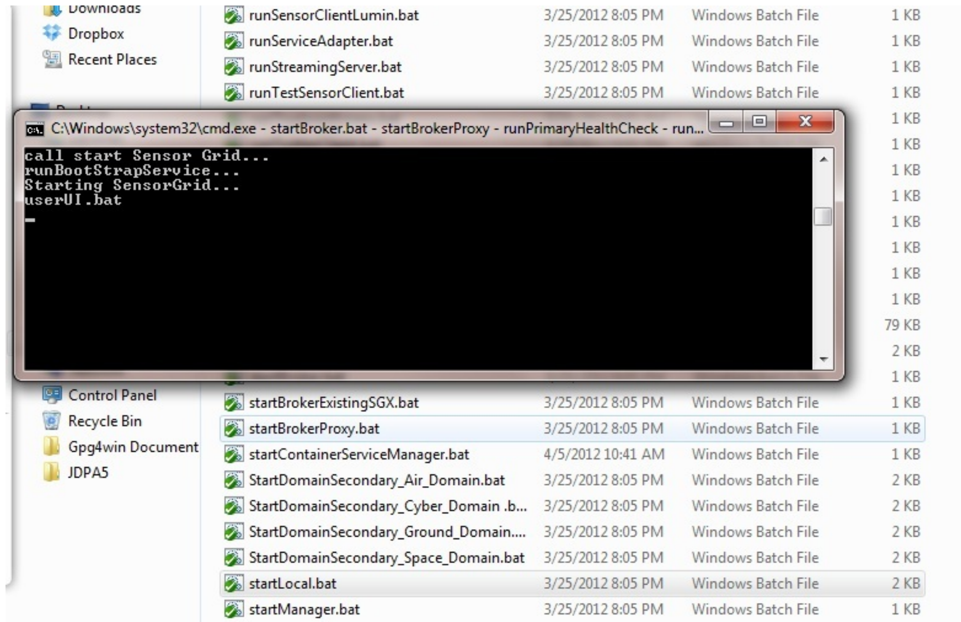
Introduction

The CloudSensor can be used to deploy various kinds of sensors. This guide states the various steps involved in deploying various sensors. Each sensor can be deployed in 2 ways. One way to deploy is using the swing based GUI client and the other way to deploy them using batch scripts.

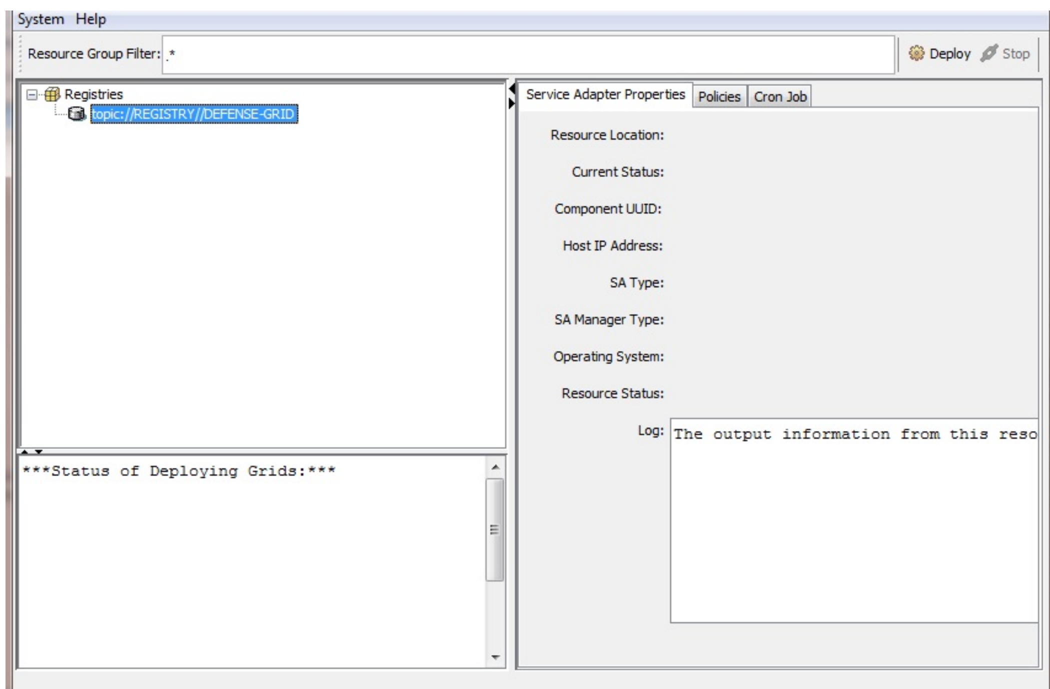
1. **<u>Twitter Sensor</u>**

GUI client Steps :

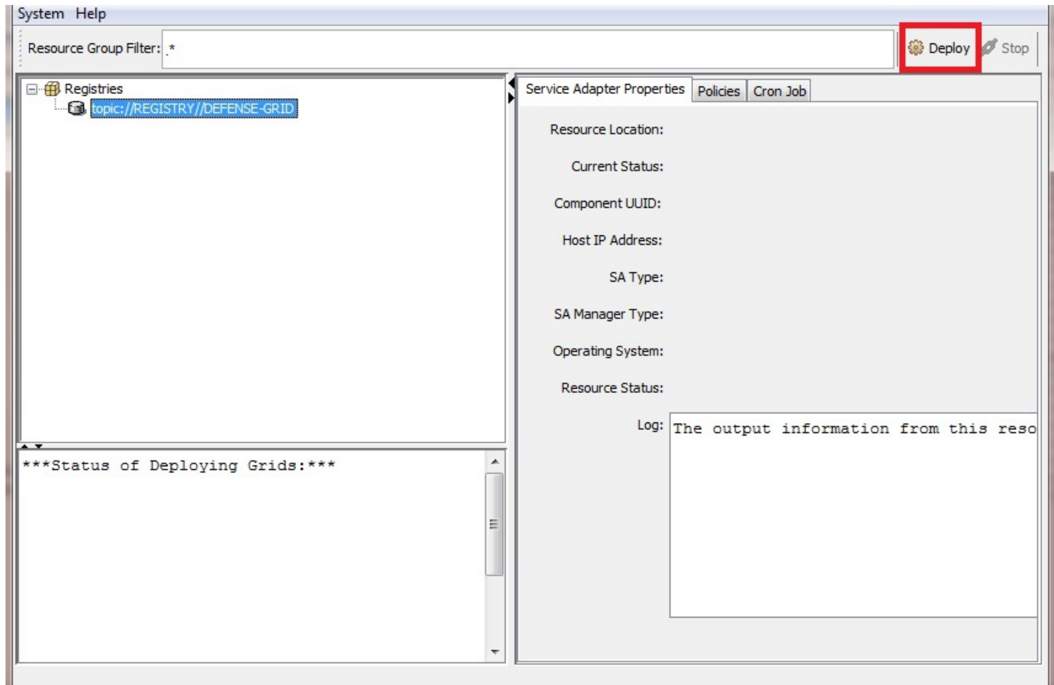
a) Run the startLocal.bat file to start the GUI interface for deploying twitter sensor.



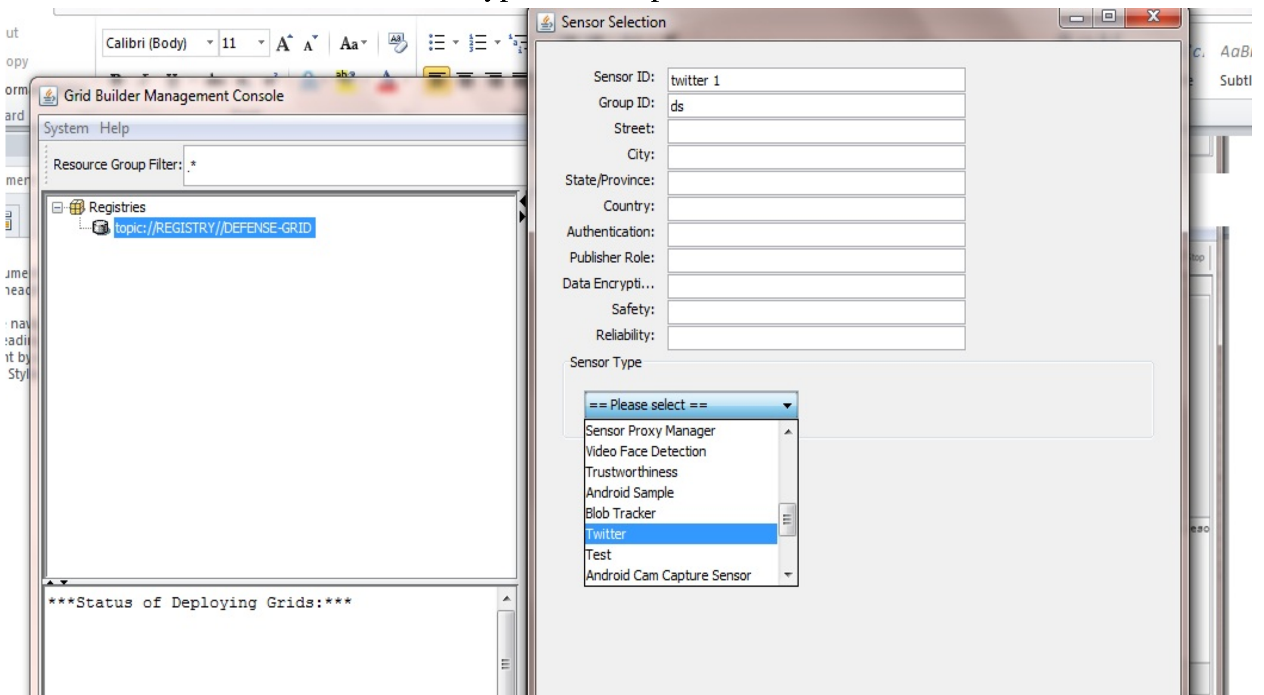
b) The startLocal script opens up the GUI



c) Once the GUI opens, click on the Deploy

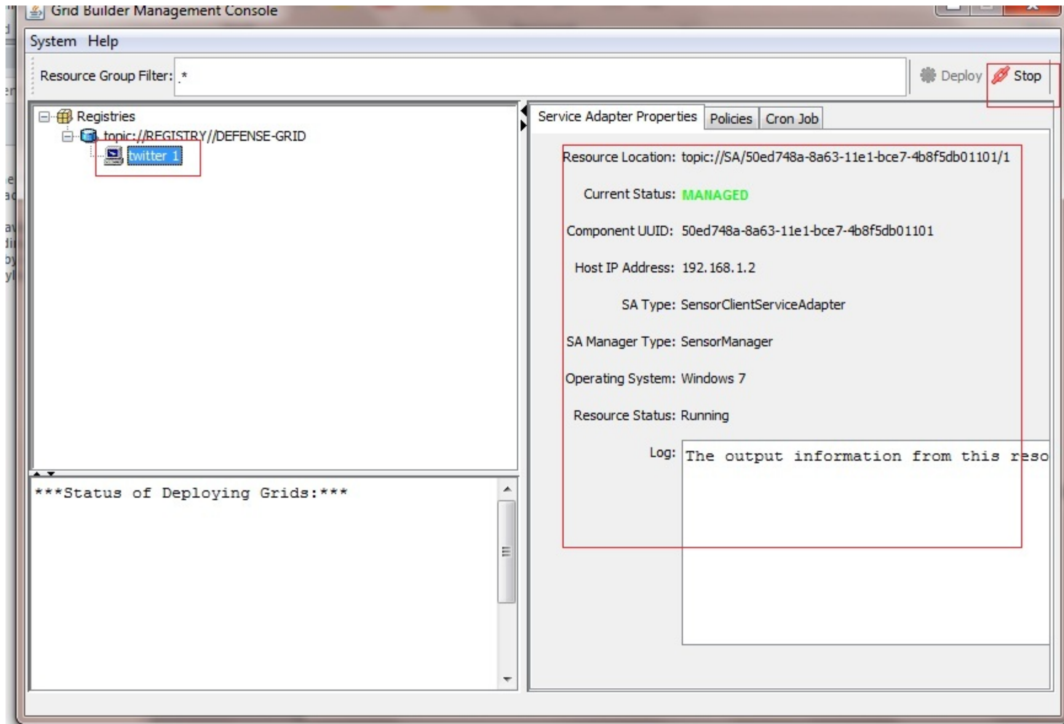


d) Clicking on the deploy button, opens a new window for sensor selection. Enter the relevant data for twitter sensor and select the type from dropdown selection box.



e) After filling the required details, click on OK button

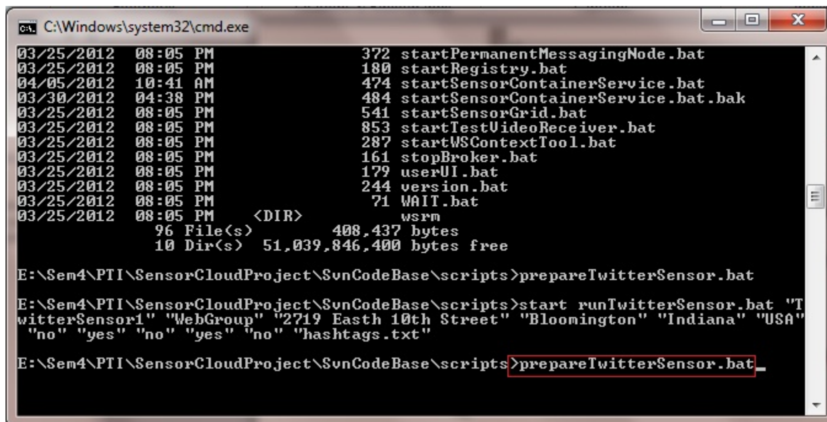
f) Then the twitter sensor gets deployed and we can see it in the GUI as shown in following screenshot.



g) The sensor can be stopped by clicking on Stop button as shown above.

Batch Script Steps:

a) Go to command prompt and run the "prepareTwitterSensor" batch script as shown



b) Then once the Twitter sensor gets deployed successfully, you can see a new command prompt opening up as shown in screenshot.

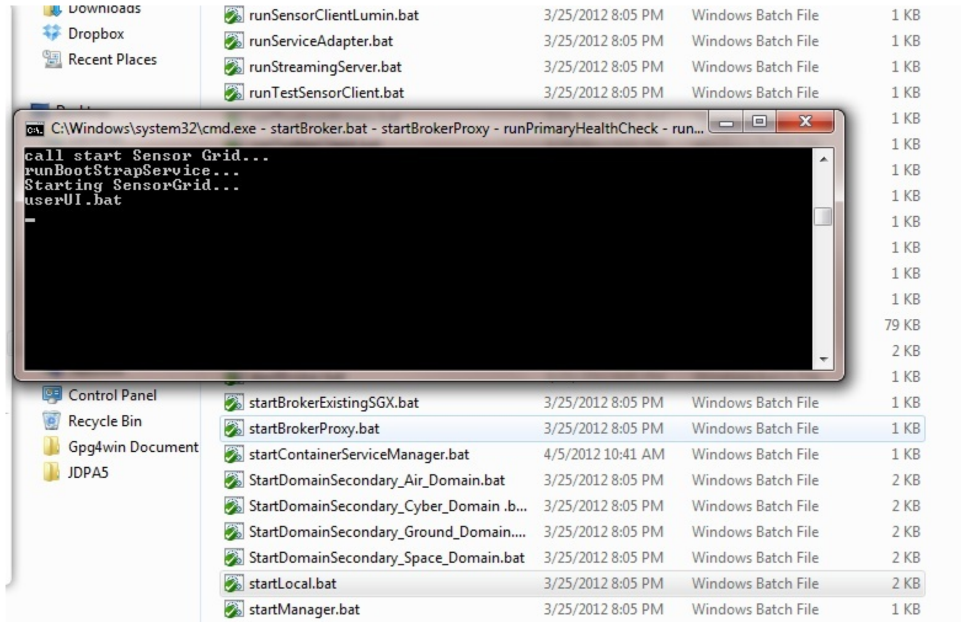
```
C:\Windows\system32\cmd.exe
               10 Dir(s)  51,039,846,400 bytes free

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>prepareTwitterSensor.bat

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>start runTwitterSensor.bat "T
witterSensor1" "WebGroup" "2719 Easth 10th Street" "Bloomington" "Indiana" "USA"
 "no" "yes" "no" "yes" "no" "hashtags.txt"
```

```
C:\Windows\system32\cmd.exe - runTwitterSensor.bat  "TwitterSensor1" "WebGroup" "2719 Easth...
[2012-04-20 12:08:45,624]<DEBUG>[ServiceAdapter] - Registry Register/Renewal Suc
cessful !
[2012-04-20 12:08:45,625]<DEBUG>[ServiceAdapter] - My current location is:topic:
//SA/43290c86-8b00-11e1-957d-49a9cccd3c4a/1
[2012-04-20 12:08:48,299]<DEBUG>[SGLogic] - received message type: alive
[2012-04-20 12:08:48,301]<DEBUG>[SGLogic] - alive 1334938128301
[2012-04-20 12:08:48,301]<DEBUG>[SGSensorView] - received message type: alive
[2012-04-20 12:08:58,303]<DEBUG>[SGLogic] - received message type: alive
[2012-04-20 12:08:58,304]<DEBUG>[SGLogic] - alive 1334938138304
[2012-04-20 12:08:58,304]<DEBUG>[SGSensorView] - received message type: alive
[2012-04-20 12:09:04,869]<DEBUG>[TwitterSensor] - Entering getFeeds for search p
attern null
[2012-04-20 12:09:08,304]<DEBUG>[SGLogic] - received message type: alive
[2012-04-20 12:09:08,309]<DEBUG>[SGLogic] - alive 1334938148309
[2012-04-20 12:09:08,310]<DEBUG>[SGSensorView] - received message type: alive
[2012-04-20 12:09:15,632]<DEBUG>[ServiceAdapter] - Registry Register/Renewal Suc
cessful !
[2012-04-20 12:09:15,633]<DEBUG>[ServiceAdapter] - My current location is:topic:
//SA/43290c86-8b00-11e1-957d-49a9cccd3c4a/1
[2012-04-20 12:09:18,304]<DEBUG>[SGLogic] - received message type: alive
[2012-04-20 12:09:18,305]<DEBUG>[SGLogic] - alive 1334938158305
[2012-04-20 12:09:18,306]<DEBUG>[SGSensorView] - received message type: alive
[2012-04-20 12:09:24,871]<DEBUG>[TwitterSensor] - Entering getFeeds for search p
attern null
```

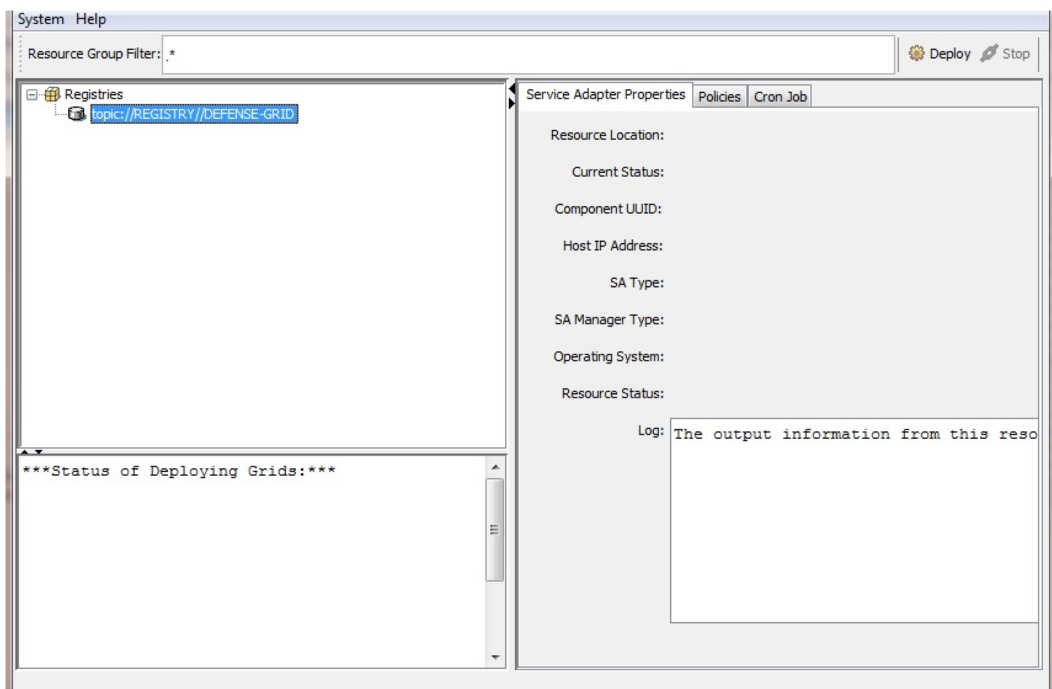
2. **Chat Sensor**

GUI Client Steps:

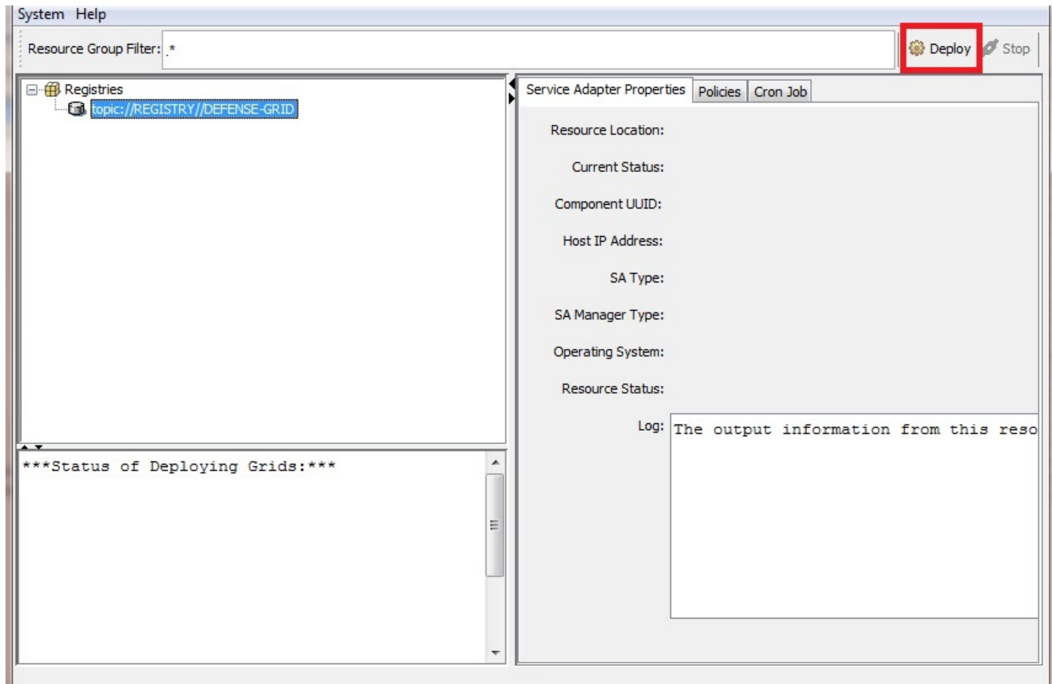
a) Run the startLocal.bat file to start the GUI interface for deploying chat sensor.



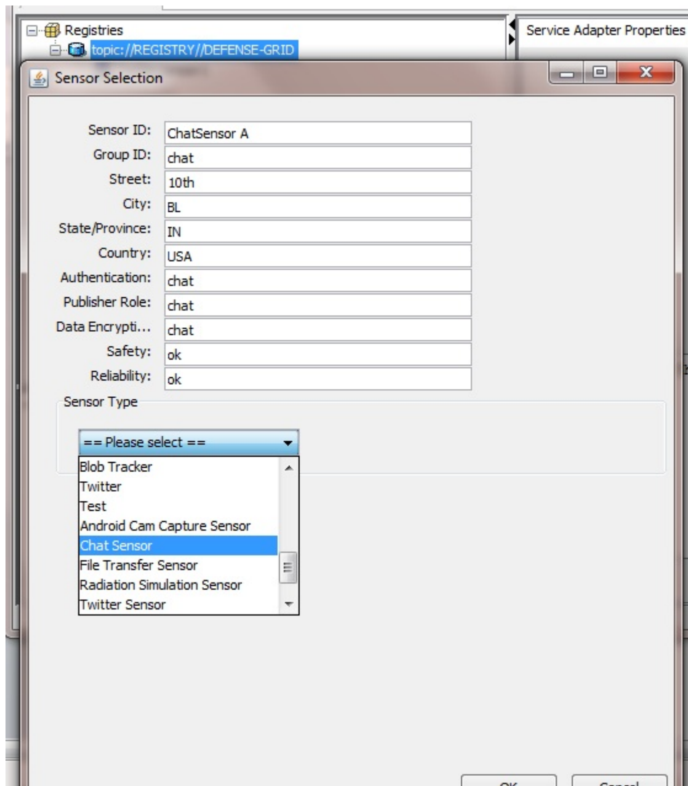
b) The startLocal script opens up the GUI



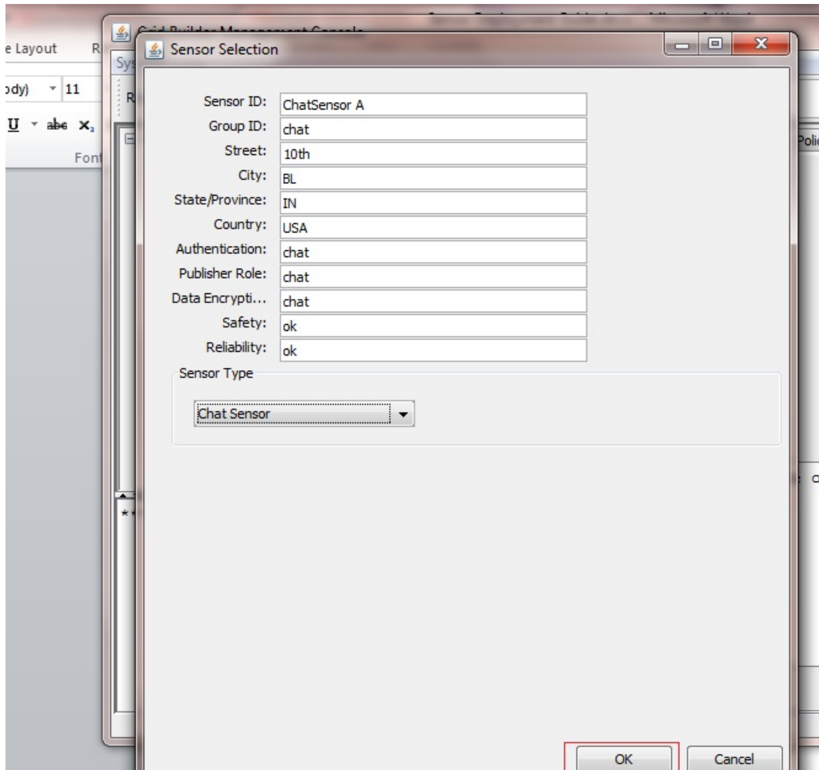
c) Once the GUI opens, click on the Deploy



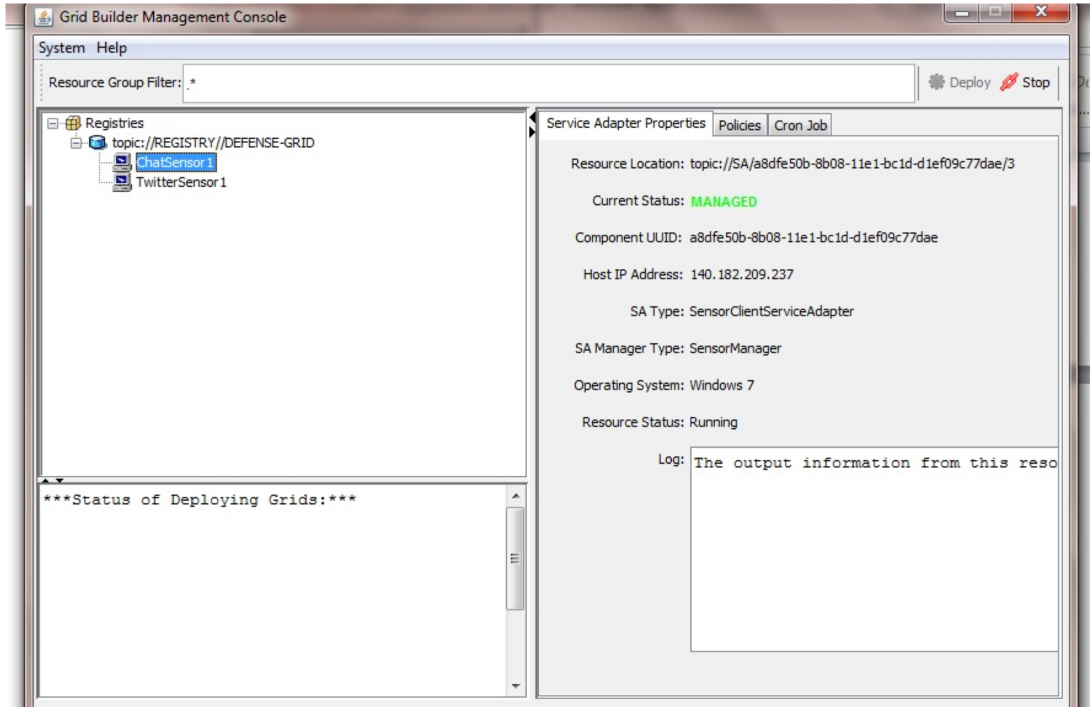
d) Clicking on the deploy button, opens a new window for sensor selection. Enter the relevant data for chat sensor and select the type from dropdown selection box.

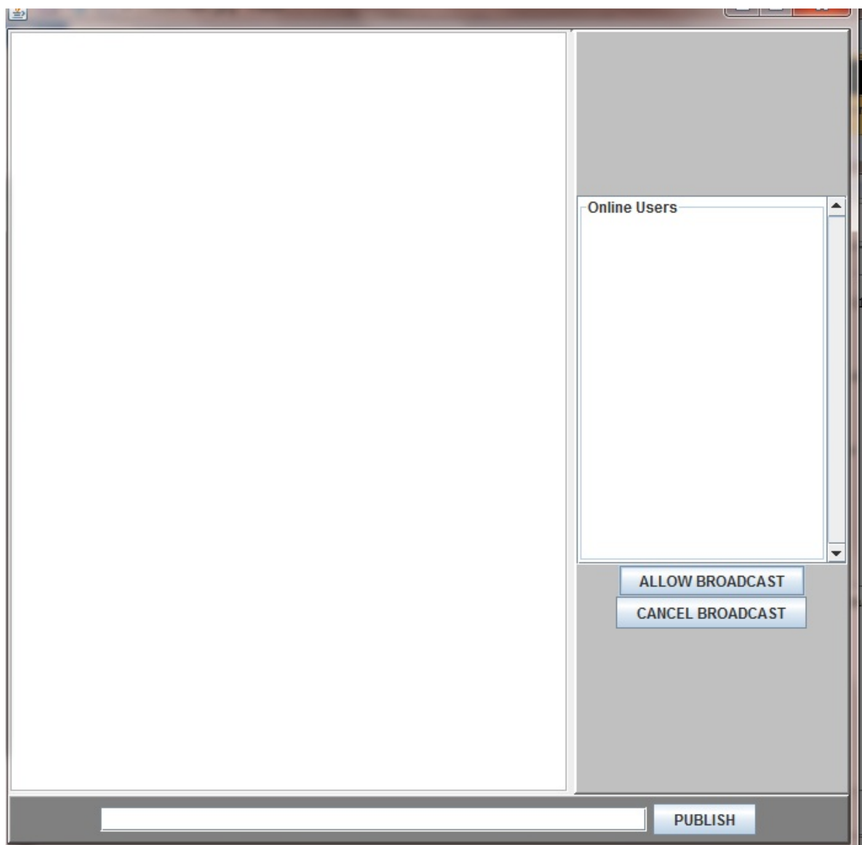


e) After filling the details, click on OK button.



f) Then the chat sensor gets deployed and we can see it in the GUI as shown in following screenshot.





g) The client can be stopped by pressing the stop button

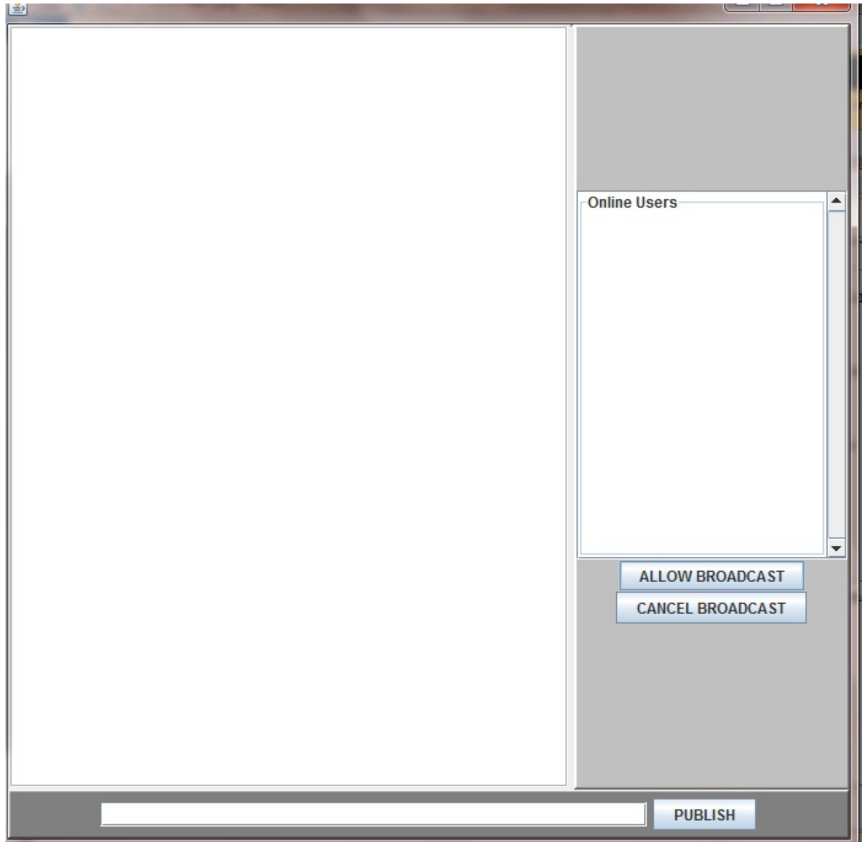
Batch Scripts Steps :

a) To deploy a chatSensor, just run the script prepareChatSensor which would inturn execute the runChatSensor script which would deploy the chat sensor.

```
C:\Windows\system32\cmd.exe - prepareChatSensor.bat
E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>prepareChatSensor.bat

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>ECHO Starting ...
Starting ...

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>ECHO Executing ...
Executing ...

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>start runChatSensor "ChatSens
or1" TestSensorClientGroup "CGL" "Indiana Univ" "Indiana Univ" "Indiana Univ" no
 no no no no

E:\Sem4\PTI\SensorCloudProject\SvnCodeBase\scripts>pause
Press any key to continue . . .
```

b) Once the Chat sensor gets deployed successfully, you can see a new command prompt opening up as shown in screenshot.

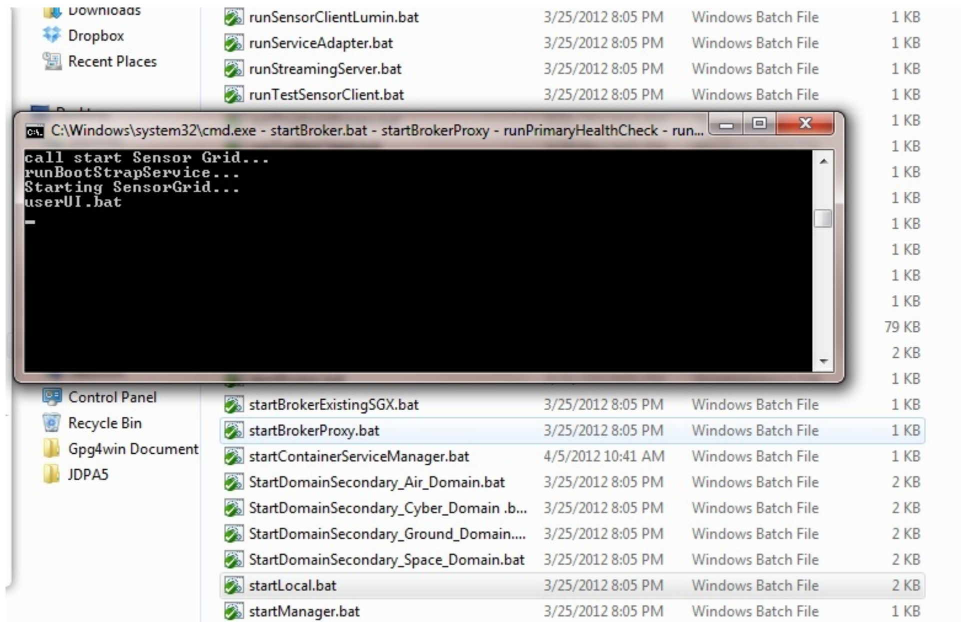
```
C:\Windows\system32\cmd.exe - runChatSensor "ChatSensor1" TestSensorClientGroup "CGL" "In...
[2012-04-20 13:04:18,702](DEBUG)[SGSensorView] - received message type: alive
[2012-04-20 13:04:27,183](DEBUG)[ServiceAdapter] - Registry Register/Renewal Suc
cessful !
[2012-04-20 13:04:27,184](DEBUG)[ServiceAdapter] - My current location is:topic:
//SA/972c44be-8b0a-11e1-8d4b-e521222d53be/5
[2012-04-20 13:04:28,700](DEBUG)[SGLogic] - received message type: alive
[2012-04-20 13:04:28,701](DEBUG)[SGLogic] - alive 1334941468701
[2012-04-20 13:04:28,702](DEBUG)[SGSensorView] - received message type: alive
[2012-04-20 13:04:38,701](DEBUG)[SGLogic] - received message type: alive
[2012-04-20 13:04:38,702](DEBUG)[SGLogic] - alive 1334941478702
[2012-04-20 13:04:38,703](DEBUG)[SGSensorView] - received message type: alive
[2012-04-20 13:04:48,703](DEBUG)[SGLogic] - received message type: alive
[2012-04-20 13:04:48,704](DEBUG)[SGLogic] - alive 1334941488704
[2012-04-20 13:04:48,705](DEBUG)[SGSensorView] - received message type: alive
[2012-04-20 13:04:57,202](DEBUG)[ServiceAdapter] - Registry Register/Renewal Suc
cessful !
[2012-04-20 13:04:57,202](DEBUG)[ServiceAdapter] - My current location is:topic:
//SA/972c44be-8b0a-11e1-8d4b-e521222d53be/5
[2012-04-20 13:04:58,709](DEBUG)[SGLogic] - received message type: alive
[2012-04-20 13:04:58,710](DEBUG)[SGLogic] - alive 1334941498710
[2012-04-20 13:04:58,711](DEBUG)[SGSensorView] - received message type: alive
[2012-04-20 13:05:08,716](DEBUG)[SGLogic] - received message type: alive
[2012-04-20 13:05:08,718](DEBUG)[SGLogic] - alive 1334941508718
[2012-04-20 13:05:08,721](DEBUG)[SGSensorView] - received message type: alive
```



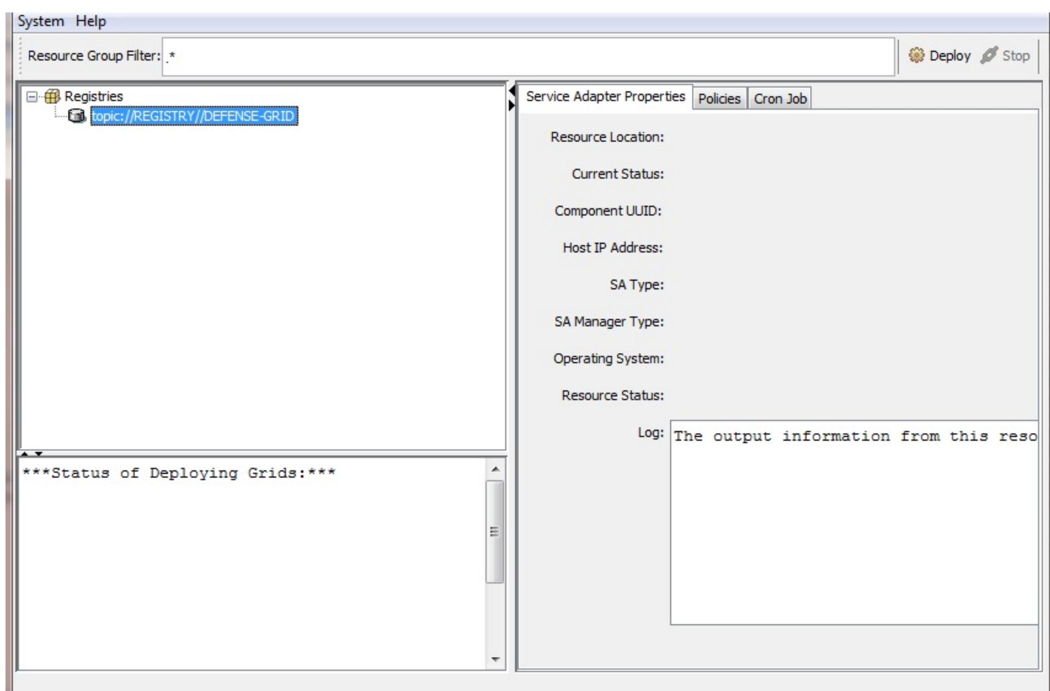
3. <u>FileTransfer Sensor</u>

GUI Client Steps:

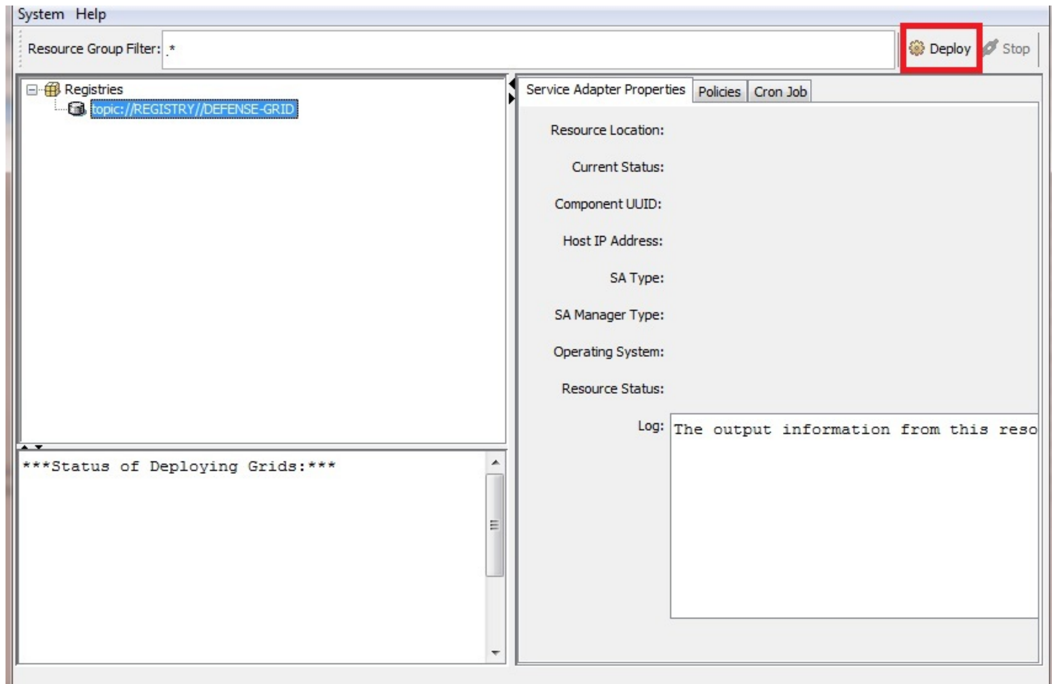
a) Run the startLocal.bat file to start the GUI interface for deploying file transfer sensor.



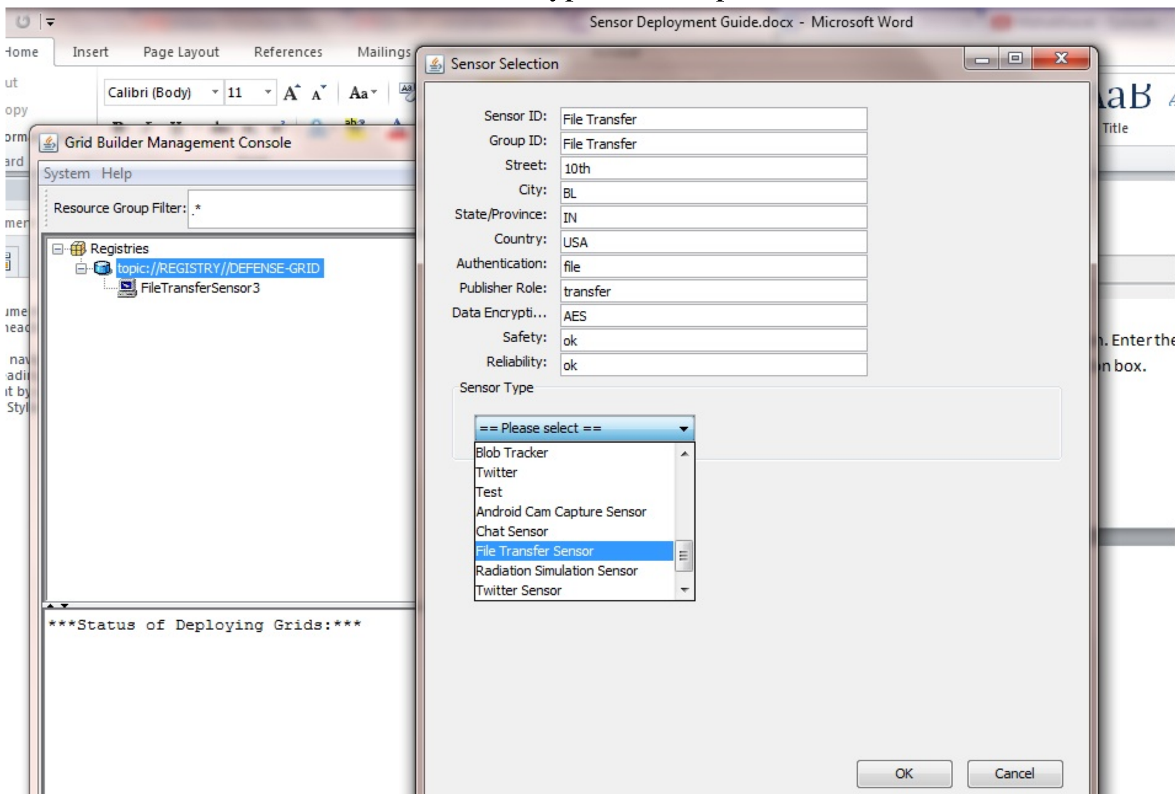
b) The startLocal script opens up the GUI



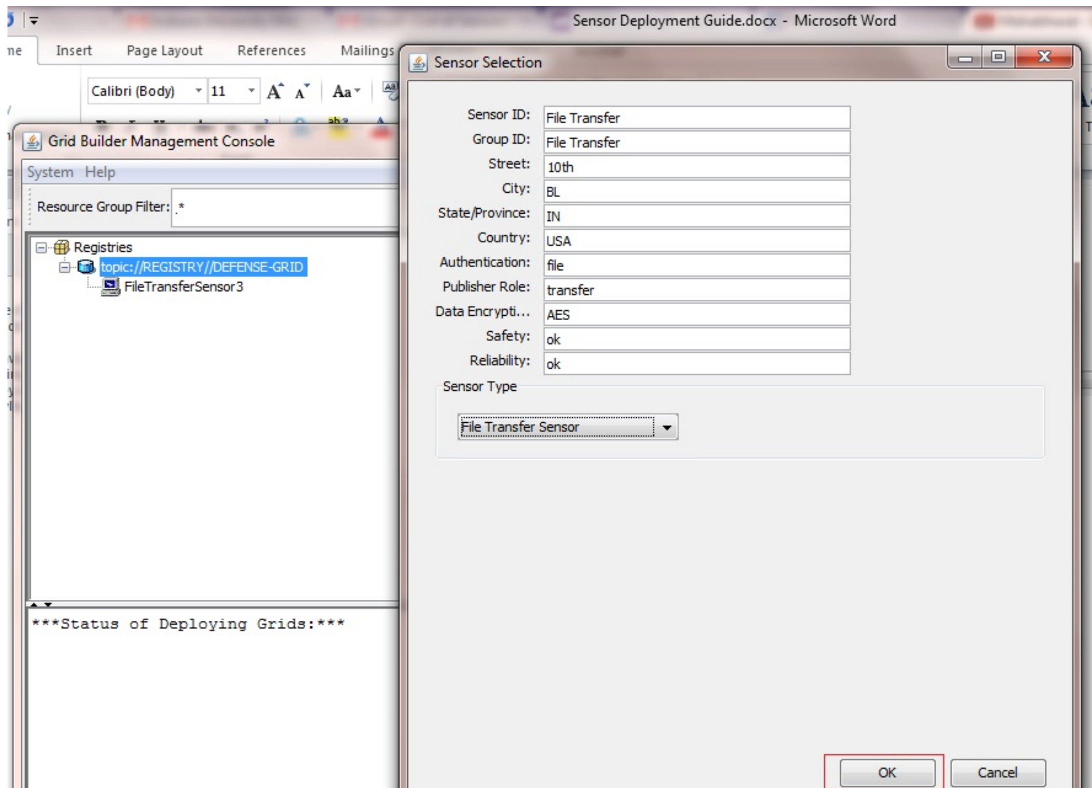
c) Once the GUI opens, click on the Deploy



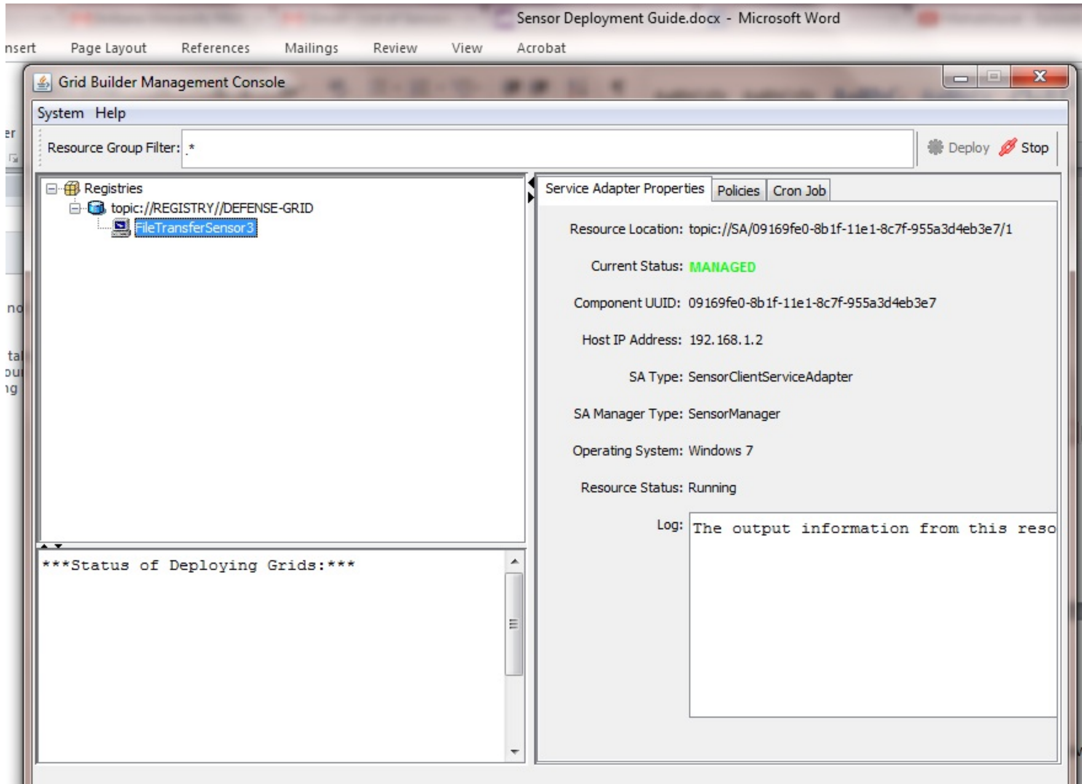
d) Clicking on the deploy button, opens a new window for sensor selection. Enter the relevant data for file transfer sensor and select the type from dropdown selection box.

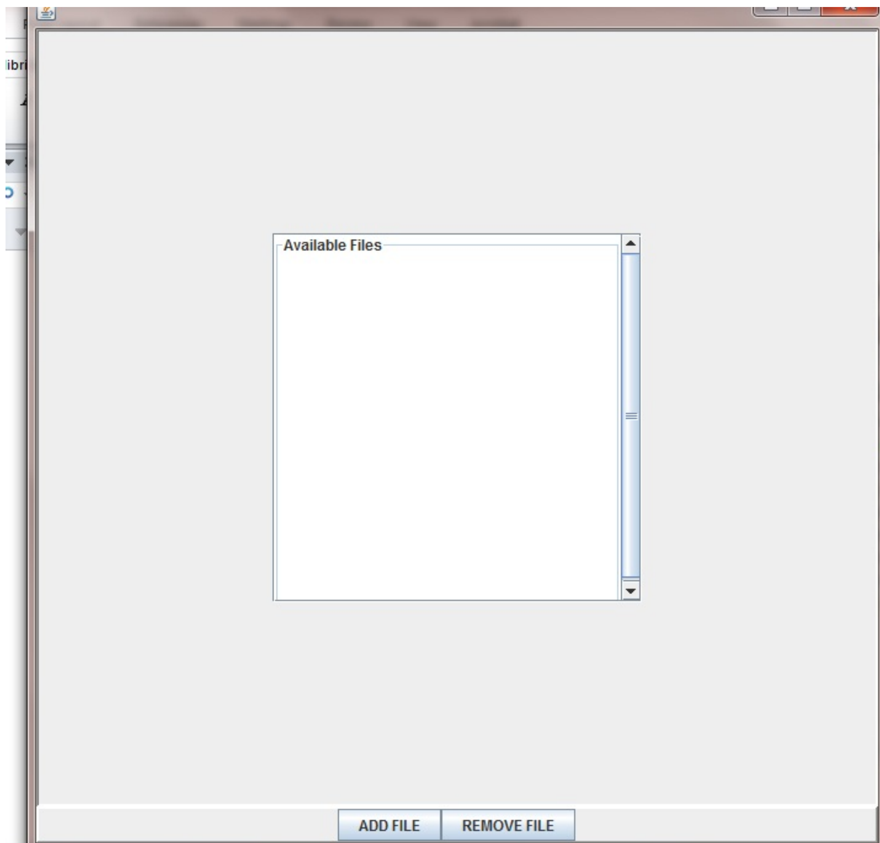


e) After filling the details, click on OK button.



f) Then the file transfer sensor gets deployed and we can see it in the GUI as shown in following screenshot.

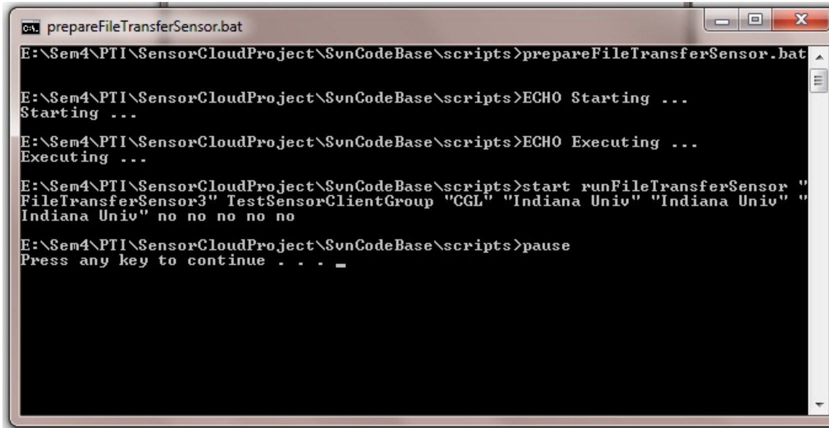




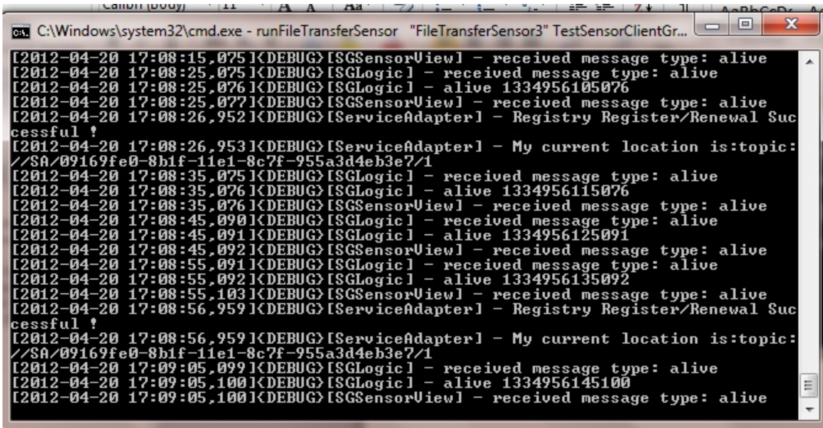
g) The sensor can be stopped by clicking on the Stop button

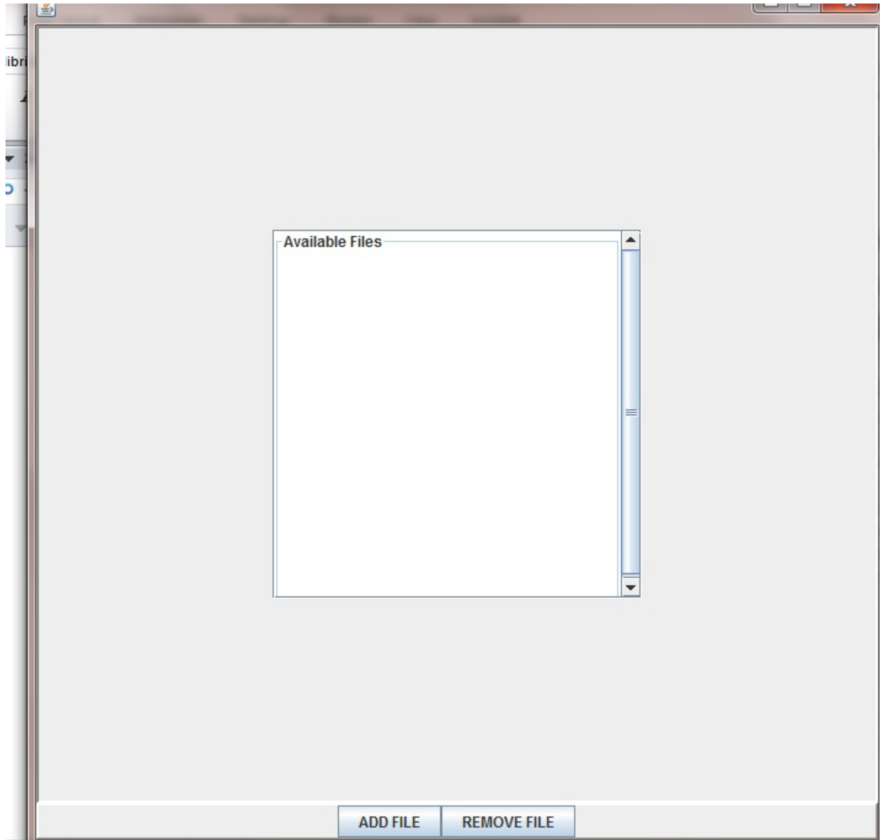
Batch Scripts Steps:

a) To deploy a File Transfer Sensor, just run the script prepareFileTransferSensor which would in turn execute the runFIleTransferSensor script which would deploy the chat sensor.



b) Once the Chat sensor gets deployed successfully, you can see a new command prompt opening up as shown in screenshot.

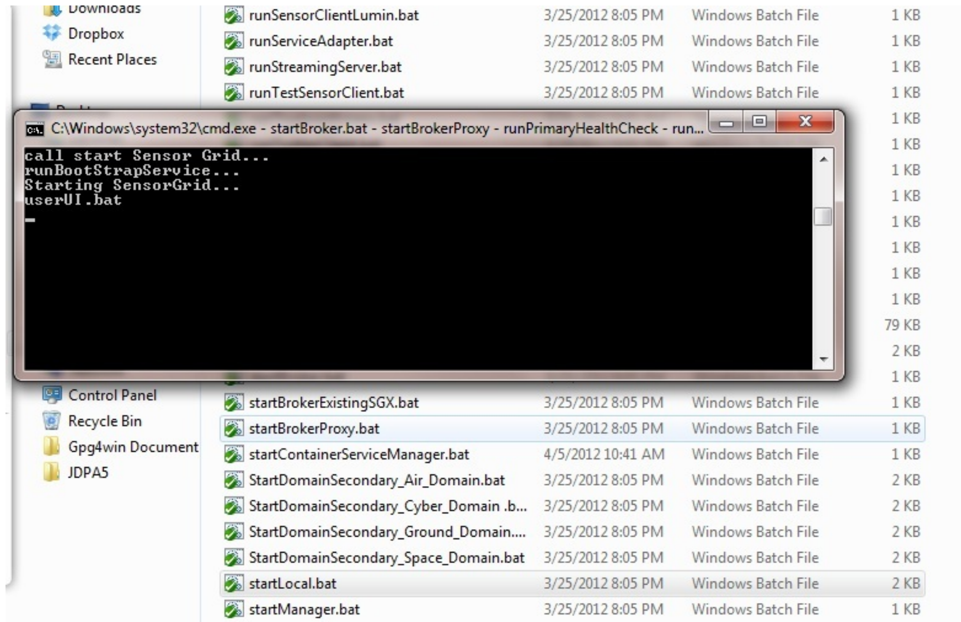




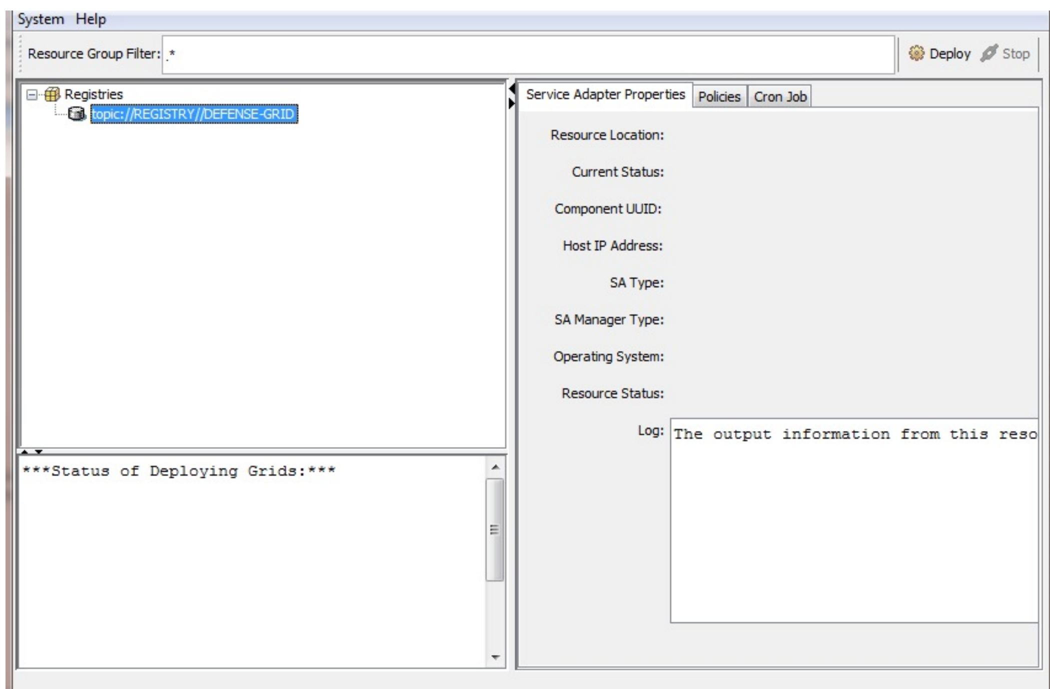
4. GPS Sensor

GUI Client Steps:

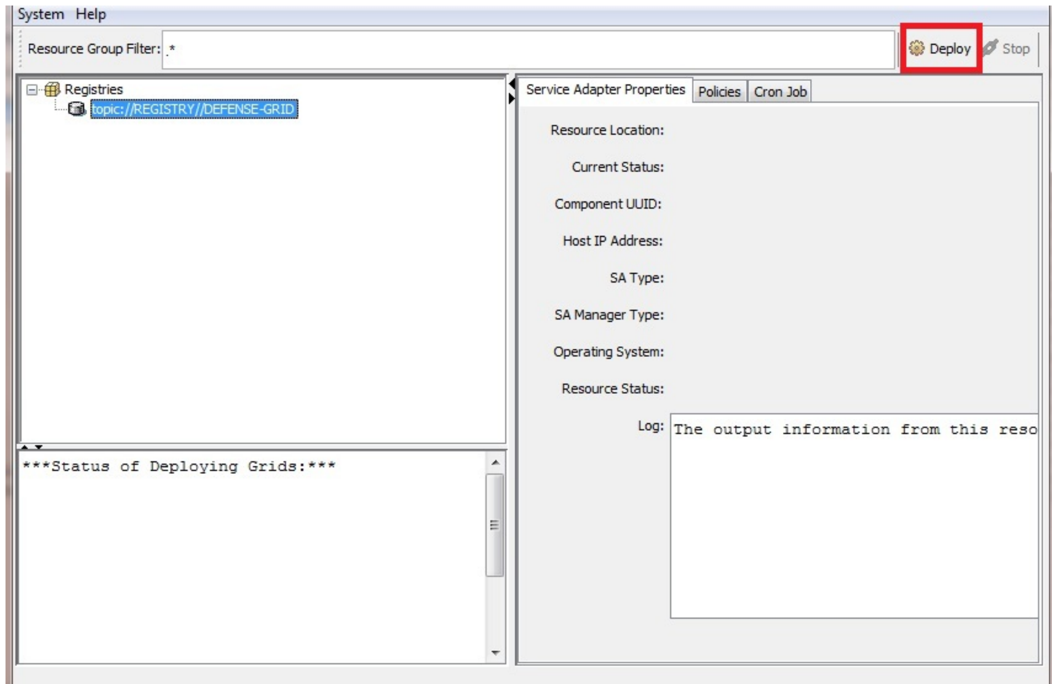
a) Run the startLocal.bat file to start the GUI interface for deploying GPS sensor.



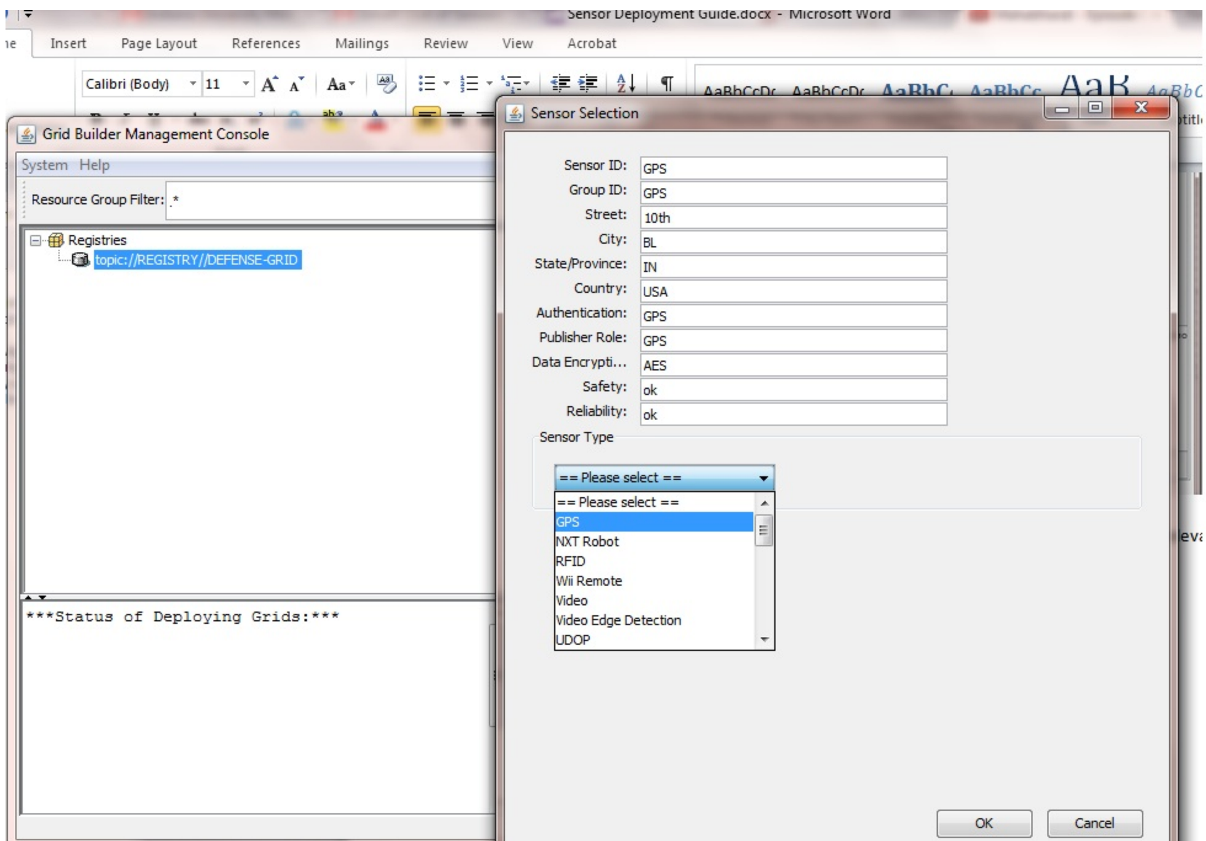
b) The startLocal script opens up the GUI



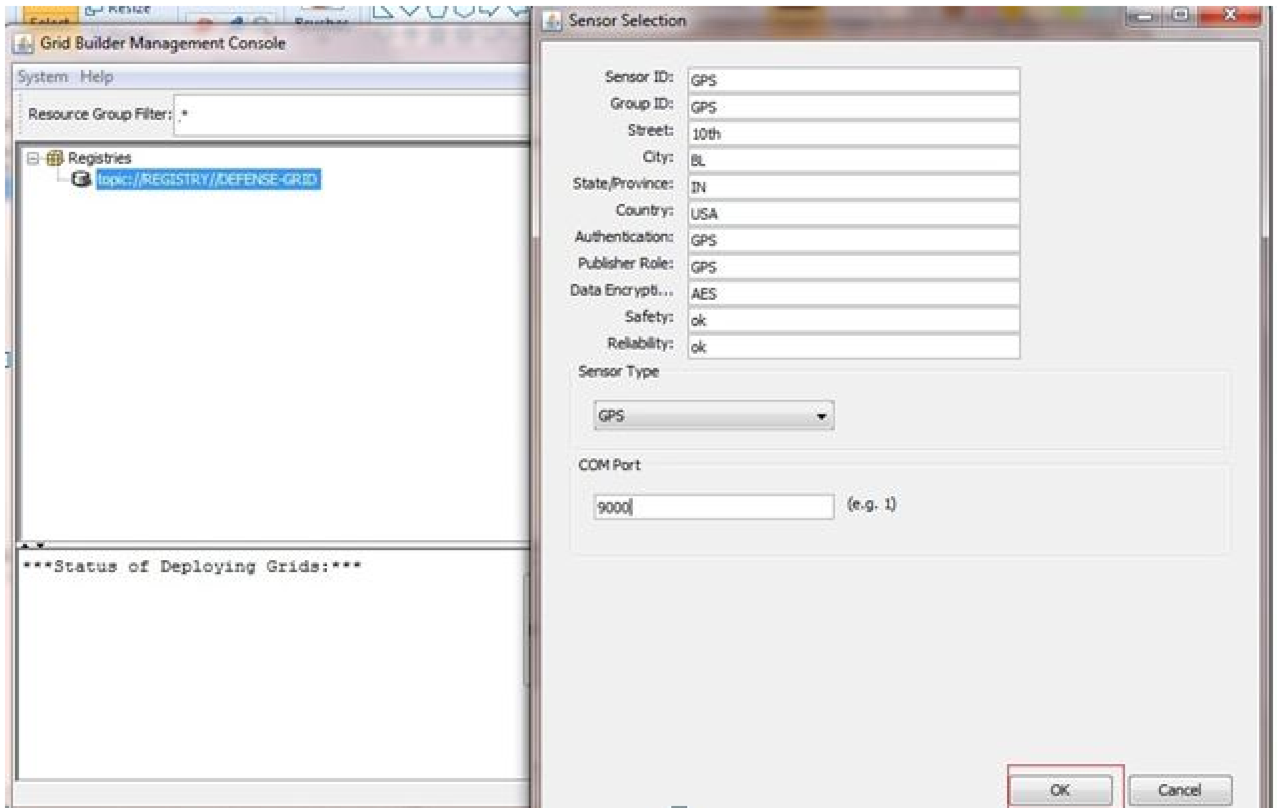
c) Once the GUI opens, click on the Deploy



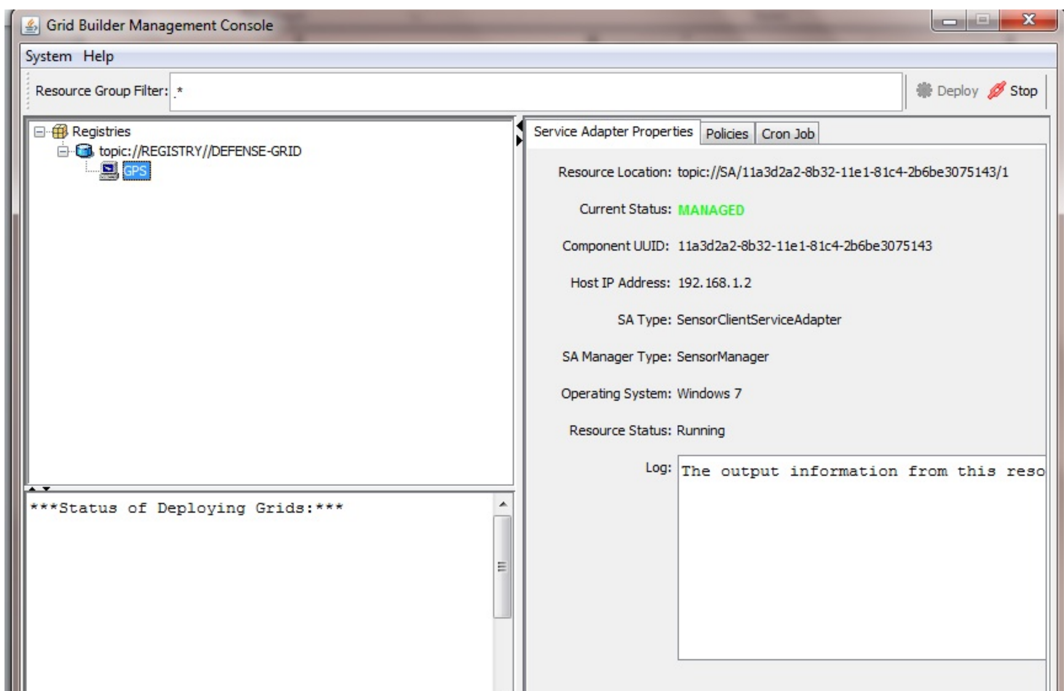
d) Clicking on the deploy button, opens a new window for sensor selection. Enter the relevant data for GPS sensor and select the type from dropdown selection box.



e) After filling the details, click on OK button.



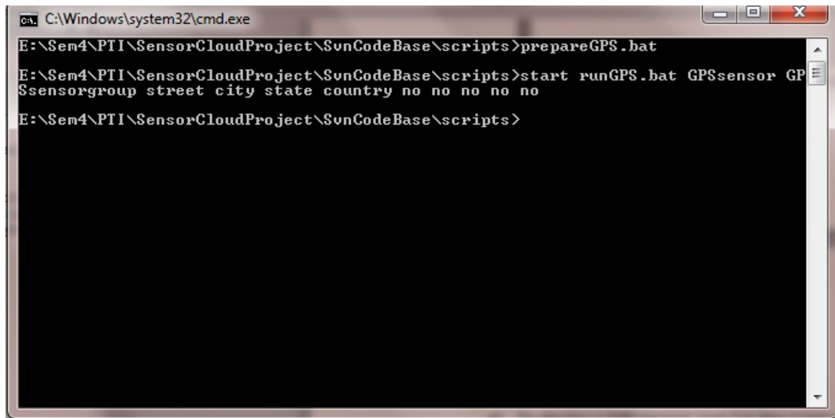
f) Then the GPS sensor gets deployed and we can see it in the GUI as shown in following screenshot.



g) To stop the sensor, the stop button can be clicked.

Batch Scripts Steps:

a) To deploy a GPS Sensor, just run the script prepareGPSSensor which would in turn execute the runGPS Sensor script which would deploy the chat sensor.



2.8 Guide to develop sensors and clients

Getting Started with Coding:

For our purposes a sensor is anything that transmits data. A video camera, a GPS unit, and for this discussion a sensor that sends message to the Grid, are all examples of sensors.

I have a sensor I want to use it to send data, what should I do now?
Great!! You need to let the sensor grid know that you have a sensor and you want to publish/subscribe the data. By doing this you are allowing the sensor grid to recognize the sensor and also adding it to the grid. In order to do that, you have to instantiate a sensor adapter object.
Let's Walk through our example QuickStart Sensor class to understand how to develop a sensor.

What is a SensorAdapter/SensorClientAdapter?

There are two main classes SensorAdapter and SensorClientAdapter which does all the primary functions in hooking up the sensor with the grid and maintaining the sensor in the grid.

SensorAdapter and SensorClientAdapter are the classes which allows you to implement the following functions

- Connecting the sensor to the sensor Grid and initializing the sensor over the grid
- Monitor the status of the sensor in the grid
- Receives data from the grid and publish the data on the grid
- Subscribe a sensor to monitor another sensor's data

The primary functions of the SensorAdapter are listed below along with the corresponding classes which implement them.
Creates and sets the policy (Sensor properties) of a sensor (SensorPolicy)
Allows the sensor to listen to the control signals (SensorGridControlListener)
Allows the sensor to create functions to handle connection loss or sensor stop requests (SensorAdapterListener)

The constructor of the SensorAdapter looks like this

```
public SensorAdapter(SensorPolicy sPolicy, SensorGridControlListener ctrlListener, SensorAdapterListener saListener) {
    init(sPolicy, ctrlListener, saListener, null);
}

public SensorAdapter(SensorPolicy sPolicy, SensorGridControlListener ctrlListener, SensorAdapterListener saListener, String uID) {
    init(sPolicy, ctrlListener, saListener, uID);
}
```

SensorClientAdapter is the adapter which basically does the same functions of the SensorAdapter except that it does from the client side but with an additional functionality. Any client which is subscribed to a sensor might want to be notified any changes in the grid about a new sensor being added or if the status of an existing sensor changes. This it implements an additional interface called ClientGridChangeListener.

The constructor of the SensorClientAdapter looks like this

```
public SensorClientAdapter   (SensorPolicy sPolicy,
SensorGridControlListener sensorCtrlListener, ClientGridChangeListener clientChangeListener,
SensorAdapterListener saListener)
{
Init(sPolicy, sensorCtrlListener, clientChangeListener, saListener, null);
```

What are those arguments in the constructor?
Let's see in detail about the arguments listed above.
SensorPolicy – The properties of the sensor are wrapped in to this SensorPolicy class. Once you create an instance of SensorPolicy you can access the property of a sensor through this class. The constructor of the SensorPolicy looks like the one below.

```
public SensorPolicy(SensorProperty sensorProperty)
        {
                this.sensorProperty = sensorProperty;
        }
```

Now,lets talk in detail about SensorProperty. Every sensor has few properties which are to be defined while creating a sensor. There are two types of sensor properties. The pre-defined generic sensor property and the user-defined sensor property. The predefined generic sensor properties include the following list

Property	Datatype	Description
sensorId	String	A description to identify the sensor. Eg:- QuickStartSensor
groupId	String	A Description to identify the logical group to which the sensor belongs to Eg:- QuickStartSensorGroup
sensorType	String	To identify the Type of the Sensor. Eg: Video, Audio, GPS etc..A list of predefined types are available in class PredefinedType.
sensorTypeId	Int	This allows the application to uniquely identify a sensor combining with the sensorType
Location	String	Geo-Spatial location of the

		sensor. E.g (United States, Bloomington)
Historical	Booelan	To identify if the sensor has time inter dependence with one another
sensorControl	Int[]	An integer array which identifies all control messages idenified by the sensor
controlDescription	String[]	Textual description of what each control message represents. Align it with the sensor control array.
userDefinedPropXml	String	Any other properties of the sensor cab be put here as XML string.

UserDefinedProperty:
Apart from the above predefined generic properties, a sensor can have a property of its own. For example, a Wii remote sensor might need Bluetooth address to connect to a System, the property of which is not present in the predefined properties. So the user can create their own property to hold Bluetooth address.
In this example of QuickStartSensor, we pass a String variable in the constructor while creating a SensorProperty instance. (Note: we can also create an empty UserDefinedProperty by using UserDefinedProperty.newInstance()).
The values for these sensor properties will be entered during the time of sensor deployment. So you need to have a SensorProperty class to bind the values to the properties.
We now know that SensorPolicy is a class which holds the properties of the sensor(SensorProperty)

SensorGridControlListener:
This is an interface which handles methods for listening to control messages from the application. It has a handleSensorControl method in it which should be implemented while implementing this interface.

```
public void handleSensorControl(String commander, int sensorControl);
public void handleSensorControl(String commander, int sensorControl, Serializable[] parameters);
```

Suppose if the sensor is capable of receiving control messages from the applicaion/sensor this class will be useful then. The string commander is the ID of the commander which sends out the control message.

ClientGridChangeListener:
Before knowing about ClientGridChangeListener, we need to know what is a SensorGridResource.

SensorGridResource is a class which facilitates the application to know about the properties of a

particular sensor along with the status of the sensor in the grid.
The constructor of this class looks like below

```
public SensorGridResource(Policy policy, short status) {
                this.policy = policy;
                this.status = status;
        }
```

Whenever this class is created, the sensor's policy and the status of the sensor is passed in the arguments. We know the first parameter, sensor policy gives access to the property of the sensor and the second one, status defines the current status of the sensor in the grid. It could be either 0(offline) or 1(online).
Now we know the SensorGridResource holds the properties and status of a sensor. But we do not know to which sensor are these values mapped to. So we now introduce a new variable *sensorInitInfo* which is a hashmap of sensor Id and SensorGridResource. With sensorInitInfo we can find a corresponding sensor's properties and status on the grid.
So, coming back to ClientGridChangeListener interface, a change happens in the Grid when

i. a new sensor is added to the grid
ii. a sensor's status change from online to offline or vice versa

To handle the first event we should initialise the new sensor that is added to the grid. This is when we implement the below method in ClientGridChangeListener interface.
public void handleClientInit(HashMap<String, SensorGridResource> sensorInitInfo);

We have the sensorInitInfo, therefore we know the sensor's properties. Hence we can intitialise the sensor by subscribing it to receive data from any other sensor.

To handle the second event we should add or remove the SensorMonitor corresponding to the sensor's status. If the sensor's status changes from offline to online, we have to subscribe the sensor to receive data from the requested sensor. If the sensor's status changes from offline to online we no longer should unsubscribe the sensor from monitoring any other sensor's data.

```
public void handleClientChange(HashMap<String, SensorGridResource> sensorChangeInfo, boolean
newFilter);
```

The newfilter is in the above method is to indicate the status of the sensor.

SensorAdapterListener:

Sometimes we might have to kill the sensor forcefully or sometimes the sensor looses the connection with the grid. We need to handle these situations. The SensorAdapterListener interface implements two methods to capture the above two situation.

```
public interface SensorAdapterListener {
        public void handleSensorStopRequest();
        public void handleSensorConnectionLoss();
}
```

We need to implement these two methods in the sensorManager class. In our QuickStartSensor example we simply exit the sensor's out of the grid.

```
public void handleSensorStopRequest() {
        log.info("Stop Request...");
        close();

        System.exit(0);
}

public void handleSensorConnectionLoss() {
        log.info("Connection Loss...");
        close();

        System.exit(0);
}
```

But, how do I actually publish and subscribe the data to the grid ?

Previously we learned,
1. how to connect a sensor to the grid?
2. What happens in the grid when a sensor is initated or when the status of the sensor is changed?
3. How does the sensor handles control signals from the application ?
4. What happens when the sensor gets terminated or when there is a connection loss ?

Now we need to know, *How to actually* ***publish and subscribe*** *a sensor's data in the grid ?*
We basically need to have a data(SensorData) that has to be published over the grid and a listener platform for the sensor(ClientGridDataListener) to listen to a specific SensorData. When you are ready with these two, you can call any of the three methods(mentioned below) of the SensorClientAdapter to publish and subscribe/unsubscibe your SensorData. The three methods are

- publishData(SensorData sensorData)
- subscribeSensorData(String sensorID, ClientGridDataListener listener)

- unsubscribeSensorData(String sensorID, ClientGridDataListener listener)

The publishData(SensorData sensorData) method is implemented in the SensorClientAdapter in such a way that it can publish the SensorData over the grid. Similarly the subscribeSensorData(String sensorID, ClientGridDataListener listener) and unsubscribeSensorData(String sensorID, ClientGridDataListener listener) does subscribing and unsubscribing of a sensor from a sensor data.
Let's see in detail about

- SensorData
- ClientGridDataListener

SensorData:
You need to define the type of data you would be publishing/subscribing from the grid. Typically we define a class to specify the type of data to deal with, which extends an abstract class SensorData.
The QuickStartSensorData which is used for the QuickStartSensor looks like the one below.

```
public class QuickStartSensorData extends SensorData {

   /**
    * Serialization Version
    */
   private static final long serialVersionUID = 1L;
   private String message;


   /**
    * @param timestamp - Message creation time
    * @param message - Message to publish
    */
   public QuickStartSensorData(long timestamp, String message) {
      super(timestamp);
      this.message = message;
   }

   public String getMessage() {
      return message + " " + Long.toString(this.getTimestamp());
   }
}
```

Since in the QuickStartSensor we are just reading strings of data and displaying on a command console, we have a string variable to take care of it. In real world we might be using a little more complicated sensors like Wii remote where we have to take care of all the keys and buttons in the remote.

ClientGridDataListener:
What should the sensor do upon receiving a data from the grid? we need to take care of the data. ClientGridDataListener is an interface which demands you to implement a method on how to handle the sensor data that is received from the grid. This is when we implement the DataMonitor class. Every

sensor has a DataMonitor class which extends the ClientGridDataListener. This DataMonitor class monitors the grid for the sensor data and upon receving a sensor data, the handleSensorData method decides on what should be done with the SensorData.

```
public interface ClientGridDataListener {
        public void handleSensorData(String sensorID, SensorData sensorData);
}
```

The above method is implemented through the ClientGridDataListener. It basically takes in the SensorData and the sensorID and decides on what should be done.

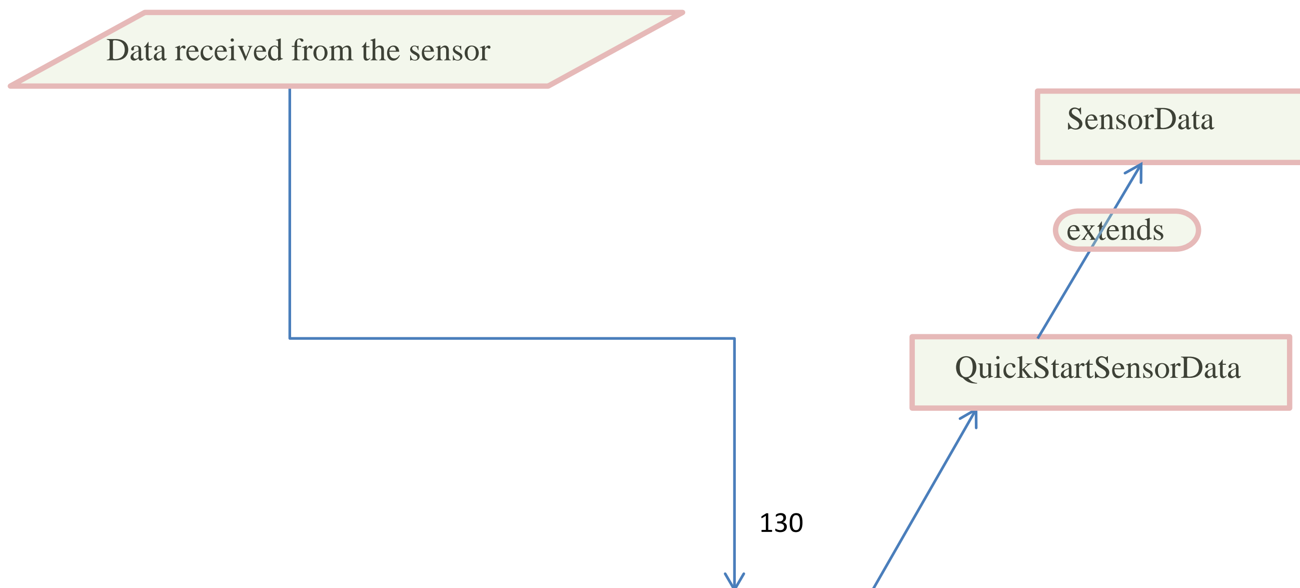
The DataListener for the QuickStartSensor looks like the one below.

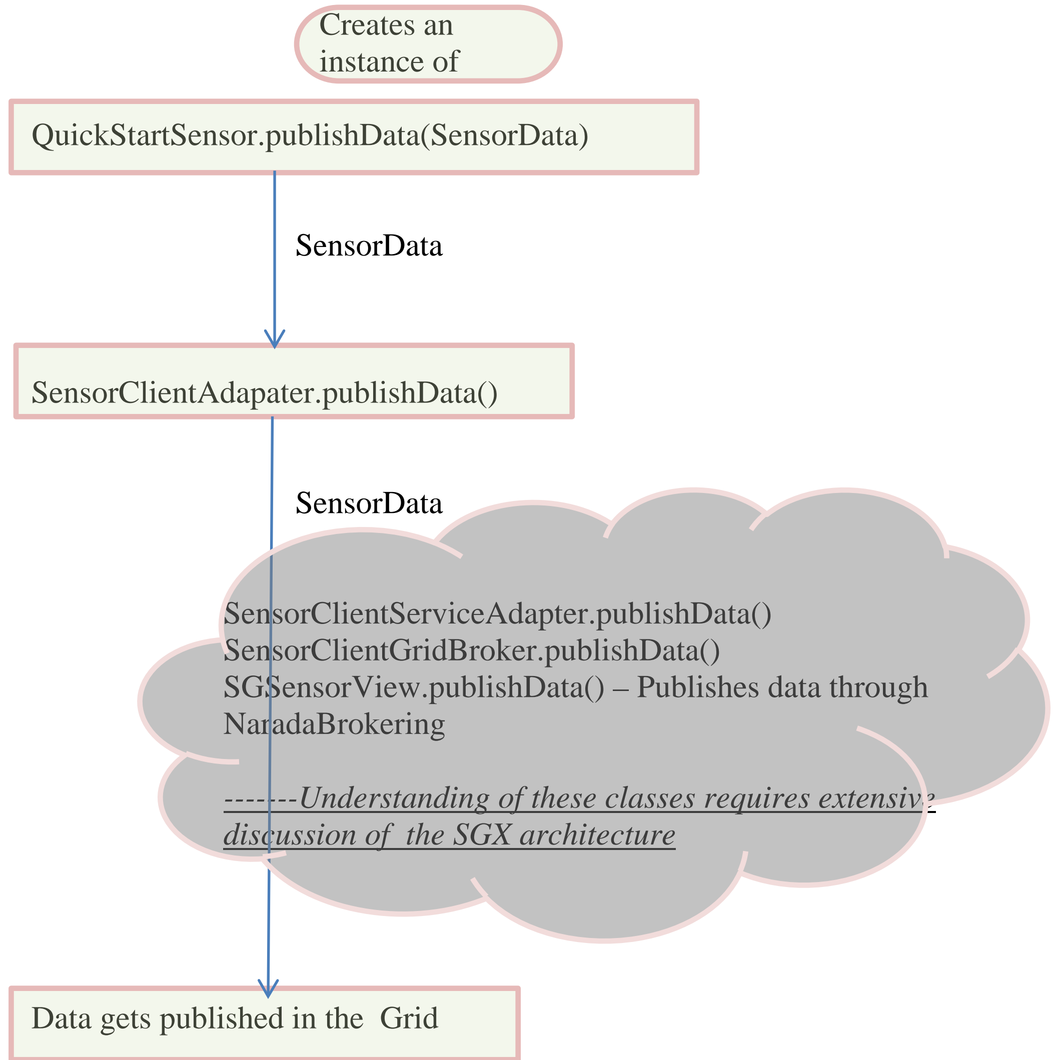
```
public void handleSensorData(String sensorID, SensorData data) {

        //We have some data, check to see if it is from a QuickStartSensor
        if( data instanceof QuickStartSensorData ) {
           // We have QuickStartSensor Data get the message
           QuickStartSensorData quickStartSensorData = (QuickStartSensorData)data;

           String output = "\nQuickStartSensor Data received, id: " + id +
           ", data: " + quickStartSensorData.getMessage() + "\n";
           log.info(output);
        } else {
           // Here we dump data from all other sensors in a generic blob.
           String output = "\nOther Data received, id: " + id + "\n";
           log.info(output);
```

```
        }
    }
```

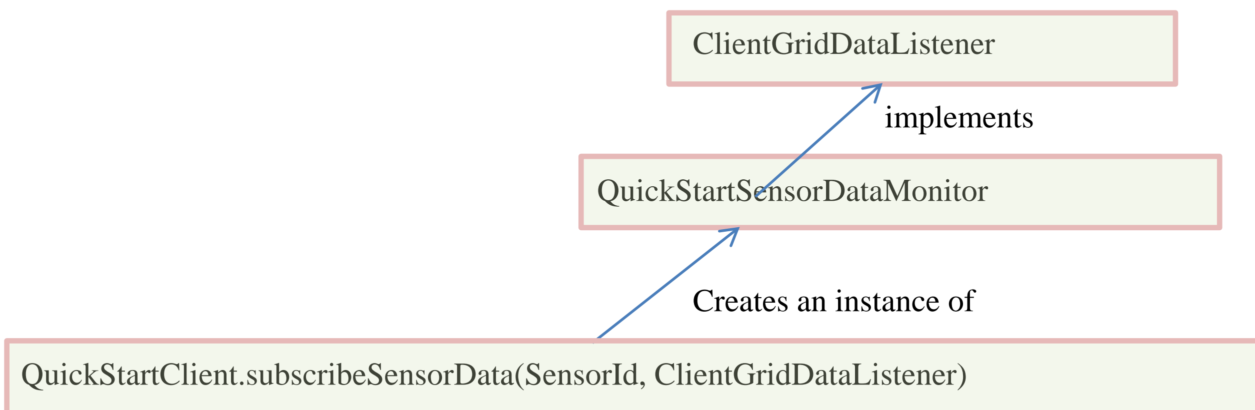
The variable quickStartSensorData would have the string value read from the QuickStartSensor. Once the data is retrieved, we have used a QuickStartClient which will display the message on the client command window.

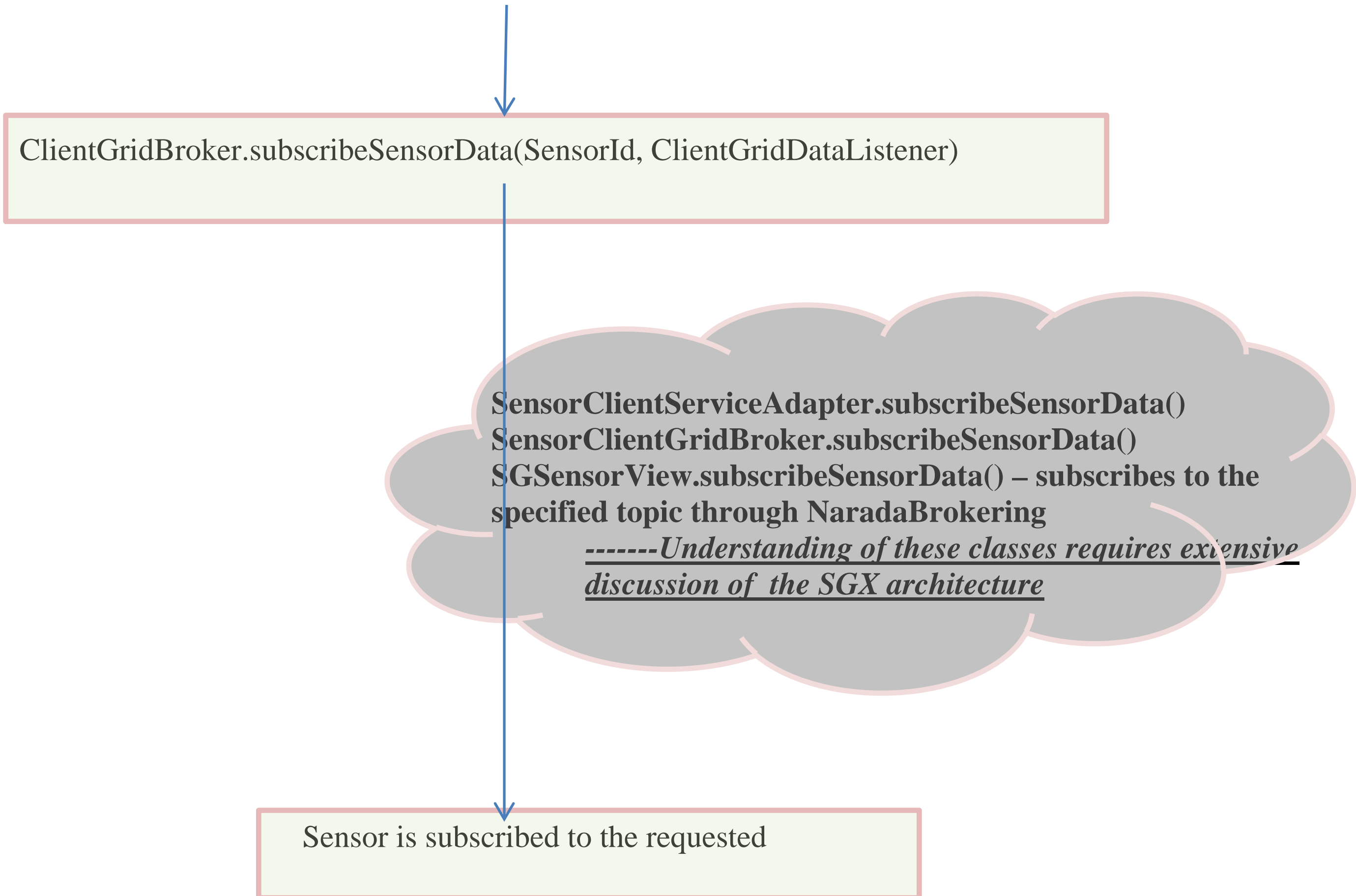
Publishing data on the Grid





Subscribing sensor Data





How a Client sends control messages to a Sensor with whome the Client has subscribed.

The Client would already contain an instance of ClientGridBroker and in case if we are looking at a Client which is ment to behave both as a Client and a Sensor (e.g. ChatSensor.java) and inisitates a SensorClientAdapter in it's constructor. In either case both of them exposes a method "sendControl(String sensorID, int sensorControl)", this is the required method which needs to be call when the Client needs to send a control message to the sensor.

Note: in case of SensorClientAdapter the actual implementation of sending controlMessage is encapsulated in SensorClientGridBroker.

Moving towards explaining how to use the QuickStart Client to send a control message to the Sensor. We can obtain the required HELP contents by typing "HELP" command directly in to the console which

provides us the required information about how to query about the Sensors with the Client has subscribed and and what are the avaliable/valid control messages. And hence to send control message to a sensor type:

<Sensor ID> <Control Message>

Please Note this demostration only applies to QuickStart Sensors and QuickStart Client.

Before moving on to the next section let's just recap what we have learned so far!

Recap:

- Once you decide on the sensor that you are going to develop the first thing to do is to instantiate a SensorClientAdapter
- A SensorAdapter is the bridge between the sensor and the grid which lets the sensor grid know about the sensor's existence
- SensorAdapter takes four different parameters in its constructor. They are
  - SensorPolicy
  - SensorGridControlListener
  - ClientGridChangeListener
  - SensorAdapterListener
- SensorPolicy has the properties of the Sensor
  - SensorProperty is the class which holds all the properties of a sensor
  - There are two types of sensor properties. Predefined properties and the user defined properties
- *SensorGridControlListener* implements handleSensorControl method which takes care of control messges from the application

*ClientGridChangeListene*r takes care of what happens when a new sensor gets registered in the grid or when the status of the sensor changes in the grid? It implements two

- methods, *handleClientInit* and *handleClientChange* to handle the above situations correspondingly
- *SensorGridResource* is a class which holds SensorProperty and the status of the sensor.This is used to identify the properties of a sensor along with its status on the grid
- *SensorAdapterListener* listens to the application specific events such as connection loss. In order to publish and subscribe a sensor data, we need to create two classes:
  - CustomSensorData which extends the SensorData class
  - DataMonitor which implement the ClientGridDataListener interface

I need to start writing the Main function. What should I do now ?
Let's start writing the Main function now.

```
public static void main(String[] args){
```

....then what ?

Its simple. Just find a way to get the properties of the sensor you are developing. Once you have the values of these properties, then you create an instance of SensorProperty class. Now that we have the SensorProperty of the current sensor, create a SensorPoilcy instance.
We now have everything to create an instance of the current sensor class (QuickStartSensor) and allow it to join the grid. The constructor of the QuickStartSensor looks like this

```
public QuickStartSensor (SensorPolicy sPolicy) {

        this.m_sensorPolicy = sPolicy;
        m_uuid=UIDGenerator.getUUID();
m_sensorClientAdapter = new SensorAdapter(sPolicy, this, this,
                                m_uuid);

        }
```

We know that inorder to let the sensor grid know the sensor's existence we have to create a SensorClientAdapter instance. We also know, what are the arguments needed to be passed to the SensorClientAdapter class. (Note that here we have an extra parameter "m_uuid"(*). We will discuss about this later).
Things are set now! The only thing remaining is publishing and subscribing the data itself. Get the data from the sensor and wrap it inside the SensorData class. Pass this SensorData to the publishMethod of the SensorClientAdapter. The data is thus published over the grid.

```
public void publishData() {
    // In our contrived example we just publish the message we stored in the sensorPolicy
    QuickStartSensorUserDefinedProperty userDefinedProp =
        (QuickStartSensorUserDefinedProperty)
        sensorPolicy.getSensorProperty().getUserDefinedProp();

    QuickStartSensorData data = new QuickStartSensorData(
            System.currentTimeMillis(), userDefinedProp.getMessage());

    sensorAdapter.publishData(data);
    System.out.println(data);
}
```

By now we should have already implemented the *handleClientInit()* method of the ClientGridDataListener interface in such a way that it can handle, when a new instance of this sensor is created and added to the grid. In case of QuickStartSensor it should automatically be added to the

DataMonitor of a similar sensor. This might not be same for all the sensor's.

For example, In case of an AndroidSensor when a new instance of the sensor is created, the *handleClientInit()* method can subscribe this instance to a NXTRobotSensor.
Similarly *handleClientChange()* method should take care of implementing details on what should happen when the sensor's status goes from online to offline or vice versa. In case of QuickStartSensor, when a sensor goes from online to offline, we need to remove the sensor from the DataMonitor.
Here the sensor's are getting subscribed/unsubscribed to the grid to listen to a data from another sensor through *handleClientInit()* method and *handleClientChange()* method. We can also allow the sensor to subscribe/unsubscribe to the grid explicitly in the main class by creating a monitor and calling the SensorClientAdapter subscribe()/unsubscribe() method.
Thus a sensor is added to the grid, allowed to publish the data on the grid and subscribed over the grid to receive the data from any sensor of your choice.

4. **Integrating the custom sensor with SGX framework**

Create a batch file for starting the sensor, which would look like:

```
@echo off

:: Use this to launch a Quick Start Sensor! You must supply the sensor

:: properties from the command line or another script.



CALL setEnv.bat

java "%JAVA_LIB%" -classpath %cp%
cgl.sensorcloud.quickstart.sensor.QuickStartSensor %*
```

2.9 Advanced Guide to develop sensors and clients

Before going in detail about this topic, let's brush up a few things that are very important to publishing data over the grid.

- The data is published using the NaradaBrokering messaging system.
- Every data is published over a particular topic in the NB messaging system.
- Data can be published either over a private topic or a public topic.
- Publishing a data on a public topic means, the data is published over the grid and the data reaches all the sensors available online. Handling those public data is up to the clients and sensors online.
- Publishing data over a private topic means, the data is actually sent to only those sensors/clients that are subscribed to the private topic.
- There are two types of data that could be published. The control data and the sensor data

Coming back to our original question, when a data is published over the grid it's actually published with the help of the NBJmsInitilializer class in the Narada Brokering.

Creation of NBJmsInitilializer for Sensors

The creation of this NBJmsInitilializer for sensors in SGX happens in the below fashion, SensorGridBroker ---initializes—SGLogic in which we create a new NaradaJMSBridge by passing all the connection properties, transport type, topic and the connection loss listener. The NaradaJMSBridge class is responsible for creating this NBJmsInitializer. The NaradaJMSBridge apart from creating the NBJmsInitializer it also takes care of creating publisher, publishing and subscribing messages to the Narada Brokering messaging system.

Creation of NBJmsInitilializer for Clients

The creation of this NBJmsInitilializer for clients in SGX happens in the below fashion, ClientGridBroker ---initializes—SGLogic, and similarly like the sensor in the previous section SGLogic encapsulates the creation and the working (publish/subscribe) of the NBJmsInitializer.

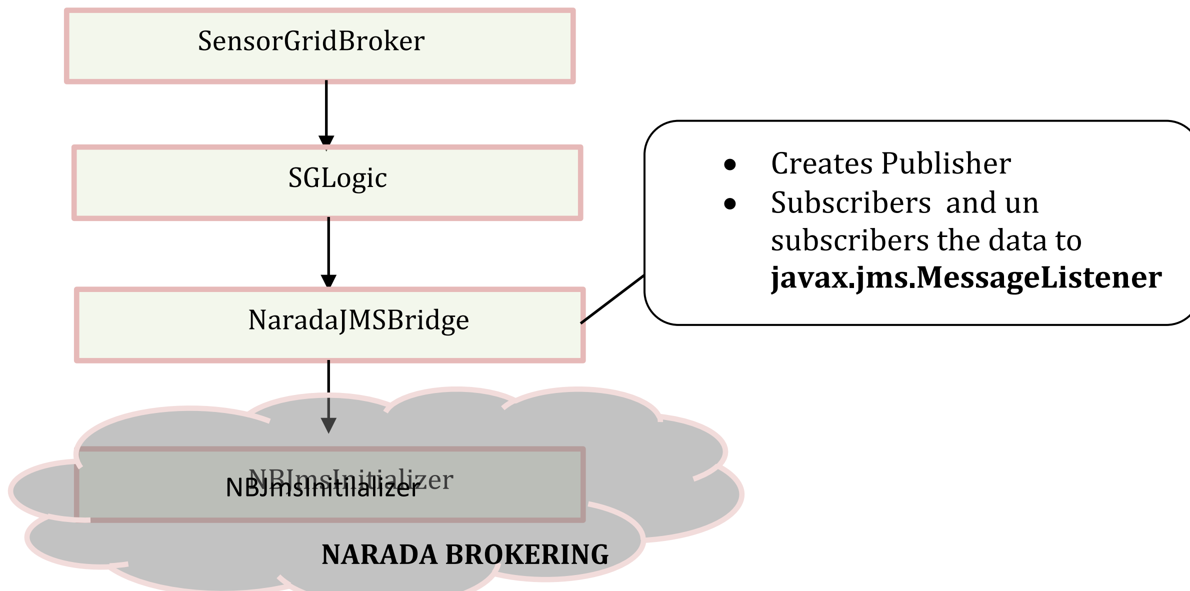


Fig 1.0 NaradaJmsBridge

How the Data is being published over the Grid?

Publishing a Sensor Data

SensorGridBroker initializes and holds on to a SGSensorView object by passing over the NaradaJMSBridge and the Sensor ID while creating it. The SGSensorView class also uses NaradaJMSBridge in its constructor to instantiate Topic/TopicPublisher (javax.jms.TopicPublisher) over which sensor data is published. SGSensorView class also takes care of creating private and public topics for the current publisher

application/sensorgrid/sensordata/m_sensorID – public topic and
application/sensorgrid/private – Private topic

Thus the data reaches the grid over a particular topic created by the sensor.

Publishing a control data

While the SGSensorView takes care of publishing a SensorData it's the responsibility of the SGClientView to publish a control data. Since mostly it's the clients who send the control signals to the sensor, the SGClientView takes care of publishing a control data. The publisher for control data creates a topic of the type

application/sensorgrid/sensorcontrol/+sensorID

and publishes the control signals over this topic. All the sensors created are by default subscribed to this topic so that they receive any control messages that are sent to this topic.

What happens to the data after publishing it on the grid?

There are two types of data that reaches the grid,

- The SensorData that is being published by a sensor
- The control data published by a client or other sensors that has to reach a particular sensor

Both SGClientView as well as SGSensorView implement the javax.jms.MessageListener interface and therefore provide concrete implementation of the onMessage method of the JMS message listener. This method is triggered every time when a data reaches the grid, and hence through the onMessage method the clients get access to the data published by the sensors and similarly the Sensors gets to listen to the control messages by the Client.

The Client's implementation of onMessage also includes differential implementation of different/multiple data types.

Filtering Sensors

All the sensors that we develop by default publish information to the Sensor Grid. A sensor could also be used subscribe to other sensors online on the grid by using SensorClientAdapter instead of SensorAdapter as in the previous case also for a sensor to be able to listen to messages from other sensors it should implement ClientGridChangeListener. Take a simple chat sensor application that we developed. It has its subscriptions made to other chat sensors, which are present on the grid and communicate with each other using the messaging broker system. In the Inter-language chat sensor application that we developed however, we establish a simple filtering mechanism in a way that the chat sensors which are deployed do not directly subscribe to other chat sensors but instead subscribe to special translate sensors that are fired at the time of launching the chat sensor provided it satisfies certain conditions. Thus we have architecture such that each chat client does not care about other clients unless they satisfy certain rules but only on the translation sensors of its default language.

In the regular sensor client model (e.g. The Quick Start Application), we have the sensors implement the SensorAdapter interface so that it can publish information to the grid and we have the clients implement the ClientGridChangeListener interface so that the clients can listen to the grid for changes like new sensors coming online or existing sensors leaving the grid so that it can make changes to its subscriptions as and when the changes occur. These two interfaces are the ones that specify boundaries between the sensor and the client as the sensor does not care about the information once it's published over the grid.

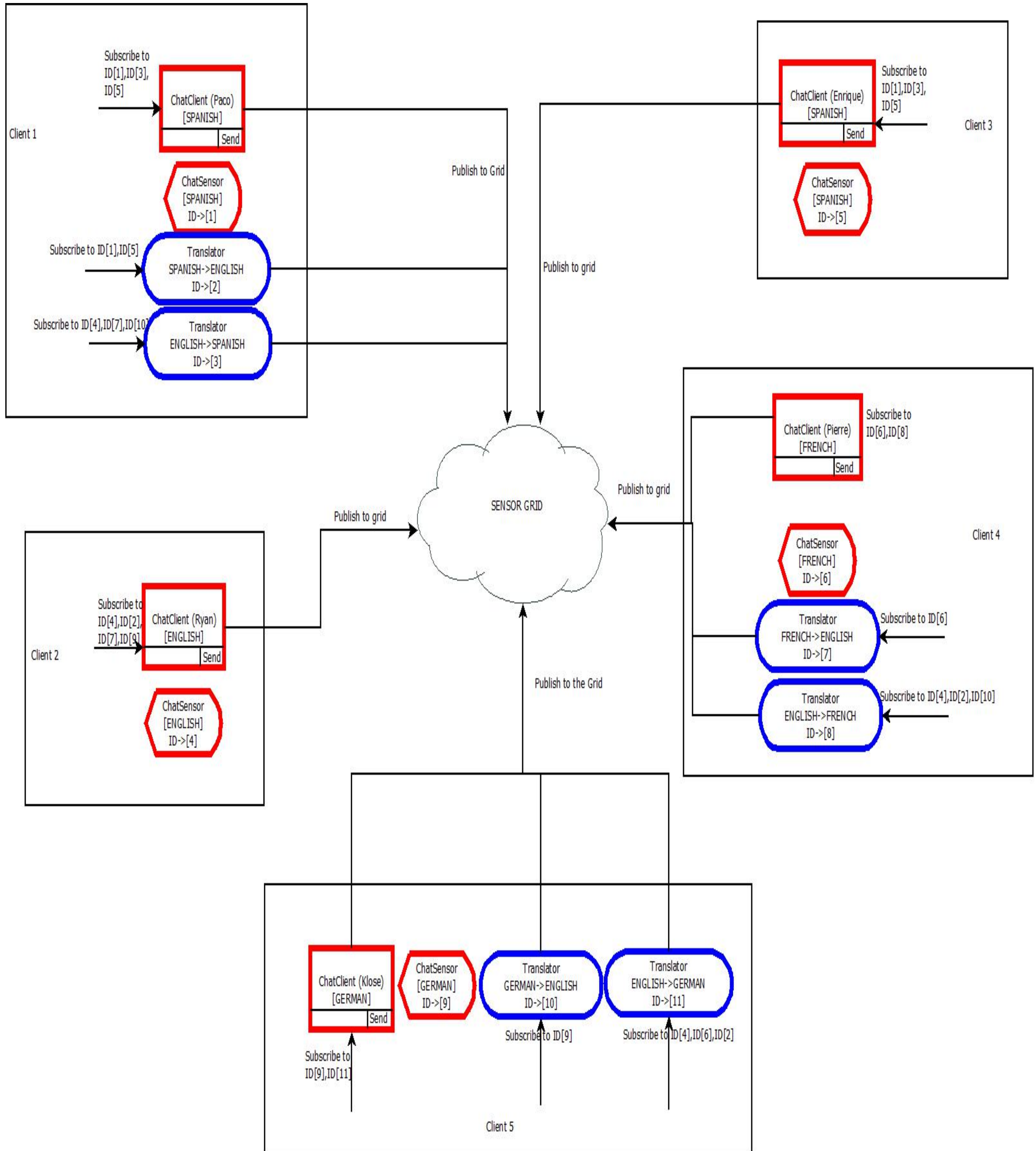
The filters that we implement for the Chat Application however act as both a client and a sensor as it implements both the SensorClientAdapter and ClientGridChangeListener interfaces to both publish and subscribe to information. This is a feature that distinguishes a filter from a regular client or sensor.

2.5 SensorCloud development: An overview

Architecture

The overall architecture of the system that we implemented is shown as follows. A few interesting things to note about the proposed model is that we assume that most of the users who come about to use our system know English and hence we assume English to be the main language for intermediate translations. Initially, the launcher client is run where the username and the default language of the chat client is selected by the user. Once this information is submitted we fire up the chat client with the default language entered in the launcher set in its User Defined properties. This language property is what is used to make comparisons to other chat clients and translation sensors to determine which to subscribe to and which not.

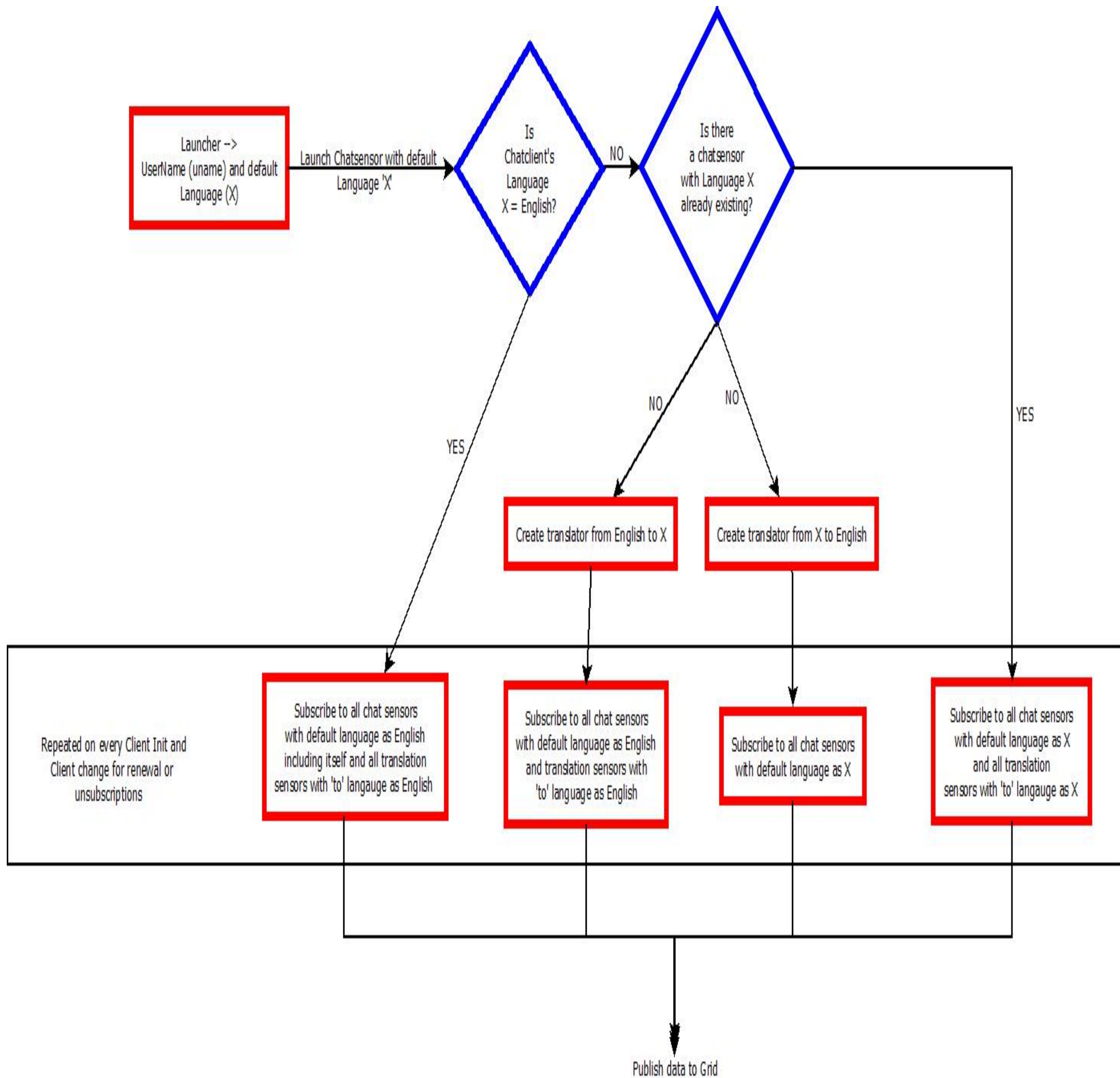
An important point to note about this architecture is that it scales well as the number of clients increases as the number of translation sensors fired up does not scale exponentially with the number of clients. This implemented model is the 2nd most efficient architecture possible compared to the more efficient method of having similar topics in the message broker for communications in a particular language in a way that other clients and translators can just subscribe to that particular topic instead of looking into its properties for the language it needs.



a. Architecture Model

Flow of Control

A snapshot of the overall flow of control in the system is shown below in the diagram.



b. Control Flow

Initially, the launcher is presented to the user who enters his Username and their default language to chat in and the launcher proceeds in creating the ChatSensor client. It sets the User Defined Language

property of the sensor to the language selected in the Launcher. This is the language that the other clients and translation sensors use to subscribe to the sensor. Once the chat client is created, a check is done to check if there is a need to fire additional translate sensors, this includes checking if the language of the chat client is English, if so there are no translators fired and we can move on to the subscription phase but if it is not English, then we check the Grid to look on the list of online sensors for one which has the same default language as the currently initialized sensor, if existing, we know that translators from and to the default language have already been created and we move on to the subscription stage. If that is not found, then we proceed forward in creating the new translators from the Language X to English and vice versa. The next stage is the subscription phase, which is where we have used translators as filters to the chat sensors.

Filtering and Subscribing

Once we have created the necessary translators and the chat clients are initialized and ready, we must let the newly initialized client subscribe to other translate sensors and clients. We must also update subscriptions of the already existing sensors to account for the latest sensor on the grid. The following rules apply for subscriptions

1. If the initialized chat client is English, then we subscribe to all the chat sensors whose language is English including itself and all the translation sensors whose 'to' language is English so that we receive translated text from all the clients.

2. In case of a translation sensor from the foreign language X to English, we subscribe to all the chat sensors which have a language X so that it will receive inputs of Language X which it translates and publishes to the Grid in English.

3. In case of a translation sensor from English to the foreign language X, we first subscribe to all chat sensors with default language English so that this English input from English clients can be converted to a foreign language and then we also subscribe to translation sensors with the 'to' language as English, this ensures that this translates the outputs of all other translators from English to its native language.

4. In case we have a chat sensor whose language is not English, we subscribe to other chat sensors with the default language as X and all translation sensors with the 'to' language as X.

In the above subscription model, we have achieved a case where the chat sensors do not subscribe directly to other chat sensors unless absolutely necessary and they pass their outputs on to another set of sensors namely the translation sensors in this case and they are dependent on only these filters to send and receive information in most cases unless we see two clients of the same native language. Hence we have achieved a sort of 'filtering' that changes the subscriptions to other sensors depending on the 'default language' property of the Chat sensor.

Another important factor to note in this case is that the subscriptions must be done and redone in two cases namely when a new chat sensor comes online i.e. during the client initialization and when some

sensor comes online or an existing sensor goes offline from the grid, i.e. when there is a change in the state of the grid.

There are two methods where the calls must be made for subscribing to new sensor data. Namely the handleClientInit() method and the handleClientChange() method.

handleClientInit():

```
public void handleClientInit(
        final HashMap<String, SensorGridResource> sensorInitInfo)

{

……………………………

if(!Language.equalsIgnoreCase("ENGLISH"))
            {
    System.out.println(Language + "Is the Source Clients Language Found");
            checkLanguageClient = 1;
    }

………………………

}
```

The above code segments shows us the handleClientInit() method which accepts a hashmap of currently available sensors on the grid as input parameters. We check if the default language is English and set a flag appropriately to call the addSensorMonitor() which is where we subscribe to other sensors.

handleClientChange():

```
public void handleClientChange(
        final HashMap<String, SensorGridResource> sensorChangeInfo,
        final boolean newFilter)

{

…………………………
if (Language.equalsIgnoreCase("ENGLISH")){
            if (sensorID2.endsWith("_TranslationSensor_English")){
            System.out.println("Calling addmonitor to subscribe French to English Translator");
        addSensorMonitor(sensorID,                              (SensorPolicy)sensorChangeInfo.get(sensorID).getPolicy(),
"display");
    }
}
…………………
}
```

The above segment of code shows how we call the addSensorMonitor() method when we confirm that we get are looking at a translation sensor with the ‘to’ language as English. The routine for subscribing to a sensor is specified in this addSensorMonitor() method.

In addition to the general signals the client receives, we can also receive control messages which are used to trigger the handleSensorControl() method. A look at the overall control signal handler is as follows

```
public void handleSensorControl(String commander, int controlMessageId,
        Serializable[] paramaters) {
    }
```

Or

```
public void handleSensorControl(String commander, int controlMessageId) {
      this.handleSensorControl(commander, controlMessageId, null);

    }
```

In general, for handling control messages, we pass the name of the sensor or entity requesting an explicit action to be taken along with the controlMessageId which is predefined to a number of commands.

Please Note Implementation of a Filter-Sensor could be viewed in TranslationChatSensor.java

Calling 3rd Party Computation Resources

It is also possible to call to some other computation resources which are from a third party source to help a sensor to go about its job. In this particular chat example, we make calls to the Google Translate© API. To go about making calls to 3rd party resources, we take the following steps:

1. We first start up by importing the JAR file consisting of the libraries of the service we might need to use into our maven dependencies. This can be done by mentioning the name of the JAR we are looking for in the POM file which we use to build or by uploading the file into our own repository and mentioning its name in the POM file. In our example, we have chosen the latter.
2. Once we have imported the JAR into our project, we simply call to the required methods as simple as we usually do in other projects. We provide an example of the calls we make in our case.
3.

Excerpts of the source code used for translation are shown below:

```
Translator translator = new Translator();
Language source = Language.valueOf(incomingLanguage);
Language destination = Language.valueOf(outgoingLanguage);
outgoingText = translator.translate(msg, source, destination);
```

In the above mentioned code block, we create a new object of the ‘Translator ‘class and we call its methods to actually translate text from one language to another.

As seen, it is very simple and can be used to call other services if needed also.

2.6 NaradaBroker Distribution

To setup a network of narada brokers, download the Core NaradaBrokering Software from http://www.naradabrokering.org or find them already present as a part of the project at %{SENSORCLOUD_HOME}\sensorcloud-middleware\src\main\java\cgl\narada. All related dependencies are imported from the maven repositories

Initially, to set up a single broker on a machine, one starts the *startBroker.sh* script present in %{SENSORCLOUD_HOME}/bin. Internally, the broker instantiates threads which listen to incoming connections from other brokers.

Brokers uses different ports for different transport level protocols that it supports. One can get all the information about these ports used by the broker from *BrokerConfiguration.txt* located in %{SENSORCLOUD_HOME}/config. In order to run two brokers on same machine, one has to change the ports used by the second broker, in other words you need to change *BrokerConfiguration.txt*.

Now, to set up a broker network, start the broker on another machine. Again, just run startBroker.sh on second machine. But, at this point we just have 2 brokers running on two machines without any connection. If you are facing some issues to run a broker, then there can be 2 reasons. Either you have already started a broker on the machine. In that case, just kill the process and start all over again. The second reason could some path variable is not set properly.

After having started brokers successfully on both machines, we need to connect these two brokers. On the second machine, say B, run brokerInteract.sh. Make sure you run brokerInteract.sh only after you have instantiated the broker on the same machine using *startBroker.sh* as *brokerInteract* by default uses the broker on localhost. Then you will be able to see '*Type h for help and Usage indicators*'. Here, we can type the command to connect the broker running on B to broker running on first machine, say A. The command to connect to another broker goes as follows.

C <IP_NEW_BROKER > <PORT> <PROTOCOL>,

c 10.0.0.3 5045 t.

The command can be interpreted as...you want your broker to connect 'c' to broker running on machine with IP address 10.0.0.3. 5045 determines the port used by transport layer service of the second broker. In other words, here we try to connect to A's broker using TCP connection and TCP connection of second broker uses 5045 port. You can establish connection between brokers using other protocols such as udp. In this case, your command would look like this.

c 10.0.0.3 3045 u.

How did we come to know about the UDP port? *BrokerConfiguration.txt* gave me this information. How did we come to know 'u' denotes our UDP service? Type h or have a look at the parser code for this

command in *BrokerNode.java*.

The connect command connects two brokers gives you a link id (Here, tcp://10.0.0.3:5045 in this case). This id is useful to assign a logical node address to broker running in machine B. The next step after connection is to get a logical address for broker running on B. type following command.

<NA> <Generated_Link_Address> 0

na tcp://10.0.0.3:5045 0

Here, the last parameter denotes address level. This parameter is useful when one wants to setup a hierarchical broker network. For a point to point connection like one in our example, the address level can be 0.This ends the connection set up between two brokers.

Now, to verify whether the connection is actually established, we can test it by downloading the NaradaBrokering-C++ bridge from http://www.naradabrokering.org/software.htm. After downloading the software, we need to create an executable test program called *pubsub*. Use the make tool and run it from src folder of the software download. Include certain standard library headers in the mentioned files in case we are faced with compile time errors. In case you face issues, try including all the necessary headers files (not many though) until 'make' is successful and it generates an executable called *pubsub* in the bin folder. Perform these steps mentioned above on both the machines, A and B. Run executable generated in last step with following command.

./pubsub 10.0.0.3 chat.

Here, 10.0.0.3 denotes the IP of the broker and chat is name of the topic. On the second machine B, run '*pubsub'* 10.0.0.3 chat. In this case , your *pubsub* client on B is using the broker running on machine A that has IP address 10.0.0.3. But, you can make it use its local broker by replacing 10.0.0.3 in the command with localhost. But remember that your broker should be connected to broker running on A using the steps mentioned in the document before trying out this command local broker. Type any message from one window and see if other client connected broker is able to see your message.

Steps

1.3) Configure the two compute nodes as an NB overlay network

1.3.1) Configure the first broker (master node, example ip:192.168.1.10).

a) open the ${SENSORCLOUD_HOME}/config/BrokerConfiguration.txt, change the ***AssignedAddress*** parameter to true

b) cd ${SENSORCLOUD_HOME}/bin;   chmod +x startBroker.sh

c) ./startBroker.sh   {start the first broker}

1.3.2) Configure the second broker (slave node, example ip:192.168.1.11).

a) open the ${SENSORCLOUD_HOME} /config/BrokerConfiguration.txt, change the ***AssignedAddress*** parameter to *false*

b) cd ${SENSORCLOUD_HOME} /bin;   chmod +x *.sh

c) ./startBroker.sh

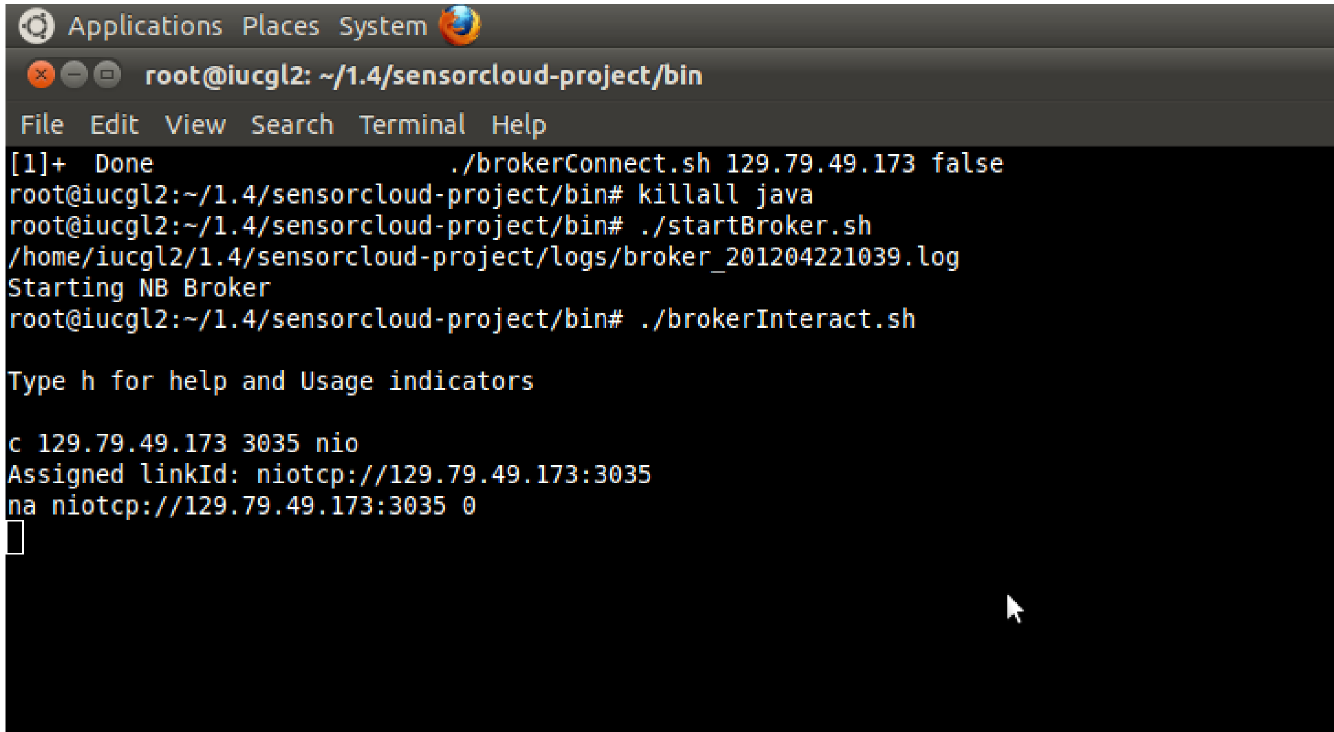
1.3.3) Require the overlay network address from the first broker for the second broker. You need to connect second broker to first broker, then require the overlay network address. Try out following commands from host running the second broker.

a)./brokerInteract.sh   //tools used to interact with other broker

b) c 192.168.1.10 5045 t

c) na tcp://192.168.1.10:5045 0

In the second machine, the commands go as follows:



On performing the above mentioned steps, we will be able to stand up independent Narada Brokers and have them interfacing with each other using the *brokerInteract* script.

While migrating to a more dynamic deployment, we come to see that brokerInteract is a script which keeps a live thread running which waits for the user to mention information of the domain which hosts the broker to which we have to connect to. Instead, we have a replicated though, slightly modified version of the BrokerNodeFront file which interfaces two brokers.

This changed script takes a single line of input in which we mention the IP address of the target Broker machine and if it should obtain its address from the broker it needs to connect to. This script file is called *brokerConnect.sh* and can be found in ${SENSORCLOUD_HOME}/bin.

The format for using this script is as follows:

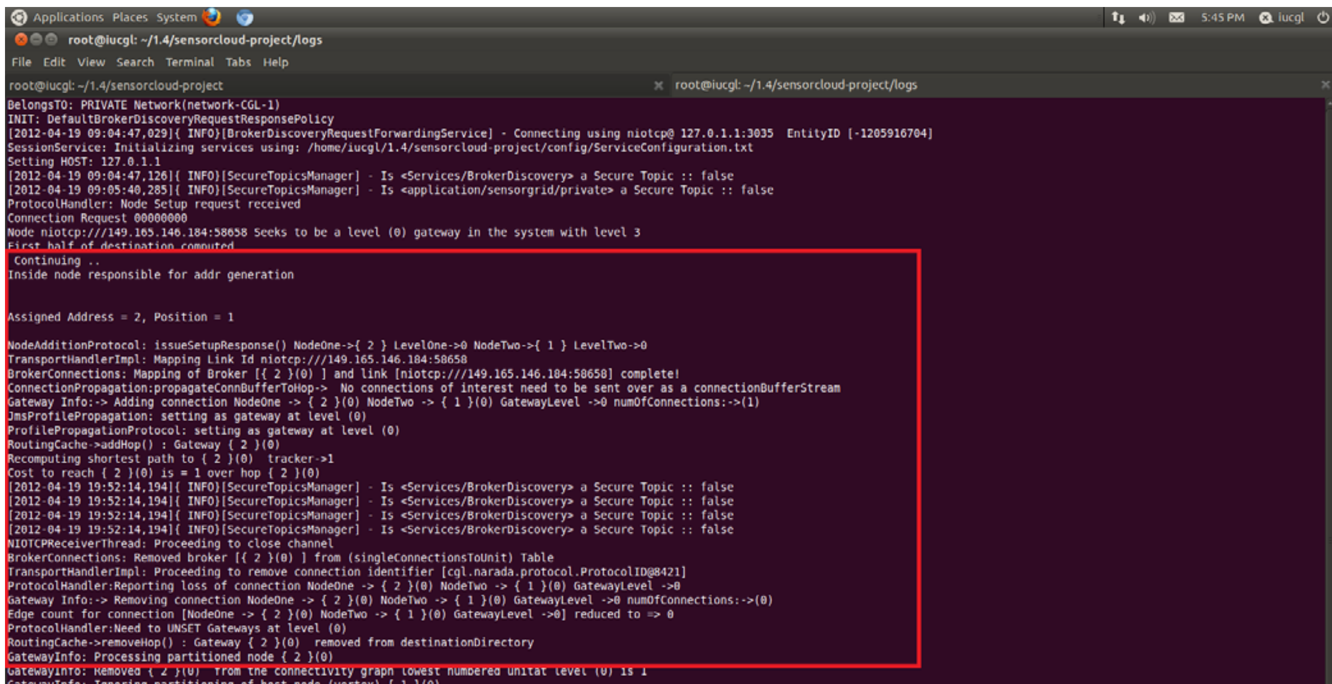
./brokerConnect <IP_TARGET_NB> <ASSIGN_ADDRESS_BOOL> &

./brokerConnect 129.79.49.173 false &

This third parameter of it being true or false is an indicator whether the broker which is just up is to behave as a standalone broker or get its logical address from another broker making it part of a network.

NOTE: Make it a point to have the *AssignedAddress* parameter in BrokerConfiguration.txt to false when setting up a network of brokers. Only the first broker is to have it set as true, Indicating that it is the only standalone broker and all the others are part of its network.

Upon successful connection to the first machine, we can check the logs for a prompt which indicates the success. This message can be found in *${SENSORCLOUD_HOME}\logs\broker_<timestamp>.log* and it looks as follows:



Any possible problems, like connection refused issues or ports in use problems can also be seen in these logs and corrected.

2.7 LDAP Security feature

What is LDAP?

LDAP, Lightweight Directory Access Protocol, is an Internet protocol that programs and other programs use to look up user authentication information from a server. In general LDAP is a set of protocols for accessing information directories. LDAP is based on the standards contained within the X.500 standard, but is significantly simpler. And unlike X.500, LDAP supports TCP/IP, which is necessary for any type of Internet access. Because it's a simpler version of X.500, LDAP is sometimes called X.500-lite.

Sensor Cloud Middleware has an authentication/authorization layer that uses LDAP as the user store. The user login and and publication/subscription rights are stored in an OpenLDAP server.

How about OpenLDAP?

OpenLDAP Software is a free, open source implementation of the Lightweight Directory Access Protocol (LDAP) developed by the OpenLDAP Project. It is released under its own BSD-style license called the OpenLDAP Public License. LDAP is a platform-independent protocol. Several common Linux distributions include OpenLDAP Software for LDAP support. The software also runs on BSD-variants, as well as AIX, Android, HP-UX, Mac OS X, Solaris, Microsoft Windows (NT and derivatives, e.g. 2000, XP, Vista, Windows 7, etc.), and z/OS.

This document describes the LDAP server configuration required; the security model employed by the

Sensor Cloud Middleware System and the implementation details of the model.

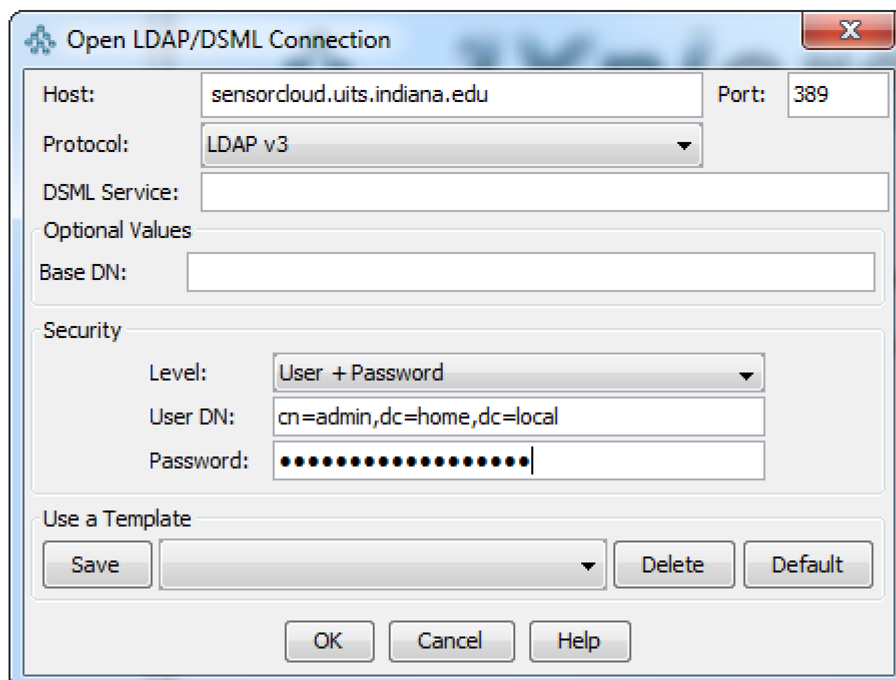
LDAP Server Configuration

For this prototype we have deployed our own LDAP server and populated it with some sample users and groups.

You can connect to our LDAP server with the following information:

Host: sensorcloud.uits.indiana.edu
Port: 389
User DN: cn=admin,dc=home,dc=local
Password: bobafet12bobafet12

You may also connect over SSL using port 636 and the authentication mode of SSL+User+Password. Please note our server only has a self-signed SSL certificate and you will have to agree to trust the SSL certificate.



The LDAP server address is *hardcoded* in:

com.anabas.sensorgrid.authentication.LDAPAuthentication.java

You may wish to modify this file to point to your own LDAP server and make whatever changes are necessary based on the schema of your directory. We will be able to assist you with those changes.

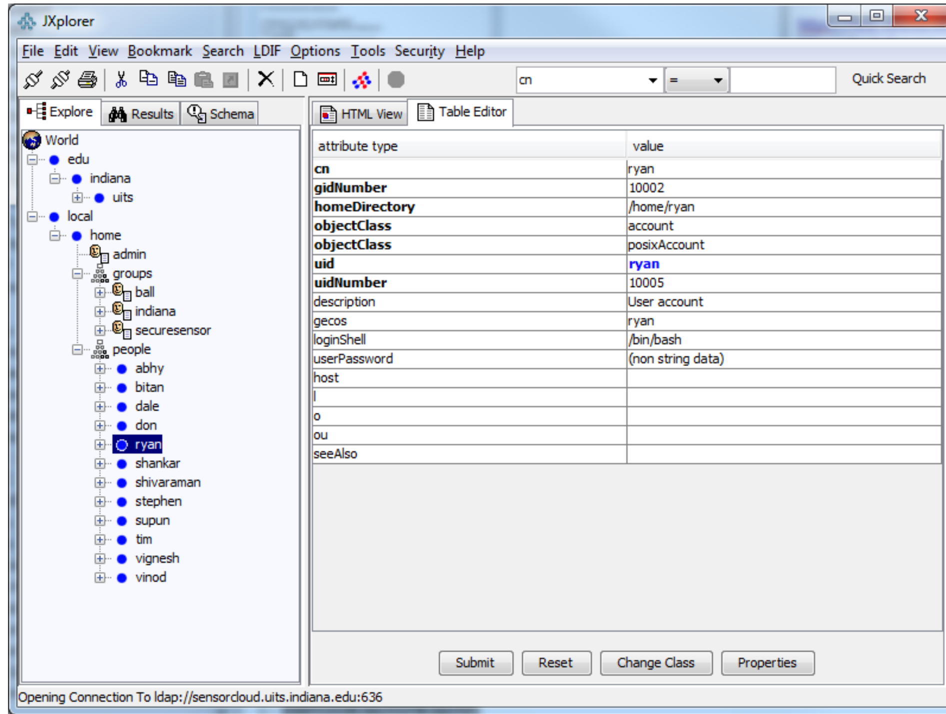
In our LDAP hierarchy user accounts are located at:

uid={user},ou=people,dc=home,dc=local

For the prototype we have created 12 user accounts:

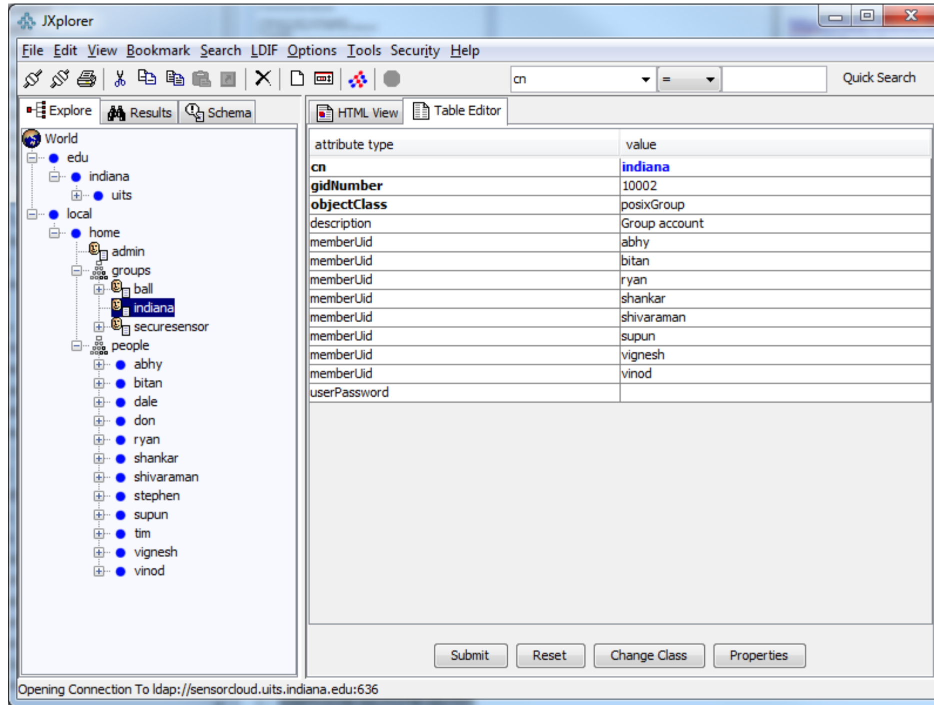
abhy, bitan, dale, don, ryan, shankar, shivaraman, stephen, tim, vignesh and vinod

We have set each account to use same password of ‘bobafet12bobafet12’.



Publisher roles (i.e. groups) are located at:

cn={role},ou=groups,dc=home,dc=local



We have created three publisher roles:

Role:	ball	indiana	securesensor
Members:	Dale, Don, Stephen and Tim	Abhy, Bitan, Ryan, Shankar, Shivaraman, Supun, Vignesh and Vinod	Ryan and Stephen

Security Model

When a sensor is deployed we collect a list of properties describing that sensor, these properties include:

Sensor ID	Authentication
Group ID	Publisher Role
Street Address	Data Encryption
City	Safety
State/Province	Reliability
Country	

Please note that 'Data Encryption', 'Safety' and 'Reliability' are not currently enabled in the middleware

Here is how sensor deployment now works:

- If the Authentication property is any value except 'yes' then
  - The system will set Authentication to 'no' and Publisher Role to 'anonymous'
  - No authentication will be done and the sensor will be deployed as an anonymous publisher
- If the Authentication property is 'yes' then
  - The system will authenticate the user against our LDAP server and determine if that user has the right to publish to the role set in Publisher Role
  - If that check fails the sensor will raise an authentication exception and will not be deployed

Examples:

GPS_1 GPS_Group "101 First St." Bloomington IN USA no no no no no

will not do authentication and publish the sensor as:

GPS_1 GPS_Group "101 First St." Bloomington IN USA no anonymous no no no

GPS_2 GPS_Group "101 First St." Bloomington IN USA yes securesensor no no no

will do an authentication checking to see if the current user is in the 'securesensor' group

GPS_3 GPS_Group "101 First St." Bloomington IN USA yes ball no no no

will do an authentication checking to see if the current user is in the 'ball' group

GPS_4 GPS_Group "101 First St." Bloomington IN USA no ball no no no

will not authentication and publish the sensor as:

GPS_4 GPS_Group "101 First St." Bloomington IN USA no anonymous no no no

Note if you are using the Grid Builder management tool to deploy a sensor to a remote machine both machines (local and remote) must have authorization to deploy to the requested sensor Publisher Role.

Here is how sensor clients now work:

Clients are able to subscribe to (and send control messages to):

- All sensors in the 'anonymous' role
- All sensors those Publisher Role matches a group that the current user belongs to

Examples:

The user 'tim' is in the 'ball' group. So a client launched by Tim can see all of the anonymous sensors and any authenticated sensors publishing in the 'ball' role.

The user 'stephen' is in the 'ball' and 'securesensor' groups. Therefore clients launched by Stephen can see authenticated sensors in these groups. (And all anonymous sensors too)

Implementation

For this prototype only the absolute minimum number of changes to the existing API was made. Therefore the current implementation of the security model described in this document won't require any changes to existing sensors, clients, or deployment scripts.

This implementation will also work well in a headless (i.e. ssh based) cloud-deployment scenario.

To get started a user must do the following steps.

1. Get the latest version of the code from https://sensorcloud.uits.indiana.edu/svn/SGX/trunk/1.4
2. Change the $SENSORCLOUD_HOME/keystore/PublicationCredentials.conf file to specify the user account you want to use
3. Optional: change com.anabas.sensorgrid.authentication.LDAPAuthentication.java to match your LDAP server.

2.8 Rest Easy

Introduction:

Cloud Sensor Rest Servlet API are programmed using concept of RestEasy API by JBoss. RestEasy is a JBoss project that provides various frameworks to help you build RESTful Web Services and RESTful Java applications. It is a fully certified and portable implementation of the JAX-RS specification. JAX-RS is a new JCP specification that provides a Java API for RESTful Web Services over the HTTP protocol.

RESTEasy can run in any Servlet container, but tighter integration with the JBoss Application Server is also available to make the user experience nicer in that environment. While JAX-RS is only a server-side specification, RESTEasy has innovated to bring JAX-RS to the client through the RESTEasy JAX-RS Client Framework. This client-side framework allows you to map outgoing HTTP requests to remote servers using JAX-RS annotations and interface proxies.

Cloud Sensor RestServlets provides a set of API’s using which an end user can get information about sensors deployed at Sensor Grid. This information will be discussed later in details.

Why Use RestServlet?

Using these rest services, client HTMLs can discover active sensors and fetch information about them. This restful service even will allow web based clients to get asynchronous sensor data. Clients should use our streaming web server to get real time video data.

2.8.1 Guide for SensorCloud Client End users

User Web Services

The first step to using the Cloud Sensor Rest Servlet is to initialize the RESTful services by hitting the following URL:

http://[ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/init/[ip_address]

Begin Client Initialization...

Figure 1 : Client Initialization

The above URL will check whether the RESTful client is already initialized or not. In case it is not initialized it will initialize it and post a message "Begin Client Initialization..." on the browser. In case it

is already initialized, the user will get "Client already Initialized" message on the browser.

After successful initialization, the user can use the following services to gather various information, as explained in the following parts of this document.

1. **Fetch list of Sensors with Id and Type:**
   The user can hit the following URL to get a list of all active sensors with their respective sensor Id and sensor types at the sensor grid.

   http:// [ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/getSensors

Figure 2: ScreenShot

Once this URL is hit, the user can see a list of all active sensors in XML format as shown above.

2. **Fetch Id and Type about a particular Sensor:**

   The user can hit the following URL to get information like sensor id and type about a particular

active Sensor.

http://[ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/getSensor/[Sensor_id]

Figure 3: Sensor by Id

If the requested sensor is active, the sensor details will get displayed as XML.
In case the sensor is not active, the user will get error message.

3. **Fetch list of properties of all active Sensors:**

The user can hit the following URL to get information about all properties of active sensors.

http:// [ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/getSensors/properties

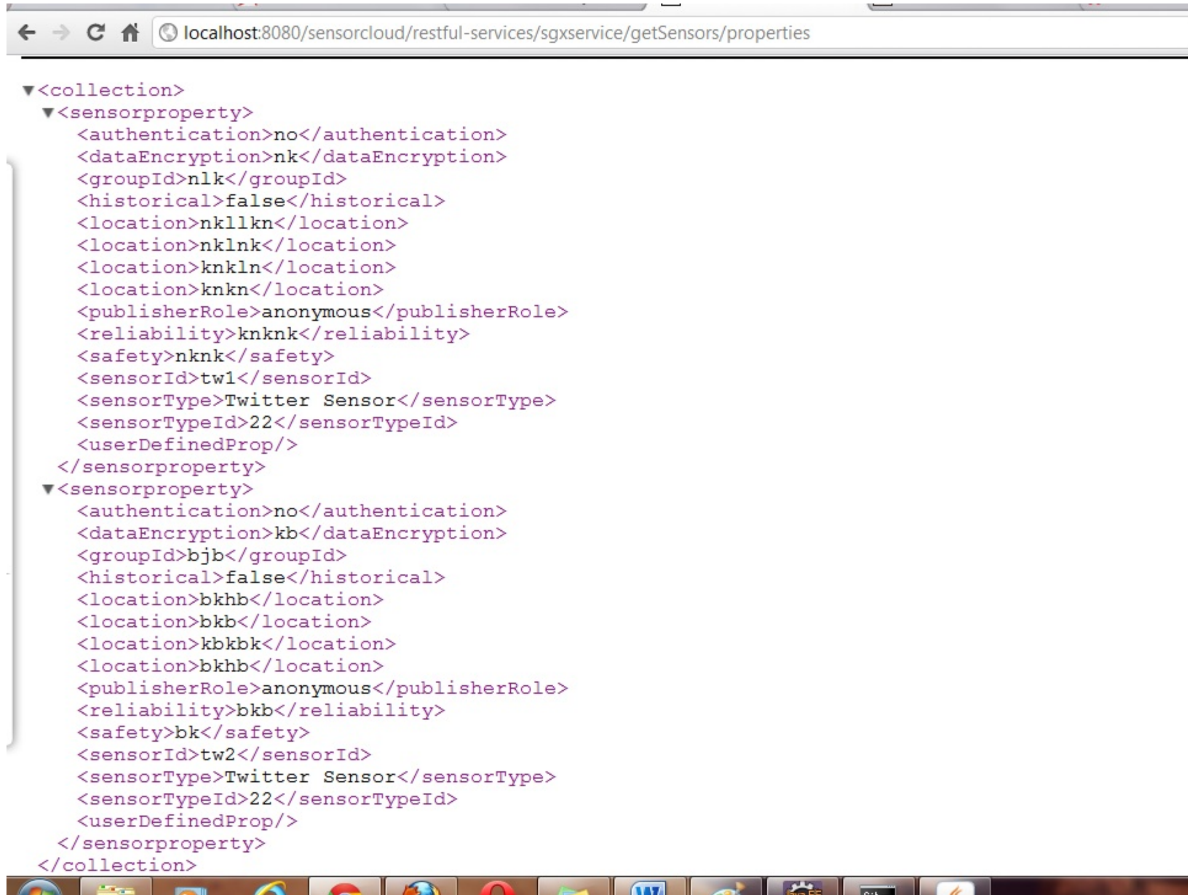


Figure 4 : Properties of all Sensors

Once the URL is hit, the user can see the information in XML format

4. Fetch list of properties for a particular active Sensor:

The user can hit the following URL to get information about all properties of a particular active sensor.

http:// [ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/getSensor/[Sensor_id]/properties

Figure 5: Properties for a particular Sensor

Once the URL is hit, the user can see the information in XML format

5. **Fetch a list of user defined properties for a particular active Sensor:**

   The user can hit the following URL to get information about all user defined properties of a particular active sensor.

   http:// [ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/getSensor/[Sensor_id]/user defined

   Once the URL is hit, the user can see the information in XML format.

6. **Send control to particular active Sensor:**

   The user can hit the following URL to send control information to a particular active sensor.

   http:// [ip_address or localhost]:8080/sensorcloud/restful-services/sgxservice/ putSensor/[Sensor-Id]/control/

Possible Error Messages and Causes

This section lists possible error codes and their cause which can be encountered by user when using these services.

1. **Error code = 500**

    Possible cause:

    a) The web service client has not been initialized. Refer to User Web services section.

Figure 6: Screenshot of Error 500

2. **Error code = 404**

    Possible cause:

    a) There is no active sensor with the Id requested.
    b) There is no property defined for the sensor.
    c) There is no user defined property for the sensor.

Figure 7: Screenshot of Error 404

Introduction to programming the web service

This section will give a programmer a basic intro on programming new web service or modifying existing web service:

Basic structure: The restservlet package consists of model package, SGXClientHandler interface, SGXClient class and SGXService class. The model package in turn consists of Online Sensor and Sensor Control classes. Let us see a brief summary of each of these components:

1. Online Sensor class: This class basically defines a Sensor by storing sensor type and sensor Id.
2. Sensor Control class: This class is used to define a control for a sensor and it defines a control by control Id and params(parameters).
3. SGXClientHandler : This interface ensures that every SGX client handler has function definitions for handling logging and connection loss.
4. SGXClient class : This class contains the business logic and attributes for an SGX client which would use SGX services.
5. SGXService class: This class contains the business logic and implementation logic for the different kind of web services which can be invoked by end user.

Notes on Annotations used while programming:

a) @GET – This annotation is used to implement HTTP GET request.

b) @Path – This annotation is used to specify the path that should be included in URL to access the service

c) @Produces – This annotation is used to specify the format of output after invoking the service.

d) @Consumes – This annotation is used to specify the format of input if any, required by the service while invoking it.

e) @XMLRootElement – This annotation is used to map a specific class to a XML root element

f) @XMLElement – This annotation is used to map a specific property to a XML element.

2.8.2 Guide for SensorCloud Client developers

Programming a sensor cloud RESTful Java client

In this section of the document, we will provide details on how to write client that uses the rest service offered by the sensor cloud. We will consider Twitter Sensor as a example and show how the rest easy client can be written that passes control messages and retrieves data from the sensor.

```
ClientRequest getSensorsReq = new ClientRequest(
        "http://localhost:8080/sensorcloud/restful-services/sgxservice/getSensors");
        getSensorsReq.accept("application/xml");
        ClientResponse<String> getSensorsResp = null;
        try {
                getSensorsResp = getSensorsReq.get(String.class);
        } catch (Exception e) {
                e.printStackTrace();
        }

        String getSensorsRespString = getSensorsResp.getEntity();
        String[]  sensorIds = getSensorIds(getSensorsRespString);
        String[] senTypes = getSensorTypes(getSensorsRespString);
```

Snippet 1

Snippet 1 depicts code snippet of a client the written using jboss rest easy client , we assume that one has already deployed sensorcloud rest service in a servlet container such as Apache Tomcat.

Before using the rest service to get data from the sensors , the rest service has to be initialized. Following url can be used to initialize the rest service. At the end of the url , we can see an ip address 127.0.0.1. This ip address can be replaced with ip address of the machine hosts servlet container. In this example , the servlet container is client is located on the same machine.

http://localhost:8080/sensorcloud/restful-services/sgxservice/init/127.0.0.1

The code snippet sends a requests to retrieve the list of active sensors. Snippet 2, depicts a response retrieved from the rest service. It contains the details of the sensor such as sensor id and sensor type. The sensor id is useful information as other services use this id. The sensor type can be helpful information if client is written for specific type of sensor such as twitter sensor in our example.

```
<collection>
    <sensor>
        <sensorId>
            f8cb8f24-8959-11e1-a858-77481f5cec79
        </sensorId>
```

```
        <sensorType>
            Twitter Sensor
        </sensorType>
    </sensor>
</collection>
```

Snippet 2

Next step would be to use this sensor id to retrieve data from the sensor. Snippet 3 depicts the code to retrieve the sensor data.

```
String getSensorDataUrl = "http://localhost:8080/sensorcloud/restful-services/sgxservice/getSensor/data/"+senId;

ClientRequest getSenDataReq = new ClientRequest(getSensorDataUrl);

getSenDataReq.accept("application/xml");
```

```
ClientResponse<String> getSenDataResp = null;
try {
	getSenDataResp = getSenDataReq.get(String.class);
} catch (Exception e) {
	e.printStackTrace();
}

String getSenDataRespString = getSenDataResp.getEntity();
```

Snippet 3

Here, the senId represents the id of the sensor. A typical repose from twitter sensor would look like one shown in Snippet 4. The data that we obtain from the sensor depends on the type the sensor. In other words the implementation of the SensorData class for the sensor and annotations.


```
<TwitterSensorData>
	<timestamp>
		1334756772925
	</timestamp>
	<feed>
		@wandy_onedy-done Gilr,.. RT @bhy_abhy @wandy_onedy : follback
	masbro ;)
```

```
        </feed>

    </TwitterSensorData>
```

Snippet 4.

In addition to retrieving data from the sensor, the sensor cloud rest service allows one to pass control messages to the sensor. Snippet 5 depicts a code snippet that shows how the put request can be passed. The <sensor control> is a message in xml format. It contains the control id and the

parameters that a client would like to pass to the sensor. After passing the control message to the sensor, client can get sensor data by the method described in last paragraph.

```
String putControlUrl = "http://localhost:8080/sensorcloud/restful-
services/sgxservice/putSensor/control/"+twSenId;

            ClientRequest putRequest = new ClientRequest(putControlUrl);

            StringBuilder putReqBody = new StringBuilder();


            putReqBody.append("<sensorcontrol>");

            putReqBody.append("<controlId>0</controlId>");
```

```
putReqBody.append("<params>"+input+"</params>");
putReqBody.append("</sensorcontrol>");

putRequest.body(MediaType.APPLICATION_XML,putReqBody.toString());
putRequest.accept(MediaType.TEXT_PLAIN);

ClientResponse<String> putReqResp = null;
try {
    putReqResp = putRequest.put();
} catch (Exception e) {
    e.printStackTrace();
}
```

Snippet 5.

Till now, we have seen methods to pass the control messages and get data from the sensor. There are also few other services that are offered. For instance, one shown in Snippet 6 can be used to get properties of a sensor. An example response is shown in Snippet 7.

```
http://localhost:8080/sensorcloud/restful -services/sgxservice/getSensor/properties/<senId>
```

Snippet 6.

For more information about the rest service , one can have look at the implementation located at

SENSORCLOUD_HOME/sensorcloud-restservlet/src/main/java/cgl/sensorcloud/resteasy/SGXService.java

```
<sensorproperty>
    <authentication>
        no
    </authentication>
    <dataEncryption>
        no
    </dataEncryption>
    <groupId>
        WebGroup
    </groupId>
    <historical>
        false
    </historical>
    <location>
        East
    </location>
    <location>
        Bloomington
    </location>
    <location>
```

```
			Indiana
		</location>
		<location>
			USA
		</location>
		<publisherRole>
			anonymous
		</publisherRole>
		<reliability>
			no
		</reliability>
		<safety>
			yes
		</safety>
		<sensorId>
			TwitterSensor
		</sensorId>
		<sensorType>
			Twitter Sensor
		</sensorType>
		<sensorTypeId>
			22
		</sensorTypeId>
		<userDefinedProp/>
</sensorproperty>
```


Snippet 7.

2.9 Streaming Web Server

The SGX Streaming Web Server demonstrates how real-time sensor data may be collected from the sensor grid and forwarded to web clients. In this specific case the Streaming Web server subscribes to video data from IPCameraSensor sensors and republishes their video to HTTP urls.

The SGX Streaming Web Server encodes the video using the WebM container, using the VP8 video and Vorbis audio codecs. As of this writing the WebM video format is supported as follows:

Browser	Version	VP8 (WebM)
Internet Explorer	9.0.2	Manual install
Internet Explorer Mobile	9.0	No

Mozilla Firefox	11.0	Yes
Google Chrome	18.0.1025.163 (Mac), 18.0.1025.162 (Linux and Windows)	Yes
Chromium	r47759	Yes
Android browser	2.3	Partial: No streaming support
Safari	5.1.5	Manual install
Opera	11.62	Yes
Konqueror	4.8.2	Yes

The SGX Streaming Web Server uses Xuggler (http://www.xuggle.com/xuggler/) for Video/Audio transcoding and Stream-m (http://code.google.com/p/stream-m/) as a light-weight WebM enabled web server.

To use the SGX Streaming Web Server:

1. Deploy a Sensor Cloud
2. Launch at least one IPCameraSensor
3. Edit the Streaming Server's configuration file
   - $SENSORCLOUD_HOME\conf\StreamingServer.conf
   - The 'server.port ' specifies the port number you want clients to connect to
   - The 'SGX Information' section specifies the location of the Sensor Cloud

TextPad - C:\dev\projects\SGX\1.4\sensorcloud-project\config\StreamingServer.conf

File Edit Search View Tools Macros Configure Window Help

StreamingServer.conf

```
#
# Sample configuration file
# Empty lines and lines starting with # and ; are ignored.
# Format: <key> = <value>
# Syntactic elements (words) can be separated by linear whitespace.
#

# server.port
# listening port
server.port = 8080

# SGX information
sgx.hostname = 127.0.0.1
sgx.port = 3035
sgx.protocol = niotcp

# stream.limit
# maximum number of clients for each stream
streams.limit = 100


# stream.<streamname>
# if defined then a stream can be started with this name
streams.first = true

# stream.<streamname>.password
# determines the password to accept the stream
streams.first.password = secret
```

Port you want to use

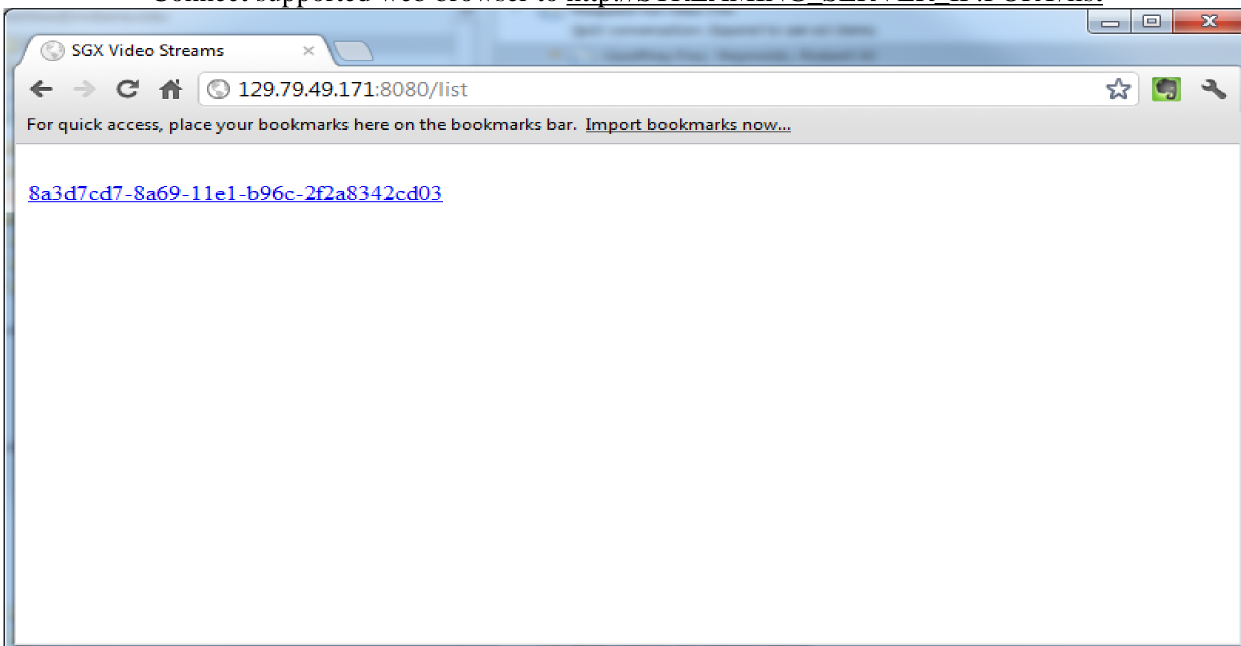
Address of Sensor Cloud

1 1 Read Ovr Block Sync Rec Caps

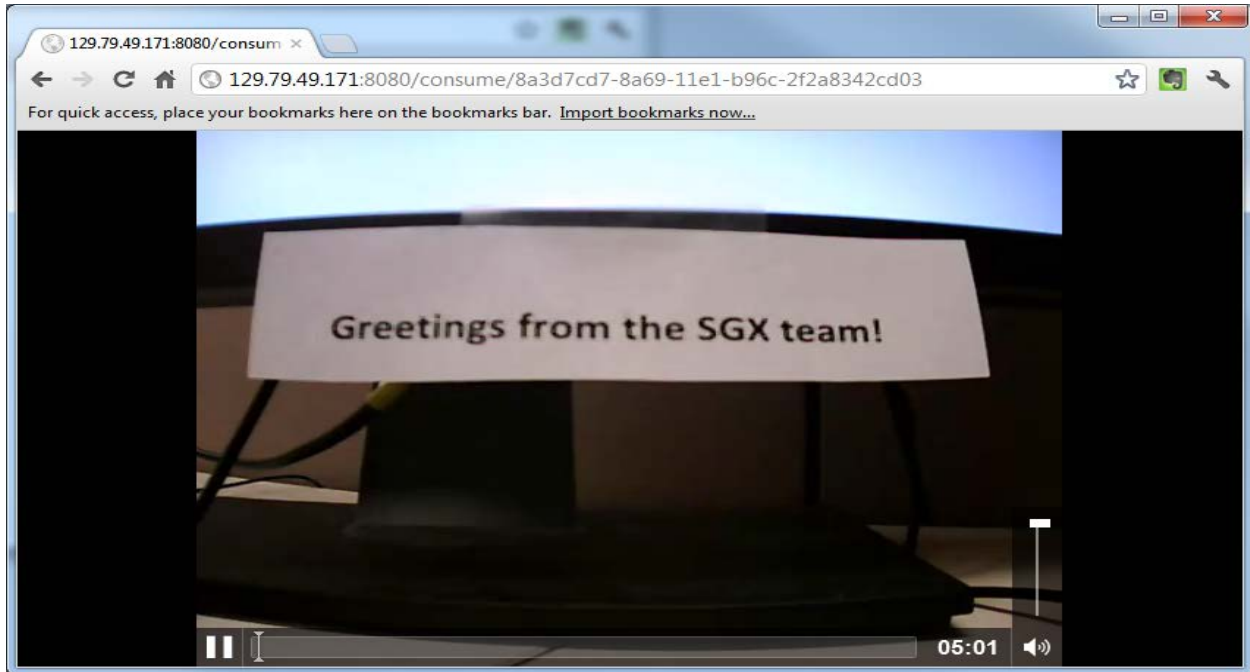
4. Launch the Streaming Web Server
    - %SENSORCLOUD_HOME%\scripts\prepareStreamingServer.bat on Windows
    - $SENSORCLOUD_HOME/bin/prepareStreamingServer.sh on Linux

```
C:\Windows\system32\cmd.exe - runStreamingServer  C:\dev\projects\SGX\1.4\sensorcloud-project\config\StreamingServer.conf
sensorgrid.loglevel:4
Log4J Config [config/log4j.properties] found.
[2012-04-19 18:09:41,568]< INFO>[SGLogic] - Constructor
[2012-04-19 18:09:41,568]< INFO>[SGLogic] - host: 127.0.0.1 port: 3035 transportType: niotcp
[2012-04-19 18:09:41,661]<DEBUG>[NaradaJMSBridge] - subscribe application/sensorgrid/public
[2012-04-19 18:09:41,661]< INFO>[NaradaJMSBridge] - ------------------------------> start JMS PING K
[2012-04-19 18:09:41,677]<DEBUG>[NaradaJMSBridge] - subscribe application/sensorgrid/public
[2012-04-19 18:09:41,677]<DEBUG>[NaradaJMSBridge] - subscribe application/sensorgrid/client/5cc7bbb0
Stating server on port: 8080
Number of sensors: 1
Policy is not null!!!
[2012-04-19 18:09:42,878]<DEBUG>[NaradaJMSBridge] - subscribe application/sensorgrid/sensordata/8a3d
Segment found
Info found


Subscibed to IPCamera: 8a3d7cd7-8a69-11e1-b96c-2f2a8342cd03
Segment found
Info found
[2012-04-19 18:09:42,878]<ERROR>[org.ffmpeg] - [mpeg4 @ 0p5311400] warning: first frame is no keyfra
Segment found
Info found
Tracks found
track no: 1, type: 1
track no: 2, type: 2
ALL'S WELL
[2012-04-19 18:09:43,970]<DEBUG>[SGLogic] - received message type: alive
[2012-04-19 18:09:43,970]<DEBUG>[SGLogic] - alive 1334873383970
[2012-04-19 18:09:54,095]<DEBUG>[SGLogic] - received message type: alive
[2012-04-19 18:09:54,095]<DEBUG>[SGLogic] - alive 1334873394095
```

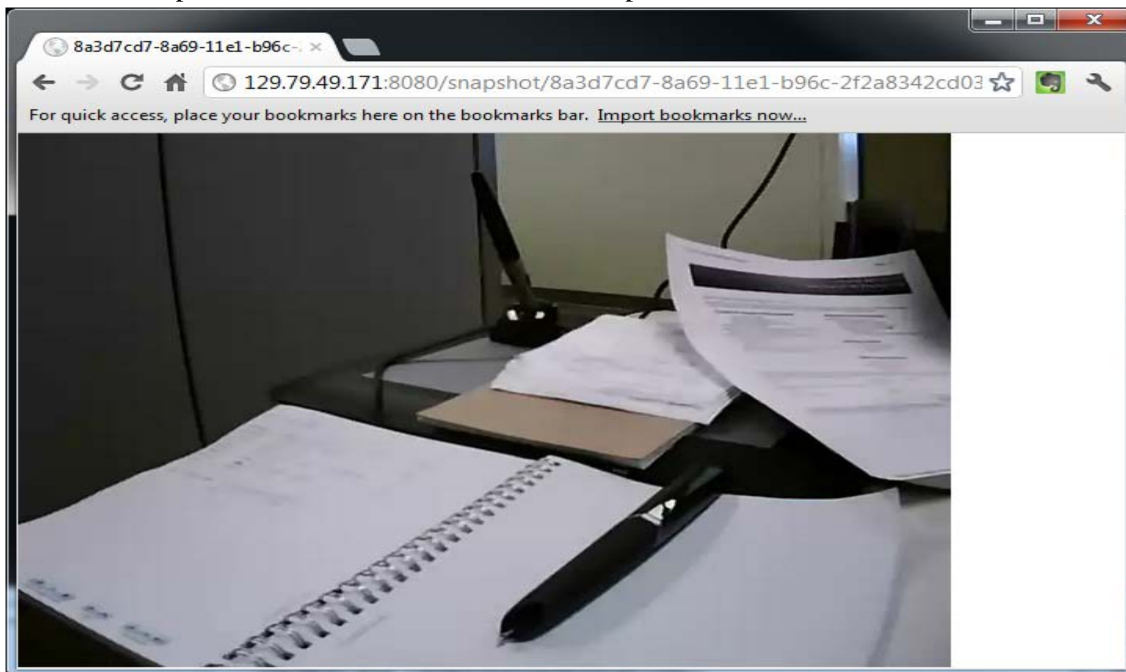
Connect supported web browser to http://STREAMING_SERVER_IP:PORT/list



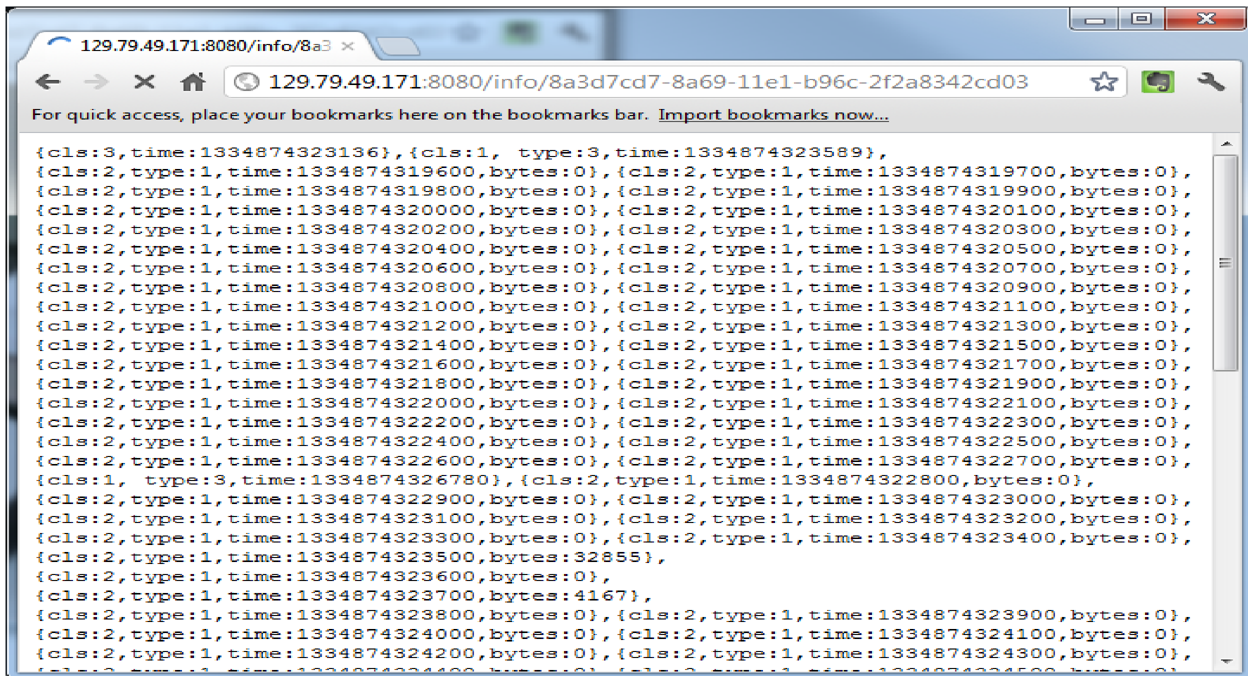
5. Clicking on a link will display the video feed. Video feeds are available at: http://SERVER_IP:SERVER_PORT/consume/SENSOR_ID



6. Still images are available at:
    - http://SERVER_IP:SERVER_PORT/snapshot/SENSOR_ID



7. A textual representations of the video data are available at:
    - http://SERVER_IP:SERVER_PORT/info/SENSOR_ID



2.10 Configuring dynamic deployment of domains

<u>The ROOT Node</u>:

Edit the Config file which contributes towards creating the ‘mgmtSystem’ as follows:

```
# -----------------------------------
# Config Entries for Fork Daemon
# -----------------------------------
## This string should be unique for different networks
## It is used to uniquely identify a Fork Daemon
ForkDaemon.UniqueString=ATGLOBAL-ISAAC
ForkDaemon.SharedRoot=TRUE

# -----------------------------------
# Config Entries for Root Bootstrap Node
# -----------------------------------
```

```
# The domain of the bootstrap program
ROOT-bootstrap.Level=/
ROOT-bootstrap.ForkProcessLocator=topic://FORKDAEMON/ATGLOBAL-ISAAC/:65535

# Number of registered subDomains
ROOT-bootstrap.NumOfRegisteredSubDomains=1

# Domain URI of subDomains and their locations

ROOT-bootstrap.RegisteredSubDomain_1=/ATGLOBAL-ISAAC
ROOT-bootstrap.RegisteredSubDomainForkProcess_1=topic://FORKDAEMON/ATGLOBAL-ISAAC/:65535

# Locaton of ForkProcess Daemons for spawning Managers
ROOT-bootstrap.NumberOfForkDaemons=0
# Locaton of Messaging Node
ROOT-bootstrap.NumberOfMessagingNodeDaemons=1
ROOT-bootstrap.MessagingNode_1=127.0.0.1

# ----------------------------------
# Config Entries for ISAAC - Bootstrap Node
# ----------------------------------
# The domain of the bootstrap program
ROOT_ATGLOBAL-ISAAC-bootstrap.Level=/ATGLOBAL-ISAAC

# Number of registered subDomains
ROOT_ATGLOBAL-ISAAC-bootstrap.NumOfRegisteredSubDomains=0

# Domain URI of subDomains and their locations

# Registry Locator
ROOT_ATGLOBAL-ISAAC-bootstrap.RegistryForkDaemon=topic://FORKDAEMON/ATGLOBAL-
ISAAC/:65535
#ROOT_ISAAC-bootstrap.RegistryPersistentStore=wscontext:ISAAC198

# Locaton of Messaging Node
ROOT_ATGLOBAL-ISAAC-bootstrap.NumberOfMessagingNodeDaemons=1
ROOT_ATGLOBAL-ISAAC-bootstrap.MessagingNode_1=127.0.0.1

# Locaton of ForkProcess Daemons for spawning Managers
ROOT_ATGLOBAL-ISAAC-bootstrap.NumberOfForkDaemons=1
ROOT_ATGLOBAL-ISAAC-bootstrap.ForkDaemon_1=topic://FORKDAEMON/ATGLOBAL-ISAAC/:65535

# ----------------------------------
# Config Entries for Service Adapter
# ----------------------------------
ServiceAdapter.NumOfMessagingNodes=1
ServiceAdapter.MessagingNode_1=127.0.0.1
ServiceAdapter.Level=/ATGLOBAL-ISAAC
# ----------------------------------
# Config Entries for User Console
# ----------------------------------
user.MessagingNode=127.0.0.1
```

```
user.MessagingNodePort=25050
user.MessagingNodeTransport=niotcp
user.RegistryMonitorInterval=30000
# ----------------------------------
# Config Entries for BootStrapService UI
# ----------------------------------
BootStrapServiceUI.MessagingNode=127.0.0.1
BootStrapServiceUI.MessagingNodePort=25050
BootStrapServiceUI.MessagingNodeTransport=niotcp
```

<u>Child Node Connecting to its Parent Node</u>:

Edit the Config file which contributes towards creating the 'mgmtSystem' as follows:

```
# ----------------------------------
# Config Entries for Fork Daemon
# ----------------------------------
## This string should be unique for different networks
## It is used to uniquely identify a Fork Daemon
ForkDaemon.UniqueString=ATGLOBAL-XPS/ATGLOBAL-ZEN
ForkDaemon.SharedRoot=FALSE
# ----------------------------------
# Config Entries for XPS - Bootstrap Node
# ----------------------------------
# The domain of the bootstrap program
ROOT_ATGLOBAL-ZEN-bootstrap.Level=/ATGLOBAL-XPS/ATGLOBAL-ZEN

# Number of registered subDomains
ROOT_ATGLOBAL-ZEN-bootstrap.NumOfRegisteredSubDomains=0

# Domain URI of subDomains and their locations
ROOT_ATGLOBAL-ZEN-bootstrap.RegisteredSubDomain_1=/ATGLOBAL-XPS/ATGLOBAL-ZEN
ROOT_ATGLOBAL-ZEN-
bootstrap.RegisteredSubDomainForkProcess_1=topic://FORKDAEMON/ATGLOBAL-XPS/ATGLOBAL-
ZEN/:65535

# Registry Locator
ROOT_ATGLOBAL-ZEN-bootstrap.RegistryForkDaemon=topic://FORKDAEMON/ATGLOBAL-
XPS/ATGLOBAL-ZEN/:65535
#ROOT_ATGLOBAL-ZEN-bootstrap.RegistryPersistentStore=wscontext:XPS199

# Locaton of Messaging Node
ROOT_ATGLOBAL-ZEN-bootstrap.NumberOfMessagingNodeDaemons=1
ROOT_ATGLOBAL-ZEN-bootstrap.MessagingNode_1=127.0.0.1

# Locaton of ForkProcess Daemons for spawning Managers
ROOT_ATGLOBAL-ZEN-bootstrap.NumberOfForkDaemons=1
ROOT_ATGLOBAL-ZEN-bootstrap.ForkDaemon_1=topic://FORKDAEMON/ATGLOBAL-XPS/ATGLOBAL-
ZEN/:65535
```

```
# -----------------------------------
# Config Entries for Service Adapter
# -----------------------------------
ServiceAdapter.NumOfMessagingNodes=1
ServiceAdapter.MessagingNode_1=127.0.0.1
ServiceAdapter.Level=/ATGLOBAL-XPS/ATGLOBAL-ZEN
# -----------------------------------
# Config Entries for User Console
# -----------------------------------
user.MessagingNode=127.0.0.1
user.MessagingNodePort=25050
user.MessagingNodeTransport=niotcp
user.RegistryMonitorInterval=30000
# -----------------------------------
# Config Entries for BootStrapService UI
# -----------------------------------
BootStrapServiceUI.MessagingNode=127.0.0.1
BootStrapServiceUI.MessagingNodePort=25050
BootStrapServiceUI.MessagingNodeTransport=niotcp
```

BrokerConfigurationExistingSGX

```
#################################################################
#  NaradaBrokering - Community Grid Labs. Indiana University.  #
#                                                               #
#  Broker Configuration Parameters                              #
#                                                               #
#  The "#" at the beginning of each line signifies comments.   #
#                                                               #
#                                                               #
#################################################################
#################################################################

#This is the Non Blocking TCP port to which the broker listens for connections.
NIOTCPBrokerPort=3035

#This is the TCP port to which the broker listens for connections.
TCPBrokerPort=5045

#This is the UDP port to which the broker listens for connections. It
#is a good idea to have this port number be identical to the TCP port.
#The UDP communication is used specifically for transient bytes, since
#there are no error corrections for UDP based communication.
UDPBrokerPort=3045

#This is the PTCP port to which the broker listens for connections.
PTCPBrokerPort=15045
PTCPStreamNumber=5
```

```
MulticastGroupHost=224.224.224.224
MulticastGroupPort=0

#This is the Non Blocking Thread pool TCP port to which the broker listens for #connections.
PoolTCPBrokerPort=6045

#This is the HTTP port to which the broker listens for connections.
HTTPSBrokerPort=7045

#This is the SSL port to which the broker listens for connections.
SSLBrokerPort=8045

#This is the HTTP port to which the broker listens for connections.
HTTPBrokerPort=9045

#This is the UP2P port to which the broker listens for connections.
UP2PBrokerPort=0
PeerID=peerA
RelayServerHost=gf7.ucs.indiana.edu
RelayServerPort=60055


#Indicates if Support for RTP should be included within the system
SupportRTP=no

#This specifies the limit on concurrent connections. Base it on the
#capabilities of the machine hosting the broker. This is also used by
#the broker locator to determine the best available broker.
ConcurrentConnectionLimit=3000



#If this is a stand alone node, this should be "true". If this broker node
#is intended to be the first node within a distributed setting this should
#be "true". If this node is to receive its address from another broker, this
#should be "false".
AssignedAddress=false

# Default Node address when AssignedAddress = true
NodeAddress=1,1,1,1



# This gives the Geographical / Institutional info about this broker
AboutThisBroker=CGL, Indiana University, Bloomington, IN, U.S.A.

# Comma seperated list of publicly known BDNs (listed in the order of preference)
#
BDNList=http://www.idonotexist.com,http://trex.ucs.indiana.edu:8080/BDN/servlet/BDN,http://www.gridserlocat
or.org/
# BDNList=http://trex.ucs.indiana.edu:8080/BDN/servlet/BDN
BDNList=
```

```
# Broker Discovery Request Response Policy
DiscoveryResponsePolicy=cgl.narada.discovery.broker.DefaultBrokerDiscoveryRequestResponsePolicy

# A String (or UUID) referring to the private broker network ID to which this broker belongs
# This value if missing OR * => this is a public broker
VirtualBrokerNetwork=network-CGL-1
# VirtualBrokerNetwork=*


# Locates the keystore to be used by the broker
BrokerKeyStore=keystore/NBSecurityTest.keys

# Maximum number of requests to store
MAXBrokerDiscoRequests=1000

# Info to connect to the existing NB Network
ConnectAddress=129.79.49.250
ConnectTransport=niotcp
ConnectPort=3035
```

---

existingNBMessageingNode

```
# -------------------------
# Prioritized Protocols
# -------------------------

PRIORITIZED_PROTOCOL_LIST.prioritizedProtocolList=niotcp,tcp,udp,http,https,ssl

# -------------------------
# Default Messaging Node properties
# -------------------------

DEFAULT_MESSAGING_NODE.NIOTCPBrokerPort=25050
DEFAULT_MESSAGING_NODE.TCPBrokerPort=25060
DEFAULT_MESSAGING_NODE.UDPBrokerPort=25070
DEFAULT_MESSAGING_NODE.HTTPBrokerPort=25600
DEFAULT_MESSAGING_NODE.HTTPSBrokerPort=25090
DEFAULT_MESSAGING_NODE.SSLBrokerPort=25080
DEFAULT_MESSAGING_NODE.PTCPBrokerPort=0
DEFAULT_MESSAGING_NODE.MulticastGroupPort=0
DEFAULT_MESSAGING_NODE.MulticastGroupHost=224.224.224.224
DEFAULT_MESSAGING_NODE.PoolTCPBrokerPort=0
DEFAULT_MESSAGING_NODE.PTCPStreamNumber=5
DEFAULT_MESSAGING_NODE.AssignedAddress=false
DEFAULT_MESSAGING_NODE.NodeAddress=1,1,1,1
DEFAULT_MESSAGING_NODE.VirtualBrokerNetwork=network-CGL-1
DEFAULT_MESSAGING_NODE.SupportRTP=no
DEFAULT_MESSAGING_NODE.BDNList=
DEFAULT_MESSAGING_NODE.ConcurrentConnectionLimit=3000
```

```
DEFAULT_MESSAGING_NODE.Discriminator=159.59.*
DEFAULT_MESSAGING_NODE.AboutThisBroker=Default Messaging Node
DEFAULT_MESSAGING_NODE.MAXBrokerDiscoRequests=1000
DEFAULT_MESSAGING_NODE.DiscoveryResponsePolicy=cgl.narada.discovery.broker.DefaultBrokerDiscoveryRequestResponsePolicy
DEFAULT_MESSAGING_NODE.BrokerKeyStore=keystore/NBSecurityTest.keys
DEFAULT_MESSAGING_NODE.BrokerTrustStore=../keystore/Broker.TRUSTSTORE
DEFAULT_MESSAGING_NODE.BrokerTrustStorePass=passpass

# These are required only if AssignedAddress is false
#DEFAULT_MESSAGING_NODE.ConnectAddress=64.151.140.115
#DEFAULT_MESSAGING_NODE.ConnectAddress=192.168.1.6
DEFAULT_MESSAGING_NODE.ConnectAddress=129.79.49.115
#DEFAULT_MESSAGING_NODE.ConnectAddress=72.44.37.62
#DEFAULT_MESSAGING_NODE.ConnectAddress=192.168.1.9
DEFAULT_MESSAGING_NODE.ConnectTransport=niotcp
#DEFAULT_MESSAGING_NODE.ConnectTransport=ssl
DEFAULT_MESSAGING_NODE.ConnectPort=25050
#DEFAULT_MESSAGING_NODE.ConnectPort=25080
```

---

standaloneMessageingNode

```
# -------------------------
# Prioritized Protocols
# -------------------------

PRIORITIZED_PROTOCOL_LIST.prioritizedProtocolList=niotcp,tcp,udp,http,https,ssl

# -------------------------
# Default Messaging Node properties
# -------------------------

DEFAULT_MESSAGING_NODE.NIOTCPBrokerPort=25050
DEFAULT_MESSAGING_NODE.TCPBrokerPort=25060
DEFAULT_MESSAGING_NODE.UDPBrokerPort=25070
DEFAULT_MESSAGING_NODE.HTTPBrokerPort=25600
DEFAULT_MESSAGING_NODE.HTTPSBrokerPort=25090
DEFAULT_MESSAGING_NODE.SSLBrokerPort=25080
DEFAULT_MESSAGING_NODE.PTCPBrokerPort=0
DEFAULT_MESSAGING_NODE.MulticastGroupPort=0
DEFAULT_MESSAGING_NODE.MulticastGroupHost=224.224.224.224
DEFAULT_MESSAGING_NODE.PoolTCPBrokerPort=0
DEFAULT_MESSAGING_NODE.PTCPStreamNumber=5
DEFAULT_MESSAGING_NODE.AssignedAddress=true
DEFAULT_MESSAGING_NODE.NodeAddress=1,1,1,1
DEFAULT_MESSAGING_NODE.VirtualBrokerNetwork=network-CGL-1
DEFAULT_MESSAGING_NODE.SupportRTP=no
DEFAULT_MESSAGING_NODE.BDNList=
DEFAULT_MESSAGING_NODE.ConcurrentConnectionLimit=3000
```

```
DEFAULT_MESSAGING_NODE.Discriminator=159.59.*
DEFAULT_MESSAGING_NODE.AboutThisBroker=Default Messaging Node
DEFAULT_MESSAGING_NODE.MAXBrokerDiscoRequests=1000
DEFAULT_MESSAGING_NODE.DiscoveryResponsePolicy=cgl.narada.discovery.broker.DefaultBrokerDiscoveryRequestResponsePolicy
DEFAULT_MESSAGING_NODE.BrokerKeyStore=keystore/NBSecurityTest.keys
DEFAULT_MESSAGING_NODE.BrokerTrustStore=../keystore/Broker.TRUSTSTORE
DEFAULT_MESSAGING_NODE.BrokerTrustStorePass=passpass

# These are required only if AssignedAddress is false
#DEFAULT_MESSAGING_NODE.ConnectAddress=64.151.140.115
#DEFAULT_MESSAGING_NODE.ConnectAddress=192.168.1.6
DEFAULT_MESSAGING_NODE.ConnectAddress=127.0.0.1
#DEFAULT_MESSAGING_NODE.ConnectAddress=72.44.37.62
#DEFAULT_MESSAGING_NODE.ConnectAddress=192.168.1.9
DEFAULT_MESSAGING_NODE.ConnectTransport=niotcp
#DEFAULT_MESSAGING_NODE.ConnectTransport=ssl
DEFAULT_MESSAGING_NODE.ConnectPort=25050
#DEFAULT_MESSAGING_NODE.ConnectPort=25080
```

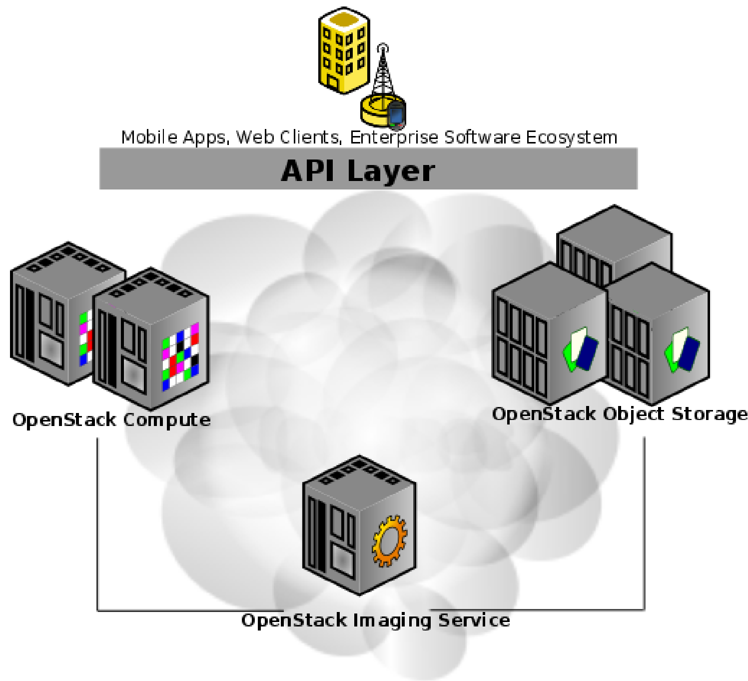
2.11 OpenStack Compute: Deployment and Overview

INTRODUCTION TO OPENSTACK

Openstack is project undertaken to provide scalable cloud computing software. Two of its main projects are Openstack compute and Openstack object storage. Compute increases computing power through Virtual Machines on multiple networks whereas Object Storage is a software that deals with a redundant scalable storage. We discuss Openstack compute in this manual and as already mentioned earlier, it is software that is used to manage a network of Virtual Machines (VM's) to make the entire system more scalable and redundant. It allows users to creating and running VM instances, creating and managing users and projects. Also work with the management of networks. This is an open source project made for supporting a variety of configurations and hypervisors.

Two of the most recent releases of the software are Diablo and Essex. Diablo, released July 2011 is the most recent and is being used. Diablo has an updated nova database schema and a few new packages including the glance object store and the swift object retrieval system. This manual assumes the

installation of Openstack Diablo. Essex is the more recent release in April 2012.



Openstack Components

The Openstack project consists of three main components, Compute, Object storage and Imaging service. Compute consists of a cloud controller, which is used to start up the VM's for the users and also to set up the network configurations that they might need to use. It is responsible for setting up the number of networks assigned to each project and so forth. Object storage is a system used as a large-scale redundant storage system which supports a number of fail-safe procedures like archiving or backup of data. It can also store secondary data and serve as a Content Delivery Network. The imaging service is an image storage system that provides image lookup and retrieval. It can be configured to use Object storage, S3 storage or using S3 storage with Object storage as an intermediate to S3.

A diagram of the presence of these three components and their relation to each other is shown below.

Openstack Compute

An introductory statement to Openstack compute has already been given earlier to make the function of the Compute project clear. The underlying Openstack project is called Nova and it provides a software that controls an Infrastructure as a service (IAAS) cloud platform. It does not include any virtualization software rather it specifies drivers to interact with the virtualization mechanisms already present.

Compute was meant to be used by many customers, developers and managers and so it supports a variety of user level access and privileges. The user management system of Compute uses a Role Based Access Control (RBAC) model which supports primarily five roles, those of a Cloud Administrator, IT Security, Project Manager, Network Administrator and a Developer. The Cloud administrator being the one who enjoys privileges of the complete system and the developer, which is the common user's default role. Every project created on compute was also separate from the others and form the basic organizational unit in Nova. The images, volumes, instances and certificates of one project are separated from the others. There are certain quota allocations like the availability of IP addresses, processor cores, number of instances and so on that limit the amount of resources dedicated to a project.

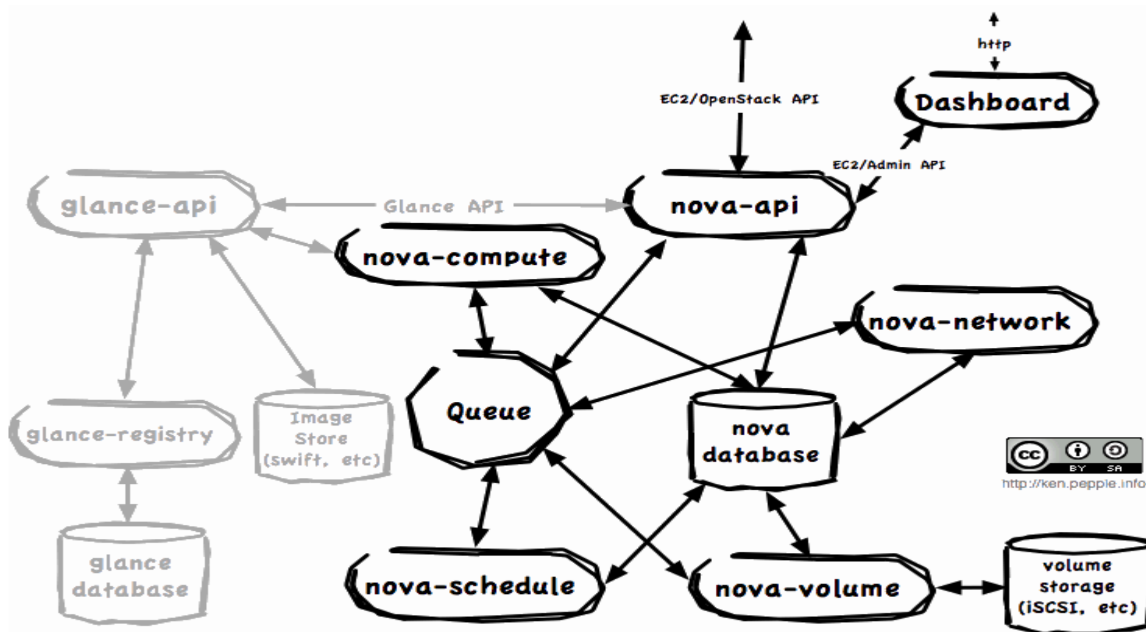
We load images on instances of a VM on Compute. An image is a file containing information about a virtual disk replicating information about a working system like an entire operating system. Compute uses Euca2ools command-line tools for managing the images. These images can be handles via the Openstack Imaging service called Glance or using the nova-objectstore service. The instances that host the images that we deploy is running inside a VM on our cloud environment. The host system must support virtualization as Compute does not come with its own. Some Openstack standards that are followed by Openstack are KVM, UML, XEN and QEMU.

Openstack compute consists of seven main components which together constitute the system's architecture. These components are:

- **Cloud Controller**: Global controller representing the state of the service and interacting with other components
- **API Server**: Provides the web service front end for the cloud controller
- **Compute Controller**: Provides the compute resources
- **Object Store component**: Provides storage facilities for images
- **Auth Manager**: Provides various user authorization and authentication
- **Network Controller**: Provides Virtual networks for the compute nodes to interact over the public network
- **Scheduler**: Selects the appropriate controller to host an instance

Compute is built as a share-nothing messaging based architecture. We run all major components on multiple servers. The controller communicates with the internal Object store using HTTP but it communicates to the volume and network controllers using an asynchronous Advanced Message Queue protocol (AMQP) to avoid blocking calls. The share-nothing policy is implemented by having redundant caches of vital information that can be accessed separately. Atomicity is hence enforced when required.

The logical architecture of Compute is shown below:



The Nova-API is the main module here, which is what the user actually interacts with. It is also referenced as the Cloud controller class and acts as an endpoint for most User queries from setting up the network to starting instances.

The Nova-scheduler is the module which is used to queue all instance requests. Whenever there are many concurrent requests from many servers, it determines which instance runs on which host and which order it has to run in. Certain user's instances might be given preferences compared to others based on

user level. We can also manually specify the scheduling algorithm which needs to be used.

The Nova-compute is the core module, which is used to start up Virtual Machine (VM) instances, and updates the status of the VM's in the database.

The Nova-volume module manages the various persistent stores of volume associated with the needed instances.

The Nova-network module makes the necessary network changes visible in the database and takes care of marking the bridge interface and so on.

The Glance modules is a separate set of modules used for retrieving the images and other objects. It has a separate API with which Nova-API interacts with to fetch images and other objects.

The Glance modules interface with the swift Image store and act as a mediator to retrieve these images.

Getting Openstack Set Up

System Specifics:

Hardware: OpenStack components are intended to run on standard hardware.

Operating System: OpenStack currently runs on Ubuntu and the large scale deployments running OpenStack run on Ubuntu 11.04 LTS, so deployment-level considerations tend to be Ubuntu-centric. Community members are testing installations of OpenStack Compute for CentOS and RHEL and documenting their efforts on the OpenStack wiki at wiki.openstack.org. Be aware that RHEL 6 is the most viable option (not 5.5) due to nested dependencies.

Networking: 1000 Mbps are suggested. For OpenStack Compute, networking is configured on multi-node installations between the physical machines on a single subnet. For networking between virtual machine instances, three network options are available: flat, DHCP, and VLAN.

Database: For OpenStack Compute, you need access to either a PostgreSQL or MySQL database, or you can install it as part of the OpenStack Compute installation process.

Permissions: You can install OpenStack Compute either as root or as a user with sudo permissions if you configure the sudoers file to enable all the permissions.

INSTALLATION PROCESS

The installation of openstack can be done manually or using a script. Using a script, the installation procedure is the simplest and the quickest. We will go through both modes of installation here. Installation can be performed in such a manner to run everything (controller and the compute services) on the same machine or can be done so as to host every service on different machines.

Manual Installation

Initially we need to have super-user permissions to run the following commands. It is essential to install the nova-services, euca2ools and all its dependencies before we move on. Initially, we set up a nova package repository so that we can upgrade nova easily later whenever needed.

```
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get autoremove
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:openstack-release/2011.3
sudo apt-get update
```

Next, we install the rabbitMQ messaging server for inter-controller communication and the python dependencies.

sudo apt-get install -y rabbitmq-server

When that is done, we now install the nova-services and its dependencies are installed automatically

```
apt-get install nova-volume nova-vncproxy nova-api nova-ajax-console-proxy
apt-get install nova-doc nova-scheduler nova-objectstore
apt-get install nova-network nova-compute
apt-get install glance
```

We finally install euca2ools and unzip

```
sudo apt-get install -y euca2ools unzip
```

Setting up MySQL on the Controller Node

Before we actually install MySQL we need to set environments with a ’pre-seed’ to set passwords and to bypass installation prompts.

Bash

```
MYSQL_PASS=nova
NOVA_PASS=notnova
cat <<MYSQL_PRESEED | debconf-set-selections
mysql-server-5.1 mysql-server/root_password password $MYSQL_PASS
mysql-server-5.1 mysql-server/root_password_again password $MYSQL_PASS
mysql-server-5.1 mysql-server/start_on_boot boolean true
MYSQL_PRESEED
```

We install MySQL next using sudo apt-get install -y mysql-server

The conf files of MySQL has to be edited to change the bind address from localhost to any address.

```
sudo sed -i 's/127.0.0.1/0.0.0.0/g' /etc/mysql/my.cnf
sudo service mysql restart
```

Next, we create the Nova database, grant it access privileges and set the password for access.

sudo mysql -uroot -p$MYSQL_PASS -e 'CREATE DATABASE nova;'

*sudo mysql -uroot -p$MYSQL_PASS -e "GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION;"*

sudo mysql -uroot -p$MYSQL_PASS -e "SET PASSWORD FOR 'root'@'%' = PASSWORD('$MYSQL_PASS');"

Setting up the Compute Node

Configuring the compute nodes involves setting up the nova.conf file and using nova-manage commands to set up the projects, project networks and the user roles.

We start up by changing the flags in the nova.conf file. This is a file that does not read white spaces and comments. This is the main file which is looked up for any information. The default file is already present in /etc/nova/ with a few default flags like the following:

--daemonize=1
--dhcpbridge_flagfile=/etc/nova/nova.conf
--dhcpbridge=/usr/bin/nova-dhcpbridge
--logdir=/var/log/nova
--state_path=/var/lib/nova

Other than these default flags, it is necessary to define certain other flags for the system to get the information it needs. A detailed description of available flags is found by running /bin/nova-api –help. A table of all the mandatory flags is shown below.

Flag	Description
--sql_connection	IP address; Location of OpenStack Compute SQL database
--s3_host	IP address; Location where OpenStack Compute is hosting the objectstore service, which will contain the virtual machine images and buckets
--rabbit_host	IP address; Location of OpenStack Compute SQL database
--cc_host	IP address; Location where the nova-api service runs
--ec2_url	HTTP URL; Location to interface nova-api. Example: http://184.106.239.134:8773/services/Cloud

--network_manager	Configures how your controller will communicate with additional OpenStack Compute nodes and virtual machines. Options: • nova.network.manager.FlatManager Simple, non-VLAN networking • nova.network.manager.FlatDHCPManager Flat networking with DHCP • nova.network.manager.VlanManager VLAN networking with DHCP; This is the Default if no network manager is defined here in nova.conf.
--fixed_range	IP address/range; Network prefix for the IP network that all the projects for future VM guests reside on. Example: 192.168.0.0/12
--network_size	Number value; Number of IP addresses to use for VM guests across all projects.

A complete usable nova.conf file excerpt is shown below. This is for a configuration file with the compute and the controller nodes on the same system

--dhcpbridge_flagfile=/etc/nova/nova.conf
--dhcpbridge=/usr/bin/nova-dhcpbridge
--logdir=/var/log/nova
--state_path=/var/lib/nova
--verbose
--s3_host=129.79.49.115
--rabbit_host=129.79.49.115
--cc_host=129.79.49.115
--ec2_url=http:// 129.79.49.115:8773/services/Cloud
--fixed_range=10.0.0.0/12
--network_size=8
--FAKE_subdomain=ec2
--routing_source_ip=129.79.49.115
--sql_connection=mysql://root:nova@129.79.49.115/nova
--network_manager=nova.network.manager.FlatDHCPManager
--flat_network_dhcp_start=10.0.0.2
--flat_network_bridge=br100
--flat_interface=eth2
--flat_injected=False
--public_interface=eth

A similar Nova config file for a working FlatDHCP Networking network is shown below:

```
--dhcpbridge_flagfile=/etc/nova/nova.conf
--dhcpbridge=/usr/bin/nova-dhcpbridge
--logdir=/var/log/nova
--state_path=/var/lib/nova
--lock_path=/var/lock/nova
--flagfile=/etc/nova/nova-compute.conf
--verbose
--sql_connection=mysql://novaUSER:novaDBsekret@127.0.0.1/nova
--network_manager=nova.network.manager.FlatDHCPManager
--flat_network_bridge=br100
--flat_injected=False
--flat_interface=eth0
--public_interface=eth1
--vncproxy_url=http://129.79.49.66:6080
--daemonize=1
--rabbit_host=129.79.49.66
--osapi_host=129.79.49.66
--ec2_host=129.79.49.66
--image_service=nova.image.glance.GlanceImageService
--glance_api_servers=129.79.49.66:9292
--use_syslog
--use_deprecated_auth
```

Next, we create a nova group to set file permissions as this file consists of our MySQL password.

```
sudo addgroup nova

chown -R root:nova /etc/nova

chmod 640 /etc/nova/nova.conf
```

It’s always good to restart all the Nova-services once we make any changes to the config file. There are 6 services to restart namely:

restart libvirt-bin;

restart nova-network;

restart nova-compute;

restart nova-api;

restart nova-objectstore;

restart nova-scheduler

Once this is done, we use the nova-manage commands to set up the database schema, users, projects and the project network as follows:

/usr/bin/nova-manage db sync

/usr/bin/nova-manage user admin <user_name>

/usr/bin/nova-manage project create <project_name> <user_name>

/usr/bin/nova-manage network create <project-network> <number-of-networks-in-project> <IPs in project>

Certain examples of these commands are:

/usr/bin/nova-manage db sync

/usr/bin/nova-manage user admin ADMIN

/usr/bin/nova-manage project create SGX ADMIN

/usr/bin/nova-manage network create 10.0.0.0/24 1 255

In this example, the number of IP's are /24 as it is within the /12 range mentioned in the nova.conf file.

We can also create and allocate specific public IP's using commands from the nova-manage API.

The following commands are used to create public IP's, list them and delete them respectively

nova-manage floating create hostname CIDR_Range

nova-manage floating list

nova-manage floating delete CIDR_Range

Once the floating point addresses are created, we need to obtain an address and then associate that allocated address to the required instance. This is done using the following euca-commands

euca-allocate-address

129.79.49.65 → allocated_address

euca-associate-address –i instance_number allocated_address

Creating certifications

Once the users have been created, it is necessary to create credentials for that particular user and project. This is done using the nova-manage command to generate the zipfile containing the credentials for the project. We have these credentials in /root/creds as follows:

mkdir –p /root/creds

/usr/bin/python /usr/bin/nova-manage project zipfile $NOVA_PROJECT $NOVA_PROJECT_USER /root/creds/novacreds.zip

A warning message "No vpn data for project <project_name>" can be ignored safely depending on the networking configuration.

We then unzip the file and add the file to our environment as shown:

unzip /root/creds/novacreds.zip -d /root/creds/

cat /root/creds/novarc >> ~/.bashrc

source ~/.bashrc

NETWORK CONFIGURATION

There are three types of network configurations that can be set using the –network_manager flag in nova.conf. These modes are

- Flat Networking
- FlatDHCP Networking
- VLAN Networking

Flat Networking

This mode is selected when we set the Network Manager to nova.network.manager.FlatManager. In such a mode compute needs to use a bridge interface. By default the interface is assumed to be br100 and is stored in the database. We need to edit the network interfaces configuration to add the unused interface eth0 to the bridge as follows

```
< begin /etc/network/interfaces >
# The loopback network interface
auto lo
iface lo inet loopback

# Networking for OpenStack Compute
auto br100

iface br100 inet dhcp
   bridge_ports       eth0
   bridge_stp         off
   bridge_maxwait   0
   bridge_fd          0
< end /etc/network/interfaces >
```

We now restart using sudo /etc/init.d/networking restart to apply the changes. No other changes needs to be made in the nova.conf file as it already has the information it needs.

FlatDHCP Networking

In this networking mode, we do not use VLAN's but we create our own bridge. We need an interface that is free and does not have an associated IP address. We can simply tell the network manager to bridge into the interface by specifying the –flat_interface flag in the flag file to the interface we need. The network host will automatically add the gateway ip to this bridge. You can also add the interface to br100 manually and not set flat_interface. If so, we edit nova.conf to have the following lines:

--dhcpbridge_flagfile=/etc/nova/nova.conf
--dhcpbridge=/usr/bin/nova-dhcpbridge
--network_manager=nova.network.manager.FlatDHCPManager
--flat_network_dhcp_start=10.0.0.2
--flat_interface=eth2
--flat_injected=False
--public_interface=eth0

Once that is done, we are all set.

VLAN Networking

This is the default networking mode in compute and is taken if we do net explicitely include the –network_manager flag in nova.conf. For use of this mode, we need to make sure the bridge compute creates is integrated to our network and we have the necessary hardware components to support VLAN tagging.

To allow users access instances in their projects, a special VPN instance called cloudpipe has to be created. This image is a Linux instance with openvpn installed. It needs a simple script to grab user data from the metadata server, b64 decode it into a zip file, and run the autorun.sh script from inside the zip. The autorun script should configure and run openvpn to run using the data from Compute.

For certificate management, it is also useful to have a cron script that will periodically download the metadata and copy the new Certificate Revocation List (CRL). This will keep revoked users from connecting and disconnects any users that are connected with revoked certificates when their connection is re-negotiated (every hour). You set the --use_project_ca flag in nova.conf for cloudpipes to work securely so that each project has its own Certificate Authority (CA).

Scripted Installation

Scripted installation is much simpler and quicker compared to the manual process, we need to grab the script from:

git://github.com/cloudbuilders/devstack.git
and just start the installation using:

cd devstack;
./stack.sh

and follow the stages of installation. A better explanation of what exactly happens in this script can be found at http://devstack.org/stack.sh.html

We copy the nova.conf file from the cloud controller node next to the compute node. After the installation we must still source the novarc file to accept the generated credentials. Some details like the following needs to be provided during the stages of the installation.

- MySQL password
- The default S3 IP.
- The RabbitMQ host IP
- The Cloud controller IP
- The MySQL IP

The installation script also runs us through the process of creating the new user, project association and the network addresses associated to the project.

RUNNING OPENSTACK COMPUTE

Starting and Deleting Instances

Once we have the required image that we need to publish to compute, we use the uec-publish-tarball command to get it done. We can get the working images of any Ubuntu release from http://uec-images.ubuntu.com/. Once we have the image to upload, we do so using the following command.

uec-publish-tarball $path_to_image [bucket-name] [hardware-arch]

We get three references, emi, eri, eki values. We use the emi value of the images while starting instances. We now need a public key to connect to an image. You might need to create and source credentials as a key pair using euca2ools commands as follows:

euca-add-keypair mykey > mykey.priv

chmod 0600 mykey.priv

We then create instances for the image using the euca-run-instances command as follows:

euca-describe-images

euca-run-instances $emi -k mykey -t m1.tiny

...OUTPUT...

RESERVATION r-1jj2a80v proj_name default

INSTANCE i-00000001 ami-00000002 10.0.0.2 10.0.0.2 building mykey (proj_name, SIX) 0 m1.tiny 2011-08-18T21:06:03Z nova aki-00000001 ami-00000000

...OUTPUT...

Once the status of the instance goes from untarring to scheduling to launching to running openstack we are ready to ssh into the instance using the following command

ssh ubuntu@$ipaddress

Where IPaddress is the one assigned to the instance when created.

To delete an instance we use the following command

euca-terminate-instances $instanceid

CREATING CUSTOM UEC IMAGES

We can create edit a standard UEC Image for us to run our own code. It involves the following steps:

- Download the required Ubuntu version's UEC image from http://uec-images.ubuntu.com/.
- Unarchive the tar.gz image to actually get the .img file we need to edit.
- We now need to expand the .img file as it is limited to a certain size and our implementation requires us t have certain extra software.
- This is done by first running the command fsck.ext3 imageName.img to change its file format followed by resize2fs imageName.img SIZE where SIZE is the required size you need it resized it to.
- Now, we need to mount this image to add the required files. This is done by using mount –o loop imageName.img mountDir where mountDir is any directory on which the image is mounted.
- Now mountDir denotes the file structure of the image. We copy the required software to run on the instance. In our case, the SGX code checked out from https://sensorcloud.uits.indiana.edu/svn/SGX/trunk/1.3
- We also copy a version of Apache Maven 3.0.3 and an appropriate version of JDK to the appropriate location in the image.
- Once that is done, we need to edit the .bashrc file of the root user to source the environment variables SENSORCLOUD_HOME, JAVA_HOME and M2_HOME to the appropriate locations to which the SGX code and Maven 3.0.3 are copied to.
- We can checkout the code from repository and install Java using the default package manager once we get into the instances by creating a script to perform the mentioned task.
- Once these changes are made, we need to unmount the image using the command umount –l mountDir.

- Once that is done, we need to zip back the .img file and its other accessories of the UEC image back into the .tar archive.
- Once we do that, we are ready to publish this image on to the grid using uec-publish-tarball as mentioned in the 'Starting Instances' section.
- Well established methods of creating images off running instances can also be found at:
    - http://wiki.openstack.org/CreatingRHELImages
    - http://open.eucalyptus.com/participate/wiki/creating-image-existing-vm-centos
    - http://kb.iu.edu/data/bbsn.html
- The above mentioned guides primarily consider RHEL and Eucalyptus but can be easily extended to cover Ubuntu and Openstack as both of them deal with using the same euca2ools API.
- Main stages of the image creation include, creating the Kernel Image and registering it, creating the Ramdisk image and registering it and creating the actual image and registering it with the previously created Ramdisk and Kernel images.

ISSUES AND FAQ's

-1-

Errors when setting up the project network using nova-manage

These errors are mostly because the range of IP's provided in this command exceeds the number of assigned IP's in the nova.conf file. Reduce the number of IP's in this command or increase the range of assigned IP's in nova.conf.

The nova-manage service assumes that the first IP address is your network , the 2nd is your gateway and the broadcast is the last IP in the range defined. If not, it is required to edit the nova database's 'networks' table.

When using some other networking scheme rather than flat mode, it is necessary to mark one of the networks already defined as a bridge so that compute knows that a bridge exists.

-2-

Access forbidden 403, 401 errors

These are the errors usually which show up because there is a credential problem with the project. Occur

when we try to perform any euca-XXX command. Through current installation methods, there are basically two ways to get the novarc file. The manual method requires getting it from within a project zipfile, and the scripted method just generates novarc out of the project zip file and sources it for you. If you do the manual method through a zip file, then the following novarc alone, you end up losing the credentials that are tied to the user you created with nova-manage in the steps before. When you run nova-api the first time, it generates the certificate authority information, including openssl.cnf. If it gets started out of order, you may not be able to create your zip file. Once your CA information is available, you should be able to go back to nova-manage to create your zipfile.

-3-

Instance startup errors

There are times when instances are scheduling indefinitely, or startup and shutdown immediately or simply stay ‘launching’ forever. These can be cause due to a number of reasons including bad networking settings or credentials. One way of finding out what the kind of problem we face is to check console output of an instance using euca-get-console-output <instance ID> to check the status of the instance or to simply check the nova-api.log in/var/logs/nova/

-4-

Unable to ping or ssh instances that are running

There are a number of reasons for this case to occur. One of the usual reasons is because we have not yet granted access permissions to access ports for ssh or pinging. Use the ‘euca-authorize’ command to enable access. Below, you will find the commands to allow ‘ping’ and ‘ssh’ to your VMs:

euca-authorize -P icmp -t -1:-1 default

euca-authorize -P tcp -p 22 default

Another common issue is you cannot ping or SSH your instances after issuing the ‘euca-authorize’ commands. Something to look at is the amount of ‘dnsmasq’ processes that are running. If you have a

running instance, check to see that TWO 'dnsmasq' processes are running. If not, perform the following: killall dnsmasq service nova-network restart

Some other cases might also occur. A possible case can be that the instance might be waiting for a response from a metadata server by default that it does not receive. Such cases can be identified from the console output of the instance. Usually metadata forwarding is done by the gateway. In flat mode, we must do it manually so requests should be forwarded to the api server. A simpler solution would be to try another networking mode like flatDHCP.

It usually helps restarting all the related services once there is a problem. Specifics can be obtained from the logs in /var/logs/nova/.

-5-

Network host issues

In certain cases, faulty deployment of the instances which might not show in the euca-get-console-output command might lead to a certain 'Destination host unavailable' when we ping the instances or the 'No route to host' when we try to ssh into a host which is not ready. These problems are fixed by re-bundling the image to the server and trying again. Most likely caused by a small glitch while bundling. This can be verified using the euca-describe-images command and looking for the required upload bucket to confirm that the status of the image we need to deploy is 'available' and not 'untarring'. These are most likely caused when the nova services are restarted immediately after the new image is uploaded without giving the system any time to actually register the image internally.

-6-

Euca-tools freezing issues once an image is registered

In most cases, after registering the image, the system will take a while to respond to euca-commands. Restarting the nova-services before it registers the image will cause its state to remain in untarring when we check using 'euca-describe-images'. Such images will not be able to startup.

4. Performance Metrics

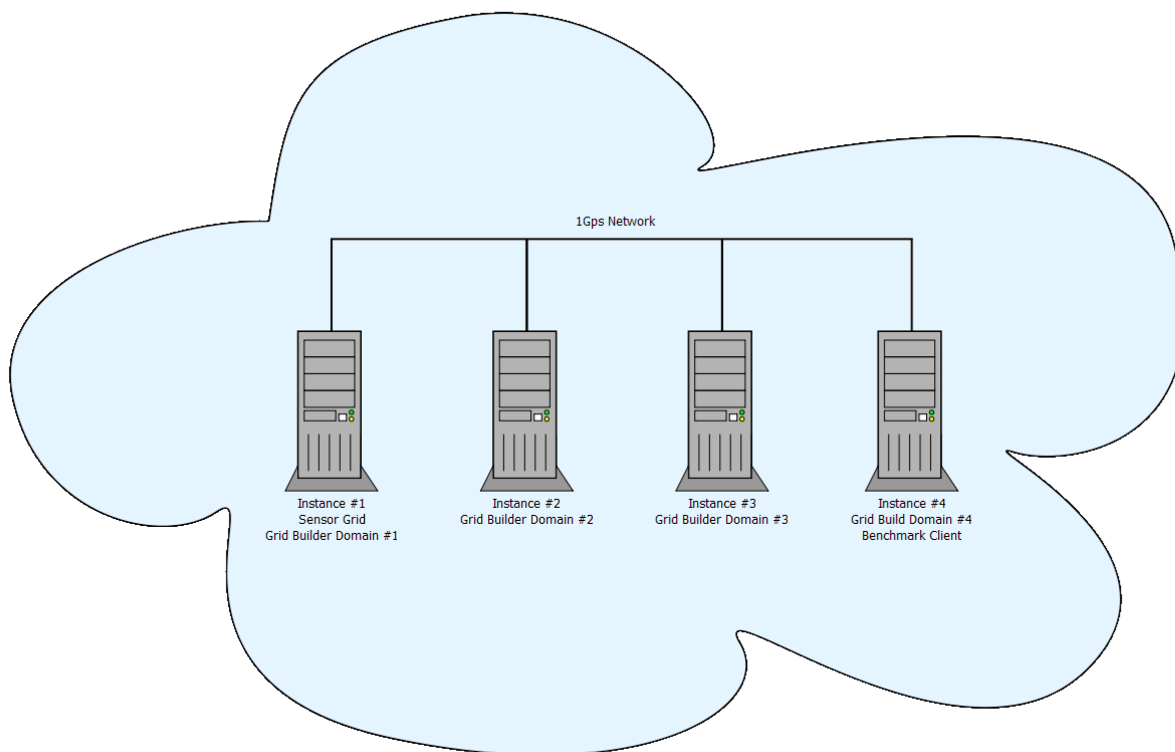
In the presentation entitled Distributed FutureGrid Clouds for Scalable Collobaroative Sensor-Centric Grid Applications general FutureGrid network throughput and performance of the underlying Narada Broker was evaluated using video sensors publishing 352x288 H264 encoded video.

For this experiment we created a virtual BenchmarkSensor to simulate a typical data stream from an IP Camera. We selected the popular TRENDnet TV-IP422WN camera as our baseline. The TV-IP422WN camera streams audio and video data over RTSP at a data rate of approximately 1800kbps when using the following encoding:

> Video: codec MPEG4; width: 640; height: 480; format: YUV420P; frame-rate: 30 frames/sec;
> Audio: codec PCM_MULAW; sample rate: 8000; channels: 1; format: FMT_S16

In order to simulate video sensors of this type we publish dummy data in 7680 bytes packets at a rate of 30 packets per second. This frame rate and packet size will also be a reasonable simulation of Microsoft Kinect sensors.

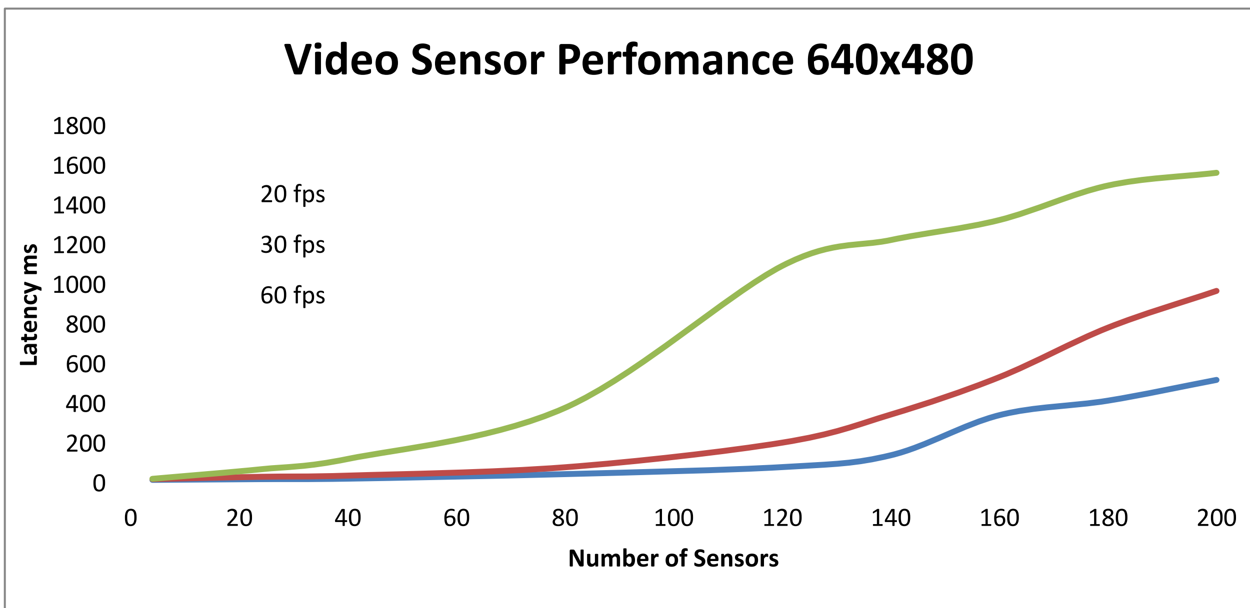
Here is our experimental arrangement:



We hosted the SGX 1.4 Sensor Grid middleware on the FutureGrid in four “large” instances.

- 2 cpu
- 6000MB ram
- 10GB disk

Various numbers of virtual video sensors were deployed evenly across the four Grid Builder domains and we measured the latency in message delivery times to subscribing client.



Appendix A: Secure Cloud Computing with Brokered Trusted Sensor Networks

Apu Kapadia, Steven Myers, XiaoFeng Wang and Geoffrey Fox
School of Informatics and Computing
Indiana University, Bloomington
{kapadia, samyers, xw7, gcf}@indiana.edu

ABSTRACT

We propose a model for large-scale smartphone based sen- sor networks, with sensor information processed by clouds and grids, with a mediation layer for processing, filtering and other mashups done via a brokering network. Final aggregate results are assumed to be sent to users through traditional cloud interfaces such as browsers. We conjec- ture that such a network configuration will have significant sensing applications, and perform some preliminary work in both defining the system, and considering threats to the system as a whole from different perspectives. We then dis- cuss our current, initial approaches to solving three por- tions of the overall security architecture: i) Risk Analysis relating to the possession and environment of the smart- phone sensors, ii) New malware threats and defenses in- stalled on the sensor network proper, and iii) An analysis of covert channels being used to circumvent encryption in the user/cloud interface.

KEYWORDS: Sensor Network, Brokered Network, Se- curity, Wireless.

1. INTRODUCTION

We consider systems in which there are large groupings of sensors reporting exorbitant quantities of potentially sensi- tive data, and the need to perform large amounts of process- ing or computation on this data with multiple large grid and cloud computing installations. The processing may need to be done in real or near-real time. Further, we consider that there are adversaries that have a vested interest in ei- ther learning information from the system, modifying the results finally output from the system (be it through modi- fication of the sensor input, filtering or processing of data), or denying access to the system. Therefore maintaining data provenance, secrecy and trust is of paramount impor- tance throughout the data life-cycle (i.e, from the point of data collection by the sensors, to its final consumption by an individual or process). All data-transformation and fil- tering, networking and sensor aspects of these systems are assumed to be susceptible to attack. Similarly the environ- ment in which some parts of the system operate is assumed to be potentially under adversarial control. In our modeling we assume the actual cloud-computing facility to be secure. Our goal is to be able to provide reliable results computed from sensor data in a manner that enables one (be it the user or the system) to make educated decisions on the reliability of that data based on trust metrics, while simultaneously preventing the loss of data-secrecy or integrity. Further, maintenance of system integrity and security is considered a core requirement. Issues such as anonymity are beyond the scope of our current research. Herein we provide a for- mal description of the networking architecture we antici- pate and the security threats. We delineate between threats and security holes for which conventional security technol- ogy suffices to solve the problem, those threats for which modifications to conventional technology are required, and those which are new and somewhat specific to the problem at hand. We next outline a largescale feasible research pro- gram to solve the many associated problems. We conclude by highlighting several of the aspects of this program for which we are actively engaged in producing solutions, and the architectures for our solutions.

1.1. Roadmap

In Section 2. we provide a high-level specification of the type of systems we are considering. This is followed, in Section 3., by a high-level threat model that depicts ways adversaries can manipulate such systems and their mal- leable environments. In Section 4., we provide more in depth discussions on three specific subsets of security prob- lems from Section 3. for which we are currently developing solutions. In Section 5. we provide related work for these

problems. Section 6. finishes off with discussion and conclusions.

2. COMPUTATION, NETWORKING & SENSING MODEL

We consider a model in which there are potentially millions of deployed sensors. The sensors may be (but are not necessarily) organized by some principle into different hierarchical layers or partitions. These sensors may be continuously publishing their observations, or supply their observations on request. In either event the observations are relayed through a brokering and filtering network, where sensor data is eventually consumed by a cloud or grid-computing infrastructure; alternatively the data can be fil- tered or processed, and stored. Importantly, we do **not** consider traditional low-power sensors such as motes, RFIDs and smartdust, where a great preponderance of wireless sensor-network research has been done. Rather, we con- sider potentially high throughput sensors attached to a — in comparison — large amount of computational and networking power, e.g., in the cloud. Specifically, we consider smartphone-class devices with reliable cellular network connectivity (with hundreds of Kbps throughput as opposed to tens of Kbps available on motes) and frequent recharging (e.g., nightly) that supports more computation- ally intensive applications than motes. Yet, this model still leaves open a large number of security issues that must be solved.

In Fig. 2. we visualize the different components of the networked system. Android smartphones denote the sen- sors in the system, and are in the possession of individu- als. The smartphones have some computational capacity, and transmit through WiFi or cellular services to a broker- ing network, running over traditional TCP/IP services. The brokering service can itself have computers performing fil- tering, processing and/or creating other mashups of sensor data.

2.1. The Sensor

Herein, we consider the sensors to be modern smartphones. These devices are diversely deployed in the field, con- tain a large number of sensors, and have moderate com- putational ability. Further, they are fully networked, and with modern 3G networks have reasonable bandwidth (e.g., 100–1000kbps). Additionally, most sensors have 802.11 WiFi radios, and may have sporadic or continuous WiFi connections in urban environments, with bandwidth of 1-50Mbps. These phones may be in the control of trusted (or semi-trusted) individuals, or be located in some poten- tially untrusted environment. Further, they have a reason- able processing capability on modern low-power proces-

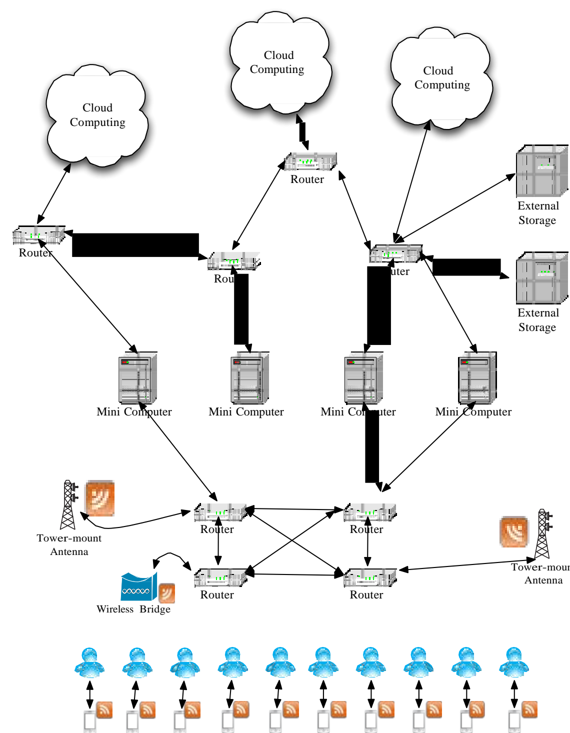


Figure 1. A Depiction of the Different Components of the Sensor and Cloud-Computing Network.

sors, such as an ARM architecture processor running at 500–800MHZ. It is assumed that the phones have standard sensors including, eGPS, 802.11x, Bluetooth v2 (Class 1, 2 or 3), temperature, orientation, acceleration, audio microphone, and camera (stills or video). In particular, our project focuses on the use of HTC G1 Android (v1.6) development phones, due to the ease of programming and their ability to multi-task (unlike the iPhone). Such plat- forms can perform a full host of cryptographic operations, but also have security issues relating to the fact that they are multi-purpose computing platforms. Thus OS secu- rity issues are larger, and it is difficult to construct a small OS, such as TinyOS [19] designed for motes, which can be more easily hardened to withstand attack. While the smart- phones are capable of more standard cryptographic proto- cols, a large number of such sensors in a region that are broadcast could overwhelm communications channels, and battery life is still a concern — if not as pressing. There- fore, low bandwidth and energy usage requirements are still a concern. However, one can easily port low-energy and bandwidth secure networking stacks, such as those pro- vided by TinySec [17] or MiniSec [21].

2.2. The Brokering Network

With potentially millions of smartphone sensors producing data at any given time, the need for a high performance networking infrastructure that is capable of self-filtering unimportant data feeds before they are transmitted for pro- cessing becomes apparent. Further, the need to funnel po- tentially very large amounts of bandwidth to a few collec- tion points for processing is also evident. The communi- cation between the sensors and the computing infrastruc- ture is mediated by a brokering network that uses a pub- lish/subscribe model. In such a model, each sensor can publish the data it is collecting on a continuous basis, along with appropriate meta-data that depict the content, prove- nance and trustworthiness of the data. Requests for specific information at the cloud or grid computing interface will drive the request for specific types and trustworthinesses of data from the sensors. Such requests will further invoke the subscription to different forms of data both real-time and stored. Typical forms of data cleaning and processing can, of course, be performed by dedicated servers who indepen- dently subscribe to sensor feeds, and then publish their own mashed data feeds for consumption by others. In such cases provenance and trustworthiness must be maintained. Ul- timately, there will be many different parallel consumers of data, and thus the network must be as responsible as is possible to prevent duplication of effort, redundant routing, streaming and processing of data.

For this project, the Narada Brokering network[1] is being used. The network can provide basic secrecy and integrity requirements, but does not by default provide any informa- tion regarding provenance or trustworthiness. While other suitable brokering networks can be used (e.g., Solar [?]) we chose Narada because of local expertise and support avail- able to our project.

2.3. Computing Model

We assume that the final consumers of data will be cloud or grid computations, as will many of the filtering and pro- cessing modules. While each cloud or grid may see its out- put as the final consumable, the desire to recycle computa- tion means that the data may itself become simply another input to an alternate computation upstream. The study of securing cloud and grid computation are separate research fields in their own right, and so our model simply assumes that these computations do not leak information, break in- tegrity of the data nor provide covert channels to the data. Computational power and storage is considered to be more or less limitless to within reasonable bounds.

[1]See www.Naradabrokering.org

3. SECURITY, PRIVACY & TRUST ISSUES

The computing environments of a sensor grid are fraught with different kinds of threats, which endanger the security and privacy assurance the system can provide. Mitigation of these threats relies on establishing trust on individual system layers through proper security control. In this sec- tion, we survey the security and privacy risks on each layer of senor-grid computing and the technical challenges for controlling them.

A sensor grid interacts with its operating environment through a set of sensors. Those sensors work either au- tonomously or collaboratively to gather data and dispatch them to the grid. Within the grid, a brokering system fil- ters and routes the data to their subscribers, the clients of the sensor grid. We now describe the security and privacy issues on each layer of such an operation. This includes the environment the sensors are working in; the sensors; the grid; the clients; and the communications between the sensor and grid, and the grid and clients.

The Environment. An adversary could compromise the sensors' working environments to contaminate the data they collect. For example, one can add ice around individ- ual sensors to manipulate the temperatures they measure; alternatively, one could imagine that GPS signals were be- ing spoofed in an area. Detection of such a compromise can be hard, when the adversary has full control of the en- vironment. A possible approach is to check the consistency of the data collected from multiple sensors and identify anomalous environmental changes as indicated by the data.

Sensors. Sensors can be tampered with by the adversary who can steal or modify the data they collect. Mitigation of this threat needs the techniques that detect improper opera- tions on the sensors and protect its sensitive data. Since we assume sensors are smartphones, they also are susceptible to a large number of security concerns of traditional PCs, which includes viruses and malware.

Cloud or Grid. Information flows within the grid can be intercepted and eavesdropped on by malicious code that is injected into the system through its vulnerabilities. Authen- tication and information-flow control need to be built into the brokering system to defend against such a threat.

Client. The adversary can also manage to evade the secu- rity and privacy protection of the system through exploit- ing the weaknesses of the clients' browsers. The current design of browsers is well known to be insufficient for fending off attacks such as cross-site scripting (XSS) and

cross-site request forgery (XSRF). Such weaknesses can be used by the adversary to acquire an end user's privileges to wreak havoc on the grid. Defense against the threat relies on design and enforcement of a new security policy model that improves on the limitations of the same origin policy adopted in all of the mainstream browsers.

Communication Channels. The communications be- tween the sensors and the brokering network, the broker- ing network and the cloud or grid, and the cloud or grid and the client, are subject to both passive (e.g., eavesdrop- ping) and active (e.g., man-in-the-middle) attacks. Coun- tering this threat depends on proper cryptographic proto- cols that achieve both data secrecy and integrity. In each case, different engineering requirements based on differ- ing scarce resources require different solutions. In the case of the wireless connection between the sensor network and the brokering network, bandwidth and power-usage are key requirements. Once on the brokering network, data prove- nance becomes a key challenge. Traditional cryptographic protocols would seemingly suffice from the cloud to the user. However, a tricky issue here is the information leaks through side channels. For example, packet sizes and se- quences. Our preliminary research shows that such infor- mation reveals the state of web applications, which can be further utilized to infer sensitive data within the applica- tion. Understanding and mitigating the problem needs fur- ther investigation.

4. PROBLEMS TO BE ADDRESSED

While there are a large number of potential security issues to be addressed, as partially scoped and enumerated in the previous section, the investigators are working on the fol- lowing specific problems.

4.1. Detection of anomalous use of sensors

A key issue involved in trusting data from the sensors in the described network is to ensure that the sensors themselves can be trusted. That is, either they are in the possession of individuals who are trustworthy, or they have not been tampered with in their environment if not possessed by an individual.

In our model if the sensor is in the possession of a trusted individual, it is more likely that its sensors are reporting an honest or legitimate environment, and not one that has been manipulated with the goal of producing faulty results that get incorporated in to final computation. Smartphones, however, can be easily stolen, misplaced or temporarily in- tercepted and reprogrammed by adversaries. If stolen or misplaced, the environment that the sensors report may be altered, and thus the data collected may be untrustworthy. The use of traditional authentication technologies to ensure a legitimate user is in control of the smartphone sensor is not practical, as said users cannot be queried to authenticate every time the sensor-net needs to report readings.

We propose a system in which a phone attempts to deter- mine if it is or is not in the possession of a legitimate user. In cases where the phone determines it is in questionable hands it deauthenticates itself. Deauthentication either re- moves it from the sensor network, or forces its sensor read- ings to be tagged as untrustworthy, with risk measurements being included in provenance data to ensure that the risk of improper readings is communicated down stream and taken into account on further processing. In order for the phone to determine whether it is under legitimate possession, we are developing a risk assessment system based on the inputs from the sensors of the phone itself. Thus the sensors are used directly to determine if the sensors' readings should be trusted. We are implementing a prototype of this system on the HTC/Google G1 Android (v1.6) Phone.

We are taking different approaches with different sensors on the phones. Note we are using these sensors to de- termine risk of improper possession independent of which sensors are of interest to the sensor network. Further, we make two broad classifications of the use of sensor input for risk determination. First, *environmental sensors* attempt to measure properties of the environment around the phone, or of the user. Second, *social-networking sensors* measure "friendly" or "unfriendly" people that surround the phone.

4.1.1. Environmental Sensors

Positioning Information. Android smartphones can de- termine their position using a combination of several differ- ent information sources, which includes cellular transmis- sions (in particular, tower location), GPS positioning and WiFi positioning. The combination of all of these pieces of information is often called eGPS, and frequently provides position far more accurately than any of the technologies alone. Our high-level goal is for the phone to learn certain geographic locations and routines that correspond to either a safe or dangerous state.

We extend the work of Farrahi and Gatica-Perez [14]. We are using a third-order Hidden Markov Model (HMM) to determine the risk of misuse of a phone based on current positional information. Farrahi and Gatica-Perez consid- ered the problem of determining location for contextual ap- plication purposes, but without specific interest in authen- tication and security mechanisms. A day is divided into blocks of 30 minutes. In any given period the phone is con-

sidered to be in one of four specified places (e.g., Home, Work, Aux 1, No Location Reading) or in a generic unlabeled place (Other). Thus the location of an individual through a time period is being converted into a string, as is depicted in Fig. 2. Currently, we are considering a super- vised learning case where a user specifically defines these five locations, with the goal of using clustering algorithms to eventually learn popular locations. Traces of individ- uals' positions are then collected, and the HMM iterative Viterbi training and Forward algorithm are used for train- ing on this past annotated data sequences and predicting risk. Based on a trained HMM, and a recent history of the phones' positions, the forward algorithm is used to deter- mine the likelihood of the recent history, and this estimate is used to determine the risk associated with the phone's current position. Of clear importance is the efficiency with which both training and evaluation can be performed. Due to the need to only occasionally perform training (say daily or weekly to update the movement model with the most re- cent trends), its efficiency is of lesser importance than that of real-time risk evaluation which needs to be performed on demand in real-time in order to prevent users form be-

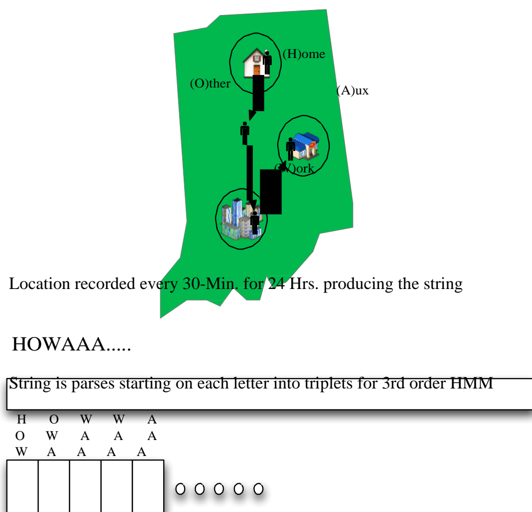
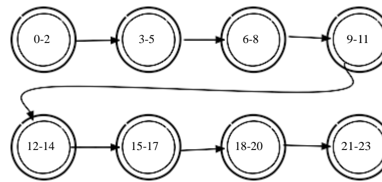


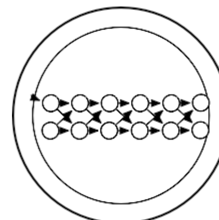
Figure 2. A Depiction of How Positional Data Through the Day is Converted in to a String Over a Small Alphabet.

As previously mentioned, risk evaluation is based on the use of the forward algorithm. The forward algorithm runs in $O(n^2 \cdot t)$ where n is the number of states and t is the

number of time-blocks being analyzed; given an HMM M the forward algorithm returns the probability that a given sequence of positions $x_1, .., x_t$ is output by an HMM, given that it terminates in state σ_t. More formally, $\Pr[M \rightarrow x_1, \ldots, x_t/\sigma_t]$, for a given $x_1, ..., x_t$, and σ_t. However, for



A hierarchical HMM model is used to learn users schedules. At the outer layer we in essence have a node for each 3 hour block of time in the day.



Each node contains within it a 3rd order multi-state HMM to learn the schedule over the corresponding hours.

Figure 3. A Depiction of the Constructed HMM for Predicting Position.

risk analysis we have no preference for any specific termi- nal state, and so we are interested in $\Pr[M \rightarrow x_1, \ldots, x_t]$. A simple modification that sums the probabilities over all final states runs in $O(n^3 \cdot t)$, and returns the value of in- terest. Given the running time is cubic in the number of states and we need near real-time evaluations of the algo- rithm, we need to minimize the state space. To minimize the state space we actually construct 8 individual HMMs to learn patterns of behavior during different 3-hour periods of the day, and link them together through a simple state- machine.[2] The model is depicted in Fig. 3.

We justify this construction as a reasonable model because the risk of one's current geographic position is a function of both one's current position and recent historical position relative to the current time, as opposed to one's longterm schedule. We are currently in the process of experimentally determining the correct recent history window that will de- liver the best ability to detect abnormal behavior.

Temperature Temperature of the phone can be used to determine information relating to whether the phone is cur- rently in someone's physical possession. If the phone reads approximately body temperature (37° C) then it is reason- able to assume that is in a person's possession.[3] Similarly, if the phone is at approximately room temperature or the outdoor ambient temperature, then the phone is likely ei- ther not directly on the person and is likely to have either

[2] This construction could be viewed as a Hierarchical HMM in which the transition distribution in the high-level HMM are all Kronecker δ-functions.

[3] There may need to be some invalidation of this metric at times when the ambient temperature is the same as body temperature.

been put down or remain in a bag.

While we believe there is strong potential to help use the phone's current temperature to monitor risks, our initial test of the Android phone is that the delay in converging to new temperatures by the phone's sensor makes this data unus- able for our intended applications. We found that when moving the phone in a pocket at body temperature and moving it onto a desk, it took on the order of tens of min- utes to converge to anywhere near the ambient room tem- perature. Further, in the same scenario it took several min- utes to decisively report non-body temperature readings.

Acceleration Acceleration measurements can be used in several manners to help determine risk. Techniques have been developed to measure a person's gait using the ac- celerometer in phones, assuming they are placed in an individual's pocket, or otherwise carried on the person [30, 15, 1]. While we do not intend to implement such a scheme ourselves, we are looking at the possibility of in- cluding the results of these works to deploy such a tech- nique in our larger sensor scheme. Further, we plan to use techniques that include simpler measurements but are based on other contexts. For example, if a user does *ex- plicitly* authenticate to the device, then at this point in time we know that the device is trusted. If the device stays in motion for the next several minutes, then one can assume that the correct user is still in possession of the device. In contrast if the phone becomes stationary for a prolonged period of time, the phone probably has been put down, and now alternative risk measurements must be used.

4.1.2. Social Networking Sensor Risk Measurement

One key aspect of our system is to use a form of social net- working for authentication and risk measurement. Imagine a scenario where a phone finds itself in a previously un- visited location, and other sensors are providing question- able risk data. However, imagine that the device can find the presence of a number of other phones that it frequently observes when in known low-risk states. The presence of these phones should indicate that the risk that an individual does not have proper possession of the phone is low: the phones of colleagues, friends and family members are near, so either the entire group is at risk (unlikely or the phone is simply in a new environment). Our system will employ a combination of white and black listing of other phones, which will alter the risk assessments made by the system. Additionally, we will learn "friendly" phones by determin- ing which other phones are frequently in the presence of the user in non-risky situations. This assessment will be done by considering both Bluetooth and 802.11 wireless networks.

Bluetooth. General Bluetooth frames are much more dif- ficult to detect than corresponding 802.11x frames *with the standard radio hardware built in to phones*.[4] There are two options to bypass this problem. The first is that the phones broadcast themselves in so called "Bluetooth dis- covery mode", this will make the phone visible to all, but can result in higher battery usage. The second is to pair specifically with those phones that are whitelisted to be considered friendly; pairing requires a one-time user inter- vention. In this case, the phones could attempt to pair when they are in close contact.

More problematically, our current implementation platform (Android v1.6) does not provide an API to interface with the Bluetooth infrastructure. Thus Bluetooth can only be accessed by the user, and not a risk-analysis program. An- droid (v2.0) does provide the implementation of such API, but there is currently no firmware upgrade for our reference platform (HTC G1 development).

WiFi (802.11x). Much of the widely deployed smart- phones allow their WiFi radios to operate in *promiscu- ous mode*, which permits the radio to listen to and com- municate the existence of frames that it can receive, even if the radio was not the target for the frame in question. This mode allows 802.11x radios to detect the presence of nearby devices. The only requirement to instantiate our social-networking risk measurement is to ensure that all the participating phones are broadcasting their position by sending beacons on regular intervals. It is yet to be deter- mined if the development platform supports such modes of operation.

4.1.3. Combining Risk Measurements.

A more sensitive risk measurement can be constructed if one does not require each sensor to independently gener- ate a risk metric in our risk model. However, in order to make our scheme flexible for different uses, and in devices with different subsets of sensors, we consider an archi- tecture that treats the sensor measurements independently, and then produces a global risk measurement. Note that this separation does not prevent the global risk measure- ment from learning co-dependencies between risk profiles of different sensors, and making use of such dependencies. There is a fair amount of research on methods for aggre- gating risk measurements in a number of different scenar- ios (e.g., Financial, Credit, Insurance, Intrusion Detection). Currently we are determining which, if any, of the current models provides a similar or appropriate model on which to base an aggregation of our sensor work. In the mean time,

[4] Relatively inexpensive hardware is available to capture general Blue- tooth packets, but it is not standard on known phones.

we use an expected value of the different risk metrics that is weighted with high-degrees to the positional and social networking schemes.

4.2. "Sensory Malware" threats and defenses

To fully understand the threat space of malware on smart- phones, we are exploring various attack scenarios. While traditional malware defenses focus on protecting resources on the computer (or as we would expect, on the smart- phone), we are specifically interested in the new class of at- tacks where *sensory malware* uses onboard sensors to steal information from the user's physical environment [5]. For example, the user carries around a video and audio sen- sor (microphone) at all times, and thus immense amounts of information such as sensitive conversations, spoken passphrases or biometrics, keyboard acoustic emanations when placed next to a keyboard, and broader surveillance becomes possible. Video "sensors" can gather visual infor- mation about a user's private environment such as pictures of colleagues [38], which may be sensitive with military and intelligence-gathering agencies. Accelerometers and GPS sensor information can be used to infer location and activity patterns of users such as soldiers, thus compromis- ing military secrecy.

While generic architectures [10, 23] have been proposed to control access to the network, for example, after soft- ware has accessed certain sensor information, various vec- tors exist for leaking garnered information. Overt channels between components on the smartphone (Android provides very little security against communicating applications, for example), or covert channels between related malware ap- plications (through a storage channel, for example) are cur- rently viable vectors for leaking sensitive data to adver- saries. It is even possible to leverage other "blessed" ap- plications on the phone to act as a carrier for such informa- tion (by invoking a web-browser with an encoded URL, for example). Thus we are interested in building a unified ar- chitecture for controlling access to sensor data, and limiting what information can be gleaned from the user's environ- ment unless he or she is making use of legitimate appli- cations. We are currently building a software prototype of one instance of sensory malware to demonstrate the reality of the threat, and to better understand defensive techniques to limit such malware.

We aim to study types of sensory malware that are stealthy and thus use few resources on the mobile device. For ex- ample, speech-based malware may use several heuristics to target analysis at only specific portions of the audio sample. Such targeted analysis can drastically reduce the amount of resources needed to analyze audio samples, thus decreasing the observability of such malware. To conserve power, such malware can also target its offline processing to when the mobile device is connected to a power source for charging. Under such circumstances the malware uses few precious resources and does not detract from the user's experience. Speech malware of this type may even operate using more general "profiles" that tune the malware to recognize sev- eral different situations, or contexts, such as a recognized phone number that is dialed. Based on the context, the speech malware can, for example, detect a credit card cus- tomer service line and target analysis to credit card number extraction. Calls to financial institutions such as banks of- ten require portions of the user's social security number, which could be extracted similarly. Such profiles can make use of other clues such as audio or video triggers to better target surveillance and transmit specific information.

To counter such threats, therefore, we need a framework that is better equipped to deal with sensory malware threats. Research is needed to understand the threat space of sen- sory malware, so that effective defenses can be deployed. As mentioned earlier, existing solutions are unable to deal with situations in which malware communicates through covert channels, and thus such work must also take into ac- count anomalous resource usage to detect such covert chan- nels. Being low-powered devices makes the job of defen- sive software much more challenging, and thus lightweight detection techniques are necessary. It is even possible that the mobile platform can leverage computation in the cloud for "outsourced intrusion detection," which might strike a tradeoff between the time to detection and power consump- tion.

4.3. Side-channel detection and mitigation

It is well known that the contents of encrypted traffic can be disclosed by its attributes observable to a eavesdrop- per, for example, packet sizes, sequences, inter-packet tim- ings. Such attributes, often referred to as side-channel in- formation, often pose a grave threat to the confidential- ity of the communication under the protection of cryp- tographic protocols. Side-channel leaks have been ex- tensively studied for decades, in the context of secure shell (SSH) [27], video-streaming [26], voice-over-IP (VoIP) [37], web browsing and others. As an example, a line of research conducted by various research groups stud- ied anonymity issues in encrypted web traffic. It has been shown that because each web page has a distinct size, and usually loads some resource objects (e.g., images) of differ- ent sizes, the attacker can fingerprint the page so that even when a user visits it through HTTPS, the page can still be re-identified [9, 29]. This vulnerability is known to be a serious concern for anonymity channels such as Tor [31],

which are expected to hide users' page-visits from eaves- droppers.

A sensor grid system can also be highly susceptible to the threat of side-channel leaks. As described before, such a system collects data through distributed sensors, processes it within a cloud, and delivers the data and related services to end clients. This highly distributed computing paradigm is fraught with the hazards of information leaks, when con- fidential data are transmitted between the sensors and the cloud, and between the cloud and the clients, despite the protection of the state-of-the-art cryptographic techniques. Such privacy risks are described as follows:

Wireless Sensor Communication. The wireless channel connecting the sensors to the cloud is extremely vulnerable to the eavesdropping attack. The sensitive data delivered through this channel can be easily intercepted and analyzed by the adversary. Though encryption can prevent a direct disclosure of the data, it does not cover the side-channel in- formation, which, under some circumstances, can be used to infer the content of the sensitive data. As an example, collaborating with Microsoft Research (MSR), we recently discovered that even for the organization deploying up-to- date WPA/WPA2 Wi-Fi encryptions, it cannot prevent an unauthorized party from collecting the query words its em- ployees enter into Google/Yahoo/Bing Search. This is be- cause the suggestion-list features of these search engines makes the sizes of the packets generated in response to different query letters distinct. As a result, the adversary who observes these packets, despite not gaining access to their contents, can map their sizes to the different letters one types into the search engines.

Cloud–consumer Communication. The encrypted data exchanged between the cloud and its customers are equally subject to the side-channel threat. Cloud computing is built upon the infrastructure of *software as a service* (SaaS), through which *web applications* are delivered as services to web clients. Unlike its desktop counterpart, a web applica- tion is split into browser-side and server-side components. As a result, a subset of its internal information flows (i.e., data flows and control flows) are inevitably exposed on the network, which reveal application states and state transi- tions. Our collaborative research with MSR reveals that the side-channel weakness of SaaS is fundamental, which can be used to infer a large amount of information from many high-profile, extremely popular web applications. The sen- sor grid system also faces the same threat: it offers services and data to its customers through web applications, whose side-channel information could lead to the disclosure of the data, even when the communication has been protected by the cryptographic protocols like HTTPS.

The seriousness of the side-channel threat varies from case to case, depending on the features of the data and the way in which they are transmitted. An important research, there- fore, becomes how to design a systematic way to detect the side-channel vulnerabilities within sensor/cloud inter- actions and the web applications that serve the sensor grid's customers. A possible solution is to use *information-flow analysis* [28], when the source code of related software is available. The software developer can first label taint sources within a program, e.g., variables that contain sen- sitive user data, and then run a detection tool to analyze its source code and track the propagation of taint data through both data flows and control flows. Whenever taint data are found to be transmitted across the network between the ap- plication's client and server components, an information- leak evaluation is performed to understand whether side- channel information, such as packet sizes, sequences and timings, can be linked back to the content of the data. When the source code is unavailable, we can use the techniques like fuzz testing to evaluate sensor-cloud interactions and cloud-client interactions on different data sets, to identify the correlation between the attributes of encrypted traffic and the content of the data.
Control of side-channel leaks can also be highly nontriv- ial, particularly when web applications are involved. Our collaborative research with MSR reveals that conventional defenses like packet padding and adding noise can be less effective and more costly than expected, without con- sidering the specific properties of individual applications. This problem comes from the difficulty in hiding the side- channel information related to state transitions specific to each application, and the limited information an application has about the attributes of the web traffic it generates, due to the extension or compression made by the web server. This vulnerability calls for a change in the current way of developing web applications to include the collaborations among multiple related parties: as an example, we could let the software developer specify the policies for padding packets at different program states, and the web-server ven- dor enforce the policies within the web server that actually generates the packets.

5. RELATED WORK

Kapadia et al. [16] list several security challenges for sim- ilar smartphone based sensing environments. While their work focuses mainly on an opportunistic sensing model where sensors are *tasked* for readings sent back as *reports* to other users or applications in urban sensing environ- ments, we focus on environments where sensors push mas- sive amounts of data to a compute cloud. We now list re- lated work for the three specific problems discussed in Sec- tion 3..

5.1. Mobile phone security and privacy

There has been some work in using sensors to establish context for different purposes on smartphones. The work of Peddemors et al. [24] uses past networking and sensor events to predict future network events. They give exam- ples of predicting network availability. The ability to pre- dict events is distinct from deviating from normal or pre- scribed behavior. Nonetheless they use the prediction of being at home or work, and for durations. Therefore, the system should be considered. Of particular problem is the complexity of computing predicted events, which would be too slow in our scenario.

The work of Tanviruzzaman et al. [30] is most similar to that discussed here. In their work, they suggest the use of a hierarchy of sensor information to establish authentica- tion, and show some work on using accelerometer data on an iPhone to produce a biometric that can be used to au- thenticate to the phone.

Other work by Jong-Kwon and Hou [18] has predicted user behavior and movements from the perspective of a large WiFi network, for the purposes of assigning scarce resources appropriately. However, we do not rely on one overarching network for our positioning system. Yet, the possibility exists that such work could be used to have the network aid in performing risk analysis.

The field of smartphone security and the security of cellphone infrastructure is now being widely researched. Traynor [32] gives a short overview of infrastructure pos- sibilities and problems. Traynor et al. [34] consider the potential effect of a malnet of smartphones on the cellular network's infrastructure. Enck et al. [13] discuss exploits in the SMS-network infrastructure, and Traynor et al. [33] discuss mitigation strategies for such exploits.

Relating to mobile phone security, there has been recent in- terest in maintaining their security. The potential to attack these devices, and that they would suffer similar security fates to personal computers, such as viruses and malware, has been long understood [8]. Specific approaches to con- sidering defense against such software on smartphones has been considered by Cheng et al. [7]. The specific strengths and weaknesses of the Android security model are explored by Ongtang et al. [23]. The ability to securely determine if software downloads are trusted on such devices is explored by Enck et al.[11]. Enck et al. [12] give an introduction to understanding the Android security model specific to the smartphones we are using for implementation.

5.2. Sensory malware threats and defenses

As mentioned earlier, researchers are already investigating attacks and defenses related to sensory malware [5]. Xu et al. [38] provide a proof-of-concept implementation of video-capture malware. Their malware captures video and transmits this video after suitable compression to lessen the burden on the network. These malware do not appear to be stealthy enough because of the large amounts of video data transferred on the network. We thus seek to develop and evaluate solutions where malware is even more stealthy, by limiting the network communication. In fact, we would like to study situations where network access is limited com- pletely using techniques such as Kirin, a lightweight secu- rity certification mechanism for applications on Android. Even in cases where a system such as Saints [23] is used to control the interaction between applications, we would like to study the use of covert channels to circumvent such mechanisms.

Detection techniques such as behavioral detection of mal- ware by monitoring system calls [3], and power consump- tion [20] already attempt to detect malware on mobile plat- forms. We aim to study the limits of such detection tech- niques since resources are limited, and how malware can circumvent detection because of the inherent limitations on the detection techniques.

5.3. Side-channel information leaks

Side-channel leaks have been known for decades: a doc- umented attack has been dated back to 1943 [22]. The threat has been extensively studied in different contexts: information is found to be exposed through electromag- netic signals (e.g., keystroke emanation [35]), shared mem- ory/registers/files between processes (e.g., the recent dis- covery of the side-channel weakness in Linux process file systems [39]), CPU usage metrics, etc. Recently, such in- formation leaks are found to threaten cloud computing plat- forms like Amazon EC2 [25].

Encrypted communications are often subject to the side-channel attacks, which leverage such information as packet timings and sizes to infer the contents of encrypted data. Prominent examples include Brumley et al.'s attack on the RSA secret keys used in OpenSSL [4], Song et al.'s work on keystroke inference from SSH [27], Wright et al. and others' analysis of phrases and sentences from the variable- bit-rate encoding in VoIP [37], and Saponas et al.'s detec- tion of movie titles in an encrypted video-streaming system (Slingbox Pro) [26]. Encrypted web communication has also been found to be vulnerable to the side-channel attack. Prior research shows that a network eavesdropper can often

fingerprint web pages using their side-channel characteris- tics to identify the pages the victim visits. This idea first appeared in the personal communication among Wagner, Schneier and Yee in 1996 [36], and was later demonstrated in a course project report in 1998 by Cheng et al. [6]. Sun et al. [29] and Danezis [9] both indicated the impacts of the attack on anonymity channels like Tor, MixMaster and WebMixes. It was also discussed by Bissias et al. [2], who studied WPA and IPSec, instead of SSL/TLS in other re- search.

6. SUMMARY

We have outlined a high-level architecture that should both be realizable, and provide for the ability to perform on- demand analysis and processing of data from a large num- ber of heterogeneous and globally placed sensors. The net- work is structured so that it is feasible to consider real or near-real time processing and interpretation of the data with appropriate resources. However, challenges remain in de- termining how to assure privacy, integrity and provenance of the data from its collection, through its life-cycle of pro- cessing to final consumption. The authors' belief is that the largest research questions based on our model lie at the tail ends of the data life-cycle; namely, there are open research questions at data-collection by smartphone sensors and in the final delivery of a processed data-consumable. Specific directions aimed at solving these problems have been dis- cussed, along with initial development of solutions. We summarize this in Table 1

Table 1. Summary of the Three Threats, Associated Dangers and Mitigation Strategies We Actively Address.

Threat	Danger	Mitigation
Sensor Ab- duction	Malicious Sensor Data	Detection of non-regular usage
Side-channel information leakage	Communication en- cryption is circum- vented by analysis of packet sizes& spac- ing	Flow-analysis and padding
Sensor mal- ware	Sensor data theft	Sensor access control mod- els

REFERENCES

[1] *Identifying users of portable devices from gait pattern with accelerometers*, volume 2, 2005.

[2] G.D. Bissias, M. Liberatore, D. Jensen, and B.N. Levine. "Privacy vulnerabilities in encrypted http streams," In proceedings of Privacy Enhancing Technologies Workshop (PET 2005), pages 1–11, 2005.

[3] A. Bose, X. Hu, K.G. Shin, and T. Park. "Behavioral detection of malware on mobile handsets," In MobiSys '08: Proceeding of the 6th international conference on Mobile sys- tems, applications, and services, pages 225–238, New York, NY, USA, 2008. ACM.

[4] D. Brumley and D. Boneh. "Remote timing attacks are practical," In proceedings of the 12th USENIX Security Symposium, pages 1–14, 2003.

[5] L. Cai, S. Machiraju, and H. Chen. "Defending against sensor-sniffing attacks on mobile phones," In MobiHeld '09: proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds, pages 31–36, New York, NY, USA, 2009. ACM.

[6] H. Cheng and R. Avnur. "Traffic analysis of SSL encrypted web browsing," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.1201\&rep=rep1\&type=url\&i=0, 1998.

[7] J. Cheng, S.H.Y. Wong, H. Yang, and S. Lu. "Smartsiren: virus detection and alert for smartphones," In MobiSys '07: proceedings of the 5th international conference on Mobile systems, applications and services, pages 258–271, New York, NY, USA, 2007. ACM.

[8] D. Dagon, T. Martin, and T. Starner. "Mobile phones as computing devices: The viruses are coming!" *IEEE Pervasive Computing*, 3(4):11–15, 2004.

[9] G. Danezis. "Traffic analysis of the http protocol over TLS," http://homes.esat.kuleuven.be/~gdanezis/TLSanon.pdf, as of Dec 2009.

[10] W. Enck, M. Ongtang, and P. McDaniel. "On lightweight mobile phone application certification," In CCS '09: proceedings of the 16th ACM conference on Computer and communications security, pages 235–245, New York, NY, USA, 2009. ACM.

[11] W. Enck, M. Ongtang, and P. McDaniel. "On lightweight mobile phone application certification," In CCS '09: proceedings of the 16th ACM conference on Computer and communications security, pages 235–245, New York, NY, USA, 2009. ACM.

[12] W. Enck, M. Ongtang, and P.D. McDaniel. "Understand- ing android security," *IEEE Security & Privacy*, 7(1):50–57, 2009.

[13] W. Enck, P. Traynor, P. McDaniel, and T. La Porta. "Exploit- ing open functionality in sms-capable cellular networks," In CCS '05: proceedings of the 12th ACM conference on Com- puter and communications security, pages 393–404, New York, NY, USA, 2005. ACM.

[14] K. Farrahi and D.G. Perez. "Learning and predicting multimodal daily life patterns from cell phones," In J.L. Crowley, Y. Ivanov, C.R. Wren, D. Gatica-Perez, M. Johnston, and R. Stiefelhagen, editors, ICMI, pages 277–280. ACM, 2009.

[15] T. Iso and K. Yamazaki. "Gait analyzer based on a cell phone with a single three-axis accelerometer," In MobileHCI '06: proceedings of the 8th conference on Human-computer interaction with mobile devices and services, pages 141–144, New York, NY, USA, 2006. ACM.

[16] A. Kapadia, D. Kotz, and N. Triandopoulos. "Opportunis- tic Sensing: Security Challenges for the New Paradigm," In The First International Conference on Communication Sys- tems and Networks (COMSNETS), January 2009.

[17] C. Karlof, N. Sastry, and D. Wagner. "Tinysec: a link layer security architecture for wireless sensor networks," In Sen- Sys '04: proceedings of the 2nd international conference on Embedded networked sensor systems, pages 162–175, New York, NY, USA, 2004. ACM.

[18] J.K. Lee and J.C. Hou. "Modeling steady-state and tran- sient behaviors of user mobility: formulation, analysis, and application," In MobiHoc '06: proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing, pages 85–96, New York, NY, USA, 2006. ACM.

[19] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. "TinyOS: An operating system for sensor networks," In Ambient Intel- ligence. Springer Verlag, 2004.

[20] L. Liu, G. Yan, X. Zhang, and S. Chen. "Virusmeter: Pre- venting your cellphone from spies," In E. Kirda, S. Jha, and D. Balzarotti, editors, *RAID*, volume 5758 of Lecture Notes in Computer Science, pages 244–264. Springer, 2009.

[21] M. Luk, G. Mezzour, A. Perrig, and V. Gligor. "Minisec: a secure sensor network communication architecture," In IPSN '07: proceedings of the 6th international conference on Infor- mation processing in sensor networks, pages 479–488, New York, NY, USA, 2007. ACM.

[22] Wired News. "Declassified NSA document reveals the secret history of tempest," http://www.wired.com/threatlevel/2008/04/nsa-releases-se.

[23] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. Mc- Daniel. "Semantically rich application-centric security in An- droid," In ACSAC, pages 340–349. IEEE Computer Society, 2009.

[24] A. Peddemors, H. Eertink, and I. Niemegeers. "Predicting mobility events on personal devices", *Pervasive and Mobile Computing, Special issue on Human Behaviour in Ubiquitous Environments,* To Appear.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," In CCS '09: proceedings of the 16th ACM conference on Computer and communications security, pages 199–212, New York, NY, USA, 2009. ACM.

[26] T.S. Saponas, J. Lester, C. Hartung, S. Agarwal, and T. Kohno. "Devices that tell on you: privacy trends in con- sumer ubiquitous computing," In SS'07: proceedings of 16th USENIX Security Symposium on USENIX Security Sympo- sium, pages 1–16, Berkeley, CA, USA, 2007. USENIX As- sociation.

[27] D.X. Song, D. Wagner, and X. Tian. "Timing analysis of keystrokes and timing attacks on SSH," In SSYM'01: pro- ceedings of the 10th conference on USENIX Security Sym- posium, pages 25–25, Berkeley, CA, USA, 2001. USENIX Association.

[28] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. "Secure program execution via dynamic information flow tracking," In ASPLOS-XI: proceedings of the 11th international con- ference on Architectural support for programming languages and operating systems, pages 85–96, 2004.

[29] Q. Sun, D.R. Simon, Y.M. Wang, W. Russell, V.N. Padman- abhan, and L. Qiu. "Statistical identification of encrypted web browsing traffic," In SP '02: proceedings of the 2002 IEEE Symposium on Security and Privacy, page 19, Washington, DC, USA, 2002. IEEE Computer Society.

[30] M. Tamviruzzaman, S.I. Ahamed, C.S. Hasan, and C. O'brien. "ePet: when cellular phone learns to recognize its owner," In SafeConfig '09: proceedings of the 2nd ACM workshop on Assurable and usable security configuration, pages 13–18, New York, NY, USA, 2009. ACM.

[31] The Tor Project. *Tor: anonymity online.* http://www. torproject.org/, 2009.

[32] P. Traynor. "Securing cellular infrastructure: Challenges and opportunities," *IEEE Security & Privacy*, 7(4):77–79, 2009.

[33] P. Traynor, W. Enck, P. McDaniel, and T.L. Porta. "Miti- gating attacks on open functionality in sms-capable cellular networks," In *MobiCom '06: proceedings of the 12th annual international conference on Mobile computing and network- ing*, pages 182–193, New York, NY, USA, 2006. ACM.

[34] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P.D. Mc- Daniel, and T.F. La Porta. "On cellular botnets: measuring the impact of malicious devices on a cellular network core," In E. Al-Shaer, S. Jha, and A.D. Keromytis, editors, *ACM Con- ference on Computer and Communications Security*, pages 223–234. ACM, 2009.

[35] M. Vuagnoux and S. Pasini. "Compromising electromag- netic emanations of wired and wireless keyboards," In *pro- ceedings of the 18th USENIX Security Symposium*, pages 1–16, Montreal, Canada, 2009. USENIX Association.

[36] D. Wagner and B. Schneier. "Analysis of the SSL 3.0 pro- tocol," In *WOEC'96: proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[37] C.V. Wright, L. Ballard, S.E. Coull, F. Monrose, and G.M. Masson. "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," In *SP '08: proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 35–49, Washington, DC, USA, 2008. IEEE Computer Society.

[38] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. "Stealthy video capturer: a new video-based spyware in 3g smartphones," In *WiSec '09: proceedings of the second ACM conference on Wireless network security*, pages 69–78, New York, NY, USA, 2009. ACM.

[39] K. Zhang and X. Wang. "Peeping Tom in the Neighbor- hood: Keystroke Eavesdropping on Multi-user Systems," In *USENIX Security '09: proceedings of the 18th USENIX Se- curity Symposium*, Montreal, Canada, 2008. USENIX Asso- ciation.

Appendix B: Overview of Status of Clouds

1. Introduction

The importance of simulation is well established with large programs, especially in Europe, USA, Japan and China supporting it in a variety of academic and government initiatives. The requirements and consequent architecture of large scale supercomputers is well understood although there are important challenges in meeting performance goals seen by international drives to reach first petascale (starting 15 years ago) and now exascale performance. Performance on closely coupled parallel simulations drives both hardware (low latency high bandwidth networks, high flop CPU's) and software that can exploit it. Grids covered both the linkage of such computers and broader computing facilities. This has spurred rise in high throughput computing, workflow and service oriented architectures (Software as a service); concepts of lasting value. Major data intensive applications like LHC data analysis highlighted the many important pleasingly parallel applications that these were a major driver of Grid and many task systems. Now the strong commercial interest is driving clouds and we can ask how they fit in? Clouds offer on-demand service (elasticity), economies of scale from sharing, a plethora of new jobs making clouds attractive for students & curricula and several challenges including security. Clouds lie in between grids and HPC supercomputers in their synchronization costs so all the high throughput jobs run on grids should perform well on clouds. In this paper, we suggest that there is a class of explicitly parallel jobs that do not need the highest performance interconnect and will have good performance and good user experience on clouds. We describe this in an application analysis in section2. Of course, HPC supercomputers can do "all applications" subject to reservations about limited I/O (disk) capabilities. However, they are overkill for many problems and it seems better to reserve such machines for the high-end applications that require them and use commodity cloud environments when appropriate.
We stress that clouds offer not just a new humongous data center architecture but striking new software models spurred by the competitive Platform as a Service PaaS market. In section 3 we focus on the possibilities suggested by MapReduce.

The term cloud is being in many ways so let's first define a public data center model that describes the major offerings of Microsoft, Amazon and Google. Their data centers are composed of containers of racks of servers which number between 10,000 and a million. Each server has 8 or more cpu cores and around 64GB of shared memory and one or more terabyte local disk drives. GPUs or other accelerators are not common. There is a network that allows messages to be routed between any two servers, but the bisection bandwidth of the network is very low and the network protocols implement the full TCP/IP stack so that every server can be a full Internet host with optimized traffic between users on the Internet and the servers in the cloud. In contrast supercomputer networks minimize interprocessor latency and maximize bisection bandwidth. Application data communications on a supercomputer generally take place over specialized physical and data link layers of the network and interoperation with the Internet is usually very limited.

2. A Cloud Defined

Each server in the data center is host to one or more virtual machines and the cloud runs a "fabric controller" which manages large sets of VMs fort scheduling and fault tolerance across the servers and acts as the operating system for the data center. An application running on the data center consists of one or more complete VM instances that implement a web service. The basic unit of scheduling involves the deployment of one or more entire operating systems, which is much slower than installing and starting an application on a running OS. Most large scale cloud services are intended to run 24x7, so this long start-up time is negligiblen although running a "batch" application on a large number of servers can be very inefficient because of the long time it may take to deploy all the needed VMs. Data in a data center is stored and distributed over many spinning disks in the cloud servers. This is a very different model than found in a large supercomputer, where data is stored in network attached storage. Local disks on the servers of supercomputers are not frequently used for data storage.

There are more types of clouds than is described by this public data center model. For example, to address a technical computing market, Amazon has introduced a specialized HPC cloud that uses a network with full bisection bandwidth and supports GPGPUs. The major commercial clouds offer higher level capabilities -- commonly termed Platform as a Service PaaS – built on a basic scalable IaaS Infrastructure as a Service. For technical computing, important platform components include tables, queues, database, monitoring, roles (Azure), and the cloud characteristic of elasticity (automatic scaling). MapReduce, which is discussed below, is another major platform service offered by these clouds. Currently the different clouds have different platforms although the Azure and Amazon platforms have many similarities. The Google Platform is targeted at scalable web applications and not as broadly used in technical computing community as Amazon or Azure, but it has been used on some very impressive projects. We expect more academic interest in PaaS as the value of platform capabilities become clearer.

"Private clouds" are small dedicated data centers that have various combinations of the properties above and typically use one of the four major open source (academic) cloud environments Eucalyptus, Nimbus, OpenStack and OpenNebula (Europe) which focus at the IaaS level with interfaces similar to Amazon. FutureGrid is an NSF research testbed for cloud technologies and it operates a grid of cloud deployments running on modest sized server clusters with support for all four academic IaaS. Private clouds do not fully support the interesting platform features of commercial clouds. Open source Hadoop and Twister offer MapReduce features similar to those on commercial cloud platforms and there are open source possibilities for platform features like queues (RabbitMQ, ActiveMQ) and distributed data management system (Apache Cassandra). However, there is no complete packaging of PaaS features available today for academic or private clouds. Thus interoperability

between private and commercial clouds is currently only at IaaS level where it is possible to reconfigure images between the different virtualization choices and there is an active cloud standards activity. The major commercial virtualization products such as VMware and Hyper-V are also important for private clouds but also do not have built-in PaaS capabilities.

3. Mapping Applications to Clouds

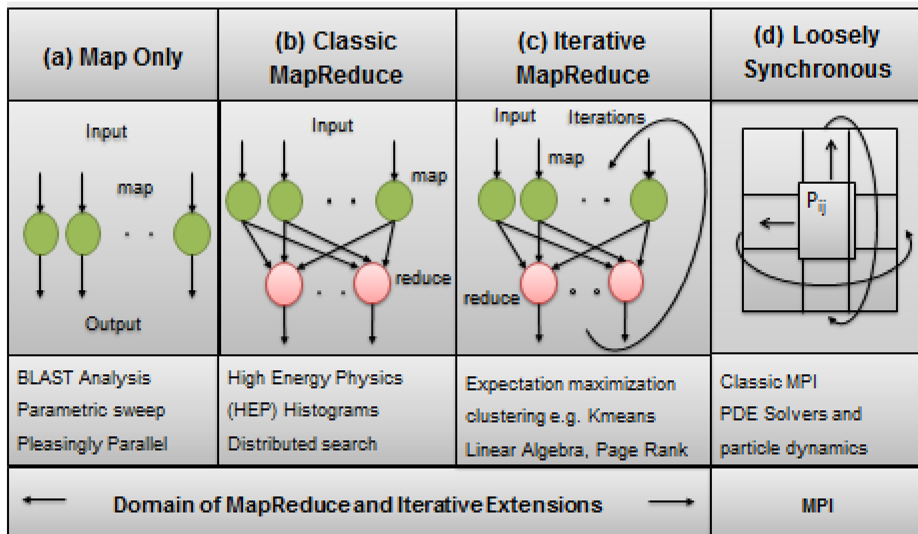


Fig 1: Forms of Parallelism and their application on Clouds and Supercomputers

Previously we discussed mapping applications to different hardware and software in terms of 5 "Application Architectures"[1] mainly aimed at simulations and extended it to data intensive computing [2, 3]. One category, synchronous, was popular 20 years ago but is no longer significant. It describes applications that can be parallelized with each decomposed unit running the identical machine instruction at each time. Another category, asynchronous is typically not important in practical computational science and engineering. There was also a category of metaproblems, which describe the domain supported by workflow with coarse grain interlinked components. The other categories were pleasingly parallel (essentially independent) and loosely (bulk) synchronous which are critical application classes that possibly combined in metaproblems describe the bulk of eScience. As mentioned above, pleasingly parallel problems whether parameter searches for simulations or analysis of independent data chunks (as in LHC events) are very suitable for clouds. Loosely synchronous problems include partial differential equation solution and particle dynamics and after parallelization, consist of a succession of compute-communication phases.

Clouds naturally exploit parallelism from multiple users or usages. The Internet of things will drive many applications of the cloud. It is projected that there will soon be 50 billion devices on the Internet. Most will be small sensors that send streams of information into the cloud where it will be processed and integrated with other streams and turned into knowledge that will help our lives in a million small and big ways. It is not unreasonable for us to believe that we will each have our own cloud-based personal agent that monitors all of the data about our life and anticipates our needs 24x7. The cloud will become increasing important as a controller of and resource provider for the Internet of Things. As well as today's use for smart phone and gaming console support, "smart homes" and "ubiquitous cities" and the current AFRL project build on this vision. We expect a growth in these areas with emergence of cloud supported/controlled robotics.

Looking at data intensive applications we can re-examine the pleasingly parallel and loosely synchronous category as shown in figure 1 above. This introduces map-only (identical to pleasing parallel), and separates off MapReduce and Iterative MapReduce classes from the large loosely synchronous class whose remaining members are the last sub category d) on the right of figure 1. This area requires HPC architectures with low latency high bandwidth interconnect. The MapReduce class b) consists of a single map (compute) phase followed by a reduction phase such as gathering together the results of queries following an Internet search or LHC data analysis (histogram) of different datasets. As implemented in Hadoop, one would normally communicate between Map and Reduce phases by writing and reading files. This leads to excellent fault tolerance and dynamic scheduling features. At SC11, there was some buzz in favor of data analytics and Hadoop but that this is not clearly reasonable as many data analysis (mining) applications involve kernels that do not fit Map only or MapReduce categories. Many algorithms including those with linear algebra (needing to be parallelized) fall into the category c) Iterative MapReduce in figure 1. Problems in this category consist of multiple (iterated) Map phases followed by reduction or collective operation communication phases. They do not have the many local communication messages typically needed in parallel simulations shown in fig 1d) but rather larger collective operations mixing compute and communication. We do not expect traditional MapReduce to be broadly useful but the Iterative extension is much more promising but the breadth of its applicability needs much more study. Iterative MapReduce is a programming model that can have the performance of MPI and the fault tolerance and dynamic flexibility of the original MapReduce. Open source Java Twister[4, 5] and Twister4Azure[6, 7] have been released as an Iterative MapReduce framework. Figure 2 compares Twister4Azure with Amazon and a classic HPC configuration on a map-only case while figure 3 shows Azure4Twister having a smooth execution structure and modest communication overhead (the uncolored gaps) on a parallel data analytics algorithm. We expect the commonly used expectation maximization (EM) approach used for example in Multidimensional Scaling MDS application of fig 3, to be particularly attractive for iterative MapReduce as EM can have large compute/communication ratios. Category c) extends the clear value of clouds in the categories a) and b) of figure 1.

3. CLOUDS AND REPOSITORIES

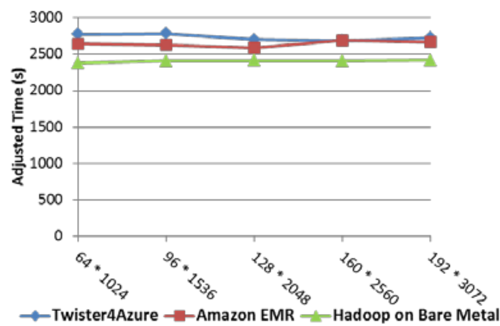


Fig 2: A Map Only example pairs sequence distances

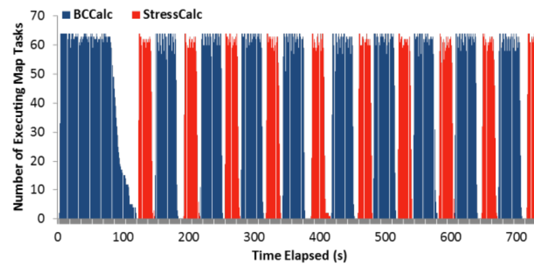


Fig 3: Parallel MDS on Azure4Twister showing communication (white) and two compute map phases

It is traditional to set up data repositories for large observational projects. Examples are EOSDIS (Earth Observation), GenBank (Genomics), NSIDC (Polar science), and IPAC (Infrared astronomy). The fourth paradigm implies an increase in data mining (analytics) based on such data and this implies repositories need computing as well as data. We also expect that one should bring the computing to the data and not vice versa. Thus we do not expect researchers to download large petabyte data samples to their local cluster; rather we expect repositories to be associated with cloud resources (as cheapest and elastic) that allow data analytics on demand. Again further work is needed here. Some questions include the data storage architecture (database or NOSQL) and how one supports mining of multidisciplinary science involving data from different fields stored in different clouds.

4. Cloud Research Issues

We list areas where is substantial research activity and where we can expect major changes.

- New applications such as Biomedical and bioinformatics applications where cloud architecture brings special challenges in the area of privacy (see later). Furthermore, Clouds have been attractive platforms for these applications as they are emerging big data areas and there is less history in using existing platforms.
- Sensor webs studied in this project are another emerging area where elastic nature of Clouds is well suited for the often bursty nature of sensor data.
- Big data applications based on new MapReduce or Iterative MapReduce environments are attractive on Clouds and result in broad research areas include addressing both programming and storage challenges. Latter include SQL and NOSQL models and the reconciliation of distributed data and centralized cloud computing
- Scheduling models optimized for MapReduce and for other Cloud usage modes such as scalable sensor webs (Sensor Grids or Clouds) where one has Clouds controlling and supporting a distributed Grid of sensors.
- Optimizing the run time features and performance for MapReduce and Iterative MapReduce. This includes new reduction primitives, polymorphic implementation on different systems with for example, exploitation of high performance networks as in classic MPI research.
- Support of federation of clouds and cloud bursting (typically the linkage of private and public Clouds) and on-demand cloud federation.
- New storage models such as data parallel HDFS and Hbase (Bigtable).
- NOSQL table structures such as Cassandra and commercial approaches such as Amazon SimpleDB and Azure Table.
- Economic models for an ecosystem with multiple cloud systems and CI.
- Research on Cloud software stacks. There is research at all levels of the software stack with two rather different emphasis areas. Research on systems that provide basic virtual machine provisioning, deployment and management. This includes Eucalyptus, Nimbus, OpenStack and OpenNebula with virtual networking as a distinct activity. At the other end are integration of capabilities to provide rich Platform-as-a-Service as offered by major commercial systems. Concepts such as appliances provide novel ways of delivering these capabilities.
- Clouds tend to achieve scalability by allowing faults. Research is needed on both, how to expose faults to users as well as services to build fault tolerant applications. Most research in HPC tends to be on forbidding faults; however Clouds highlight a different philosophy with resilient applications running on faulty systems.
- Green IT is naturally synergistic with Clouds and related research includes examining the impact of Cloud features on power use, including the cost of powering idle machines supporting elastic clouds as well as a application aware approaches to power management.

Security policies and mechanisms: Clouds tend to emphasis the need for quality security mechanisms due to the sharing of storage and computing. One research area investigates hybrid architectures with algorithms broken into two; a low cost but non privacy preserving part running on an intrinsically secure private clouds, and a time consuming but privacy preserving part executing on a public cloud. Genomic data (human) and other health records are demanding here. The concept of differential privacy and health data anonymization is an active research topic. As well as basic security for computing and storage there is research on privacy preserving search with the elegant but time consuming concept of Homomorphic Encryption which allows encrypted data to be searched by encrypted queries.

Standards: There are many important standard activities, from those specifying the basic virtual machine structure to higher-level standards defining the PaaS environment, for example, queue and table structures. Although there is some support for these standards – such as OCCI (from OGF) in OpenNebula and OpenStack – this area is still under development. NIST and IEEE are playing leadership roles.

5. References

1) Fox, G.C., R.D. Williams, and P.C. Messina, *Parallel computing works!* 1994: Morgan Kaufmann Publishers,
2) calculating all Jaliya Ekanayake, Thilina Gunarathne, Judy Qiu, Geoffrey Fox, Scott Beason, Jong Youl Choi, Yang Ruan, Seung-Hee Bae, and Hui Li, *Applicability of DryadLINQ to Scientific Applications*. January 30, 2010, Community Grids Laboratory, Indiana University.

3) Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, Mina Rho, and H. Tang, *Data Intensive Computing for Bioinformatics*. December 29, 2009.
4) SALSA Group. *Iterative MapReduce*. 2010 [accessed 2010 November 7]; Twister Home Page Available from: http://www.iterativemapreduce.org/.
5) J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, and G.Fox, *Twister: A Runtime for iterative MapReduce*, in *Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010*. 2010, ACM. Chicago, Illinois.
6) *Twister for Azure*. [accessed 2011 May 21]; Available from: http://salsahpc.indiana.edu/twister4azure/.
7) Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, and Judy Qiu, *Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure*, in *IEEE/ACM International Conference on Utility and Cloud Computing UCC 2011*. December 5-7, 2011. Melbourne Australia. http://www.cs.indiana.edu/~xqiu/scientific_applications_of_twister4azure_ucc_17_4.

Appendix B: Overiew of the Status of Clouds

1. Introduction

The importance of simulation is well established with large programs, especially in Europe, USA, Japan and China supporting it in a variety of academic and government initiatives. The requirements and consequent architecture of large scale supercomputers is well understood although there are important challenges in meeting performance goals seen by international drives to reach first petascale (starting 15 years ago) and now exascale performance. Performance on closely coupled parallel simulations drives both hardware (low latency high bandwidth networks, high flop CPU's) and software that can exploit it. Grids covered both the linkage of such computers and broader computing facilities. This has spurred rise in high throughput computing, workflow and service oriented architectures (Software as a service); concepts of lasting value. Major data intensive applications like LHC data analysis highlighted the many important pleasingly parallel applications that these were a major driver of Grid and many task systems. Now the strong commercial interest is driving clouds and we can ask how they fit in? Clouds offer on-demand service (elasticity), economies of scale from sharing, a plethora of new jobs making clouds attractive for students & curricula and several challenges including security. Clouds lie in between grids and HPC supercomputers in their synchronization costs so all the high throughput jobs run on grids should perform well on clouds. In this paper, we suggest that there is a class of explicitly parallel jobs that do not need the highest performance interconnect and will have good performance and good user experience on clouds. We describe this in an application analysis in section2. Of course, HPC supercomputers can do "all applications" subject to reservations about limited I/O (disk) capabilities. However, they are overkill for many problems and it seems better to reserve such machines for the high-end applications that require them and use commodity cloud environments when appropriate.
We stress that clouds offer not just a new humongous data center architecture but striking new software models spurred by the competitive Platform as a Service PaaS market. In section 3 we focus on the possibilities suggested by MapReduce.

The term cloud is being in many ways so let's first define a public data center model that describes the major offerings of Microsoft, Amazon and Google. Their data centers are composed of containers of racks of servers which number between 10,000 and a million. Each server has 8 or more cpu cores and around 64GB of shared memory and one or more terabyte local disk drives. GPUs or other accelerators are not common. There is a network that allows messages to be routed between any two servers, but the bisection bandwidth of the network is very low and the network protocols implement the full TCP/IP stack so that every server can be a full Internet host with optimized traffic between users on the Internet and the servers in the cloud. In contrast supercomputer networks minimize interprocessor latency and maximize bisection bandwidth. Application data communications on a supercomputer generally take place over specialized physical and data link layers of the network and interoperation with the Internet is usually very limited.

2. A Cloud Defined

Each server in the data center is host to one or more virtual machines and the cloud runs a "fabric controller" which manages large sets of VMs fort scheduling and fault tolerance across the servers and acts as the operating system for the data center. An application running on the data center consists of one or more complete VM instances that implement a web service. The basic unit of scheduling involves the deployment of one or more entire operating systems, which is much slower than installing and starting an application on a running OS. Most large scale cloud services are intended to run 24x7, so this long start-up time is negligiblen although running a "batch" application on a large number of servers can be very inefficient because of the long time it may take to deploy all the needed VMs. Data in a data center is stored and distributed over many spinning disks in the cloud servers. This is a very different model than found in a large supercomputer, where data is stored in network attached storage. Local disks on the servers of supercomputers are not frequently used for data storage.

There are more types of clouds than is described by this public data center model. For example, to address a technical computing market, Amazon has introduced a specialized HPC cloud that uses a network with full bisection bandwidth and supports GPGPUs. The major commercial clouds offer higher level capabilities -- commonly termed Platform as a Service PaaS – built on a basic scalable IaaS Infrastructure as a Service. For technical computing, important platform components include tables, queues, database, monitoring, roles (Azure), and the cloud characteristic of elasticity (automatic scaling). MapReduce, which is discussed below, is another major platform service offered by these clouds. Currently the different clouds have different platforms although the Azure and Amazon platforms have many similarities. The Google Platform is targeted at scalable web applications and not as broadly used in technical computing community as Amazon or Azure, but it has been used on some very impressive projects. We expect more academic interest in PaaS as the value of platform capabilities become clearer.

“Private clouds” are small dedicated data centers that have various combinations of the properties above and typically use one of the four major open source (academic) cloud environments Eucalyptus, Nimbus, OpenStack and OpenNebula (Europe) which focus at the IaaS level with interfaces similar to Amazon. FutureGrid is an NSF research testbed for cloud technologies and it operates a grid of cloud deployments running on modest sized server clusters with support for all four academic IaaS. Private clouds do not fully support the interesting platform features of commercial clouds. Open source Hadoop and Twister offer MapReduce features similar to those on commercial cloud platforms and there are open source possibilities for platform features like queues (RabbitMQ, ActiveMQ) and distributed data management system (Apache Cassandra). However, there is no complete packaging of PaaS features available today for academic or private clouds. Thus interoperability between private and commercial clouds is currently only at IaaS level where it is possible to reconfigure images between the different virtualization choices and there is an active cloud standards activity. The major commercial virtualization products such as VMware and Hyper-V are also important for private clouds but also do not have built-in PaaS capabilities.

3. Mapping Applications to Clouds

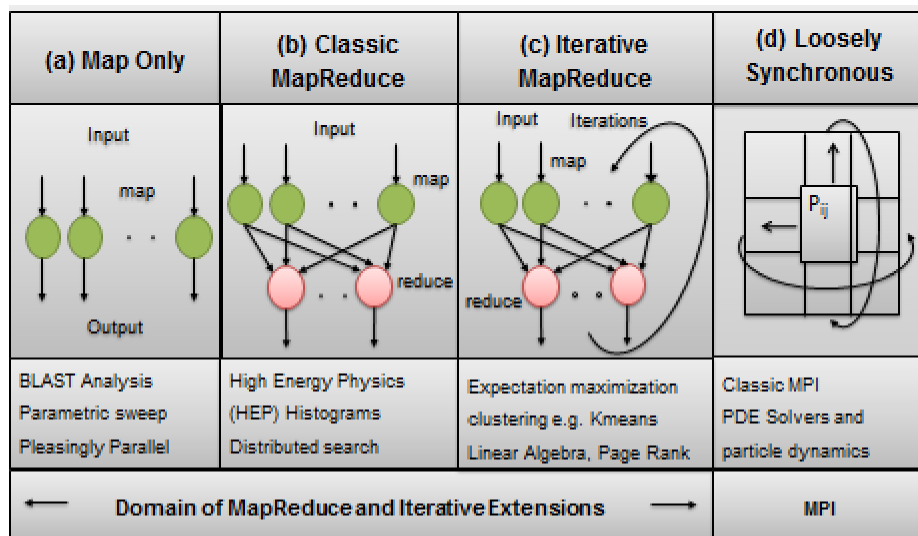


Fig 1: Forms of Parallelism and their application on Clouds and Supercomputers

Previously we discussed mapping applications to different hardware and software in terms of 5 “Application Architectures”[1] mainly aimed at simulations and extended it to data intensive computing [2, 3]. One category, synchronous, was popular 20 years ago but is no longer significant. It describes applications that can be parallelized with each decomposed unit running the identical machine instruction at each time. Another category, asynchronous is typically not important in practical computational science and engineering. There was also a category of metaproblems, which describe the domain supported by workflow with coarse grain interlinked components. The other categories were pleasingly parallel (essentially independent) and loosely (bulk) synchronous which are critical application classes that possibly combined in metaproblems describe the bulk of eScience. As mentioned above, pleasingly parallel problems whether parameter searches for simulations or analysis of independent data chunks (as in LHC events) are very suitable for clouds. Loosely synchronous problems include partial differential equation solution and particle dynamics and after parallelization, consist of a succession of compute-communication phases.

Clouds naturally exploit parallelism from multiple users or usages. The Internet of things will drive many applications of the cloud. It is projected that there will soon be 50 billion devices on the Internet. Most will be small sensors that send streams of information into the cloud where it will be processed and integrated with other streams and turned into knowledge that will help our lives in a million small and big ways. It is not unreasonable for us to believe that we will each have our own cloud-based personal agent that monitors all of the data about our life and anticipates our needs 24x7. The cloud will become increasing important as a controller of and resource provider for the Internet of Things. As well as today's use for smart phone and gaming console support, "smart homes" and "ubiquitous cities" and the current AFRL project build on this vision. We expect a growth in these areas with emergence of cloud supported/controlled robotics.

Looking at data intensive applications we can re-examine the pleasingly parallel and loosely synchronous category as shown in figure 1 above. This introduces map-only (identical to pleasing parallel), and separates off MapReduce and Iterative MapReduce classes from the large loosely synchronous class whose remaining members are the last sub category d) on the right of figure 1. This area requires HPC architectures with low latency high bandwidth interconnect. The MapReduce class b) consists of a single map (compute) phase followed by a reduction phase such as gathering together the results of queries following an Internet search or LHC data analysis (histogram) of different datasets. As implemented in Hadoop, one would normally communicate between Map and Reduce phases by writing and reading files. This leads to excellent fault tolerance and dynamic scheduling features. At SC11, there was some buzz in favor of data analytics and Hadoop but that this is not clearly reasonable as many data analysis (mining) applications involve kernels that do not fit Map only or MapReduce categories. Many algorithms including those with linear algebra (needing to be parallelized) fall into the category c) Iterative MapReduce in figure 1. Problems in this category consist of multiple (iterated) Map phases followed by reduction or collective operation communication phases. They do not have the many local communication messages typically needed in parallel simulations shown in fig 1d) but rather larger collective operations mixing compute and communication. We do not expect traditional MapReduce to be broadly useful but the Iterative extension is much more promising but the breadth of its applicability needs much more study. Iterative MapReduce is a programming model that can have the performance of MPI and the fault tolerance and dynamic flexibility of the original MapReduce. Open source Java Twister[4, 5] and Twister4Azure[6, 7] have been released as an Iterative MapReduce framework. Figure 2 compares Twister4Azure with Amazon and a classic HPC configuration on a map-only case while figure 3 shows Azure4Twister having a smooth execution structure and modest communication overhead (the uncolored gaps) on a parallel data analytics

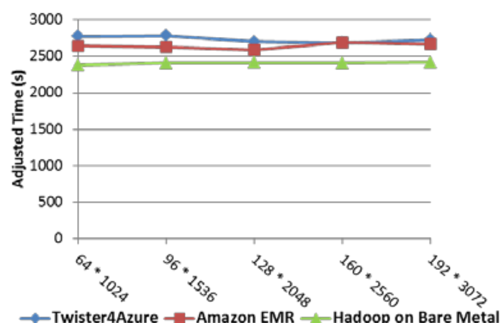


Fig 2: A Map Only example pairs sequence distances

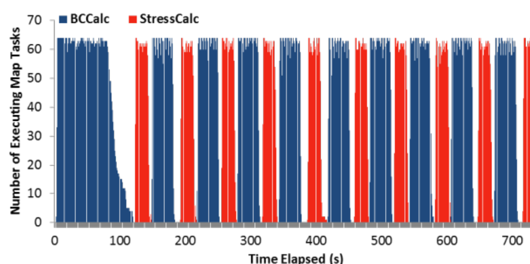


Fig 3: Parallel MDS on Azure4Twister showing communication (white) and two compute map phases

algorithm. We expect the commonly used expectation maximization (EM) approach used for example in Multidimensional Scaling MDS application of fig 3, to be particularly attractive for iterative MapReduce as EM can have large compute/communication ratios. Category c) extends the clear value of clouds in the categories a) and b) of figure 1.

3. CLOUDS AND REPOSITORIES

It is traditional to set up data repositories for large observational projects. Examples are EOSDIS (Earth Observation), GenBank (Genomics), NSIDC (Polar science), and IPAC (Infrared astronomy). The fourth paradigm implies an increase in data mining (analytics) based on such data and this implies repositories need computing as well as data.

We also expect that one should bring the computing to the data and not vice versa. Thus we do not expect researchers to download large petabyte data samples to their local cluster; rather we expect repositories to be associated with cloud resources (as cheapest and elastic) that allow data analytics on demand. Again further work is needed here. Some questions include the data storage architecture (database or NOSQL) and how one supports mining of multidisciplinary science involving data from different fields stored in different clouds.

4. Cloud Research Issues

We list areas where is substantial research activity and where we can expect major changes.

- New applications such as Biomedical and bioinformatics applications where cloud architecture brings special challenges in the area of privacy (see later). Furthermore, Clouds have been attractive platforms for these applications as they are emerging big data areas and there is less history in using existing platforms.
- Sensor webs studied in this project are another emerging area where elastic nature of Clouds is well suited for the often bursty nature of sensor data.
- Big data applications based on new MapReduce or Iterative MapReduce environments are attractive on Clouds and result in broad research areas include addressing both programming and storage challenges. Latter include SQL and NOSQL models and the reconciliation of distributed data and centralized cloud computing
- Scheduling models optimized for MapReduce and for other Cloud usage modes such as scalable sensor webs (Sensor Grids or Clouds) where one has Clouds controlling and supporting a distributed Grid of sensors.
- Optimizing the run time features and performance for MapReduce and Iterative MapReduce. This includes new reduction primitives, polymorphic implementation on different systems with for example, exploitation of high performance networks as in classic MPI research.
- Support of federation of clouds and cloud bursting (typically the linkage of private and public Clouds) and on-demand cloud federation.
- New storage models such as data parallel HDFS and Hbase (Bigtable).
- NOSQL table structures such as Cassandra and commercial approaches such as Amazon SimpleDB and Azure Table.
- Economic models for an ecosystem with multiple cloud systems and CI.
- Research on Cloud software stacks. There is research at all levels of the software stack with two rather different emphasis areas. Research on systems that provide basic virtual machine provisioning, deployment and management. This includes Eucalyptus, Nimbus, OpenStack and OpenNebula with virtual networking as a distinct activity. At the other end are integration of capabilities to provide rich Platform-as-a-Service as offered by major commercial systems. Concepts such as appliances provide novel ways of delivering these capabilities.
- Clouds tend to achieve scalability by allowing faults. Research is needed on both, how to expose faults to users as well as services to build fault tolerant applications. Most research in HPC tends to be on forbidding faults; however Clouds highlight a different philosophy with resilient applications running on faulty systems.
- Green IT is naturally synergistic with Clouds and related research includes examining the impact of Cloud features on power use, including the cost of powering idle machines supporting elastic clouds as well as a application aware approaches to power management.

Security policies and mechanisms: Clouds tend to emphasis the need for quality security mechanisms due to the sharing of storage and computing. One research area investigates hybrid architectures with algorithms broken into two; a low cost but non privacy preserving part running on an intrinsically secure private clouds, and a time consuming but privacy preserving part executing on a public cloud. Genomic data (human) and other health records are demanding here. The concept of differential privacy and health data anonymization is an active research topic. As well as basic security for computing and storage there is research on privacy preserving search with the elegant but time consuming concept of Homomorphic Encryption which allows encrypted data to be searched by encrypted queries.

Standards: There are many important standard activities, from those specifying the basic virtual machine structure to

higher-level standards defining the PaaS environment, for example, queue and table structures. Although there is some support for these standards – such as OCCI (from OGF) in OpenNebula and OpenStack – this area is still under development. NIST and IEEE are playing leadership roles.

5. References

8) Fox, G.C., R.D. Williams, and P.C. Messina, *Parallel computing works!* 1994: Morgan Kaufmann Publishers,
9) calculating all Jaliya Ekanayake, Thilina Gunarathne, Judy Qiu, Geoffrey Fox, Scott Beason, Jong Youl Choi, Yang Ruan, Seung-Hee Bae, and Hui Li, *Applicability of DryadLINQ to Scientific Applications*. January 30, 2010, Community Grids Laboratory, Indiana University.
10) Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, Mina Rho, and H. Tang, *Data Intensive Computing for Bioinformatics*. December 29, 2009.
11) SALSA Group. *Iterative MapReduce*. 2010 [accessed 2010 November 7]; Twister Home Page Available from: http://www.iterativemapreduce.org/.
12) J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, and G.Fox, *Twister: A Runtime for iterative MapReduce*, in *Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010*. 2010, ACM. Chicago, Illinois.
13) *Twister for Azure*. [accessed 2011 May 21]; Available from: http://salsahpc.indiana.edu/twister4azure/.
14) Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, and Judy Qiu, *Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure*, in *IEEE/ACM International Conference on Utility and Cloud Computing UCC 2011*. December 5-7, 2011. Melbourne Australia. http://www.cs.indiana.edu/~xqiu/scientific_applications_of_twister4azure_ucc_17_4.pd