AFRL-RH-WP-TR-2012-0094

ANALYST PERFORMANCE MEASURES, VOLUME I: PERSISTENT SURVEILLANCE DATA PROCESSING, STORAGE AND RETRIEVAL

Ed Wasser
Dr. Sanjay Boddhu
Niranjan Kode
Telford Berkey
Qbase, LLC
2619 Commons Boulevard
Dayton OH 45431

SEPTEMBER 2011
Final Report

AIR FORCE RESEARCH LABORATORY
711™ HUMAN PERFORMANCE WING,
HUMAN EFFECTIVENESS DIRECTORATE,
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

//Signature//	//Signature//
BRIAN TSOU	LOUISE CARTER, Ph.D
Work Unit Manager	Forecasting Division
Human Analyst Augmentation Branch	Human Effectiveness Directorate
	711th Human Performance Wing
	Air Force Research Laboratory

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-Sep-2011	2. REPORT TYPE Final	3. **DATES COVERED** *(From - To)* June 2009 – September 2011
4. TITLE AND SUBTITLE Analyst Performance Measures, Volume I: Persistent Surveillance Data Processing, Storage and Retrieval		**5a. CONTRACT NUMBER** FA8650-09-6939 0005
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 65502F
6. **AUTHOR(S)** Ed Wasser; Dr. Sanjay Boddhu; Niranjan Kode; Telford Berkey		**5d. PROJECT NUMBER** 7184
		5e. TASK NUMBER 0005
		5f. WORK UNIT NUMBER 7184X18W
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Qbase, LLC 2619 Commons Boulevard Dayton OH 45431		8. **PERFORMING ORGANIZATION REPORT NUMBER**
9. **SPONSORING / MONITORING AGENCY NAME(S) & ADDRESS(ES)** Air Force Materiel Command Air Force Research Laboratory 711th Human Performance Wing Human Effectiveness Directorate Forecasting Division Human-Analyst Augmentation Branch Wright-Patterson AFB OH 45433		10. **SPONSOR/MONITOR'S ACRONYM(S)** 711 HPW/RHXM
		11. **SPONSOR/MONITOR'S REPORT NUMBER(S)** AFRL-RH-WP-TR-2012-0094

12. **DISTRIBUTION / AVAILABILITY STATEMENT**
Distribution A: Approved for public release; distribution is unlimited.

13. **SUPPLEMENTARY NOTES**
88ABW/PA cleared on 10 July 2012, 88ABW-2012-3865

14. **ABSTRACT**
The Air Force is seeking innovative architectures to process and store massive data sets in a flexible and open platform – with real-time, forensic and predictive analytic capabilities. The architecture and design work described herein is being developed to anticipate applications that are highly relevant for both national security and environmental defense. Characteristics of this architecture include robustness, flexibility, and scalability. To accomplish this, novel event collaboration message based architecture is proposed.

15. SUBJECT TERMS
Persistent Sensor Storage Architecture, Extensible Messaging & Presence Protocol, Geographic Information System, Distributed Common Ground System, Cursor on Target

16. SECURITY CLASSIFICATION OF: UNCLASSIFED			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Mr. Brian Tsou
a. **REPORT** U	b. **ABSTRACT** U	c. **THIS PAGE** U	SAR	131	**19b. TELEPHONE NUMBER** *(include area code)* NA

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

THIS PAGE LEFT INTENTIONALLY BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Central State University and Qbase appreciate greatly the opportunity to contribute to solving the important problems that this Research & Development program addresses. In addition, Qbase would like to acknowledge and express appreciation for the input, guidance and continued support of Dr. Robert Williams and Mr. Brian Tsou of AFRL.

1.0 SUMMARY

Over the past several years, persistent surveillance devices have advanced in design and become increasingly useful to our nation. Federal, state and local agencies, and the private sector, are deploying new remote-sensing devices that can capture ever-increasing amounts of data, to improve significantly both situational awareness and decision-making. These sensing devices can contribute greatly to:

- Environmental and climate monitoring;
- Utility generation and distribution;
- Intelligence gathering and reconnaissance;
- Law enforcement; and
- Premises security.

However, the systems currently available for data capture and storage are disconnected and disjointed, which sharply decreases their utility. Also, they have very limited storage capacity. These shortcomings make it difficult to analyze and correlate data captured across multiple sensors. And, it's impossible to store enough data online, for long enough, that analysts can efficiently detect patterns and fully understand potentially important changes in the physical environment, geographic terrain and atmospheric and climatic conditions.

Compounding the problem is the physical and virtual separation of expert analysts, divided by the same "stovepipes" where the sensing data is stored. Consequently, experts in climate change, national defense and other domains now lack a reliable, collaborative pathway into the significant data collections gathered from remote-sensing systems. The experts could improve their interpretations of data sets markedly – and improve decision-making -- if they could access and share remote-sensing data broadly, utilizing a common interface for observation and analysis. Much like modern medical radiology now allows luminary diagnosticians to read radiographic images anywhere in the world, a new remote-sensing architecture and platform could broaden access to the best experts, regardless of their physical location. Additionally, a common platform could offer better opportunities for securing information more comprehensively, with tiered access on a more consistent, need-to-know basis than is now possible with the existing, disjointed systems.

The U.S. Air Force Research Laboratory (AFRL) has contracted with Qbase and Central State University to research and design technological solutions to address these important challenges. The Air Force is seeking innovative architectures to process and store massive data sets in a flexible and open platform – with real-time, forensic and predictive analytic capabilities. All of the design work is being undertaken to anticipate applications that are highly relevant for both national security and environmental defense.

The initial phase of this project focused on the architecture and design of an architecture — the Persistent Sensor Storage Architecture (PSSA) to support these types of applications. In addition, a reference implementation of the architecture was developed to demonstrate the capabilities and applications of the architecture. The reference implementation included two demonstration capabilities, one for environmental data exploitation and the other, for video surveillance. Briefly, the proposed new architecture features:

- High capacity, high throughput computing at attractive costs, well below comparable production-system operations;
- Low cost commodity hardware, coupled with a custom high-performance parallel file system and a flexible software framework;
- A fault-tolerant, high performance parallel file system that will allow very large volumes of sensor data to be maintained in online storage for extended periods of time, at a cost that approximates that of offline storage;
- And, a light-weight infrastructure that allows ingestion of sensor data and new exploitation components, to be developed quickly and easily, and then plugged into the system — plus rapid deployment with existing sensors and technologies.

At the Tec^Edge Innovation and Collaboration Center, AFRL has already leveraged the reference implementation that Qbase created in this initiative as part of the Summer at the Edge (SATE) program. College undergraduate and graduate students utilized the PSSA reference implementation to demonstrate applications of layered sensing to monitor disease outbreak among farm animals for the Ohio Department of Agriculture (ODA) and to demonstrate the use of smart phones to collect and display sensor data.

Based on the work already accomplished in this project, four examples of applications that could be built from the new architecture are outlined below:

Persistent Surveillance Platforms – Improvised Explosive Device (IED) Detection and Intervention. Faster identification and disarming of IEDs could save the lives in theatres of operations. By comparing prior data to live data from visual and ground penetrating sensors, change detection algorithms could more readily alert analysts to the potential locations of IEDs. Live surveillance video and infrared imagery could highlight personnel activity patterns at locations and times that are known to be indicative of IED placement. After discovery of IEDs on the ground, historical video analysis could enhance discovery of who the bombers were, and from where they travelled to plant devices.

Environmental Remote Sensing – Maritime Oil Spill Response. The new computing platform could combine satellite imagery, data from remotely deployed environmental/meteorological buoys and on-scene sensor data. The integrated information would provide decision-makers who are responsible for directing spill containment and clean-up actions a common-operating-picture of the crisis. Analysts and others could track and trend more effectively surface and sub-surface oil plume propagation, in combination with information about threatened locales, assets and wildlife. Barriers and booms could be more optimally deployed to protect eco-sensitive regions prior to contamination.

Video Surveillance – Major Metropolitan Transit Systems. Persons carrying suspect parcels could be identified more easily, and, parcels that become separated from their owners could be highlighted, as well as providing enhanced tracking of where the departing owners went. Abnormal patterns of traveler behavior could trigger enhanced video monitoring to include facial recognition and comparison to high-interest suspect databases. Immediate replay of events could identify pre-event target definition for searches throughout the system's video archive for similar targets or behaviors.

Environmental Remote Sensing – Freshwater Resource Management. Combining satellite and ground-based sensor data, systems could report actual irrigation and domestic water needs to determine levels of necessary reservoir-release rates -- versus arbitrary quotas. More accurate and current evaporation rates and oxygen levels in downstream waterways could determine the need for increased flow rates of freshwater -- or justify retention for later discharge, to ensure susceptible aquatic life is sustained.

This Interim Report also summarizes key components of a proposed $3 million second phase. It would deliver initial operational functionality of the PSSA solution for a selected U.S. military service or Federal civilian agency within one year of award.

2.0 INTRODUCTION

2.1 Overview and Scope

This document addresses the technology, architecture, and design issues associated with the processing, storage and retrieval of large amounts of sensor data generated in rapidly expanding and changing persistent surveillance and layered sensing scenarios. Newly developed sensors for wide-area persistent staring continue to increase the size, density and frequency of data being captured. This, coupled with the ever increasing deployment of ground based sensors, requires new and innovative techniques for storing, processing, and querying the massive amounts of data being generated. The large amount of data overwhelms traditional processing, storage and retrieval systems.

Today, limited pre-processed data can be viewed in real time, but if a segment of data from a past date and time is required for analysis, the user generally cannot access it because the existing real-time systems do not store it. Further, more complex processing or exploitation algorithms cannot be run against the data stream for analysis, due to the enormous size and relatively slow throughput rate of the data. Additionally, appropriate metadata and geo- spatial/temporal search & retrieval techniques do not exist for these datasets.

This document will describe the capability to capture a deep archive of sensor data from a variety of different sensor types indexed in space and time that can be quickly and easily searched to support real-time, forensic, and anticipatory exploitation algorithms. Metadata and results generated by these algorithms can also be stored and indexed with the original sensor data to provide a rich set of data for communicating the quality level of the data as well as how much the results of the exploitation algorithms can be trusted. The data captured by this system can also be used for research, simulation and demonstration purposes.

There are a wide variety of scenarios for persistent surveillance and layered sensing ranging from military support and homeland security to commercial site security and environmental monitoring. However, in all of these scenarios there are some common requirements that this architecture attempts to address:

- Continuous long-term storage of sensor data
- Real-time monitoring and exploitation of the sensor data stream
- Rapid temporal and spatial retrieval and processing of stored sensor data
- Flexible ingestion interfaces to support new sensors and sensor platforms as they become available
- Standards-based visualization interfaces to support new and existing visualization tools
- Temporal and spatial consistency in the data captured by different sensors with overlapping fields of view.

The PSSA presented in this document is intended to co-exist and enhance the capabilities of existing sensor data processing architectures not replace them. Through the use of industry standard interfaces and a plug and play approach to extending the system, it is expected that this storage architecture can be extended and leveraged as the core storage and processing platform for all future persistent surveillance or layered sensing needs. While many existing sensor architectures focus on either real-time exploitation or forensic analysis, the architecture we put forward in this document is intended to address both needs in an optimal manner and to provide a framework to support the pressing need for performing predictive analysis using sensor data from a variety of different sources.

With this architecture we also provide the support necessary to communicate the quality of the information captured by the sensor and subsequently generated by exploitation algorithms as well as the level of trust associated with the data based on trust metrics captured by the processing components of the system, and functionality that provides secure, tiered access and defined entitlement rights.

2.2 Scope

This document focuses on concepts and issues that are relevant to a Persistent Sensor Storage (PSS) system in a layered sensing environment and presents a logical architecture that addresses many of the needs that are not being met by existing “stovepipe” sensor management systems. The document also describes how the architecture can be applied to address different types of scenarios in which persistent surveillance is required or desirable.

As part of the research effort to define and test the concepts developed for this architecture, Qbase developed a prototype implementation to demonstrate many of the concepts outlined in this document. This prototype includes basic implementations of several of the core components of the architecture including the directory service component, a remote launcher component, a streaming media storage component, a text/metadata storage component, the messaging system, and the PSS Software Development Kit (SDK). Using the PSS SDK, we also developed reference implementations of ingestion services, application services, and dissemination services.

This section describes the architecture that was developed over the course of the project to meet the needs and objectives discussed earlier. We start out with a high-level architectural overview describing the concepts associated with Persistent Surveillance Data Processing, Storage and Retrieval. We then present a more detailed view of the issues and concerns that we identified and, specifically, what functionality we determined to be required to field a system that addresses these issues.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Architecture Overview

The following diagram provides a conceptual overview of the key processes required of a sensor processing system:

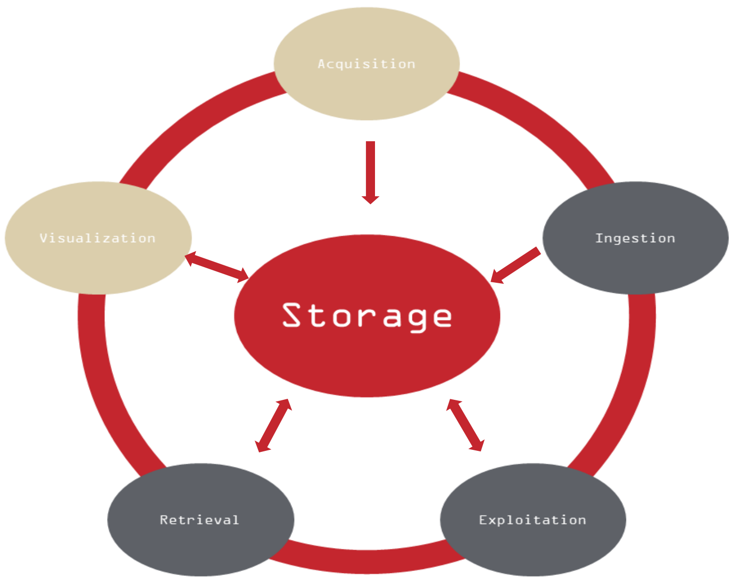


Figure 1: Conceptual View of the PSSA

The PSSA supports persistent surveillance activities by providing a platform for storing, processing, and disseminating data captured via remote sensors. As depicted above, storage is central to our architecture. The storage subsystem provides a scalable, high performance, fault- tolerant and flexible repository for all of the sensor data ingested into the system. The ring around the storage subsystem represents a messaging capability that allows sensor data to be simultaneously communicated synchronously or asynchronously to all of the other services surrounding the storage subsystem.

Services can be "plugged" into this messaging backbone (or "message bus") as needed to support new sensors and sensor processing without having to rebuild or even shut down the system. Services defined as part of the PSSA (dark gray items) provide the means for sensor data to be ingested, exploited, stored and retrieved. Although storage is central to the PSSA, the other services are optional and can be plugged into or removed from the PSS system without affecting any other service or requiring shutdown or restart of the system. Acquisition and Visualization Services (beige items) can be developed for specific sensor applications and connected directly to the PSS message bus or communicate with the system via standard or customized Ingestion and Retrieval services. These services are described in more detail below:

3.1.1. Acquisition

Acquisition platforms are the hardware and software required to capture the sensor data and metadata and transform it into a digital data stream (or file) that can be processed and stored within the PSSA. This includes the physical sensors as well as any onboard or ground based processing components used to transform sensor data and associated metadata from the physical sensor device to a digital electronic format. This component of the sensor processing system is typically provided by the sensor manufacturer or operator. As a result, the PSSA must provide a flexible mechanism for interacting with acquisition components to process and store the sensor data captured by these components. In the PSSA, the interaction with the acquisition platform is the responsibility of the ingestion services.

3.1.2. Ingestion

The ingestion services are responsible for importing the digital data generated by sensors and their associated processing systems into the PSS system. These services are the point of entry for external data to the PSS system. An ingestion end-point is required for each type of digital data stream fed into the system. These end-points support a multitude of different digital data formats (encoded video streams, compressed imagery, encoded audio streams, text, etc.) and protocols (Transmission Control Protocol (TCP)/Internet Protocol (IP), Hypertext Transport Protocol (HTTP), Real-Time Streaming Protocol (RTSP), etc.).

If a new sensor is added to the system, either an existing end-point is used (if the data format and protocol is already supported), or a new end-point is developed to support the new data format and protocol. After data is ingested by the ingestion service, it is immediately made available to the other components of the PSSA. Thus, an application service can operate on the data immediately to perform any exploitation operations while a dissemination service waiting for the raw data can immediately push it out to a visualization platform. There are two methods for ingestion of data into the system:

- **Asynchronous** (push) – this type of ingestion relies on the acquisition system to provide data. The acquisition system connects to the ingestion component and sends sensor data as it becomes available. Alternatively the acquisition system “broadcasts” sensor data to a well-known destination (broadcast socket, file system directory, etc.) that the ingestion component is monitoring.
- **Synchronous** (pull) – this type of ingestion component periodically polls the acquisition system for new data. The ingestion component connects to the acquisition system on a periodic basis to get the latest sensor data.

The ingestion component is responsible for formatting, packaging and publishing the sensor data being brought into the system so that it can be accessed by other components of the system. Processing and storage of the sensor data is handled by application services and data services specifically designed to perform those tasks.

3.1.3. Exploitation

Exploitation within the PSSA is performed by Application Service Components. Application services are the core processing components of the PSSA. These services subscribe to specific sensor data in order to assess, enhance, and transform it. Transformation includes such tasks as projection of data to a new coordinate system or combining data from multiple sources to generate new data (e.g. using satellite imagery and weather data to calculate moisture flux) or utilizing reference imagery to create a geo-registered image. Enhancement includes such tasks as running image processing algorithms to improve the color, brightness, contrast, sharpness, etc. of the data or utilizing external data sources such as Differential Geographic Positioning System (DGPS) or Continuously Operating Reference Stations (CORS) to improve the accuracy of the Geographic Positioning System (GPS) location information.

Assessment includes such tasks as examining the data to provide quality and trust metrics, applying algorithms to detect objects, utilizing data collected previously or by other sensors to detect moving objects, etc. Application services also include purpose built algorithms such as target identification, tracking algorithms, threshold alerting, etc. required for specific sensor applications. These services can be plugged into the messaging architecture and immediately begin processing data streams generated by the ingestion services or other exploitation services.

One of the goals of our architecture is to allow exploitation algorithms to operate on the data without impacting the real-time performance of the system. To accomplish this, the architecture will support simultaneous distribution of data to visualization components and exploitation components as well as the dynamic distribution of processing across nodes in a cluster of computing systems.

The architecture also supports exploitation tuning capabilities. These are control messages that can be sent to the exploitation algorithm to adjust the way it performs its processing. For example: adjusting the sensitivity/quality of the algorithm, overriding automatic/default settings, zeroing-in on particularly interesting data for enhanced processing.

The exploitation components are responsible for publishing the data resulting from the application of their algorithm(s) so that other components of the system can access it.

3.1.4. Retrieval

Retrieval of data from the PSSA for distribution to external systems or for visualization by end-users is handled by dissemination services. Dissemination services provide data to end users or to external systems through the use of custom or standards-based interfaces.

The dissemination components are responsible for providing an efficient mechanism for locating and retrieving data stored in the PSSA-based system (typically using PSSA Data Storage Services) and for either displaying this data to the end-user or for making it available to an external system — such as Google Earth, the Video LAN Client (VLC) media player, and the Environmental Systems Research Institute corporation's (ESRI) ArcGIS product — to process or display.

Dissemination components may implement standards-based interfaces as well as purpose-built interfaces to meet the specific needs of consumers of the persistent surveillance data. Examples of standards based interfaces include Open Geospatial Consortium (OGC) Web Feature Service (WFS) for vector-based Geographic Information System (GIS) data, OGC Web Mapping Service (WMS) for raster-based GIS data, RTSP for streaming video, etc. These services also provide intelligent caching of the data to minimize processing time associated with retrieving data. Open source GIS Servers such as GeoServer and MapServer can be easily modified to act as PSSA dissemination services by plugging these tools into the PSSA messaging system using the PSSA SDK.

3.1.5. Visualization

There are numerous commercial and open source GIS and media visualization tools that can be used to retrieve and display PSS data by virtue of the support of OGC and other industry standards provided by dissemination components. Examples of such tools include Google Earth, Quantum GIS, VLC Media Player, NASA WorldWind, ESRI ArcGIS and many others.

Open source GIS and media visualization software can also be modified to display data ingested and processed by PSSA components in near real-time by connecting this software directly into the PSSA messaging system using the PSSA SDK.

3.1.6. Storage

Data Storage Services (DSS) provide a high-level interface to the storage and retrieval of data so that processing components do not have to know the details of how or where the data is stored. Three types of DSS are defined in the PSSA:

- Media Storage Services are used to store and retrieve raster data and long-running/continuous streaming media data to the high speed parallel file system. The media storage services work in conjunction with the Internal Data Storage services to make sure raster and streaming media data is spatially and temporally indexed in the spatial database.
- Internal Data Storage services are used to store and retrieve text-based sensor data, metadata and vector data ingested by or generated by other components of the PSSA system into a spatial database. Internal Data Storage services work in conjunction with the Media Storage Services to insure that raster data stored in the high speed parallel file system is properly linked to its associated metadata.
- External Gateways are Data Storage services designed to store and retrieve data from external systems whose data is required to perform processing by the processing components of the PSSA system.

The core of the PSSA is the storage subsystem. This storage subsystem is a high-performance, fault-tolerant, clustered array of storage nodes that includes a spatial database for temporal/spatial indexing of the data and a high-speed parallel file system for storing and retrieving large blocks of raster data and media files.

The goal of this subsystem is to provide data to any component in the system that requests it as quickly as possible. At the same time, it must be capable of storing the large amounts of data being brought into the system from hundreds or perhaps thousands of sensors.

The key to this solution is the use of clustered storage nodes that work cooperatively to provide redundant high-speed storage of data. Low-cost Serial Attached SCSI /Serial Advanced Technology Attachment (SAS/SATA) disk arrays, solid state disks and in-memory data caching provide storage options to address varying requirements for the cost, size, performance, and volatility of the data. The core of the storage subsystem is a spatiotemporal database that provides the ability to perform queries and combine and compare data based upon time and space relationships.

3.1.7. Summary

Sensor acquisition platforms are responsible for capturing data and converting it into a digital data format. These systems are typically provided by the sensor manufacturer. Sensor data enters the PSSA system via standards based or purpose built ingestion services. These services are responsible for providing a common spatiotemporal frame of reference and for packaging the data in a standard format that can be propagated throughout the system.

A common spatiotemporal frame of reference is necessary to allow sensor data from many independent sensors to be related in space and time for exploitation, retrieval and data fusion purposes. Application services implement exploitation algorithms to assess, enhance, and transform data from one or more sensors sequentially, in parallel and/or recursively to create new and enhanced data and metadata products.

Retrieval services provide standards based and purpose built interfaces for retrieving data captured and generated by the system using a common frame of reference to relate disparate data elements based on time and location. Standards based retrieval services provide the ability for many Commercial Off-The-Shelf (COTS) and open source GIS visualization tools to be used to visualize the data stored by the PSSA system. For higher performing and more sophisticated user interaction, custom applications can be developed that communicate with PSSA services directly on the PSSA message bus.

3.2 Persistent Sensor Storage Logical Architecture

3.2.1. Overview

From a logical perspective, the PSSA consists of four customizable software layers: Ingestion Services, Application Services, Data Services, and Dissemination Services. In addition, the PSSA includes a common Core Services layer which is responsible for providing the infrastructure required to support the integration of the components in the customizable software layers. The PSSA SDK provides access to the services provided by the Core Services layer. The diagram below provides a logical view of these software layers and examples of some of the components that might be present in these software layers.

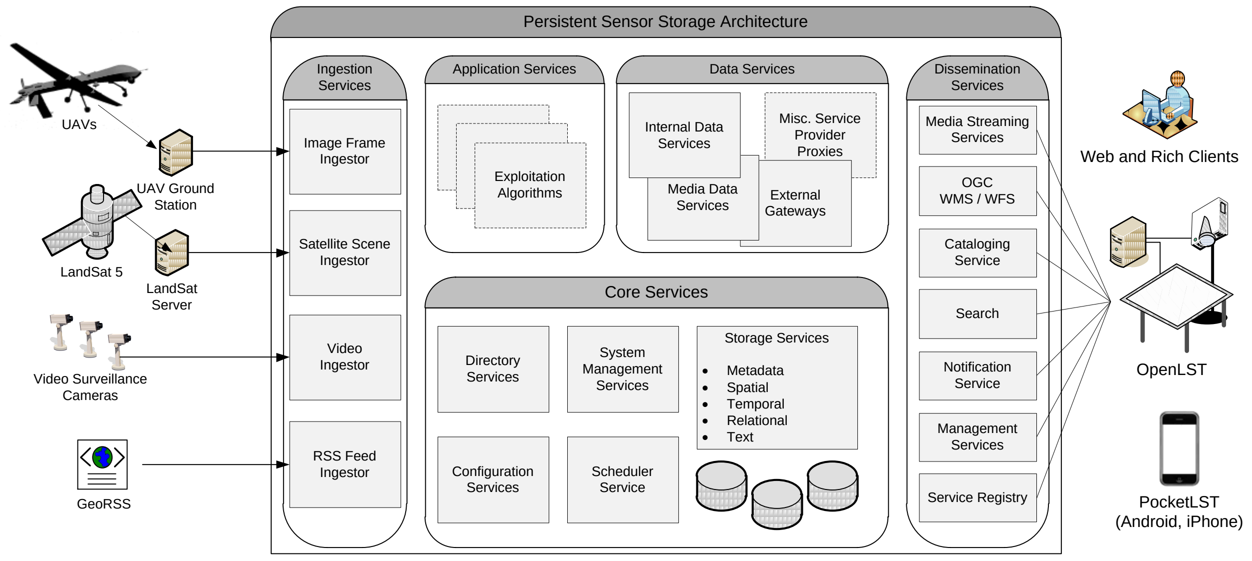


Figure 2: Logical View of the PSSA

Although the diagram, above, only shows four types of sensor sources, the PSSA is designed to support the storage and exploitation of data from any sensor source by "plugging" in new ingestion services. Similarly, the PSSA is designed to support the distribution of sensor data and related exploitation data to any visualization platform by "plugging" in standards based and custom built dissemination services.

The PSSA is designed to allow components in the customizable software layers to be added or removed from the system, dynamically, without interrupting the operation of other components of the system. This is accomplished through the use of a message bus approach. The PSSA message bus provides the following messaging modes:

- **Publish/Subscribe:** Source components publish messages on the PSSA message bus as "topics." Sink components subscribe to "topics" to receive messages in which they are interested. Filters can be applied by sink components to reduce the number of messages received by the component. Any number of sink components can subscribe to the same "topic." This is the typical communication mode for pluggable components as it allows them to be completely decoupled from one another.
- **Request/Response:** A sink component issues a data request to a source component and the source component sends its response back to the sink component. This process is typically how components communicate with core services that are guaranteed to be present and external gateways that are used to retrieve data from external systems.
- **Stream Request/Response:** A sink component requests streaming data from a source component. The source component sends the requested data stream to the sink component until the request is satisfied or the sink component tells the source component to stop streaming. This process is typically used by visualization components that want to display live or recorded streaming media.

By decoupling the components from each other using the message bus architecture, four important goals are achieved:

- Operators and developers can add and remove components from the system dynamically without impacting other components of the system (“plug and play”)
- Data is “broadcast” to several components at the same time allowing processing of the data to take place in parallel
- Processing is distributed across multiple compute nodes providing virtually unlimited horizontal scaling of the system
- Sensor data becomes available for visualization as soon as it is ingested by the system -- without waiting for it to be stored and indexed

The following diagram summarizes the key components and types of services that comprise the PSSA along with the typical flow of data into and out of the PSSA messaging system which we refer to as the PSS or PSSA Cloud (represented by the cloud in the center of the diagram, below).

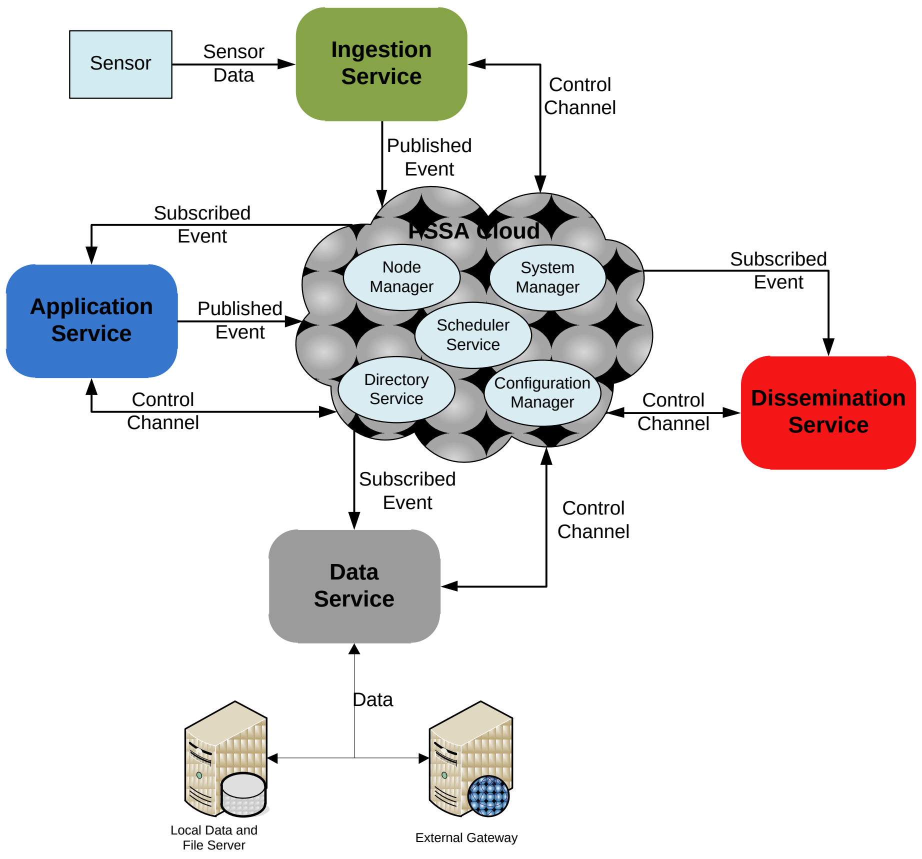


Figure 3: Key Components of the PSSA

The reason we refer to the PSSA messaging system as the “Cloud” is because it hides the internal implementation of the core services and integration mechanisms from the user developed components of the system. Developers of PSSA components need only know the public interfaces exposed by the PSSA SDK in order to participate in the event collaboration model of integration used by a PSSA-based system.

A brief description of each of the software layers and their functions is provided below:

- **Ingestion Services** – Receive or retrieve sensor data from external sensors and bring the data (sensor readings and associated metadata) into the PSS system. The main responsibilities of these components are to communicate with the external sensors or sensor systems and to make the data available to other components of the system, via the PSSA message bus. The PSSA reference implementation provides several different types of ingestion components that can be used as examples for creating custom ingestion components.
- **Application Services** – Exploit one or more sensor data stream or exploitation data stream and, optionally, external data sources to turn the sensed data into useful information (e.g. identify anomalies or other data of interest, analyze the quality of the data, improve the quality of the data, etc.). These components do not need to know how to communicate with the sensors, but they do need to understand the semantics of the data generated by the sensors in order to exploit it. Application service components implement algorithms to exploit data to which it is subscribed on the PSSA message bus. Results of the exploitation processing are published to themessage bus and are thus available to be stored, further exploited or displayed using a visualization tool. The PSSA reference implementation includes several examples of application service components that can be used as examples for creating new components using the PSSA SDK.
- **Data Services** – Store and retrieve data from the PSSA internal data stores and/or external systems for use by other components of the system. A data service component encapsulates the access to data by providing high-level message based interfaces to the data. This allows the consumer of the data to focus on how to use the data rather than on how to get the data. It is the responsibility of the Data service component to fulfill the data request of the other component making the data request. Data service components hide the details of communicating with internal and external databases and with other sensor storage systems from PSSA components that require the data to do their processing. The PSSA message interface to retrieve data from a data service component will typically be a request/response message. For storing data, data service components subscribe to the message topic for which they are responsible and perform whatever processing is required to store the data.
- **Dissemination Services** – Provide standard and custom built interfaces to provide access to data ingested by and/or stored within the PSSA system to users or external systems. A PSSA dissemination component can either push the data to a remote system or wait for the remote system to request the data. The PSSA reference implementation includes several examples of PSSA dissemination components use the PSSA message bus to subscribe to topics generated by other components of the system. Typically, the interfaces provided by these components utilize a synchronous request/response message mode to send data from the PSSA based system to other systems. However, some dissemination components support the retrieval of live data as well as previously stored

data using the stream request/response message mode. In addition, a dissemination component can be developed to provide event notifications or alerts asynchronously when certain events occur.

- **Core Services** – Provide the infrastructure required to connect the other components of the system together and to store the data ingested by the system. This infrastructure includes a directory service that allows sensor processing components to be dynamically added to and removed from the system; a scheduling service to ensure optimal use of the computing resources by load balancing and sequencing tasks; a message bus for moving data between components of the system; a configuration and management service to allow processing components and compute nodes to be added and removed from the system; and a storage service to provide storage, indexing, and association of data required by the system.

The following sections describe these components in more detail.

3.2.2. Pluggable Components

One of the key characteristics of the PSSA is the support of dynamic "pluggable" components. What this means is that new components developed using the PSSA SDK can be added to the system simply by "dropping" the component assemblies into the appropriate PSS component directory. When a new component is detected, the assembly is loaded and registered within the system. This process includes the registration of message bus topics associated with that component and, optionally, the creation of one or more database tables in which data associated with the new component will be stored.

There are four core types of components that can be "plugged" into the architecture: ingestion services, application services, data services and dissemination services. When a new sensor is added to the system, the installation package may include all four types of pluggable components in order to provide a complete processing and distribution stream for the sensor data. At a minimum, a new sensor will provide an ingestion component. Visualization tools can use the PSSA message bus or existing dissemination services to display the data generated by the sensor. As new exploitation algorithms are developed for the sensor, they can be plugged into the system. If specialized dissemination services are required to take full advantage of the sensor and its related exploitation algorithms, these can be plugged into the system as well. It is important to note that all of this activity and these operations can take place without shutting down the system or stopping the flow and exploitation of data from other sensors in the system. This capability is essential for a 24x7 production environment.

3.2.3. Ingestion Services

Ingestion services are software modules that receive sensor data from remote sensors and transform that data into one or more "topics" on the PSS message bus. The builder of an ingestion service is responsible for writing the code that will connect the sensor to the ingestion component endpoint. There are two modes of connecting to the sensor: synchronous/pull mode and asynchronous/push mode. In the synchronous mode, the ingestion service establishes a connection to the sensor and periodically polls the sensor for new data. This activity is referred to as "pulling" data from the sensor. In the asynchronous mode, the ingestion service either registers itself with the remote sensor or the remote sensor is configured to connect to a well- known endpoint provided by the ingestion service. After the connection is established, the ingestion component waits for the remote sensor to send data. This is referred to as the remote sensor “pushing” data to the ingestion service. It is expected that there will be a need for ingestion services to have some type of configuration data to specify the information needed to connect to the remote sensor or to configure the endpoint by which remote sensors will connect to them.

The designer of the ingestion service is responsible for defining the format of this configuration data and the means by which the data will be retrieved (e.g. data file, Uniform Resource Locator (URL), database connection, message bus, etc.). Another responsibility of the designer of an ingestion service is to define the format of the “topic” or “topics” that the ingestion service will generate when data is received from the remote sensor.

After a new sensor ingestion service is added to the system, one or more physical sensors can be configured which will use that ingestion service to capture data generated by the sensor. When a physical sensor is added, the ingestion service that will process data from that sensor is specified. Subsequently, when the physical sensor is activated, an instance of the ingestion service is launched to begin capturing data from that sensor.

NOTE*: Ingestion services can be designed to handle multiple physical sensors simultaneously or a single sensor per instance. Which approach is taken depends largely on the nature of the sensor and the frequency and volume of data expected.*

A key part of the design of the ingestion service is the definition of the message bus “topics” used by the component and the mapping of this data to a database schema. The PSS Core Services includes a generic storage service that will map the data elements defined in the message bus topic to a corresponding database table. This service uses the topic name and data elements to map the data to the appropriate database table and columns. As a result, specific naming conventions and data elements are required to be part of the message “topic.” These requirements are outlined later in this document. If specialized data storage and retrieval processing is required, the ingestion service’s installation package must include the software modules to perform this processing.

In order to provide a common context and frame of reference for the data stored within the PSS system, all ingestion services are required to include location and time metadata within their message bus “topics.” This metadata is used to synchronize sensor data from disparate sensors for exploitation and visualization purposes. More information on the issues associated with spatial and temporal synchronization can be found in Appendix D – Synchronization Issues.

For ingestion services that generate streaming video feeds, the PSS Core Services includes a Media Storage Service that supports the storage and retrieval of video streams. This service “chunks” the video stream into files of a predetermined playback length (e.g. five minutes) for storage purposes while at the same time providing a logical view of a single contiguous media stream. Gaps in the media stream are provided to visualization tools as metadata that can be used to indicate time periods for which no data was collected. The Media Storage Service can also provide seamless transition between live playback and previously recorded video.

3.2.4. Application Services

Application services are software modules that apply algorithms to analyze and/or combine data from one or more of the following sources to create new and useful information:

- Live sensor feeds
- Data from other application service components
- Previously recorded sensor data
- Current and historical data from other (external) systems

For example, an application service can be used to compare a real-time image with reference imagery to attempt to geo-locate the real-time image or to compare sequential video frames looking for moving targets or to analyze one or more video frames to establish quality metrics such as noise or blur. Another example is a moisture flux algorithm that uses data from satellite imagery, ground based sensors, and an external weather service to calculate moisture flux values for the region covered by the satellite imagery. The PSS architecture does not natively define any application services since the implementation of these components is necessarily application specific. However, the PSSA does provide support in the SDK for the development of such components as well as several reference implementations that can be used to get started.

Application services typically fall into one or more of the following categories:

- **Exploit** – the service creates new data as a result of running algorithms against the input data (e.g. detecting an object within an image). Typically, a new topic is created and published to the message bus.
- **Transform** – the service changes the format or data type of existing data to normalize it or make it more useful to other application services (e.g. warping an image to a new coordinate system). In this case, the same topic is typically reused; however, the data and metadata will be changed to reflect the transformation of the data.
- **Enrich** – the service increases the value of the data by improving/enhancing the data or the metadata associated with the data (e.g. measuring the quality of the data and adding quality information to the metadata, improving the accuracy of location metadata by applying geo-registration/photogrammetric techniques to the data, improving color of image data using white-balance or radiometric corrections). Similar to transformation, the same topic will typically be reused; however, as a result of this processing, the metadata and perhaps even the sensor data will be more accurate and complete.
- **Filter** – the service subscribes to a single input event and generates a single output event.
- **Aggregate** – the service subscribes to multiple input events and combines the data from each of the events to enrich, transform, or exploit the data in some meaningful way. This may result in one or more output events.
- **Split** – the service subscribes to a single input event but generates multiple output events in the process of enriching, transforming, or exploiting the data.

The designer of an application service component must know the format of the message bus "topics" for the primary data source (sensor or other application service) and, if required, the request/response protocol used to retrieve data from other components of the system. A message from the primary data source will be the event that triggers execution of the application service exploitation algorithm. If previously recorded data from the PSS storage subsystem or an external gateway is required, then appropriate request/response messaging must be designed to retrieve the data. The designer of the application services may need to build application specific storage components to retrieve this data from the appropriate source. If these components are required by the application service, they should be included as part of the application service installation process.

3.2.5. Storage Services

***Data Storage Service*:** The storage service provides a flexible mechanism for storing and retrieving data generated by the ingestion and application service components of the system. The generic storage mechanism monitors the PSSA message bus for topics and source ids that are identified as persistent. The topic name is used to identify the table in which the contents of the message are stored. The contents of the message are then examined matching attribute names in the content portion of the message with column names in the target table and the corresponding values from the content portion of the message are extracted to insert a new row in the target table.

For many topics, this capability is sufficient for storing the data. However, for more complex topics, it may be necessary to provide specialized components to store the data. These components are dynamically loaded as needed to process persistent topics that cannot be handled by the generic storage component.

Similarly, a generic data retrieval component is provided as part of the storage service to retrieve previously stored data based on time range, location, and/or source sensor. For data generated by application services, the source is the sensor that generated the data exploited by the application service. The generic retrieval component operates on a single topic and returns data in the same format as the original message. For some complex topics, it may be necessary to provide specialized components to retrieve the data required to format the response.

***Media Storage Service*:** For streaming media data, a special storage service is provided called the "Media Storage Service." This component creates a logical abstraction that isolates the visualization platform from any knowledge of how the media stream is physically stored within the system. From the perspective of the visualization platform, the stored media stream looks like a continuous stream of data -- despite the fact that it is actually broken up into a series of smaller files.

The Media Storage Service also has the ability to "normalize" the incoming video stream to a standard media encoding and compression (currently we are using Moving Pictures Expert Group (MPEG)-4) before placing the data on the PSSA message bus. The PSSA SDK provides MPEG-4 encoding and decoding capabilities so that application services can exploit the media stream in a standardized manner regardless of the encoding format used by the source sensor, if any. In addition, use of the MPEG-4 standard significantly reduces the amount of bandwidth utilized on the message bus and the physical storage space required to store the stream on disk.

***External Gateway*:** External gateways are typically developed to support the needs of a specific application service, however, in some cases these gateways may be generic enough to be used by many different applications (e.g. weather data, census data, certain environmental data). In order to leverage these external gateways across multiple applications, we decided to decouple them from the application services code and allow them to standalone. Application services request data from the external gateways through a well-defined request/response protocol. This protocol is made public so that other application services and, in some cases, dissemination components may issue similar requests for the data.

The designer of an external gateway should be responsible for defining the request/response protocol that will be used to retrieve the data and for implementing the software that connects to the external system to retrieve the data. The protocol should support as broad a set of data requests as possible in order to maximize the usefulness of the gateway across applications.

3.2.6. Dissemination Components

Dissemination components are software modules designed to retrieve information from the PSS Storage system or in some cases to push real-time data feeds to third party visualization tools. The PSS SDK supports the development of “integrated” visualization tools such as the PSS dashboard application or the Open Layered Sensing Testbed (OpenLST). These tools subscribe to “topics” on the PSS message bus to provide tightly integrated, high performance visualization of the data generated by the ingestion and application service components.

However, in many cases, it is desirable to use COTS or unmodified open source tools to view the sensor data. For that purpose, a number of dissemination components are included in the PSS reference implementation including a RTSP media server for broadcasting streaming video to RTSP clients and GeoServer to support WFS and WMS requests for spatial data. Other dissemination components can be custom built as needed to support the needs of COTS/Government Off-The-Shelf (GOTS) applications or any other visualization application that can’t interface directly to the PSS message bus.

3.2.7. Core Services

The PSSA Core Services provide the infrastructure to allow PSSA components to connect with and communicate with one another. They also provide management services that allow users to monitor the operation and performance of the system.

Dashboard and System Management Services: The PSSA Dashboard and System Management Services provide software and a user interface to display and control the components of a PSSA-based system. The dashboard provides the following capabilities:

- View the status of each PSSA hardware node configured within the PSSA system
- View the status of each PSSA services installed in the system including the node(s) upon which they are installed
- Start, stop, and pause PSSA services. When a service is started, the system scheduler determines the best node upon which to start the component. If desired, the user can override this selection and choose a different node upon to start the service.
- Install PSSA services “on-the-fly” on any compatible PSSA hardware node
- Remove PSSA services “on-the-fly” from any compatible PSSA hardware node

- Filter the events received by a PSSA service based on the source component from which the event is sent. This allows the system scheduler to balance processing load, as needed. It also allows the user to control the type of processing that occurs on a given data stream (e.g. the user may desire that some video streams be exploited for motion detection while others are not).
- Monitor (for diagnostic purposes) the messages generated for any topic(s) published by components of the system.

The System Management Services described above will be built into the dashboard but will also be provided as interfaces within the PSSA SDK to allow software developers to integrate these capabilities into their own visualization or system management tools. The System Management Services rely on the PSSA Directory service to keep track of all of the hardware nodes and software services installed in the system, their capabilities and operational status.

3.2.8. Directory Service

The directory service is a key component of the core services. This service is used to track which sensors and application services are installed and whether or not they are active (collecting/processing data). It also lists the message bus "topics" generated and consumed by each of the sensors/application services. This allows application services and visualization tools to dynamically subscribe to any source for which they know how to process the topic. For example, when a user initiates an application service that exploits video streams to track vehicles, the PSSA dashboard queries the directory service to find all of the sensors and/or application services that generate video stream "topics."

The user can then choose the video streams on which to apply the vehicle tracking exploitation algorithm. Similarly, a user could select a sensor and be presented a list of all of the application services that can be applied to the topics generated by that sensor. Since the ingestion components and application services are decoupled from one another using the PSS message bus, any application service that can process a specific message bus topic, can process data generated by any sensor or any other application service.

As sensors and application services are added and removed from the system or taken on or off line, visualization tools can also use the directory service interface to dynamically update the list of sensors and application services available to the user. The PSSA dashboard is one such visualization tool.

The directory service component also plays an important role in monitoring the health of the system. After starting up, each PSSA hardware node generates a Node Heartbeat event at regular intervals (nominally every 30 seconds). Each heartbeat event provides information about the "health" of the node including the status of each component running on the node and the Central Processing Unit (CPU)/Memory utilization of the node. The directory service subscribes to these events and updates an internal state table for each of the nodes and software components. If a node fails to send a heartbeat event within a certain period of time that node and all of the software components running on the node are designated as "dead" and an event message is generated to notify components running on other nodes to close their connections with components running on the "dead" node.

3.2.9. Scheduler Service

Whenever a PSSA component needs to be started, the Scheduler Service is invoked to determine which node on the network would be best suited to run the component. This is accomplished by evaluating the relative loading — Central Processing Unit/Graphics Processing Unit (CPU/GPU) utilization, memory, etc. — of each of the nodes on which the component is installed and the network capacity between the component and the other component(s) of the system from which the component will be receiving events. The goal of the scheduler service is to optimize network performance while not overloading the computing/memory capacity of a given node. Although the scheduler service can be expanded with a very sophisticated set of rules, initially, the following set of rules will be used to provide simplistic load balancing:

- **Ingestion Services** – The least heavily loaded node on which the service is installed.
- **Application and Dissemination Services** – The node that the service from which it is receiving events is running on, unless one of the following is true:
  - ➢ The service is not installed on that node
  - ➢ The CPU or memory utilization is greater than 80% of the capacity of the system
  - ➢ If either of the above is true, then the application service will be started on the least heavily loaded node on which the service is installed.
- **Storage Services** – The internal data storage service will run on the same node as the underlying database that stores and indexes the metadata. Streaming media data storage service(s) will be started on the least loaded node connected by high-speed networking (Infiniband®, 10gigE, gigE, etc.) to the high-performance file system. External Gateway data services will be started on the least loaded node upon which the gateway service is installed. Once the node on which a component will be started is determined, the scheduler service sends an event to the Node Management service to start the component.

3.2.10. Node Management Service

The PSSA Node Management Service runs on each individual hardware node of the PSSA system. It is responsible for the following activities associated with the hardware node:

- Starting/Stopping/Pausing the execution of services running on the node.
- Monitoring the status of PSSA services running on the node by subscribing to the Heartbeat messages generated by the service components.
- Monitoring the processing activity on the node (CPU/memory utilization)
- Periodically generating a Node Heartbeat message that reports the status of the node

The Node Heartbeat message consolidates the status information from all of the services running on the node as well as the processing activity of the node. This means that the dashboard and other system management services only need to subscribe to the Node Heartbeat message to get a complete picture of the operational status of the system.

3.2.11. Health Monitoring

Each component within a PSSA-based system periodically generates a heartbeat message. This message communicates the "health" of the PSSA component to the Node Management Service. The heartbeat message is built into the PSSA SDK so that the developers of the component do not have to worry about how or when it is generated. In most cases, they won't even have to know it is there.

Each heartbeat message contains the current processing state of the component (e.g. initializing, waiting for an event, processing an event, paused, shutting down), the results of the last event processed (e.g. Good , Warning, Error), tallies for the number of messages received/sent and the number of warnings/errors generated, and the average processing time per event.

Any component can subscribe to the heartbeat messages to determine whether or not a component is still available. However, it is the responsibility of the local node manager to generate an event when a component is determined to be "dead." This event message should be subscribed to by any component listening for events from the "dead" component so that they can close their connections.

3.2.12. Configuration Management Service

Like the Node Management Service, the Configuration Management Service runs on each hardware node of the PSSA system. The responsibilities of the Configuration Management Service include:

- Installing PSSA services on the node
- Removing PSSA services from the node
- Collecting/storing information about the hardware configuration and software configuration of the node. Examples of hardware/software configuration information that might be provided:
  - Number of network cards and corresponding network type (gigE, Infiniband, etc.) and address
  - # of CPU/GPU cores
  - Amount of physical memory
  - Operating System and version (e.g. Windows 2008 R2, Ubuntu Linux 10.10, etc.)
  - Software runtime dependencies installed (e.g. Matrix Laboratory (MATLAB), .NET, C++, etc.)

This information is used to determine whether or not a particular software service can be installed on the node when that service is added to the PSSA-based system.

To install PSSA services, the Configuration Management Service subscribes to events generated by the System Management Service indicating that a service is available to be installed. The event includes a URI specifying the location of the service installation package and hardware/ software requirements of the service. If the node is capable of running the service, the Configuration Management service will use the URI to copy the service installation package to the local node and initiate the installation process. If the installation process is successful, the Configuration Management service will notify the directory service that the software component is available on the node. Each service installation package is required to provide a removal script that can be executed if the service needs to be removed.

3.2.13. Event Logging Service

It is desirable for a number of reasons to be able to record and playback all of the events generated by the system including interactions with external systems. This is the responsibility of the Event Logging Service. This service runs on each node of the PSSA system and records all of the events that are published by components running on that node in the PSSA database. The events are stored with the common PSSA header metadata broken out into individual columns in the database so that they can be queried against.

Topic specific metadata and sensor payload data, if present, is stored as a blob in the database (or possibly on the file system, if necessary). Using this data it will be possible to reconstruct a complete message flow through the system for testing, diagnostic, and auditing purposes. The Event Logging Service is also responsible for recording interactions with external systems through external gateway data services. The PSSA SDK will provide Application Programming Interfaces (APIs) to allow the Event Logging Service to hook into the communication channel between the gateway data service and the external system so that this data can be captured.

3.3 Component Metadata

In order to describe a component to the system so that it can be registered with the directory services, the following information must be defined for each component:

- Component Name (freeform text)
- Component Description (freeform text)
- Component Category (one of: Ingestion, Exploitation, Storage, Dissemination)
- Event Topics published by the component
- Event Topics subscribed to by the component
- Special processing needs of the component
    - ➢ CPU and/or other hardware required, such as GPU, Field-Programmable Gate Array (FPGA), Application Specific Integrated Circuits (ASIC), and so on.
    - ➢ Operating System requirements (Windows 2008, Linux, Mac OS/X, etc.)
    - ➢ Runtimes and/or shared libraries used (.NET, Java, MATLAB®, C/C++)

This information is used by the system management and node management services to determine upon which processing nodes in a distributed processing system the software component can be installed.

3.4 Event Topic Definition

PSSA event topics are defined by the component developer(s) to meet the processing needs of the component. However, there are some common metadata elements included in the context section of all event topics that is required by the PSSA-based system to index, correlate and query data across disparate sensors in a common manner. This common metadata contained in the Event context header includes the following data elements:

- **Producer**. The unique identifier of the component generating the event. This data is automatically supplied in the context header by the PSSA SDK.

- **Reference Sensor ID**. The unique identifier of the sensor from which the sensor reading or derived information originated.

  ***NOTE**: This could include multiple sensors if the payload of the event contains information derived from multiple sensors.*

- **Presentation Time**.[*] The Universal Time Coordinated (UTC) time index of the data contained in the event payload. This is represented as a TimePoint structure which includes a point in time (in UTC) and the precision associated with the time (+/-) based on where the time originated. This is the time by which all of the data associated with the event will be indexed. The method of determining this time may be different for every ingestion component. In some cases the sensor may have a highly accurate time included with the metadata it generates for each reading. In other cases, the sensor may have no time data at all and the ingestion component must provide an educated guess as to the time the data was acquired based on the current system time.
- **Acquisition Time**.[*] The time that the data contained in the event was received or generated by the system. Ingestion components capture the time at which the sensor data was read/detected by the system (e.g. for a polling component, this is the time at which the request returned with the data, for an asynchronous component, this is the time at which the data was received, for a file detection, this is the time at which the new file was detected). Exploitation components that generate new data use the latest acquisition time contained within the events that were used to generate the new data (e.g. some exploitation events may require multiple frames of data to identify an object or they may use data from one or more different types of events). All other exploitation components leave the acquisition time unchanged.
- **Event Time**.[*] This is the time that the event was published by the component (automatically inserted by the PSSA SDK).
- **Processing Duration**. This is the amount of time it took for the component to process the data to generate the event (automatically inserted by the PSSA SDK).
- **Sensor Location**. This represents the (point) location at which the sensor was located when the sensor reading was captured. Commonly this will be provided as metadata with the sensor reading. If the sensor is attached to a non-moving platform, the location data may be part of the configuration data for the sensor. In either case, the event message must contain location information for indexing purposes in the database.

---

[*] Time is standardized within PSSA for display and for transport between components to be UTC time using the standard ISO 8601 time representation with up to 6 decimal places of precision (e.g. 2010-10-28T20:08:25. 182302Z). The PSSA hardware architecture specifies that a high-precision NTP server (or PTP server) be utilized within the system to synchronize the system time on all of the computational nodes of the system. This Network Time Protocol (NTP) server and the PSSA infrastructure software will ensure that any times requested from PSSA nodes will be correct to within 10 milliseconds (1 µs, if a PTP server is used) of the actual UTC time at the moment the time is requested. See Appendix D for a broader discussion of the issues associated with time synchronization and how it is handled by PSSA.

- **Sensor Location Accuracy (Optional)**. If known, this is included as part of the common event data.

  NOTE: *Downstream exploitation components may be able to improve the location accuracy or, if the accuracy is unknown, it may be possible to estimate the accuracy of the sensor location metadata using historical data or data from other sources. If this is the case, a new event will be published with the corrected location metadata and/or accuracy information.*

- **Footprint (Optional)**. This represents the coverage area of the sensor. For some types of sensors this information may not be known at ingestion time and may be calculated by downstream exploitation algorithms. In other cases, the footprint might be calculated to some degree of accuracy by the sensor or sensor ground-station and included in the metadata received by the ingestion component.
- **Footprint Accuracy (Optional)**. If known, this is included as part of the common event data.

  NOTE*: Downstream exploitation components may be able to improve the footprint accuracy or, if the accuracy is unknown, it may be possible to estimate the accuracy of the sensor footprint metadata using historical data or data from other sources. If this is the case, a new event will be published with the corrected footprint metadata and/or accuracy information.*

For the reference implementation of the PSSA system, we defined the event message to be a multi-part mime that includes a header and one or more mime data segments to transmit the sensor data. The header is comprised of a context section that contains the common data described above and a content section that contains metadata associated with the subsequent mime data segments. This works very well within the context of our messaging system.

4.0 RESULTS AND DISCUSSION

This section describes the physical architecture developed for Phase I of the Persistent Surveillance Data Processing, Storage and Retrieval project and the reference implementation (prototype) we built to test many of the concepts present in the architecture. The previous section provided an overview of the functional capabilities we envisioned for the PSSA. In contrast, this section provides a detailed description of how we have implemented or plan to implement many of those capabilities. In addition we present discussion around various usage scenarios in which the PSSA might be applied and how a PSSA-based system might interoperate or integrate with other sensor-based systems and processing architectures.

Additional detailed descriptions of specific areas of our work and research are also provided in the following appendices:

- Appendix A – PSSA Dashboard Reference Implementation
- Appendix B – Tutorial for Building PSSA Components (SDK Reference Implementation)
- Appendix C – Java Bindings for PSS SDK
- Appendix D – Synchronization Issues

4.1 Reference Implementation – SDK, Components and Demonstration

4.1.1. Overview

As mentioned in the earlier sections, the PSSA consists of 4 customizable software layers: Ingestion, Application Services, External Gateways, and Dissemination. In addition, the PSSA includes a standardized Core Services layer which is responsible for providing the infrastructure required to support the integration of the components in the customizable software layers. From a software developer's perspective, to build a customized PSSA component in former layers, one needs to have access to a specialized SDK that can encapsulate the low-level communication and data-specific implementation details into a well-structured PSSA-specific API call.

In this vein, this section starts by providing the description of the developed PSSA SDK, followed by which the details to employ the SDK-specific functions/procedures to build few customizable PSSA components/service will be provided. Further, a description of already built reference PSSA components will be presented with inkling to a reproducible demonstration of PSSA capabilities. Finally, a procedure is provided to build a working demonstration of the reference PSSA implementation, which would include in it the appropriate recommendations to configure and employ the developed PSSA components and services.

4.1.2. PSSA SDK

As part of the effort to design and develop the PSSA reference implementation to ingest, exploit, store and visualize disparate real-time sensor data, an SDK has been developed in C++ language, using a light-weight messaging middleware library called ZeroMQ. This developed PSS SDK aptly provides the required API interfaces for any consumer of the PSSA architecture, to build low-level PSSA modules like Publisher, Subscriber and Service Provider. These low-level PSSA modules can be grouped (according to the custom business logic) to build high level customized components like Sensor Data Ingestor, Exploitation Service, Data Storage Service and any helper services like eXtensible Messaging & Presence Protocol (XMPP) message bridge as shown in Figure 4. Further, the PSSA SDK has been design to effectively abstract the architecture and communication (along with few media processing routines) complexity from the consumer and aid him/her in building the high-level business logic components that can effectively leverage the PSSA framework.

The PSSA SDK was originally developed in C++, with pertinent classes and structures, later wrapped with C "DLL" procedures to make it cross-language compatible. Further, the JAVA and C# bindings/wrappers are also available for the present version of the PSSA SDK. The PSSA SDK has proven to be beneficial to develop PSSA components in Java, MATLAB and C# languages for various capability demonstrations (discussed in this report). This sub-section will provide the functional description of the APIs that are available as part of the C-DLL interfaces which reasonably extrapolates into the other language bindings.

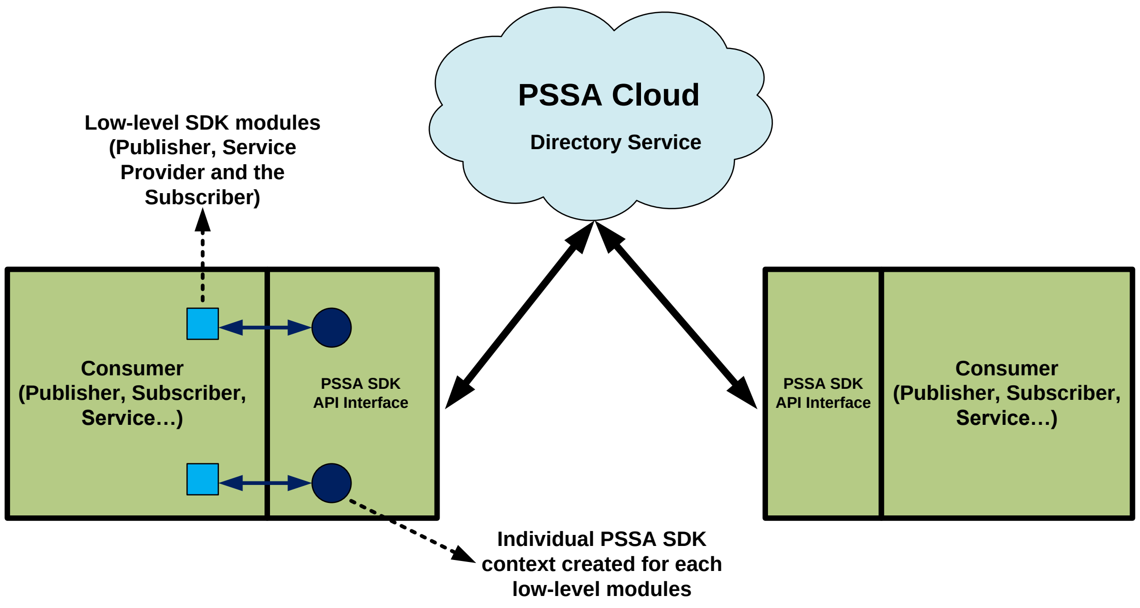

PSSA Cloud
Directory Service
Low-level SDK modules (Publisher, Service Provider and the Subscriber)
Consumer (Publisher, Subscriber, Service…)
PSSA SDK API Interface
PSSA SDK API Interface
Consumer (Publisher, Subscriber, Service…)
Individual PSSA SDK context created for each low-level modules

Figure 4: PSSA SDK Interface

Figure 4 is a pictorial representation of employing PSSA SDK API interfaces to build customized (Business Logic) components. The developed PSSA SDK provides the required API interfaces for any consumer of the PSSA Cloud architecture, to build low-level PSSA components like Publisher, Subscriber and Service Provider. These low-level components can be grouped (for building Exploitation algorithms that would require Subscriber and Publisher components in them) appropriately to build customized components like Video Ingestor, Media Storage Service and other Exploitation Algorithm Services. It can be seen in Figure 4 that each of the low-level SDK modules have their own independent contexts created in the SDK layer to effectively manage resource allocation and release as explained in the details in the text of this section.

4.1.3. Setup and Teardown API Calls

The "Directory Service," as shown in Figure 4 is one of the main and required core service that would define a unique PSSA cloud instance, i.e. every PSSA cloud can have only one directory service at a pre-designated address. Directory Service developed for the present version of the PSSA reference implementation is responsible for keeping track of all the publishers and service providers present in its PSSA cloud instance and provide the new or existing subscribers with this tracked information at runtime.

The PSSA SDK provides below API calls to initialize and release the resources like memory, threads and network ports, which are encapsulated in form of creating independent contexts for each low-level SDK modules. These established contexts would provide the low-level modules with information and interface required by the underlying message passing and event registration mechanism:

```
PSS_RESULT_CODE PSS_Initialize(  const char* directory_endpoint,
                                 const int starting_port_number,
                                 const int ending_port_number )
```

The PSS_Initialize takes as its first input parameter as a string, which specifies the directory service's protocol address, like *"tcp://XXX.XX.XX.XX:Port",* in the endpoint address format adopted from the 0MQ standards. The next two input parameters specify the available contiguous port range (start and end) that is available on a given machine for the PSSA component to use at the runtime . Only successful initialization the PSS_RESULT_CODE value "0" is returned. This call internally does default (for example one context per application) allocations of the resources primarily the number thread resources for underlying 0MQ library routines. But, if the developer chooses to specify the number of the application threads and the I/O threads, with some pre-conceived knowledge about the PSSA component in terms of their context, than the below call can support it:

```
PSS_RESULT_CODE PSS_InitializeEx( const char* directory_endpoint,
                                  const int starting_port_number,
                                  const int ending_port_number,
                                  const int number_application_threads,
                                  const int number_io_threads )
```

Where the *number_application_threads* specifies the number of application threads that will utilize SDK communication services and *number_io_threads* specifies the number of number of I/O threads SDK will use internally to manage communication.

The next step in the initialization process is to acquire a context, mentioned earlier, to the SDK's communication threads using the below call:

```
PSS_RESULT_CODE PSS_InitializeContext( const char* directory_endpoint,
                                       const int starting_port_number,
                                       const int ending_port_number,
                                       PSS_CONTEXT_HANDLE* contextex  )
```

Where the **PSS_CONTEXT_HANDLE** type pointer (*contextex* above) provides the encapsulated access to the context (created and) returned by the underlying initialization routine from the SDK.

The individual context creation is obligatory and will be used to create the low- level publisher, subscriber and the service provider modules in a PSSA component in the context. Accordingly the context release call is also provided to for these low-level modules, which takes as it input a context pointer to be released as shown below:

```
PSS_RESULT_CODE PSS_ReleaseContext( const PSS_CONTEXT_HANDLE contextex )
```

Further, the overall cleanup for the SDK can be invoked with below call which can release the all the allocated resources employed during the lifetime of the PSSA component, as shown below: SDK_API

```
PSS_RESULT_CODE PSS_Cleanup( )
```

4.1.4. PSSA Message Format and Utility API Calls

The PSSA currently uses an event message structure that can hold multiple data parts in it, which can be visualized as a multipart mime structure shown next in Figure 5.

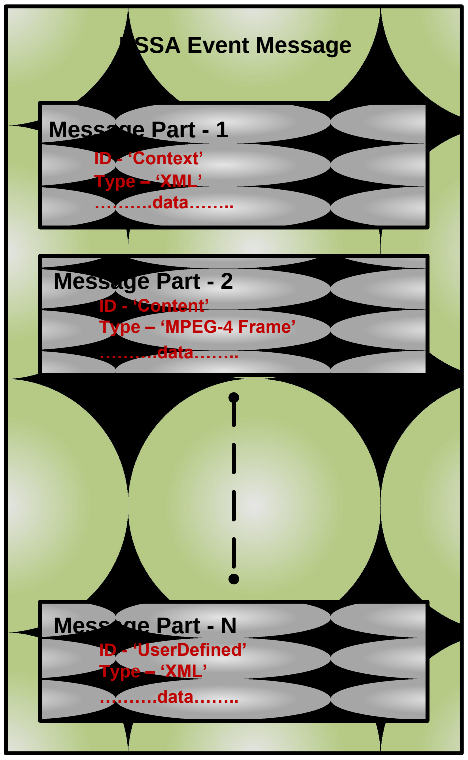


Figure 5: PSSA Event Message Structure

Figure 5 is a pictorial representation of PSSA event message structure. It can be noticed that multiple and independent message parts can be embedded into the event message with appropriate attributes like ID, Type and Index. The primary rationale behind designing the current event structure is to support the mixed sensors platforms, with multiple and varied data generating sensor sources.

Each of these data parts can be individually attached and manipulated with the SDK's message routines. Further, each data part can have independent attributes like content type, ID and index in a given event message. The primary rationale behind designing the current event structure is to support the mixed sensors platforms, with multiple and varied data generating sources on it, like for example a Global Positioning System (GPS) and Inertial Measurement Unit (IMU)-enabled mobile camera would generate a single PSSA event message, with three data parts in it holding the GPS data, IMU data and the video frame data in it.

Further, it is conceived that every event message would at least have two message parts in it, one for the context of the event and another one holding at least one content of the event. The context of the event message would hold in it the metadata like timestamp, location and generation source for the data parts in the content area of the event.

A sample message would be described later in the section. Below are the message related SDK API calls that would aid in the creation, manipulation, retrieval and destruction the PSSA event message.

```
PSS_RESULT_CODE PSS_CreateMessage( PSS_MESSAGE_HANDLE* message )
PSS_RESULT_CODE PSS_ReleaseMessage( const PSS_MESSAGE_HANDLE message )
```

Where the **PSS_MESSAGE_HANDLE** type pointer (*message* above) provides the encapsulated access to the PSSA event message (created and) returned by the underlying message creation routine from the SDK. The PSSA event message can be released by the above *PSS_ReleaseMessage* call. Further, the message parts can be added to the created message handle using the below call:

```
PSS_RESULT_CODE PSS_AddMessagePart( const PSS_MESSAGE_HANDLE message,
                                    const char* content_id,
                                    const char* content_type,
                              const void* content,
                                    const unsigned long length,
                                    const bool firstPart )
```

Where the *message* is the created message handle, *content_id* is the user specified identification for the part, *content_type* specifies the well-defined format (at least defined in the PSSA system) of the data of the message part and the *content* is the pointer to the byte array of the raw data to be embedded in to the message part. The *length* is the length of the content data in bytes and the *firstpart* flag specifies if the part being added is a first part of the event message. The content_id, content_type and firstpart are encapsulated into a **PSS_MESSAGE_PART_INFO** structure as shown below to the developer thru the PSSA SDK:

```
typedef struct
{
  const char* content_id;
  const char* content_type;
  bool firstPart;

} PSS_MESSAGE_PART_INFO
```

Additionally, SDK provides below API calls to retrieve the information from the existing PSSA event message:

```
PSS_RESULT_CODE PSS_GetMessagePartCount ( const PSS_MESSAGE_HANDLE message,
                                           int* part_count )
```

The above call returns the number of parts in *part_count*, which are present in a given PSSA event specified by the *message* handle in the context. It is also possible to retrieve information of all the parts present in the given message along with the part's count, as shown below:

```
PSS_RESULT_CODE PSS_GetMessagePartInfo( const PSS_MESSAGE_HANDLE message,
                                        PSS_MESSAGE_PART_INFO** parts,
                                        unsigned short* part_count )
```

Further, individual part content from the event message can be retrieved by using below two calls, first by a known index in the event and the second by a known content_id :

```
PSS_RESULT_CODE PSS_GetMessagePartByIndex ( const PSS_MESSAGE_HANDLE message,
                                             const unsigned short index,
                                             void** content,
                                             unsigned long* content_length )

PSS_RESULT_CODE PSS_GetMessagePartByContentId
                                       ( const PSS_MESSAGE_HANDLE message,
                                         const char* content_id,
                                         void** content,
                                         unsigned long* content_length )
```

Upon successful call completion, the *content* points to the raw data array and the *content_length* specifies the length of the retrieved raw data array in bytes. The current SDK version provides a procedure to specify filters that can be applied on any given message part. There exists a reference implementation for two flavors of the MPEG-4 decode filters — built using Fast-Forward MPEG (FFMPEG) library routines — which have been encapsulated in to a **PSS_MESSAGE_FILTER_TYPE** enum as shown below:

```
typedef enum
{
  PSS_MPEG4_DECODE = 0,
  PSS_MPEG4_DECODE_FLIPPED = 1
} PSS_MESSAGE_FILTER_TYPE
```

The message filter instance encapsulated in **PSS_MESSAGE_FILTER** handle can be created and released by below API calls:

```
PSS_RESULT_CODE PSS_CreateMessageFilter
                              ( PSS_MESSAGE_FILTER_TYPE messagefilterType,
                                PSS_MESSAGE_FILTER* messageFilter )


PSS_RESULT_CODE PSS_ReleaseMessageFilter
                              ( const PSS_MESSAGE_FILTER messageFilter )
```

Further, individual part content from the event message can be retrieved, with a specified filter applied to it, by using below two calls (the terms in the below call hold the same meaning mentioned above in the original API calls):

```
PSS_RESULT_CODE PSS_GetFilteredMessagePartByIndex
                                  ( const PSS_MESSAGE_HANDLE message,
                                    const unsigned short index,
                                    const PSS_MESSAGE_FILTER messagefilter,
                                    void** content,
                                    unsigned long* content_length )

PSS_RESULT_CODE PSS_GetFilteredMessagePartByContentId
                                  ( const PSS_MESSAGE_HANDLE message,
                                    const char* content_id,
                                    const PSS_MESSAGE_FILTER messagefilter,
                                    void** content,
                                    unsigned long* content_length )
```

PSSA Publisher & Service Provider API Calls: As mentioned earlier, the current version of the PSSA SDK provides necessary APIs to build the low-level publishers and the service providers in a PSSA component. This sub-section provides those concerned API details, with insights into the underlying publisher/service provider registration process with the directory service to provide the “plug and play” view of the PSSA. Further, the details of the directory service entry structure will be described in this sub-section, which is pertinent to understanding the subscriber APIs calls to be described in the later section.

The current version of the PSSA SDK encapsulates the publisher instance creation and its registration with the directory service in one API call shown below:

```
PSS_RESULT_CODE PSS_RegisterEventPublisher( const char* publisher_name,
                                            const char* publisher_address,
                                            const char** topics_published,
                                            const unsigned short topic_count,
                                            PSS_PUBLISHER_HANDLE* publisher,
                                            const char* publisher_latitude,
                                            const char* publisher_longitude,
                                            const char* publisher_altitude,
                                            int publisher_motility )
```

The above call on successful completion would return a **PSS_PUBLISHER_HANDLE** type pointer (*publisher* above), which provides an encapsulated access to the PSSA publisher instance (created and) returned by the underlying publisher utility routines from the SDK. As seen in the API call, every publisher instance is characterized by below attributes:

- **Publisher name** – a user provided name for the publisher instance (*publisher_name* parameter in the call).
- **Publisher address** – a network IP address assigned to the machine running the publisher instance, which is later combined with an appropriate port number (chosen dynamically from the provided available port range by the context) to form a protocol IP (mentioned earlier in the section). (*publisher_address* parameter in the call).
- **Topics published** – the list of the event topics, which would be published by the publisher (*topics_published* array parameter in the call).
- **Topics count** – the number of the event topics, which would be published by the publisher (*topic_count* parameter in the call).
- **Initial source location information** – the initial/starting latitude and longitude information of the original source (sensor or the sensor platform) of the data, which is being encapsulated into the PSSA events by the publisher in the context (*publisher_latitude, publisher_longitude and publisher_altitude* parameters in the call).
- **Source motility information** – the information of the original source (sensor or the sensor platform) of the data, which is being encapsulated into the PSSA events by the publisher in the context (*publisher_motility* parameters in the call).

The above information attached with each publisher instance is communicated to the directory service in the PSSA cloud, thru the registration process, as shown in Figure 6.

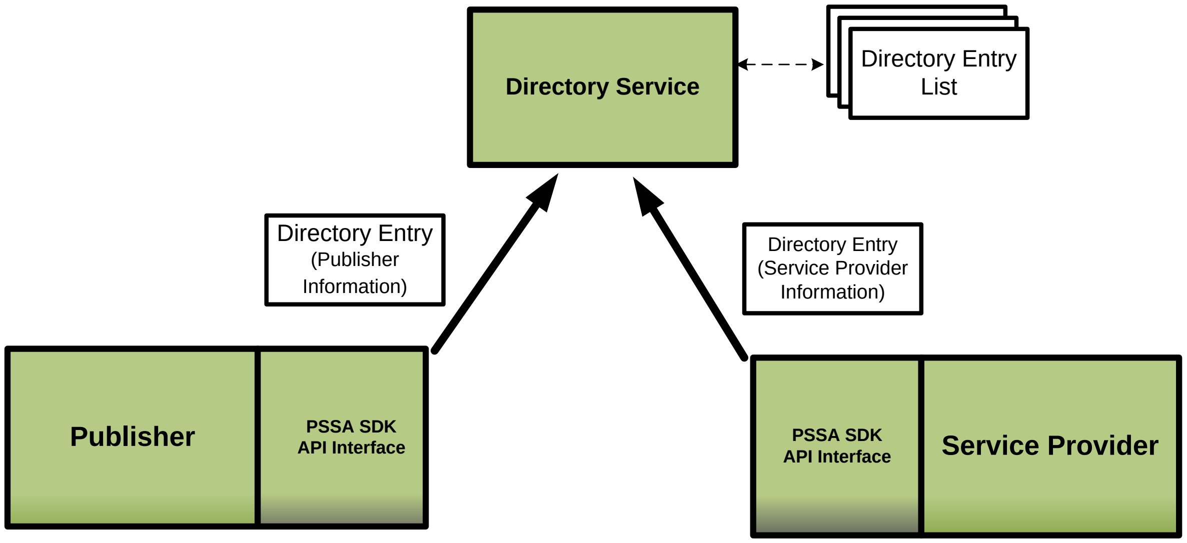


Figure 6: PSSA Registration Process

Figure 6 is a pictorial representation of the publisher and service provider component registration process in the PSSA system. The publisher's or a service provider's registration process in the PSSA cloud with the directory service, is one of the key implementation concepts to support the "Plug and Play" view of the PSSA. As seen in the figure, each publisher or service provider instance communicates their individual information to the directory service which in return provides them to the interested subscribers in PSSA cloud, as they become available in the deployed solution at runtime.

A directory service entry structure is provided to encapsulate the above publisher information as shown below:

```
typedef struct
{
PSS_DIRECTORY_CHANGE_ACTION action;
PSS_DIRECTORY_ENTRY_TYPE entry_type;
PSS_DIRECTORY_MOTILITY motility_type;
PSS_DIRECTORY_GEOLOCATION entry_location;
const char* name;
const char* endpoint;
const union
{
const char* topic;
const char* service_contract;
};

} PSS_DIRECTORY_ENTRY
```

Further, the directory service keeps track of every publisher instance and the service provider in the cloud, thru a collection of these directory entries, as shown in Figure 6. The most important aspect of the above directory service entry structure, which is not provided directly from the publisher attributes, is its status, i.e. if the publisher has been started or stopped, and this information is captured in **PSS_DIRECTORY_CHANGE_ACTION** entry as “ADDED” or “REMOVED” respectively. This entry will invoke necessary subscription and unsubscription logic in the PSSA subscriber components, to be discussed later in the next sub-section.

With respect to the service provider, a similar API calls is provided for creating and registering a service provider module, as shown below:

```
PSS_RESULT_CODE PSS_RegisterServiceProvider( const char* provider_name,
                                             const char* provider_address,
                                             const char* service_contract,
                                             PSS_SERVICE_HANDLE* service )
```

Conceptually, the basic functional difference between the publisher and the service provider comes from the fact that the publisher, publishes the events at a defined rate (or appropriately as required by the customized logic governing it) and the service provider just responds appropriately, to the request from the other components in the PSSA cloud. The request and response message formats for the service provider is same the event message template discussed previously in this section. Further, the publisher handle is used to publish the formatted event messages into the cloud using the below API call:

```
PSS_RESULT_CODE PSS_PublishEvent( PSS_PUBLISHER_HANDLE publisher,
                                  const char* topic_name,
                                  PSS_MESSAGE_HANDLE message )
```

Whereas, the service handle is used to wait on the request (using appropriate timeout period) and respond accordingly using the below API calls:

```
PSS_RESULT_CODE PSS_ReceiveRequest( const PSS_SERVICE_HANDLE service,
                                    long milliseconds_timeout,
                                    PSS_MESSAGE_HANDLE message )

PSS_RESULT_CODE PSS_SendResponse( const PSS_SERVICE_HANDLE service,
                                  PSS_MESSAGE_HANDLE message  )
```

Finally, the SDK provides required API calls to unregister publisher and service provider module, by removing the appropriate entry from the directory service and also releasing the allocated resource for the related underlying SDK context, as shown below:

```
PSS_RESULT_CODE PSS_UnregisterEventPublisher
                                        (const PSS_PUBLISHER_HANDLE publisher)

PSS_RESULT_CODE PSS_UnregisterServiceProvider
                                        (const PSS_SERVICE_HANDLE service)
```

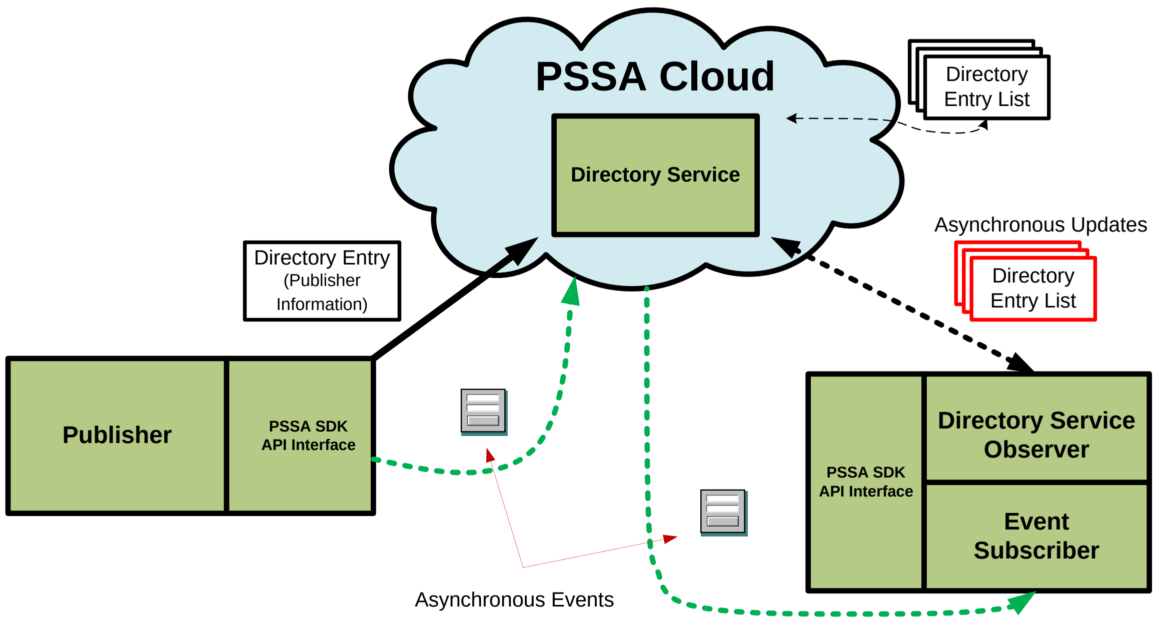


Figure 7: PSSA Directory Service Notification Event

As depicted in Figure 7, every PSSA subscriber component requires at least one directory service observer instance in it, to retrieve asynchronous updates from the directory service in the cloud. Further, based on the envisioned processing or business logic might require one or more event subscriber or service consumer instances to fully establish and complete the "Plug and Play" capability and view of the PSSA architecture. The PSSA SDK provides required API calls to build these above mentioned subscriber and directory observer low-level SDK modules.

***PSSA Subscriber & Directory Service Listener API Calls*:** The "Plug and Play" capability of the PSSA architecture can be complete and possible only with appropriate subscriber components that can be positioned / built to receive the updates, in form of the directory service entries, from the directory service. Followed by which, will eventually subscribe to the publisher or service provider instances (present in the PSSA component of interest) and receive event messages in a pub-sub mode (for the publisher consumers) or in a request-response mode (for the service provider consumers) of communication. As a consequence of which, a PSSA subscriber component should have at least one low-level SDK module called directory service observer and could have more than one subscriber or service consumer modules in it, as shown in Figure 7.

The PSSA SDK provides below APIs to build and destroy a directory service observer module, which is responsible for acquiring the directory service entries dynamically (asynchronously) as they are added or removed as a consequence of starting new publishers or closing the existing publishers in the PSSA cloud:

```
PSS_RESULT_CODE PSS_AttachDirectoryServicesObserver
                              ( const PSS_DIRECTORY_ENTRY_CALLBACK callback,
                                const unsigned long hint,
                                PSS_DIRECTORY_OBSERVER_HANDLE* observer )
```

```
PSS_RESULT_CODE PSS_DetachDirectoryServicesObserver
                        ( const PSS_DIRECTORY_OBSERVER_HANDLE observer )
```

As seen in the above creation API, to support the required asynchronous mode of the directory service update acquisition, a call-back method with the hint option is provided. This call back function of format shown below…

```
      typedef void ( *PSS_DIRECTORY_ENTRY_CALLBACK )( PSS_DIRECTORY_ENTRY*
      changes, const unsigned short change_count, const unsigned long hint )
```

…provided by the user will receive updates (as list of directory entries mentioned in the previous sub-section) accordingly by regularly invoking the below API call:

```
PSS_RESULT_CODE PSS_CheckForDirectoryServicesUpdates
                        ( const PSS_DIRECTORY_OBSERVER_HANDLE observer,
                          long millisecondsTimeout )
```

On reception of the publisher topic/entry (mainly the publisher's endpoint address and the topic), which is of interest for the PSSA component's business logic, the subscriber instance can be created to receive events using the below API call for creation:

```
PSS_RESULT_CODE PSS_Subscribe( const char* publisher_endpoint,
                               const char* topic_name,
                               PSS_SUBSCRIBER_HANDLE* subscriber )
```

Upon successful creation of the subscriber instances, below API call can invoked at user-defined rate with appropriate time out to receive the published event s from the publisher in pub-sub communication mode (asynchronously) :

```
PSS_RESULT_CODE PSS_ReceiveEvent( const PSS_SUBSCRIBER_HANDLE subscriber,
                                  long milliseconds_timeout,
                                  PSS_MESSAGE_HANDLE message  )
```

The subscriber instance and its related resources can be released at the time of unscrewing, if at all the publisher was stopped in the PSSA cloud (indicated thru the "REMOVED" entry updates from the directory service) or its determined by a business logic to stop receiving events by using the below API call:

```
PSS_RESULT_CODE PSS_Unsubscribe( const PSS_SUBSCRIBER_HANDLE subscriber )
```

Similar APIs are available to create service consumer instances, i.e. create connection, send request and receive response as shown below for a request-response communication mode:

```
PSS_RESULT_CODE PSS_CreateConnection( const char* service_address,
                                      PSS_CONNECTION_HANDLE* connection )
```

```
PSS_RESULT_CODE PSS_CloseConnection( const PSS_CONNECTION_HANDLE connection )

PSS_RESULT_CODE PSS_SendRequest( PSS_CONNECTION_HANDLE connection,
                                 const PSS_MESSAGE_HANDLE message )

PSS_RESULT_CODE PSS_ReceiveResponse( const PSS_CONNECTION_HANDLE connection,
                                     PSS_MESSAGE_HANDLE message )
```

As it is observed, these above service consumer/ subscriber API calls are complementary to the service provider /publisher API calls mentioned in the previous sub-section. Finally, apart from the mentioned default API calls the PSS SDK also provides the advance publisher, subscriber and directory observer creation calls, thru which user can create and specify a more than one independent context (which is used as a default setting in the SDK) to effectively use the computational resources available to him on a given machine by using below API calls:

```
PSS_RESULT_CODE PSS_SubscribeEx( const char* publisher_endpoint,
                                 const char* topic_name,
                                 const PSS_CONTEXT_HANDLE contextex,
                                 PSS_SUBSCRIBER_HANDLE* subscriber  )

PSS_RESULT_CODE PSS_AttachDirectoryServicesObserverEx
                               ( const PSS_DIRECTORY_ENTRY_CALLBACK callback,
                                 const unsigned long hint,
                                 const PSS_CONTEXT_HANDLE contextex,
                                 PSS_DIRECTORY_OBSERVER_HANDLE* observer )

PSS_RESULT_CODE PSS_RegisterEventPublisherEx
                               ( const char* publisher_name,
                                 const char* publisher_address,
                                 const char** topics_published,
                                 const unsigned short topic_count,
                                 const PSS_CONTEXT_HANDLE contextex,
                                 PSS_PUBLISHER_HANDLE* publisher,
                                 const char* publisher_latitude,
                                 const char* publisher_longitude,
                                 const char* publisher_altitude,
                                 int publisher_motility )
```

A tutorial to develop publisher, subscriber and service provider using the C++ based SDK library is provided in Appendix B. Nonetheless, the concepts and APIs presented in this section still are applicable to understand and implement the contents of the tutorial effectively.

4.1.5. Implemented Sample PSSA Components

Using the above mentioned PSSA SDK library APIs and following the guidelines presented in the tutorial provided in Appendix B, sample PSSA components have been developed, to validate the "Plug and Play" nature of the PSSA. Below is the list of the sample PSSA components developed as part of this effort of PSSA validation:

- Image Video Ingestion Service
- RTSP Video Ingestion Service

- HTTP Video Ingestion Service
- Geographically Encoded Objects Really Simple Syndication (GeoRSS) Ingestion Service
- Video Quality Exploitation Service
- Video Tracking Service
- Media Storage & Retrieval Service
- RTSP Video Dissemination Service
- Data Storage Service
- Enhanced Directory Service
- XMPP-PSSA Bridging Service
- Dashboard Client
- OpenLST Client
- Remote Service Launcher Service
- Directory Service Observer

Image Video Ingestion Service: In brief, this ingestion service is developed with an aim to ingest time stamped real-time image frames from image capturing sensor platforms, like IP-based internet cameras or on-board EO/IR sensors of an Unmanned Aerial Vehicle (UAV), into the PSSA cloud as the video frame events. As the image capturing sensor platforms vary in their image format and communication protocol, any exploitation service or application, interested in its imagery, has to develop customized communication module and image conversion module before actually doing the real exploitation processing on the imagery data.

This customized exploitation service creation would not scale well in, effort and resources, as different sensor platforms are introduced in the system. Thus, it would be a necessity to provide a normalized view of these different sensors as the virtual sensors, which generate a common image format and use a common communication protocol.

This normalized communication protocol can be easily achieved by employing PSSA event publisher-based communication and further as part of this effort, the normalized image format is chosen to be MPEG-4 encoded image stream. A pictorial representation of this described internal process is shown next in Figure 8.

During the investigative study of the existing video format standards, the MPEG-4 encoding has proved to be efficient for network streaming the vide frame data and as well as effective on consuming lower disk space by utilizing an MP-4 file format, that can store MPEG-4 compressed frames with highly accurate temporal artifacts, as compared to other existing video format manipulation standards available under Lesser General Public License (GNU), aka LGPL, license.

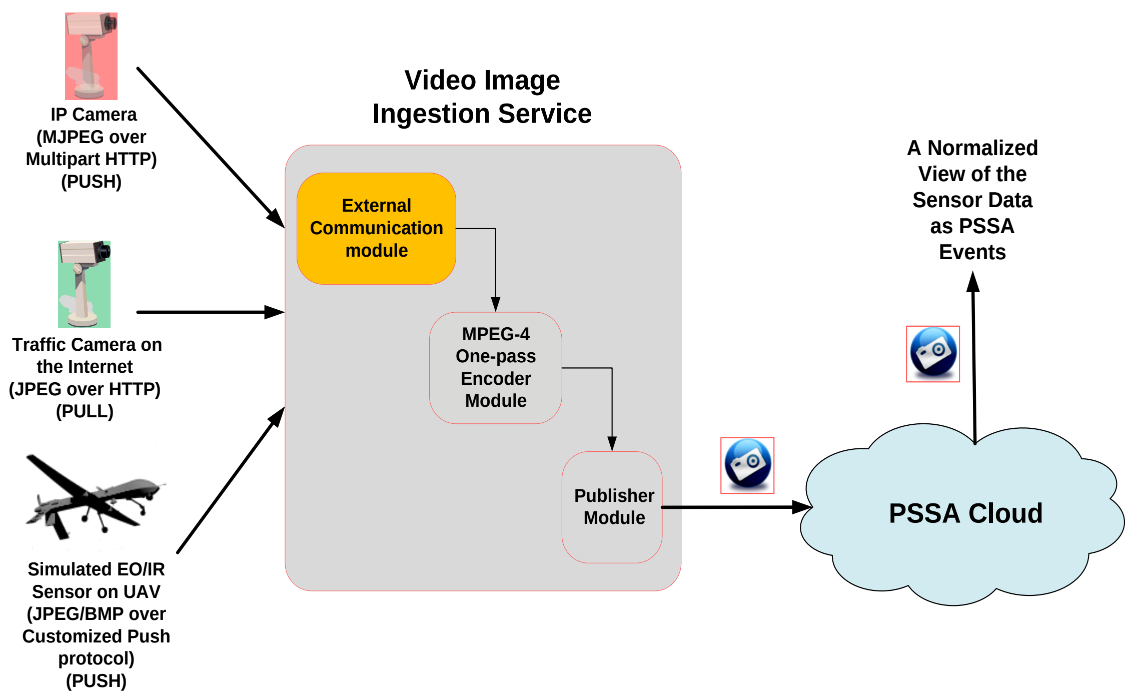

IP Camera (MJPEG over Multipart HTTP) (PUSH)
Traffic Camera on the Internet (JPEG over HTTP) (PULL)
Simulated EO/IR Sensor on UAV (JPEG/BMP over Customized Push protocol) (PUSH)
Video Image Ingestion Service
External Communication module
MPEG-4 One-pass Encoder Module
Publisher Module
PSSA Cloud
A Normalized View of the Sensor Data as PSSA Events

Figure 8: Video Image Ingestion

The Video Image Ingestion Service normalizes the incoming video frames in various formats into the MPEG-4 encoded PSSA video events. It also deals with the various external communication interfaces to acquire the captured data from the sensors. Thus, the Video Image Ingestion Service has been developed to communicate with sensors that support image retrieval via different protocols (currently only simple HTTP, multipart HTTP and a simulated custom push protocol are considered). Further, the different formats of imagery — which are currently Motion JPEG (MJPEG), Joint Photographic Experts Group (JPEG), and Bitmap (BMP) — are considered retrieved from these sensors are encoded to the MPEG-4 stream in real-time with one-pass encoding techniques, with possible key frames (complete) and partial frames. Finally, these encoded MPEG-4 video frames, along with their sensor and timestamp information, are converted PSSA event messages with two parts, namely context and content parts as shown below:

```
Content-Type: text/xml
Content-Location: event
<Event topic="sensor.observation.media.video:acquired?sensorid=1234">
   <Context>
      <Producer>VideoIngestor@192.168.1.104</Producer>
      <ReferenceSensorId>1234</ReferenceSensorId>
      <Timestamp>2010-02-08T14:21:13.3320001</Timestamp>
      <Location>
         <Position>POINT(30 30)</Position>
      </Location>
   </Context>
   <Content>
      <VideoFrame type="partialFrame|keyFrame" bitsPerSecond="20000" width="320"
         height="240" samplePeriod="P0.033S" presentationTime="2010-02-
         08T14:21:13.332001" href="cid:frame1" />
   </Content>
```

```
</Event>

Content-Type: x-application/mpeg4-frame
Content-Location: frame1

...frame bits...
```

As seen above PSSA event message, the first part of the video frame event message is considered to provide context and concerned content information of the publishing event in terms of below attributes:

- **Topic of the publishing event** - This specifies the unique category of the event publishers, I.E. more than one publisher can publish the same event topic type. But, can be made unique by adding a sensorID tag at the end of it.
- **Producer details** - This identifies the PSSA component type and related machine in the PSSA cloud generating it.
- **Reference SensorID** - The unique identification assigned to the sensor, to differentiate and manage the related PSSA events generated from the given sensor in the context.
- **Timestamp** - The time stamp in GMT format, that specifies the image acquired time from ingestion service's perspective.
- **Location** - The geographic location of the sensor, when the present imagery data has been acquired.
- **Content** - This holds the information specific to the image frame captured by the sensor, like if the frame is key or partial (after MPEG-4 encoding normalization), the image quality in terms of bits per seconds, width, height, sample period of the sensor (to capture the frame, which specifies the image frame rate) and the presentation time (which specifies the frame presentation time after the encoder pass). Further, it holds the reference to the part of the event message that contains the raw bits of the frame.

Further, the second part of the event message holds the raw bits of the MPEG-4 encoded frame, that are referenced in the content information in the first part of the message. Thus, this ingestion service on startup, initiates the communication with sensor of interest and before publishing this above designed event message for every acquired image from the sensor, it creates a publisher instance that registers with the directory service in the PSSA cloud, that provides the general details specified in the SDK section.

The developed Image Video Ingestion Service has been tested and verified to work with below sensors, with a maximum frame rate of 30 fps and with video quality of 256 Kbytes per second:

- CISCO PVC 300 IP camera with MJPEG imagery over multipart HTTP protocol.
- AXIS P1311 IP camera with MJPEG imagery over multipart HTTP protocol.
- Generic internet-friendly installed cameras with JPEG imagery over HTTP protocol.
- Simulated UAV sensor feeds with Electro-Optical (EO) and infrared (IR) imagery over customized push protocol.
- Real-time user controlled video distortion feed (MATLAB integrated) simulator over customized push protocol.

Apart from the PSSA SDK routines FFMPEG, Wininet, Curl and other utility libraries are employed to develop the Video image ingestion service.

RTSP Video Image Ingestion Service: The Video Image Ingestion Service, discussed earlier, provided a generic way to develop a video event ingestion service, which has to deal with varied communication protocols and image formats pertaining to a given sensor in the context, by invoking appropriate normalization process in real-time. But, the RTSP Video Image Ingestion Service is a specialized ingestion service that targets only the IP-based cameras which stream out MPEG-4 encoded video over the RTSP protocol. Thus, this RTSP video ingestion service skips the one-pass encoding step mentioned in Image video ingestion service (see Figure 8, above). Further, it employs the same PSSA video frame event message format and registration mechanism described in the previous sub- section, in the context of the Video Image Ingestion Service.

The developed RTSP Video Ingestion Service has been tested and verified to work with below sensors, with maximum frame rate of 30 fps and the video quality of 256 Kbytes per second:

- CISCO PVC 300 IP camera.
- AXIS P1311 IP camera.

Apart from the PSSA SDK routines, Live555 media library and other utility libraries are employed to develop the RTSP Video ingestion service.

HTTP Video Image Ingestion Service: When Summer at the Edge (SATE) started, there was a need to ingest video from different sensors like smartphones, tablet devices, robots etc. The HTTP Video Ingestor Service was developed to provide the capability to ingest real-time video using the HTTP protocol. This allows any sensor device that can connect over TCP/IP to the HTTP Video Ingestor Service to publish streaming video on the PSSA cloud. This service has two components; one provides a HTTP service and the other provides frame publishing service. The HTTP server opens a user-specified port and listens for HTTP requests from various sensors. The HTTP service is designed to handle multiple different types of incoming video streams. However, in order to support the SATE 2011 iStream project, we have only implemented an H.264 handler that is used to process H.264 frames sent from an iPhone application using the HTTP protocol. The H.264 frames received by the HTTP service are transcoded as bitmap images to create video event messages that are then published into PSSA cloud. This service employs the same PSSA video frame event message format and registration mechanism used by the Video Image Video Ingestion Service described earlier in section 0. The HTTP Video Imaging Ingestion Service has been initially tested with iStream project. iStream is an iPhone application which captures raw video frames on the iPhone, decodes them as H.264 frames and transmits them over HTTP to a server.

GeoRSS Ingestion Service: In brief, as shown later in this document in Figure 12, this GeoRSS Ingestion Service, as suggestive of its name, is built to ingest the time-stamped GeoRSS feeds from the real-time or archived news/government/user blog websites available in the internet, into the PSSA cloud as the GeoRSS text events. Conceptually, these websites are visualized as text streaming sensors and there exists three different GeoRSS feed formats that any given GeoRSS source can employ as mentioned below:

- World Wide Web Consortium (W3C) version.
- Simple GeoRSS

- GML/Pro GeoRSS

As with any ingestion service, the PSSA SDK publisher instance creation and registration is required for this service also. Further, like the previous Video Image Ingestion Service, this ingestion service has to provide a normalized view of the GeoRSS sources with varied GeoRSS feed formats. As a consequence of the data being simple text/XML format, the logic is designed to use a customized PSSA event message format as its normalized GeoRSS data format for all the varied GeoRSS sources. Thus, all the different GeoRSS feeds are converted in real-time into the below GeoRSS text stream event:

```
Content-Type: text/xml
Content-Location: event
<Event topic="sensor.observation.text.georss:acquired?sensorid=GeoRSS03">
     <Context>
           <Producer>GeoRSSIngestor@192.168.1.104</Producer>
            <ReferenceSensorId>GeoRSS03</ReferenceSensorId>
             <Timestamp>2010-04-21T15:30:29.623000</Timestamp>
                   <Source>
                        <Description>USGS</Description>
                         <InetAddress>www.4qbase.com</InetAddress>
                    </Source>
     </Context>
      <Content>
          <FeedItem
                title="Major event happened here"
                description="Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean
auctor, metus in iaculis fermentum, massa ligula imperdiet libero, eget accumsan nunc nisl
vitae est. "                     pubdate="2010-04-15T13:40:59"
                     geolocation="POINT( -83.0196 40.00173 )" />
       </Content>
</Event>
```

As seen above PSSA event message, there exists only one message part in the event message and it is considered to provide context and concerned complete content information of the publishing GeoRSS event in terms of below attributes :

- **Topic of the publishing event** - This specifies the unique category of the event publishers, i.e. more than one publisher can publish the same event topic type. But, can be made unique by adding a sensorID.
- **Producer details** - This identifies the PSSA component type and related machine in the PSSA cloud generating it.
- **Reference SensorId** - The unique identification assigned to the sensor, to differentiate and manage the related PSSA events generated from the given sensor in the context.
- **Timestamp** - The time stamp in GMT format specifies the GeoRSS feed acquired time from ingestion service’s perspective.
- **Location** - The geographic location of the senor i.e. GeoRSS source, when the feed is acquired, which is different from the actual feed location.
- **Content** - This holds information specific to the GeoRSS feed acquired by the sensor, like the title of the feed, the complete description, the publication date and time and finally the geolocation of the GeoRSS feed.

Apart from the PSSA SDK routines' eXtensible Stylesheet Language Transformations (XSLT) transformation library, Curl and other utility libraries are employed to develop the GeoRSS ingestion service.

***Video Quality Exploitation Service*:** Functionally, an exploitation service can be seen as the PSSA component that employs in its application at least three low-level PSSA SDK modules, namely the directory service observer module, subscriber module and the publisher module, and glues them with the required business i.e. exploitation logic. The first two modules, directory observer and subscriber are necessary to receive the events of interest that are active in the PSSA cloud to run the exploitation algorithm on them. The last publisher module is required to publish back into the cloud, the new events, whose content is based on the exploitation outcome on the subscribed event messages. In this vein, the Video Quality Exploitation Service in the present context, subscribers to the publishers in the cloud, that publishes video frame events (as mentioned and introduced in the early sub- sections) and generates the various video quality metrics that will be published back as video quality events in to the cloud.

Getting into the specifics of the Video Quality Exploitation Service, when the directory entry updates are received in the directory service observer module, the "topic" entry in the directory service entry structure (mentioned in the SDK sub-section) is checked for a match for the video frame events, i.e. for "sensor.observation.media.video:acquired" and once confirmed, an exploitation session is created that targets the given video event publisher in the context and publishes the video quality events. Thus, as shown in Figure 12 (coming up), the created exploitation session, is responsible to initiate low-level subscriber module that receives the video frames from the publisher and then provide the decoded video frame image (i.e. using the MPEG-4 decode filter from the SDK) to the quality estimating algorithm that generates in real-time the quality metrics like noise and Structural Similarity (SSIM) Index for all three bands in the video frame image. These generated quality metrics are then converted into below PSSA video quality event:

```
Content-Type: text/xml
Content-Location: event
<Event topic="edu.ualr.quality-metric.media.video:qualityestimated?sensorid=1234">
  <Context>
    <Producer>VideoQaulityPublisher@192.168.1.104</Producer>
    <ReferenceSensorId>1234</ReferenceSensorId>
    <Timestamp>2010-02-08T14:21:13.3320001</Timestamp>
  </Context>
  <Content>
   <VideoFrameQualityMetric metric_name="NoiseRed" metric_value="0.998″ unit="db" window="30"
      frame= "233″ />
   <VideoFrameQualityMetric metric_name="NoiseBlue" metric_value="0.998″ unit="db" window="30"
      frame="233″ />
   <VideoFrameQualityMetric metric_name="NoiseGreen" metric_value="0.998″ unit="db" window="30"
      frame="233″ />
   <VideoFrameQualityMetric metric_name="SSIMRed" metric_value="99″  unit="%" window="30"
      frame="233″ />
   <VideoFrameQualityMetric metric_name="SSIMRed" metric_value="99″  unit="%" window="30"
      frame="233″ />
   <VideoFrameQualityMetric metric_name="SSIMRed" metric_value="99″  unit="%" window="30"
      frame="233″ />
 </Content>
</Event>
```

As seen in the video quality event message, which is a single message part event, the context contains information like producer (publisher) details and the reference sensor ID that uniquely distinguishes this publisher from other possible publishers with same event publishing topics. Further, it contains the timestamp information, which is the time at which the exploitation session completed computing the quality information for the specific video frame event (frame) in the context.

The content part of the message contains the quality metrics like noise and SSIM calculated for the video frame and the related information like the frame number in the subscribed incoming event, metric name, units of measurement and finally the estimation window, which specifies the estimation window size of the frames over which the exploitation session computes the quality (i.e. how many frames are to be skipped or accounted before computing/estimating the quality of the video frame) for a given sensor in the context. A publisher module is created and registered with the directory service with the topic specified in the event for the sensor in the context thru this dedicated exploitation session per video sensor in the cloud.

Finally, the Video Quality Exploitation Service is built to handle multiple live sensor feeds (from multiple ingestion services for different sensors) simultaneously and compute/ publish quality events in real-time, back into the PSSA cloud. Apart from the PSSA SDK routines OpenCV and other utility libraries are employed to develop the Video Quality Exploitation Service.

Video Tracking Exploitation Service: In brief, the video tracking exploitation algorithm subscribes to the video frame events, which are being generated from a video sensor registered with the PSSA cloud, and computes the possible moving tracks of objects present in the field of view of the sensor. Further, it also publishes the computed tracks on the video stream events back into the PSSA cloud by creating independent track computing sessions per sensor. The Video Tracking Exploitation Service employs the same logic described and developed in the context of the Quality Exploitation Service to subscribe to the video frames and publish related video track events.
Finally, like the Video Quality Exploitation Service, it is built to handle multiple live sensor feeds simultaneously and compute/ publish tracking events in real-time, back into the PSSA cloud, as shown in Figure 12. Moreover, by virtue of the pub-sub communication model embedded in the architecture, it should be observed that a single publishing event (from a given sensor) is available for exploitation to two different exploitation services simultaneously, which has been tested and verified accordingly. Apart from the PSSA SDK routines OpenCV and other utility libraries are employed to develop the Video Tracking Service.

Media Storage & Retrieval Service (MSRS): In brief, the MSRS is one of the core services of the PSSA, that is primarily, responsible to capture the video stream events (from multiple video frame ingestion services) in the PSSA cloud and store them efficiently and accurately on a storage disk, in manageable file segments.
Secondly, provide a service provider in the cloud, that would support retrieval of the stored video file segments (stored in an organized fashion per sensor and with temporal information intact with millisecond accuracy) and further provide "conceptually" a continuous and accurate single logical file view, with the retrieval query logic that effectively hides the disk-level file segments.

As mentioned in the previous section, the normalized MPEG-4 compressed video frames being published into the PSSA cloud are available simultaneously to multiple subscribers like Video

Quality Exploitation Service, Vehicle Tracking Services and any other video stream processing services. Leveraging the above property of the architecture and using the developed (multiple subscription) logic for the exploitation services, the storage functionality of the MSRS is accomplished by storing the live sensor video frame events (the raw MPEG-4 frame data (from the second message part of the event message) and its related attributes provided in the context part of the event) in real-time, as a temporally indexed standard MP4 video stream format on a distributed file system.

As shown next in Figure 9, the storage functionality of the MSRS has been designed appropriately to handle multiple sensor feeds simultaneously by providing apt storage strategy for each individual sensor (leveraging the multiple video frame event subscription logic explained earlier in the exploitation service sections).

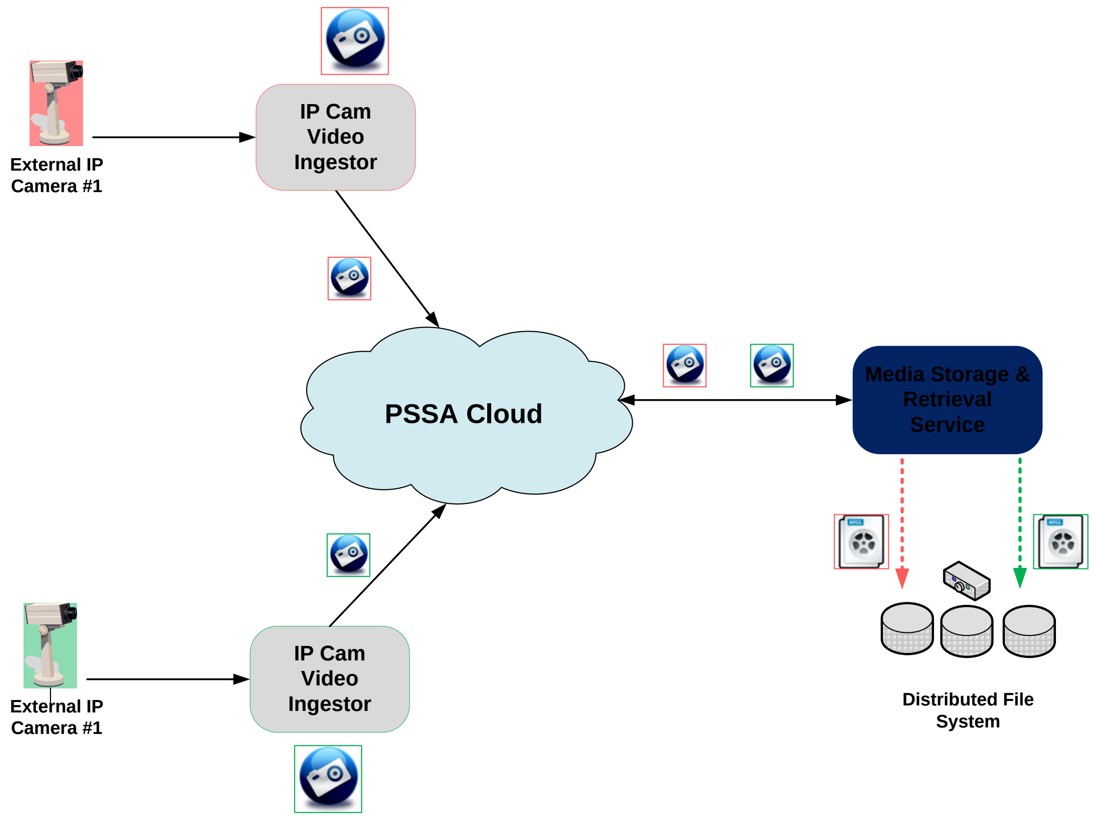
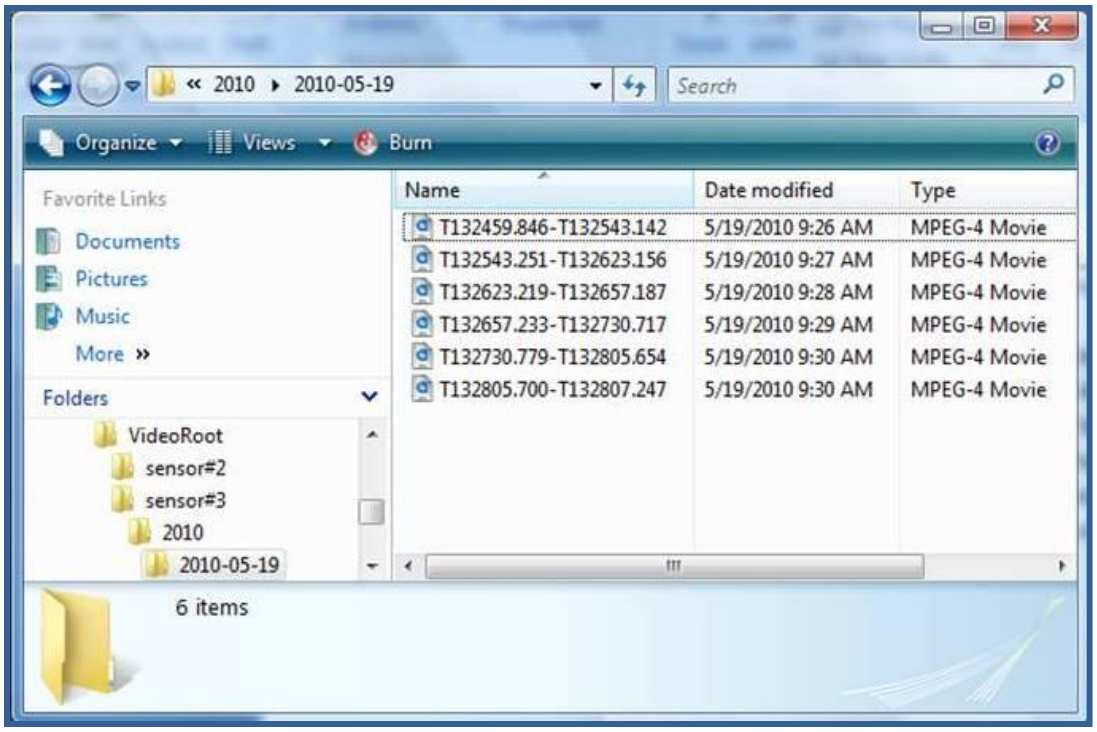


Figure 9: Multiple Simultaneous Video Sensor Feeds Supported by Media Storage Service

To make the stored video format file more manageable and accessible by a standard retrieval interface, a file chunking strategy has been employed. The chunking strategy for a given sensor starts with a root folder, with the sensor identification number, followed by the Gregorian year, day and the real chunk files of the video segments.

The naming convention of the individual video file segments is designed to reflect the time interval of the chunk like “StartTime- EndTime.MP4” and usually limited to few minutes of duration, for quick file access during the retrieval mechanism to be explained later in the section. The “Start Time” and “End Time” are represented in ISO 8601 standard convention and the sample file organization strategy can be seen in Figure 10, following.



Name	Date modified	Type
T132459.846-T132543.142	5/19/2010 9:26 AM	MPEG-4 Movie
T132543.251-T132623.156	5/19/2010 9:27 AM	MPEG-4 Movie
T132623.219-T132657.187	5/19/2010 9:28 AM	MPEG-4 Movie
T132657.233-T132730.717	5/19/2010 9:29 AM	MPEG-4 Movie
T132730.779-T132805.654	5/19/2010 9:30 AM	MPEG-4 Movie
T132805.700-T132807.247	5/19/2010 9:30 AM	MPEG-4 Movie

Figure 10: MSRS Chunking Strategy

The chunking strategy for a sensor starts with a root folder, with the sensor identification number, followed by the Gregorian year, day and the real chunk files of the video segments. As shown above, the naming convention of the chunk file used is “Start Time-End Time.MP4” and further the “Start Time” and “End Time” use ISO 8601 time format convention.

The retrieval functionality of MSRS is designed to use low-level service provider and publisher modules (from SDK) with a simple but millisecond accurate query logic to bind them. The retrieval mechanism is accomplished using below steps:

1. At startup of the MSRS, the application creates a media streaming service provider module, which registers and announces its availability in the PSSA cloud thru the directory services and listens for any query on the advertised address.

2. The interested service consumer (media consumer) has to create the low-level service consumers (from SDK) and sends a request to the media streaming to retrieve the media for particular sensor ID and starting from the particular time in already introduced ISO 8601 standard.
3. Upon the request from the consumer, the service provider logic that is aware of the file chunking strategy to store the file segments does a query against the chunking strategy with the requested sensor ID and the start time. Based on the presence of the request file segment or segments, it creates streaming session, with a unlisted and dedicated publisher (this is a low-level publisher that doesn't register with the directory service, so it's not visible to other PSSA components in the PSSA cloud) and provides this private publisher's subscriber details with the unique streaming session identification value to the service consumer, as part of the response to the request mentioned in Step 2, in the request-response communication mode mentioned in the PSSA SDK section.
4. Upon the response the interested service consumer logic creates a low-level subscriber that is now ready to receive the video stream for the requested sensor and with requested starting time, as the video frame event messages (explained earlier in the Video ingestion service section). The readiness of the subscriber is communicated to the media streaming service provider in the request-response fashion, with the allocated session identification value and the start time.
5. The Media Streaming Service, upon the request with the session identification value, initiates the video frame event publishing logic that retrieves the frames from the contiguous file segments (arranged based on the file chunking strategy) and streams (publishes) them continuously to the private subscriber in the client, until the next step happens, thus providing the continuous view of the archived sensor video stream. The streaming/ publishing is continuous and also acknowledges seeks by the client thru the request pattern mentioned in Step 4 (i.e. streaming session details with the different start time).
6. When, the client decides to tear down the streaming session, he sends the request with the allocated session identification value and the tear down command, upon which the streaming session is de-allocated and the publisher is gracefully shut down.

Thus, the above logic is built into the MSRS to provide the client with a continuous and accurate single logical file view of the archive video segments, which are stored over the sensor's registered life time in the PSSA system. Further, the MSRS is built to support multiple simultaneous storage sessions, to store the video events from different ingestion services simultaneously and also to support multiple simultaneous streaming sessions, for the different and same sensors with the archived video segments.

RTSP Video Dissemination Service (RVDS): It is evident from the last sub-sections that only the PSSA-based media consumer components, are positioned to retrieve the video feeds from the video streaming sensors that are registered with the PSSA cloud, via direct subscription to the ingestion service or thru the Media Storage and Streaming Service.
To demonstrate the feasibility that any third party, off-the-shelf, media consumer/player (like a VLC player) can be employed to access the live and archived video stream active in the PSSA

cloud, the RVDS has been built. Internally, the RVDS is a combination of a media consuming PSSA component and standards-compliant RTSP streaming server. Thus, as shown in Figure 11, based on the user (via VLC player) request sent thru the RTSP server interface for a particular sensor in PSSA, the video event subscription for that requested sensor is established and on the reception of the every video frame event, the server component streams the frames to the requested user in the context.

Currently, the RVDS is built to handle multiple simultaneous media streaming sessions for different or same sensor that are active in the PSSA cloud. Apart from the PSSA SDK routines Live555 media server library and other utility libraries are employed to develop the RTSP Video Dissemination Service.

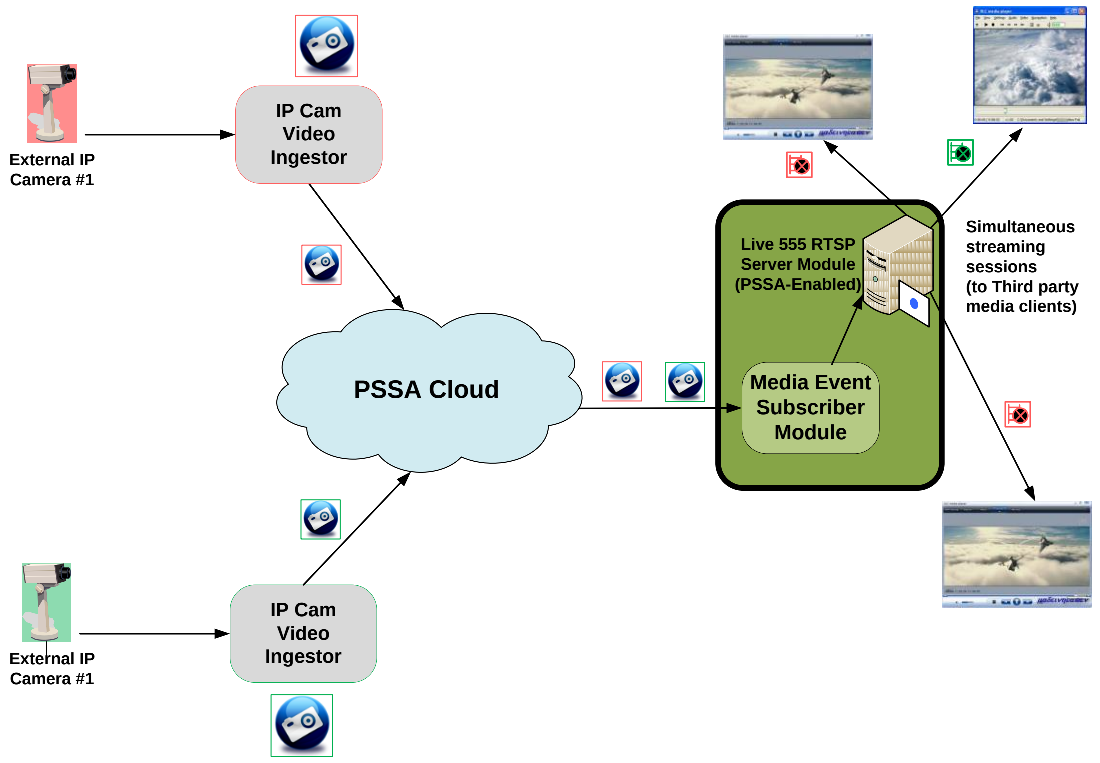

IP Cam Video Ingestor
External IP Camera #1
Live 555 RTSP Server Module (PSSA-Enabled)
Simultaneous streaming sessions (to Third party media clients)
PSSA Cloud
Media Event Subscriber Module
IP Cam Video Ingestor
External IP Camera #1

Figure 11: RTSP Video Dissemination Service

The RVDS is built to handle multiple simultaneous media streaming sessions for different or same sensor that are active in the PSSA cloud. In brief, it utilizes a PSSA-enabled open source Live555 RTSP streaming server to accomplish its functionality.

Data Storage Service (DDS): In brief, as shown in Figure 12, the DSS is one of the core services of the PSSA that is responsible to store the time stamped text-based PSSA events active in PSSA cloud (like Video Quality, Video Tracking and GeoRSS events mentioned earlier) in a database using a user- defined database schema (unique for every event type [topic]) in real-time.

The DSS is implemented with same basic glue logic, mentioned earlier for other event driven services, to employ the directory observer and the subscriber modules to receive and check for events of interest in the PSSA cloud. Further, the DSS is built to handle multiple event generation services (exploitation and Ingestion) by creating simultaneous and exclusive storage sessions for each of them. Current version of the implementation doesn't support retrieval mechanism, but which can be easily be built employing the retrieval mechanism developed for the Media Storage and Streaming Service. Apart from the PSSA SDK routines, Postgresql client library and other utility libraries are employed to develop the DSS.

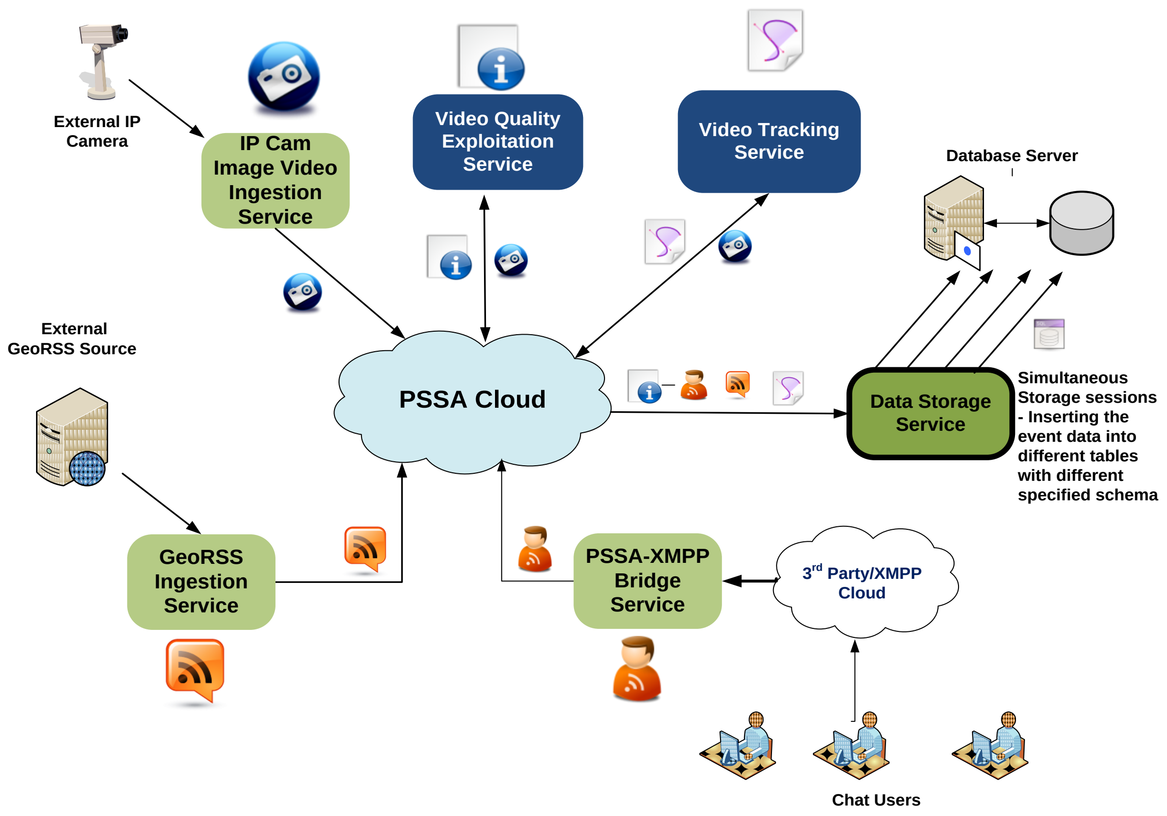


Figure 12: PSSA Data Storage Service

The DSS is built to handle multiple event generation services (exploitation and Ingestion) by creating simultaneous and exclusive storage sessions for each of them. The inserts in the backend database is done by using a user-defined database schema (unique for every event type (topic)) in real-time.

Enhanced Directory Service (with Persistent Repository): The Enhanced Directory Service is a key core service which provides the "plug and play" capability to the PSSA architecture, by providing a mechanism for various services to publish their information and making these information available to the subscribed services in the architecture. The initial version of this directory service implementation had successfully addressed the need to dynamically publish notifications of various service's and sensor's information (as shown below) to the subscribed components in the architecture in real-time, but did not persist this information.

```
typedef struct
{
  PSS_DIRECTORY_CHANGE_ACTION action;
  PSS_DIRECTORY_ENTRY_TYPE entry_type;
  PSS_DIRECTORY_MOTILITY motility_type;
  PSS_DIRECTORY_GEOLOCATION entry_location;
  const char* name;
  const char* endpoint;
  const union
  {
        const char* topic;
        const char* service_contract;
  };

} PSS_DIRECTORY_ENTRY
```

As part of our prototype testing work during SATE and Year at the Edge (YATE) programs at Tec^Edge, we learned that persisting the directory service entry information would increase the reliability and scalability of the directory service and PSS. Thus, the directory service has been enhanced to persist the entry information to a generic repository interface, which can be extended to persist the information on any data storage and indexing system and for the present implementation; a Network File System (NFS) repository is implemented.

The enhanced directory service was successfully tested and it has been verified that this enhancement provides more reliability (i.e. in case of a directory service crash, the next instance can pick up the up-to-date information from the repository seamlessly) and scalability (i.e. in an inter-PSSA (Sky) setup, a PSSA cloud instance's directory service can provide the status of the services in another PSSA cloud instance thru its persistence repository interface).

PSSA-XMPP Bridging Service: In brief, the PSSA-XMPP Bridging Service is designed to bridge/ingest the chat messages from users participating in a given "topic" chat/conferencing room, as PSSA event messages in to the PSSA cloud and also vice-versa. This service provides two major capabilities to the PSSA system; firstly, it provides a concept of viewing a chat user (with any third party, off-the-shelf, XMPP chat client application) as a unique sensor (generating time stamped and related chat (text and image) messages) that can publish events into the PSSA cloud like any other ingestion service described earlier. Secondly, it provides a collaborative environment for consumers of the PSSA system, to collaborate and analyze the data in the events being published into the cloud, which are now can be accessed in real-time thru an off-the-shelf XMPP chat client. As shown in Figure 13, this service is built by combing two bridging modules, the XMPP to PSSA bridge module and the PSSA to XMPP bridge module.

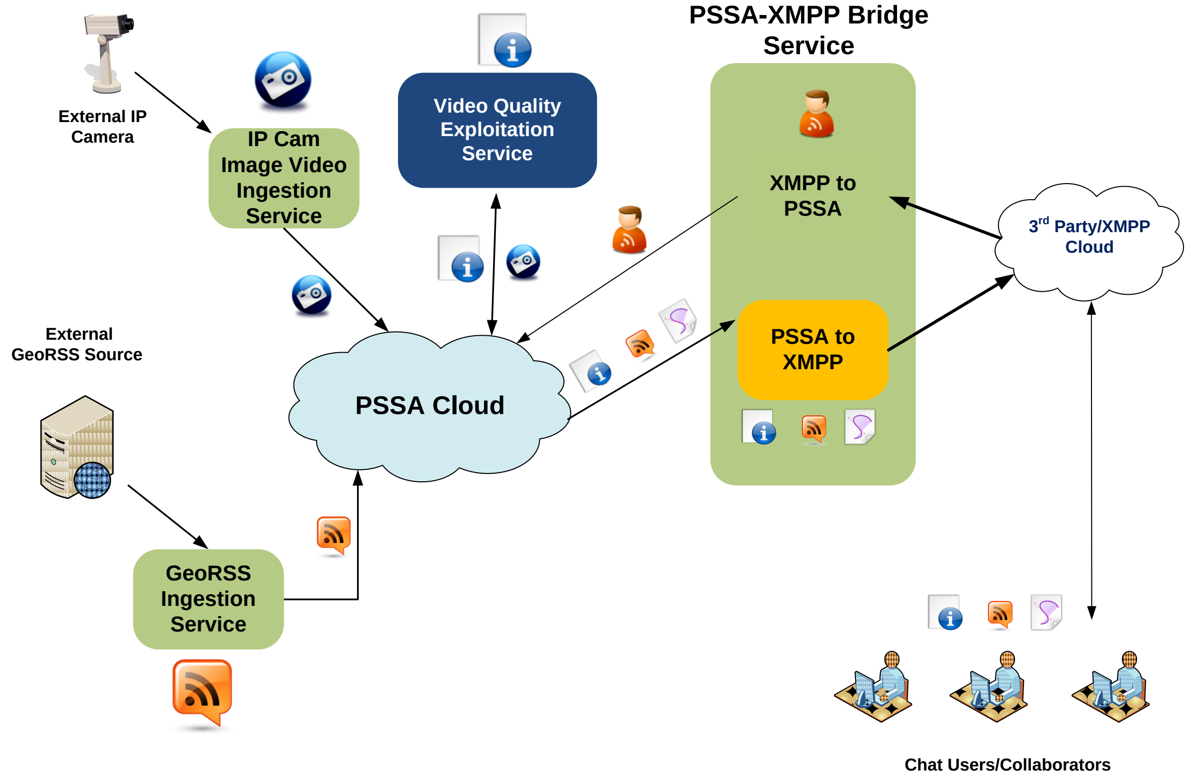


Figure 13: PSSA-XMPP Bridging Service

The PSSA-XMPP Bridging Service is designed to bridge/ingest the chat messages from users participating in a given "topic" chat/conferencing room, as PSSA event messages in to the PSSA cloud and also vice-versa. The XMPP to PSSA bridge module is responsible for observing a given XMPP chat room with specific "topic" and dynamically creating individual PSSA chat event publishers per each registered user (registered with PSSA system with unique ID) in the chat, as they become available. A sample chat event PSSA message is shown below:

```
Content-Type: text/xml
Content-Location: event
<Event topic="sensor.observation.text.chat:acquired?sensorid=24">

  <Context>
   <Producer>XMPPPSSABridge@192.168.1.104</Producer>
   <ReferenceSensorId>24</ReferenceSensorId>
   <Timestamp>2010-02-08T14:21:13.3320001</Timestamp>
  </Context>
  <Content>
   <XMPPMessage UserID="test[24]@programmer-art.org/Smack">
        <Message>Test message from user with id 24</Message>
        <GeoLocation Lat="40.73088" Lng="-84.05452" Elv="0.0" />
        <Date>30 Jul 2010 16:51:41 GMT</Date>
```

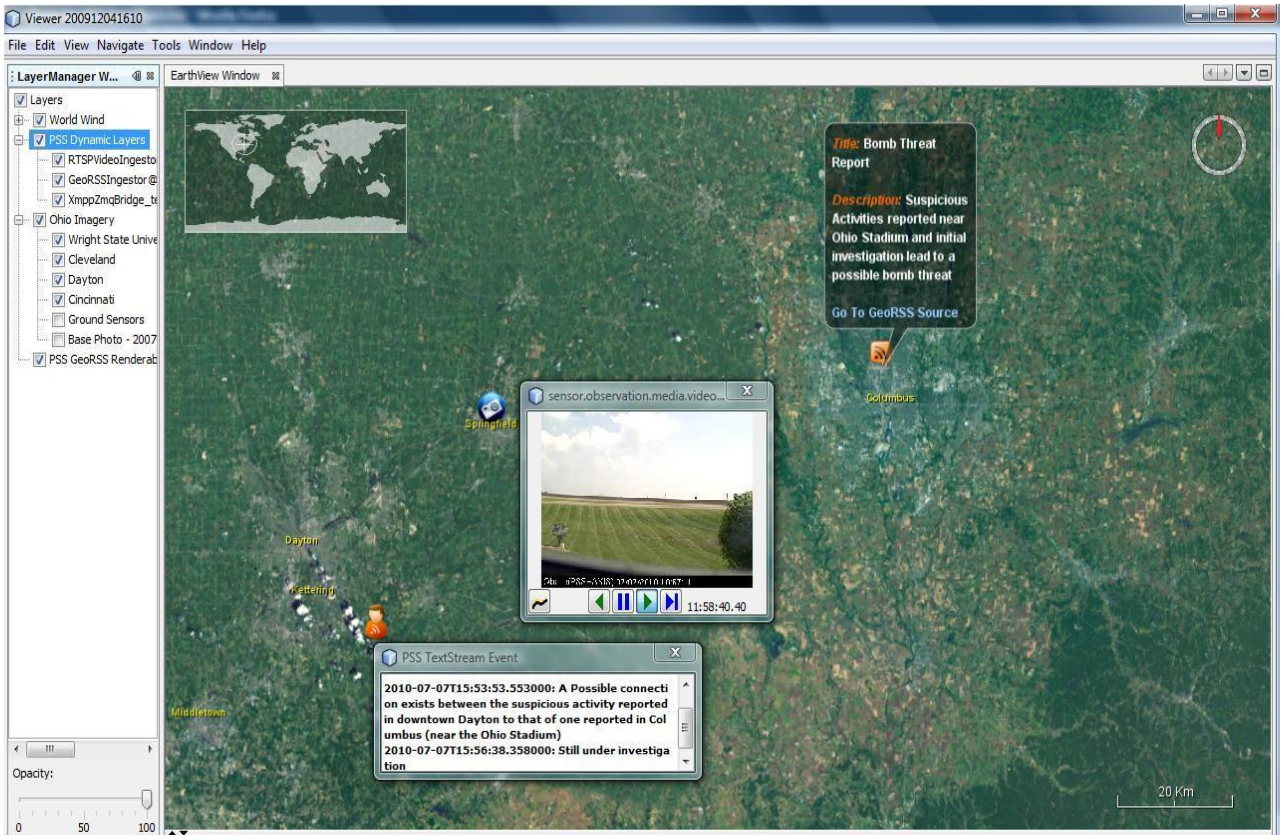
```
          <Geocode Str="null" Cty="null" Ste="null" />
          <ImgAttached>true</ImgAttached>
          <Image type="complete-image" size="100" name="test.jpg" seq="0" 
href="cid:chatimage"  />
      </XMPPMessage>
     </Content>
   </Event>

   Content-Type: x-application/base64
   Content-Location: chatimage

   ...frame bits...
```

As observed, the chat event message format is similar to the video event message format, except for the attached second message part being base64 encoded image in place of MPEG-4 encoded frame and with new XMPP Message block in place of the Video Frame block. The XMPP Message provides the chat message information in "Message" node and besides captures the complete geographic location information to track and support mobile chat users (i.e. as mobile sensors in system). Similarly, the PSSA to XMPP bridge module subscribes to the events of interest and bridges them into the XMPP chat cloud appropriately. Apart from the PSSA SDK routines, AGSXMPP library and other utility libraries are employed to develop the PSSA-XMPP bridge service.

Dashboard Client: The main goal of developing an application like dashboard is to demonstrate the basic capabilities of the PSSA and further provide a means to possibly develop interfaces to build a complete sensor control and data management system. Further, like any other sensor in the PSSA cloud, the dashboard has a built-in capability to publish text streams into the message cloud, as a text stream sensor thru the PSSA-XMPP bridge service. More implementation details of the dashboard can be found in Appendix A – PSSA Dashboard Reference Implementation.

OpenLST Client: OpenLST is the visualization platform developed to demonstrate various sensor fusion capabilities being developed at the Discovery Labs at AFRL's Tec^Edge facility. The OpenLST client was built using Java language SDK and as part of the effort to demonstrate the PSSA capabilities and its ease of integration with OpenLST, the java bindings for the PSSA SDK were developed as mentioned in Appendix C – Java Bindings for PSS SDK. Currently, the PSSA-integrated OpenLST can support live ingestion of various sensor data with appropriate visualization modules in real-time, with a new ability to detect and add new sensors to the platform seamlessly. In addition, multiple OpenLST visualization platforms are able to simultaneously access the same live data and exploitation results. As another benefit of integration, the PSSA provided collaboration capabilities allowing users of the OpenLST to share text communication and sensor feeds with one another.

These capabilities in the OpenLST are possible by utilizing the same low-level SDK module gluing logic mentioned and described for the dashboard client in Appendix A – PSSA Dashboard Reference Implementation. A screenshot of the PSSA Integrated OpenLST is shown in Figure 14, with three sensors Video, GeoRSS and the Chat User. The PSSA Integrated OpenLST can support live ingestion of various sensor data with appropriate visualization modules in real-time, with a new ability to detect and add new sensors to the platform seamlessly.



Title: Bomb Threat Report

Description: Suspicious Activities reported near Ohio Stadium and initial investigation lead to a possible bomb threat

Go To GeoRSS Source

```
2010-07-07T15:53:53.553000: A Possible connecti
on exists between the suspicious activity reported
in downtown Dayton to that of one reported in Col
umbus (near the Ohio Stadium)
2010-07-07T15:56:38.358000: Still under investiga
tion
```

Figure 14: PSSA Integration with OpenLST

Remote Service Launcher Service (RSLS): The RSLS is developed in the PSSA to provide a means to possibly develop interfaces to build a complete service control and status management system. As like any service in the PSSA cloud, the RSLS can receive request from the service consumer, to launch or disable a particular service in the PSSA cloud. Further, its backend server logic is designed with simple non-intelligent resource utilization technique to distribute different services in the given cloud over a cluster of compute and store nodes appropriately**.**

The RSLS is developed using the PSSA SDK's java bindings, described in Appendix D, used to integrate PSSA with OpenLST and the backend logic server is developed using Windows Communication Foundation (WCF) technology. Moreover, the RSLS is tested and verified to function, with a RSLS consumer plug- in developed for OpenLST platform as shown in Figure 15.

The Remote Service Launcher Service is developed to provide a mechanism to invoke and control different services like Ingestion, exploitation and storage service in the PSSA cloud. Moreover, a simple non-intelligent distributed resource utilization approach is implemented in the backend to support this service.

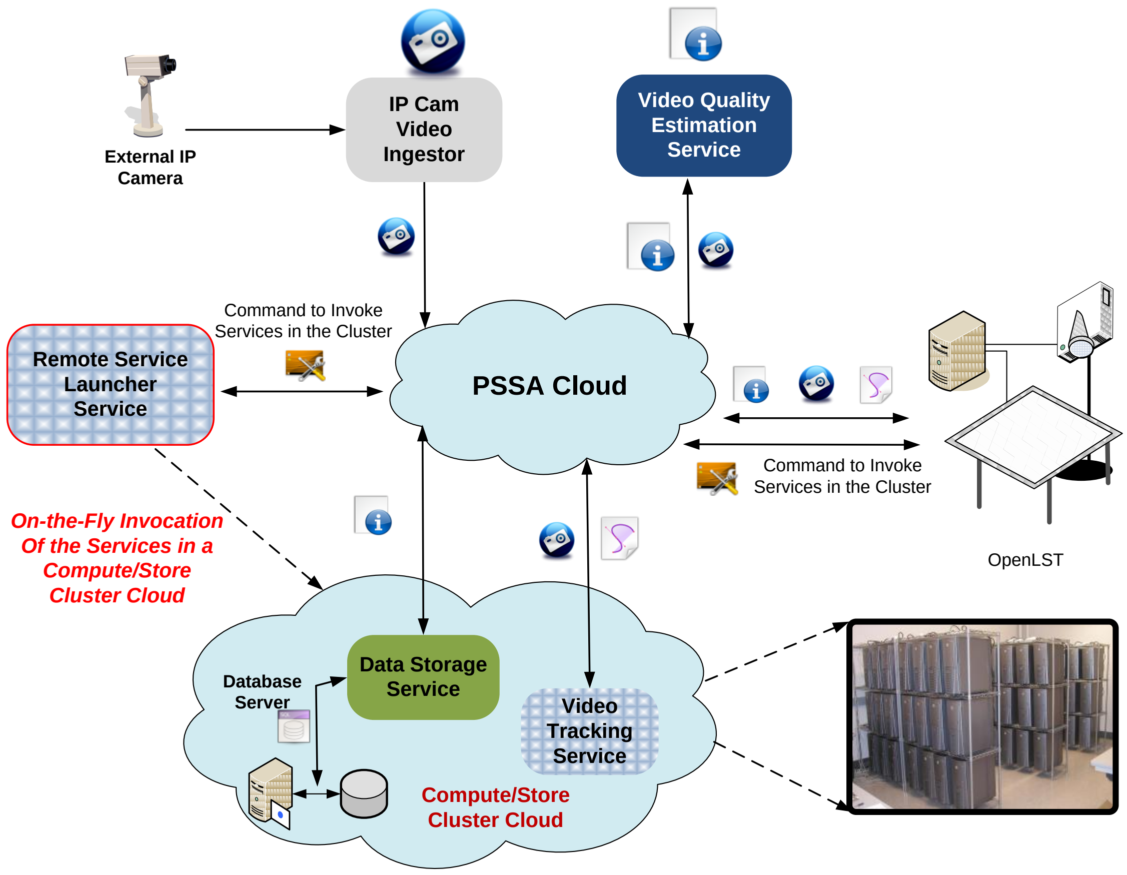

External IP Camera
IP Cam Video Ingestor
Video Quality Estimation Service
Command to Invoke Services in the Cluster
Remote Service Launcher Service
PSSA Cloud
Command to Invoke Services in the Cluster
OpenLST
On-the-Fly Invocation Of the Services in a Compute/Store Cluster Cloud
Database Server
Data Storage Service
Video Tracking Service
Compute/Store Cluster Cloud

Figure 15: Remote Service Launcher

Directory Service Observer: Like the Remote Service Launcher Service (RSLS), the Directory Service Observer (DSO) was developed in the PSSA architecture to provide interfaces to support a complete service control and status management system. In brief, the DSO is an HTTP wrapper around the existing directory service in a given PSSA instance, which allows software components that are not connected to the message bus interface thru the PSSA SDK to receive and monitor service and sensor activity in the PSSA cloud. As part of our prototype testing activity during SATE and YATE programs at Tec^Edge, the DSO, in combination with RSLS, was tested successfully using a browser instance from an Open-SIM and Second-Life virtual world environments for a project that required mapping real-world sensor activity to the virtual world and vice versa.

4.1.6. PSSA Collaboration Services

Collaboration is an important capability for users of sensor visualization tools, so we've built into the PSSA reference implementation a simple mechanism to allow integrators and developers of visualization tools to easily incorporate collaboration features into their visualization tools. These collaboration capabilities are built upon the XMPP standard so that many CoTS and open source XMPP clients can be used to collaborate, as well. XMPP technologies provide a lightweight middleware to support instant messaging, remote presence, multi-user chat, voice and video calls, collaboration and routing of eXtensible Markup Language (XML) data. Utilizing this technology, the PSSA supports a rich collaboration environment in which users of the system can exchange text and, if the client device is so equipped, voice and video information.

In addition, PSSA supports the exchange of application context information allowing two or more collaborators to view the same sensor data at the same time. In essence, this exchange allows one user of the system to hand-off sensor data to other users of the system so that they can see the same data while simultaneously maintaining text/voice/video communication. Collaboration services developed for the PSSA provide the ability for users to create and administer dedicated logical groups called “topic” rooms, to participate in multiple “topic” rooms, and to collaborate between “topic” rooms. The messages sent from the users are published to message bus using a PSS-XMPP message bus bridge, which translates the message sent from the user into a PSS event message. Users and PSS services subscribed to the messages will receive messages.

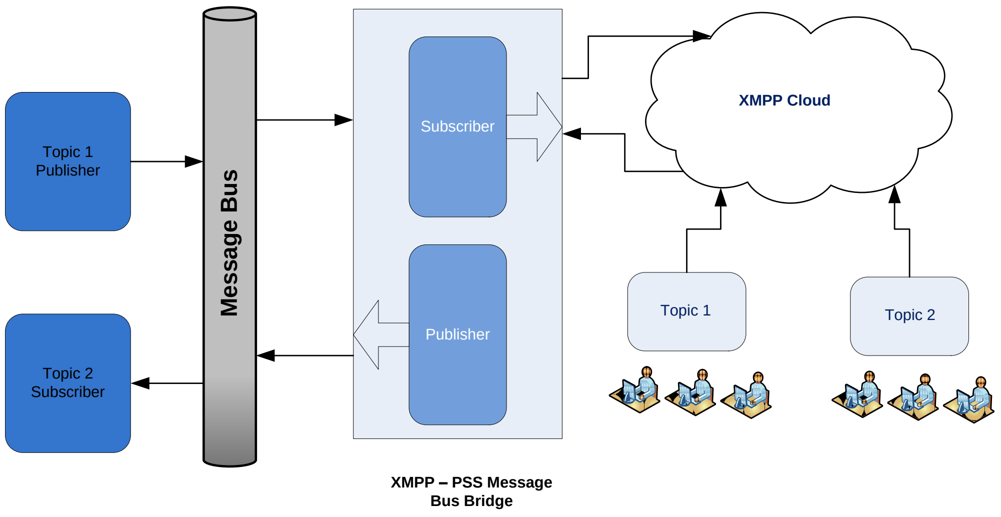


Figure 16: PSSA Collaboration Approach

This approach effectively bridges the PSS message bus across the Internet allowing remotely connected users to interact with the PSS system and, in addition, permits users using third party XMPP-based tools to participate in the collaboration, as well.

### 4.1.7.	PSSA SCALABILITY

A PSSA cloud instance is defined by its core services, which are intrinsically characterized by the “Plug & Play” capabilities. This flexible feature of the PSSA architecture facilities the possibility to further group multiple individual PSSA clouds in a higher level “Plug & Play” sky architecture shown in Figure 17. This envisioned sky architecture is possible by extending the functionality of the currently existing directory service (one of the core services in a PSSA cloud instance) to register itself (when it becomes available) with a global directory service (which defines the sky instance) and also asynchronously push its local directory updates of publishers and services (registered with it) into the sky’s directory services.

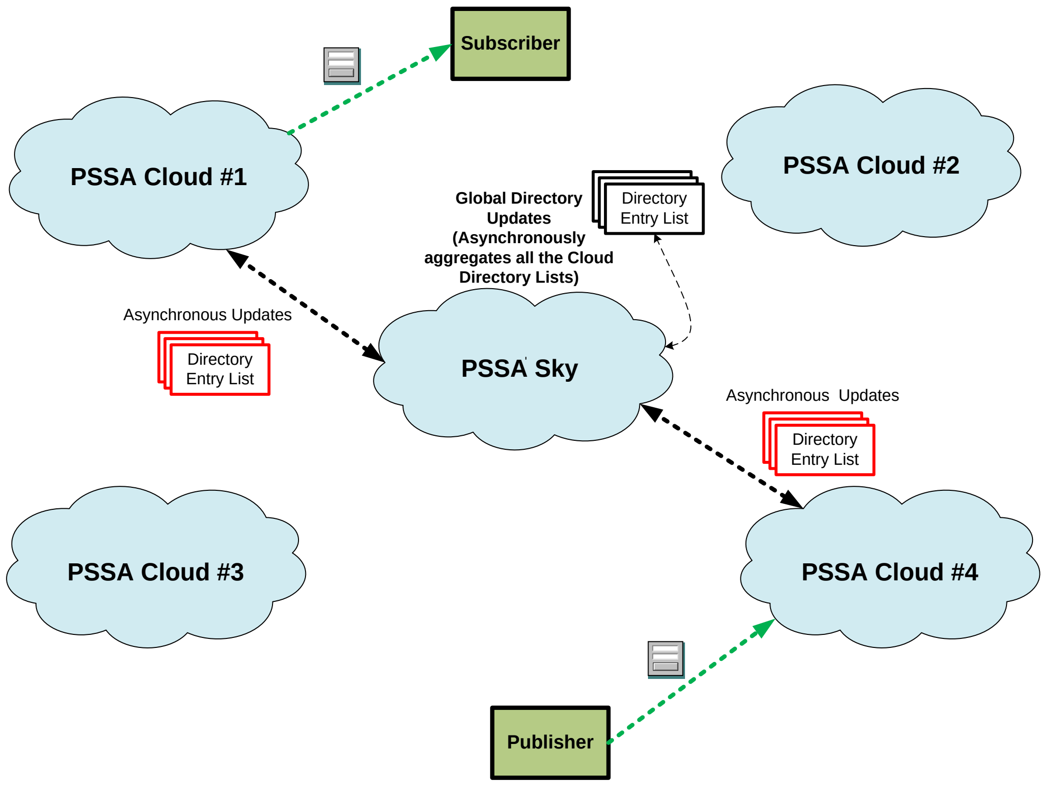


Figure 17: Integration of Multiple PSSA Clouds

Multiple PSSA clouds be grouped to build a PSSA sky architecture, which allows the possibility to make an Ingestion Service (i.e. Publisher) information and events from one instance of the cloud in the sky, available to the exploitation service (i.e. Subscriber) from another cloud in the same sky.

Additionally, the Sky's global directory service is responsible to push all the aggregated updates (from all the clouds) back to all the clouds (which will have duplicate elimination process in it) present in the Sky. Thus, an exploitation service registered with one of the clouds in the sky would have information to subscribe to the PSSA sensor events that are being published in another instance of the cloud in the same sky.

Further, from implementation perspective the PSSA SDK provides API calls to build the low-level communication modules (like Publisher, Subscriber, Directory Service Observer and the service provider) which are independent of each other in any given PSSA component (a PSSA-based application). Hence, it is also possible to build complex Multi-Cloud PSSA components that can be part of more than one PSSA cloud at any given time, like as shown in Figure 18.

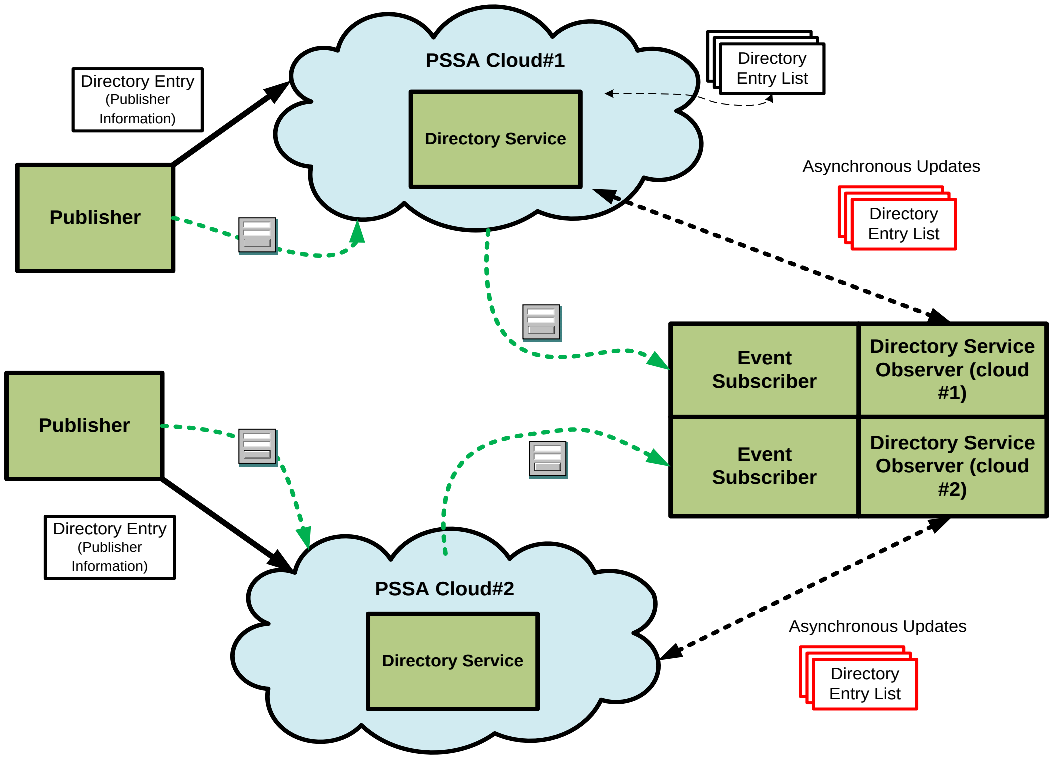


Figure 18: Multi-Cloud Connectivity

By leveraging the flexibility of the PSSA and using current PSSA SDK, it is possible to build complex Multi-Cloud PSSA components that can be part of more than one cloud.

4.2 Interoperability

As mentioned in the Introductory Section, the goal of the PSSA is not to replace or supplant existing sensor processing systems. Rather, it is to provide a platform for enhancing the integration, exploitation, storage and visualization of sensor data from a variety of different sources. The PSSA "plug and play" approach supports the development of ingestion services that can receive sensor data from other sensor processing systems and make it available for storage and/or exploitation within our high throughput computing environment. In addition, dissemination services can be developed that implement the interfaces required to provide data to other systems and external gateway data services can provide other PSSA components with interfaces to store and retrieve data on external systems. The following subsections provide examples of PSSA-based systems have integrated or could integrate with other systems.

4.2.1. Open Layered Sensing Testbed (OpenLST)

The OpenLST is a test environment developed at the Tec^Edge Discovery Lab in conjunction with AFRL and several industry and academic partners. The OpenLST was originally designed to display GIS vector and raster data that was pre-processed and stored on the local hard-disk or a network connected GIS server. Prior to integration with PSSA, the OpenLST platform did not have the ability to store or play back live sensor data. It also did not have the ability to apply algorithms to exploit the sensor data streams as the data was received. This limited its effectiveness as a platform for future research, experimentation and demonstration.

PSSA provides the ability to store and playback live vector and raster data as it is being collected as well as the ability to “plug” in exploitation components as needed to meet application specific needs. Integrating the OpenLST with PSSA allows OpenLST to take advantage of these abilities and to expand its reach as an experimentation, research and demonstration platform. As an integral part of the OpenLST platform, PSSA provides the ability for researchers to quickly and easily add new sensors, exploitation algorithms and visualization layers to the platform.

In addition, multiple OpenLST platforms on the same network are able to simultaneously access the same live data and exploitation results. As another benefit of integration, the PSS messaging system provides collaboration capabilities that allow users of OpenLST to share text communication and sensor feeds with one another and with third party chat users.

4.2.2. MATLAB Integration

MATLAB® is a tool used by many scientists and engineers to develop algorithms for processing sensor data. PSSA provides software components to allow exploitation algorithms written in MATLAB to be easily integrated into a sensor processing flow. This makes PSSA an ideal platform for testing exploitation algorithms with a variety of different sensor data sources both real-time and recorded. Examples of MATLAB components built to work with the PSSA reference implementation are included with the PSSA SDK.

4.2.3. Sensor Web Enablement (SWE)

The Open Geospatial Consortium (OGC) has published a set of standards and best practices known as SWE. SWE compliant systems are currently in use or being developed for military, security, and environmental monitoring purposes by organizations such as NASA, National Oceanic and. Atmospheric Administration (NOAA), Consortium of Universities for the Advancement of Hydrologic Sciences (CUAHSI), and the European Space Agency.

A PSSA system can act as an SWE client to collect and store sensor data generated by an SWE compliant system. In this role, the SWE compliant system is treated as just another sensor type. A PSSA ingestion component is developed to interact with the SWE compliant system using the Sensor Observation Service (SOS) interfaces for synchronous data requests or a combination of Sensor Planning Service (SPS) interfaces and Web Notification Service (WNS) interfaces for asynchronous data requests.

In addition, if a SWE compliant system supports the generation of asynchronous alerts, an ingestion component can be developed to receive these alerts using the SWE Sensor Alert Service (SAS) interfaces to establish connection to the system(s) generating the alerts. The alerts become just another type of asynchronous sensor data received and processed by the system.

Similarly, PSSA retrieval components can be developed that implement SWE compliant SOS interfaces, SPS interfaces and/or WNS interfaces to allow the data ingested by the PSSA system to be transmitted to an SWE client application before or after it is processed by the PSSA exploitation services. By implementing OGC SWE standards, a PSSA-based system can become an integral part of any existing or future SWE compliant system.

4.2.4. Distributed Common Ground System (DCGS)

DCGS is a program sponsored by the Department of Defense (DoD) to integrate Intelligence, Surveillance and Reconnaissance (ISR) capabilities so that the information collected and processed by these systems can be shared across all of the armed services. Integration to DCGS is accomplished via the DCGS Integration Backbone (DIB).

The PSSA can support integration with DCGS through a purpose-built DCGS gateway that implements the DIB 1.3 metadata framework and conforms to version 3.0 of the DCGS Discovery Metadata Service. Using this gateway, PSS exploitation components can retrieve data from a DCGS and data stored within the PSS system can be retrieved for use by DCGS. The DCGS gateway component will be responsible for encapsulating all interactions between the PSS components and the DCGS system including security, protocols and interfaces.

4.2.5. Cursor on Target (CoT)

CoT is a DoD standard for interchanging temporal and spatial data between sensor/weapon systems. The PSSS can support interaction with CoT systems by providing purpose-built ingestion and retrieval components that are CoT compliant. This allows data to be interchanged between a PSS system and any other system that generates or accepts CoT messages.

4.2.6. Qbase Data Transformer (QDT)™

The QDT is a powerful and flexible tool for data analytics, transformation, and workflow management. A PSSA message bus source and destination component can be developed for QDT™ that will allow any data published to the PSSA message bus to become a source data stream for QDT processing and any data created by a QDT job to be published to the PSSA message bus. Developing a PSSA message bus source and destination for QDT will allow a QDT job to be created that can function as a PSSA ingestion component, application service, and/or dissemination component. This capability could be used to provide non-programmers with the capability to build complex exploitation solutions by graphically connecting QDT data transformation and processing modules in a defined workflow and inserting the resultant QDT jobs into the PSSA processing environment.

4.3 Usage Scenarios - Environmental Monitoring

4.3.1. Moisture Flux Assessment

Background**:** The total extent of irrigated acreage in the US is 56.6 million acres shared by about 301,000 farms. Of this irrigated area, an extent of 38.5 million acres is in the arid Western US. The drought situation has exacerbated the concerns about water allocation and accounting of the different sectors of water use. The larger water management concerns about efficiently irrigating and about meeting the overall pressure on water demands owing to expansion of the cities, especially under the circumstances of water shortages, logically have fallen into properly estimating water requirements due to moisture flux or evapotranspiration (ET) in the irrigated areas.

The operation managers from the United States Bureau of Reclamation (USBR) were seeking to apply advanced newer technologies for the estimation of ET to manage the water systems better. The use of remote sensing techniques offered a pathway to answering questions related to quickly estimating ET over large areas. In 2003, the USBR entered into a strategic alliance with the

International Center for Water Resources Management at Central State University in Ohio, the Ohio View Consortium, and Colorado State University (Alliance Universities (AU)) for the development of advanced remote sensing technologies for use in operational decisions to deal with future constraining events.

This collaborative effort is a multi-year collection of ground based daily ET values for riparian vegetation in the Lower Colorado Region (LCR) at locations with different densities of species. The synchronous collection of data on parameters needed for remotely sensing ET at different altitudes and on the ground was achieved during two intense phases of data collection in the summers of 2007 and 2008. Sensor measurements were taken on the ground, using low altitude flight (helicopter) and medium altitude flight (aircraft) around the time of passage of Land Remote Sensing Satellite System (LANDSAT) over the pilot study area.

Scenario: The USBR and AU, with assistance from US Geological Survey (USGS) and the US Department of Agriculture (USDA), embarked on a systematic plan to conduct research on estimating ET in the agricultural and riparian areas using remote sensing. The research plans included ground based, state of the art instrumentation for measuring ET in the field. The emphasis was placed on measuring ET in the riparian vegetation corridors wherein, the invasion of undesirable species which apparently consume precious water occurs. The LCR between Parker and Imperial Dams was chosen as the main pilot study area for riparian vegetation, see Figure 19. This included the Palo Verde Irrigation District (PVID) and Cibola Wild Life Refuge (CWLR). A secondary group headed by Colorado State University chose pilot study areas in the Arkansas River and South Platte River basins in the Great Plains (GP) Region.

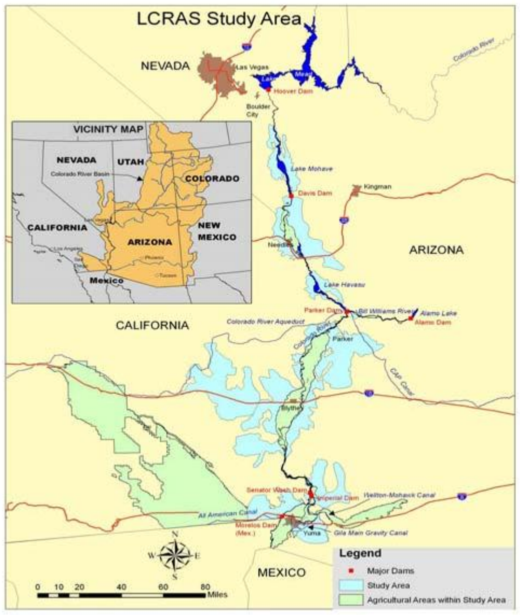


Figure 19: Region of Study

Example: To leverage the PSSA, ingestion components are being developed to ingest, normalize and store data from the aerial and ground based remote sensors from this project.

Inputs from the following sensors are stored within the PSSA:

- Bowen Ratio Towers
- Scintillometer Sensors
- Land Remote Sensing Satellite System (LANDSAT) Thematic Mapper
- Evaporation data from Lower Colorado River Accounting System (LCRAS)
- Meteorological data from California Irrigation Management Information System (CIMIS)
- Multiple derived imagery products, such as derived vegetation indices, Normalized Deviation Vegetation Index (NDVI), Enhanced Vegetation Index (EVI), etc.
- Spectroadiometric measurements from low altitude flights
- Thermal sensing from medium altitude flights

This information is spatially and temporally indexed to facilitate comparative analyses by the AU group. This architecture allows the AU to perform historical analysis of the data in addition to allowing updated forecasts to be made as new data becomes available.

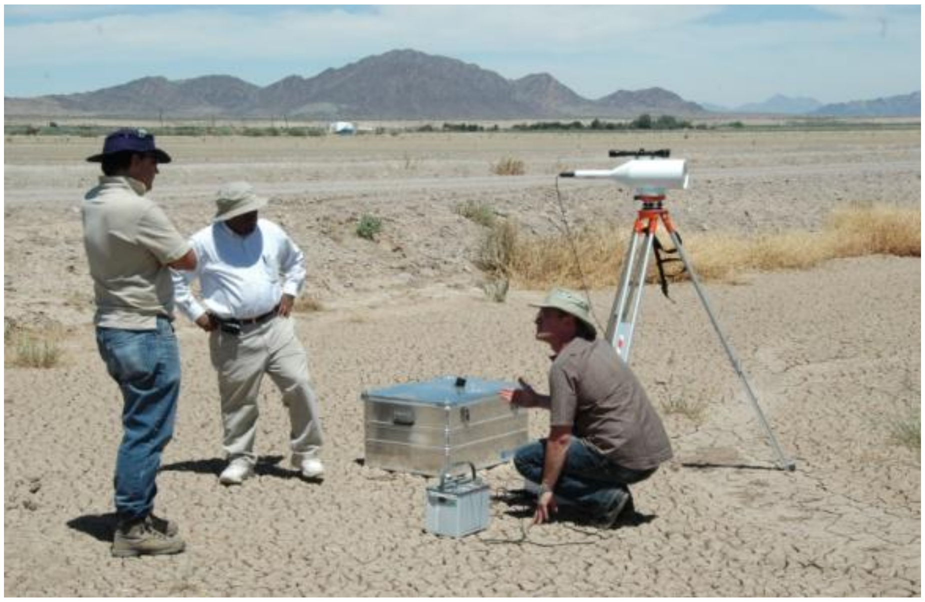


Figure 20: Scintillometer Sensors

This capability would enable USBR to perform automated blind testing of different methodologies for estimating ET using remote sensing. As new methodologies become available, the platform will allow a rapid assessment of their validity. This enhancement will facilitate the work being performed by a panel from the team to evaluate the best existing methodology for the remote sensing of ET using the ground based data. This unique feature would enable the USBR to adopt the winning method for its use and quickly transition this methodology to practice.

The remotely sensed data requires various processing tasks to be performed to provide meaningful results. Some of the algorithms that will be implemented within this framework are:

- Remote Sensing EvapoTranspiration (ReSET) model – AU-developed model allows the generation of daily and seasonal ET estimates using satellite imagery (LANDSAT 5 and 7)
- Statistical models of ET from the PVID using CIMIS data and Moderate Resolution Imaging Spectro-Radiometer (MODIS) EVI
- Statistical models of ET from the Bowen Ratio Towers
- Correlation between water use and bio-mass through an allometric equation to calculate biomass
- Application of the Surface Energy Balance Algorithm for Land (SEBAL) and Simplified-Surface Energy Balance Index (S-SEBI) energy balance models for the pilot study area.
- Processing of the scintillometer data using Monin-Obukov correction

The implementation of these algorithms within the PSSA would facilitate the researchers in performing more studies and experiments to validate their models. The outcome of these algorithms allows for the prediction of ET and the identification of invasive species, such as Tamarix. Multiple passes from the LANDSAT satellite can be more easily combined to monitor the seasonal progression of Tamarix growth. In addition, the LANDSAT imagery is used for the classification of ten land-cover categories; river, unvegetated land, fields, salt cedar, mesquite, cottonwood and willow, arrowweed, atriplex, grass and marsh grass, creosote bush.

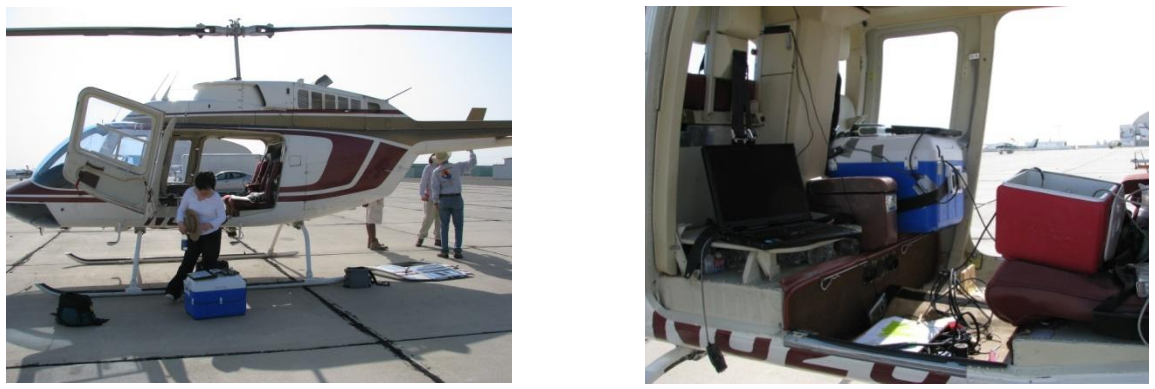


Figure 21: Spectroadiometric Measurements Using Low Altitude Flight

Figure 22: Thermal Sensing Using Medium Altitude Flight

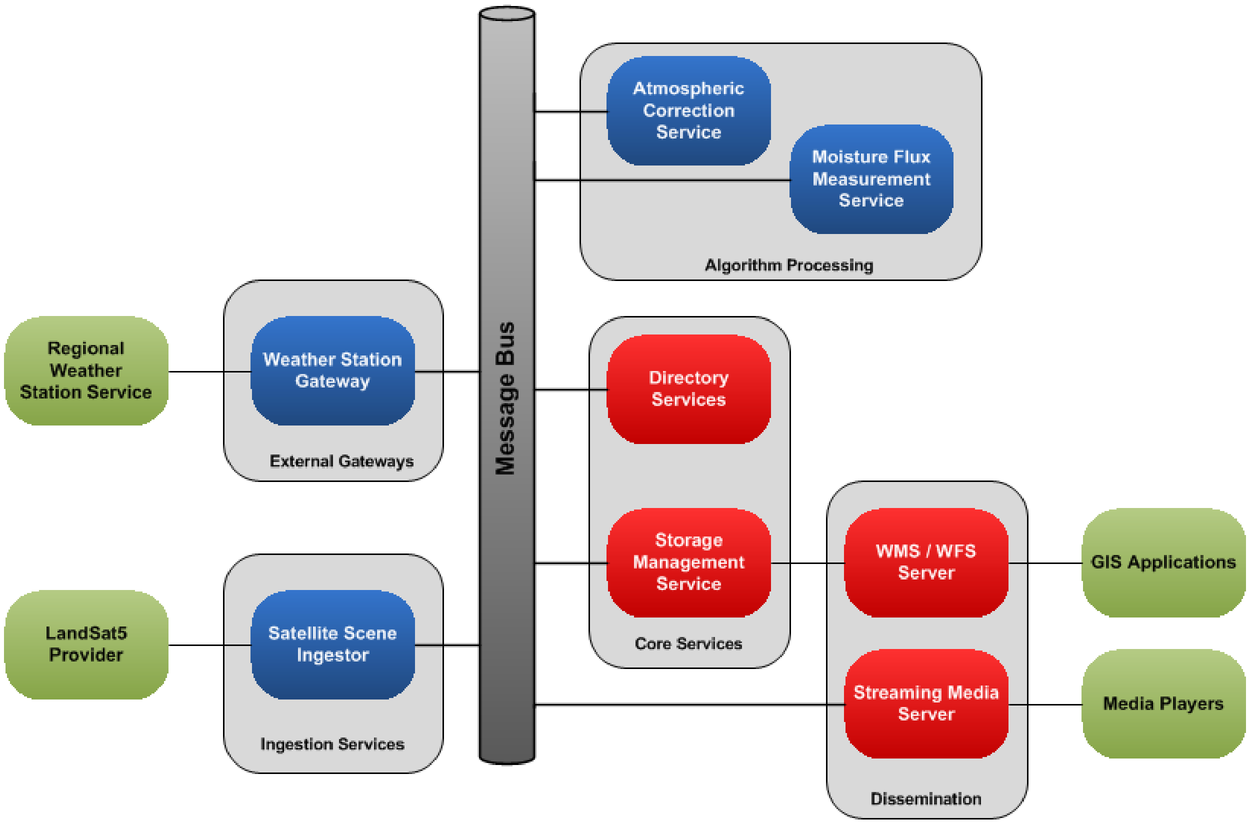


Figure 23: Logical Overview of the Water Management Scenario

4.3.2. Environmental Disaster (Oil Spills) Background

Oil spills in the marine environment may have a wide spread impact and long-term consequences on wildlife, fisheries, habitats, human health and livelihood, as well as recreational resources of coastal communities. The degree of damage caused by these spills depends heavily upon the amount of oil spilled, the properties of the oil and the sensitivity of the resources impacted.

The effects upon surface and subsurface marine life after a spill may be due smothering and physical contamination of the chemical characteristics of the oil. The longevity of the oil is another key factor of its impact on the marine environment.

The impact upon the coastal environment depends upon the composition of the coastline. For example, mangroves and salt marshes are more sensitive to oil, whereas rocky coastlines are less sensitive. Spills impact coastal and marine environments in many forms, some which are:

- Mortality of birds and other marine life
- Damage and loss of marine habitats
- Viability and diversity of coastal ecosystems
- Damage to fisheries and tainting of commercial seafood
- Damage to inter-tidal and inter-coastal flora and fauna
- Contamination of coastal recreational resources
- Disruption of power station cooling water or desalination operations

  In general, marine oil spills may be classified into three categories:

  - **Port** – These generally occur while ships are in port and may occur during bunkering or berthing. They usually involve the discharge of oil from the bilges and usually are the result of accidents or poor practices.
  - **Incidents** – These are usually the result of shipping incidents; such as fires, collisions, groundings, etc. Other causes may be drilling platforms or pipelines which are damaged due to adverse weather or accidents.
  - **Intentional** – These are the result of those who intentionally dump waste oil or oily waters into the environment to save time or money.

Scenario: To effectively respond to an oil spill in the marine environment, responders need to know the answer to the following questions.

- "Where is the spill going?"
- "How fast will it get there?"

To answer these questions, satellite, aerial, surface and sub-surface sensor information must be analyzed and fused to create a holistic operating picture. Federal, State and Local agencies need the ability to have a common operating picture to coordinate their response activities. These agencies have different capabilities and expertise and ideally would provide a coordinated response.

Example: Leveraging the PSSA, ingestion components would be developed to ingest, normalize and store data from some of the sources that would be available.

- Oil spill trajectory forecasts, National Weather Service, over flight information, tidal amplitudes, etc.
- Tracking of sea surface oil – TerraSAR-X, MODIS and RadarSat-2 data
- Floaters – Provide real-time environmental data and oil slick observations
- Area base maps and vector data such as; slick expansion, boom placement and sensitive habitats
- Insight from local experts
- Onsite reports; fishermen, boaters, etc.

The information from all of these sources would be stored in the PSS system and spatially and temporally indexed. The PSSA would not only allow historical analysis, but the exploitation of real-time data to make better predictions by fusing disparate data sources. An example of this may be seen in Figure 24 to Figure 28. These projections were created using only over flight information and NOAA forecast models. The accuracy of these forecasts may have been improved by leveraging additional sensor information that would be available with a system based upon PSS.

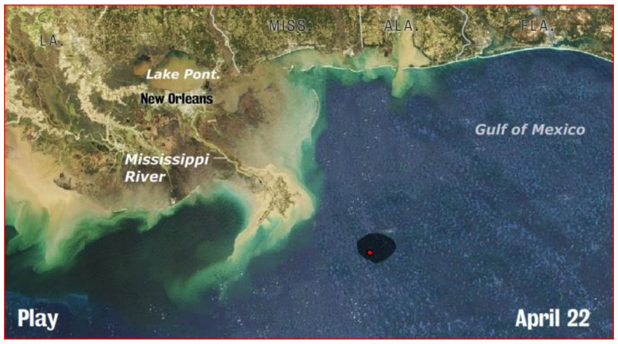


Figure 24: Estimated Oil Boundary - 22 April

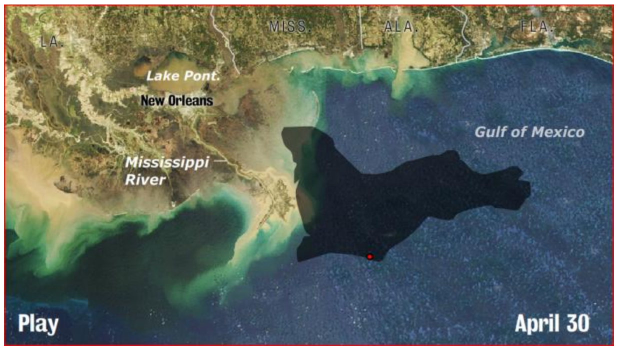


Figure 25: Estimated Oil Boundary - 30 April

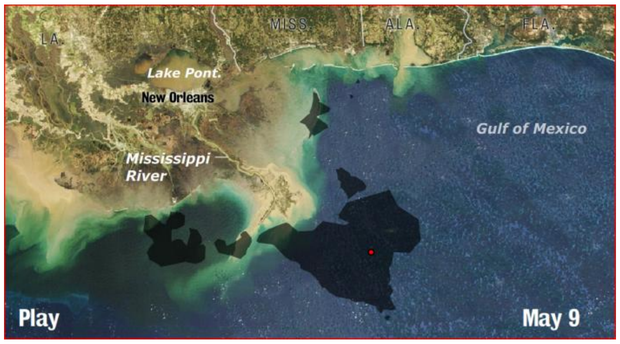


Figure 26: Estimated Oil Boundary - 9 May

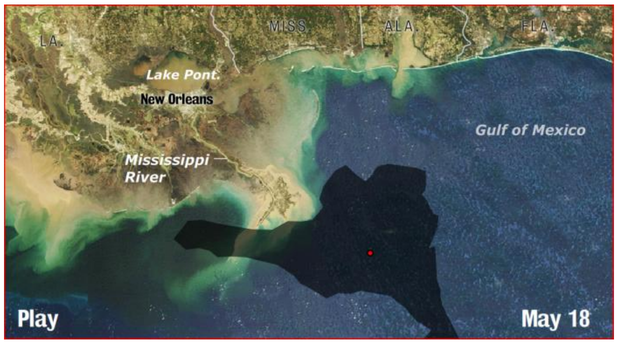


Figure 27: Estimated Oil Boundary - 18 May

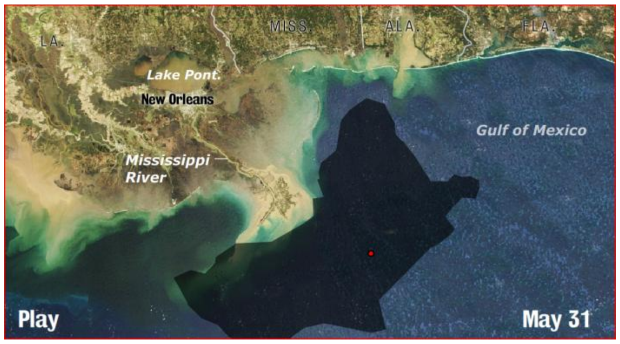


Figure 28: Estimated Oil Boundary 31 May

Figure 29 shows components that might be part of a solution for monitoring an oil spill. The ingestion and gateway components are responsible for acquiring and normalizing data and then placing it on the PSS message bus for application services to exploit.

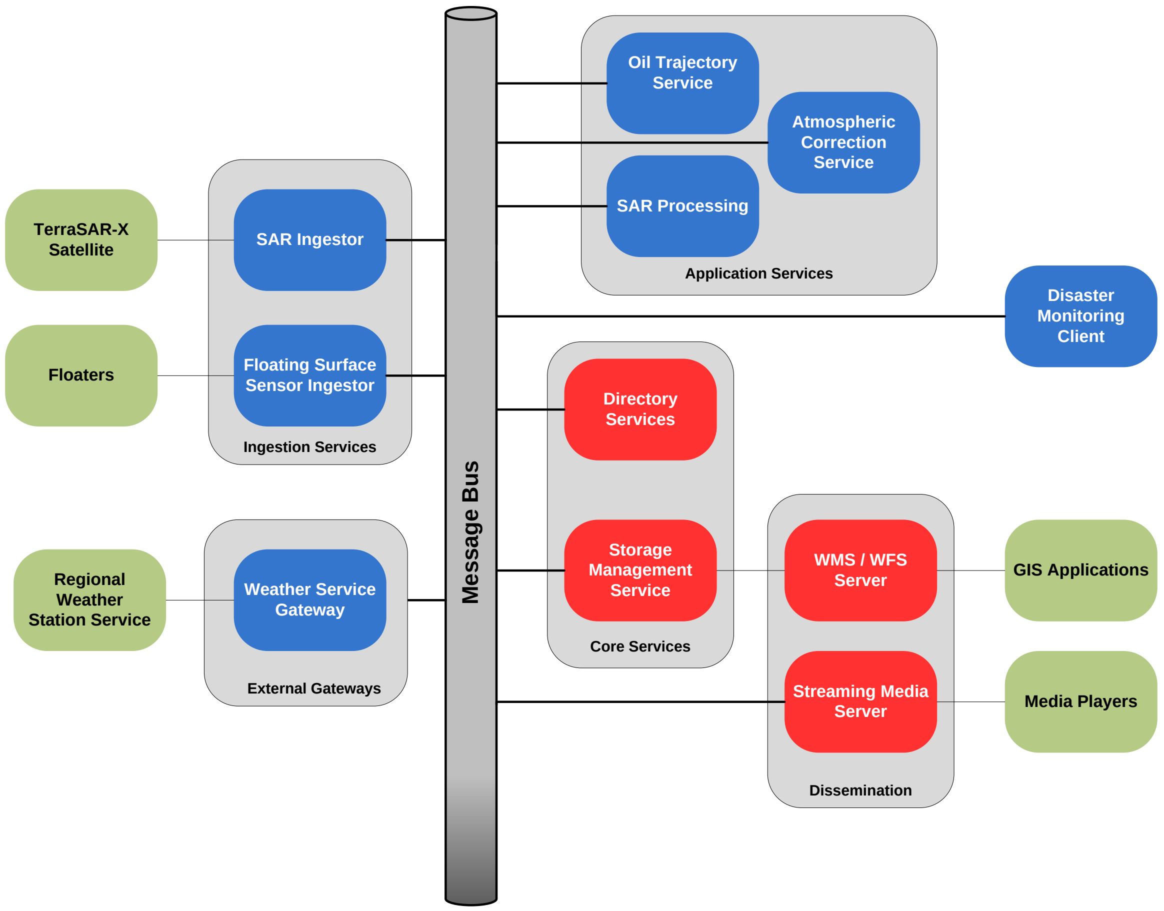


Figure 29: Logical Overview of the Environmental Scenario

In this scenario, Satellite Synthetic Aperture Radar (SAR) data is being received every eleven days. The data could already be preprocessed into usable imagery or this could be raw data that would need to be further processed by other services. In this case, the SAR data and atmospheric correction data would be pulled off of the bus by the SAR processing service. The resultant processed SAR data would then be placed back onto the bus for storage, indexing, retrieval and further exploitation.

As processed SAR scenes become available, the oil trajectory service would process these and produce additional scenes that highlight possible areas where oil may be present and estimated trajectories.

All of this data would be available to existing products through standard interfaces such as WMS/WFS. In addition, custom clients can be created to provide enhanced functionality that eases the integration and analysis of these large and disparate data sets.

Central Statue University is working on algorithms to detect the presence of surface oil and sub-surface plumes. To accomplish this, hyperspectral images from LANDSAT 5 are being ingested and analyzed from the Gulf of Mexico oil spill.

Figure 30 shows the results of the detection algorithm on a control image taken from the USGS on 7 April 2010 prior to the oil spill. It shows no oil present in the water (clouds highlighted in blue for reference).

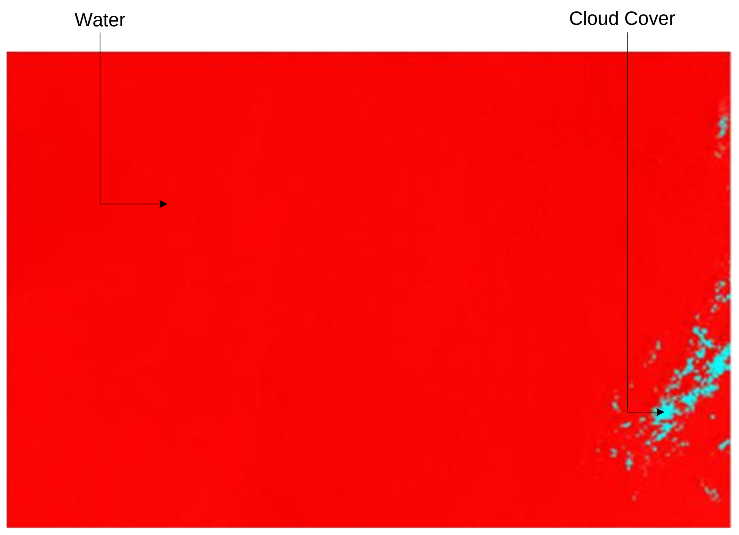


Figure 30: Reference Gulf of Mexico Image Prior to Oil Spill

Figure 31 (next) shows the results of applying the same detection algorithm to a LANDSAT 5 image taken 25 May 2010. Surface oil is highlighted in blue and sub-surface oil is represented by an offset in red region. The veracity of this technique needs to be verified by samples taken in the Gulf of Mexico. Central State is working to acquire additional data to support this work, such as dip data from various depths and locations within the Gulf. These results would allow policy makers to more easily answer the two key questions:

- “Where is the spill going?”
- “How fast will it get there?”

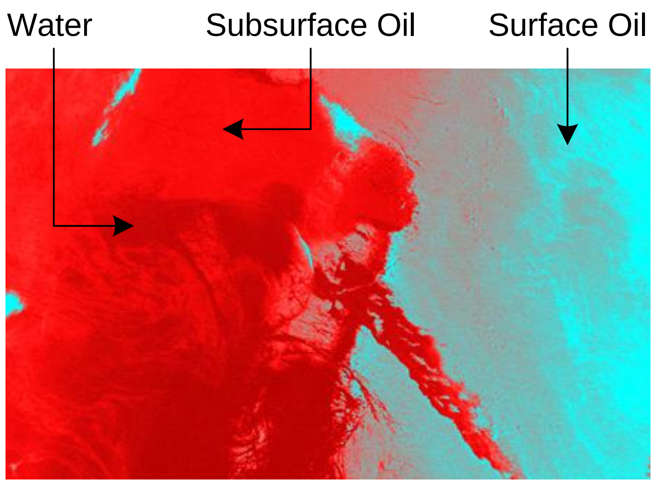


Figure 31: Oil Detection with Hyperspectral Imaging

4.4 Usage Scenarios - Intelligence, Surveillance and Reconnaissance

4.4.1. Homeland Security

Problem Description: Protecting the health and well-being of war-fighters from chemical and biological hazards is a key concern of the United States Armed Services that mirrors the ODA mission to protect Ohio's food supply (livestock and poultry) from similar threats both intentional and unintentional. As a result, ODA partnered with AFRL to sponsor a project at Tec^Edge during the summer of 2010 to implement a system for detecting, tracking and responding to a simulated outbreak of bovine hoof and mouth disease among Ohio cattle farms.

Scenario: COWPATH was based on the hypothetical scenario that, after the initial outbreak of an epidemic, a predictive model (seeded by initial outbreak information) is deployed at a command center to provide estimation in terms of time and geospatial spread of the disease in real-time. Further, this predicted spread of the disease is validated against real-time observations from lab veterinarians testing samples and field veterinarians present in the predicted infected areas diagnosing the disease symptoms.

The flexible design and simple integration model provided by the PSSA framework allows it to be easily adapted to support a variety of defense and homeland security related problems. A group of Qbase engineers and AFRL sponsored student interns at Tec^Edge during the summer of 2010 utilized the PSSA framework to implement a system for detecting, tracking and responding to a simulated outbreak of bovine hoof and mouth disease among Ohio cattle farms. This project utilized the PSSA framework to connect real-time image and text data feeds and disease data generated by a simulation program to a command and control center. The command and control center utilized the PSSA enabled OpenLST platform to provide a geospatial view of the data as it was being received in real-time. PSSA plug-ins, added to the OpenLST platform, provided the ability to view data from various sensor feeds including smartphones and the disease simulator. It also provided the means to collaborate with smartphone users in real-time via instant messaging protocols.

Utilizing the PSSA SDK, the team developed several components to support this demonstration. These included an ingestion component to receive event data from the simulation program, an ingestion component to receive alerts from smartphone equipped field veterinarians, and a PSSA-enabled OpenLST platform for dissemination and visualization of the data collected by the system. In addition, several layers of GIS data were also loaded into the system to provide relevant geographic information such as farm locations, airport locations, soil types, and road/river networks.

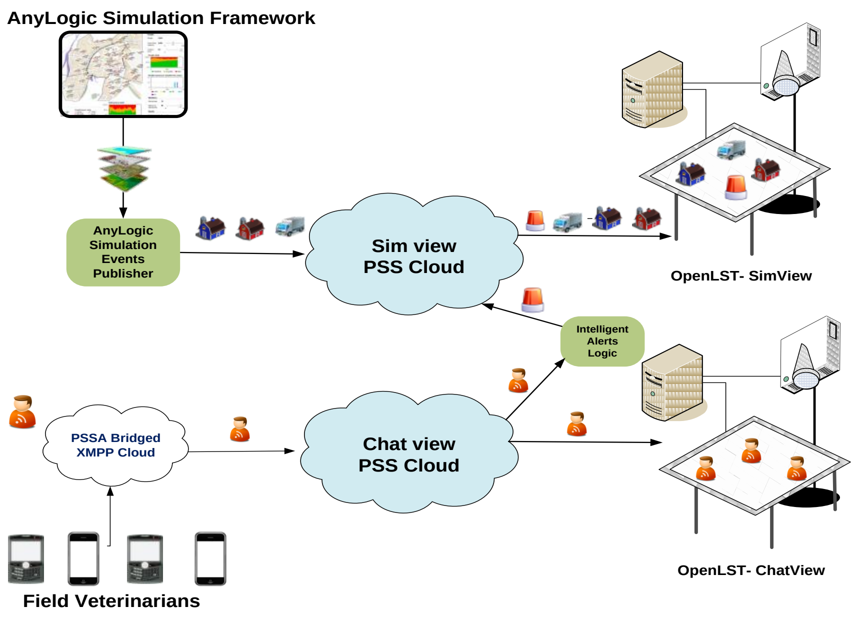


Figure 32: Integrated View of PSSA-Based COWPATH Demo

The resulting demonstration system, shown in Figure 32, was successfully developed and demonstrated over an approximate eight week period using primarily undergraduate student resources to perform the programming work using the PSSA framework as the basis for integrating the components.

4.4.2. Premises Security (Video Surveillance)

Problem Description: Many large security-conscious organizations utilize surveillance cameras, security doors (with keycard, pin-pad, or biometric access), motion detectors, and sometimes other sensors to monitor and control access to various parts of their campus and/or facilities. In many cases, these are independent systems with limited ability to correlate or synchronize the data collected across the various sensors. In addition, it may not be possible for limited security staff to continuously monitor all of the surveillance video feeds. Oftentimes, if an incident occurs (theft, assault, etc.), forensic analysis of hours of video from several surveillance cameras is required to determine what happened.

Scenario: In order to provide security for its service members and employees, a large military base deploys hundreds of video surveillance cameras with pan/tilt/zoom capability to monitor buildings, streets, and walkways. Some buildings (e.g. laboratories) require keycard access while others are more accessible. All building entrances are in the field of view of one or more surveillance cameras. Both the more public and secured entrances utilize electronic locks that generate an alarm if a door is forced open. Interior motion detectors unlock doors from the inside to permit people to leave the building when a door is locked.

Utilizing the PSSA, ingestion components can be developed to ingest and store data from surveillance cameras, keycard access control systems, and motion detectors. The data captured from these devices is time synchronized and stored in the PSS database. This capability significantly improves forensic analysis by allowing analysts to easily identify segments of video corresponding to periods of time when door lock alarms and/or motion detectors were triggered.

Even more important, however, is the ability to exploit the real-time sensor data to identify anomalous activity as it occurs using the PSS application services to implement the exploitation algorithms. Algorithms can be developed and plugged into the PSSA to identify anomalous signatures such as activity occurring in a time and/or place it shouldn't be occurring or a secure door being opened without a keycard. This information can be used proactively to alert security personnel as well as to automatically increase the sensitivity and adjust the orientation of nearby sensors to provide more detailed situational information.

Example: A PSS-ingestion component receives a signal from a secure door whenever it is opened or closed. Metadata associated with this signal indicates where the door is located, whether the door was opened or closed, and what time the door was opened/closed. If the door was opened, additional metadata could include whether an interior motion detector or an exterior keycard device unlocked it or if it was locked and forced open. If a keycard was used to unlock it, the unique ID of the keycard is included in the metadata. If the door was closed, additional metadata includes the length of time the door was open. This metadata could come from the security system that controls the door or, if the information is not available from the security system, it could be generated by the ingestion component. Either way, the ingestion component does not perform any further interpretation of this data; it merely puts the data on the PSS message bus and lets a downstream application service component exploit it.

Another PSS-ingestion component receives the video generated by a surveillance camera pointed at a secure door. Metadata associated with the video includes where the camera is located, the time interval in which the video frame was captured, the camera pan, tilt, and zoom settings, etc. This ingestion component does not interpret the data; it just puts the video data and associated metadata on the PSS message bus for a downstream application service component to process.

A PSS application service component analyzes the video stream to look for moving objects. If moving objects are detected, the exploitation algorithm generates a track message and puts it on the PSS message bus. The track message includes the location, direction and velocity of the motion.

Another PSS application service component implements a rule-based algorithm to determine whether or not the opening of the door represents an anomaly and, if so, generates an alert message. For instance, an alert might be generated if any of the following rules are true:

- The door was forced open (i.e. neither the interior motion detector or the external keycard reader unlocked the door before it was opened)
- The door was opened at an unusual time. The definition of an unusual time could be based on a statistical model that represents the normal usage of that door. (e.g. In the

past, the door was never been opened between the hours of 1 am and 5 am, but in this particular instance it was opened at 3 am).

- The door was opened at an unusual time by a person not having designated access and perhaps demonstrating anomalous entry and exit behaviors. Again, statistical norms could be used to determine what constitutes normal usage of that door.

Statistical data used by rules 2 and 3, above, is derived from door sensor data previously captured and stored by the PSS system.

A third PSS application service component receives the door sensor message and looks for track messages from video sensors whose field of view includes the doorway. If a track is identified that indicates additional anomalous activity (e.g. someone entering building when an egress is expected), an alert message is generated and placed on the PSS message bus.

PSS storage components subscribe to the messages described above and, when received, store the data and metadata contained within these messages in the PSS database for later retrieval.

A visualization client application could be developed that uses the PSS SDK to subscribe to alert messages and request live/recorded video streams associated with the alert from the PSS streaming media service. Alternatively, off-the-shelf GIS tools could be used to poll the PSS WFS service for alerts and, once received, an off the shelf streaming video player could be used to retrieve a live video stream or request a recorded video stream containing the data captured just prior to, during, and after the alert was generated.

The diagram below illustrates the components that might be part of a solution for this scenario:

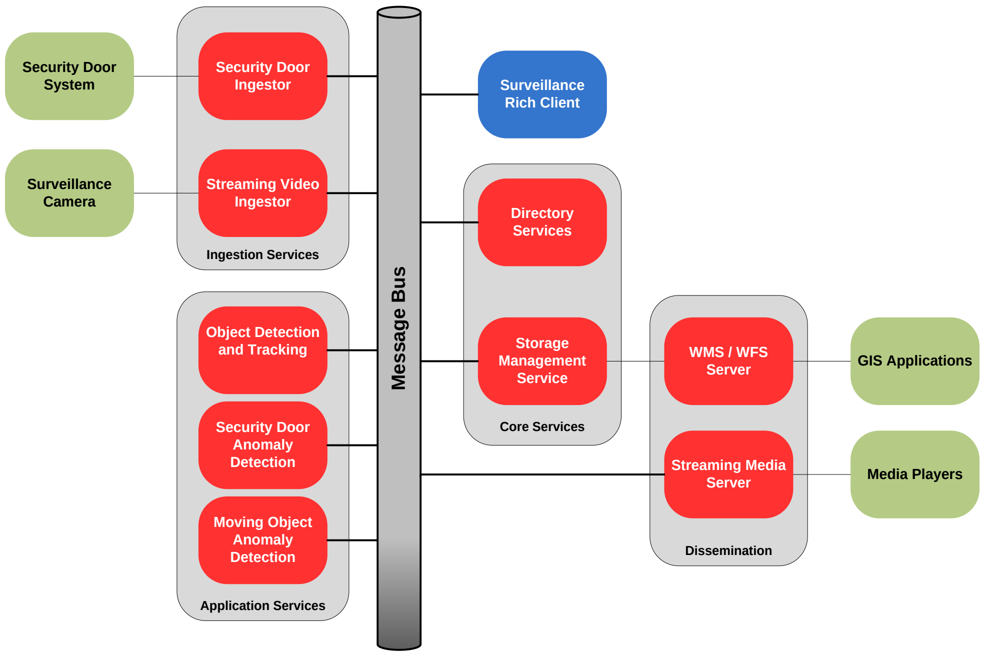


Figure 33: Application of PSSA to Premises Security

4.4.3. Pinpoint – Audio Localization Using Smartphones

Problem Description: As smartphones with audio capture capabilities become more common, it is desirable to use these devices to monitor background noise levels to detect anomalous noise events (such as gunfire or explosions). If enough of these devices "hear" the suspicious noise, then using triangulation techniques, it should be possible to identify the source location of the anomalous sound(s).

Scenario: During a period of potential terrorist activity, concerned citizens and/or law enforcement can activate the pinpoint application on their smartphone devices. When an event occurs that generates a loud enough noise, each phone that "hears" the noise can generate an alert notification that includes a recording of the noise event and the time and GPS location of the smartphone at the time the noise was "heard." If enough (minimum of four) smartphones detect the noise event, the data from the phones can be used to determine an approximate location whence the noise occurred.

During SATE 2011, an intern program sponsored by AFRL at the Wright Brothers Institute - Innovation and Collaboration Center (WBI-ICC) Tec^Edge Discovery Lab, a group of students developed a prototype sound localization solution using Android smartphones with the ability to capture sound events and transmit them over a WiFi or Cellular 3G/4G network to a processing platform that could utilize this data to determine the location of the sound. Applying PSSA as a solution architecture for this problem required the development of the following components:

1. Smartphone application to monitor for abnormal sound levels and when detected send this information along with relevant metadata (time/location) via a TCP/IP connection to a PSSA ingestion component.
2. A PSSA Pinpoint ingestion component that listens on a well-known TCP/IP socket for a sound event from a Smartphone. This component packages the sound packet with its associated metadata and publishes it as a PSS sound event.
3. A PSSA audio location registration component subscribes to the sound events and uses the metadata (time and location) associated with the sound events to determine if they correspond to the same event, and, if so, whether there is enough data to determine the location of the sound event. If there is sufficient data, an estimated location and time of occurrence is calculated and published to the PSS cloud along with the original sound recordings.
4. A PSSA Sound Identification exploitation component subscribes to the output of the audio location component in an attempt to classify the sound based on the audio (frequency/amplitude) waveforms of the corresponding sound recordings. This analysis could be used to distinguish between a gunshot, a car backfiring, an explosion, etc. If the type of sound can be determined, then it is published to the PSS cloud.
5. OpenLST could be used as the visualization platform showing the estimated location and time of the noise event along with the location of the smartphone devices used to locate the noise.

The following diagram illustrates the components utilized in this scenario:

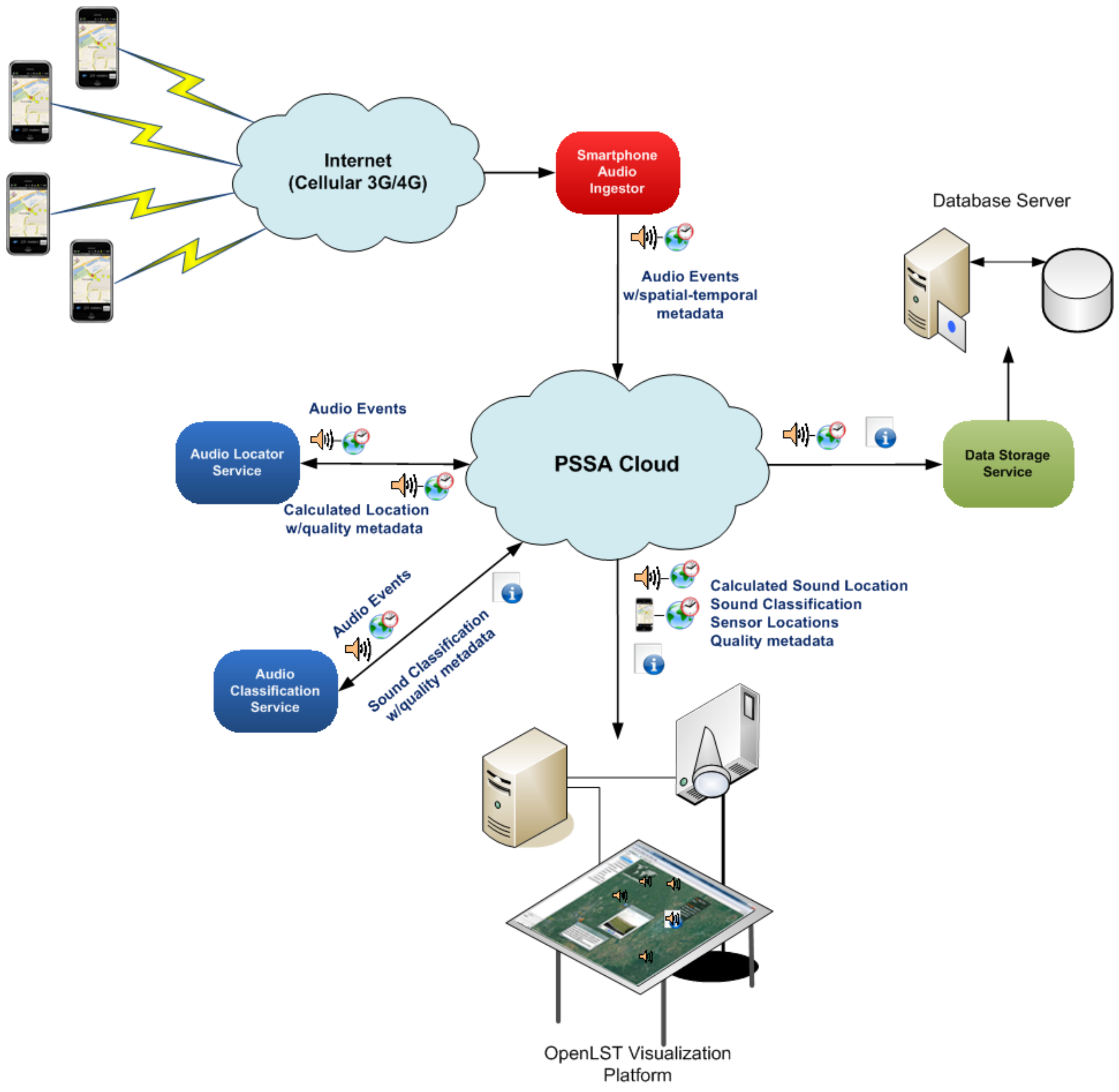

Internet
(Cellular 3G/4G)
Smartphone Audio Ingestor
Audio Events w/spatial-temporal metadata
Database Server
Audio Events
Audio Locator Service
Calculated Location w/quality metadata
PSSA Cloud
Data Storage Service
Audio Events
Sound Classification w/quality metadata
Audio Classification Service
Calculated Sound Location
Sound Classification
Sensor Locations
Quality metadata
OpenLST Visualization Platform

Figure 34: PSSA Pinpoint Solution

4.4.4. iStream – Using A Smartphone as a Real-Time Video Sensor

Problem Description: With the widespread deployment of smartphones with video recording capabilities, it is desirable to use these devices as "on the spot" video sensors when necessary to support law-enforcement, anti-terrorist and/or search and rescue activities. Working together, these "on the spot" video sensors can be used by command center personnel to get a much better perspective of ground conditions and improve overall situational awareness.

Scenario: During a natural disaster emergency response personnel equipped with smartphones can shoot video to provide command center personnel with a ground-level view of the situation. The

GPS positioning capabilities of the smartphone can be used to geo-locate the source of the video footage and tag that location on a satellite image of the affected area.

If several smartphones are actively shooting video within a given geographical area, command center personnel can choose which video streams to view and also select archived video streams from the same location to gain a better awareness of the situation.
During SATE Summer at the Edge 2011, an intern program sponsored by AFRL at the WBI-ICC Tec^Edge Discovery Lab, a group of students developed a prototype video streaming capability using Apple iPhones as the video capture device. This solution was capable of delivering 10 frames per second (fps) video from the iPhone to a server where the streaming video could be viewed then be viewed.

The iPhone application utilized built-in functions and third3rd party libraries to capture the video, decode it and post it to a server using an HTTP post. To apply this solution to PSSA required the development of a custom video stream ingestion component that implemented the HTTP post functionality to receive video frames from the iPhone and publish them as MPEG-4 frames to the PSSA cloud. This allows PSSA components already developed to process MPEG-4 frames to store, exploit, and display the streaming video data.

Previously, we developed PSSA modules to perform moving object detection and to measure video quality attributes. By virtue of the “plug and play” nature of the PSSA framework, these modules can be used to process the video captured by the smartphones, as well. The diagram, below, illustrates the PSSA components that could be utilized to enable this capability.

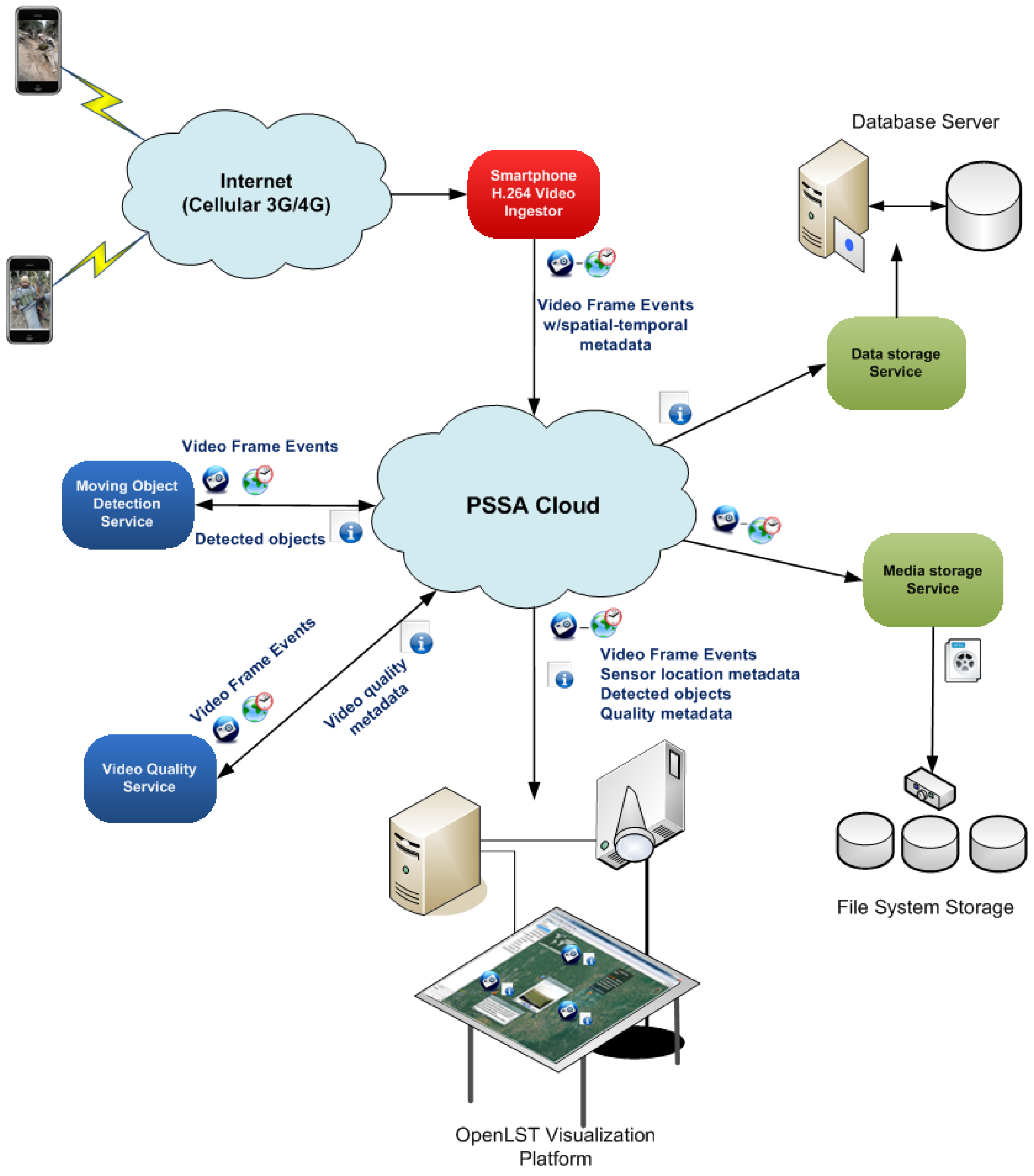


Figure 35: PSSA iStream Solution

5.0 CONCLUSIONS AND RECOMMENDATIONS

The first phase of the Persistent Surveillance Data Processing, Storage and Retrieval project focused upon the research and prototyping of core technologies required to process, store and retrieve high volumes of sensor data being delivered asynchronously at varying data rates. During our research, we determined that the most pressing and technologically complex need was the need to handle simultaneous streaming video data from air and ground based optical sensors.

As a result, we focused much of our prototyping effort for phase one in building components to make sure that the architecture met those needs. Through various prototypes and research efforts, we designed and evolved the PSSA to address many of the issues associated with processing, storing and retrieving sensor data.

Although the existing PSSA prototype system lacks many of the desired capabilities described in this document, it has been successfully used to support a number of demonstrations both at Qbase and in conjunction with AFRL. Through these demonstrations, the versatility of this architecture and the simplicity by which it can be applied to solve a variety of sensor related integration problems have been proven. During the summer and fall of 2010, a number of undergraduate student teams sponsored by the AFRL Discovery Lab at Tec^Edge have used the PSSA framework and SDK to quickly develop solutions that integrate sensors with storage, visualization and control platforms.

During the spring and summer of 2011, we were provided some additional funding to enhance the PSSA reference implementation to support Year at the Edge (YATE) and Summer at the Edge (SATE) projects. These projects are summarized in the previous section (Pinpoint and iStream).

Once again, PSSA proved to be flexible and easy to integrate with the student led projects to further enhance the application of layered sensing to solve real-world problems identified by AFRL. In addition, PSSA continues to be used to support phase II of the Information Quality project (TO-006). To this end, we added the capability to easily integrate MATLAB® based exploitation algorithms and incorporate sensor persistence into the PSSA directory services.

In phase two of this project, we plan to build upon these core technologies by implementing components necessary to field an operational system including expanded directory services, system management and deployment tools, user management and security tools, integration with high speed storage and messaging systems, etc.

Figure 36 shows a sample of the key components of such a system.

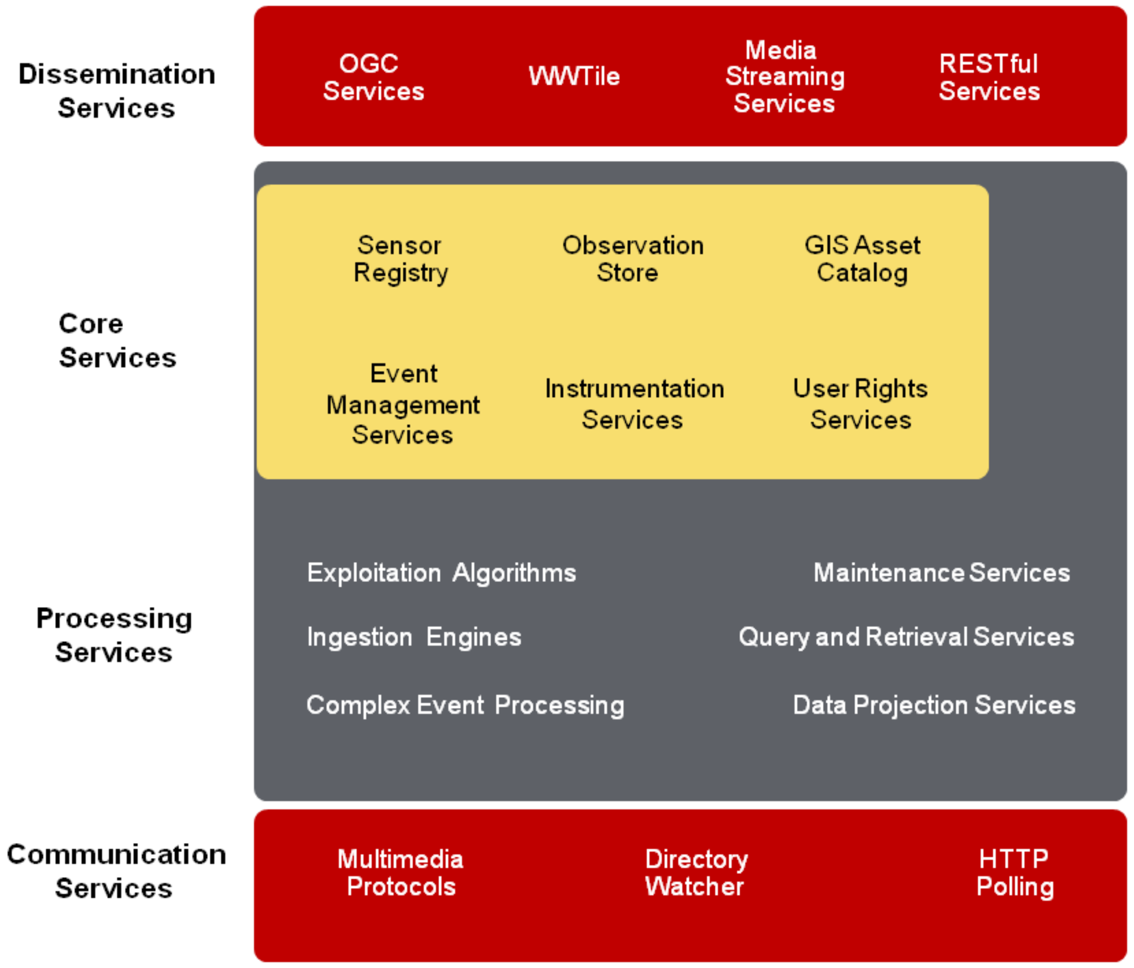


Figure 36: PSSA Logical Architecture

5.1 Dissemination Services

These services provide standard and custom interfaces for querying and retrieving data and products from the system. In Phase I, we implemented dissemination services for media streaming and geospatial vector and raster dissemination. In phase two, we will expand the set of available interfaces to provide a richer set of connections for third party clients. In collaboration with our partners, additional components will be developed that leverage these interfaces for providing data to other systems.

5.2 Core Services

The PSSA includes a standardized Core Services layer which is responsible for providing the key infrastructure required to support the integration of the components in the customizable software layers. In the first phase, we developed the minimal capabilities required to demonstrate the system.

The expansion of these services will be a focus of phase two and are critical to standing up and supporting an operational system. In addition, we will enhance the capabilities of the PSSA Software Development Kit to provide access to these additional core services.

5.3 Processing Services

These components subscribe to information published by ingestion or other application service components and process (“exploit”) the data thereby adding value. In Phase I, we developed reference implementations of text and video storage and processing services. In Phase Two, we will expand the suite of algorithms and ingestion services available to users of the PSSA system including adding support for algorithms written in MATLAB.

During Phase II, we will also investigate the integration Qbase’s QDT tool to allow users to create custom visual data flows that will execute within the PSS architecture. This will allow non-technical users the ability to easily create a data flow that extends the capabilities of the system.

5.4 Communication Services

These are interfaces by which external services and systems access the PSSA. They allow information and sensors to be ingested and products to be disseminated. In Phase I, we implemented services to support web, text and streaming media. In phase two, we will expand the set of supported protocols. These will be chosen in collaboration with our partners.

5.5 Hardware and Networking

The proposed hardware architecture for Phase II of the Persistent Surveillance Data Processing, Storage and Retrieval project consists of a high throughput compute cluster (HTCC) and a high performance storage cluster (HPSC) interconnected using a low latency, high bandwidth Infiniband network. Using currently available Infiniband switches, a network consisting of up to 648 total nodes can be configured. The optimal number of nodes and the mix between HTCC nodes and HPSC nodes is dependent upon the processing and storage requirements of the solution.

The high throughput compute cluster consists of a collection of commodity multi-core rack mount servers. Depending upon the processing needs of the system, each node can scale from entry level systems at a price point of $3000 to high-end systems at a price point of $20,000. The following is a nominal configuration for a potential compute node:

- 2U Rack Server: 4 x AMD Opteron 6136 (2.4GHz, 8 cores/each, 32 cores total)
- Windows Server 2008 R2 HPC Edition
- 64 GB RAM (32 x 2GB 1333MHz Dual Ranked RDIMMs)
- 900GB RAID 10 storage (6 x 300GB hot-swap 15K RPM SAS 6.0GB/s hard drive)
- 40GB/s IB Network Interface Card
- NVIDIA Tesla C1060 GPU (240 streaming processor cores @ 1.3GHz, 4GB DDR3 RAM)

Cost of this configuration is estimated to be approximately $12,500 per node.

The following is a nominal configuration for a Storage node:

- 3U Rack Server: 2 x AMD Shanghai Opteron 2384 (2.7 GHz, quad core, 8 core total)
- Windows Server 2008 R2 HPC Edition
- 32GB RAM (16 x 2GB 677MHz Registered DDR2 ECC)
- 4.8TB RAID 10 storage (8 x 600GB hot-swap 10K RPM SATA 6.0GB/s hard drive)

- 40GB/s IB Network Interface Card

Cost of this configuration is estimated to be approximately $11,500 per node.

For the reference implementation of the hardware and network architecture, the ideal configuration will be 4 compute nodes and 4 storage nodes. This should provide enough of a test bed to build a robust sensor processing system with the capability to benchmark system performance while addressing real-world processing problems such as geo-registration, object detection and tracking, change detection, sensor data fusion and information quality.

In addition to the hardware listed above, an Infiniband 40 GB/s switch will be required to interconnect the nodes of the network. Total cost of the hardware portion of the phase II reference implementation is expected to be approximately $120,000.

5.6 Commercialization

Qbase is actively investigating commercial opportunities that can leverage the research performed under this task order. Several promising leads are being pursued with federal government agencies as well as state and local government organizations. The architecture and design principles resulting from our research and described within this document provide a solid framework upon which Qbase can build commercial solutions to capture, process, and display sensor data from a variety of sensor devices in a "layered" format synchronized in time and space.

6.0 REFERENCES

Bibliography

Rizzi, A. and Gatta, C. and Marini, D., A new algorithm for unsupervised global and local color correction, PRL(24), No. 11, July 2003, pp 1663-1677.

Geo Frame Works. (2009). http://www.geoframeworks.com/articles/WritingApps2_3.aspx

Dilution of Precision (GPS); (2009). http://en.wikipedia.org/wiki/Dilution_of_precision_(GPS)#Meaning_of_DOP_Values

The Times-Picayune (2010). http://www.nola.com/news/gulf-oil-spill/index.ssf/2010/05/gulf_of_mexico_oil_spill_anima.html

Gilbert, T.D. (1997); Remote Sensing and Surveillance of Oil Spills, in Proceedings of 7th National Plan Scientific Support Coordinators Workshop, 22-26 September 1997, Darwin.

Gilbert, T.D. (1998); Maritime Response Operations- Requirements for Met/Ocean Data and Services. In proceedings of the Conference on Meteorological and Oceanographic Services for Marine Pollution Emergency Operations (MARPOLSER 98) 13-17 July 1998, Townsville, Pub WMO/IMO.

Brekke, Camilla and Solberg, Anne. (2004) Oil Spill detection by Satellite Remote Sensing. In Remote Sensing of Environment 95, November 2004, Elsevier.

NOAA. (2009). http://www.noaa.gov/

Buttingsrud, Bård. 2005, Superresolution of Hyperspectral Images. Chemometrics and Intelligent Laboratory Systems, pp. 62-69.

Subramaniana, Suresh. Methodology for hyperspectral image classification using novel neural. Methodology for hyperspectral image classification using novel neural. [Online] April 1997. [Cited: 04 20, 2010.] http://www.techexpo.com/WWW/opto-knowledge/NNET.pdf.

Zhaozhong Wang. Image Analysis and Recognition. tronto : Springer, 2006.

Bakker, W.H. 2002. Hyperspectral Edge Filtering for Measuring Homogeneity. ISPRS Journal of Photogrammetry, pp. 246– 256.

ISL; Image Change Detection for IED Detection. www.isl.eu. [Online] [Cited: 4 20, 2010.] http://www.isl.eu/Content/Image%20Change%20Detection%20for%20IED%20Detection.aspx.

Additional Resources

The following websites were referenced to gather background information used to put together certain sections of this report:

- ZeroMQ Messaging System:
  - http://www.zeromq.org/
- Time synchronization and leap second issues

    - http://www.ucolick.org/~sla/leapsecs/onlinebib.html

- Spatial synchronization and information quality metadata:
    - http://www.fas.org/irp/doddir/usaf/afpam14-210/part13.htm
    - http://gpsinformation.us/main/errors.htm
    - http://users.erols.com/dlwilson/gps.htm
    - http://portal.opengeospatial.org/files/?artifact_id=33234
- Integration with other systems:
    - DCGS: http://www.dtic.mil/cjcs_directives/cdata/unlimit/3340_02.pdf
    - DCGS/DIB: http://www.disa.mil/nces/dev_tech_resources.html
    - DCGS/DIB: http://www.dtic.mil/ndia/2004interop/Tues/meiners.ppt
    - DIB: http://www.hanscom.af.mil/news/story_print.asp?id=123153749
    - DDMS: http://metadata.dod.mil/mdr/irs/DDMS/
    - PULSENet: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5067463
    - PULSENet: http://www.is.northropgrumman.com/products/pulsenet/index.html
    - SensorNet/CBRN: http://www.sensornet.gov/net_ready_workshop/Tom_Johnson_August_02_2006_Brief.pdf
    - SensorNet/Briefings: http://www.sensornet.gov/net_ready_workshop/
    - SensorNet: http://www.sensornet.gov/sn_overview.html
    - IEEE P1451: http://ieee1451.nist.gov/
    - CCSI: http://www.jpeocbd.osd.mil/packs/Default.aspx?pg=860

APPENDIX A – PSSA DASHBOARD REFERENCE IMPLEMENTATION

Overview:

Processing Services Commercialization: Qbase is actively investigating commercial opportunities that can leverage the research performed under this task order. Several promising leads are being pursued with federal government agencies as well as state and local government organizations. The architecture and design principles resulting from our research and described within this document provide a solid framework upon which Qbase can build commercial solutions to capture, process, and display sensor data from a variety of sensor devices in a "layered" format synchronized in time and space. This section provides a description of a reference implementation of a sample dissemination and collaboration application, which will be referred hereafter as "PSSA Dashboard" or "Dashboard." As shown in Figure A-1, the main goal of developing an application like dashboard is to demonstrate the basic capabilities of the PSSA and further provide a means to possibly develop interfaces to build a complete sensor control and data management system. Further, like any other sensor in the PSSA cloud, the dashboard as a built-in capability to publish text streams into the message cloud, as a text stream sensor (I.E. the concept of representing people as feed generating sensors).

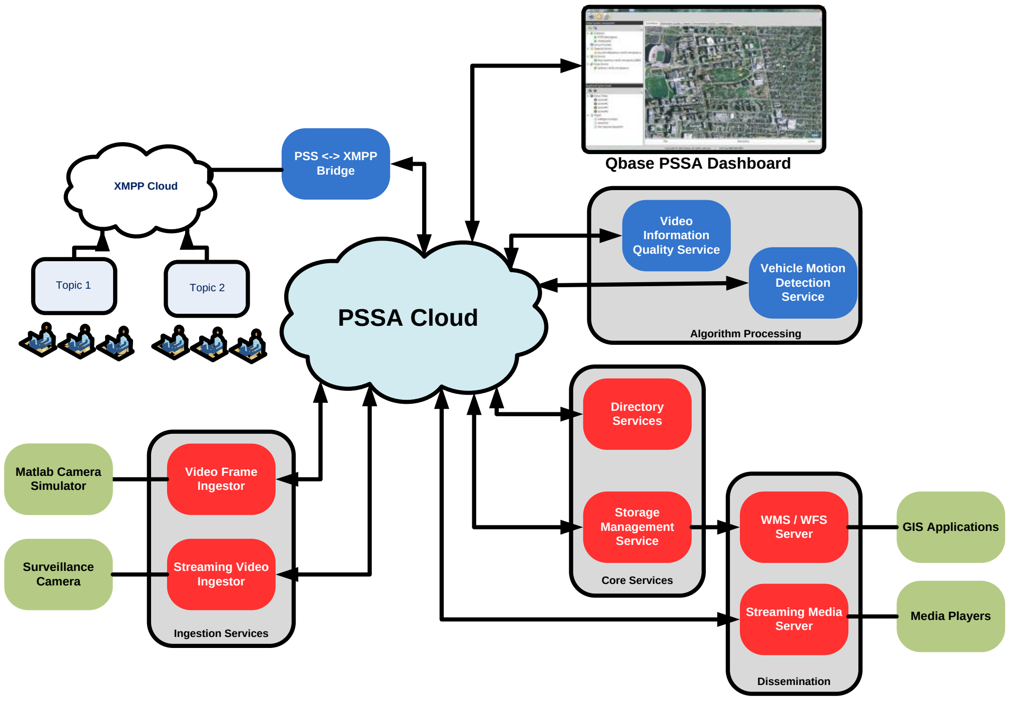


Figure A-1: PSSA with PSSA Dashboard

Figure A-1 is pictorial representation of PSSA cloud's reference implementation that can support collaboration among multiple users and support ingestion, storage and dissemination of the sensor data. The primary goal of developing an application like Qbase PSS dashboard is to demonstrate the basic capabilities of the PSSA.

Layout, Login & Initialization: As shown in Figure A-1, the layout of PSSA dashboard, as of today, has been designed to accommodate five functional areas in it:

- Sensor Feed Visualization or Manipulation (SFVM) (contains a tab view of Surveillance, Information Quality and the Collaboration windows).
- Registered Sensor Status Tracker (RSST).
- PSSA Components & other Infrastructure Tracker (PCIT).
- Text Event Stream Visualization (TESV).
- External Service Initiator (ESI) (Quick Launches to Google Earth, VLC Player and Pidgin Client).

The Modal "Select Session" window, on the application startup, provides a way to select and assign a unique identification to the user/analyst so that he/she can employ the dashboard for possible collaboration with other dashboard or third party application users present in the given PSSA cloud instance.

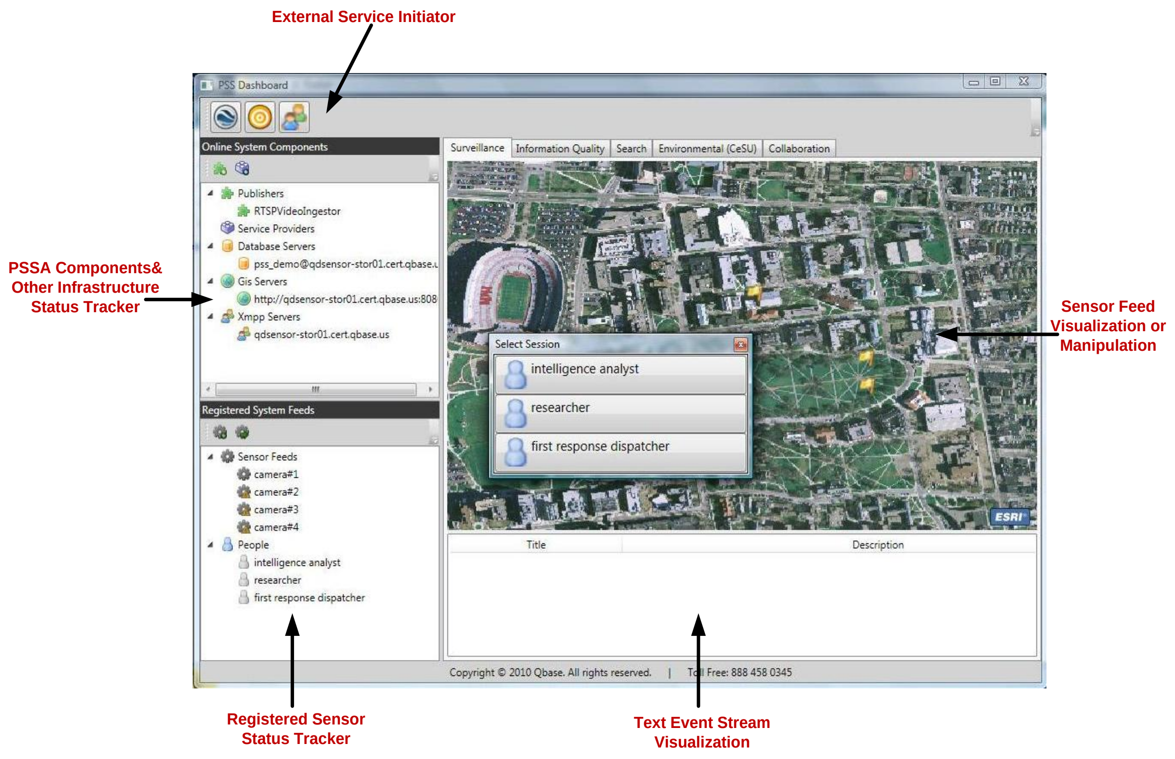

External Service Initiator
PSSA Components& Other Infrastructure Status Tracker
Sensor Feed Visualization or Manipulation
Registered Sensor Status Tracker
Text Event Stream Visualization
PSS Dashboard
Online System Components
Publishers
RTSPVideoIngestor
Service Providers
Database Servers
pss_demo@qdsensor-stor01.cert.qbase.u
Gis Servers
http://qdsensor-stor01.cert.qbase.us:808
Xmpp Servers
qdsensor-stor01.cert.qbase.us
Registered System Feeds
Sensor Feeds
camera#1
camera#2
camera#3
camera#4
People
intelligence analyst
researcher
first response dispatcher
Surveillance
Information Quality
Search
Environmental (CeSU)
Collaboration
Select Session
intelligence analyst
researcher
first response dispatcher
ESRI
Title
Description
Copyright © 2010 Qbase. All rights reserved. | Toll Free: 888 458 0345

Figure A-2: Dashboard Screen Layout

Figure A-2 shows a screenshot of the PSSA Dashboard startup view with modal window to select the user's identification (as a sensor-id), to represent themselves as a sensor feed publisher into the PSSA cloud. Additionally, the indicators point to the five functional areas of the dashboard's layout.

Besides the directory service, (and other core PSSA services, like Data Storage and Media Storage Services, shown previously in Figure 35) which defines the PSSA cloud instance, the dashboard requires three external servers to support the above mentioned first four functional areas. These external servers are:

- WMS Server
- Database Server
- XMPP Server

Presently, the GeoServer is installed as the WMS server, which serves the image tiles of Orthophoto data of the Ohio campus to the Surveillance tab of "Sensor Feed Visualization or Manipulation" area. The database server requirement is accomplished by the Postgis/Postgresql server instance, which presently stores the static data like the limited user identification values (Intelligence Analyst, Researcher & First Response Dispatcher for Login window and Registered Sensor Status Tracker) and the registered video camera sensor's metadata (primarily the location, as a point data accessible via a WFS request) to populate the "Registered Sensor Status Tracker" and also to place active/inactive flags on the Map display (as shown in Figure A-2 in the Surveillance tab of "Sensor Feed Visualization or Manipulation" area. An OpenFire real- time collaborative XMPP server is employed as the XMPP server to support the collaboration activities thru the dashboard (details would be explained in the later sub-sections). The dashboard presently receives the information regarding the presence of these servers and the PSSA directory service thru a configuration file as shown below:

```
<!-- common -->
    <add key="GisServerHostAddress" value="http://qdsensor-
stor01.cert.qbase.us:8080/geoserver/wfs?service=wfs"/>

<!-- pss.dashboard.modules.systemcomponentsobservationservice -->
    <add key="DirectoryServicesAddress" value="tcp://127.0.0.1:5555" />

<!-- Pss.Dashboard.Modules.FeedRepositoryService -->
    <add key="WfsGetFeatureSensorRegistry" value="&lt;?xml version=&quot;1.0&quot;
?&gt;&lt;GetFeature version=&quot;1.1.0&quot; service=&quot;WFS&quot;
xmlns=&quot;http://www.opengis.net/wfs&quot;
xmlns:pss=&quot;http://us.qbase.afrl/pss&quot; outputFormat=&quot;text/xml;
subtype=gml/3.1.1&quot;&gt;&lt;Query
typeName=&quot;pss:sensor_registry&quot;/&gt;&lt;/GetFeature&gt;" />
    <add key="WfsGetFeaturePersonRegistry" value="&lt;?xml version=&quot;1.0&quot;
?&gt;&lt;GetFeature version=&quot;1.1.0&quot; service=&quot;WFS&quot;
xmlns=&quot;http://www.opengis.net/wfs&quot;
xmlns:pss=&quot;http://us.qbase.afrl/pss&quot; outputFormat=&quot;text/xml;
subtype=gml/3.1.1&quot;&gt;&lt;Query
typeName=&quot;pss:users&quot;/&gt;&lt;/GetFeature&gt;" />

<!-- pss.dashboard.modules.infrastructuremonitoring -->
    <add key="GeoServerHostAddress" value="http://qdsensor-
stor01.cert.qbase.us:8080/geoserver/wfs?service=wfs&amp;version=1.1.0&amp;request=GetCapa
bilities&amp;namespace=pss" />
    <add key="SharedDatabaseServerConnectionString" value="Server=qdsensor-
stor01.cert.qbase.us;Port=5432;Database=pss_demo;uid=postgres;pwd=Password123!"/>
        <add key="XmppMonitorUserName" value="xmpp_monitor_user"/>
        <add key="XmppMonitorPassword" value="xmpp_monitor_pwd"/>

<!-- pss.dashboard.modules.feeditemservice -->
```

```
    <add key="XmppServiceAddress" value="qdsensor-stor01.cert.qbase.us" />
    <add key="XmppRoomName" value="sensor.observation.text.georss.acquired" />

<!-- Quick Launch paths -->
    <add key="GoogleEarthAppPath" value="C:\Program Files\Google\Google
Earth\client\googleearth.exe" />
    <add key="VlcAppPath" value="C:\Program Files\VideoLAN\VLC\vlc.exe" />
    <add key="XmppClientAppPath" value="C:\Program Files\Pidgin\pidgin.exe" />
```

Sensor Feed Visualization or Manipulation & Registered Sensor Status Tracker: The sensor feed visualization or manipulation area consists of tab view of Surveillance, Information Quality and the Collaboration windows, which are functionally active and are completely integrated with the PSSA cloud components at the runtime.

Surveillance Module: The Surveillance tab view provides a perceived surveillance scenario of monitoring the four video camera sensors that are installed and registered with the PSSA system for a given campus area. From the developer's perspective, the ESRI Windows Presentation Foundation (WPF) components was employed to render the orthophoto of the Ohio campus imagery at different zoom levels. ESRI map controller was proven to easily integrate into the dashboard and allowed placing the WPF controls and sensor location icons on the rendered image. The ESRI map controller module pulled the imagery from the above mentioned Geoserver, which already had static data loaded into it. The real-time on- line or off-line status of the video camera is communicated to the Surveillance tab view module, from the PCIT functional module of the dashboard, in response to which the golden flag at the camera location is turned into green indicating the status of the camera to be on-line (as shown in Figure A-3).

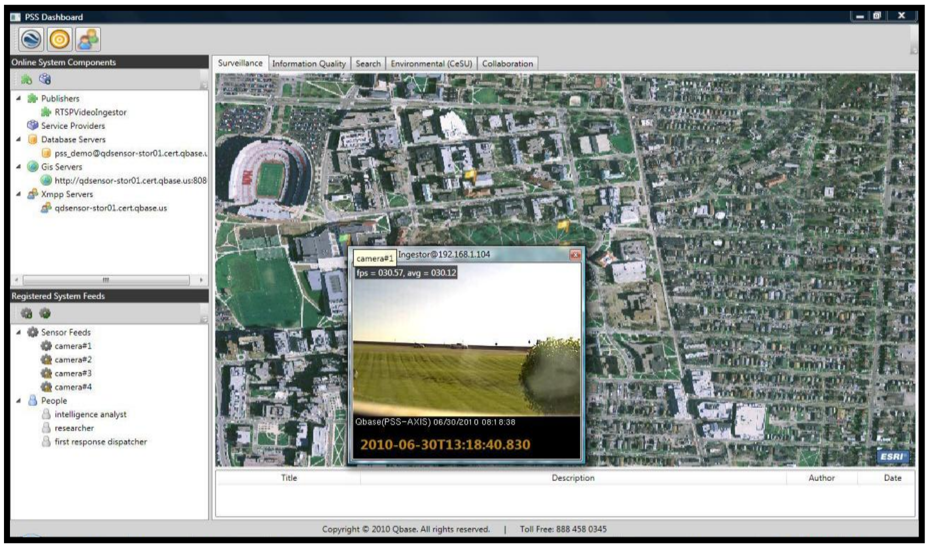


Figure A-3: PSSA Dashboard with Live Video Feed

Figure A-3 shows a screenshot of the PSSA Dashboard, with a live camera (camera #1) transmitting real-time video frames into the surveillance tab view. It should be noticed that the golden flag, indicating the off-line camera is turned into green flag for the on-line camera. Moreover, the RSST and the flag coloring on the map, corroborate each other for the camera #1 status. Further, the PICT listed the RTSP Video Ingestor to be active component publishing video frame events into the PSSA cloud from an on-line camera. Finally, the user can click on the on-line camera flag and visualize the real-time video feed in a modeless window, with appropriate timestamp information intact with the video frames.

Further, as shown in Figure A-3, the user can click on the live cameras and receive the real-time video frames from the installed camera thru the PSSA – RTSP Video Ingestor component in the cloud as the video frame events. The process of detecting, receiving and visualizing PSSA events is explained in the below sub-section.

Detecting, Receiving and Visualizing the (Real-Time PSSA Video) PSSA Events in the Dashboard (Developer's Perspective): The communication between the PSSA cloud and the dashboard is established thru the managed C++ wrappers around the C++ PSSA SDK library. The SDK provides a way to receive the list of the PSSA components (like RTSP Video Ingestor, GeoRSS Ingestor, Media Storage Service) as they become available, to all subscribers present in the PSSA cloud. This runtime notification is possible thru the process of registration, that every PSSA component as to complete as part of their initialization process.

This component registration process, thru SDK, communicates its presence to the directory service with its endpoint address, publishing event "topics" and optional target sensor identification (for ingestion components). The dashboard has the directory service subscriber module in it, which gets the event publisher information (endpoint, event "topic" and the sensor id) present in the directory service. This information is communicated to an event subscriber that can receive the events and has the necessary logic to interpret the event messages.

The current version of the dashboard modules, particularly the PICT, relies on the directory service subscription and maps the sensor id (from the directory service entry) to the camera id (present in the database) to determine its status (I.E. on-line or off-line). The RTSP Video Ingestor, publishes the video frame events using the below established video event message template:

```
MIME-Version: 1.0
Content-Type: multipart/related; boundary=MIME_boundary;
--MIME_boundary
Content-Type: text/xml
Content-Location:
event
<Event topic="sensor.observation.media.video:acquired?sensorid=1234">
 <Context>
   <Producer>Ingestor@192.168.1.104</Producer>
   <ReferenceSensorId>12</ReferenceSensorId>
   <Timestamp>2010-02-08T14:21:13.3320001</Timestamp>
   <Location>
                <Position>POINT(30
                  30)</Position>
   </Location>
 </Context>
 <Content>
   <VideoFrame type="partialFrame|keyFrame" bitsPerSecond="20000"
        width="320" height="240" samplePeriod="P0.033S"
        presentationTime="2010-02-08T14:21:13.3320001" href="cid:frame1"
        />
```

```
  </Content>
</Event>
--MIME_boundary
Content-Type: x-application/MPEG-4-frame
Content-Location: frame1
...frame bits...
```

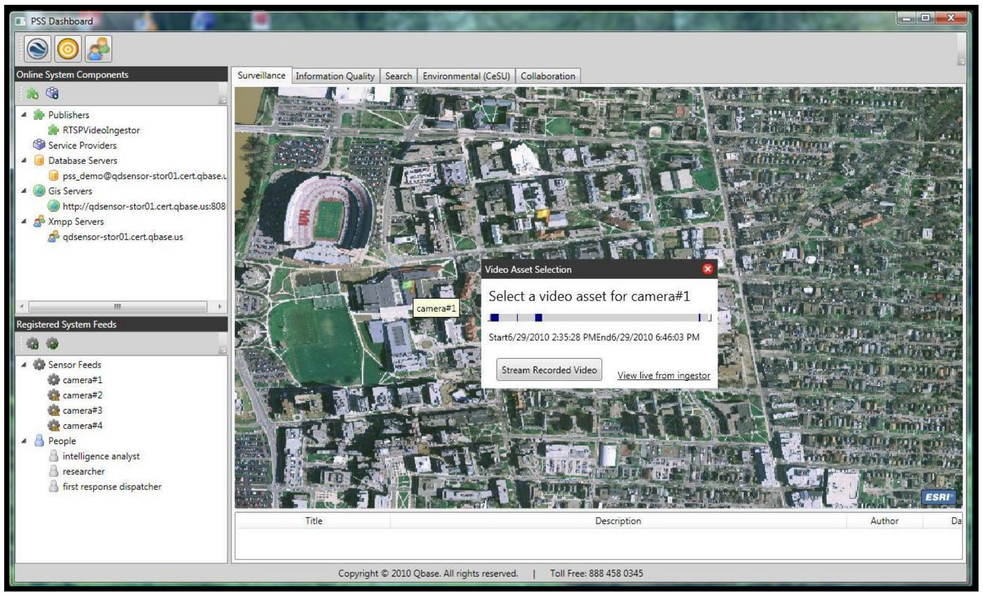


Figure A-4: Selecting a Recorded Video Stream

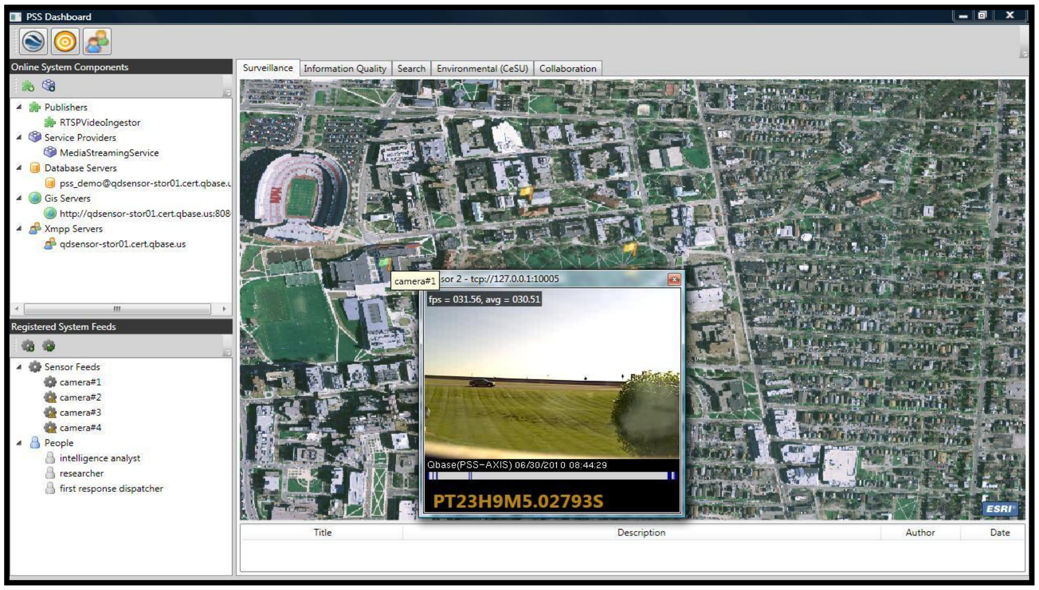


Figure A-5: Viewing a Recorded Video Stream

Figure A-4 and Figure A-5 are screenshots of the PSSA Dashboard, with a Media Storage (for camera #1) transmitting archived video frames into the surveillance tab view. The selection timeline in Figure A-4 and Figure A-5 indicates the camera on-line period (video availability) using blue strip and off-line period using grey strip. In the present dashboard version, the timeline is received from the Post GIS server, which is populated by the Media Storage Service during the recording phase. Further, it can be noticed that the PCIT module is indicating the presence of the Media Storage Service component present in the system. As noticed, the message template for the individual video frame event consist of the context and the content information, using which a given frame can be decoded into an appropriate image format (currently a bit map) to render it in the video window. The PSSA SDK provides MPEG-4 video decoding routine, which is used in the video display module of the dashboard.

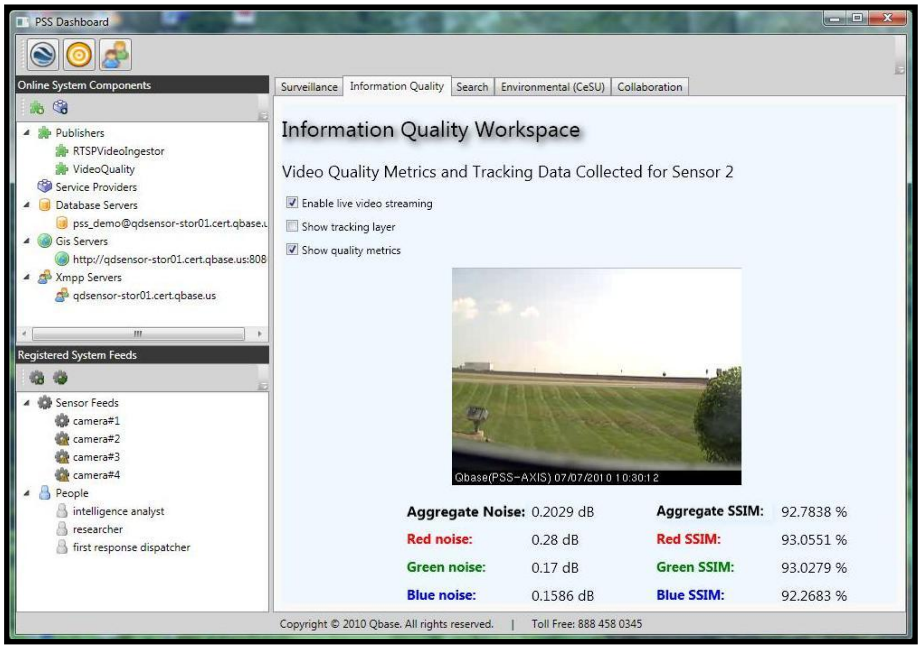
The same concept of receiving and visualizing the video frames from a live ingestion component (RTSP Video Ingestor) also applies to the archived video stream for a particular camera/sensor, which can be streamed on-demand from the Media Storage Service (as shown in Figure 35 and Figure A-3).

Currently, though the archived video frames are received thru the PSSA cloud, the timeline for the playback shown in Figure A-2 that indicates the camera on-line period (blue strip) and off-line period (grey strip) is received from the postgres server (which contains the timestamps of the video segments populated by the Media Storage Service, during the live stream recording). Thus, the surveillance module uses two video event subscribers and a database query to accomplish its presumed surveillance functionality. Though, the TESV module is part of the Surveillance module, because of its dependency on other modules of the dashboard, its functionality would be explained later in this report.

Information Quality Module: PSSA can support real-time exploitation of the sensor data stream and to demonstrate this capability, an Information Quality Module is built into the SFVM module. The goal of this module is to visualize the real-time data stream from a specific video camera (registered with the PSSA system) thru a ingestion component and also capture the generated quality metric events , which are being simultaneously computed on the same video events thru an exploitation (publisher) component. As shown in the figure, it is obvious that the video frame decoding and display module uses the same technique, which was mentioned with respect to the surveillance view module. Further, the quality metrics generated event template’s content are as an XML as shown below.

```
<Content>
<VideoFrameQualityMetric  metric_name = “NoiseRed”  metric_value = “0.99”
    unit = “db window= “1”  frame= “30” / >
<VideoFrameQualityMetric  metric_name = “NoiseBlue”  metric_value = “0.99”
    unit = “db window= “1”  frame= “30” / >
<VideoFrameQualityMetric  metric_name = “NoiseGreen”  metric_value =
    “0.99”  unit = “db window= “1”  frame= “30” / >.....................
  </Content>
```

The above quality event message is handled by an event subscriber module, present in the information quality module, which displays the metrics appropriately in the window as shown in Figure A-4. Thus, the information quality module uses one video event subscriber and a quality event subscriber to accomplish its presumed real-time exploitation functionality. Current version of the dashboard is hardcoded to support quality metric visualization for only on the camera with sensorid “2.”



Information Quality Workspace

Video Quality Metrics and Tracking Data Collected for Sensor 2

[✓] Enable live video streaming
[ ] Show tracking layer
[✓] Show quality metrics

Aggregate Noise:	0.2029 dB	**Aggregate SSIM:**	92.7838 %
Red noise:	0.28 dB	**Red SSIM:**	93.0551 %
Green noise:	0.17 dB	**Green SSIM:**	93.0279 %
Blue noise:	0.1586 dB	**Blue SSIM:**	92.2683 %

Copyright © 2010 Qbase. All rights reserved. | Toll Free: 888 458 0345

Figure A-6: Dashboard Displaying Information Quality Metrics

Figure A-6 shows a screenshot of the PSSA Dashboard, with a live camera (camera #2) transmitting real-time video frames into the Information Quality View, along with the display of the real-time quality metrics like noise and SSIM that were computed on the video frames by a Information Quality Exploitation service. Further, it can be noticed that the PCIT module is indicating the presence of the video quality (i.e. the exploitation component) publisher component present in the system.

***Collaboration Module*:** Collaboration services developed in the PSSA provide the ability for users to create and administer dedicated logical groups called “topic” rooms, participate in multiple “topic” rooms, and collaborate between “topic” rooms. The messages sent from the users are published to message bus using a PSSA-XMPP message bus bridge, which translates the message sent from the user into PSSA text/image event messages, as shown in Figure A-5. Users and PSSA components subscribed to these topics will receive messages. Currently, the collaboration module developed in the PSSA dashboard is a customized chat application module that only supports sharing text messages as shown in Figure A-6. As mentioned, the text chat messages are sent to the XMPP server present in the system or any server on the internet, from which a PSSA- XMPP bridge service has to be configured to publish these text messages as a text/image stream events into the PSSA cloud. Currently, the dashboard can support collaboration through a single hardcoded “topic” room on a given XMPP server. Further, the image share support is not provided in the current version of the dashboard.

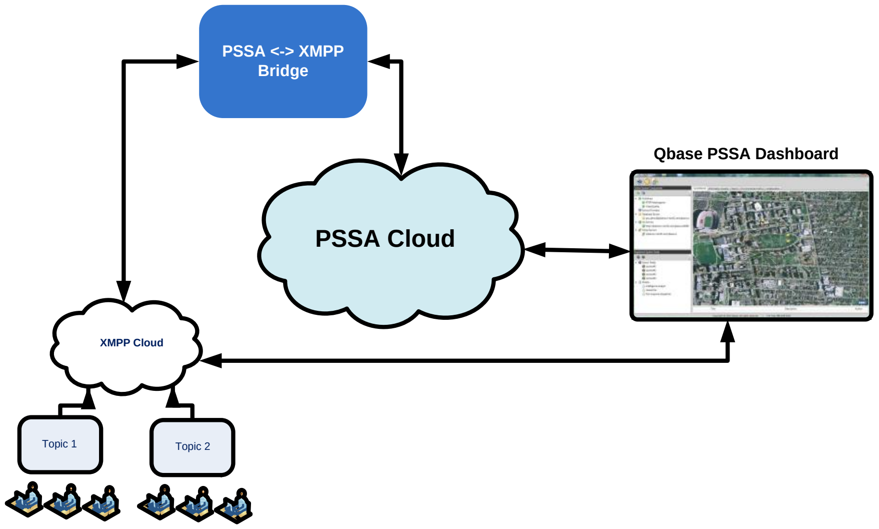
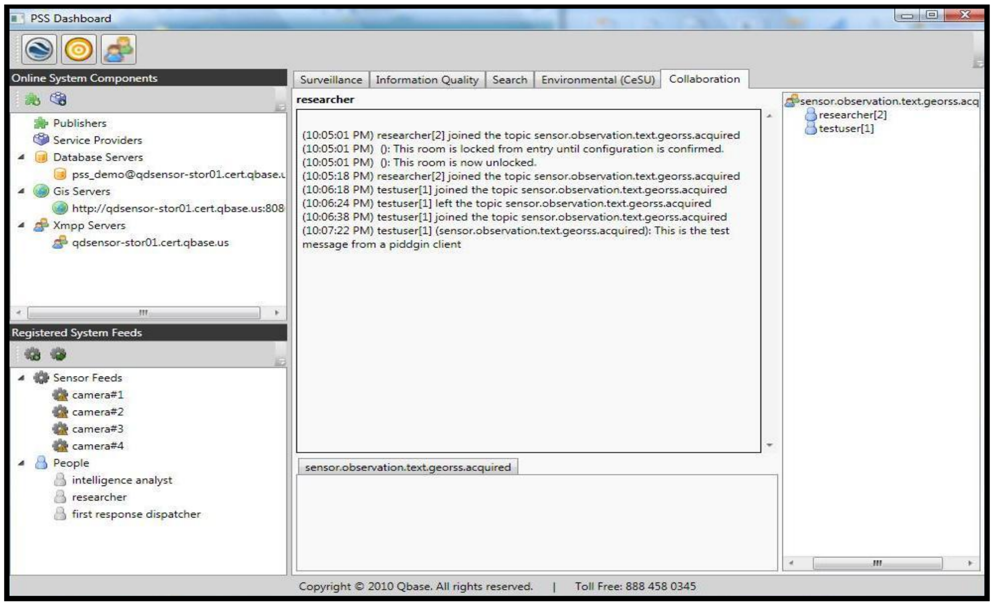


Figure A-7: Collaboration Using PSSA<->XMPP Bridge

The figure above shows that this approach effectively bridges the PSSA message bus across the Internet allowing remotely connected users to interact with the PSSA system and, in addition, permits users using third party XMPP-based tools to participate in the collaboration, as well.



```
(10:05:01 PM) researcher[2] joined the topic sensor.observation.text.georss.acquired
(10:05:01 PM)  (): This room is locked from entry until configuration is confirmed.
(10:05:01 PM)  (): This room is now unlocked.
(10:05:18 PM) researcher[2] joined the topic sensor.observation.text.georss.acquired
(10:06:18 PM) testuser[1] joined the topic sensor.observation.text.georss.acquired
(10:06:24 PM) testuser[1] left the topic sensor.observation.text.georss.acquired
(10:06:38 PM) testuser[1] joined the topic sensor.observation.text.georss.acquired
(10:07:22 PM) testuser[1] (sensor.observation.text.georss.acquired): This is the test
message from a piddgin client
```

Figure A-8: PSSA Dashboard Collaboration View

PSSA Component & Infrastructure Tracker: This is the critical module of the PSSA dashboard, which provides the information of all the available PSSA components present in the given PSSA cloud and also the information regarding the other external servers under the below five categories:

- Publishers
- Service Providers
- Database Servers
- GIS Servers
- XMPP Servers

The publishers and the service providers are the categories of the PSSA components that become available in the PSSA cloud at runtime. The presence of these components in the PSSA cloud is communicated to the directory service (provided as a part of the dashboard configuration data) thru a registration process (explained earlier in this report). The PICT module has a directory service subscriber module in it, which gets the event publisher's and the service provider's information that is present in the directory service at runtime and updates the information in the PICT area.

In the current version of the dashboard, the status or health of the PSSA components is not monitored directly, but instead it is deduced from the component's entry in the directory service only. As already discussed earlier, it can be noticed in Figure A-1 through Figure A-4 that the PICT tracks the PSSA components in the cloud and also drives the SFVM and RSST modules appropriately.

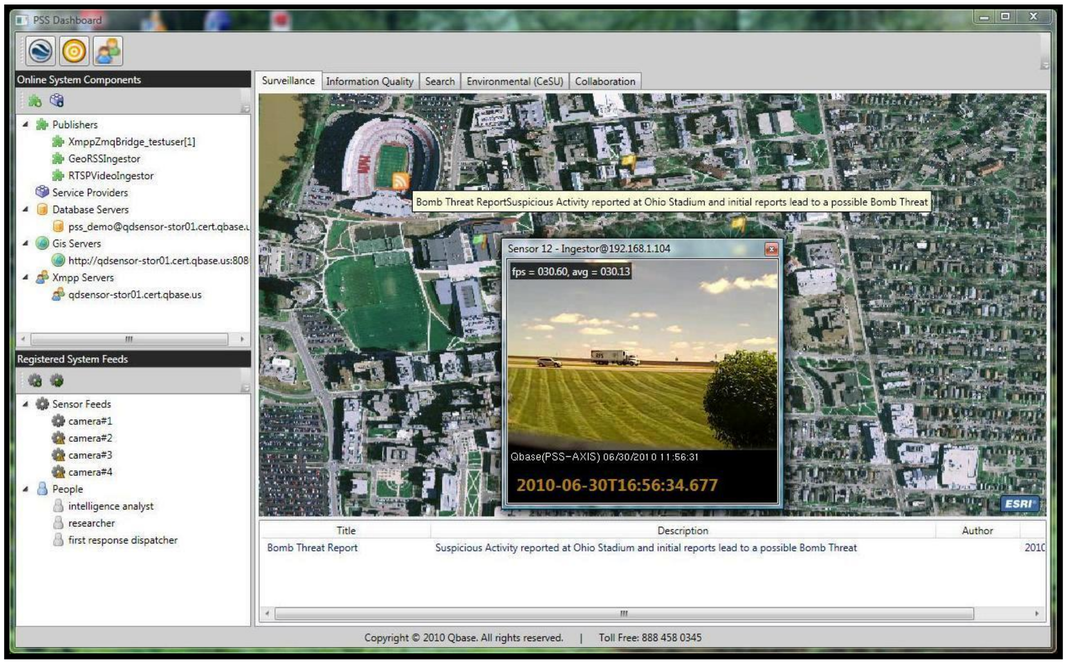
The database servers, GIS servers and XMPP servers are tracked by different tracking modules that take the information regarding these servers present in the system (provided as part of the dashboard startup configuration file). These tracker modules ping these servers for their availability on regulr intervals and update the information in the PICT area, as shown in the Figure A-1 through Figure A-4.

Text Event Stream Visualization: This is the sub-module of the surveillance module and it is responsible to capture the chat messages from the topic room of the XMPP server (as shown in Figure A-5) and detect (filter) only those messages that follow a GeoRSS message event template as shown below:

```
   <Event topic="sensor.observation.text.georss:acquired?sensorid=GeoRSS03">
                   <Context>
                                <Producer>producer1</Producer>
                                <ReferenceUserId>GeoRSS03</ReferenceSensorId>
                                <Timestamp>2010-04-
21T15:30:29.623000</Timestamp>
                                <Source>  <Description>USGS</Description>
<InetAddress>http://somewhere.com</InetAddress>
                                </Source>
                005</Context>
                <Content>
       <FeedItem title="Major event happened here" description="Lorem ipsum
dolor sit amet, consectetur adipiscing elit. Aenean auctor, metus in iaculis
fermentum, massa ligula imperdiet libero, eget accumsan nunc nisl vitae est. "
pubdate="2010-04-15T13:40:59" geolocation="POINT( -83.0196 40.00173 )" />
                </Content>
</Event>
```

These filtered messages are then parsed for the location, title, description and author information that is present in the above sample message and is appropriately displayed in the TESV area and also a GeoRSS icon is placed with the description information on the map area . A sample GeoRSS message event that was displayed in the surveillance view is shown in Figure A-9.

Currently, the dashboard doesn’t have the direct implementation of the GeoRSS message event subscriber in it, but the GeoRSS messages from the PSSA cloud are bridged into the XMPP “sensor.observation.text.georss.acquired” room thru the same collaboration bridge service mention in the Collaboration Module Section.



Bomb Threat ReportSuspicious Activity reported at Ohio Stadium and initial reports lead to a possible Bomb Threat

Title	Description	Author
Bomb Threat Report	Suspicious Activity reported at Ohio Stadium and initial reports lead to a possible Bomb Threat	2010

Figure A-9: PSSA Dashboard Showing GeoRSS Feed

Figure A-9 shows a screenshot of the PSSA Dashboard, with a GeoRSS event being displayed in the TESV area and on the map area of the surveillance module. Currently, the GeoRSS events are bridged from the PSSA cloud to an XMPP chat room (like "sensor.observation.text.georss. acquired"), through which the messages in the chat room are filtered based on the GeoRSS event template mentioned in the report.

External Service Initiator: This is an experimental module that was placed to start external services like Google Earth, Pidgin chat client and a VLC player.

APPENDIX B – TUTORIAL FOR BUILDING PSSA COMPONENTS (SDK REFERENCE IMPLEMENTATION)

Introduction:

Overview: PSSA is an event driven message bus architecture that supports ingestion, storage and exploitation of disparate sensor data. Further, this composable architecture model supports collaborative framework for data analysts and decision makers to access sensor data and exploited data in real time. The architecture supports on the fly addition and deletion of loosely bound ingestion/storage/exploitation components without taking down the whole architecture down. Getting into the details, as mentioned the PSSA can support below logical services:

- Ingestion services (a gateway service to receive sensor data from outside PSS architecture) provide endpoints, within the architecture for collecting sensor data.
- Exploitation services exploit information (from ingested senor data) by analyzing and producing appropriate metadata.
- Core services provide high-speed data storage for system management.
- Dissemination services provide client based interfaces to subscribe and publish data to and from the message bus.

PSSA has a plug and play approach. “Plug and play” means the system does not have to be shut down to add new capabilities. Interfaces are well defined so anyone can build components that “plug and play” with the system.

PSSA Software Development Kit (SDK): As part of the effort to design and develop the PSSA composable Message Bus Architecture to ingest, exploit, store and visualize disparate real-time sensor data, an SDK has been developed in native C/C++ language. This developed PSSA SDK aptly provides the required API interfaces for any consumer of the PSS Message Bus Architecture, to build low-level PSSA components like Publisher, Subscriber and Service Provider. As shown in Figure A-8, these low-level PSSA components can be grouped (according to the custom business logic) to build high level customized components like Sensor Data Ingestor, Exploitation Service, Data Storage Service and any helper services like XMPP Message Bridge. Further, the PSSA SDK has been designed to effectively abstract the architecture and communication level (along with few media processing level) complexity from the consumer and aid him/her in building the high-level business logic components that can use the PSS Message Bus Architecture.

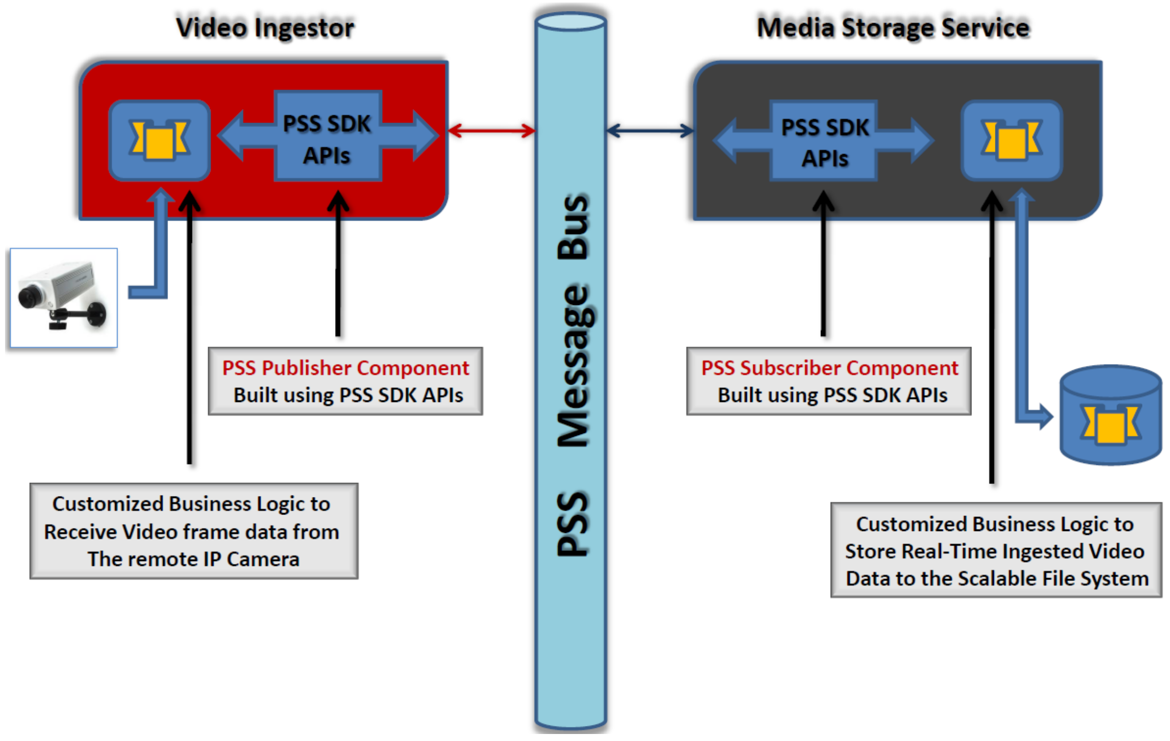

Video Ingestor
PSS SDK APIs
Media Storage Service
PSS SDK APIs
PSS Message Bus
PSS Publisher Component Built using PSS SDK APIs
PSS Subscriber Component Built using PSS SDK APIs
Customized Business Logic to Receive Video frame data from The remote IP Camera
Customized Business Logic to Store Real-Time Ingested Video Data to the Scalable File System

Figure B-1: Implementation of Custom Processing Using PSSA SDK

The developed PSSA SDK provides the required API interfaces for any consumer of the PSS Message Bus Architecture, to build low-level PSSA components like Publisher, Subscriber and Service Provider. These low-level components can be used individually (as shown in Figure B-1) or grouped (for building Exploitation algorithms that would require Subscriber and Publisher components in them) appropriately to build customized components like Video Ingestor, Media Storage Service and other Exploitation Algorithm Services.

Installation/Quick Start

To Begin:

Download the PSS_SDK library for the language of your project. Include the library in your project.

To Execute The Directory Service:

1. Launch command prompt.
2. Change directory to the directory where DirectoryService.exe was downloaded to.
3. Pass DirectoryService.exe your endpoint address and lower and upper bound port numbers as arguments.

```
DirectoryService tcp://172.16.10.138:5555 6601 6700
```

The above highlighted IP address should be the IP address of the computer hosting/running the directory service.

4. Press enter to execute. The directory service will then launch by listening to services running on the endpoint address. (At this point no services will be detected).
5. Leave the command window open.

To Load An Existing Ingestor:

***NOTE**: Make sure you have a directory service running.*

1. Launch a text editor to create a command file which will launch your ingestor executable file by passing the appropriate arguments. All ingestors will need the directory service's endpoint address, lower port number, and upper port number.

An example of the command file to run the RTSP Ingestor can be found below:

```
RTSPVideoIngestor tcp://127.0.0.1:5555 6901 7000 127.0.0.1 1
   rtsp://172.16.24.177/live2.sdp bits-per-second:1000000;buffer-
   window:0;width:320;height:240;frames-per-second:29.7;sample-time-
   factor:330
```

2. Save the command file in the same directory as your ingestor executable file.
3. Run the command file.

If directory service is started and running, a new ingestor entry, from the above RTSP Video Ingestor, it should be registered with the directory service.

Ingestion Services

Ingestion Overview: The ingestion services are responsible for importing the data generated by sensors and their associated processing systems into the PSSA system. These services are the point of entry for all external data to the PSSA system. Ingestors create published messages to be sent to the messaging interface. The messages can contain different content types so ingestors should accommodate different modalities and delivery formats. Ingestors should know how to build its message completely meaning the service must know the storage format and sensor ID. Ingestors provide means to normalize data streams and meta-data with agreed PSSA message event schemas. After informing the interface of all information, the SDK will publish feed items to the message bus interface using 0MQ protocols. 0MQ is an API for the transportation of message on the message bus**.**

Building New Ingestors

Required Information To Build An Ingestor

- Know the event message schema.
  - Content type

        - ✓ Xml
            - GeoRSS simple
            - GeoRSS w3c
            - Video frames (MPEG-4)
- Content location
    - ➢ Point
        - ✓ Longitude, Latitude
    - ➢ Polygon
        - ✓ Surrounding area of points
    - ➢ Context (Details of feed to be published)
        - ✓ Publisher Title
        - ✓ Sensor ID
        - ✓ Timestamp
        - ✓ Source
- Agree on a sensor ID.
- Know the information regarding the endpoint of the Directory Service in your Architecture.
- Know how to build the message completely.
    - ➢ Get publisher address.
    - ➢ Get content to be published.
        - ✓ Examples:
            - GeoRSS source configuration.
            - Media URL.
    - ➢ Synchronize data streams and meta-data (if required).
    - ➢ Provide different modalities and delivery formats to normalize data. (EO, RF, Text, Frame-based, Streaming)

To write an ingestor in C++

1. Add the PSS SDKLib C++ project to your solution (if it is not already).
2. Allocate a port provider by passing the appropriate start and end port numbers. The Port Provider provides a synchronized allocation mechanism to keep track of valid ports available.

```
PortProvider *yourPortProvider = new PortProvider(startingPortRange,endingPortRange);
```

3. Create and initialize a new context class object. The context class has multiple constructors for initializing the 0MQ to transport the message onto the message bus, for the publishers and subscribers that would be built within this context.

```
boost::shared_ptr<pss::sdk::Context> yourContext = new Context();
```

This constructor takes the directory endpoint address, the port provider created in step 2, and the number of application and IO threads to setup the 0MQ context.

```
Context(directoryServicesEndpoint,portprovider,numberAppThreads =
   DEFAULT_APPLICATION_THREADS, numberIoThreads = DEFAULT_IO_THREADS);
```

4. Create a published event topic schema. The SDK creates a queue of event topics called TopicNames.

Push the topic name onto queue using the push_back function. Pass the full topic string that includes the sensor ID that was agreed on. An example for a GeoRSS event topic is below:

```
TopicNames YourPublishingTopic;

YourPublishingTopic.push_back("sensor.observation.text.georss:acquired?sensorid
   =" + sensorid );
```

5. Call method to build the publisher. This method requires the publisher’s name, address, and queue of topic from step 4 above.

```
yourContext->buildPublisher(publisherName,publisherAddres,topics)
```

A publisher object is returned from the SDK so it must be assigned to a created publisher. An example of this can be seen below:

```
_yourPublisher = yourContext->buildPublisher(. . .);
```

6. Create a method that uses business logic to wrap around the SDK publish Event method.

```
_yourPublisher->publishEvent(publishingTopics, message);
```

Use the SDK’s message class functionality to call the wrapper method created above. The message class takes a created topic that was allocated in step 4 above. An example of this can be seen below:

```
Message* message =  new Message();
message->addMessagePart(contentId, contentType, data, dataLength, isFirstPart)
```

Example:

```
message->addMessagePart("event","text/xml",(void *)(
   data.c_str()),data.length(),1);
```

7. Publish the message by calling the method created in step 6.

Exploitation Services:

Exploitation Overview: Exploitation components need to know what kind of publisher it will be talking to. The exploitation extends sensor data through meta-data. Exploitation services are information quality algorithms and metrics off the message bus. Exploitation services use the Software Development Kit (SDK) in its interface to initialize, subscribe, and create messages. The SDK receives events, counts message parts, and gets message part by index. These application services will generate additional meta-data or new data streams that are published to the messaging interface.

Message Filters: Publishers normalize video data by encoding them into MPEG-4 frames. Message filters are used to decode these video frames into JPEG's or BMP's accessible in memory. The PSS SDK can create and release message filters in the PSS interface known as the message filter factory. Here a message filter is declared and then initialized by passing the type of filter. The type below is a MPEG-4 to BMP decoder filter.

```
boost::shared_ptr<::pss::sdk::MessageFilter> messageFilter;

messageFilter(pss::sdk::MessageFilterFactory::createMessageFilter(PSS_MPEG4_DECODE));
```

The message filter is then applied within the message interface. The MPEG-4 to BMP filter creates a PSS_SDK_BMP_STRUCT to store the BMP files. This data structure holds general file information and DIB specific information.

```
yourBMPFileStruct = messageFilter->applyfilter(yourSource, sourceLength,
   filterLength);
```

Sample Code:

```
PSS_MPEG4_DECODE is the filter type of your message.

::boost::shared_ptr<::pss::sdk::MessageFilter>
   messageFilter(::pss::sdk::MessageFilterFactory::createMessageFilter(PSS_MPEG4_DECO
   DE));

pss_bmpfile *bitmap_ptr = 0;

bitmap_ptr = reinterpret_cast<pss_bmpfile*>(messageFilter-
   >applyfilter(static_cast<const void*>(&eventData[0]), eventData.size(),
   bitmapSize));
```

Building New Exploitation Services

To Write a Directory Service Listener Thread:

The PSS system implements threads using the shared Boost library. A boost::thread object represents a single thread of execution and uses the native CreateThread and related calls. You can easily write portable code that runs across all major platforms.   A boost::thread object is constructed by passing the threading function or method it is to run. Below are the steps to write a directory listener thread:

1. Add the PSS SDKLib C++ project to your solution (if it is not already).
2. Initialize a directory service listener thread with the directory service endpoint address and a created context class object for your exploitation service.
3. Define a static callback method.
4. Provide handler for a notification change.
5. Obtain notification changes from the PSS directory entry data structure. `PSS_DIRECTORY_CHANGE_ACTION` is a data structure containing the action taken by a service (i.e. added, removed). `PSS_DIRECTORY_ENTRY_TYPE` is a data structure containing the type of service (i.e. publisher, PSS service).

```
typedef struct
{
   PSS_DIRECTORY_CHANGE_ACTION action;
   PSS_DIRECTORY_ENTRY_TYPE entry_type;
   const char* name;
   const char* endpoint;
   union
   {
      const char* topic;
      const char* service_contract;
   };
} PSS_DIRECTORY_ENTRY;
```

Next we need to obtain the notification change count from the PSS SDK. An example of creating and passing a callback would look like one below:

```
static void callback(PSS_DIRECTORY_ENTRY* changes, const unsigned short
   change_count, const unsigned long hint)
{
   //  Use the directory changes entry from the callback
   //from the DirectoryServiceObserver
```

```
            /// You can use the hint for casting back to the
            ///appropriate class object, if a valid hint is passed
            ///intially as below::
            // context->buildDirectoryServicesObserver(callback(unsigned
               long)this));
        }
```

6. Build the directory service observer by passing it the callback function and a unique ID to identify itself for casting back to the appropriate class object (hint).

To Write A Subscriber In C++:

1. Add the PSS SDKLib C++ project to your solution (if it is not already).
2. Create a context class object from the PSS SDK.
3. Call the buildSubscriber method by passing in the publisher endpoint and a blank topic.

A new subscriber object will be returned and ready to receive events.

```
_subscriber = yourContext->buildSubscriber(publisherEndpoint, topicName)
```

Initialize a new PSS Message object.

```
Message* msg = new Message();
```

Receive an event by passing the message object created above and the timeout in milliseconds. To do this, we call the receiveEvent function included in the SDK.

```
_subscriber->receiveEvent(eventMessage,
```

Sample Ingestor Component: It is assumed that directory service is running at "tcp://127.0.0.1:5555."

```
boost::shared_ptr<pss::sdk::PortProvider> portprovider =
   boost::shared_ptr<pss::sdk::PortProvider> (new PortProvider(6666,7777));

boost::shared_ptr<pss::sdk::Context> _context = boost::shared_ptr<Context>( new
   Context( Endpoint("tcp://127.0.0.1:5555"),portProvider ) );

pss::sdk::TopicNames _publishingtopics;

_publishingtopics.clear();

_publishingtopics.push_back(std::string("sensor.observation.text.test:acquired?sensor
   id=") + "1234");

_publisher = _context->buildPublisher("TestIngestor", "127.0.0.1",
   _publishingtopics);

string feed = "TestFeed-it doesn't follow any event schema for this example";
```

```
while(true)
{

   Message* message =  new Message();
   message->addMessagePart("event","text/xml",(void*)feed.c_str(),feed.length(),1);
   ///only event topic is present for this Ingestor
   _publisher->publishEvent( _publishingtopics.front(), *msg );
   delete message;
}
```

Sample Subscriber Component: It is assumed that directory service is running at "tcp://127.0.0.1:5555" and the above Ingestor can be reached at "tcp://127.0.0.1:6666."

```
///PortProvider is not needed for the subscriber
boost::shared_ptr<pss::sdk::PortProvider> portprovider =
   boost::shared_ptr<pss::sdk::PortProvider> (new PortProvider(7777,8888));
boost::shared_ptr<pss::sdk::Context> _context = boost::shared_ptr<Context>( new
   Context( Endpoint("tcp://127.0.0.1:5555"),portProvider ) );
boost::shared_ptr<pss::sdk::Subscriber> subscriber_ =
   boost::shared_ptr<pss::sdk::Subscriber>(context_-
   >buildSubscriber(Endpoint("tcp://127.0.0.1:6666"),"" ));
while(true)
{
   Message* msg = new Message();
   subscriber_->receiveEvent( *msg, 1000);
   /// Get the first part from the message
   const std::vector<unsigned char>& eventdata = msg->messagePart(0);
   std::string eventdataT( (const char*) &eventdata[0], eventdata.size() );
   std::cout << std::endl << "From the Sample Ingestor" << eventdataT << std::endl;
   delete eventdata;
   delete msg;
```

Data/Media Retrieval Services:

Retrieval Service Overview: These Retrieval components provide access to data stored internally within the PSS system or provide "gateways" to retrieve data from external systems. The Retrieval component provides an abstract interface to hide the physical location and structure of the data. Open standards protocol can be used to retrieve this archived/live/processed data. Customized interfaces can be "plugged" into the system to meet application specific retrieval processing needs.

To Write A Retrieval Service In C++:

1. Create a context class object from the PSS SDK.

2. Build a new service provider by passing the service provider's name, network address, and service contract.

```
yourContext->buildServiceProvider(PROVIDER_NAME, NETWORK_ADDRESS,
SERVICE_CONTRACT);
```

3. Set up a request handler by passing the built service provider. The handler waits to receive a request. If a request is not received, destroy the session and continue waiting.

```
boost::scoped_ptr<pss::sdk::Service> requestHandler(yourContext->
    buildServiceProvider(PROVIDER_NAME, NETWORK_ADDRESS, SERVICE_CONTRACT));

if( !requestHandler->receiveRequest( request, 1000 ) )

{

    destroyUnneededSessions();

    continue;

}
```

4. Once a request is received, parse the requested message (XML).
5. Create a response message by passing all parts of the message. This includes the content id and type, the data and data length, and whether or not it is the first part of the message.

```
            pss::sdk::Message response;

    response.addMessagePart("top","text/xml",result.c_str(), result.length(),
    true);
```

6. Send the response.

```
    requestHandler->sendResponse( response );
```

APPENDIX C – JAVA BINDINGS FOR PSS SDK

PSS SDK: The developed PSSA SDK has been successfully employed to build various customized components in the PSS architecture that were developed in C++ language. But, as mentioned earlier, the diligent approach to develop the PSSA SDK in native C/C++ language aided in generating native C *dll* for the PSSA SDK APIs. This native PSSA SDK C *dll* provides "Language Interoperability" across other languages like MATLAB® , C#, Python and JAVA. Thus, this "Language Interoperability" of the PSSA SDK provides the opportunity for the consumers developing components in other languages (other than C or C++) to port or integrate their existing components with PSS Message Bus Architecture with minimal development effort. Further, it is been demonstrated effectively that the PSS SDK can used in C# (.NET framework, employed to build PSSA dashboard, which is completely integrated (can Publish and Subscribe) with PSS Message Bus Architecture using the PSSA SDK) and in MATLAB® (used to build PSS Info Quality simulator employing the UALR Image Quality Algorithms), which is successfully ingesting (publishing) data into the PSSA using the PSSA SDK).

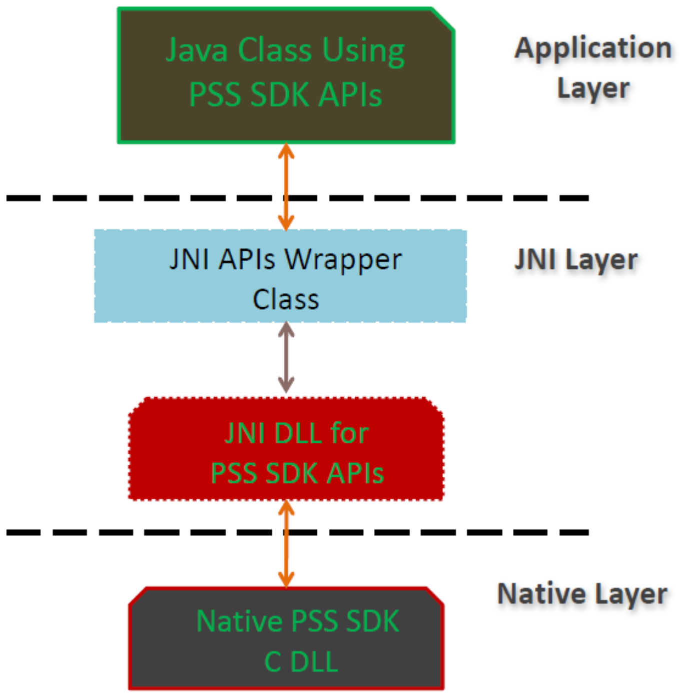


Figure C-1: Illustration of Java Bindings for the PSSA SDK

The developed Java bindings for PSSA SDK provides the required Java Native Interface (JNI) API callable wrapper class/interfaces for any Java class in the application layer to employ the Native PSSA SDK C *dll* at runtime. The function signatures exposed to the application layer would be same as that of Native PSSA SDK, when built with Simplified Wrapper and Interface Generator (SWIG), which in turn introduces a SWIG layer (not shown in the figure) that would hide the Java's name decoration in JNI wrapper interface.

Since, one of the primary goals of this project is to provide capabilities to integrate the developed PSS Message Bus Architecture with prominent cutting-edge processing and visualization tools like OpenLST and AVTAS Multi-Agent Simulation Environment (AMASE), an agent-based simulation software for UAV mission planning and Control), which are developed in Java language, there exists a need to explore and develop the java bindings for the PSSA SDK APIs.

Developed Java Bindings for PSS SDK Using SWIG: Sun's Java Development Kit, provides a programming framework feature called JN to call native APIs written in C/C++ languages. But, the C *dll*, along with the native library has to be compiled with appropriate JNI name decorations into a new JNI *dll* that would be callable from a simple java class present in any java package or module. Thus, employing this standard procedure to generate the appropriate java bindings for PSSA SDK would enable to use the SDK APIs from a java class as shown in Figure C-1. To reduce the development effort and time, we have employed SWIG to generate the above mentioned java bindings for the PSSA SDK. Further, employing the SWIG to generate the java bindings, it is easy to preserve most of the function signatures used in the native C *dll*, as shown below.

Native C Function Call:

```
// Initializes the library and must be called

// (or PSS_InitializeEx) before any other function

// is invoked.

long PSS_Initialize( char* directory_endpoint, int starting_port_number, int
    ending_port_number);
```

Exposed Java Interface Function Call:

```
int PSS_Initialize( String directory_endpoint, int starting_port_number, int
    ending_port_number);
```

It can be noticed that apart from the data type mappings from C to Java language, the signature of the function stays the same, which is helpful to easily maintain the SDK code and provide a unified tutorials or documentation for the APIs. The appropriate practices needed to call native *dll* from the java (http://www.swig.org/tutorial.html) would still apply like for example the JNI *dll* (PSS_SDK_JNI) has to be loaded into the application before referencing any of the PSS SDK functions like below:

```
try {

    System.loadLibrary("PSS_SDK_JNI");

} catch (UnsatisfiedLinkError e)

{

    System.err.println("Native code library failed to load\n" + e);

    System.exit(1);

}
```

And the APIs calls from the application would be called with parameters as below:

```
PSS_SDK_JNI.PSS_Subscribe ("tcp://192.168.10.112:6601",
   "sensor.observation.media.video:acquired", subp);
```

The generated Java bindings for the PSS SDK have been successfully integrated with the LSTVisulization module of the OpenLST project by building a Video Frame PSS Subscriber and Directory Service Listener, with minimal helper classes to stream live IP camera directly to OpenLST platform. Figure C-2 provides the snapshot of the Qbase PSS Message Bus Architecture driven video being integrated in the OpenLST.

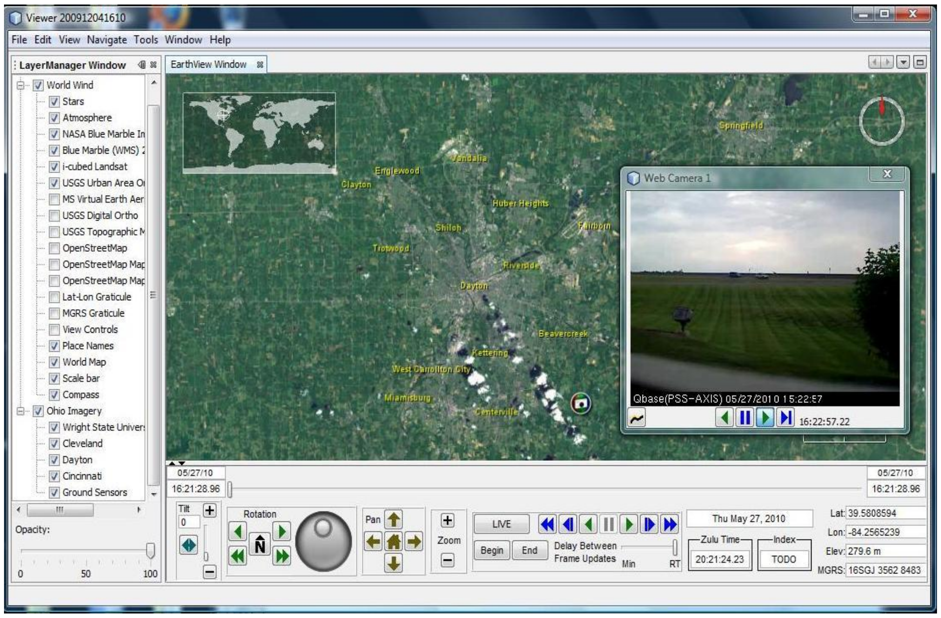


Figure C-2: Screenshot Showing Integration with OpenLST using Java SDK

The developed Java bindings for PSS SDK have been successfully employed to integrate the OpenLST platform with PSS Message Bus Architecture. The screenshot in Figure C-2 shows the capability of Java bindings of PSS SDK to build a live video stream subscriber that can receive live video frames from a remote IP camera via the PSS Message Bus Architecture and use OpenLST modules to display the video frames in real time.

APPENDIX D – Synchronization Issues

Temporal Synchronization: Time synchronization of multiple sensor feeds is a critical aspect of persistent surveillance and layered sensing applications. This makes time one of the most important attributes that must be captured for the storage and retrieval of persistent surveillance data. In order to perform queries and compare results across multiple layers of sensor data, a consistent temporal frame of reference must exist among the layers. A consistent temporal frame of reference allows events identified in one sensor's coverage area to be correlated with events in the coverage areas of other sensors. This, in turn, allows more sophisticated analytics to be performed on the sensor data. For example, if we are tracking a vehicle across the field of view of a Wide Area Persistent Surveillance (WAPS) sensor, then we can predict when that vehicle will enter the field of view of a high resolution ground camera. The ground camera in turn can provide much more detailed information about the vehicle then can the WAPS sensor. This information can then be used to increase the level of certainty that the WAPS sensor is tracking the correct vehicle. Time synchronization between the WAPS sensor and the ground camera sensors is critical for this scenario to work.

Unfortunately there is no way to guarantee the time synchronization of the individual autonomous sensors that may participate in a layered sensing scenario. In order to accommodate the wide variety of sensors that could be used in conjunction with the PSS system, we have established the conventions outlined in this section of the document.

The PSSA defines three time concepts which are used to establish a temporal frame of reference for the sensor data captured:

1. **Acquisition time** – This is time information provided by the sensor in the metadata that accompanies the sensor data. This data may not be present for every type of sensor. Even if it is present, the sensor time could be based on a number of different time standards — GPS, International Atomic Time (TAI), National Institute of Standards and Technology (NIST) UTC, or an internal time clock. In addition, there are a number of formats by which time could be represented (e.g. GPS Week/Seconds, ISO 8601, number representing the time elapsed since a specific date — Julian Date, Modified Julian Date (JD, MJD), Unix epoch, etc.)
2. **Ingestion time** – this is the system time at which the sensor ingestion component received the sensor data. The system time is based on the NIST UTC time reported by a Stratum 2 Network Time Protocol (NTP) server (or for more precise time, a Precision Time Protocol (PTP) server). This time is used to ensure that, in a distributed system, the time clocks of each of the ingestion systems are synchronized.
3. **Presentation time** – this is the normalized time used by the persistent surveillance sensor to synchronize the sensor data from this sensor with data received from other sensors. Since not all sensors will have synchronized time clocks, it is the responsibility of the ingestion component to determine the most likely time span during which the sensor data was collected. This could be based upon the acquisition time, the ingestion time or a combination of the two depending upon the type of sensor for which the data is being ingested. In order to accommodate variations in the precision of the presentation time between different sensor types, the precision of the time measurement is stored along with the time as the base 10 logarithm of the number of nanoseconds of precision represented by

the time value. For example, a value of 0 represents one nanosecond precision, a value of three represents one millisecond of precision, and a value of six represents one second of precision. For query and reporting purposes, the sensor reading is treated as having occurred during the entire period represented by the Presentation time and the precision. For example, if I am looking for a sensor reading from a particular sensor that occurred at 12:00:00 with a time precision value of 6 (meaning 1 second), the system will return all sensor readings that occurred between 11:59:59.5 and 12:00:00.5. The system will consider a sensor reading to have occurred within that interval if the combination of the presentation time and the time precision of the reading fall within that interval. For example, a sensor reading with a Presentation Time of 12:00:00.525 with a time precision of 5 (meaning 1/10 second precision) will fall into the range specified.

***NOTE**: In addition to precision, the level of confidence associated with the presentation time should be captured as additional information quality metadata (see next section). This level of confidence could vary based on the source of the presentation time and various historical/statistical factors.*

***Time-Related Information Quality Metadata*:** The time metadata associated with each sensor reading includes all of the times listed above along with the information quality metadata and statistics listed below:

- Accuracy/Reliability of the Acquisition Time Data – this information would typically be reported by the sensor as part of its metadata stream. If acquisition time is not provided by the sensor then this metadata will not be present.
- Accuracy/Reliability of the Ingestion Time Data – this information is determined by the ingestion component used to bring the sensor data into the system.
- Accuracy/Reliability (confidence) of the Presentation Time Data – this information is determined by the ingestion component based upon whatever algorithm/conversions are used to determine the presentation time.
- Latency statistics:
    - Delta between Acquisition Time and Ingestion Time, if known.
    - Average delta (moving average).
    - Deviation of current delta from moving average.
    - Deviation of current delta from expected delta.
- Acquisition time statistics:
    - Delta between current acquisition time and previous acquisition time.
    - Average delta (moving average).
    - Deviation of current delta from moving average.
    - Deviation of current delta from expected delta.
- Ingestion time statistics:
    - Delta between current ingestion time and previous ingestion time.

- Average delta (moving average).
- Deviation of current delta from moving average.
- Deviation of current delta from expected delta.

The ingestion component is responsible for tracking this quality metadata and also providing a presentation time for the sensor data. The presentation time is a time including a measure of precision for which the ingestion component is confident that the sensor reading was captured. A confidence metric is reported as part of the information quality metadata associated with the presentation time that reflects the accuracy/reliability of the presentation time span.

***Ingestion End-Point Component Responsibilities*:** It is the responsibility of each sensor ingestion end-point to timestamp the sensor data with the time at which the data was received by the system. This point is considered the ingestion time. The ingestion time may be used to determine the presentation time for the data.

As mentioned above, it is also the responsibility of the ingestion end-point to determine the presentation time that will be used to index the sensor data received.

Time Storage Formats

***Acquisition Time Storage Format*:** The storage format for the acquisition time reported by the sensor is dependent upon the sensor and will be maintained in the same format as that reported by the sensor. This will allow time- based queries for data from a single sensor feed to be performed using the native time format of the sensor.

NOTE*: Any queries that include data spanning more than one sensor feed must use the presentation time.*

Ingestion and Presentation External Time Representation: The ISO 8601:2004 standard for representing UTC time will be used for all external representations of ingestion time and presentation time. Under this standard there are several formats that can be used, however, to simplify display and comparison, we will adopt the following format as our standard: “YYYY-MM-DDThh:mm:ss.ssssssZ.”

Decomposing this string: “YYYY” represents the 4 digit year, “MM” represents the two digit month, “DD” represents the two digit day, ”T” separates the date portion of the time from the hours/minutes/seconds portion, “hh” represents the two digit hour (0 to 23), “mm” represents the two digit minute, “ss.ssssss” represents the second including fractional seconds to nanosecond precision, and the “Z” at the end indicates the time is represented as UTC.

Precision can be indicated by the number of digits used to represent the time value. For example: “1970-01-01T00:00:00.000000” indicates a precision of 1 nanosecond, “1970-01-01T00:00:00” indicates a precision of 1 second, and “1970-01-01T00:00” represents a precision of 1 minute.

Ingestion and Presentation Internal Time Representation: To conserve storage space and simplify time comparisons, internally all-time values will be stored using the Microsoft Windows FILETIME structure. This structure stores a 64-bit signed integer value stamp for which the value 0 represents the beginning of the year 1601 on the Gregorian calendar and for which each integer value represents a 100 nanosecond time interval. This provides a date range of approximately 58,000 years (roughly 27000BC to 30000AD). As defined by Microsoft, the value stored in this structure represents UTC time which means the time value is adjusted whenever leap seconds are used to keep UTC time in sync with the earth's rotation. From a persistent surveillance perspective this introduces the potential for temporal discontinuities within the data (i.e. any time a leap second is added, it will appear that time is repeating itself for that one second – if a leap second is removed, it appear as a one second "hole" in time – see the section below on leap seconds for more details).

In order to avoid these temporal discontinuities, the PSSA definition of this value will be based on the international atomic time scale known as TAI. The PSSA SDK will provide functions to convert UTC and GPS time to and from TAI by adding/subtracting the appropriate number of leap seconds.

***NOTE**: The PSSA database will need to maintain a table to track changes to the offset between TAI and UTC (TAI is currently 34 seconds ahead of UTC). The offset between TAI and GPS time is fixed at +19 seconds (TAI is 19 seconds ahead of GPS time).*

Precision information regarding the time value is stored internally along with the time value. The precision value is a floating point value that represents the base 10 logarithm of the precision of the time value in nanoseconds (e.g. for one nanosecond of precision the value is 0, for one millisecond of precision the value is 3, for one second of precision the value is 6, etc.)

Leap Seconds: Due to irregularities in the Earth's rotation, the addition or subtraction of a leap second is occasionally needed to bring UTC time into alignment with the true solar time (a.k.a. UT1). The International Earth Rotation and Reference Systems Service (IERS) determine when a leap second should be inserted (or removed). On most computer systems, the addition of a leap second is accommodated by setting the system clock back by 1 second when the leap second is added – this typically occurs when the computer receives an update from an NTP server. As a result, the possibility exists that an event occurring just before the clock is set back and an event occurring just after the clock is set back will appear to have occurred in a different order. This situation is referred to as a temporal discontinuity.

Time discontinuities can also occur as a result of manual adjustments to the reference clock being used by the system as well.

The PSSA SDK in conjunction with the PSSA Time Server will handle time discontinuities associated with leap seconds and will provide the correct time to ingestion components. The ingestion components are responsible for handling any time discontinuities associated with acquisition time data provided by the sensor.

***NOTE**: Sensors that use GPS time should not experience this condition.*

PSSA Time Server: As part of the implementation of the PSS physical architecture, an NTP Time server will be specified to provide time synchronization for all of the hardware components of the PSS system. The NTP time server is a hardware platform that synchronizes at regular intervals to the WWVB (the Standard Time and Frequency Station, 60KHz, Fort Collins, Colorado) long-wave radio broadcast and/or the GPS timestamp provided by the GPS satellite network. This service will ensure that the ingestion times recorded with the sensor data across the PSSA are accurate to within +/- 10 milliseconds of actual UTC time.

If more accurate time synchronization is desired, a PTP (– IEEE 1588) time server can be installed and compatible hardware/software installed on each of the hardware components of the system to achieve time synchronization to within 1 microsecond of UTC time.

Time-Based Queries: There are several forms that a time-based query for sensor data can take. These are enumerated below:

1. Retrieve the sensor readings captured between time t1 and time t2.
2. Retrieve the latest sensor reading.
3. Retrieve the sensor reading closest to time t1.
4. Retrieve X number of sensor readings captured after time t1, if X = 0, all sensor readings after t2 are returned. Sensor readings are returned in chronological order.
5. Retrieve X number of sensor readings captured prior to time t2. If X = 0, all sensor readings prior to t2 are returned. Sensor readings are returned in chronological order.

The presentation start time and end time are based on the combination of the presentation time and the presentation time precision value that are stored with the sensor reading. The presentation time stored with the sensor reading is considered to be the midpoint of the range specified by the precision value. For example, if the precision value represents a precision of one second and the presentation time is 12:00:00.000000 then the presentation start time is 11:59:59.500000 and the presentation end time is 12:00:00.500000. These values are used to determine what data should be returned as the result of a query. To determine whether or not a sensor reading should be returned, the PSS retrieval components apply the following rules for each scenario listed above:

1. Returns the reading if presentation start time > t1 or presentation end time < t2
2. Returns the sensor reading with the latest presentation end time.
3. Returns the sensor reading for which the midpoint of the presentation start time and end time is closest to t1
4. Returns earliest X sensor readings for which the presentation end time > t1
5. Returns latest X sensor readings for which the presentation start time < t2

The following examples demonstrate how these queries could be performed using WFS Simple service requests (WFS Simple is an Open Geospatial Consortium standard for web-based spatial/temporal queries).

NOTE*: Times are specified according to the ISO 8601:2004 standard.*

Example 1: *Get all webcam sensor readings located within the bounding box region delineated by 39.98N, 83.00W and 40.02N, 83.01W between 15:18:22.0183600 EST and 15:33:22.0183600 EST on February 25, 2010.*

```
https://qbasesensorplex.com/sensor/webcams/wfs?service=WFSSIMPLE&R
EQUEST=GetFeature&BBOX=-83.00,39.98,-83.01,40.02&TIME=2010-02-
25T15:18:22.0183600-05:00/2010-02-25T15:33:22.0183600-05:00
```

This example uses the TIME parameter of the WFS Simple service request to specify the 15 minute time interval in which we are interested.

Example 2: *Get the last 30 sensor readings for webcam cmh042.*

```
https://qbasesensorplex.com/sensor/webcam/cmh042/wfs?service=WFSSI
MPLE&REQUEST=GetFeature&TIME=/2020-02-
25T15:33:22.000Z&MAXFEATURES=30
```

Since the TIME parameter specifies only an end time and that end time is some time in the future, our WFS Simple service request handler interprets this request to be for the latest 30 sensor readings (as specified by the MAXFEATURES parameter).

Example 3: *Get the sensor reading for webcam cmh042 that is closest to the time 15:18:22.0183600 EST on February 25, 2010*

```
https://qbasesensorplex.com/sensor/webcam/wfs?service=WFSSIMPLE&RE
QUEST=GetFeature&TIME=2010-02-25T15:18:22.0183600-05:00/2010-02-
25T15:18:22.0183600-05:00&MAXFEATURES=1
```

Since the start time and the end time in the TIME parameter are set to the same value, our WFS Simple service request handler interprets this request to be for the sensor reading closest to the specified time. It is up to the application to decide, based upon the presentation time of the sensor reading returned, whether or not it is close enough to the specified time to be used. If MAXFEATURES is greater than 1, then the features closest to the specified time are returned in chronological order (earliest to latest) up to the number specified by MAXFEATURES.

Example 4: Get the first 60 sensor readings for webcam cmh042 that occurred after 2010-02-25T15:18:22 EST on February 25th, 2010.

```
https://qbasesensorplex.com/sensor/webcam/cmh042/wfs?service=WFSSI
MPLE&REQUEST=GetFeature&TIME=2010-02-25T15:33:22.000-
05:00&MAXFEATURES=60
```

Since the TIME parameter specifies only a start time, our WFS Simple service request handler interprets this request to be for the first 60 sensor readings (as specified by the MAXFEATURES parameter) following the time specified by the TIME parameter.

Example 5: *Get the last 20 sensor readings for webcam cmh042 that occurred prior to 2010-02-25T15:18:22 EST on February 25th, 2010.*

https://qbasesensorplex.com/sensor/webcam/cmh042/wfs?service=WFSSIMPLE&REQUEST=GetFeature&TIME=/2010-02-25T15:33:22.000-05:00&MAXFEATURES=20

Since the TIME parameter specifies only an end time, our WFS Simple service request handler interprets this request to be for the latest 20 sensor readings (as specified by the MAXFEATURES parameter) prior to the time specified by the TIME parameter.

Spatial Synchronization: Spatial synchronization is just as important as temporal synchronization in a layered sensing environment. Knowing where a sensor is located and the spatial dimensions of its field of view are critical to providing a comprehensive situational awareness and exploitation capability. For example, an image analyst viewing data from a wide area persistent surveillance sensor feed could task ground-based cameras to zoom in on a suspect vehicle that is being tracked using the wide area sensor data. This functionality will only work effectively if both the wide area sensor and the ground camera are or can be synchronized to a common spatial reference system.

At a minimum, for every sensor reading, the PSS system needs to know the two-dimensional area of coverage of the sensor reading (e.g. ground footprint) in order to index it and spatially relate it to other sensor readings. In many cases, the spatial metadata provided by the sensor will not include the area of coverage. In these cases, it is the responsibility of the ingestion component to determine the area of coverage based on known information about the sensor and the metadata provided by the sensor. The calculation to determine the area of coverage will vary from sensor to sensor. However, in general it can be determined from the external location and orientation of the sensor coupled with knowledge about the internal configuration of the sensor (e.g. CCD size and orientation, lens focal length and distortion characteristics, etc.).

NOTE*: It is important for the ingestion component to determine the area of coverage as quickly as possible, particularly on high frame rate data feeds. In general, it is better to compute an approximate area of coverage using a simple calculation than to fall behind the data feed using a more complex calculation. If a more accurate area of coverage is required, it should be deferred to an exploitation component.*

If the location, orientation and zoom level of a camera sensor is fixed, then the footprint of the sensor will be fixed as well, so it does not need to be recalculated with every sensor reading. For fixed camera sensors that have Pan, Tilt, and Zoom (PTZ) capability, the PTZ values must be included in the metadata so that the spatial coverage area can be calculated. For moving sensors, the sensor location (longitude, latitude, altitude) is constantly changing and therefore must be included in the sensor metadata.

In addition, the exterior orientation (omega, phi, kappa) of the sensor may change as a result of changes in the attitude (i.e. roll, pitch, yaw) of the platform or gimbal on which the sensor is mounted. In some cases for camera-based sensors, the focal length of the lens (i.e. zoom) may be variable as well. All of these parameters, if they change, must be included in the metadata associated with the sensor reading. For non-image based sensors there may be similar variable parameters that could affect the coverage area of the sensor (e.g. sensitivity settings). If this is the case, then those parameters must be included in the sensor metadata so that the area of coverage can be calculated.

For moving sensors or sensors with a variable coverage area, it is the responsibility of the ingestion component to quickly establish an initial coverage area (ground footprint) for the sensor reading unless this information is already provided as metadata by the acquisition hardware/software.

***NOTE**: Downstream exploitation algorithms can further refine the coverage area, if needed. It is also the responsibility of the ingestion component to provide an estimate of the accuracy of the coverage area as part of the Information Quality Metadata.*

Location Related Information Quality Metadata: The metadata used to determine the initial coverage area of the sensor should be evaluated to determine the accuracy of that coverage area. For instance, if GPS data is used to locate the platform, the dilution of precision (DOP) of the GPS data should be captured as information quality metadata. Oftentimes, the DOP is provided by the GPS receiver itself as part of the metadata. However, if the receiver does not provide this information, a theoretical DOP for any given time and location can be predicted using a GPS Satellite Almanac (ephemeris data) and assumptions regarding which satellites are visible to the receiver. The DOP value can be used to determine whether or not to use the GPS data. A general rule of thumb is to take the published accuracy of the GPS device and multiply it by the DOP Value to get a maximum error for the GPS reading. For example, if the accuracy of the GPS device is +/- 3m and the DOP value is 3, then the actual location is within 9m of the GPS reading (3m x DOP of 3 = 9m). For reference, the table below lists ratings of DOP values from two different Internet sources.

Table D-1: Ratings of DOP Values

DOP Value0	DOP Value0	Rating	Description
1	1	Ideal	This rating is the highest possible confidence level to be used for applications demanding the highest possible precision at all times.
2-3	1-2	Excellent	At this confidence level, positional measurements are considered accurate enough to meet all but the most sensitive applications.
4-6	2-5	Good	This rating represents a level that marks the minimum appropriate for making business decisions. Positional measurements could be used to make reliable in-route navigation suggestions to the user.
7-8	5-10	Moderate	Positional measurements could be used for calculations, but the fix quality could still be improved. A more open view of the sky is recommended.
9-20	10-20	Fair	This rating represents a low confidence level. Positional measurements should be discarded or used only to indicate a very rough estimate of the current location.

These descriptors should be viewed as guidelines since the accuracy level of GPS devices sometimes can vary widely (e.g. if a GPS device has a published accuracy of 6m, than even a DOP of 1 will only ensure accuracy to within 6m).

Some GPS devices support differential GPS which can significantly increase the accuracy of the GPS reading (typically better than one meter horizontal accuracy within 100 nautical miles of a DGPS transmitting site).

For spatial measurements, we are primarily concerned with the Positional Dilution of Precision (PDOP) and the Horizontal Dilution of Precision (HDOP). HDOP represents the dilution of precision in 2 dimensional space (latitude/longitude) and PDOP represents the dilution of precision in 3 dimensional space (latitude/longitude/altitude (LLA)).

Normalization of Spatial Data: In order to be able to index, query and compare spatial metadata from a variety of sensors, it is important to store the spatial metadata associated with a sensor reading using a common spatial frame of reference. The common spatial coordinate reference system used by the PSSA is the LLA coordinate system based on the WGS-84 geodetic datum (EPSG:4326) using meters as the unit of measure. Minimally, the spatial data required for PSS (e.g. area of coverage) is indexed in two-dimensions to simplify index and query processing.

If desired, three dimensional indexes including altitude information can also be stored and indexed to support more sophisticated analytics and exploitation algorithms. Any distance measures (including altitude) stored in the database should be stored in meters to ensure consistency between data sources.

NOTE*: Image data may be projected or warped to a coordinate system other than the WGS-84 coordinate system mentioned above. If this is the case the Spatial Reference System for the image should be stored along with the affine transform used to locate and orient the image within the spatial reference system.*

LIST OF DEFINITIONS

Event Collaboration	The application integration pattern upon which the PSSA is based. Processing components are loosely coupled using a messaging model in which data sources publish events and processing components subscribe to specific event topics.
Exploitation Algorithm	An algorithm that operates on the sensor data to enhance the usefulness of the data in some way. For example, an exploitation algorithm could measure and/or improve the quality of the sensor data or it could create new data by analyzing the sensor data and comparing or combining it with data captured previously or with data captured by other sensors.
Layered Sensing	The use of multiple types ("layers") of remote sensing technology to simultaneously provide different views of overlapping geographic regions and disparate types of information, collected at different times.
Metadata	Any contextual information that relates to a specific sensor reading. This could include time, location, sensor parameters, environmental conditions, or anything else related to the sensor reading that is needed to fully exploit the data captured by the sensor. Metadata can come from the sensor or it may be generated by components of the PSS system as the sensor data is being processed.
Observation	See "Sensor Reading"
Persistent Surveillance	The use of remote sensing technology to continuously monitor a geographic region.
PSSA Cloud or PSS Cloud	This is the messaging system that forms the core of the PSSA. Through the use of the PSSA SDK, it transparently provides the connectivity between various components of the PSSA system.
Sensor Data	The data (sensor reading and related metadata) associated with a specific sensor including new metadata derived from the sensor reading and/or metadata.
Sensor Reading	The data captured by the sensor at a given point in time and represents the actual bits generated by the sensor as it converts the sensed data into digital form.

LIST OF ACRONYMS

AFRL	Air Force Research Laboratories
AMASE	AVTAS Multi-Agent Simulation Environment
API	Application Programming Interface
ArcGIS	Not a true acronym, this is the registered name of one of ESRI's (q.v.) products. See also GIS.
ASIC	Application Specific Integrated Circuits
AU	Alliance University
AVTAS	Aerospace Vehicles Technology Assessment and Simulation
BMP	Bitmap graphics file
CIMIS	California Immigration Management Information Service
CORS	Continuously Operating Reference Stations
CoT	Cursor on Target
COTS	Commercial, off-the-shelf product
CPU	Central Processing Unit
CUAHSI	Consortium of Universities for the Advancement of Hydrologic Sciences
CWLR	Cibola Wild Life Refuge
DCGS	Distributed Common Ground System
DGPS	Differential GPS
DIB	DCGS Integrated Backbone
DSO	Directory Service Observer
DSS	Data Storage Service
DoD	Department of Defense
DOP	Dilution of Precision
EO	Electro-Optical
ESI	External Service Initiator
ESRI	Environmental Systems Research Institute
ET	Evapotranspiration
EVI	Enhanced Vegetation Index
FFMPEG	Fast-Forward MPEG
FPGA	Field-Programmable Gate Array
FPS	Frames per Second
GIS	Geographic Information System
GPS	Geographic Positioning System
GPU	Graphics Processing Unit

GeoRSS	Geographically Encoded Objects Really Simple Syndication
GOTS	Government Off-The-Shelf
GP	Great Plains
HTTP	Hypertext Transport Protocol
IED	Improvised Explosive Device
IERS	International Earth Rotation & Reference Systems
IMU	Inertial Measurement Unit
IP	Internet Protocol
ISR	Intelligence Surveillance & Reconnaissance
JD	Julian Date
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
LANDSAT	Land Remote Sensing Satellite System
LCR	Lower Colorado Region
LCRAS	Lower Colorado River Accounting System
LGPL	Lesser General Public License (GNU),
LLA	Latitude/Longitude/Altitude
MATLAB	Matrix Laboratory
MJD	Modified Julian Date
MJPEG	Motion JPEG. See also JPEG.
MODIS	Moderate Resolution Imaging Spectro-Radiometer
MPEG	Moving Pictures Expert Group
MSRS	Media Storage Retrieval Service
NDVI	Normalized Deviation Vegetation Index
NFS	Network File System
NIST	National Institute of Standards and Technology
NOAA	National Oceanic and. Atmospheric Administration
NTP	Network Time Protocol
ODA	Ohio Department of Agriculture
OGC	Open Geospatial Consortium Open
LST	Open Layered Sensing Testbed
PCIT	PSSA Components & Other Infrastructure Tracker
PDOP	Positional Dilution of Precision
PSS	Persistent Sensor Storage (used interchangeably with PSSA)
PSSA	Persistent Sensor Storage Architecture
PTP	Precision Time Protocol

PTZ	Pan, Tilt, Zoon
PVID	Palo Verde Irrigation District
QDT	Qbase Data Transformer
ReSET	Remote Sensing EvapoTranspiration
RSS	Really Simple Syndication or Rich Site Summary
RSST	Registered Sensor Status Tracker
RTSP	Real-Time Streaming Protocol
RVDS	RTSP Video Dissemination Service
SAR	Satellite Synthetic Radar
SAS	Sensor Alert Service
SAS/SATA	Serial Attached SCSI /Serial Advanced Technology Attachment
SATE	Summer at the Edge
SCS	Remote Launcher Service
SDK	Software Development Kit
SEBAL	Surface Energy Balance Algorithm for Land
SFVM	Sensor Feed Visualization or Manipulation
SPS	Sensor Planning Service
SOS	Sensor Observation Service
S-SEBI	Simplified-Surface Energy Balance Index
SSIM	Structural Similarity
SWE	Sensor Web Enablement
SWIG	Simplified Wrapper Interface Generator
TAI	International Atomic Time
TCP	Transmission Control Protocol
TESV	Text Event Stream Visualization
UAV	Unmanned Aerial Vehicle
URL	Uniform Resource Locator
USBR	US Bureau of Reclamation
USDA	US Department of Agriculture
USGS	US Geological Survey
UTC	Universal Time Coordinated
VLC	Video LAN Client
W3	Worldwide Web Consortium
WAPS	Wide Area Persistent Surveillance
WBI-ICC	Wright Brothers Institute - Innovation and Collaboration Center
WCS	Web Coverage Service (OGC Standard)

WCF	Windows Communication Foundation
WFS	Web Feature Service (OGC Standard)
WMS	Web Mapping Service (OGC Standard)
WNS	Web Notification Service
WWVB	Not a true acronym, this refers to the Standard Time and Frequency Station in Fort Collins, Colorado, long-wave 60KHz radio broadcast.
WSU	Wright State University
XML	eXtensible Markup Language
XMPP	eXtensible Messaging & Presence Protocol
XSLT	eXtensible Stylesheet Language Transformations
YATE	Year at the Edge