PGS Core Architecture
Version 1.0

Custodian: Cecilia S. Cheng
Rajesh R. Patel

Email: csc@mipl.jpl.nasa.gov
rrp@mipl.jpl.nasa.gov

Date: December 16, 2004

Document Change Log

Date	Name	Description
01/22/2004	R. Patel, C. Cheng	• Draft Release
12/16/2004	R. Patel, C. Cheng, L.Ly	• Added modification history appendix • Modified scheduling section to include sequencing info • Added design decision section • Moved added benefits section to beginning of the document • Added requirements section • Integrated redlines from Document Review with L. Preheim • Updated the diagrams • Changed all references from TPS (Telemetry Processing Subsystem) to PGS (Product Generation Subsystem)

Table of Contents

1. Scope

The purpose of this document is to describe in detail the design of the Multi-mission Image Processing Laboratory's (MIPL) Product Generation Subsystem (PGS).

PGS was previously known as the Telemetry Processing Subsystem (TPS). However, as we adapt to new missions, there is a need to process more than just telemetry (SFDU's – Standard Formatted Data Units). The data can be in CFDP format, which is file based. Therefore, we have renamed TPS to PGS, to better describe the function of the subsystem.

As of the publication date of this document, the 820-061 name has not been updated and is still 601.1 Real-time Processing.

2. Introduction

In the past, a typical telemetry process in MIPL involves obtaining raw telemetry data, deciphering and assembling telemetry into products and shipping products to customers. Some projects perform additional tasks such as cataloging products, visually displaying products, and generating and shipping telemetry processing reports. Of all the tasks mentioned above, deciphering and assembling telemetry is the only project specific task.

The intention of PGS is to bundle the common tasks into an automated and fail safe framework for processing of telemetry, while allowing plug-in of customized project specific telemetry processing code.

The organization of the document is as follows:

- Section 3 describes the reasons for moving to this new redesign
- Section 4 describes the driving requirements behind the architecture
- Section 5 describes the major components of the PGS architecture
- Section 6 describes the failure and recovery design.

3. Benefits of PGS

This section outlines the benefits of PGS over the current system and the added benefits of the new design.

3.1. Drawbacks of the Current System

The current PGS system has a set of centralized (i.e. only one running instance available per system) services that supports multiple missions. One such service is the TLM_Factory (telemetry factor). The TLM_Factory is responsible for managing streams. A stream is composed of three distributed services: a logger, a TDSIF (Telemetry Data Source Interface) and an Instrument Constructor (project specific telemetry processing component). Streams may be started by users and registered with the factory for future use (thus pre-started streams) or stared by the factory as needed. Components of a stream that are started by the factory are collocated in a single process (thus, collocated stream). There are several inherent problems with this design.

- The first problem has to do with user permissions. Each mission has its own processing environment. The processing environment may include working mission storage where the Instrument Constructor writes its product files and utilizes the MIPL DBMS for cataloging product files and other related ancillary information. Since the centralized services are shared by multiple missions, the user that starts the stream factory needs to have access to all projects' processing environment. This is because the collocated streams, which is started by the stream factory, needs to have access to the working environment of each project. Since the factory does not perform any user authentication, an unauthorized user may end up utilizing a pre-spawned stream. The current solution is to have a super user (one that is member of all project specific user groups) to start the centralized processes. However, with this approach, the above mentioned problem with pre-started streams still exists.

- Second, the current system is not very robust. For example, if any component of the stream fails, then the stream fails. However, the stream failure is not noticed by the factory until its next attempt to use the same stream. In addition, although exception-raising is in place, exceptions are not handled at all appropriate places.

- The third drawback has to do with upgrades and server migration. With one centralized core, upgrades or server migration means coordination among all projects for down time. As simple as it sounds, this is a very problematic. For one, each project may have its own critical period of downlink and processing, which delays upgrades or migrations. Secondly, not all projects may be willing for upgrade as they may have gone in final phase of software validation and verification. Thus, the idea of having one centralized core server all projects seems a little risky.

3.2. Overcoming the Drawbacks

The new design overcomes the drawbacks of the current system as follows:

- User permission problem is overcome as discussed in Section 5.2.1
- System robustness is achieved through design and implementation practices
- Server upgrades and migration issue is solved by having each project run their own set of core services. This also allows projects to customize PGS to fit project's needs.

3.3. Added Benefits

Besides overcoming the problems mentioned in the previous section, the new design provides the following benefits.

- PGS provides load balancing and fail over when more than one Service Factories that can handle the request exists.
- The use of the Persistent Store allows safe restart of servers and requests while providing a means of bookkeeping of user activities and processing history.
- The monitoring hierarchy (that is, the Controller monitoring the Service Factories and each Service Factory monitoring its requests) distributes error detection work among appropriate services while the reporting mechanism enables quick response, thus reducing downtime.
- The scheduling service will minimize user presence.
- The scheduling service, along with error detection and reporting, provides groundwork for automation of request processing.

4. Key Requirements

This section lists key requirements of the PGS system and points to design components and decision that fulfills the requirements

4.1. Scalability

The system must be scalable. Scalability means the ability to adjust to handle and process various data loads. The PGS design fulfills the scalability requirement by a) dividing responsibilities among servers, and b) instantiating instances of streams as needed.

4.2. Data Retrieval

The PGS shall receive data from the TDS, DOM (Distributed Object Manager) or FEI (File Exchange Interface). This is satisfied by the implementation of the Data Source Interface that will connect to the correct data source for data. Different customers may have different specifications on the data source, which will be made transparent to them.

4.3. Product Creation

The PGS shall create products based on each mission's requirements and write them to the Working Mission Storage. This is satisfied by the mission-specific instrument constructor's process.

4.4. Log

The PGS shall capture initialization, error, failure, and anomaly information to the system log. It shall detect and log system events and errors. This is satisfied by the logger interface.

4.5. Reliability

The PGS will be highly reliable and well documented with sufficient standardization of usage. This is fulfilled by analyzing the possible failure points in the system and addressing the issues and providing alternative solutions. (See Section 6 for details.)

4.6. Security

The PGS shall provide a means to authorize and authenticate users. This is achieved by requiring each user to provide an assigned user ID and a password when interacting with PGS by means of secure communications (namely SSL) among the components of the PGS systems.

5. PGS Core Components

Figure 1 depicts the high level interaction among the three core components of the PGS:

- The Persistent Store
- The Service Factory (SF) and
- The Controller.

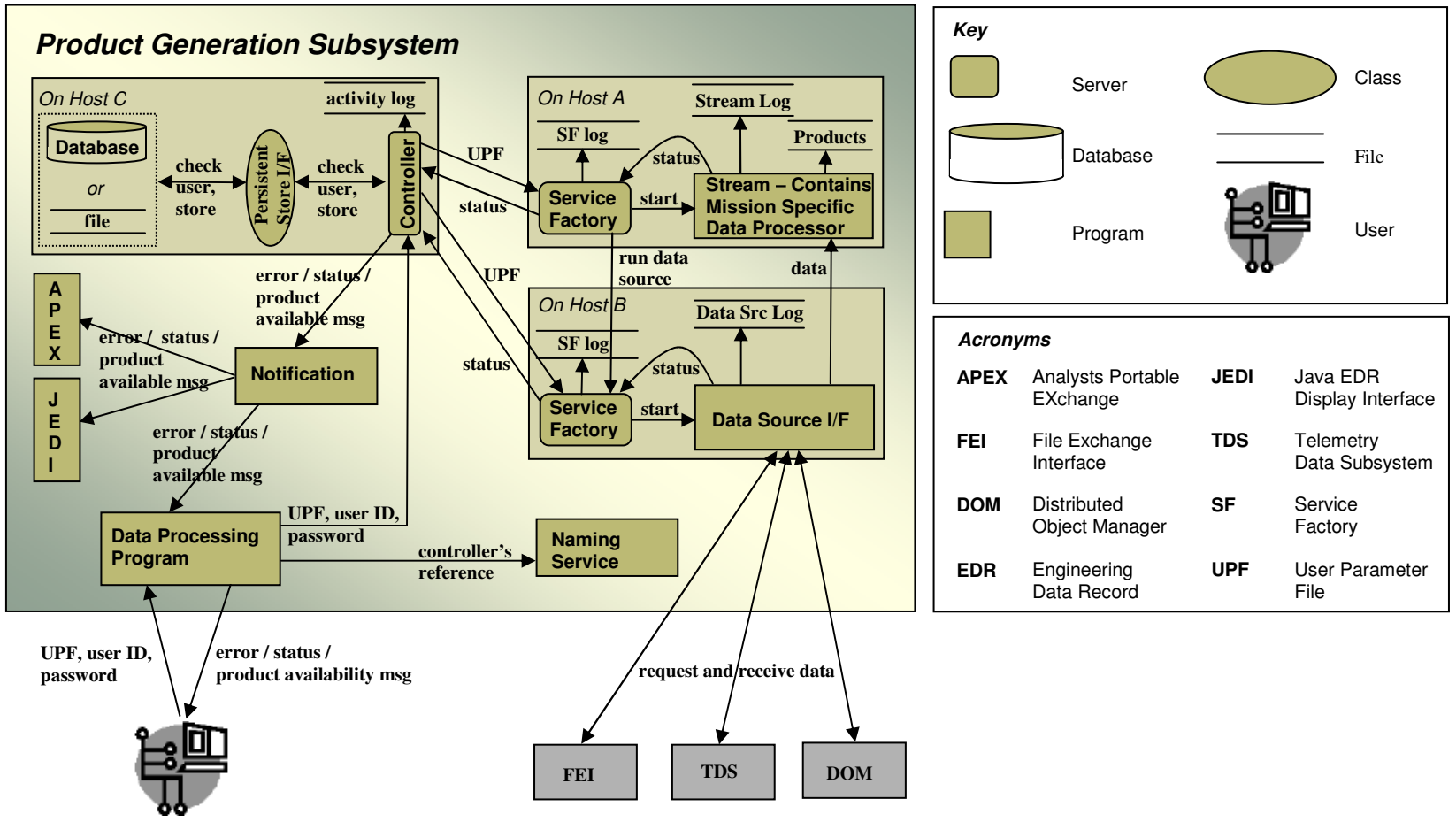

Figure 1. PGS Component Interaction

As one can notice from the figure, the Controller is the only component of the PGS system that is exposed to users. The user makes a processing request (or a request for short) to the Controller using the Client software component of the PGS system. A request commands the Controller to execute appropriate telemetry processing software on the requested range of data. All the parameters necessary for processing the given request are passed to the Controller. Then, the Controller finds an appropriate SF and delegates the request to it.

Before diving into the details, let's get familiar with the high level functionality of each of these components.

- ***Naming Service***
  The Naming Service enables registration and discovery of different components of the PGS system. This service is used for publishing a reference to the Controller component of the PGS system. Note that is an out of the box service.
- ***Notification Service***
  The Notification Service may be utilized for event notification, such as component status. This is also an out of the box service.
- ***Persistent Store***
  The PGS utilizes the Persistent Store for recording processing information vital to recovery after failure and for storing server configuration and user information. This also serves as bookkeeping of users' activities.
- ***Client***
  The Client encapsulates the communication between external users and the Controller.
- ***Controller***
  The Controller is the gateway to the PGS. It is responsible for accepting and processing users' requests and for monitoring overall health and safety of the system.
- ***Service Factory (SF)***
  The SF is responsible for starting components necessary for processing telemetry. There are two components started by SF: the PGS Data Source and the PGS Stream Service (stream for short). The Data Source encapsulates the low level communication between the stream and the under laying source of raw telemetry data (either the Telemetry Data System (TDS) or the Distributed Object Manager (DOM) service). The stream is responsible for deciphering telemetry into products. Readers will realize that the most practical SF configuration is to have multiple SFs running on different hosts.

5.1. Controller

The main responsibilities of the Controller are authenticating and authorizing users, dispatching data processing requests to the SF, perform load balancing between SF's, monitoring the health of SFs and acting as a middleware between the Persistent Store and reset of the PGS components.

5.1.1. Interaction with the Persistent Store

The Controller interacts with the Persistent Store through interfaces defined in Section 5.3. This design decouples the Controller from the implementation of the Persistent Store. However, readers should note that the MIPL database infrastructure is the primary and default Persistent Store for the PGS. For ease, the rest of the discussion is based on the MIPL database infrastructure being the Persistent Store.

Only designated personnel, i.e., the PGS administrators, have access to the Persistent Store. Since the Controller is the only component of the PGS that has direct access to the Persistent Store, the Controller has to be started by the administrator.

There are several reasons to only allow Controller to have direct access to the Persistent Store. The first reason is to reduce the number of concurrent connections. Secondly, allowing multiple concurrent direct user access to the database increases chances of data corruption. Third, if necessary, only the Controller needs to refresh any credentials needed to keep the database connection opened. The Controller utilizes the supplied user ID and password to authenticate and authorize as described in Section 5.1.2.

Later sections discuss the design and mechanism used to ensure robustness in PGS in the event of component failures, including database failures.

5.1.2. Authentication and Authorization Scheme

As depicted in Figure 1, the Client (either a user or a piece of automated software) makes a processing request directly to the Controller. A processing request is a command given to the Controller to start the components necessary to process data as indicated by the User Parameters File (UPF). Along with the UPF, the identity, (i.e., the assigned user ID) and the one way hashed password (OWHP) of the user, is also supplied. The Controller authenticates the user by using the User

Personal and Configuration Information Interface (Section 5.3.1). The sequence diagram in Figure 2 gives details about the authentication and authorization process. Wary readers should note that secure communications will be utilized among different components of PGS.

5.1.3. Request Dispatching and Load Balancing

Once the requesting user passes the authentication and authorization process, the Controller executes the request by asking appropriate SFs to start the necessary components. The Controller utilizes the Mission Service Factory Configuration Interface (Section 5.3.4), and the Service Factory Interface (Section 5.3.5[1]) to locate appropriate SFs. Once the SFs are located, the Controller selects the SF with the least load as the most qualified SF for starting the component. The order in which components are started is shown in Figure 3.

The host machine where a service is to be started (service host) may be specified through UPF parameters (for example, the UPF parameter TDSIF_HOST specifies the host where the TDSIF is to be started). Although this mechanism is provided for flexibility, it is less robust when compared to the Controller's host selection process. This is discussed in more detail in Section 6.1.

5.1.4. Monitoring the Service Factory (SF)

Periodically, the Controller pings each SF in its reference list to ensure that it is running. If a ping fails, the Controller finds all active requests that belong to that SF by using the Requests Interface (Section 5.3.2). It then notifies requesting users of each request and the administrator of the SF (discussed in Section 5.) about the failure and removes the SF from its list (see Figure 4 for details).

5.1.5. Scheduling and Sequencing Service

The scheduling capability of the PGS system will allow the release of requests on a specific future time. The scheduling service utilizes the Schedule Interface to store and retrieve scheduling information.

The sequencing capability will allow the release of processes, pre and post data processing. This information is manipulated using the Sequence Interface.

[1] The SF Info Table is a Persistent Store used by the Controller at start up to obtain references to all SFs in the system as depicted in Figure 4.

The scheduling and sequencing is work in progress.

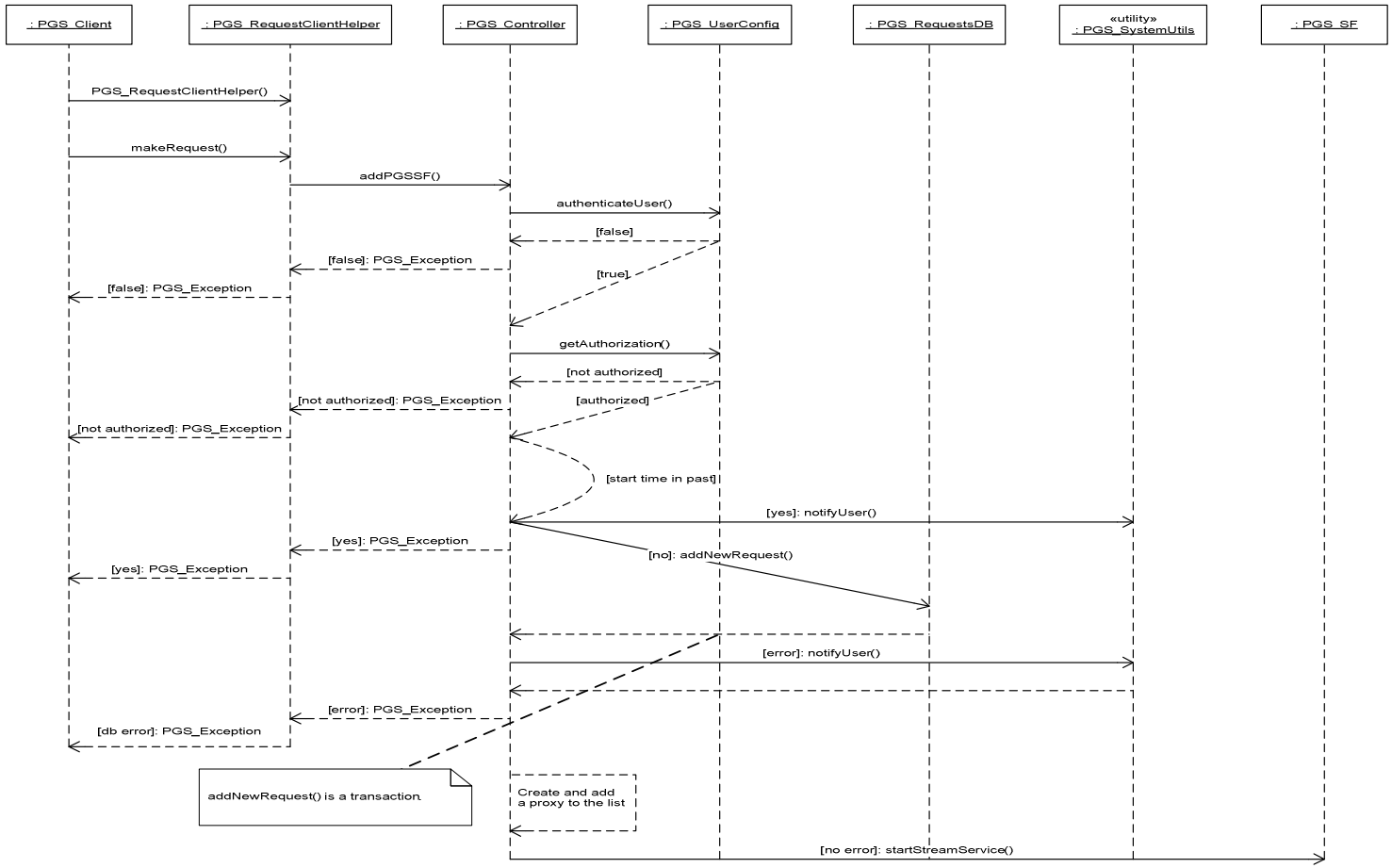

Figure 2. User authentication and authorization

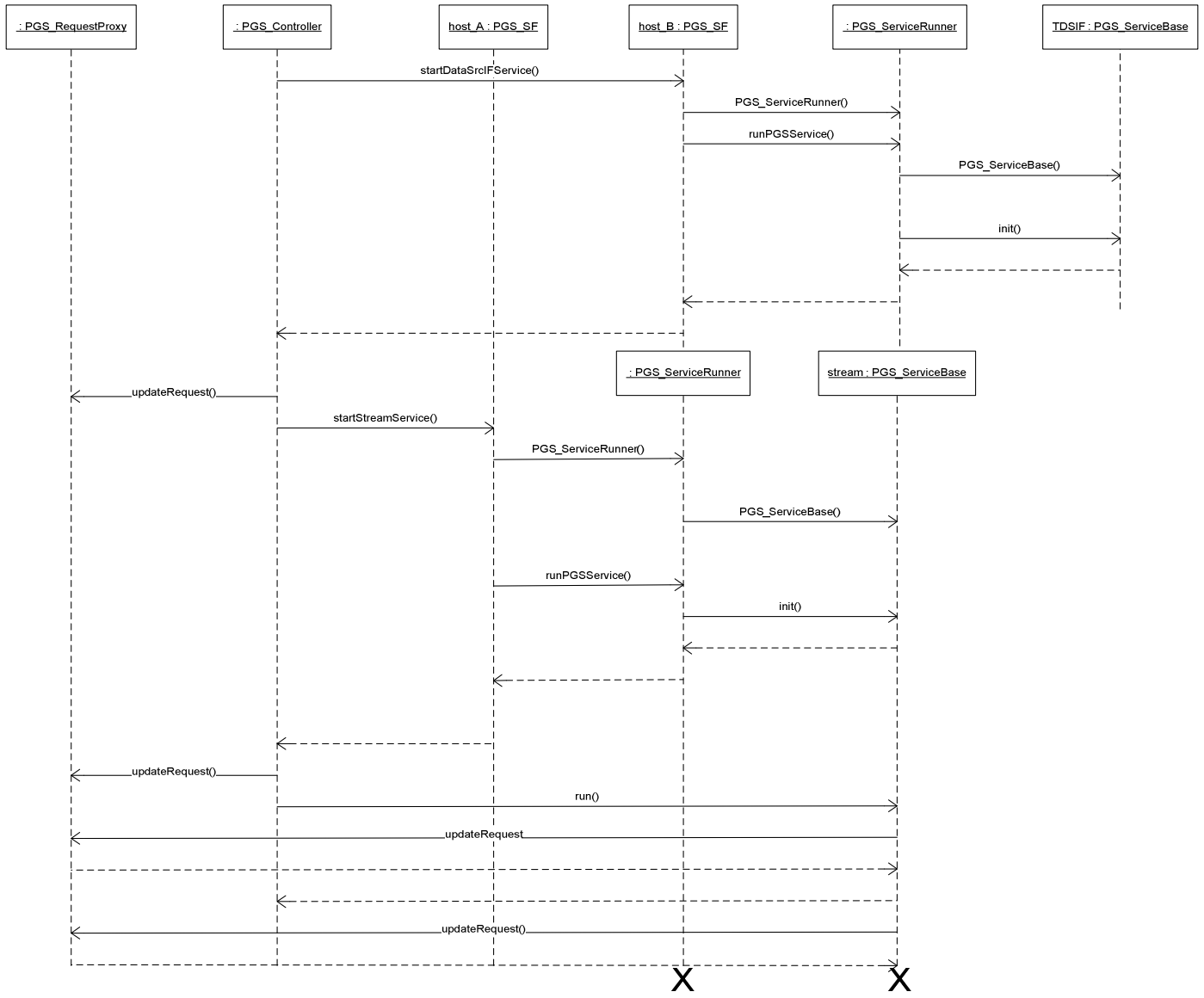

Figure 1. Request Dispatching

5.2. Service Factory

The SF is responsible for starting components necessary to complete the Client's request. Each SF may start either one or both of the PGS_DataSource and the PGS_Stream components. However, to successfully complete the Client's request, it is necessary that the SF be running with the right privileges. For example, if the Client wants products to be placed in a specific directory, then to successfully complete the Client's requests, the SF must have the necessary privileges to the specified directory, and the SF must start the components with the same privileges.

There will be multiple SFs running on one or more hosts. The purpose of these is to enable load balancing and to provide fail over.

5.2.1. Service Factory (SF) startup

In the MIPL environment, for the SF to have the same privileges as the requesting user, the SF must be started by that requesting user. It is possible to have each user starts its own SF and the Controller will be modified to dispatch the requests to the correct SF. However, this requires each user to start its own SF. Also, to utilize PGS fail over and load balancing mechanism, the analyst has to start more than one SF on different hosts, thus resulting in unnecessary resource usage. A different approach is to have a super user (from now referred as the SF admin) that starts the SFs.

Usually, each project has an operational environment set with all its analysts having the same privileges to the operational environment. Thus, it is okay to have the SF admin start the required SFs and have the SFs be used by all authorized analysts. This scheme dramatically reduces the number of SFs required, thus allowing a better resource utilization.

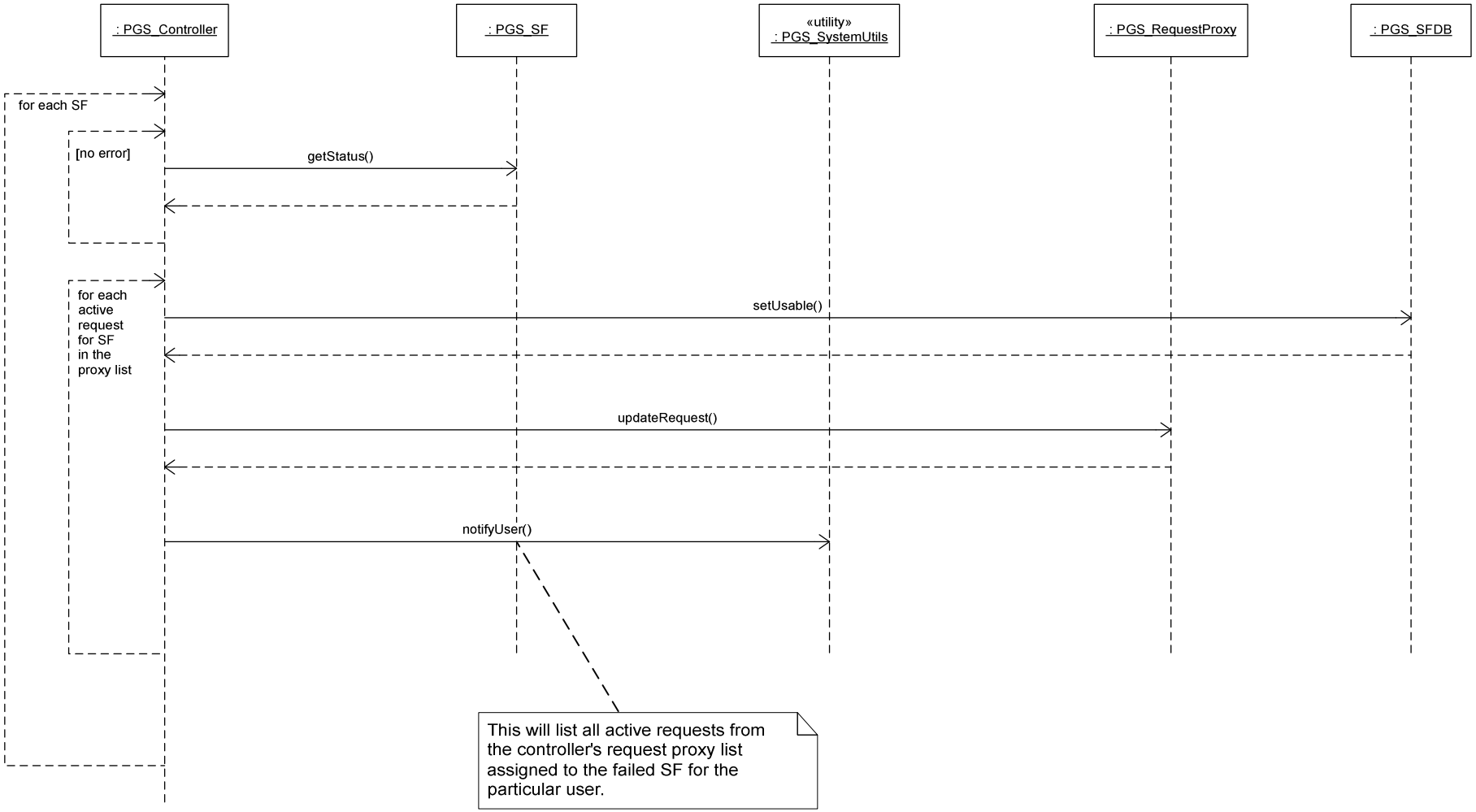

Figure 2. Controller Monitors SFs

For request processing integrity, only the Controller holds and uses the reference of the SF. As with the Controller, the SF requires the admin to provide correct login information. Figure 5 illustrates the startup sequence for the SF.

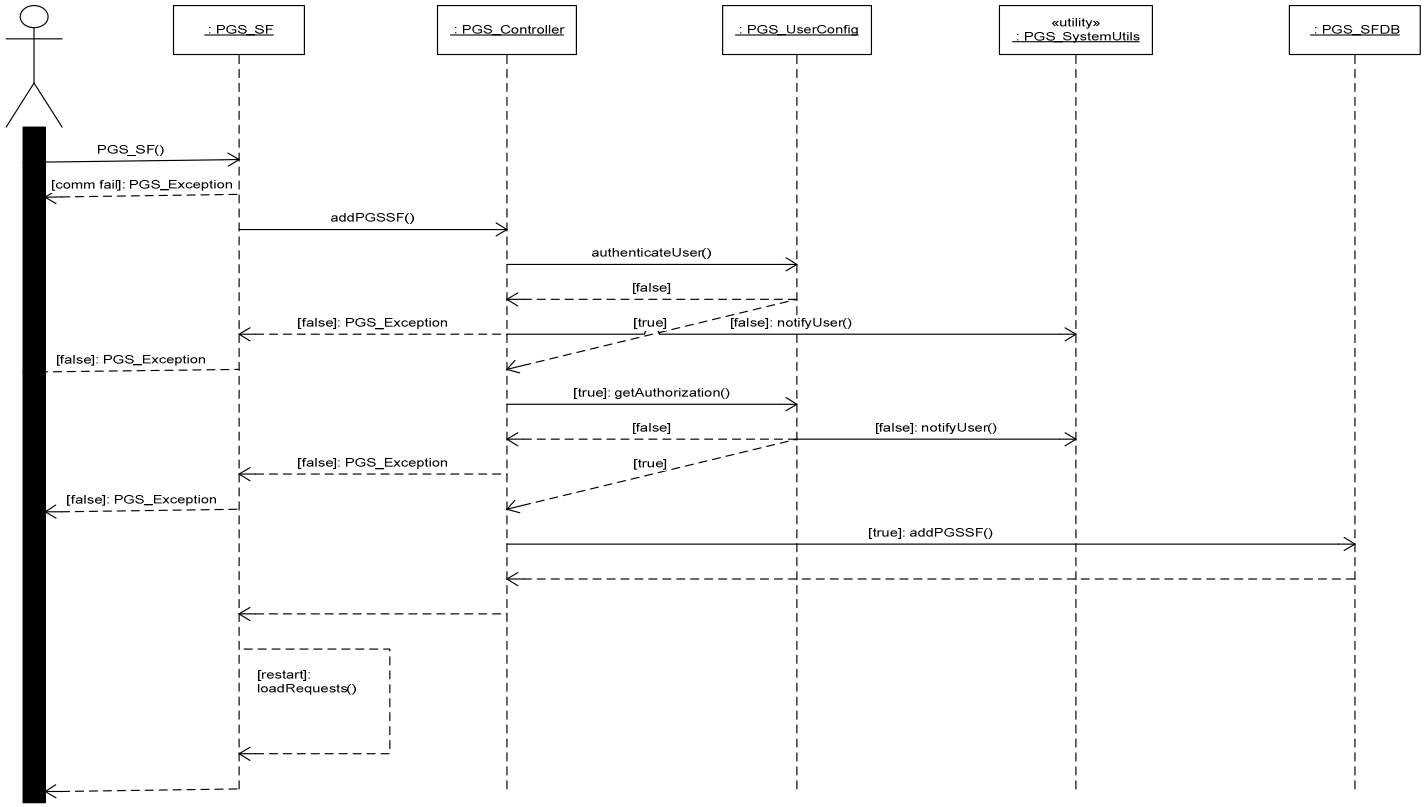

Figure 3. SF Startup Sequence[2]

[2] All UML diagrams were converted to PDF and than to JPG. As a result, some of the doted arrows appear as solid lines. Please refer to attach diagrams for details.

5.2.2. A Stream and Its Components

The two components (or services) started by the SF are the Data Source (PGS_DataSource) and the stream (PGS_Stream). The purpose of the PGS_DataSource is to encapsulate the communications with the actual raw data source (i.e., TDS or DOM). The stream encapsulates the creation and utilization of project specific data processing code (depicted as PGS_WorkerObject in Figure 6). The stream also provides additional functionality such as stream-level logging.

The SF will allow either of the components to be started as a separate process or a separate thread. In addition, the framework will support TIE approach (delegation approach) instead of the depicted inheritance approach for integration of project specific worker objects.

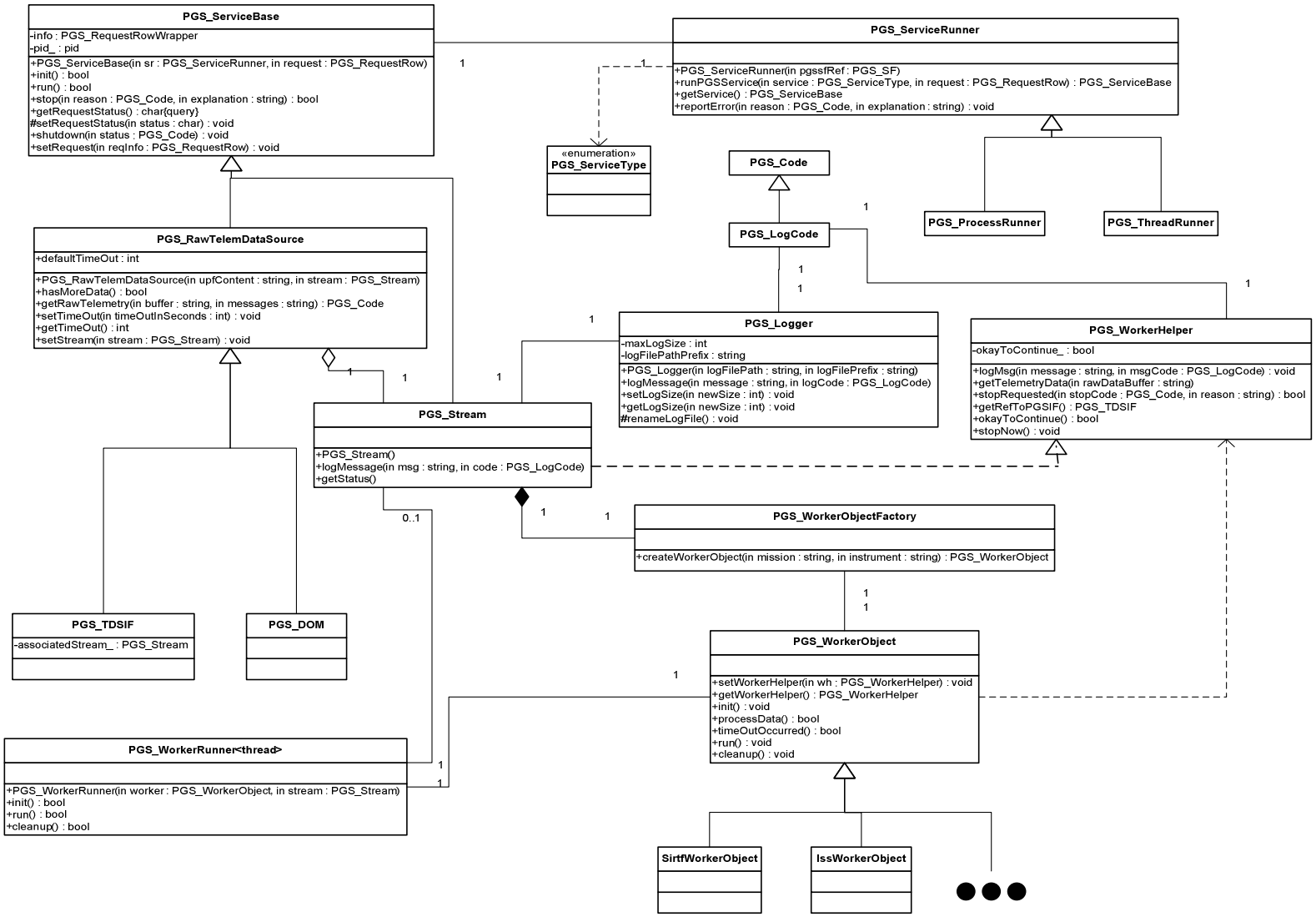

Figure 4. PGS Telemetry Service Class Diagram

5.2.3. Load Calculation

For load balancing purpose, the SF needs to keep track of its load factor. To do so, the SF utilizes the PGS_LoadCalculator, created by the using the PGS_LoadCalculatorFactory (see Figure 7 for design details). The use of the factory design pattern is to allow project specific creation of load calculators. The default load factor calculation implementation will take into consideration the available system memory and other processing parameters.

The load calculator is informed when a service is started and when it completes; thus allowing the load calculator to adjust its load factor by accessing differences in the resources, such as available memory and CPU usage.

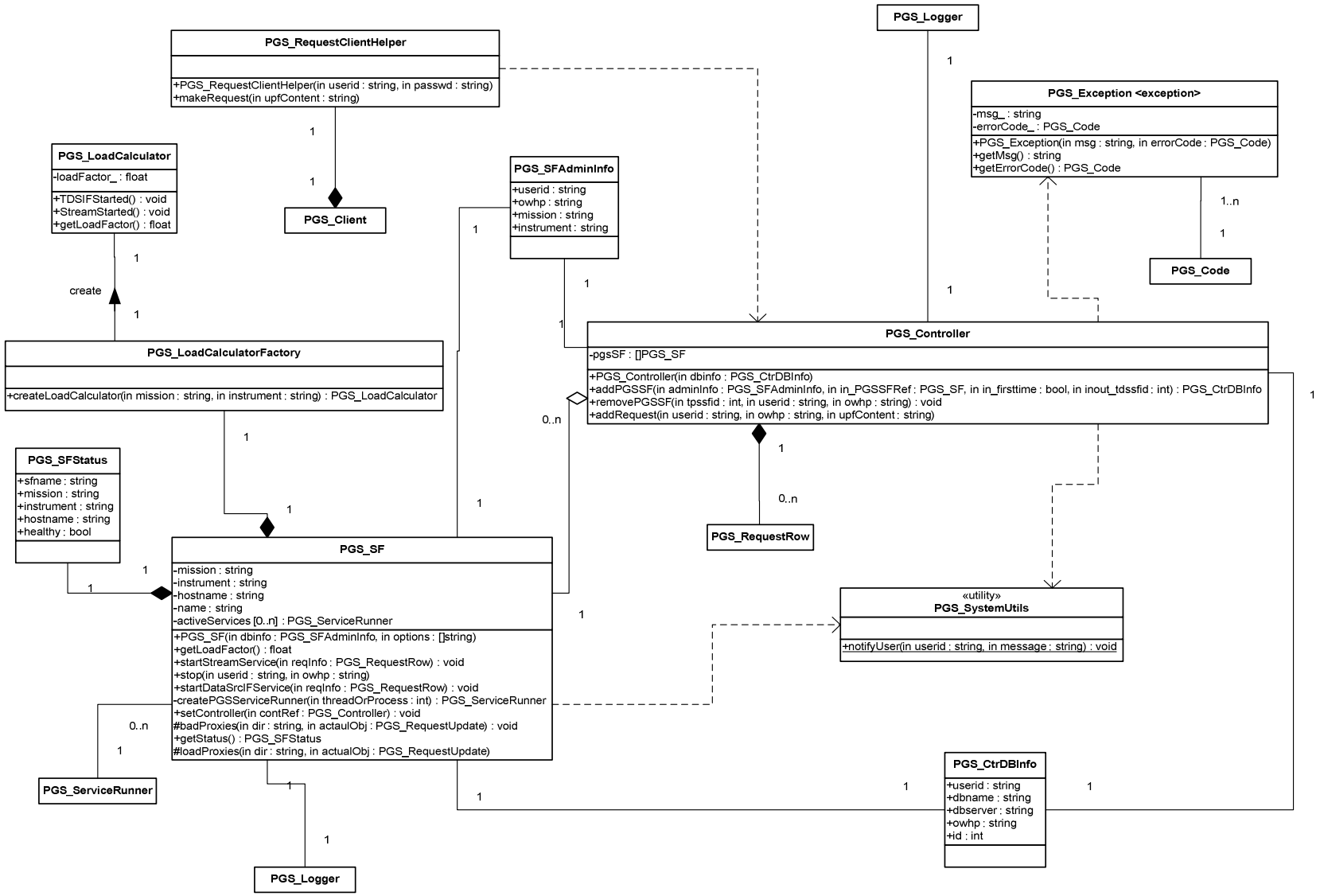

Figure 5. PGS Core Service Class Diagram

5.2.4. Service Monitoring

The SF monitors each stream it starts. That is, it pings each stream periodically till the stream completes. If a negative response is received from the stream, or if it is not reachable, the SF notifies the requesting analyst and the SF admin about the failure and puts the requests in an error state. Note that the stream is not restarted since a cleanup may be required before a restart (e.g., cleanup the output directory or the product catalog). The sequence diagram in Figure 8 shows detailed object interaction information.

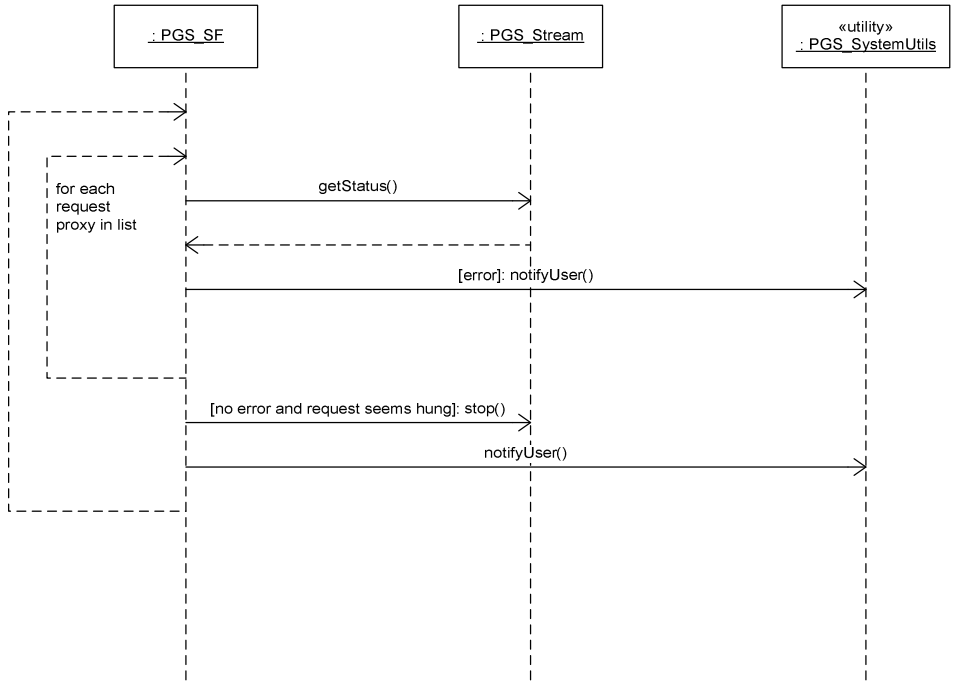

Figure 6. Request Monitoring by SF

5.3. Persistent Store Definition

PGS relies on the Persistent Store for saving processing information and for storing server and user configurations. It is assumed that only designated personnel, the PGS administrators, will have direct access to the Persistent Store. Section 4.1.1 describes about the justification for this restriction.

Since PGS is designed to be a piece of multi-mission software, it is made highly configurable. Thus, this section describes the abstract interfaces that the Controller depends on for retrieving and storing information into the database[3].

5.3.1. User Personal and Configuration Information Interface

This interface is used for retrieving information about the users: both personal information and configuration information. The user's personal information includes items such as the full name, email and pager number. The user's configuration information captures the relationship between the user and his given permissions. It is possible that one user is authorized to run requests for multiple instruments, and for each instrument, the user may have different permissions. The access methods are listed in the following sections.

[3] For readability, Java syntax is used.

5.3.1.1. getUserConfig

Retrieves user configuration information for the given user, mission and instrument

Signature:
```
Hashtable [] getUserConfig (String userID, String mission,
                            String instrument)
```

Input Parameters:

- userID — A string value containing a unique abbreviated ID for the user. This will be similar to the MIPL assigned 3 letter user ID if one is available. This is the primary search key.
- mission — A string value containing the mission name for which the user's information is to be retrieved. (Optional).
- instrument — A string value containing the instrument for which the user's information is to be retrieved. (Optional).

Return Value:

- table — An array of Hashtables containing the following fields for each record:

userID	A string value containing the user ID.
mission	A string value containing the mission name.
instrument	A string value containing the instrument name.
permissions	A bit encoded integer value that indicates what the user is authorized to do. The encoding is as follows (bit 0 is the right most bit): 0 – Read allowed. That is, the user can tap into running streams and inquire about the stream info. 1 – User can start and stop streams 2 – 7 – Not used.
OWHP	The one way hash password of the user for the particular mission and instrument.

5.3.1.2. authenticateUser

Authenticates the given user.

Signature: `bool authenticateUser (String userID, String OWHP, String mission, String instrument)`

Input Parameters:

- userID — A string value containing the user ID. This is the primary search key.
- mission — A string value containing the mission name. This is optional.
- instrument — A string value containing the instrument name. This is optional.

Return Value:

- authenticated — A boolean value if the given userID and OWHP matches what is returned by getUserConfig for the specified mission and instrument

5.3.1.3. getAuthorization

Get the authorization status of the given user.

Signature: `int getAuthorization (String userID, String mission, String instrument)`

Input Parameters:

- userID — A string value containing the user ID. This is the primary search key.
- mission — A string value containing the mission name. This is optional.
- instrument — A string value containing the instrument name. This is also optional.

Return Value:

- permissions — A bit encoded integer value that indicates what the user is authorized to do. The encoding is as follows (bit 0 is the right most bit):
  0 – Read allowed. That is, the user can tap into running streams and inquire about the stream info.
  1 – User can start and stop streams.
  2 to 7 – Not used.

5.3.1.4. getUserInfo

Get the information on the given user.

Signature: `Hashtable [] getUserInfo(String userID, String mission, String instrument)`

Input Parameters:

- userID A string value containing the user ID. This is the primary search key.
- mission A string value containing the mission name. (Optional).
- instrument A string value containing the instrument name. (Optional).

Return Value:

- table An array of Hashtables containing the following fields for each record:

userID	A string value containing the user ID.
fullName	A string value containing the full name of the user identified by the user ID.
eMail	A string value containing the email address of the user.
pager	A numeric value containing the pager number of the user.
mission	A string value containing the mission name. Only if a mission and instrument is given.
instrument	A string value containing the instrument name. Only if a mission and instrument is given.
permissions	A bit encoded integer value that indicates what the user is authorized to do. Only if a mission and instrument is given.
OWHP	The one way hash password. Only if a mission and instrument is given.

5.3.2. Requests Interface

This interface is used for storing new requests and for retrieving and modifying existing requests that are active or have been completed but not archived. The access methods are listed in the following sections.

5.3.2.1. addNewRequest

This access method is utilized for adding new requests into PGS.

Signature: `Hashtable addNewRequest (Hashtable request)`

Input Parameter:

- request — A Hashtable containing the following fields:

userID	A string value containing the user ID**.**
mission	A string value containing the mission name.
instrument	A string value containing the instrument name.
streamName	A string value containing the name of the request.
startTime	A string value containing the time when this request is to be started.
description	A string value containing the description of the request.
priority	An integer (1-5) for the priority of the request. 1 is the lowest, and 5 is the highest.
upfName	A string value containing the absolute path name of the UPF that the Controller created using the content of the UPF passed to it when the addRequest method was invoked. It is constructed from the upfDir parameter and the request ID

Return Value:

- table — A Hashtable containing the following fields:

[…]	All of the fields from the input Hashtable
requestID	This is the unique integer id that is automatically generated and assigned by the Persistent Store.
status	A character value indication the status of this request. This is defaulted to ‘W’ meaning waiting for execution. Other possible value for this field are: ‘R’ meaning the request is currently executing, ‘E’ means an error occurred, ‘C’ means the request completed successfully and ‘A’ means that the request was aborted.

5.3.2.2. updateRequest

Used for updating various fields of the request.

Signature: `void updateRequest (Hashtable request)`

Input Parameter:

- request A Hashtable containing the following fields:

requestID	The unique request ID (integer) that is used to identify the request.
dsRef	A string value containing the reference to the data source that is used for obtaining data for this request. This is populated by the Controller after successful startup of the data source service.
dsPID	The integer containing the process ID of the data source if it is started as process or the thread ID if it is started as thread.
dsSFID	The id of the SF that started the data source.
streamRef	A string value value containing the reference to the stream that is responsible for processing the request. This is populated by the Controller after successful startup of the stream service.
streamPID	Same as dsPID, but for stream.
streamSFID	Same as dsSFID, but for stream.
exitMessage	A string value containing the request completion status.
endTime	A string value containing the completion time of the request.

Return Value:

- *None*

5.3.2.3. getActiveRequests

Get all the active requests for the given parameters.

Signature:
```
Hashtable [] getActiveRequests (String mission, String instrument,
                                int dsSFID)
```

Input Parameters:

- mission — A string value containing the mission name.
- instrument — A string value containing the instrument name.
- dsSFID — The id of the SF that started the data source. It will be matched with the streamSFID.

Return Value:

- table — An array of Hashtables containing the following fields for each record:

userID	A string value containing the user ID.
fullName	The full name of the requester.
mission	The mission name.
instrument	The instrument name.
streamName	A string value containing the name of the request.
startTime	A string value containing the time when this request is to be started.
description	The description of the request.
priority	A priority of the request.
upfName	The absolute path name of the UPF.
requestID	This is the unique ID that is used to identify the request.
status	A character value indication the status of this request.
dsRef	A string value containing the reference to the data source.
dsPID	The integer value containing the process/thread ID of the data source
dsSFID	The id of the SF that started the data source.
streamRef	A string value containing the reference to the stream that is responsible for processing the request.
streamPID	Same as dsPID, but for stream.
streamSFID	Same as dsSFID, but for stream.
exitMessage	A string value indication the request completion status.
endTime	A string value containing the completion time of the request.

5.3.3. Archived Requests Interface

This interface is utilized for archiving and retrieving of normally or abnormally completed requests. The access methods are listed in the following sections.

5.3.3.1. archiveRequest

Archive the given request.

Signature: `void archiveRequest (Hashtable request)`

Input Parameter:

- request — A Hashtable containing the following fields:

userID	The ID of the user making this request.
fullName	The full name of the requester.
mission	The mission name.
instrument	The instrument name.
streamName	A string value containing the name of the request.
upfName	Same as dsPID, but for stream.
description	The description of the request.
status	A character value indication the status of this request.
exitMessage	A string value indication the request completion status.
startTime	A string value containing the time when this request is to be started.
endTime	A string value containing the completion time of the request.
dsRef	A string value containing the reference to the data source.
dsPID	The integer value containing the process/thread ID of the data source.
dsSFID	The id of the SF that started the data source.
streamRef	A string value containing the reference to the stream that is responsible for processing the request.
streamPID	Same as dsPID, but for stream.
streamSFID	Same as dsSFID, but for stream.

Return Value:

- *None*

5.3.3.2. getArchivedRequests

Retrieve archived requests that match the given criteria.

Signature: `Hashtable[] getArchivedRequests(Hashtable criteria)`

Input Parameter:

- criteria — An array of Hashtables containing the following fields for each record that matches the given criteria:

userID	The ID of the user making this request.
fullName	The full name of the requester.
mission	The mission name.
instrument	The instrument name.
streamName	A string value containing the name of the request that is known to the external user.
upfName	Same as dsPID, but for stream.
description	The description of the request.
status	A character value indication the status of this request.
exitMessage	A string value indication the request completion status.
startTime	A string value containing the time when this request is to be started.
endTime	A string value containing the completion time of the request.
dsRef	A string value containing the reference to the data source.
dsPID	The integer value containing the process ID of the data source if it is started as process or the thread ID if it is started as thread.
dsSFID	The id of the SF that started the data source.
streamRef	A string value containing the reference to the stream that is responsible for processing the request.
streamPID	Same as dsPID, but for stream.
streamSFID	Same as dsSFID, but for stream.

Return Value:

- array — An array of Hashtables with the following fields for each record that matched the given criteria:

userID	The ID of the user making this request.
fullName	The full name of the requester.
mission	The mission name.
instrument	The instrument name.
streamName	A string value containing the name of the request that is known to the external user.
upfName	Same as dsPID, but for stream.
description	The description of the request.
status	A character value indication the status of this request.
exitMessage	A string value indication the request completion status.
startTime	A string value containing the time when this request is to be started.
endTime	A string value containing the completion time of the request.
dsRef	A string value containing the reference to the data source.
dsPID	The integer value containing the process ID of the data source if it is started as process or the thread ID if it is started as thread.
dsSFID	The id of the SF that started the data source.
streamRef	A string value containing the reference to the stream that is responsible for processing the request.
streamPID	Same as dsPID, but for stream.
streamSFID	Same as dsSFID, but for stream.

5.3.4. Service Factory Configuration Interface

This interface is used to find out what hosts the service factories that serve the given mission are located. The access methods are listed in the following sections.

5.3.4.1. getSFConfig

Get the Service Factory Configurations that match the given parameters.

Signature:

```
Hashtable [] getSFConfig (String mission, String instrument,
                          short serviceType)
```

Input Parameters:

- mission — The mission name.
- instrument — The instrument name.
- serviceType — A short value indicates what service type this pertains to. The valid values and their meaning is as follows: 0 – TDS, 1 – DOM, 2 – Stream, and 3 – FEI.

Return Value:

- array — An array of Hashtables containing the following fields for each record:

hostName	A string value containing the hostname of the server running the SF that is to be used for starting the service indicated by the next field.
serviceType	A short value indicates what service type this pertains to. The valid values and their meaning is as follows: 0 – TDS 1 – DOM 2 – Stream 3 – FEI
[…]	All of the fields returned by the getSF access method of the Service Factory Interface when invoked with a given mission, instrument, and serverHost.

5.3.5. Service Factory Interface

This interface is used by the Controller to add a new SF into the PGS, to remove an existing SF, and to disable/enable the running of an SF. The access methods are listed in the following sections.

5.3.5.1. addSF

This access method is used by the Controller to add a new Service Factory to PGS.

Signature: `Hashtable addSF (Hashtable sfInfo)`

Input Parameter:

- sfInfo A Hashtable containing the following fields:

SFRef	A string value reference to the SF.
hostName	A string value containing the hostname on which the SF is running.
userID	The ID of the user who starts the SF.
mission	The mission the SF is to serve.
instrument	The instrument of the mission that the SF is to serve.
maxServicesToRun	An integer value representing the maximum number of services (Data Source I/Fs and Streams combined) that this SF can run at the same time.
runServiceAs	The short value that indicates whether the service is to be run as a thread or a process: 0 – thread 1 -- process
logDir	The directory where the log file resides. The log file is used by the SF for reporting system level events such as starting of the service, service completion etc.
backupDir	The directory that is to be used as the Persistent Store if the SF is unable to update the request state directly. The stream class utilizes this collection for similar purposes.

Return Value:

- table A Hashtable with the following fields.

sfRef	A string value reference to the SF.
hostName	A string value containing the hostname on which the SF is running.
userID	The ID of the user who starts the SF.
mission	The mission the SF is to serve.
instrument	The instrument of the mission that the SF is to serve.
maxServicesToRun	The maximum number of services (Data Source I/Fs and Streams combined) that this SF can run at the same time.
runServiceAs	The number that indicates whether the service is to be run as a thread or a process: 0 – thread 1 -- process
logDir	The directory where the log file resides. The log file is used by the SF for reporting system level events such as starting of the service, service completion etc.
backupDir	The directory that is to be used as the Persistent Store if the SF is unable to update the request state directly. The stream class utilizes this collection for similar purposes.
sfID	The unique ID of the Service Factory. This is automatically generated and assigned by the Persistent Store.
isUsable	A boolean value indicating if this SF is usable. The default is true.

5.3.5.2. getSF

This access method is used for obtaining information about all of the Service Factories that match the given parameters.

Signature:
```
Hashtable [] getSF (int sfID, String mission, String instrument,
                    String userID, String hostName)
```

Input Parameters:

- sfID — A string value reference to the SF.
- mission — The mission the SF is to serve.
- instrument — The instrument of the mission that the SF is to serve.
- userID — The ID of the user who starts the SF.
- hostName — A string value containing the hostname on which the SF is running.

Return Value:

- array — An array of Hashtables containing the following fields for each record:

sfRef	A string value reference to the SF.
hostName	A string value containing the hostname on which the SF is running.
userID	The ID of the user who starts the SF.
mission	The mission the SF is to serve.
instrument	The instrument of the mission that the SF is to serve.
maxServicesToRun	The maximum number of services (Data Source I/Fs and Streams combined) that this SF can run at the same time.
runServiceAs	The number that indicates whether the service is to be run as a thread or a process: 0 – thread, 1 – process.
logDir	The directory where the log file resides. The log file is used by the SF for reporting system level events such as starting of the service, service completion etc.
backupDir	The directory that is to be used as the Persistent Store if the SF is unable to update the request state directly. The stream class utilizes this collection for similar purposes.
sfID	The unique ID of the Service Factory. This is automatically generated and assigned by the Persistent Store.
isUsable	A boolean value indicating if this SF is usable. The default is true.

5.3.5.3. removeSF

This access method is used for removing the information about a Service Factory.

Signature: `int removeSF (Hashtable criteria)`

Input Parameter:

- criteria — A Hashtable containing one or more of the following fields:

sfID	An integer value representing the SF ID.
sfRef	A string value reference to the SF.
hostName	A string value containing the hostname on which the SF is running.
userID	The ID of the user who starts the SF.
mission	The mission the SF is to serve.
instrument	The instrument of the mission that the SF is to serve.
maxServicesToRun	The maximum number of services (Data Source I/Fs and Streams combined) that this SF can run at the same time.
runServiceAs	The number that indicates whether the service is to be run as a thread or a process: 0 – thread 1 -- process
logDir	The directory where the log file resides. The log file is used by the SF for reporting system level events such as starting of the service, service completion etc.
backupDir	The directory that is to be used as the Persistent Store if the SF is unable to update the request state directly. The stream class utilizes this collection for similar purposes.

Return Value:

- count — An integer value indication the number of Service Factories removed.

5.3.5.4. setUsable

This access method is used for enabling or disabling the user of the Service Factory identified by the given ID.

Signature: `void setUsable (int sfID, bool val)`

Input Parameters:
- sfID A string value reference to the SF.
- value "true" for setting this SF to be usable; "false" for setting it to not usable.

Return Value:
- *None*

5.3.6. Schedule Interface

This interface is used by scheduling service to obtain scheduling information. This is work-in-progress.

5.3.7. Sequence Interface

This interface is utilized by scheduling service to specify what processes are to be executed pre and post telemetry processing execution. Abnormal termination of any process will result in pause of execution, unless specified otherwise.

6. Failure and Recovery

This section explores scenarios involving failure of different components of the PGS system and presents solutions that make the PGS as robust as possible against failures.

6.1. SF Failure

For each Client's request, the Controller starts two services: the PGS_RawDataSource and the PGS_Stream . Unavailability of the appropriate SF will result in request failure. As discussed in Section 5.1.3, the Controller provides load-balancing mechanism, which readily provides a fail-over functionality as described below. While performing load balancing, if the most qualified SF is nonfunctional, the Controller utilizes the next most qualified SF in the list to start the service; thus providing a fail-over mechanism and increasing the chances of request dispatching. This mechanism is depicted in Figure 9; a sequence diagram that gives details about the request dispatching. However, this by no means guarantees request dispatching. For example, if the service host for a service is provided in the UPF (as mentioned in Section 5.1.3) and the SF on that host is unreachable, the Controller will simply reject the request. Later sections discuss mechanisms to make PGS a high available system.

6.2. Service Failure

After a request has been successfully dispatched, it is possible that its component may fail, i.e. either the raw data source or the stream itself fails. Since the stream queries the data source for data, a data source failure will be detected and reported by stream. The stream failure will be detected and reported by the SF (as discussed in Section 5.2.4). Since the worker object (project specific telemetry code) executes in the same address space as the stream, the failure of worker object will result in stream failure.

6.3. Controller Failure

The Controller is at the heart of the PGS system. Unavailability of the Controller will cripple the PGS system. Having a process monitor and restarting the Controller is not adequate since one or more Client requests may be missed between detecting the failure and restarting the Controller. Also, since Controller sits between the Persistent Store and other components of the PGS system, Controller failure can impact system integrity. There are out of the box solutions available to cope with this problem. The next section discusses a solution utilized to circumvent database failure, which will be utilized to overcome the system integrity problem in the event of Controller's failure.

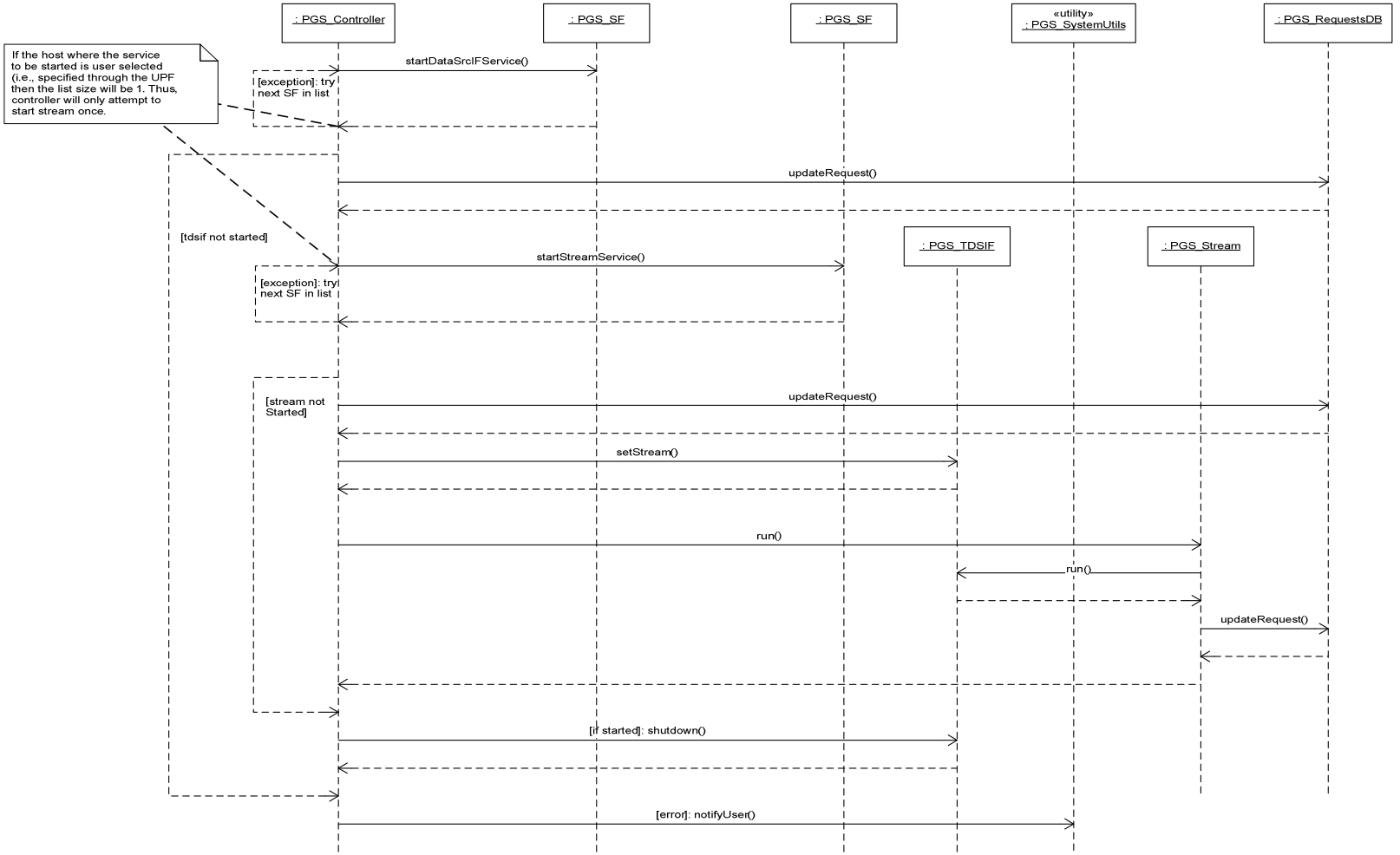

Figure 7. Service Dispatch and SF Failure

Regardless of the mechanism used, upon restarting, the Controller will utilize the PGS_PS_SF interface (see Section 5.3.6) to obtain references to registered SFs and it will ping them as mentioned in Section 5.1.4. Figure 10 gives the Controller's startup scenario that shows how the Controller utilizes the PGS_PS_SF interface.

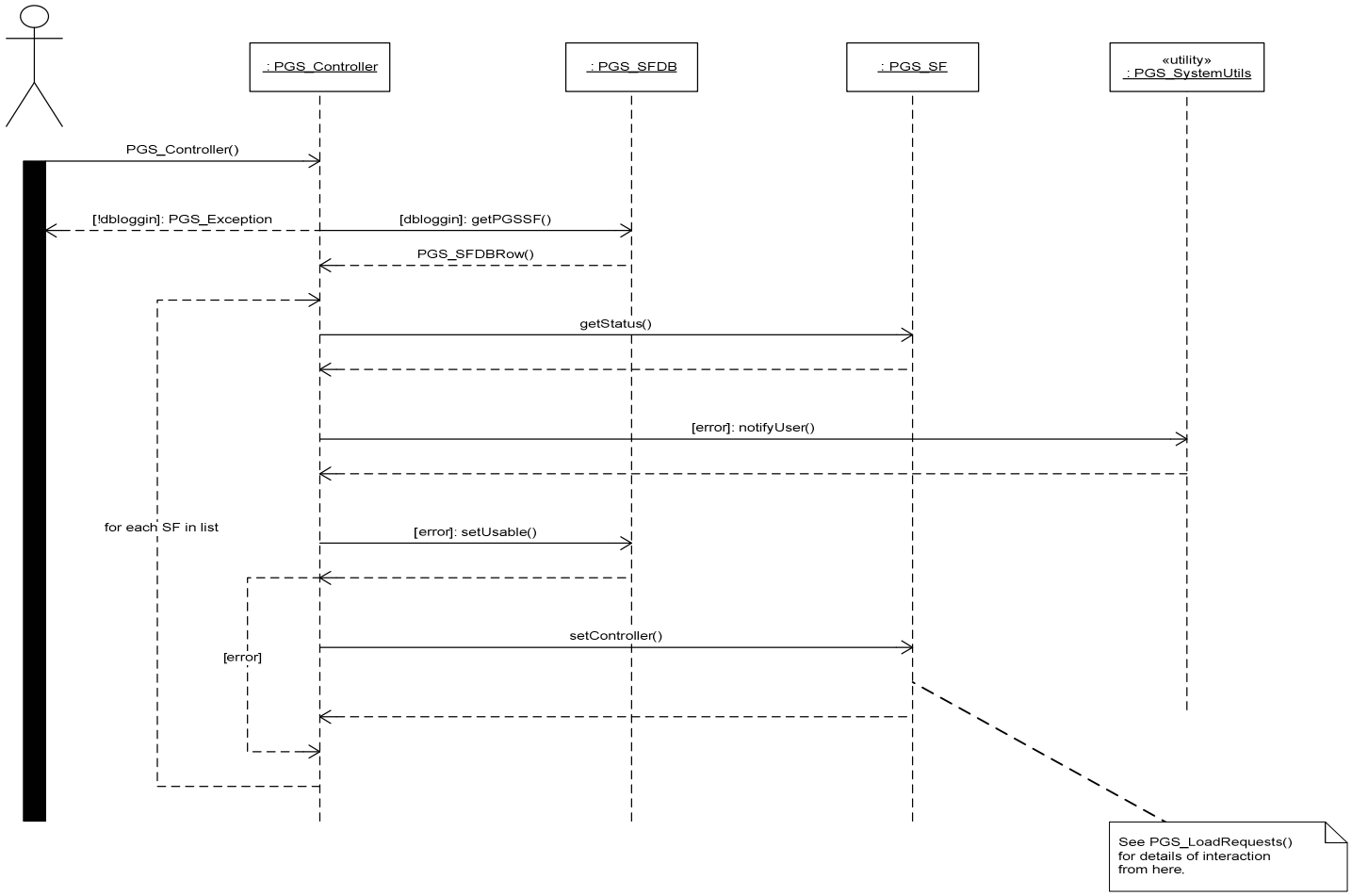

Figure 8. Controller Startup Sequence

6.4. Persistent Store Failure

Foundation to the failure and recovery mechanism of the PGS is the Persistent Store: the MIPL database. From service startup to request processing the database plays a vital role. However, failure to communicate with the database does not

necessarily mean PGS failure. For example, the request status is set to completion once the request is finished. However, failure to communicate with the database in this case does not mean request failure. On the other hand, failure to communicate with the database at time of request creation definitely means request failure.

6.4.1. Request Proxy

The fields of a row in the request database get modified at several places. When the Controller starts services, it modifies the service specific fields in the Requests table (see Section 5.3.3). The state of the service is modified by the stream when it starts and when it completes. However, as mentioned earlier, only the Controller has direct access to the database. Thus, updates from streams go through the Controller. However, unavailability of either the Controller or the database does not mean request failure. At the same time, to ensure PGS database data integrity, the fields of a request need to be eventually modified to reflect the request's correct state.

This problem is solved by use of a slightly modified proxy (request proxy from now on) design pattern. The purpose of the proxy object is to act as a surrogate and encapsulate communications with the actual object. The purpose of the request proxy is to encapsulate communications with the Controller for the stream and with the database for the Controller.

The request proxy is created in the same address space as the process that utilizes it. The proxy starts with the same state as that of the request it is representing. When an update call is made to the proxy object, it first updates its own state and then forwards the calls to the actual object. However, if the actual object becomes unreachable, the proxy stops forwarding calls to it. Instead, it stores its state on to a different kind of Persistent Store (e.g., onto the file system). When the actual object becomes available, the proxy object synchronizes its state with the actual object. Figure 11 shows a class diagram and Figure 12 shows the startup sequence of the proxy manager that loads the stored proxies and then synchronizes it with the actual object.

Another advantage of the request proxy object and how it is used is that the state is stored onto secondary storage only by one remote object. For example, if the database becomes unavailable, then only the Controller stores the state of the proxy. The communication between the stream and the Controller for the request modification is not affected at all.

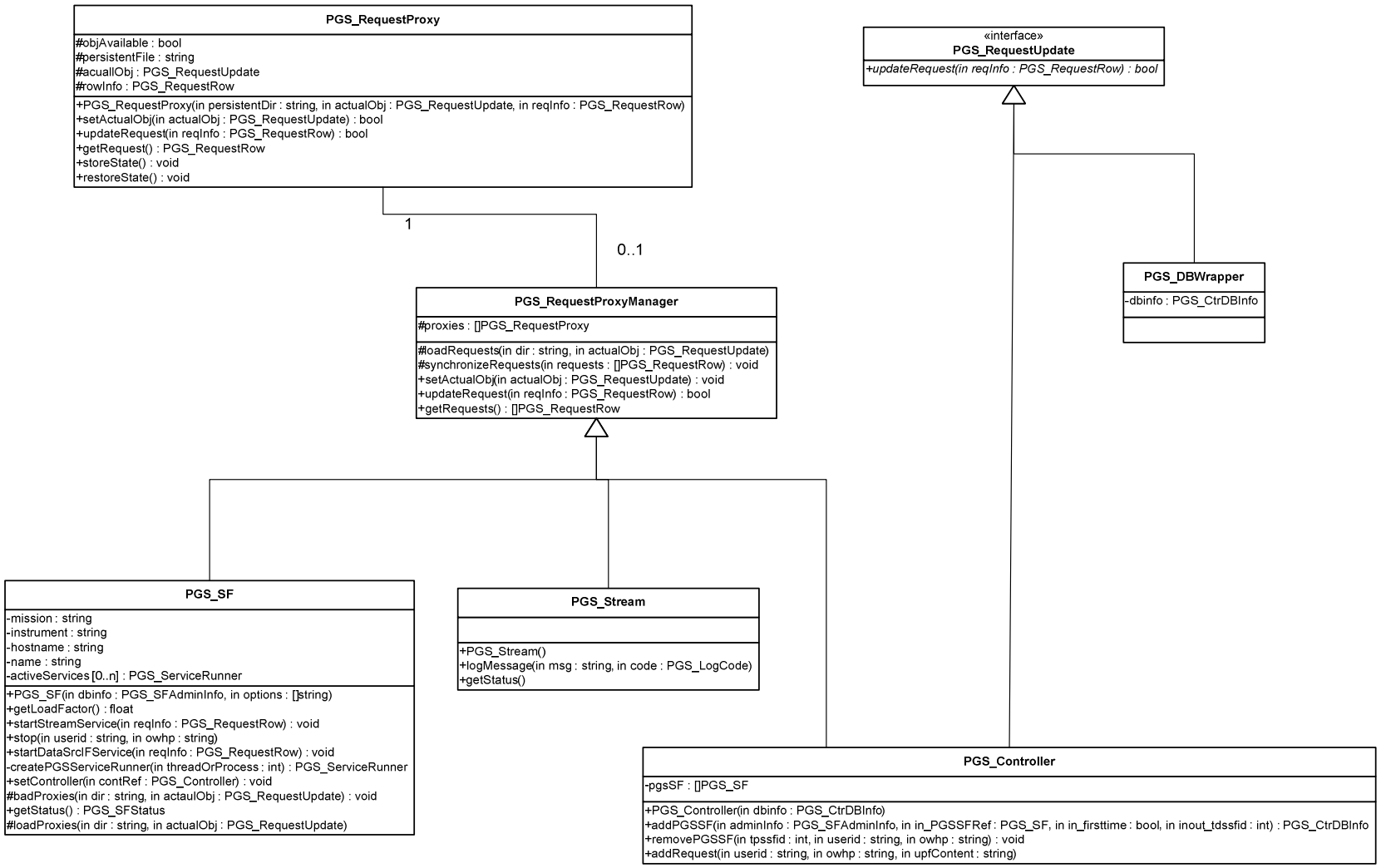

Figure 9. Request Proxy Class Diagram

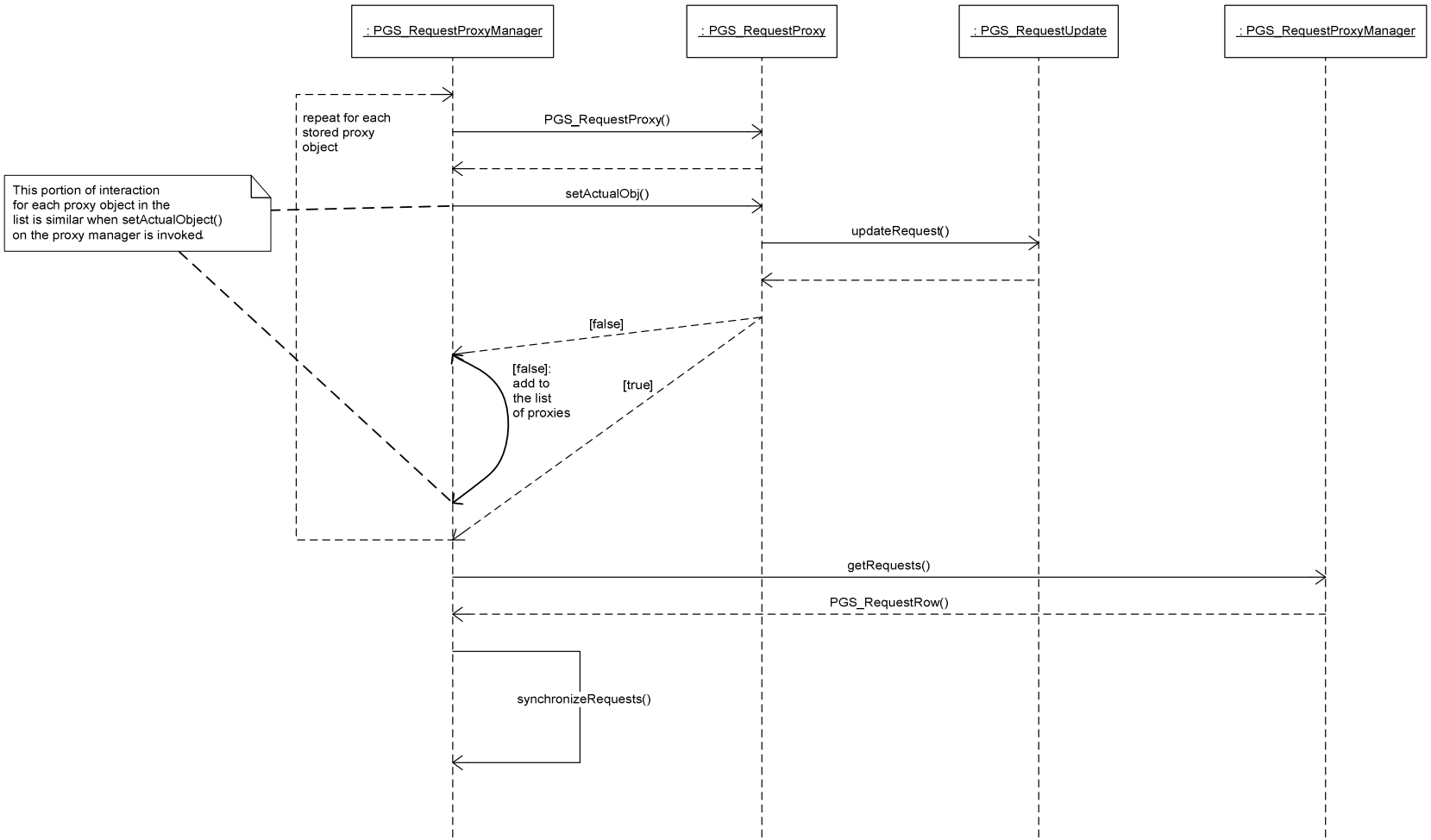

Figure 10. Proxy Manager Startup

Appendix A: Database Schema

In progress.

Appendix B: Design Decisions

Design meetings were held between August 2003 and January 2004. Design meetings' minutes can be found at `https://eis-lib.jpl.nasa.gov/eis-lib/dscgi/ds.py/View/Collection-11302`. This section lists deviations from the current PGS design and choices made with explanation.

- One set of core services per mission: The decision to run one set of core services per mission is based on past experience. With one set of core services, overcoming problems related with server upgrades and user permissions (see Section 6.1) is very difficult.
- Not reusing streams: Again, past experience on stream reuse suggests that it is best not to reuse stream. Stream reuse requires proper cleanup on PGS_WorkerObject side, which places a requirement on the worker object developers (and thus on the project).
- Java as primary programming language: A trade study is provided in Appendix C.
- Use RMI: A trade study is provided in Appendix D.
- Means of Notification: Components of the current system utilizes TAO's notification utility to report events. The current scenario is that the core sends a message to the notification server which is published or sent to all registered interested consumers. The core does not block until the message is received by consumer(s).
- There are three consumers of the messages created by core: stream logger, APEX and JEDI. In new design, the logger is no longer a server, thus it does not need this facility. This leaves APEX and JEDI. Since both APEX and JEDI are written in Java, the Java Messaging service is our primary choice. Just like CORBA's notification, the JMS supports both synchronous and asynchronous messaging with a set of quality of service such as message acknowledgement, message priority, message expiration, and most importantly, message persistent, that is, guarantee that a message will be delivered to the JMS. Moreover, there will be a JMS server running per select system for SPIDER, which may be used by PGS.

Appendix C: Trade Study: Java vs. C++

Both Java and C++ are advance and highly expressive object oriented programming language. However, there are subtle differences between the two languages. A trade study was done on February 10, 2004, and the decision is to use Java. The following is a list of differences that directly impact TPS implementation.

- Java can be ported across multiple platforms with little or no effort.
- Java alleviates developers from burdensome and erroneous task of memory allocation, de-allocation and memory manipulation. As a result, Java allows rapid development.
- Java has strict type checking at compilation and runtime that results in detectable errors. For example, if an array index is out of bound, an exception is thrown.
- Java uses exceptions for error reporting and enforces strict exception handling which results in better error handling.
- Built in data structures and algorithms reduces amount of coding.
- Language support for multithreading facilitates for thread safe environment.
- Java has automatic document generation.
- Java has a well defined interface for interacting with software written in C/C++.
- Java has build in support for Web and network development.
- Java's GUI development packages are portable and easy to use.
- JDBC provides a standardized support for database interface.
- JEDI, APEX and FEI5 are all in Java. Thus, using Java for TPS will provide better coupling.

Appendix D: Trade Study: CORBA vs. RMI

Both RMI and CORBA handle distributed programming. The following study was done on February 10, 2004. The decision is to go with RMI.

- RMI specification is short, simple and adequate for TPS.
- RMI supports distributed garbage collection.
- RMID (RMI daemon) supports for automatic start and restart of RMI servers.
- RMID supports grouping of servers, which enables load balancing. Moreover, grouping may be used to separate non-related servers.

Appendix E: Acronyms

APEX	Analysts' Portable EXchange
CORBA	Common Object Request Broker Architecture
DOM	Distributed Object Manager
EDR	Experimental Data Record
FEI	File Exchange Interface
JEDI	Java EDR Display Interface
JMS	Java Message Service
MIPL	Multi-Mission Image Processing Laboratory
OWHP	One Way Hash Password
PS	Persistent Store
RMI	Remote Method Invocation
SF	Service Factory
SFDU	Standard Formatted Data Units
TAO	The ACE ORB (Adaptive Communication Environment Object Request Broker)
TDS	Telemetry Data System
TDSIF	Telemetry Data System Interface
PGS	Telemetry Processing System
UPF	User Parameter File