Comando, control y telemetría satelital de próxima generación

Pablo Soligo*

Universidad Nacional de La Matanza - Comisión Nacional de Actividades Espaciales
psoligo@unlam.edu.ar

August 30, 2017

Abstract

Este trabajo presenta los resultados de las experiencias obtenidas en el desarrollo de un segmento terreno de próxima generación. Se desarrolló como parte de la materia proyecto integrador de la Maestría en Desarrollos Informáticos de Aplicación Espacial (MDIAE). Teniendo como objetivo el comando y control del satélite (SCC) y la recepción de telemetría. Se aplicaron técnicas y herramientas como camino alternativo a las tecnologías propietarias y al comando y control sobre lenguajes específicos. Utilizando Python como lenguaje y aprovechado sus capacidades de reflexión, se desarrolló un entorno multimisión capaz de interpretar secuencias de comandos, incluyendo todas las estructuras de control y disponibilizando todas las capacidades del interprete. Utilizando Django como Framework y específicamente su Object Relation Mapping (ORM) se logra generar scripts de comandos con acceso simplificado a los diccionarios de comandos y valores de telemetría mediante un lenguaje estándar, de propósito general y multiplataforma. Como segmento de vuelo se ha utilizado satélite de formación 2017 (FS2017) que también forma parte del proyecto integrador en conjunto con las Maestría en Tecnología Satelital (MTS) y la Maestría en Instrumentos Satelitales (MIS).

I. Introducción

En la industria cuando se trata de comandar un equipo remoto existen varios enfoques dependiendo de las necesidades. Desde sencillas implementaciones *Maestro-Esclavo* sin ningún tipo lógica de control asociada, pasando por sistemas algo mas complejos donde se aplican restricciones o sistemas basados en maquinas de estado. En el área espacial, una práctica común, es el uso de secuencias de comandos sobre estructuras de control. Empresas y agencias espaciales en todo el mundo han desarrollados sus propios lenguajes e intérpretes para cumplir con este objetivo:

STOL: Satellite Test and Operation Language. Desarrollado por la Nasa y ampliamente utilizado en varias misiones.

PLUTO: presente en algunas misiones de la ESA (Satellite Control and Operation System 2000).

Otros: desarrollados o utilizados por diferentes compañias SOL(GMV), CCL(Harris), PIL(Astrium), SCL(ICS).

En el caso del segmento terreno del *FS2017* se optó por usar un lenguaje de propósito general en lugar de crear un lenguaje especifico o utilizar los existentes en CONAE (Comisión Nacional de Actividades Espaciales). Este enfoque presenta múltiples ventajas,

Portabilidad: escogiendo correctamente la herramienta se puede lograr una buena portabilidad entre distintas plataformas.

Capacidad: las herramientas y capacidades generales de un buen lenguaje y entorno de desarrollo de propósito general superan ampliamente las posibilidades que puede ofrecer un entorno de propósito especifico.

*A thank you or further information

Base de usuarios: una importante base de usuarios implica soporte, documentación y mejoras además de una base de posibles recursos ya capacitados.

En el caso del proyecto integrador la opción de lenguaje propio de la misión se descartó por no disponer del tiempo que implica desarrollar y validar un intérprete de propósito específico. La opción de utilizar los interpretes de CONAE quedó relegada por sobre la opción Python, con una base de usuarios mucho más grande, multiplataforma, con mejores capacidades de depuración, documentación disponible en la comunidad y de dominio de todos los estudiantes.

II. Arquitectura

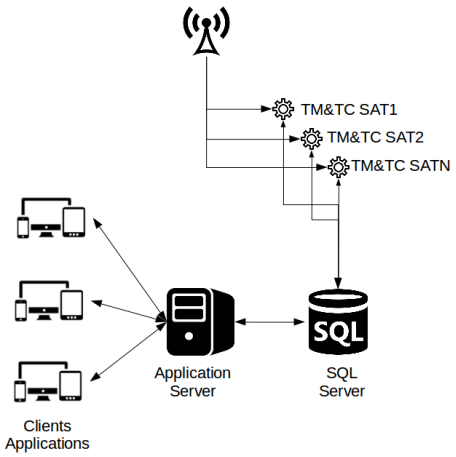
La solución propuesta propuesta para el segmento terreno se basa en una arquitectura cliente-servidor estándar y está influenciada por el framework de desarrollo Django, el que también influyo en un análisis y diseño orientado a objetos. Como repositorio de datos se utiliza un Postgresql 9.0 donde se almacenan la definición de datos y los datos mismos. Todo el acceso se realiza mediante el ORM (Object Relational Mapping) y no hay en todo el software un acceso directo al motor. Los componentes de software son los siguientes:

Database server (1): Un servidor SQL Postgres donde se almacenan telemetría, telecomandos y sus definiciones para todas las misiones.

Application server (1): Un servidor de aplicación responsable de la generación de interfaces y lógica de operación.

Telemetry and Telecomand Processor (n): modulo encargado de decodificar, calibrar y transformar en variables de ingeniería la telemetría según su definición y de codificar y enviar los comandos al segmento de vuelo. En este caso existe un proceso por satélite incluso si son del mismo fabricante. Para el caso de *FS2017* debe conectar al puerto 3210 por TCP/IP tanto para recibir telemetría como para el envío de comandos.

Figure 1: *Arquitectura*



La figura 1 muestra un esquema de alto nivel de la arquitectura.

III. Python y Django

Python es un lenguaje portable, abierto, de alto nivel. Es dinámico y tiene orientación a objetos no estricta (Permite programación estructurada). Concebido durante los 80 se popularizó en los 90, muestra una performance superior a otros interpretes, producto de su código intermedio (bytecode). Además de la portabilidad, que no mostró ninguna fisura, su punto fuerte es la gran comunidad que brinda soporte y la buena integración con entornos de desarrollo como Eclipse. El lenguaje tiene buena productividad [3], estructuras complejas de datos ya incorporadas, manejo de excepciones, recolector de basura y capacidad de reflexión, mandatoria para cumplir con el objetivo. Otros beneficios de trabajar con Python, o con otro lenguaje maduro de propósito general, son las capacidades de depuración, los test unitarios y la generación de documentación automatizada. Este enfoque además permite pleno acceso a herramientas de comunicación como Sockets, http, RPC (Remote Procedure

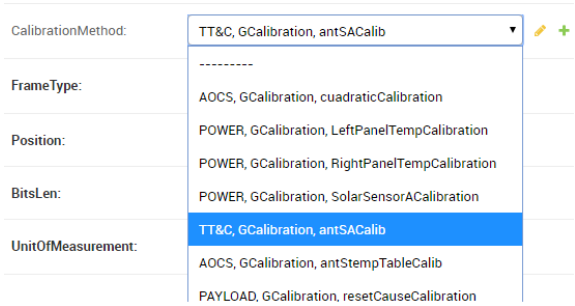
Call), Web Services, email y a bibliotecas de manejo de archivos XML entre otros.
Django es el framework web para python más popular, además de ofrecer herramientas para el desarrollo web incorpora un robusto ORM que se ha usado intensivamente en todos los módulos del segmento terreno, incluyendo los que no tienen interfaces web. Django propone un desarrollo basado en modelos, con una orientación a objetos estricta, pudiendo aprovechar las capacidades de abstracción, herencia, polimorfismo y las ventajas de un mejor modelo de reusabilidad. El acceso al motor de base de datos es transparente al desarrollador de la aplicación como al operador, ingeniero o equipo encargado de desarrollar los scripts de comandos, toda la responsabilidad sobre la recuperación y persistencia de datos recae sobre el ORM.

IV. Telemetría

Si bien la arquitectura del segmento terreno es agnóstica de la fuente de datos se ha utilizado como segmento de vuelo el *FS2017*. El *FS2017* envía la telemetría en tramas AX.25 por TCP/IP [1] puerto 3210 según especificación del fabricante. La trama tiene un fragmento de *Payload* donde viene codificada la telemetría. La definición de la telemetría esta persistida en una entidad *(TlmyVarType)* donde se establece el tipo, valor, rangos, límites, posición dentro del fragmento de *payload* y la función que la transforma en una variable de ingeniería. En el área satelital y en algunos sistemas de control y automatización es común recibir la telemetría en valores crudos o *raw*. A estos valores crudos se les debe aplicar una función que los transforme en unidades de ingeniería. En algunas áreas, incluida la satelital, esta transformación también es utilizada para calibrar sensores o equipos que puedan sufrir desgaste o variaciones por las condiciones de uso. Estos ajustes pueden ir desde un simple ajuste por función lineal hasta una discretización de valores por tablas de *look-up*. El sistema debe ser lo suficientemente flexible como para permitir

[1] Alternativamente se puede obtener por puerto serie

Figure 2: *Selección de método de calibración*



aplicar cualquier función de transformación a toda variable de telemetría y permitir ajustarla a medida que el desgaste de los sensores lo requiera. La figura 2 muestra como al configurar una variable de telemetría se selecciona la función a aplicar para convertirla en variable de ingeniería.

Mediante técnicas de reflexión se carga en tiempo de ejecución la función seleccionada de calibración y se le aplica al valor *raw* extraído de la trama de telemetría proveniente del paquete AX.25. La función de calibración puede ser cualquier secuencia de comandos programable en python sin ningún tipo de restricción mas allá del tiempo de procesamiento. Todas las bibliotecas y estructuras de datos están disponibles incluyendo funciones matemáticas e incluso el acceso al la base de datos completa mediante ORM. Al tener acceso a la base de datos se pueden obtener coeficientes actualizados de calibración permitiendo:

Reutilizar funciones: es posible crear una única función para ajustes típicos (lineal, cuadrático) y reutilizarla con distintos coeficientes según la variable de telemetría requiera.

Calibración fina: creada una función modificando los coeficientes se puede realizar un ajuste fino, por ejemplo por desgaste de un sensor, sin necesidad de recodificar la misma.

Como ejemplo, el segmento de vuelo *FS2017* requiere que a muchos de sus valores crudos

se les aplique una ganancia (*GAIN*) y un desplazamiento (*OFFSET*). En todos los casos se realiza mediante una única función llamada *linealCalibration* aunque cada tipo de variable de telemetría posea sus propios valores para ambos coeficientes.

Para que un método sea considerado de calibración o ajuste debe estar desarrollado como método de una clase heredada de *BaseCalibration*. El software realiza periódicamente una exploración de todos los métodos públicos de clases derivadas de *BaseCalibration* y los disponibiliza para su aplicación a las distintas variables de telemetría 2. La figura 6 muestra la clase *GCalibration*, heredada de *BaseCalibration*, donde se implementan algunos métodos de calibración incluido el seleccionado en la figura 2.

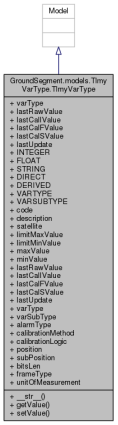


Figure 3: *Clase TlmyVarType*

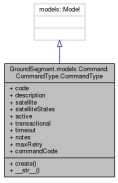


Figure 4: *Clase CommandType*

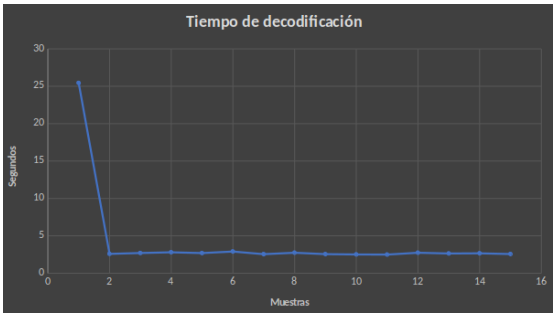
El siguiente código muestra parte de la implementación del método *setValue* de la clase *TlmyVar*.

```
class TlmyVar(models.Model):
  ...
  if raw!=self.tmlyVarType.lastRawValue:
    self.tmlyVarType.lastRawValue = raw
    if self.tmlyVarType.calibrationMethod:
      if not self.tmlyVarType.calibrationLogic:
          klass =
            globals()[self.tmlyVarType.calibrationMethod.aClass]
          instance = klass()
          methodToCall =
            getattr(instance,
                    self.tmlyVarType.calibrationMethod.aMethod)
            self.tmlyVarType.calibrationLogic =
              methodToCall
      else:
          pass #Calibracion ya cargada

      if self.tmlyVarType.varType==self.tmlyVarType.INTEGER:
          self.tmlyVarType.lastCalIValue =
              self.tmlyVarType.calibrationLogic(self.tmlyVarType,  raw )

    elif self.tmlyVarType.varType==self.tmlyVarType.FLOAT:
        self.tmlyVarType.lastCalFValue =
          self.tmlyVarType.calibrationLogic(self.tmlyVarType,  raw )
    else:
        self.tmlyVarType.lastCalSValue =
          self.tmlyVarType.calibrationLogic(self.tmlyVarType,  raw )
  else:
      if self.tmlyVarType.varType==self.tmlyVarType.INTEGER:
          self.tmlyVarType.lastCalIValue = raw
      elif self.tmlyVarType.varType==self.tmlyVarType.FLOAT:
          self.tmlyVarType.lastCalFValue = raw
      else:
          self.tmlyVarType.lastCalSValue = raw

  """
  Si el tipo no es cadena llevo el dato a cadena
  """
  value = self.tmlyVarType.getValue()

  if self.tmlyVarType.varType!=self.tmlyVarType.STRING:
      if (value>=self.tmlyVarType.limitMaxValue and
          value<=self.tmlyVarType.limitMinValue):
          raise Exception("Invalid value in var "+
                          self.tmlyVarType.code)

  if saveifchange:
      self.tmlyVarType.lastUpdate = datetime.now(utc)
      self.tmlyVarType.save()


  if self.tmlyVarType.varType==self.tmlyVarType.INTEGER:
      self.calIValue = self.tmlyVarType.lastCalIValue
      self.calSValue = str(self.calIValue)
  elif self.tmlyVarType.varType==self.tmlyVarType.FLOAT:
      self.calFValue = self.tmlyVarType.lastCalFValue
      self.calSValue = str(self.calFValue)
  else:
      self.calSValue = self.tmlyVarType.lastCalSValue



  return self.tmlyVarType.getValue()
```

Cuando se recibe un nuevo valor de telemetría se persiste mediante una instancia de *TlmyVar*. Toda recepción de nueva telemetría se registra en el histórico, haya cambiado o no su valor. También se deja el ultimo valor en la variable *TlmyVarType*, esta desnormalización tiene como objetivo ganar eficiencia al consultar los valores de tiempo real. En lugar de consultar a la tabla de histórica de valores recibidos, se consulta a la tabla de tipos que tiene un tamaño acotado. Si el valor *raw* para una variable determinada no cambio desde la ultima actualización entonces se guarda el registro de la recepción pero no se aplican las calibraciones. Esto ultimo hace ganar eficiencia ya que no se vuelve a transformar cuando se sabe que el resultado sera el mismo, se debe tener en cuenta, sin embargo, que ante un cambio en las funciones de calibración se debe forzar mediante software a una nueva evaluación aunque el *raw* no haya cambiado[2]. Si se produce un cambio en el *raw* se carga dinámicamente la función de ajuste

[2]No implementado aún

Figure 5: *Tiempo de decodificación*



self.tmlyVarType.calibrationLogic solo si no fue cargada anteriormente.

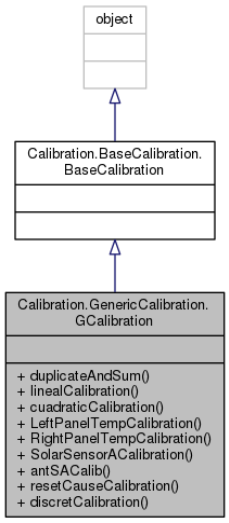
La carga del método de calibración a una atributo del tipo de variable de telemetría se realiza, si no ocurren cambios, una única vez durante la ejecución. Esto también permite mejorar los tiempos de ejecución ya que evita la carga por cada valor de telemetría recibido. Con una frecuencia configurable, el software analiza si alguna función de calibración fue actualizada, si asi ocurriera se procede al limpiar el atributo que contiene la función para forzar su recarga [3]. La figura 5 muestra los tiempos de procesamiento para un bloque de 15 paquetes donde se decodifican 5000 variables de telemetría, cantidad compatible con un satélite científico de envergadura.

La primera decodificación demora mas que las siguientes dado que tiene que realizar la carga de las funciones de transformación. La primera telemetría recibida difiere de la anterior y al ser la primera calibración la función no esta precargada. Luego el tiempo se estabiliza entre 3 y 3.5 segundos para el conjunto de las 5000 variables con una probabilidad de cambio de 10% entre muestra y muestra. El tiempo de procesamiento incluye además la verificación de que los valores de las variables de ingeniería estén dentro del rango de seguridad y el almacenamiento en el histórico. Los pruebas han sido realizadas en maquinas virtuales sobre equipos de escritorio.

El siguiente código muestra una calibración lineal (que puede ser usada en muchas tipos

[3] No implementado aún

Figure 6: *GCalibration*



de variable de telemetría) y que obtiene sus coeficientes utilizando el ORM.

```
#Clase GCalibration hereda de BaseCalibration
class GCalibration(BaseCalibration):
    ...

    #Metodo generico de calibración lineal
    def linealCalibration(self, obj, raw):
        #Multiplica el valor raw por la ganancia y
        #offset configurado para ese tipo de
        #de variable de ingeniería. Los obj tiene
        #el tipo de variable de ingeniería y
        #por medio del ORM se accede a los
        #valores configurados.
        return raw*
            obj.coefficients.get(code="GAIN").value +
            obj.coefficients.get(code="OFFSET").value
```

V. Telecomandos

En el caso del *FS2017* los telecomandos deben ser enviados al segmento de vuelo por el mismo canal en donde se recibe la telemetría, TCP/IP puerto 3210. Los comandos deben ser codificados en una trama AX.25. Para permitir el la creación de scripts de comandos, de la misma forma que con la telemetría IV se utilizaran las capacidades de reflexión para analizar, en tiempo de ejecución los scripts a ejecutar. Los scripts de comandos pueden ejecutarse por acción explicita de un operador o porque fueron aplicados a una pasada. El operador tiene pleno acceso al ORM de donde

puede obtener el diccionario de comandos y los valores de telemetría si necesitara aplicar condicionales que dependieran del estado del segmento de vuelo u otro valor disponible en el sistema. El siguiente código muestra como se envía un comando de encendido de *heater* si la temperatura de la OBC esta por debajo de un valor determinado.

```
#Instancio el satélite FS2019
sat = Satellite.objects.get(code="FS2019")
...

if not sat.isConnected():
  raise Exception("Abort, flight segment offline")

#Consulto si la temperatura de la OBC esta por
#debajo de 10 grados y tambien
#Verifico que el heater este apagado
if(sat.tmlyVarType.get(code="obcT1").getValue()<
  sat.parameters.get(code="MinTolOBCTemp"))
  and
  (sat.tmlyVarType.get(code="HeaterOn").getValue()
    ==False):

   #Creo un nuevo comando para el satelite, con fecha
   #de vencimiento 5 minutos desde la
   #fecha de creación.
   #Es un comando de ejecución en tiempo real
   #y por tanto no se agrega un tercer parametro
   #de fecha de ejecución
   cmd = sat.SendRTCommand("HeaterTurnOn",
                           datetime.utcnow()+
                           timedelta(minutes=5))
```

Por consola se obtiene el satélite con el que se desea trabajar. Se consulta por la temperatura de la OBC mediante una lectura del valor en tiempo real de la variable de telemetría *obcT1*. Si ese valor es inferior a limite parametrizado (El método *getValue()* siempre retorna variable de ingeniería) y los *heaters* esta apagados entonces se envía un comando de encendido. Sentencias como *sat.tmlyVarType.get(code="obcT1").getValue()* que retornan la instancias de *TlmyVarType* son provistas por el ORM de django a pesar de ser utilizadas por consola.

VI. Resultados

i. Interprete y Framework de desarrollo

El lenguaje (python) es sencillo aprender, pero por sobre todo, esta disponible una extensa comunidad con ayudas ante los problemas o desafíos que puedan aparecer. Para las posibles limitaciones es fácil encontrar soluciones alternativas (*workarounds*). Las experiencias realizadas con lenguajes de propósito especifico requirieron de la asistencia constante de un experto. Trabajando sobre python esto no fue necesario y la mayoría de los problemas pueden ser resueltos mediante información disponible en la comunidad. Las herramientas de depuración y análisis se mostraron poderosas, incluso superando las expectativas, como contrapartida el hecho de ser un lenguaje dinámico limita las capacidades de *Code Insight/IntelliSense* que el entorno de desarrollo puede ofrecer.

ii. Desarrollo multiplatafoma

La solución fue ejecutada en ubuntu linux 15.10 como en windows 10.0 y windows 7.0. Se realizaron pruebas sobre dispositivos físicos y virtuales. En todos casos el software se ejecuto sin cambios. En este punto tanto el interprete como el framework mostraron uno de sus atributos mas destacables.

iii. ORM

El ORM de Django se mostró estable, se registraron pocos problemas o *bugs* y cuando ocurrieron se encontró información disponible que lo documentaba. Para la creación y modificación de modelos, su flujo divido en dos etapas [4] *Make Migrations* y *Migrate* permitió el trabajo colaborativo y con múltiples servidores. El acceso a datos requiere de la práctica de convenciones propias que, aunque muy bien documentadas, afectan la curva de aprendizaje por poseer particularidades propias de la filosofía de convención por sobre configuración *Convention over configuration/coding by convention*. Por otro lado el modelo no acepta atributos privados, cuestión que afecta el encapsulamiento.
La capacidad del ORM para trabajar con polimorfismo esta limitada y se requiere añadir paquetes especiales, este ultimos puntos se

[4] Disponible en la versiones de Django iguales o superiores a 1.7

pueden considerar como una limitación de relevancia.

iv. Reflexión

Quizá por ser un lenguaje híbrido y su definición algo mas antigua, se juzga a los mecanismos de reflexión disponibles en python mas complejos que los disponibles en otras herramientas como C#.Net. Estos, sin embargo, fueron suficientes para cumplir el objetivos. Los tiempos de respuesta no presentan una limitación a su implementación. La reutilización de métodos previamente cargados colabora en mantener los tiempos entre margenes aceptables.

VII. Discusión

Los lenguajes específicos del sector espacial fueron creados hace décadas. Durante los 70, si bien existían opciones interpretadas , están no eran de uso extendido. En la actualidad existen varias opciones con una amplia cantidad de usuarios de base y múltiples proyectos que avalan su robustez. Python, Perl, VBScript, JavaScript son solo algunas opciones. Incluso la técnica aplicada en este trabajo es implementable, con algunas restricciones, en lenguajes compilados, siempre y cuando tengan capacidades de reflexión avanzadas. Java o C#, herramientas que generan código intermedio son aplicables así como compilados puros. Delphi mediante su RTTI (Run-time type information) para citar un ejemplo, requeriría que los métodos de calibración sean compilados previamente para luego ser cargados dinámicamente como complementos o *plug-ins* durante la ejecución del modulo principal. A esta opción se la juzga mas compleja y si bien puede ofrecer mejoras en cuanto a rendimiento, los resultados no justifican su implementación. Las opciones ampliamente probadas de la actualidad no justifican el desarrollo de un lenguaje propietario, no tener una base de usuarios implica ausencia de documentación, soporte, herramientas y por sobre todo recursos como explica [4] refiriéndose a ADA. “Using Ada could potentially offer lower life cycle costs compared to other programming languages, but it seems more likely that using Ada would raise life cycle costs due to a dearth of tools and compilers and lack of trained, experienced programmers”. La creación de una herramienta propietaria limita las capacidades disponibles en términos de estructuras de datos y bibliotecas en general y obliga a su desarrollo con el costo que esto conlleva y su aprendizaje carece de valor al ser inaplicable por fuera del ámbito donde se implemente.

References

[1] Garcia, Gonzalo, *Use of Python as a Satellite Operations and Testing Automation Language*, GSAW2008 Conference, Redondo Beach, California, 2008.

[2] Galal, Ken and Hogan, Roger P, *Satellite Mission Operations Best Practices*, Space operations and support technical committe american institute of aeronautics and astronautics, 2001.

[3] Prechelt, Lutz, *An empirical comparison of seven programming languages*, IEEE, 2000.

[4] Smith, II and others, *What About Ada? The State of the Technology in 2003*, Software Engineering Institute, 2003.