DOI 10.5162/ettc2018/3.3

A JSON transactional data server for flight tests

A. Óscar Gigato Rodríguez[1]
[1] *Airbus Defence and Space Sevilla-SPAIN*
oscar.gigato@airbus.com

Abstract:
Airbus Defence and Space has been using a protocol for gathering Flight Tests data, called FxS (Flight Test Data Exchange Service), based on XML and transactions among clients and servers. An evolution of this protocol is proposed, covering current implementation desired characteristics, like being **object-oriented**, **metadata support**, **encryption** and/or **compression**, and based on **JSON** instead of XML (more lightweight).

Key words: object-oriented, JSON, encryption, metadata, compression.

Introduction

In Airbus Defence and Space we use a client/server model for gathering data from a Flight Test, because of its tremendous advantages, such as centralized store of data, ease of backup's processes, access control and so. Since beginning of 2000's, we have been developing a protocol called FxS (Flight Tests Data Exchange Service), in order to use the same layer for access different data formats stored on heterogeneous systems.

After more than 15 years after entry into production of FxS data servers, we think that it's time to an evolution of this concept, keeping the good ideas, and introducing some new technologies, in order to fix a few drawbacks detected during this period, so we can maximize performance and reliability of our systems, and give the best possible service to our users.

In general lines, we can proudly say that FxS data servers works very well and stable, but we want to polish it even more.

A bit of history

Before the developing of FxS, access to data from flight tests was performed using proprietary applications developed in-house, with direct access to data. That is, tools used by analysis team had to have visibility of stored data, with no intermediary. And each ground station had its own suite of applications, being incompatible among them.

In this situation, former CASA and DASA ground stations teams met and agreed the development of a new interoperable protocol, so that an engineer of one team can connect to data of the other team without problems. Then FxS was born. The key of this concept was to create a new layer over stored data, and harmonize the way data is accessed by both ground stations teams.

The concept of „Data server"

With the development of FxS, data server architecture was also deployed. Instead of access data directly, so that each client had to have visibility of stored data, from now on, a new actor enters into scene. The data server is the only application that has access to all data repository, and clients have to connect with it, if they want to gather stored data.

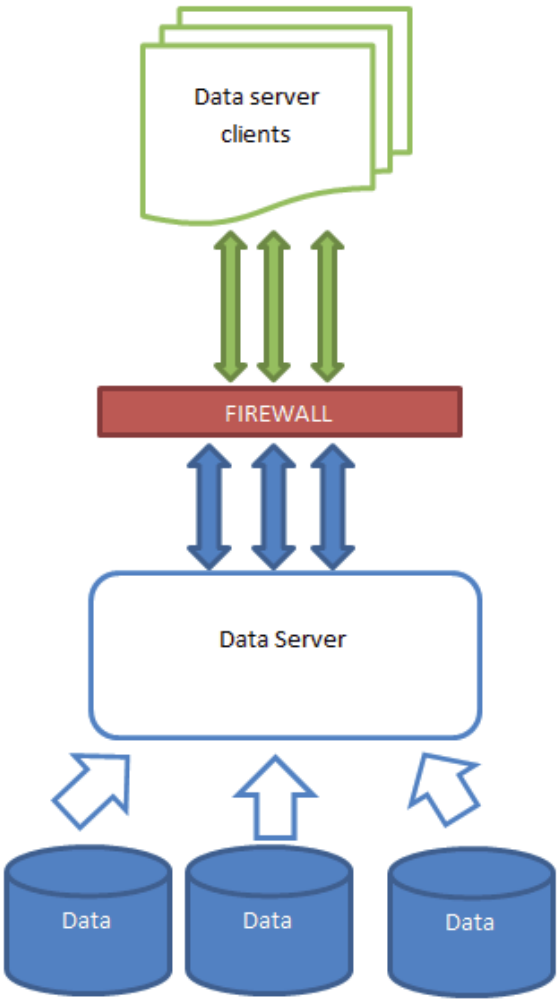

Fig.1: Data server architecture.

Among the multiples advantages of this philosophy, we have:

- Permissions management is now centralized on data server itself. This simplifies this task enormously.
- Clients only need to "speak" FxS. It permits access to different formats of data, in different systems.
- Clients read only the data they need. They have no longer to access all the data for gathering, for example, 4 parameters. The data server have access to all test, but only delivers required parameters, at the required rate during the required period of time. This saves enormous bandwidth and time.
- Some data of flight tests have been recorded in "raw format", and need to be converted to engineering format. This is called "calibration". Before data server deployment, calibrated parameters can be either directly stored in calibrated units, or having to be calculated by clients. Now, it is normally the data server who performed these calculations.
- Use of this architecture permits to be independent from Operating System and hardware. Clients only have to know the FxS protocol. No more.

However, during all this time, we have detected some lacks or drawbacks that have to be fixed:

- Use of metadata is mandatory. In some situations, we have to reserve some valid values to indicate that data is not valid, for example. Metadata shall avoid this problem.
- Full support of strings. Actually, we need to make some workarounds in order to use strings.
- Opaque parameters native support. Sometimes, data recorded in one parameter is the traffic from one data bus, and thus it does not have a fixed size.
- In offline mode, use of only one socket will simplify passing through a firewall.
- Compression and/or encryption is also desirable when using offline data server from a remote location.
- User identification can be stronger.

Today: The FxS protocol

FxS (Flight Test Data Exchange Service) protocol is based on XML transactions among server and clients. All messages are validated before processed, using XML schemas. This makes all transactions consistent, and prevents use of messages that don't fully respect FxS specification.

The following transactions are defined:

- Status Announcement.
- Client Identification.
- Special parameter.
- Data File List.
- Data File Info.
- Parameter List.
- Tag-Size List.
- Program/Setup.
- Offline Test.
- Start.
- Stop/Close.

- Pause.
- Resume.
- Remove.
- Rate.
- Refresh.

Current data server uses two channels with the client. The first one, TCP, is reserved for transactions, and the second one, UDP unicast or multicast, is reserved for data transmission.

Introducing a new data server architecture

Keeping in mind all the advantages of FxS data server, I propose a new (evolutionary) architecture for the data server. It tries to overcome all the drawbacks of current software, as well as maximize both speed and network bandwidth efficiency. It also tries to simplify both transactions and connections management. The way it addresses these points is:

- Use of JSON instead of XML (lighter messages, faster transactions generation).
- Use of only one TCP port in OFFLINE mode (it eases tunnel traffic forwarding for use behind a firewall).
- In ONLINE mode, it uses an UDP connection for data, just like the current data server. But this connection is made by the client to the server (in current data server, is the server who opens this port). This avoids some problems with local firewalls (in client equipment).
- Use of compression is supported in OFFLINE mode. Although LZMA2 is recommended, there is no obligation to use that one.
- Redesign some transactions.
- Permit the use of TTD and CVT data in the same connection (current data server needs two different connections for this purpose).
- Things that work quite well will be maintained, like XML use for configuration files. In this case, XML files are more readable for a human, hence this decision.
- Time for samples sent by data server will be in PTP time format, an 8 byte time specifications who allows precisions of nanoseconds, while specifying the year too.

The JSON Data Interchange Syntax

JSON (JavaScript Object Notation) is a lightweight, text based, independent syntax for defining data interchange formats[1]. Is like XML, but simpler and lighter. It is harder to read for a human, but in this case, these messages will not be read by persons. It is also faster to parse and generate, and there are too many libraries in many programming languages that support this standard. Possibility of validation (just like XML schemas) exists, as there is a draft of a standard for validation.

An example comparing XML and JSON:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Fig.2 A sample JSON definition.[2]

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
  <gender>
    <type>male</type>
  </gender>
</person>
```

Fig.3 A sample XML definition.[2]

As shown in previous figures, for the same definition JSON uses less character, thus making it faster for transmitting it over the network. Clearly XML is powerful than JSON, but in our case, is enough with JSON, and so we can worth its advantages over XML.

Transactions for the new data server

Some of the transactions defined in the new data server will be redesigned. One of them disappears, while a new one is created. Transactions contents might vary in its final implementation.

Tab 1 Comparison between transactions.

FxS Transaction	New transaction
Status Announcement	Status Announcement
Client Identification	Client Identification
Special Parameter	-
-	Connection Options
Data File List	Tests List
Data File Info	Test Info
Parameter List	Parameter List
Tag-Size List	-
Program/Setup	Program
Offline Test	Offline Test
Start	Start/Retrieve
Stop/Close	Stop/Close
Pause	Pause
Resume	Resume
Remove	Edit Parameter List
Rate	Modify Rate
Refresh	Refresh

There is something more than a change of names. Special Parameter has no sense since now we have metadata sent associated to parameters values. Let's explain these transactions:

- Status Announcement: This one does not change. Data server sends, each second, a broadcast message for announcing itself.
- Client Identification: This transaction identifies the client in the data server. It is useful for both audit and access control purposes.
- Connection Options: Used for negotiating connections aspects, like compression (only for OFFLINE), local UDP port for ONLINE mode, etc.
- Tests List: In OFFLINE mode, this transaction returns the list of available tests, according with access permissions of the user.
- Parameter List: The client asks the server for the list of parameters using this transaction. Regular expressions can be used.
- Program: This transaction tells the server the list of parameters that have to be sent to the client. It also specifies each parameter mode (CVT, TTD), and rate (only for CVT).
- Offline Test: Specifies the server the test to work with, and the time span to be used (within test time limits).
- Start (ONLINE)/Retrieve (OFFLINE): Makes the data server start sending the data to the client.
- Stop (ONLINE)/Close (OFFLINE): Makes the data server stopping the data transmission. After data stop, connection with the client is closed.
- Pause: Pauses data sending.
- Resume: Resumes data sending.
- Modify Rate: Changes the rate for one or more parameters (CVT mode).
- Refresh: Forces the send of last value for all programmed parameters, in CVT mode. Normally, only changed values are sent.

Transfer of data

Due to the great amount of data that can be transmitted over the network, binary connection is the most efficient one. We define two different frames, although in the same channel, for both CVT and TTD modes. This allows the use of these two modes in the same connection. In OFFLINE mode, only one port is used by each client, for both transactions and

data connection. Each message specifies the type of information before the payload. That is, for a transaction, a TRANSACTION identifier is prepended. For data, a DATA identifier is also prepended. All information is sent in a TCP connection. Here is the frame definition, for both CVT and TTD modes.

Tab. 2: CVT DATA frame definition

	SIZE (IN BYTES)	DESCRIPTION
FRAME TYPE	1	VALUES: TRANSACTION, DATA, STATUS
FRAME SIZE (IN BYTES)	2	-
CURRENT TIME	8	PTP TIME
NUMBER OF PARAMETERS	2	NUMBER OF TAG/VALUE PAIRS
TAG 0	4	FIRST TAG SENT
METADATA 0	1	FIRST VALUE METADATA
SIZE 0 (OPTIONAL)	2	SIZE OF FIRST VALUE
VALUE 0	VARIABLE	FIRST VALUE SENT
…	…	…
TAG N-1	4	LAST TAG SENT
METADATA N-1	1	LAST VALUE METADATA
SIZE N-1 (OPTIONAL)	2	SIZE OF LAST VALUE
VALUE N-1	VARIABLE	LAST VALUE SENT

Tab. 3: TTD DATA frame definition

	SIZE (IN BYTES)	DESCRIPTION
FRAME TYPE	1	POSSIBLE VALUES: TRANSACTION, DATA, STATUS
FRAME SIZE (IN BYTES)	2	-
CURRENT BASE TIME	8	PTP TIME
TAG 0	4	FIRST TAG SENT
DELAY 0	2	DELAY OVER BASE TIME (ns)
SIZE 0	2	SIZE OF VALUE 0
METADATA 0	1	FIRST VALUE METADATA
VALUE 0	VARIABLE	FIRST VALUE SENT
…	…	…
TAG N-1	4	LAST TAG SENT
DELAY N-1	2	DELAY OVER BASE TIME (ns)
SIZE N-1	2	SIZE OF LAST VALUE
METADATA N-1	1	LAST VALUE METADATA
VALUE N-1	VARIABLE	LAST VALUE SENT

When sending transactional data, first byte indicates that it is a transaction, next two bytes gives frame length, and just after this information, JSON serialized message is appended.

The only difference between ONLINE and OFFLINE frames, is that the first one does not use TRANSACTION type of frames.

There are two aspects that I want to emphasize:

- Strings and variable size data are permitted. When necessary (in CVT mode), a size field will be used. The existence of this filed can be checked in metadata.
- Metadata includes some flags, like Valid Data Flag, No Data Flag, Size field present flag, etc.

ONLINE (real time) mode

We must say “quasi real time” mode, so we are using neither a real time device nor Operating System. In this mode, we use two connections per client. The first one, TCP, will transmit transactions between client and server. And a second one, UDP, will be used to send data.

This is due to TCP connections are designed for reliability, but not for quick response. In order to avoid problems with local firewalls, both connections will be open by the client.

As this mode will be used mainly in local area networks, compression algorithms shall not be provided.

Once the client sends the Start transaction, data server begins sending programmed data at the desired rate. Only the data that has changed since the last data sent is delivered. This saves both bandwidth and CPU resources in client and server.

If the client wants the server to send all parameters values, it can send a Refresh transaction to the server. Upon the reception of this message, the server delivers last acquired parameters values.

Client can also pause and resume data delivering to itself, by sending both Pause and Resume transactions.

During the receiving of data from the data server, client can send an Edit Parameter List transaction, adding/deleting parameters to program. Upon the receipt of this transaction, data server will refresh all data to his client.

Once the client no longer needs data from the server, it sends a Stop transaction, making it to finish data sending, and the subsequent close of connection.

OFFLINE (processed) mode

In this mode, clients will require previously stored and processed data for its study by specialists. As we can be connected from both LANs and remote locations (remote location means other center, for example), only one TCP port will be used. This eases port forwarding over a SSH tunnel, for example.

In addition to this, compression can be enabled using the Connection Options transaction.

The usual lifecycle for OFFLINE data servers starts receiving a Client Identification transaction from one client, then a Connection Options one. After this, normally a Test List Request is made, and then a Parameter List can be made (is optional). Next step is an Offline Test Request, specifying test and time slices required, followed by a Program request.

Once the client sends the Start transaction, the server will send it all required data, at the fastest speed permitted by both client and server.

Like in ONLINE mode, when a Stop transaction is sent, data server will close its connection to the client.

Frame Type field is used to distinguish among TRANSACTIONS, DATA or STATUS messages.

Future: SSH tunneling and Web Services

Tunneling over a SSH connection could be made by OFFLINE data server. This approach is useful for encrypting data and facilitates its pass through a firewall. Moreover, this tunnel can also provide compression, avoiding the use of it directly in the protocol.

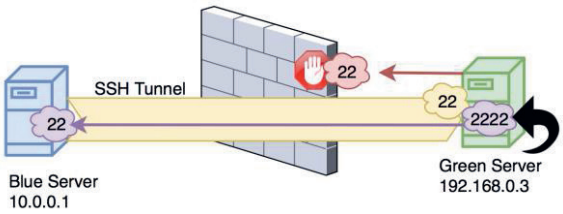

Fig. 4 An example of a SSH tunnel.

A Web Service is a technology that enables interchange of information between two applications, using WEB standards. The client is normally run inside a web browser, which eases enormously deployment of client applications (every computer has a browser installed), security restrictions and so on.

Technologies like SOAP, JSON, AJAX, REST and others are used in web services.

The idea is to embed data server functionality into a web service, adapting the way it works to offer its services and make them accessible from web clients.

Making a data server operate as a web service opens a world of interoperable services, and the possibility of development of thin clients, that can be run from a laptop to a smartphone or tablet. For example, a light client for checking live parameters could be easily deployed in a tablet, or opened from a web browser. This would allow us checking parameters inside an aircraft in real time, from any place of it.

Today it's possible to implement this scenario, but web services make rather simple deploying clients in any site. Firewalls are not a problem using this approach.

Among drawbacks of using a web service, the main one could be performance loss, due to overhead created by HTTP/SOAP layers to transmit data to the clients.

References

[1] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[2] Wikipedia: https://en.wikipedia.org/wiki/JSON