Asynchronous messaging as backbone for the MCS

Noé Casas[1] and Carlos Estévez.[2]
GMV Aerospace

As the size of satellite fleets grows every year, so do the telecom operators' expectations regarding Mission Control System performance, reliability and flexibility. This manuscript describes GMV's experience in an ongoing R&D activity to improve the software architecture of its spacecraft fleet management suite, *hifly*®, by migrating its current communications layer to a Message-Oriented Middleware, in order to face the challenges posed by future operational requirements in the upcoming years.

I. Introduction

Based on their experience in satellite operations, the European Sp ace Agency developed their current generic mission control software suite, called **SCOS-2000**, taking into account the insight obtained while developing and operating its predecessors MSSS, SCOS-I and SCOS-II. SCOS-2000 is used to control ESA missions such as Radarsat 2, XMM Newton, Integral, MSG, Cryosat, GOCE, Herschel Plank and Rosetta.

On 2002 GMV forked SCOS-2000 version 2.3e to use it as starting point to create a new MCS for telecom satellite operator Eutelsat, intended to control its spacecraft fleet. Such development later derived into **hifly**, GMV's multi-mission control system. Although SCOS-2000 and *hifly* have diverged substantially over time, their software architecture remains very similar.

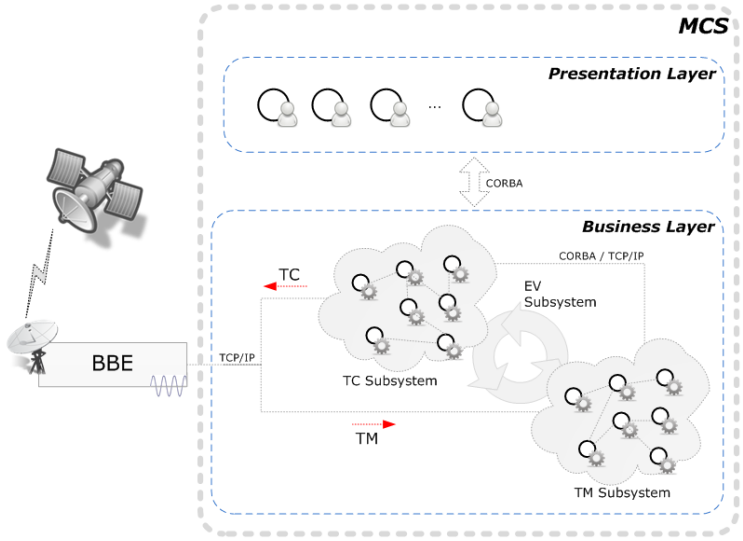

Figure 1 General organization of the Mission Control System

SCOS-2000 *and hifly* have **distributed process architecture**, using **CORBA** and raw **TCP/IP** as mechanisms for communication between different processes.

TCP/IP is mainly used in cases where performance is top priority and where the transmitted information is close to the satellite communication protocol. The distribution and retrieval of spacecraft telemetry or telecommand data are two examples of where raw RCP/IP is used.

CORBA is used where the information is close to the MCS operative semantics, usually between services and client applications.

There are three types of information flowing through the MCS that are especially important:

- **Telemetry data**: data associated with the state of the spacecraft, e.g. the temperature of thruster 1 in Kelvin.
- **Telecommands**: commands generated from the mission control system to be executed by the spacecraft.
- **Events**: information messages generated by the MCS subsystems for the operator, e.g. no telemetry data has been received in the last 30 seconds.

[1] Software Architect, Satellite and Mission Control Business Unit, ncasas@gmv.com.
[2] Software Architect, Satellite and Mission Control Business Unit, cestevez@gmv.com.

In SCOS-2000 and *hifly*, such operational data is wrapped in information containers called **S2K packets**, which are distributed to any interested MCS component and filed in the packet archive. Packet consuming applications can subscribe to LIVE packets and can retrieve historical ones. Client applications do not normally work at packet level, but the intermediary **services layer** digest such packets and provide high level information structures. In this way there is no need for client applications to be aware of the packet management subsystems or the encoding format of the packets themselves.

Our experience extending and maintaining SCOS-2000 and *hifly* has shown us that both raw TCP/IP and CORBA present some **problems** that make it hard to work with them.

Working directly with **TCP/IP** forces the developer to handle by hand **low level** details that are not easy to manage, such as:

- Data framing: the TCP/IP communication is stream-oriented, thus the data units have to be segmented, either by making the data unit size fixed, by specifying it in the data itself, or by introducing some sort of synchronization mark.
- Data format: in SCOS-2000 and *hifly*, TCP/IP communications mainly rely on XDR serialization standard to encode/decode messages other than spacecraft TM/TC. XDR, as used in SCOS-2000 and hifly, is not a schema-driven representation layer, thus it is needed to write ad-hoc code that to parse and encode messages, which is not reusable from different programming languages (unless language bindings are used).
- Point-to-point: communication has two clear endpoints; if either of them is down or the connection is broken, the communication will not take place.
- Network failures and latencies: TCP is meant to be tolerant to network failures, and in case the connection suffers problems, the protocol stack will retry the communication, until certain timeout is elapsed; normally such timeout is in the order of minutes. The networked application logic has to take that fact into account and set custom timeout mechanisms in case they are needed.
- Reconnection: the developer has to manually handle the cases where the connection is lost (e.g. when no answer is received after a certain length of time). The error can be detected either by the protocol stack or by custom timeout mechanisms.
- Static port assignation management: when a distributed system scales, the number of required ports can be in the order of hundreds. Managing the configuration of the system to avoid port clashes can be troublesome especially when several instances of the system can coexist on the same machines. In those cases each instance has to be given its own port range.
- Firewall friendliness: network security policies tend to oppose opening TCP ports for external access. The management of port access allowance rules can turn complex as the number of ports used grows.

CORBA also presents problems despite being at a higher level. The root cause is that it tries to **disguise network communications as local invocations**, that is, the very nature of remote procedure call paradigm. The most notable drawbacks of CORBA are:

- Favors ignorance of network problems: when inexperienced programmers face CORBA programming, they tend to ignore the fact that their local invocations result in network communications that may take an undefined amount of time to complete, or may fail. This is even worse when the concrete programming language CORBA mapping does not force the programmer to handle errors (such as in C++ and Java). This leads to code that is not robust to network total or partial failures, or latencies.
- Point-to-point communications: as in TCP/IP, both endpoints have to be available for the communication to succeed.
- Chatty protocols: at the source code level, CORBA invocations look like local ones, which causes developers to think that both are equally cheap. Because of their chattiness, the protocols do not behave well under high-latency networks, such as WANs; invocations come and go from side to side, each of them getting the impact of the network latency.
- Open source ORB implementations in some languages: our organization tries to use open source software in order to avoid vendor lock-in; however, the open source CORBA ORBs available for Java are not as mature as the ones available for C++.
- Firewall friendliness: as each servant occupies a TCP port, CORBA suffers the same problem as raw TCP/IP communication regarding firewalls. This may even be worse when using dynamic port assignation, as establishing static network traffic allowance rules becomes impossible.

Due to these problems and as part of the continuous improvement of *hifly* as a product, we have investigated suitable **replacements for TCP/IP and CORBA**. After some research, we identified the following mainstream middleware types: web services, Enterprise Service Buses (ESB), Message-oriented Middleware (MoM).

Web services are normally implemented using SOAP protocol, which is basically an XML-driven remote procedure call over HTTP. It suffers many of the same problems as CORBA, but adds the complexity of the protocol itself. Although proprietary SOAP development tools successfully hide such detail, open source ones tend not to reach an acceptable maturity level for the software industry productivity standards. Also, their throughput may be constrained by the transport used, and their high bandwidth usage may not be appropriate for the MCS.

Enterprise Service Busses aim at interoperability by introducing intermediate software layers. Unfortunately, their design reduces the determinism of the operational performance of the whole system, as they sometimes sacrifice performance in exchange of flexibility.

Message-oriented Middleware is *a priori* the most **suitable** replacement for TCP/IP and CORBA because:
- Network awareness: requests to remote services do not look like local ones, forcing the developer to write code that is robust in the face of the asynchrony posed by the network.
- Message delivery assurance: our code would be no longer responsible for dealing with retries and reconnections. It would be more tolerant to network failures and latency.
- Publish/subscribe communications paradigm, which is one of the cornerstones of MoM, fits very well in some communication patterns among MCS processes.

Using an asynchronous messaging system may open the door to opportunities for **evolving the system** that were not feasible before. Examples of such *enablers* are:
- Enterprise integration patterns, to rationalize the responsibilities of each component.
- Routing, to improve the architecture of the system, aiming at fault tolerance, scalability and throughput.
- Message flow monitoring tools, to boost our knowledge on the dynamic behavior of the system under operational conditions.

However, message-oriented middleware also presents some apparent **drawbacks**:
- The communication between some MCS processes is synchronous in nature, thus the publish/subscribe paradigm may not fit.
- The ignorance of client applications regarding who is processing their requests and whether or not they are being processed may not be acceptable for critical components such as telecommanding.

Given the plethora of different TCP/IP and CORBA point-to-point communications among SCOS-2000 and *hifly* processes, their hypothetical replacement may be divided according to the following finer grained and staged **subtask decomposition**:
1. Replace the TCP/IP-based S2K packet distribution layer.
2. Replace the CORBA-based communications between client applications and services.
3. Replace any remaining CORBA and raw TCP/IP communications among server processes.

As the cost of the migration of the complete communications layer of the MCS would be very high, GMV decided to start an internal, limited-scope **R&D activity** aimed at:
- Studying the actual suitability of MoM as replacement of TCP/IP and CORBA in the MCS.
- Selecting third party middleware to be used.
- Identifying potential problems, e.g. technology not mature enough.
- Devising “migration guidelines” that map common communication patterns used in TCP/IP and CORBA across *hifly* processes to proper functional equivalents using MoM.
- Gathering experience to design the final solution.
- Estimating the effort needed for the complete migration.

Such R&D activity is the subject of this paper. In the following sections we describe its scope and the details on how it was executed (section II); we expose the lessons learned during its development (section III); and we present opportunities for evolving the MCS that are made possible by the introduction of the message-oriented middleware (section IV).

II. Evaluation of MoM as replacement to raw TCP/IP and CORBA in *hifly*

A. Scope

In order to have the scope of the aforementioned R&D activity cover as many representative cases as possible while keeping its cost low, we chose to replace TCP/IP and CORBA from a whole functional area, namely the **events subsystem**.

Events are pieces of information that describe a situation detected by the MCS that should be known by the operator. When an application decides to "publish" an event it encodes the information associated to it (e.g. event message, severity, origin subsystem) in a S2K packet and makes use of the packet distribution subsystem to share it. Such an event packet is then distributed to any interested component, and is also stored in the historic files archive for retrieval later. A dedicated client application called the "Event Logger" shows in a graphical user interface the events that are being published by all MCS subsystems. The Event Logger also allows the operator to browse the historical events.

The study consisted of the reengineering of all the communications involved in the **generation** of events by applications, their **distribution** across the MCS in LIVE and their later **retrieval** from the historical archive. Such modifications altered the whole events subsystem, removing all other middleware but MoM. It enabled us to obtain feedback applicable to all subareas within the hypothetical complete replacement, and made it possible to evaluate the impact on the subsystem's performance.

The packet distribution and filing infrastructure in **SCOS-2000** is almost the same as *hifly*'s. The **S2K2EUD** project is incorporating *hifly*'s service layer into SCOS-2000. With these two facts in mind, the reengineering described in this section is almost directly applicable to SCOS-2000.

The current **generation and LIVE distribution of events** in *hifly* is depicted in Figure 2. It comprises of the following steps (see the circled numbers in the diagram):

1. The event generation is requested via CORBA by a client application to the Events Injection Service. This step only takes place in client applications; server ones start in step #2.

2. The Event Injection Service encodes the information sent by the client application in a S2K packet and sends it via TCP/IP to the Event Packets Distribution Subsystem. However, the information contained in the packet is not complete, as applications only specify the numeric code of the message; extra information will be added to the packet in the next step.

3. Once the Packet Distribution Subsystem (PDS) receives the S2K event packet, it *enriches* its contents by filling the missing information, and then stores it in the historic files archive database (3b) and, in parallel, forwards it to the Client Packet Distributer (CPD) (3a). The sole role of this process is to minimize the bandwidth used for LIVE packet distribution. In each machine there is an instance of the CPD process, to which PDS forwards every published packet. Each CPD forwards the packet on to any other interested process within the same machine. All these data exchanges between PDS and CPD are done by means of raw TCP/IP communications.

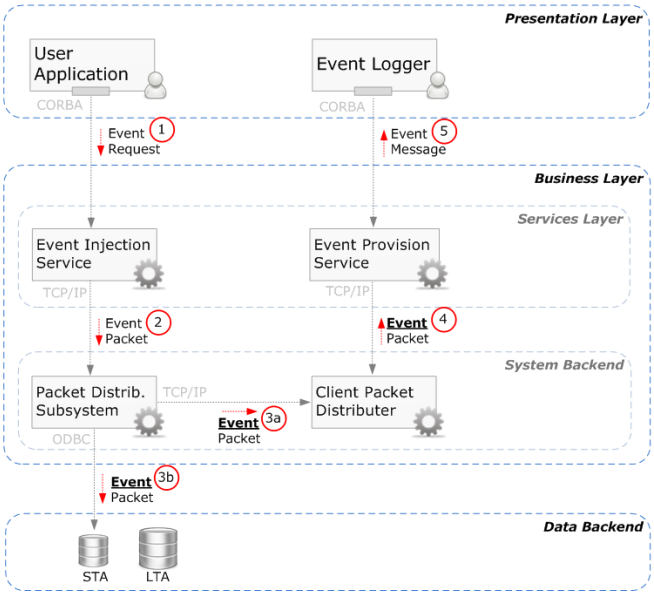

Figure 2 LIVE distribution of events in *hifly* (currently)

4. The Event Provision Service receives the S2K event packet from CPD via TCP/IP and extracts the information it carries.

5. Finally, the Event Provision Service forwards the event information it via CORBA to any client application that has registered its interest in LIVE events, such as the Event Logger.

The **retrieval of events** from the historic files archive in *hifly*, as it is currently, is depicted in Figure 3 and comprises the following steps (see the circled numbers in the diagram):

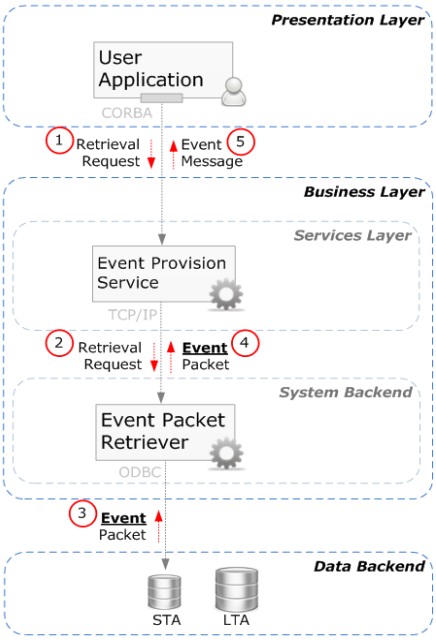

Figure 3 Access to historical events (currently)

1. The application that wants to retrieve historic events requests them from the Event Provision Service via CORBA, specifying the time range and any filter that is to be applied to the resulting events.

2. The Event Provision Service emits an equivalent event packet request to the Event Packet Retriever, usually referred to as Event Packets Historic File Archive (HFAEV), via raw TCP/IP.

3. The Event Packet Retriever issues a query to the packets database by means of ODBC (which then maps to the database-specific communication).

4. The Event Provision Service extracts the information contained in the S2K event packets.

5. Finally, the Event Provision Service supplies the requested events information to the user application as the result value of the initial CORBA invocation.

The **scope** of the activity comprises the migration of the communications depicted in steps 2 to 5 of events generation and LIVE distribution, and steps 1 to 5 of event retrieval, so only bus-based communications is used in them.

B. Technological considerations

Prior to the implementation, we needed to select a message-oriented middleware third party solution that could be relied on. Such piece of technology had to meet certain requirements regarding performance, availability of client libraries for certain programming languages (C++, Java and Python), flexibility, robustness, etc. Also, as we intend to avoid vendor lock-in, so we discarded proprietary COTS.

After some research, we identified three families of candidates:

- **JMS**: is a Java API to access the functionality provided by message-oriented middleware. There are several JMS open source implementations; however, from the ones we identified, only Apache ActiveMQ has bindings to our target programming languages.

- **AMQP**: is an open wire protocol specification for message-oriented middleware. Version 1.0 of the standard was published on October 2011. There are some open source implementations like RabbitMQ, OpenAMQ and Apache Qpid.

- **DDS**: OMG's specification for real-time publish-subscribe systems. There are some open source implementations like OpenSpliceDDS and OpenDDS.

OMG's DDS has very interesting QoS features, but its messaging system is constrained to pub-sub paradigm, it is not very interoperable and its data-centrism makes it difficult to suit the MCS in some cases. AMQP is a very young standard, not yet widely adopted and its implementations have incompatibilities. **Apache ActiveMQ**, although deemed unstable by some users, is widely adopted, has bindings to the programming languages we use, and presents

a very complete set of messaging features, such as broker federations and failover; for these reasons, we chose it as the messaging solution for our reengineering of the MCS.
Aside from the messaging, we also had to choose a **serialization framework** to encode and decode the messages that are sent through the bus. As we were concerned about performance, bandwidth usage and CPU consumption, we decided to drop text-based options, such as XML and JSON, and focused on binary ones.

We identified several options, such as Google Protocol Buffers, Apache Thrift and ASN.1 BER. From them, our preference is Protocol Buffers, as ASN.1 is not widely adopted outside of X.509 certificates and Thrift lacks unsigned integer types.

However, given the cost constraints in our reengineering activity, we decided to stick to the most affordable solution: we decided to keep **CORBA's data representation layer**, namely CDR, so we could reuse the types we already have for our current service implementations.

C. Development

Regarding the event generation and LIVE distribution, the reengineering comprised the following modifications:

1. The notification of events by client applications was kept unmodified, as there are very few cases.

2. The classes used by the Event Injection Service and the rest of C++ server processes to publish messages where modified to no longer use raw TCP/IP connections but instead send messages containing the event information.

3. As the events sent by applications only contain certain information (see paragraph A), we extracted the logic that filled the missing fields in the event structure from PDSEV and created a new process whose sole purpose is to do so, following the *message enricher* pattern.

4. This way, such new process listens to the bus and, when an incomplete event packet is received, it enriches it with the missing information and re-publishes it to the bus.

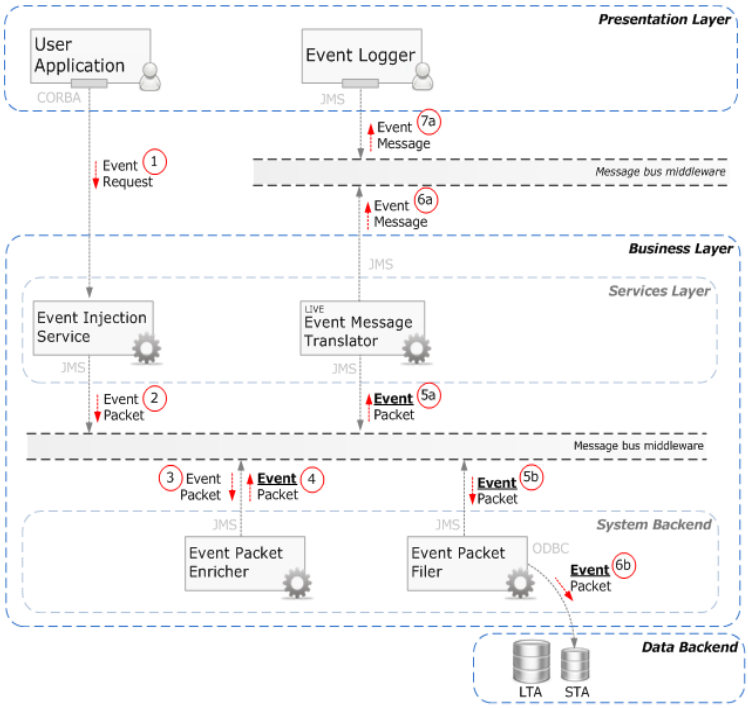

Figure 4 Event generation and LIVE distribution after the reengineering

5. Every application that has registered its interest on event packets is notified about the new published one. This relieves PDSEV from the role of distributing messages; such role is now played by the message broker itself. Now PDSEV just listens to event packets (5b) and files them in the packet storage database. At the same time, any other application listening to such packets also receives them; this is the case for a new process called *Event Message Translator* (5a).

6. For each LIVE event packet published on the bus, the Event Message Translator digests it, produces a high level event message and publishes it to the bus. Any client application is then notified about such message, e.g. the Event Logger. Previously, client applications had to "subscribe" to LIVE event messages in the Event Provision Service; now the subscription to LIVE event messages is handled at bus-level and the logic for handling LIVE event packets and providing digested event messages to subscribers has been factored out of the Event Provision Service into a new process whose sole role is to translate any event packet to an event message and publish it.

Regarding event retrieval, the reengineering comprised the following modifications:

1. The previous CORBA communication from client applications to the Event Provision Service has been replaced with a request message that is published in the event retrieval requests queue.

2. Such message is then read by the Event Provision service, which fulfils the request by querying the Event Packet Retriever (HFAEV).

3. 4. 5. The retrieval of packets is kept as before, that is using raw TCP/IP and ODBC to perform the requests.

6. The digested event message is then published by the Event Provision Service to the bus, in certain queue that was specified by the requestor in the original request message.

7. Finally, the user application reads the replied messages that fulfil its request.

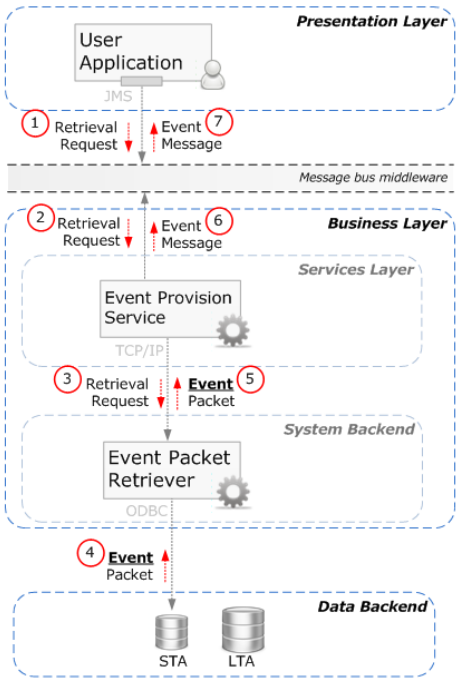

Figure 5 Retrieval of events after the reengineering

As seen in the previous paragraphs, the reengineering for introducing a message-oriented middleware as replacement for raw TCP/IP communications and CORBA, led to a rationalization of the responsibilities of each process:

- The responsibilities of LIVE packets distribution and the subscriptions to them are now assumed by the middleware itself.
- The translation of LIVE messages has been factored out of the Event Provision Service into a new process that only has such role.
- The enrichment of event packets has been extracted from PDSEV and encapsulated in a new process that only performs such operation.

The modifications depicted over section II are still in progress. Currently, the activity completion status is as follows:

- The raw TCP/IP-based LIVE packet distribution system has been completely reengineered to use the message-oriented middleware. This comprises not only Event Packets, but also Telemetry and Telecommand ones. We have successfully tested the reengineering in a fully functional *hifly* deployment that monitors and controls a simulator of a commercial spacecraft bus whose TM downlink is 10 Kbps.
- The reengineering of CORBA-based communications for distributing LIVE event messages and retrieving historical events, as of today, is functional, but not yet complete. Some features such as filtering LIVE events are not implemented. Aside, the middleware layer has only been tested under out-of-the-box configurations, needing still to undergo a tuning process driven by empirical experiences in scenarios that are comparable to real-world deployments.

Once the activity is finished, we will conduct performance testing to evaluate the impact of the reengineering regarding throughput, bandwidth consumption, latency and responsiveness.

III. Learned lessons

In this section we summarize the lessons we learned during the execution of the reengineering described in section II regarding different aspects:

A. Using a message-oriented middleware instead of raw TCP/IP and CORBA

The hope when replacing ad-hoc code (either meant for communications or whatever other purpose) with a third party product, is that you will be able to remove the code that the library renders no longer necessary. This was especially true in our case, as *hifly* has its ad-hoc packet distribution mechanism (i.e. PDS and CPD processes) whose functionality overlapped completely with that of a message-oriented middleware. This way, once the reengineering of the complete system is finished, we will be able to **delete thousands of no longer used lines of code** and **remove several processes** from our deployments. Removal of the code improves the maintainability of the system, not only due to the decrease in the size of the code base, but also because the purpose of most of those pieces of code was to handle very low level functionality that is now managed by the MoM.

However, even more important is the *mindset change* imposed by the use of a **middleware that exposes the asynchrony of the network**. This, unlike raw TCP/IP and CORBA, asynchronous messaging forces the developer to handle the problems derived from the remote communications, preventing him or her from ignoring them. This does not mean that the business logic has to deal with such aspects; proper isolation of the transport layer is always essential to a decoupled design. In case of *hifly*, the use of TCP/IP and CORBA was already encapsulated in components that did not leak transport knowledge to the domain logic layer, easing the straightforward replacement of the communications API used.

It is also remarkable how the RPC paradigm (i.e. CORBA) *influences* our way of facing software engineering problems: when there is already a CORBA interface in place, it seems appropriate to *enrich* it with new methods in case the client application that already interacts with it needs more functionality. Over time this leads to bloated CORBA interfaces that offer all kinds of unrelated functionality. In contrast to this, a message style communication does not make it convenient to put more functionality into an already existing server process. In fact it is annoying. It is frequent to find the most appropriate process and implement the new functionality there, or create a new process when no suitable one already exists. This is because client applications do not speak directly to processes but just send messages to the bus, thus making it **irrelevant *who* fulfils the client's requests**. The result is usually the **better distribution of responsibilities among processes**. This also **improves the scalability** of the system, as it eases the possibility of simply *plug ging* replicas of the functional elements into the bus.

B. Migrating pub-sub TCP/IP and CORBA communication patterns

In the MCS, telemetry data, events and dynamic configuration changes[3] that control certain processes have to be distributed in real-time. In SCOS-2000 and hifly, either raw TCP/IP or CORBA are used, depending on the type of information distributed.

The reengineering of a publish/subscribe style of communication into a bus-based solution seems natural and straightforward. However, in most cases such subscriptions have an associated *filtering logic* that is evaluated at the server side in order to save bandwidth.

In a MoM, the same functionality can be implemented by means of *selectors*: when a message consumer subscribes to the messages from certain topic, it can specify the criteria for it to accept messages. Such filter is specified normally as a Boolean expression based on the message properties, that is, a set of key-value pairs stored in the message header.

[3] In SCOS-2000 and *hifly*, variables that control how certain processing is done are implemented by means of MISC dynamic variables mechanism.

This way, in order to keep the same functionality regarding LIVE messages filtering, every piece of information that is suitable to be filtered by, should be placed in the message header so message consumers can express their selectors based on them.

C. Migrating request-response TCP/IP and CORBA communication patterns

Messaging-based solutions are not only prepared for publish/subscribe, but are also suitable to situations in which the client application needs a reply. This communication scheme is widely used and is usually referred to as "request-reply" interaction pattern.

It is normally implemented by having a "requests queue" in which client applications place their requests; each request message contains the identifier of another queue in which the server will place the reply once the request has been fulfilled; also, normally the client sets a "request ID" that is used to mark the reply, so the original request of a reply can be identified.

D. Checking the services connection status

In raw TCP/IP and CORBA communications, it is straightforward for the client to monitor the connection status because of the coupling between client and server processes. When an explicit error is detected by the operating system during an invocation or in case the reply takes too long (i.e. timeout), then the service is flagged as unavailable and the reconnection mechanism takes place.

In message-oriented middleware, as the connection is established with the broker, it is not possible to do the same type of monitoring. At first, it is tempting to implement a *ping* mechanism, in which the client application frequently asks the server if it is still alive and working. Alternatively, the server could state its availability in a publish and subscribe manner (pub-sub), so anybody can subscribe to those notifications and tell whether the last notification was published too long ago to consider the server dead.

The two previous solutions fail to harness the real power of message-oriented middleware: there is no need for client applications to know whether the server that provides the functionality they need is up or not; instead, it is important to know if there is *some process* that can provide such functionality, regardless of which process it is.

This way, a new concept arises: *capabilities*. In the English language, "capability" is the ability to perform actions. Thus, from a Service-Oriented Architecture (SOA) point of view, the MCS provides a set of capabilities, no matter which is the concrete software entity or entities that provides them.

To a client application, it is important to provide visual feedback to the user about whether the capabilities needed to fulfil the application's purpose are available or not. When a server process would publish a message for each capability it provides. Any client application then knows that there is some process that will reply in case they request such functionality. The *grain* at which capabilities are specified is very important, and should make any assumptions on which servers are going to implement each capability, but only care about the conceptual boundaries of each piece of functionality.

This mechanism maintains the decoupling between client and server processes, making it possible to replicate the server process in order to achieve high throughput or fault tolerance. While at least one of them is up, client applications will know that there is a process capable of fulfilling their needs. Also, it should be possible for a server providing several capabilities to be split into several processes, each one providing one capability, thus favouring the single responsibility principle and improving the scalability of the system.

IV. Opportunities for evolution

Having a messaging infrastructure as backbone of the MCS opens the door to enhancements to the system that were not feasible before. Most of them derive from the decoupling between client and server and the delocalization of the service providers.

In the following paragraphs we superficially describe some of the opportunities for evolving the system that we have identified once the complete reengineering is finished:

A. Scalability and high availability through replication

Most of the processing done in the MCS, either CPU-bounded or I/O bounded, is suitable to parallelization. In such cases, doing so would ideally be a matter of just replicating the processing element *n* times, until the desired throughput is achieved.

However, as TCP/IP and CORBA present strong coupling between the processes that take part in the communication, it was not easy to achieve transparent load balancing without creating ad-hoc mechanisms. Once the whole message-oriented middleware reengineering is in place, it would be possible to design services that can scale following the depicted replication strategy.

Having several instances of the same process not only leads to higher throughput, but also to tolerance to failures: if an instance of the processing elements crashes, any of the other redundant instances can keep providing service.

B. Service Hot-plug

Patching an operational MCS can be concerning for the operator, as due to the dependencies among processes, restarting the ones that are to be patched may imply having to restart the whole system.

The decoupling between interacting processes offered by message-oriented middleware makes it easier to keep the whole system running while restarting only the strictly needed components. It would even be possible to have both the old and the new versions of the processes up at the same time in a shadowing migration approach.

C. Improved flexibility in the deployment schemes

Most, if not all the spacecraft operators have multiple geographically distant sites for monitoring and controlling their fleet. Those sites are usually connected and are subject to some sort of data forwarding and alignment so both have the most up-to-date operational data, thus enabling switching the control to the other site if needed.

Those data forwarding pieces of logic may be moved to routing rules into the messaging system. Dropping the custom code that handled such functionality will open the door to having more flexible deployments in which the redundancy policies suit each customer's needs.

D. Interoperability

While many ESBs publicize their compatibility with CORBA, their support is usually very limited and usually requires custom code. Also, when the CORBA communication implies call-back interfaces, sometimes it is simply not possible to achieve transparent plugging into the ESB.

On the other hand, mainstream asynchronous message APIs, such as JMS, are supported by most ESB vendors, making it easy to transparently integrate the MCS with other ground segment subsystem.

V. Conclusions

Message-oriented Middleware has proven to be a suitable replacement for raw TCP/IP and CORBA communications in the MCS. Insightful lessons have been learned during the development of the activity, confirming that the reengineering will lead to a more scalable architecture that would open the door to a new set of features that, relying on the middleware's features, will enable the MCS to perform better and in a more reliable, robust and flexible way.

Appendix A
Acronym List

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The Self
CPD	Client Packet Distributer
DBMGR	Database Manager
DDS	Data Distribution Service
ESA	European Space Agency
ESB	Enterprise Service Bus
HFA	Historic Files Archive
JMS	Java Message Service
LOC	Lines Of Code
LTA	Long-Term Archive
MCS	Mission Control System
MoM	Message-oriented Middleware
NASA	National Aeronautics and Space Administration
ODBC	Open Database Connectivity
OMG	Object Management Group
QoS	Quality of Service
S2K	SCOS-2000
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
STA	Short-Term Archive
TC	Telecommand
TM	Telemetry
WAN	Wide-Area Network
XDR	External Data Representation

Appendix B

Glossary

CDR	Data representation format used in CORBA to marshal/unmarshall information.
CORBA	RPC-style communication standard based on an interface definition language that can be compiled to different programming languages, disguising the network-based communications as local invocations.
CORBA servant	Server implementation of a CORBA interface.
ESA	Intergovernmental organization dedicated to the exploration of space.
GMSEC	NASA's bus-based ground segment component integration infrastructure software platform.
hifly®	GMV's mission control system, based on SCOS-2000.
JMS	Java API to interact with generic message-oriented middleware.
Mission Control System	Software system used to operate a spacecraft from the ground.
NASA	Agency responsible for the USA's civilian space program and for aeronautics and aerospace research.
ODBC	Standard C interface to access relational databases in a homogeneous way.
Pub-sub	Publish-subscribe messaging paradigm.
Queue (MoM)	Channel in a message-oriented middleware in which each message sent by a message producer is assured to be received by one and only one message consumer; the message will be kept in the queue until a consumer accepts it.
SCOS-2000	ESA's mission control system.
TCP/IP	Communication protocols that allow processes in different machines communicate over a network in a stream-oriented manner.
Topic (MoM)	Channel in a message-oriented middleware intended to behave in a publish/subscribe manner: message producers simply "publish" whatever messages on the bus, while message consumers subscribe to be notified of such messages. Not either producers or consumers are aware of each other.
X.509	Standard for public key infrastructure that, among other things, specifies digital certificates.
XDR	Serialization format developed by Sun Microsystems in the mid-80s, used in NFS as representation layer protocol.

Bibliography

The Evolution of ESA's Spacecraft Control Systems
http://www.esa.int/esapub/bulletin/bullet89/baldi89.htm

ESA's introduction to SCOS-2000
http://www.egos.esa.int/portal/egos-web/products/MCS/SCOS2000/

A Note on Distributed Computing
Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall
Technical Report (Sun Microsystems), 1994

MOM vs. RPC: Communication Models for Distributed Applications
Daniel A. Menascé
IEEE Internet Computing, Volume 9 Issue 2, March 2005

RPC under fire
Steve Vinoski
IEEE Internet Computing, Volume 9 Issue 5, September 2005

Convenience over correctness
Steve Vinoski
IEEE Internet Computing, Volume 12 Issue 4, July 2008

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe, Bobby Woolf
Addison-Wesley Professional, October 2003

Message-oriented Architecture
Jesús Santana
Presentation at ESAW 2009