Project 2





**FDIR**

*Spacecraft fault protection system*

**Euro Team**

|  |  |
| --- | --- |
| **Mikko AHVENNIEMI** | 20096680 |
| **Pierre ALAUZET** | 20096699 |
| **Julien COLIN** | 20096706 |
| **Benoît STARCK** | 20096705 |

**Table of contents**

December 12th, 2009

**CS554 - Design for Software & Systems**

[Illustration table 3](#_Toc245713292)

[Introduction 4](#_Toc245713293)

[1. ACME, a powerful ADL 5](#_Toc245713294)

[2. Utility tree 6](#_Toc245713295)

[3. Architecture 7](#_Toc245713296)

[4. Architectural approach analysis 8](#_Toc245713297)

[5. Discussions & alternatives 9](#_Toc245713298)

[Conclusion 10](#_Toc245713299)

[References 11](#_Toc245713300)

[Annexes 12](#_Toc245713301)

[1. 12](#_Toc245713302)

**Illustration table**

Aucune entrée de table d'illustration n'a été trouvée.

**Introduction**

In the purpose of applying and studying real case project for the *Design for Softwares and Systems* course, our team is required to understand and design a fault protection system for a spacecraft as described in the article by Steve Easterbrook et al. [Eas98]. The **first part** of this global project was to understand the **problems** we have to respond to, to specify the **needs of our client** and to start thinking about a **user system interface**.

In this second report, the purpose will be to use the clear problem’s understanding we obtained thanks to the first part to specify a **global architecture** for the FDIR system. The first step of this job is to remind FDIR’s functional requirements and **quality attributes**, as identified and improvised during our analysis phase, and improved by the use of the **ATAM process** for architectural evaluation, that we had the chance to learnt during this course.

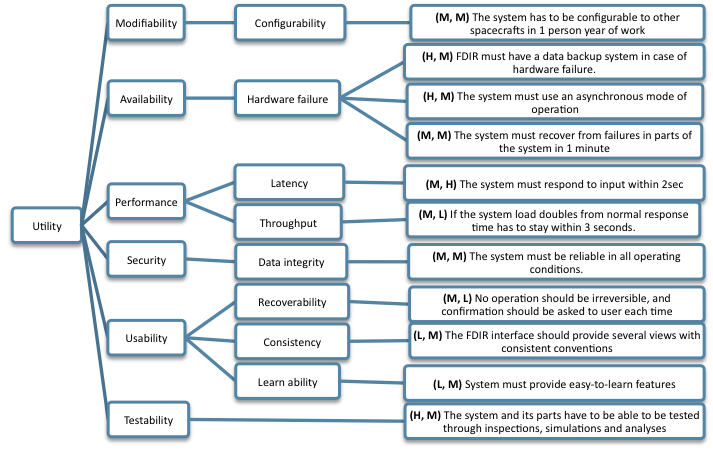
For constructing this architecture, we will use the **ACME architectural description language** that we presented during our OP5. This language allows us to describe a complete architecture using several architectural **styles** to draw it in a fashion manner using the ACME studio software.

First of all we are going to present several outputs of the **ATAM process** such as the **utility tree** presenting the system’s quality attributes, and several **scenarios** describing how our architectural decisions should meet the non functional requirements of our system. Then we will list the **systems** we want to describe as components of our system, we will discuss our choices on the architectural styles we want to use, while proposing several approaches to describe the overall system. We will then present our final architectural choice, how this choice match to our quality attributes. The final part will consists of a discussion about the **risks**, the non-risks, sensitivity points, and tradeoff point related to our finalized architecture. We will also provide some **alternatives** and **criticizes** about our work.

The material presented on this report is a synthesis of our previous work for OP4 and OP6, including some refactoring of our architectural diagrams and modifications based on the feedbacks we obtained from professor and TA.

1. **System description & business case**

1. **Utility tree**



*Fig1. Utility tree*

- Use case scenarios

- No operation should be irreversible, and confirmation should be asked to user each time

- user action should be done at any moment

- The FDIR interface should provide several views with consistent conventions

- Growth scenarios

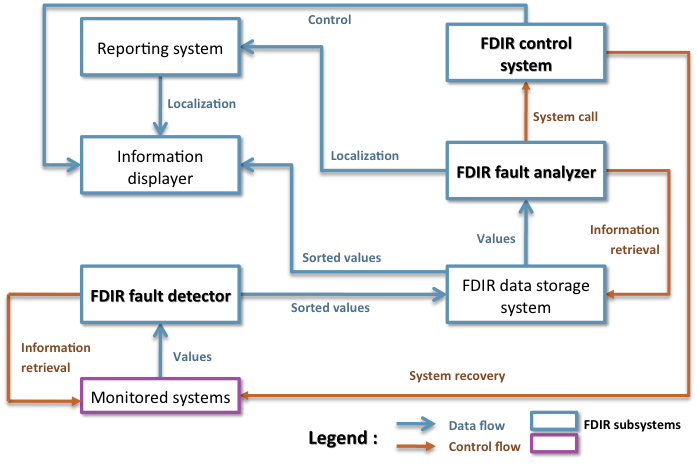
- A new sub-system must be able to be installed in to the FDIR in 1 person day of work

-Exploratory scenarios

- If the system load doubles from normal response time has to stay within 3 seconds.

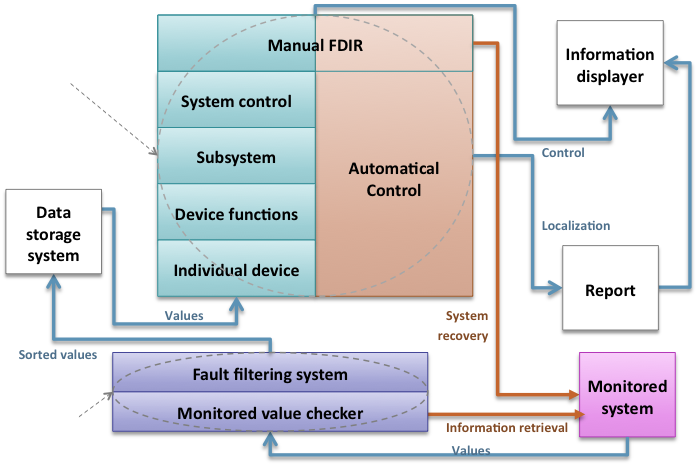
- if a FDIR sub-system is crashing, FDIR should still work

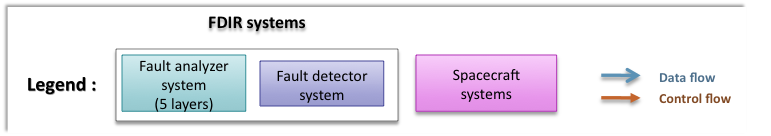
- The system has to be configurable to other spacecrafts in 1 person year of work



*Fig2. Data flow and Control flow architecture proposal*

This architectural proposal was our first thought for FDIR architecture. As FDIR system is composed of a lot of systems interacting with each other by sharing data and/or instructions, the idea appeared to describe it using a mix between data flow and control flow architectural styles. The monitored systems are including every spacecraft’s systems whose values are treated by the FDIR. The fault detector is collecting all the values, treat them and sort them before transferring it to the storage system. Stored values are displayed and checked by the fault analyzer to determine fault localization. Localization is sent to fault report maker system that display issues on the screen. The fault analyzer call the control system (can be either automatically or manual control). The control system is able to perform recovering actions on monitored systems.





*Fig3. Data flow and Control flow architecture proposal*

This architecture is our second proposal. In this description we also use data flow and control flow interactions, but we decided to describe several systems with a layered architecture. The previously saw FDIR fault detector is here describe as a two-layered system; the monitored value checker is collecting the data and is accessible to the fault filtering system that filters and sort the data. We also merged the fault analyzer and the control system in the same layered block. 4 layers are taking care of fault analyzing, when one layer can’t resolve it the upper layer access to it and take care of the fault Each of these 4 layers are interacting with the automatical control part. When none of these layers can resolve the problem, it goes to the manual layer that wraps the control part in order to provide manual control.

1. **Architecture**

User has to subscribe to an event if they are interested in receiving notifications. These notifications are generated by a publisher. User (**subscriber**) is subscribing while **publisher** is notifying him about several news he asked for. When an event is created, it is generating some notifications from the publisher by a *publish()*function/method. An event service between publishers & subscriber allows the system to work under several abilities:

1. **Space decoupling**: publisher & subscriber work indirectly between them without knowing each other. In fact, the event service links the publisher to the subscriber in order to make it work.
2. **Time decoupling**: publisher can send notification while subscriber is disconnected. Vice & versa, user can see notification while publisher is not running.
3. **Synchronization decoupling**: concurrent activities can be performed by publishers & subscribers. They are asynchronously notified of an event.

**Topic-based P/S**

The topic-based P/S is strongly similar to “groups” notion. Bundle communication peers, with methods to characterize & classify event content (divided by keys in a string shape).The first systems using P/S were based on group of communication.

Difference between groups & topics is that groups are used for maintaining strong consistency between the replicas of a critical component in a local area network (LAN), whereas topics are used to model large-scale distributed interactions. Individual topics are linked to distinct communication channels. Hierarchies are orchestrating topics. Topics regroup event in content and structure. It is static & primitive but efficiently implemented.

**Content-based P/S**

A content-based P/S corresponds to a subscription scheme based on the actual content of the considered events. The user specifies what he wants using filters. Participants can subscribe to logical combinations of elementary events and are only notified upon occurrence of the composite events. It is highly expressive but sophisticated protocols to put in place.

Event driven architecture

Devices subscribe to sub-systems which in turn listen to events broadcasted by the devices.

Such events can be for example “announce value” event

Choice based on identified quality attributes

•Enables asynchronous processing

•High potential for resilience in case of failure

•Load can be balanced efficiently between systems

Architectural approach follows logically from system architecture

This part is talking about the different choices or types we have to implement a P/S system.

**Events**

1. **Message**: Made of a header (message specific-information) and payloads data (user specific-information). It is a king of records or data with several fields.

2. **Invocation**: it is calling methods of the subscriber in order to make some actions.

**Media**

Media is a transmission of data between producer (publisher) & consumer (subscriber). It is tested through three different aspects: architecture, reliability or persistence.

1. **Architecture:**

- Centralized architecture: Messages are sent to a single one entity which stores everything (reliability, consistency & transactional support). Message goes to the producer to the consumer passing by the entity. Centralized architecture is following this scheme:

producer🡪 entity 🡪 consumer

- Distributed architecture: at the contrary, distributed architecture is asynchronous & anonymous. In this case, messages are going faster and the delivery is even more efficient. Entity is not present anymore. That means that there is directed link and direct relations between the producer & consumer.

Producer 🡨🡪 consumer

2. **Dissemination:**

Message queuing systems are using point-to-point communication between P/S and centralized broker. Performance is highly affected by the cost of event filtering on each of the servers. Hopefully, new algorithms allow filtering data in P/S systems.

**Quality of Service**

1. **Persistence:**

Message sent without generating replies. Transmission message is not controlled. Durability of information is really important, even more than guarantee in reliability. Has to check that message would not be lost.

Persistence is present in centralized architecture. Indeed, entity is checking and storing message till it is delivered. Nevertheless, it is not present in distributed architecture.

2. **Priorities:**

Priority is working with persistence. When messages have to be sent, priority can check whether there are some “real-time” events which need to be send before others. This affect messages in transit. It is actually a best effort QoS.

Priority aspect is present in both centralized and distributed architectures.

3. **Transactions:**

It is used to join a sequence of message into one block in order to send it in one time. Transaction is really useful when we encounter a failure. In this, none of the sequence is sent.

4. **Reliability:**

Reliability allows making sure that messages or sequences are delivered to the entities. It is really close to persistence aspect.

1. **Architectural approach analysis**

Advantages

Loosely-coupled: Publishers are loosely coupled to subscribers, and needn't even know of their existence. With the topic being the focus, publishers and subscribers are allowed to remain ignorant of system topology. Each can continue to operate normally regardless of the other. In the traditional tightly-coupled client-server paradigm, the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. Many pub/sub systems decouple not only the locations of the publishers and subscribers, but also decouple them temporally. A common strategy used by middleware analysts with such pub/sub systems is to take down a publisher to allow the subscriber to work through the backlog (a form of bandwidth throttling).

Scalable: For relatively small installations, pub/sub provides the opportunity for better scalability than traditional client-server, through parallel operation, message caching, tree-based or network-based routing, etc. However, as systems scale up to become datacenters with thousands of servers sharing the pub/sub infrastructure, this benefit is often lost; in fact, scalability for pub/sub products under high load in large deployments is very much a research challenge.

The most serious problems with pub/sub systems are a side-effect of their main advantage: the decoupling of publisher from subscriber. The problem is that it can be hard to specify stronger properties that the application might need on an end-to-end basis:

\* As a first example, many pub/sub systems will try to deliver messages for a little while, but then give up. If an application actually needs a stronger guarantee (such as: messages will always be delivered or, if delivery cannot be confirmed, the publisher will be informed), the pub/sub system probably won't have a way to provide that property.

\* Another example arises when a publisher "assumes" that a subscriber is listening. Suppose that we use a pub/sub system to log problems in a factory: any application that senses an error publishes an appropriate message, and the messages are displayed on a console by the logger daemon, which subscribes to the errors "topic". If the logger happens to crash, publishers won't have any way to see this, and all the error messages will vanish.

As noted above, while pub/sub scales very well with small installations, a major difficulty is that the technology often scales poorly in larger ones. These manifest themselves as instabilities in throughput (load surges followed by long silence periods), slowdowns as more and more applications use the system (even if they are communicating on disjoint topics), and so-called IP broadcast storms, which can shut down a local area network by saturating it with overhead messages that choke out all normal traffic, even traffic unrelated to pub/sub.

For pub/sub systems that use brokers (servers), the agreement for a broker to send messages to a subscriber is in-band, and can be subject to security problems. Brokers might be fooled into sending notifications to the wrong client, amplifying denial of service requests against the client. Brokers themselves could be overloaded as they allocate resources to track created subscriptions.

Even with systems that do not rely on brokers, a subscriber might be able to receive data that it is not authorized to receive. An unauthorized publisher may be able to introduce incorrect or damaging messages into the pub/sub system. This is especially true with systems that broadcast or multicast their messages. Encryption (e.g. Transport Layer Security (SSL/TLS)) can be the only strong defense against unauthorized access.

1. **Discussions & alternatives**

**Conclusion**

The role of publish/subscribe systems is to permit the exchangeof events between producers andconsumers in an asynchronous manner.Thanks to the three dimensions decoupling (time, space, synchronization), participants (producers & consumers) can operate independently. None of the P/S is perfect. Scalability remains a big issue for P/S. Studying these solutions let us think that it could be better to try to merge and take the best parts of every P/S principle.

**References**

**Web Sites**

**Books**

1. [BCK98] L**. Bass, P. Clements, R. Kazman**, *Software Architecture in Practice (2nd ed.)*, Addison-Wesley, 2003.
2. [Eas98] **Steve Easterbrook, and et al**., “Experiences Using Lightweight Formal Methods for Requirements Modeling,” *IEEE Transactions on Software Engineering, Vol. 24, No. 1*, January 1998.
3. [KKC00] **R. Kazman, M. Klein, P. Clements**, *ATAM: A Method for Architecture Evaluation*, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, 2000.
4. [SG96] **M. Shaw and D. Garlan**, *Software Architectures – Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
5. **P. T. Eugster, P. A. Felber, R. Guerraoui and A. M. Kermarrec**, *The Many Faces of Publish/Subscribe*, in ACM Computing Surveys, vol. 35, no. 2, June 2003, pp. 114-131

**Annexes**