PROJECT 2

FDIR

Spacecraft fault protection system

Euro Team

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

CS554 - Design for Software & Systems

December 12th, 2009

Table of contents

Illustration table

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 & ATAM process

a. System description & business context

The purpose of this part is to recall the business cases and requirement of the Fault Detection, Isolation and Recovery system (FDIR). We extracted, as much as we could, the requirements from the [Eas98] paper.

This system is specially designed for a spacecraft and provides specific functions needed and requested by the client, which make this product unique. Customer presents two main needs about this system: the guarantee of the completion of any time critical activities, and the control of the spacecraft with **safety, observability and commandability**. Indeed each spacecraft device has a predefined set of operating parameters that have a normal operating range. Values beyond this range are called "out-of-tolerance". Besides that, an out of-tolerance condition may come from any possible causes. That is why information from multiple sources must be combined in order to locate the fault.

The FDIR system is unique and specially designed to generate appropriate responses when out-of-tolerance conditions are detected in hardware and software components. It means that it will be able to recover the data, locate the fault with precision and to fix them as well by a restart of a system for example. Moreover this system is designed to answer all actions launched by the crew or the flight controller on Earth. Both actors are able to interact together and launch different actions at the same time. In this purpose, the requirements of the system are the followings:

1. **Guarantee the completion of any time critical activities of the spaceship**. Even if a device has a problem, the system must be able to analyze, locate the fault, and restart the action again, even if it has to fix by anyway the problem before (recover the data, or restart the device for example).
2. **The system provides a manual control for the user**. The crew or the flight controller must be able at any time to restart, shutdown, or switch to a spare system.
3. To control the spacecraft and the FDIR system, the crew should know information about each device. That is why **the system will display information continuously to the both actors**.
4. **The system will be able to collect data to data storage**. All data during the journey will be store in a safe place on the spacecraft. After that, the crew must be able during the complete mission to **retrieve old information about a device**, to compare with actual data for example.
5. **System must display the failure localization**. In this goal, system has specific tools to detect and locate failure in a part of the ship, and display information with precision to the crew
6. FDIR Storage System contains the collected values or data from devices as said before. FDIR checks the inputs from the storage system, and analyses these inputs to determine if irresolvable conditions has been reached. **Information about irresolvable conditions is written into a report sent as a notification to the crew.**
7. **System presents an automatic recovery to failure.** If a system is failing, FDIR is required to act on its own to recover the crash. FDIR provides responses automatically for a lot of casual issues, by shutting down or restarting the faulty part. FDIR is also able to provide responses under specific or more critical context (hazardous conditions or unresolved problems).
8. **Keep the control of the spacecraft with safety, observability&commandability.**

b. ATAM process

ATAM offers a method for architecture tradeoff analysis. This method can be proceeding early in the software development life cycle to assess the consequences of architectural decisions in light of quality attributes. Here is the diagram presenting the different steps to respect for using ATAM method:

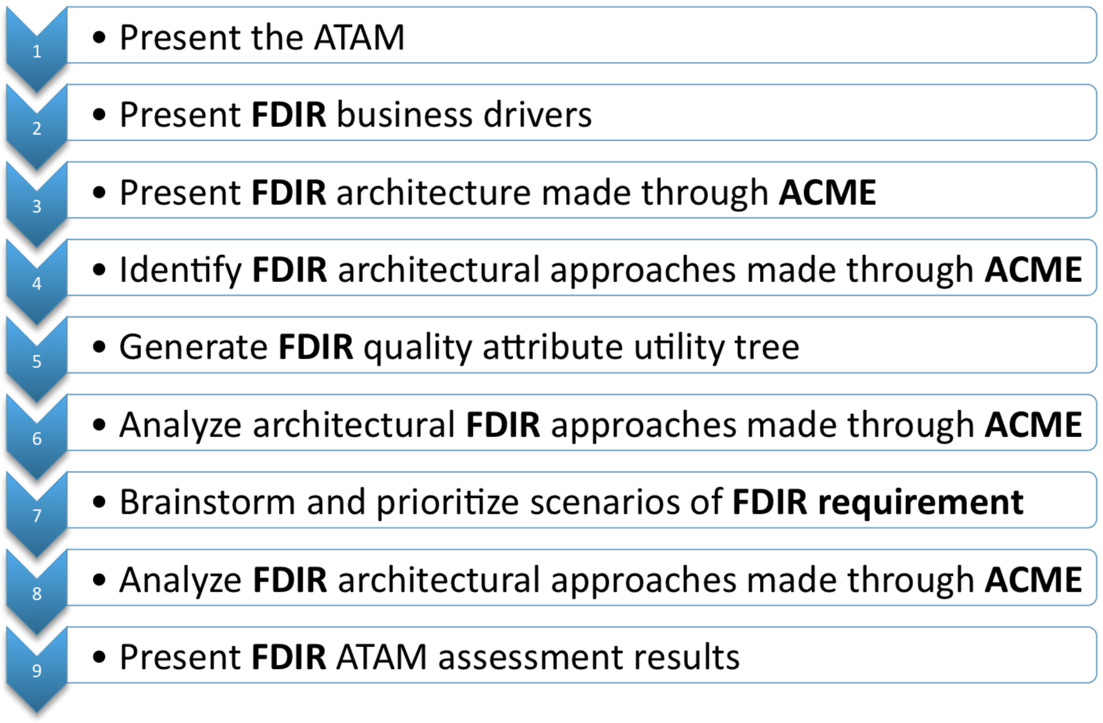

1
• Present the ATAM
2
• Present **FDIR** business drivers
3
• Present **FDIR** architecture made through **ACME**
4
• Identify **FDIR** architectural approaches made through **ACME**
5
• Generate **FDIR** quality attribute utility tree
6
• Analyze architectural **FDIR** approaches made through **ACME**
7
• Brainstorm and prioritize scenarios of **FDIR requirement**
8
• Analyze **FDIR** architectural approaches made through **ACME**
9
• Present **FDIR** ATAM assessment results

Figure 1: ATAM process

The three major goals of ATAM are the followings:

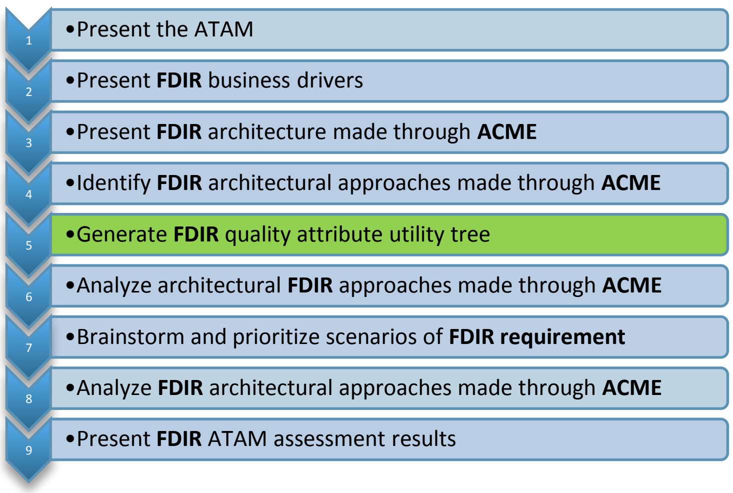
- Elicit and refine a precise statement of the architecture's driving quality attribute requirements
- Elicit and refine a precise statement of the architectural design decisions
- Evaluate the architectural design decisions to determine if they satisfactorily address the quality requirements

The output of the ATAM process is an out-brief presentation and/or a written report that includes the major findings of the evaluation. These are typically:

- The architectural styles identified
- An "utility tree" - a hierarchic model of the driving architectural requirements
- The tradeoff points
- The sensitivity points
- A set of identified risks
- A set of identified non-risks

2. Utility tree & prioritized scenarios

We will base this part on step 5 of ATAM process. We will first generate the FDIR quality attributes and then talk about different kind scenarios classified three different scenarios *families* that ATAM suggests.


1 •Present the ATAM
2 •Present **FDIR** business drivers
3 •Present **FDIR** architecture made through **ACME**
4 •Identify **FDIR** architectural approaches made through **ACME**
5 •Generate **FDIR** quality attribute utility tree
6 •Analyze architectural **FDIR** approaches made through **ACME**
7 •Brainstorm and prioritize scenarios of **FDIR requirement**
8 •Analyze **FDIR** architectural approaches made through **ACME**
9 •Present **FDIR** ATAM assessment results

a. Utility tree

The ATAM process helped us to collect and to formulate the requirements and the usage scenarios. During our previous work for requirement analysis, we established a list of quality attributes the FDIR system should respond to. Those main attributes were the modifiability, availability, performance, security, usability and testability. The following utility tree prioritizes these attributes, from the most important on the upper nodes.

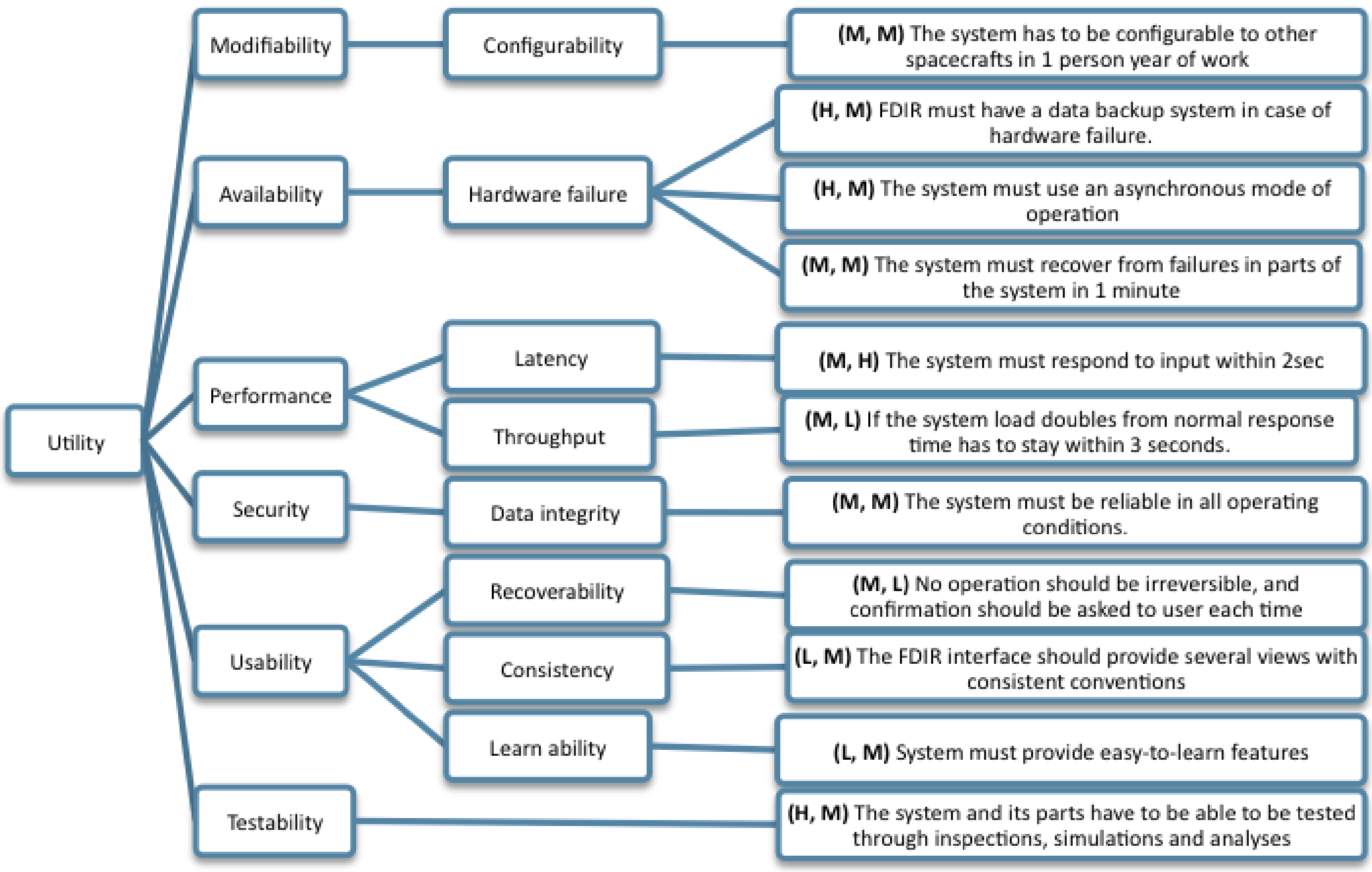

Utility
Modifiability
Configurability
(M, M) The system has to be configurable to other spacecrafts in 1 person year of work
Availability
Hardware failure
(H, M) FDIR must have a data backup system in case of hardware failure.
(H, M) The system must use an asynchronous mode of operation
(M, M) The system must recover from failures in parts of the system in 1 minute
Performance
Latency
(M, H) The system must respond to input within 2sec
Throughput
(M, L) If the system load doubles from normal response time has to stay within 3 seconds.
Security
Data integrity
(M, M) The system must be reliable in all operating conditions.
Usability
Recoverability
(M, L) No operation should be irreversible, and confirmation should be asked to user each time
Consistency
(L, M) The FDIR interface should provide several views with consistent conventions
Learn ability
(L, M) System must provide easy-to-learn features
Testability
(H, M) The system and its parts have to be able to be tested through inspections, simulations and analyses

Figure 2: FDIR utility tree

This prioritized utility tree is focusing on the 3 mains quality attributes we want to reach:

- **Availability**
- **Reliability**
- **Recoverability**

b. Prioritized scenarios

In addition, all these scenarios can be defined by a stimulus, a response and sometimes an environment. Here are some examples to understand how it works, the others scenarios follow these rules too:

i. Use case scenarios

- No operation should be irreversible, and confirmation should be asked to user each time he does a critical action
- User action should be done at any moment
- The FDIR interface should provide several views with consistent conventions

ii. Growth scenarios

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

iii. 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

From all these scenarios, we prioritized two of them:

- User action should be done at any moment
- If a FDIR sub-system is crashing, FDIR should still work

NB: We will detail these scenarios in the last part to show the risks, non risks and sensitivity points of this architecture.

Legend :	Stimulus	Environment	Response

3. Architectural approach, decisions and propositions

In this third part, we will bring decisions about our architectural design and show different drafts about it. We will base this architecture on the step 3 & 4 of ATAM process.

1. •Present the ATAM
2. •Present **FDIR** business drivers
3. •Present **FDIR** architecture made through **ACME**
4. •Identify **FDIR** architectural approaches made through **ACME**
5. •Generate **FDIR** quality attribute utility tree
6. •Analyze architectural **FDIR** approaches made through **ACME**
7. •Brainstorm and prioritize scenarios of **FDIR requirement**
8. •Analyze **FDIR** architectural approaches made through **ACME**
9. •Present **FDIR** ATAM assessment results

a. Overall architecture

Here is our first architecture:

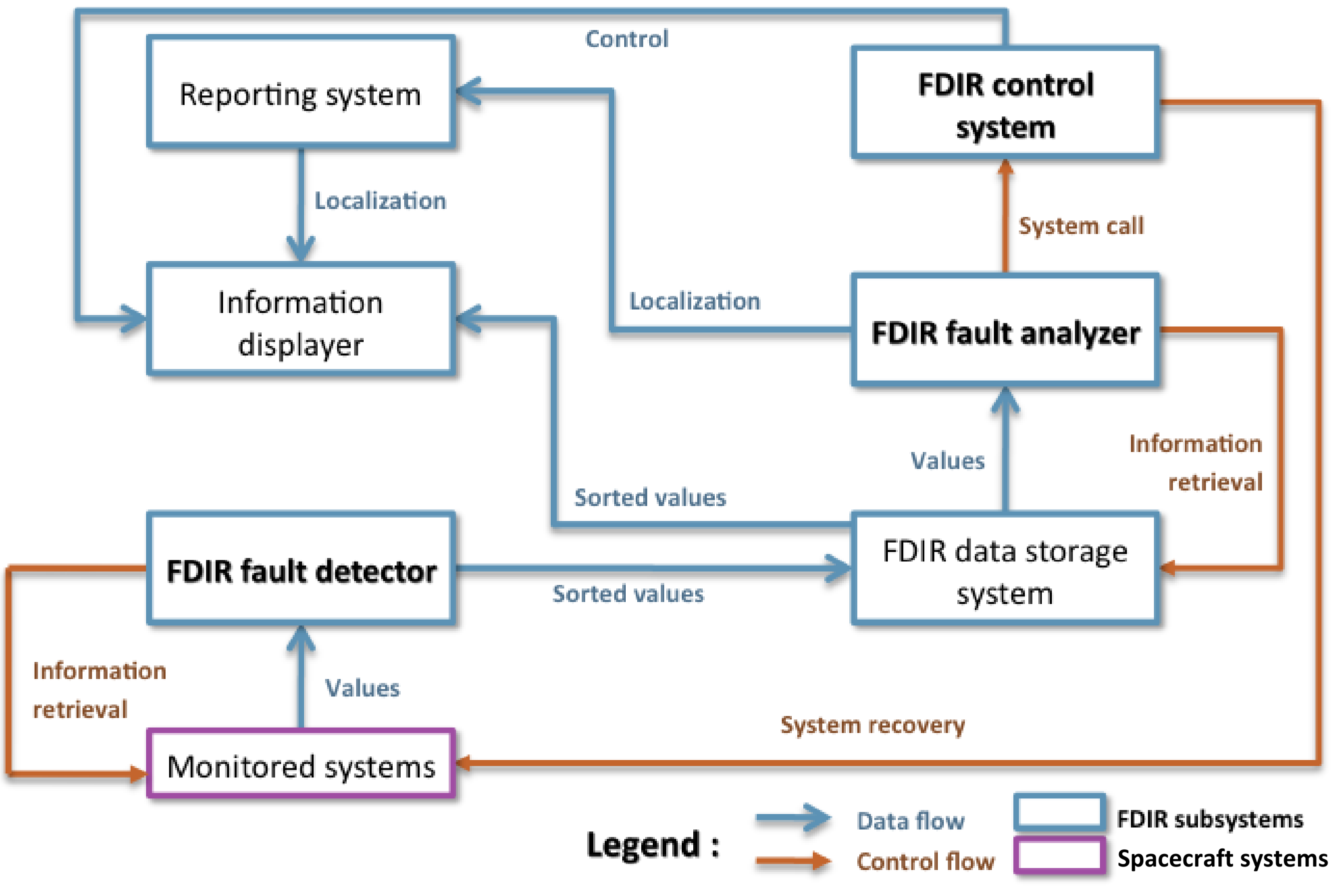


Figure 3: Dataflow and Control flow architecture proposal 1/2

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 different elements are the ones described in the text, and summarize here:

- The monitored systems which include every spacecraft's systems whose values are treated by the FDIR.
- The fault detector which collect all these values, treat them and sort them before transferring it to the storage system.
- Stored values which display and check by the fault analyzer to determine fault localization. This system store each data of the entire system, like reports, all information displayed, all the bugs and information about these bugs, the conclusion of the fault analyzer concerning a problem, the current person who have the control at a moment,…
- 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.

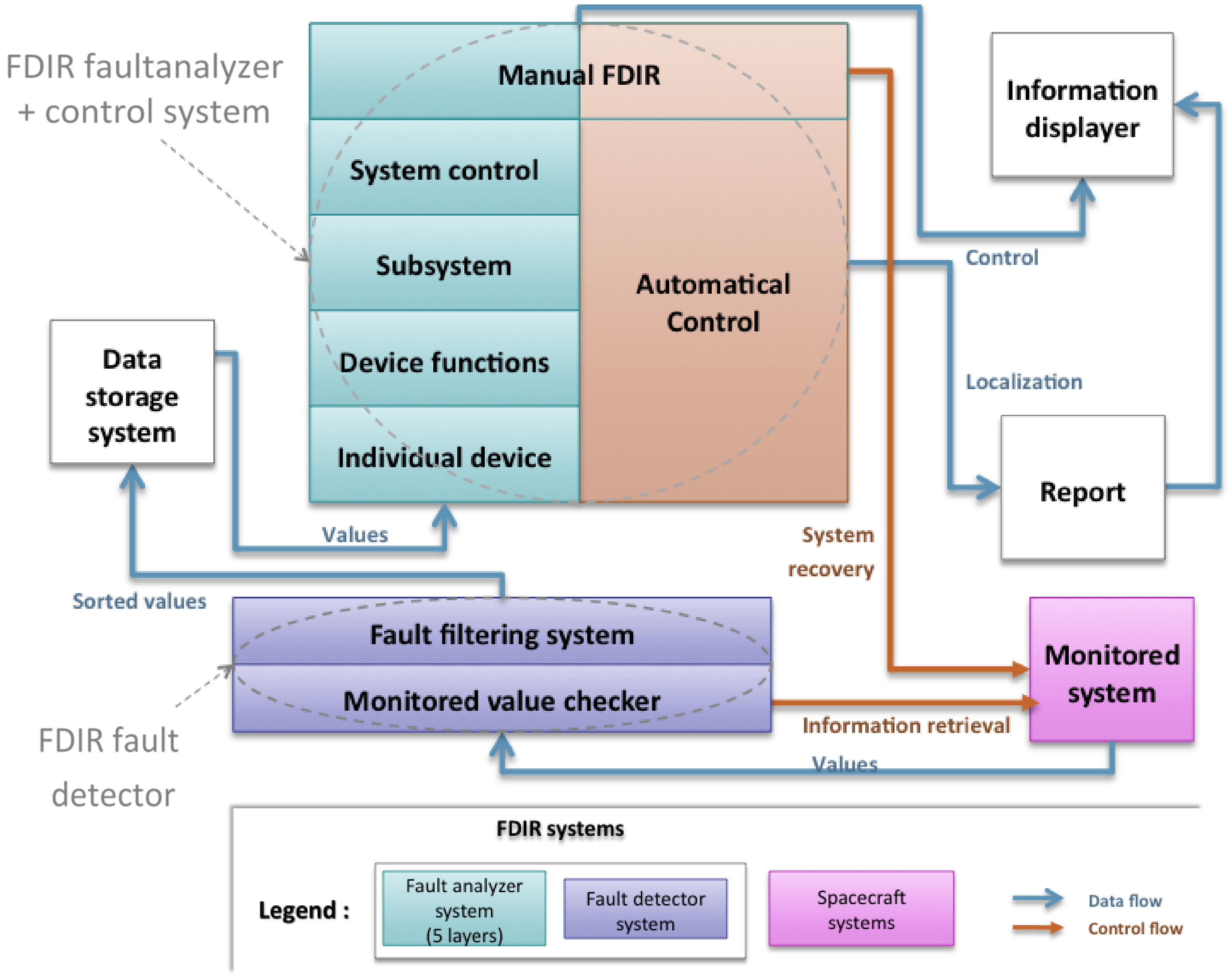


Figure 4: Dataflow and Control flow architecture proposal 2/2

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. Four 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 is interacting with the automatic 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.

We notice that this architecture works with "events". Indeed, it's when a problem appears that this system will respond. When a component needs information, he will launch a procedure to get this information. And the most important, when a component react to information he got earlier, he'll send data after a treatment, but only at this time. If there is no problem for a little while, the most components will not react during this period. So that's why we based our proposal to **an event-based architecture**.

The event-based architectures are mostly based on a famous architectural style: the publish subscribe style.

b. Publish/Subscribe architectural style description

Before building our system with this style, we'll define it briefly to see the likeness with the functioning of our system.

Publish/subscribe (pub/sub) is an asynchronous messaging paradigm where senders, the publishers, post different messages on a server (an event service or event bus); then the subscribers pay or not to see the different posts. In fact, subscribers express several interests about few topics, and only receive messages that are of interest, without knowledge of who, or where, are the publishers. Users have to subscribe to an event if they are interested in receiving notifications of this topic. These notifications are generated by a publisher. User (the subscriber)is subscribing while publishers are 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.

The publish subscribe style can work in two different ways:

- **Topic-based P/S:** the topic-based P/S is strongly similar to "groups" notion. There are 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.

There are two types of architecture that you can use when you decide to use this style. The first one is a centralized architecture, that is to say that 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

The second way is a 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

To conclude this presentation, we'll present now the main qualities of the pub sub style:

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.

c. Choices, justification and application to the FDIR system

i. Architectural type choice

So, in the FDIR system, the different devices subscribe to sub-systems which in turn listen to events broadcasted by the devices. For our case, these events can be for example “announce value” event.

Moreover, the FDIR system looks like to a **topic-based** publish subscribe: indeed, the components are interested in a certain topic. For example, the report component just needs localization and a description of the failure to do a report.

We can also notice that the different component of the FDIR system communicate directly between each others to send or get some information. So it is a **distributed architecture.** We don’t have a central unit which forwards all the messages according the topic. The components ask directly the components that have the requested information.

Besides the publish/subscribe style brings strong advantages:

- Enables asynchronous processing
- High potential for resilience in case of failure
- Load can be balanced efficiently between systems

So it’s corresponding well for the FDIR system and its goals. As you know, we chose to prioritized3 qualities attributes: the availability, the reliability and the recoverability. So, the publish subscribe style answer all these needs. That’s why we choose it!

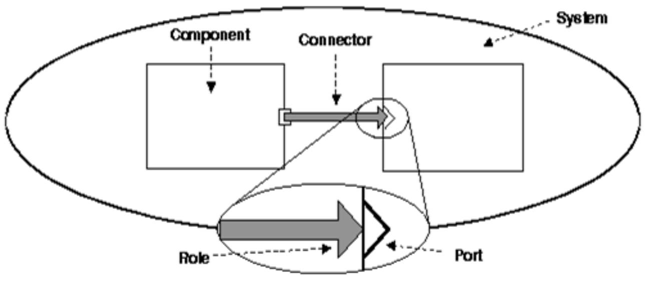
Besides, as we explained, we chose to prioritized two scenarios:

- User action should be done at any moment (Use case scenario)
- If a FDIR sub-system is crashing, FDIR should still work (exploratory scenario)

→So we’ll create our architecture based on these two scenarios.

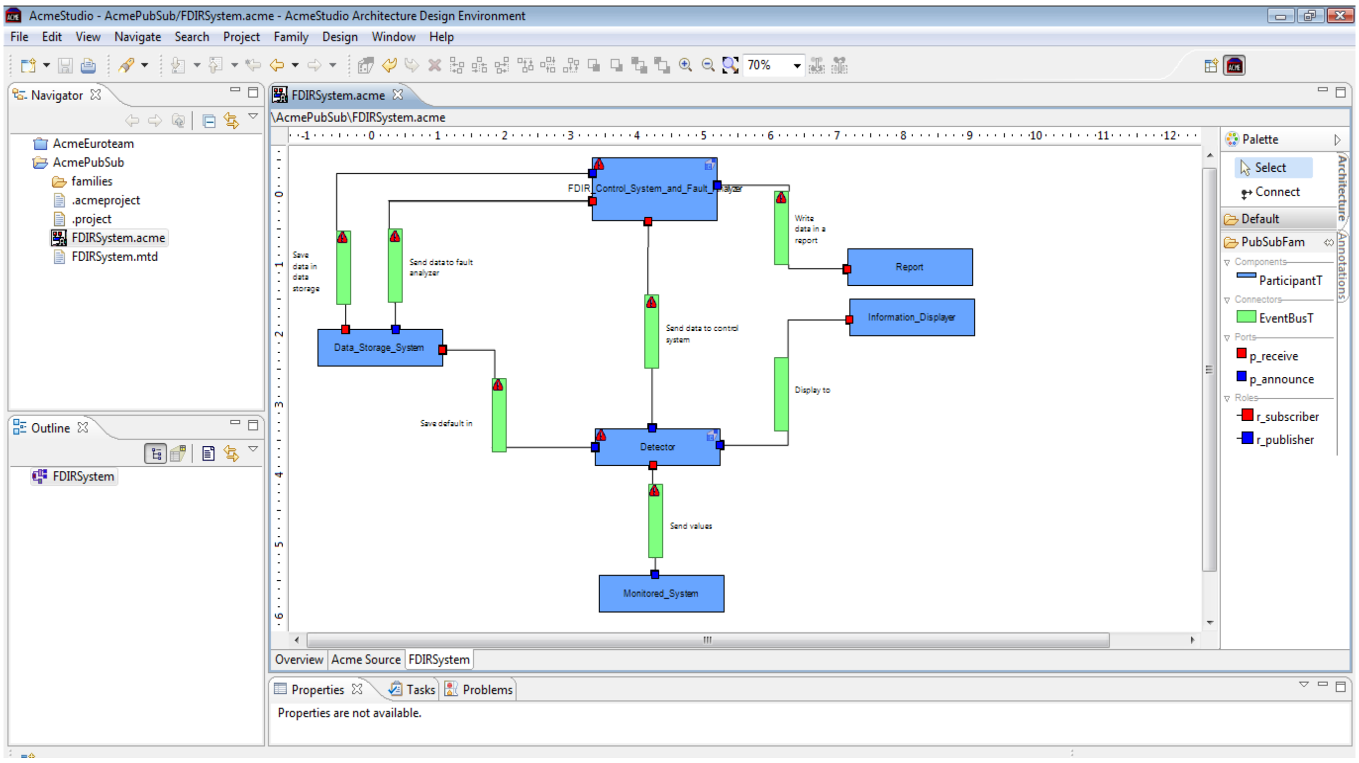
ii. ADL choice: the ACME project

In order to create our architecture, we had to choose an ADL (Architecture Description Language). An ADL is a language for modeling a software system’s conceptual architecture, distinguished from the system’s implementation.ADLs brings the tools for architecture evolution and reusability. There are different ADLs: Wright, Rapide, Unicon, Abacus…



But we decided to choose another ADL: ACME. We did this choice for several reasons. First of all, it is well know and developed in the same university as ATAM method – *Carneggie Melon University.*It provides a generic, extensible infrastructure for describing, representing, generating, and analyzing software architecture language description. Moreover it provides descriptions that are easy to understand for everyone.ACME describes a whole system thanks to a library of 7 architectural elements (components, connectors, ports, roles, systems, representations and representation maps), and manage several architectural families (Pipe & filters, Client & servers, Pub-Sub,..).

In addition, Acme provides a tool to build easily our architecture: *AcmeStudio*. *AcmeStudio* is very simple to use. It is based on an Eclipse interface, with a main view to draw the architecture, a group tabs with properties, and some tools to draw in another tab. We have to choose at the beginning which family of components we want to use in order to build our architecture, that's all! Besides, the corresponding code is automatically generated when we draw the architecture. That's the main window of *ACMEStudio*:



Properties are not available.

Figure 5: *AcmeStudio* main window

The next picture represents our FDIR architecture by using this tool.

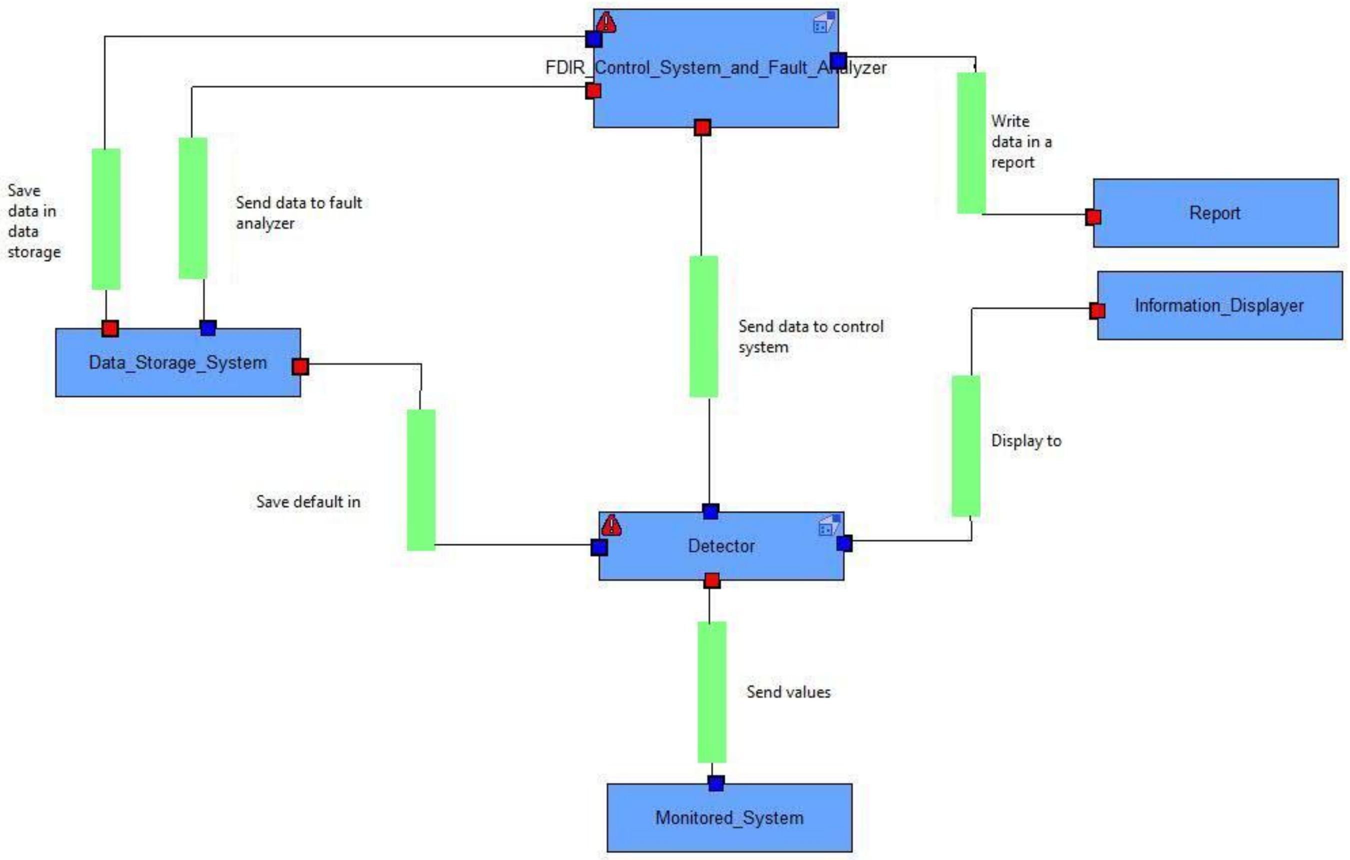


Figure 6: Proposed architecture using P/S style (global view)

Here we can see the different components of the architectures we made before. So we have the monitored systems which send values to the detector. All the components are connected by using an event bus. This bus stored the information during the time the receiver component process the previous data. The detector is connected to almost all systems in order to analyze, store and display for the crew the problems that occurred. The control system and fault analyzer is connected to the data storage system in order to save and bring the data it needs, and writes them in a report.

But the top component is divided in two, the control system and the fault analyzer:

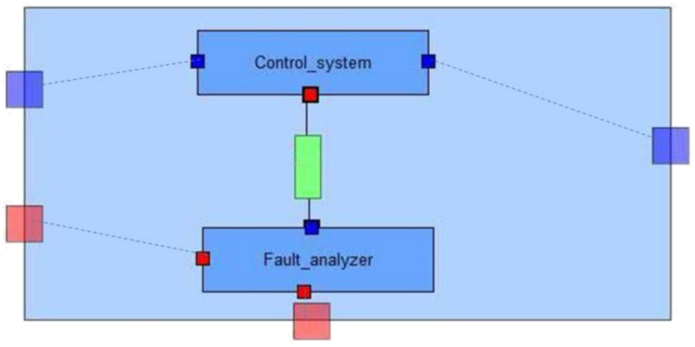


Figure 7: Proposed architecture using P/S style (control system and fault analyzer

That’s a strength of ACMEStudio : we can represent a subsystem inside a component, in order to have a better understanding of the system.

The top component has a subsystem too:

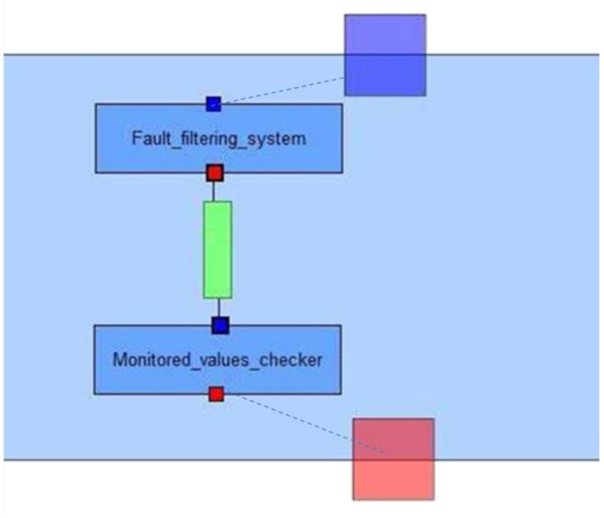
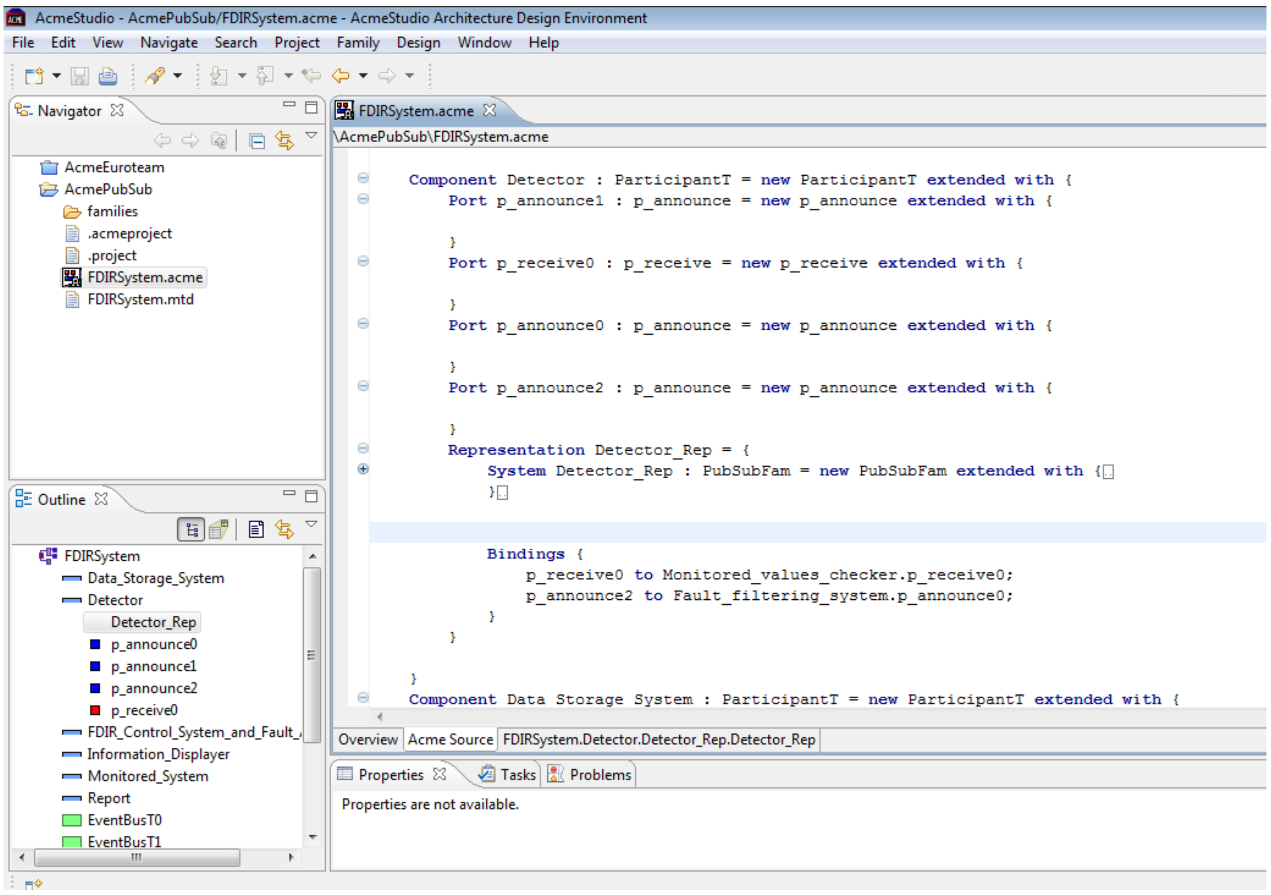


Figure 8: Proposed architecture using P/S style (fault filtering system & monitored values checker)

Here is a screen of the source code tab, which displays in real time the code corresponding to the architecture we draw:



```
Component Detector : ParticipantT = new ParticipantT extended with {
    Port p_announce1 : p_announce = new p_announce extended with {

    }
    Port p_receive0 : p_receive = new p_receive extended with {

    }
    Port p_announce0 : p_announce = new p_announce extended with {

    }
    Port p_announce2 : p_announce = new p_announce extended with {

    }
    Representation Detector_Rep = {
        System Detector_Rep : PubSubFam = new PubSubFam extended with {□
        }□


        Bindings {
            p_receive0 to Monitored_values_checker.p_receive0;
            p_announce2 to Fault_filtering_system.p_announce0;
        }
    }

}
Component Data_Storage_System : ParticipantT = new ParticipantT extended with {
```

Properties are not available.

Figure 9: Proposed architecture using P/S style (source code sample)

A possible alternative to this publish/subscribe architecture would be to use the layered components that we described before for our overall architecture. As ACME studio allows us to use representations, that is to say to define sub-architecture within one component, we can easily think of describing the Detector component and the *FDIR_Control_System_and_Fault_Analyzer* component with their own layered architecture. Unfortunately, ACME tool doesn’t provide the layered architecture family, so we cannot build this directly with ACME studio. Our wish is to provide this alternate architecture by drawing it on this report.

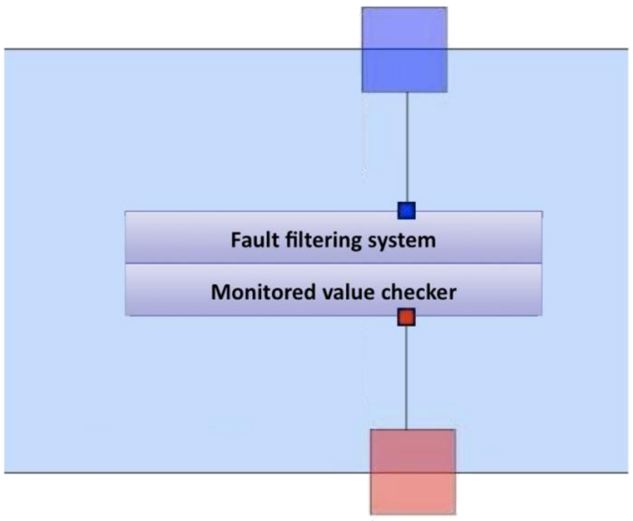


Figure 10: Proposed architecture alternative using layered style (Layered detector)

Here is the layered Detector component. We can imagine that, instead of components sending simple message to each other, these two sub-systems act as independent data processors. The monitored value checkers just collect the upcoming values from the several spacecraft’s systems. The fault filtering system access to this layer directly and so can perform various operations on the collected values.

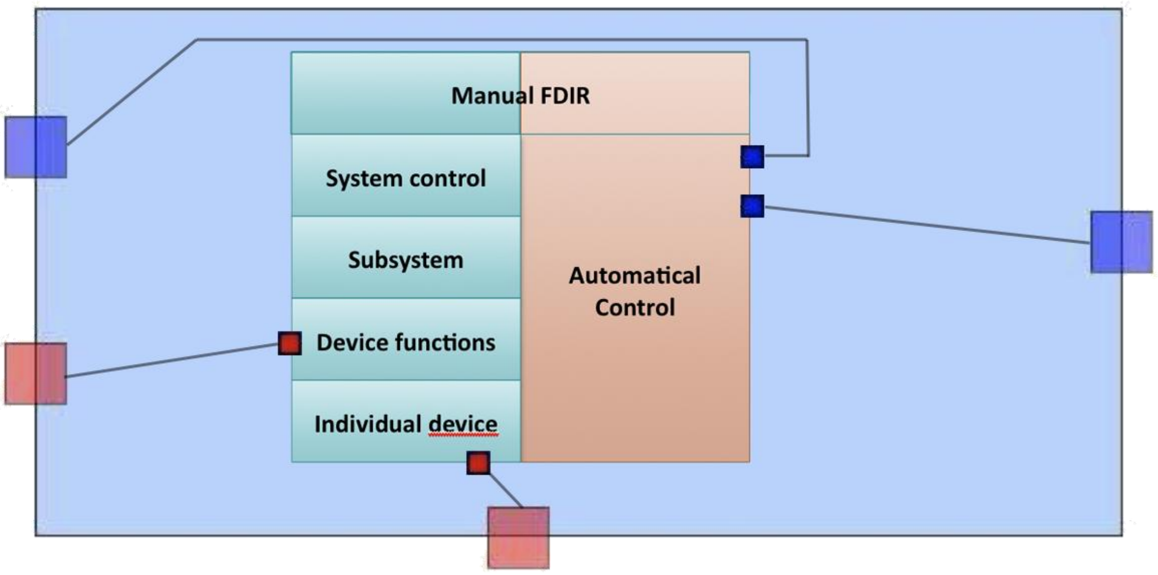
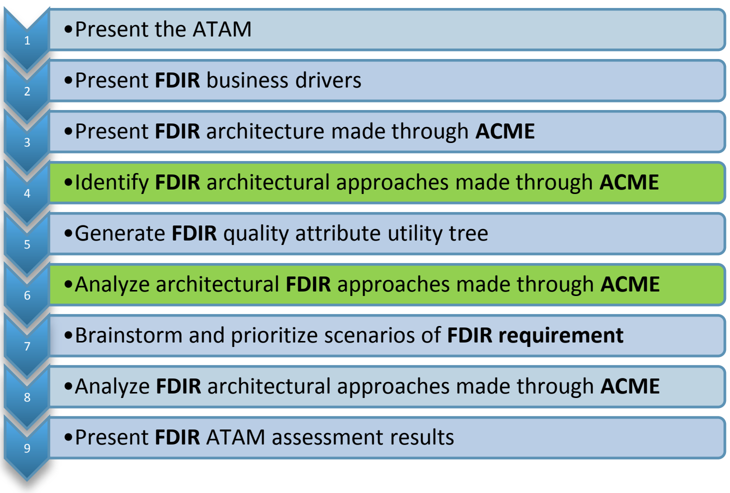


Figure 11: Proposed architecture alternative using layered style (Layered control system and fault analyzer)

This figure shows the layered fault analyzer and controller. The four analyzer layers are performing operations on values from different level of spacecraft’s systems: at the individual device level, then at the functions level, subsystem and system control. If the fault cannot be analyzed by the first layer with a satisfying result, the data is sent to the next upper layer, and so on… If the fourth layer cannot resolve the problem, the data is sent to the manual layer. The control is performed automatically by the FDIR when a fault is resolved by one of the fourth first layers. If the fault isn’t caught, the fault is sent to the manual layer and the control has to make manually by the user. We can think this system with a layered architecture. Each layer are so working independently and each layer can access to the data coming from the first underneath layer and can perform actions on its.

4. Architectural approach analysis

We will base our architectural analysis on the step 6 and 7 of ATAM approach. This will help us to identify and analyze the FDIR architectural approaches made through ACME. We will see and analyses how we respect our requirements and quality attributes in the architecture we proposed. We will also talk about the strengths and weaknesses of our architecture compared to others. To conclude, we will propose other architectures and discuss about that.


1 •Present the ATAM
2 •Present FDIR business drivers
3 •Present FDIR architecture made through ACME
4 •Identify FDIR architectural approaches made through ACME
5 •Generate FDIR quality attribute utility tree
6 •Analyze architectural FDIR approaches made through ACME
7 •Brainstorm and prioritize scenarios of FDIR requirement
8 •Analyze FDIR architectural approaches made through ACME
9 •Present FDIR ATAM assessment results

a. FDIR & ADL specifications compliance

i. The Publish/Subscribe pattern

FDIR system is a monitoring system involving several interactions between several components. This is really important to have an architecture allowing direct and reliable communications between components of FDIR sub-systems. Interacting with each other will allow them to localize, treat and fix the errors on systems. Moreover, we need asynchronous communications between components. P/S style is one the architecture point out this possibility of having direct and indirect multiple and multi directional messages. Thanks to *topic-based* P/S architecture, we can focus on a limited number of kinds of event like control operation, monitored values, analysis result, reports, etc. To finish, we can of course use P/S architecture directly into ACME Studio which furnish a complete interface for this architecture type.

ii. FDIR system organization

The layered fault analyzer and control system are better defined using a waterfall of publishers/subscribers. Moreover, we decided to combine P/S style with layers family for some sub-systems as it is respecting even more the waterfall and the way the FDIR system works.

As we tried to divide the architecture into different systems, we were glad to be able to respect this idea of design into ACME Studio. This division was allowed thanks to the feature called “representation”, where we can design a system regrouping sub-systems, and entering into this system to design more detailed sub-systems or even change of architectural style. Powerfulness of our proposed architecture is to combine the P/S style for the global system overview with the layers style for some sub-systems like the *fault analyzer*.

iii. Utility tree and non functional requirements

Availability & reliability are reached thanks to the loosed-coupling components in publish/subscribe architectural style. Even if one sub-system is down or not available, global FDIR system still work and will be able to localize and fix error, or even to let the crew switch to manual control. Crew has to trust the FDIR system as it is available at any time. Availability bring reliability!

Recoverability is respected thanks to the independence of entities (publishers & subscribers) that can recover from failure while FDIR global system is still working. As FDIR system is able to repair bugs on his own, it should also be able to recover from failure on his own.

b. Pros & cons of our architectural design

i. Architectural strengths

One of the strength of the FDIR Publish/Subscribe architecture is that it is **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).

FDIR will be **scalable** as well if we respect the P/S architecture. 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.

ii. Architectural weaknesses

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.

Even if our FDIR system architecture does not use brokers (used in topic-based P/S, but not in event-based P/S), 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 as the **fault analyzer** or the **monitored value detector** FDIR sub-systems.

c. Architectural type comparisons

Here is a comparison table of different architecture type that can used for FDIR architecture:

Architectural style	Scalability	Availability	Security	Maintenance
Pub-Sub	Good in the small	Good, because of decoupling	Potentially compromised because of decoupling	Possible complex due to distributed nature
Client-Server	Good in the large	Potentially compromised because of single point of failure	Strong because of central control	Usually simple because of centralized control
Layered	Good in the small	Bad, because of dependencies between components	Potentially compromised because of decoupling	Mitigated: good because of decoupling but bad because of independency
Pipe & filter	Good in the small	Bad, because of dependencies between components	Potentially compromised because of decoupling	Mitigated: good because of decoupling but bad because of independency

In P/S, systems are just in relation because of communications processes. They can stay independent from each other and this is what we want for FDIR system. In layered or pipe and filter architecture, if one system fell or break down, this will generate general error in the central system and will entirely block the system working.

Moreover, communications in pipe & filter and layered architectures are made from one component to another in a circular way: every component has one entrance and exit and will be related to 1 or 2 components. In P/S architecture, components will be able to communicate between each others, and with everybody. Rules will be determines between them to determine the publishers and the subscribers.

In order to study and compare more precisely the architecture we made using the P/S architectural style, we decided to build it using another style we though that could fit: the client-server architecture:

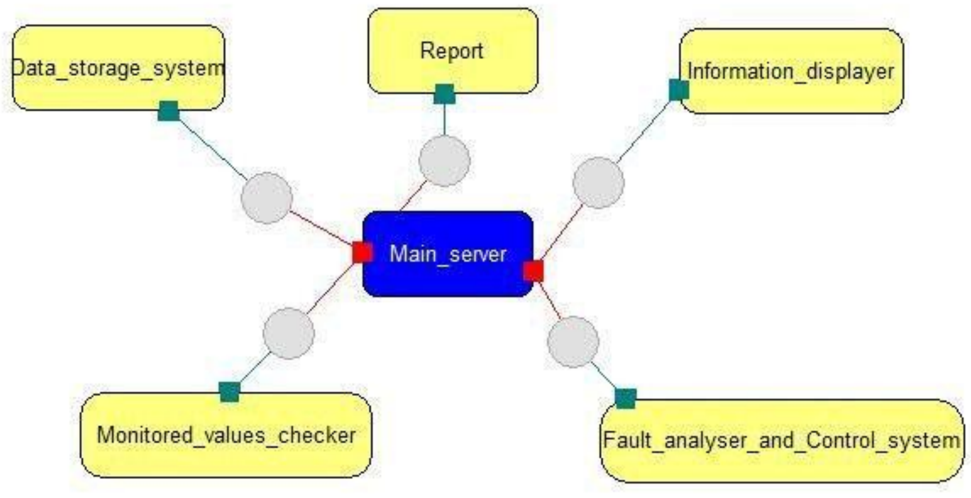
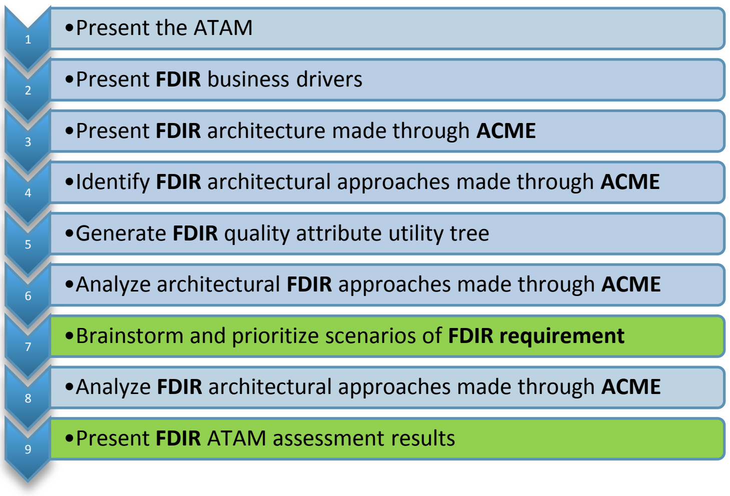


Figure 12: Client-server FDIR alternative architecture

Powerfulness comes with the centralization of commands and data. The problem with our FDIR system is that the system has to be reliable and available at any time that cannot be assured by client server architecture. Indeed, if a single server is broken, we lose data and control. FDIR cannot be able to work properly. This is the main advantage of P/S compared to client server architecture.

After representing the FDIR architecture through client-server family, we discovered some good and bad points. But as our main concern is availability and reliability, we were more confident choosing our first architectural proposition using P/S family type.

5. Discussions & evaluations


1
•Present the ATAM
2
•Present FDIR business drivers
3
•Present FDIR architecture made through ACME
4
•Identify FDIR architectural approaches made through ACME
5
•Generate FDIR quality attribute utility tree
6
•Analyze architectural FDIR approaches made through ACME
7
•Brainstorm and prioritize scenarios of FDIR requirement
8
•Analyze FDIR architectural approaches made through ACME
9
•Present FDIR ATAM assessment results

This last part will discuss about the risks, non-risks, sensitivity points, and tradeoff points. As part of this discussion, we will list our key architectural decisions and evaluate a set of possible alternatives. We are currently using step 7 & 9 of ATAM process to make this part. This section contains analysis of some scenarios as well as discussion about identified risks, non-risks, sensitivity points, and tradeoff points.

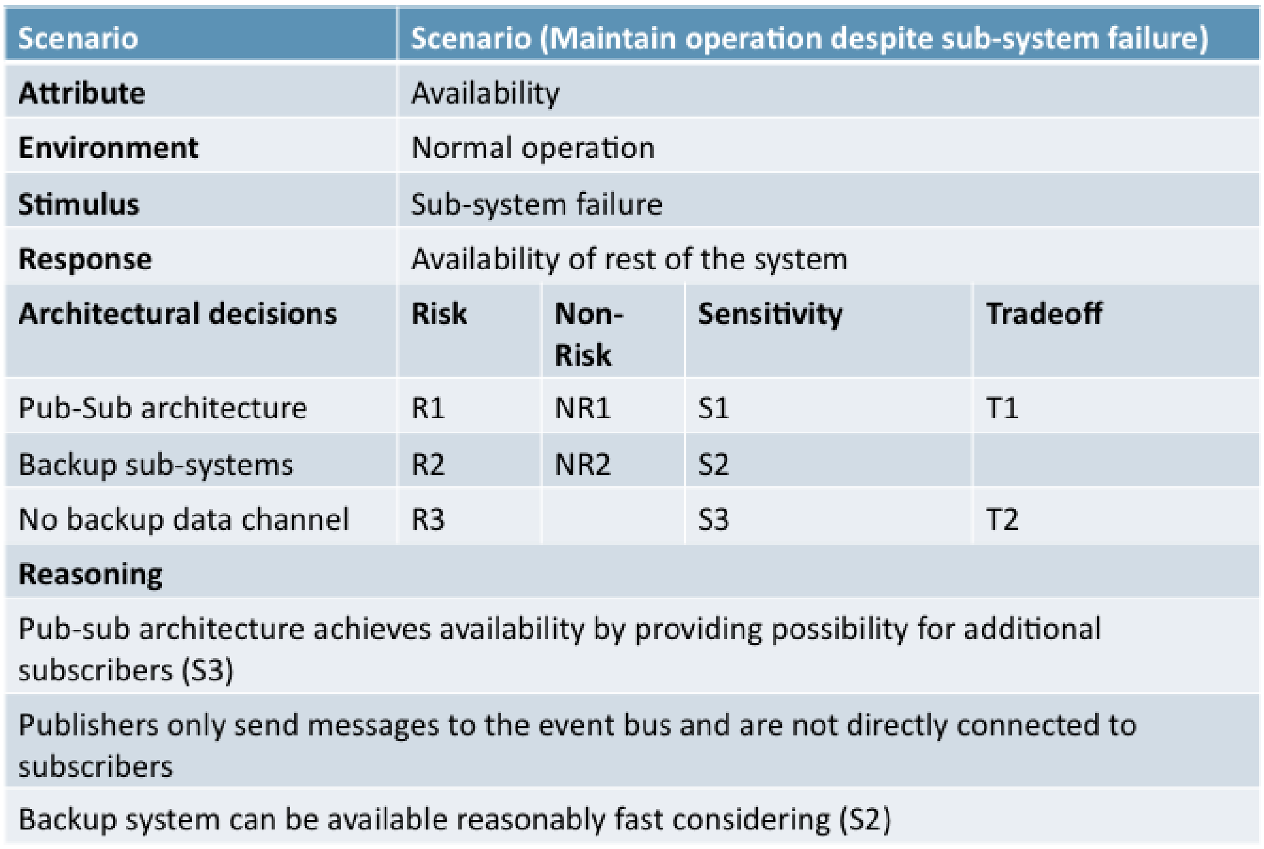
a. Scenarios

i. Maintain operation despite sub-system failure

Figure 13 shows the analysis for scenario “maintain operation despite sub-system failure”. The scenario in question is related to availability and happens under normal operating conditions. The stimulus in this case is the sub-system failure and expected response is the availability of the rest of the system.

There are three architectural decisions that relate to this scenario. In particular, Pub-Sub architecture, Backup sub-systems and the fact the there is no backup data channel. These decisions bear some risks, non-risks, as well al, sensitivity, and tradeoff points, as illustrated in the figure.

The reasoning behind this scenario relates to the possibility given by Pub-Sub architecture. In particular by providing the possibility for additional subscribers as detailed by S3. Also, publishers are not directly connected to subscribers. In addition, the backup system can be reasonably fast as well.



Scenario	Scenario (Maintain operation despite sub-system failure)			
Attribute	Availability			
Environment	Normal operation			
Stimulus	Sub-system failure			
Response	Availability of rest of the system			
Architectural decisions	**Risk**	**Non-Risk**	**Sensitivity**	**Tradeoff**
Pub-Sub architecture	R1	NR1	S1	T1
Backup sub-systems	R2	NR2	S2	
No backup data channel	R3		S3	T2
Reasoning				
Pub-sub architecture achieves availability by providing possibility for additional subscribers (S3)				
Publishers only send messages to the event bus and are not directly connected to subscribers				
Backup system can be available reasonably fast considering (S2)				

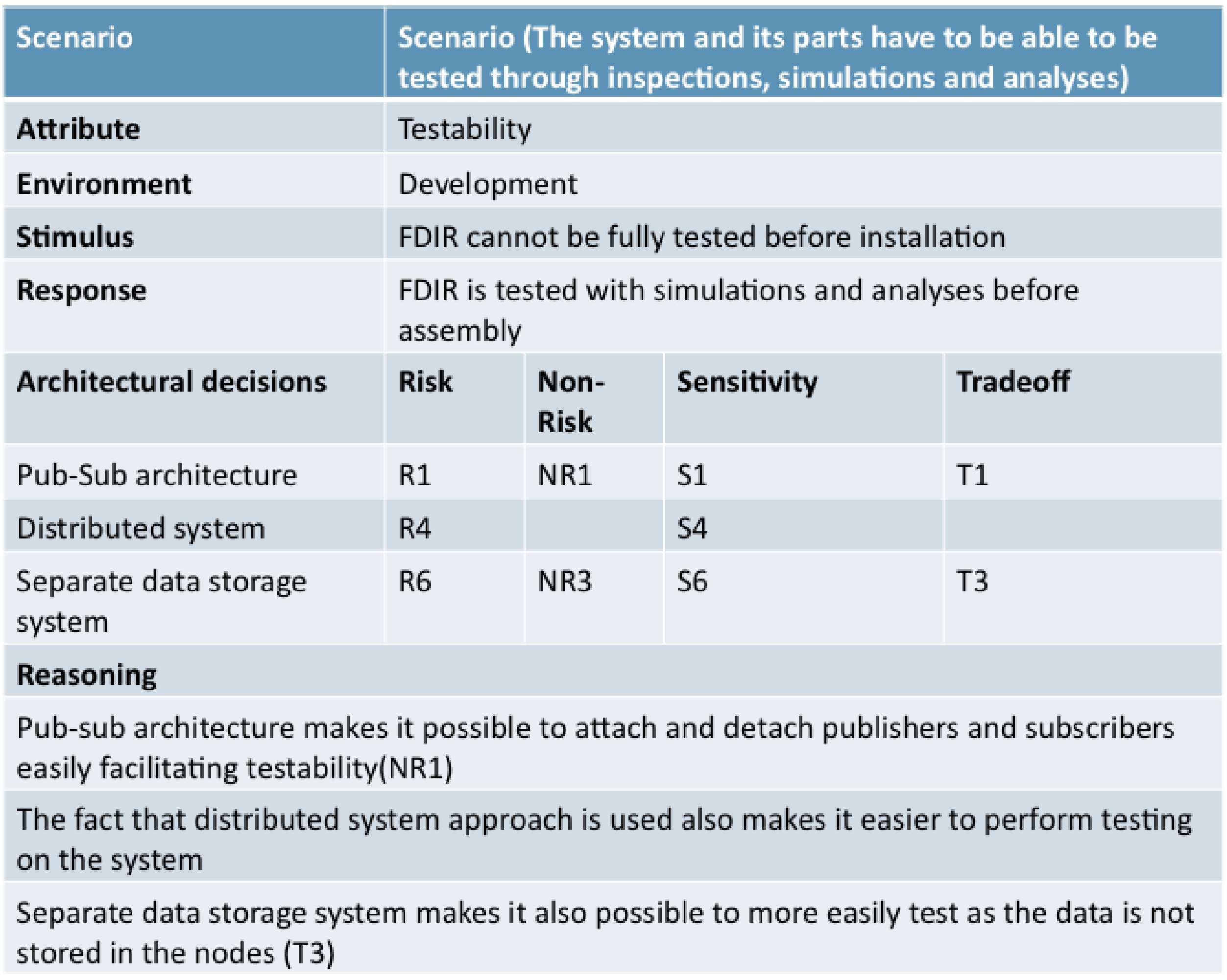
Figure 13: Scenario (Maintain operation despite sub-system failure)

ii. The system and its parts have to be able to be tested

Figure 14 illustrates the scenario "The system and its parts have to be able to be tested through inspections, simulations and analyses" The attribute in question here is testability and the environment is developmental. The fact is the FDIR cannot be assembled and fully tested before installation to the space station. Therefore it is tested with simulations and analyses and inspections before assembly.

The associated architectural decisions are Pub-Sub architecture, distributed system and the fact that there is separate data storage system installed. They all have some risks and sensitivity points to them.

The reasoning behind this scenario is that Pub-Sub architecture makes it possible to attach and detach publishers and subscribers, which facilitates testability. In addition, distributed system approach also makes testability easier. When data is not scattered throughout the system and is stored in one place it's also easier to devise testing scenarios and such.



Scenario	Scenario (The system and its parts have to be able to be tested through inspections, simulations and analyses)			
Attribute	Testability			
Environment	Development			
Stimulus	FDIR cannot be fully tested before installation			
Response	FDIR is tested with simulations and analyses before assembly			
Architectural decisions	**Risk**	**Non-Risk**	**Sensitivity**	**Tradeoff**
Pub-Sub architecture	R1	NR1	S1	T1
Distributed system	R4		S4	
Separate data storage system	R6	NR3	S6	T3
Reasoning				
Pub-sub architecture makes it possible to attach and detach publishers and subscribers easily facilitating testability(NR1)				
The fact that distributed system approach is used also makes it easier to perform testing on the system				
Separate data storage system makes it also possible to more easily test as the data is not stored in the nodes (T3)				

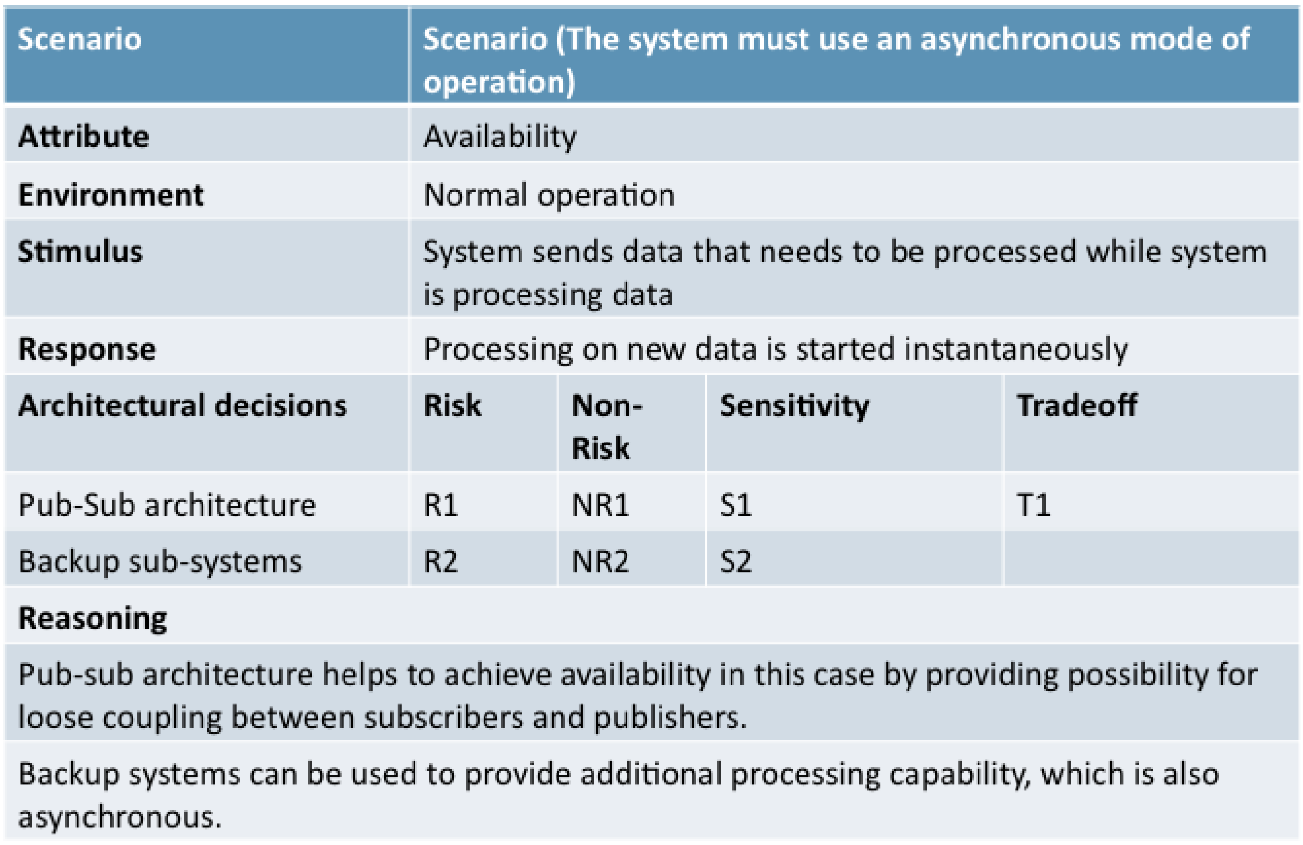
Figure 14: Scenario (The system and its parts have to be able to be tested through inspections, simulations and analyses)

iii. The system must use an asynchronous mode of operation

Figure 15 illustrates the scenario “The system must use an asynchronous mode of operation”. This scenario concerns availability and happens under normal operating conditions. The basic premise is that the system is processing data, while a new input is received. Under these conditions the system should be able to start processing new inputs simultaneously.

Architectural decisions involved are the Pub-Sub architecture and backup sub-systems. The former has risk, non-risk, sensitivity point and tradeoff point included, while the latter is lacking the tradeoff point.

Reasoning is that the Pub-Sub architecture helps to achieve availability in this case by providing loose coupling between the entities in the system. In addition backup system can in some cases provide additional processing capability to levitate the processing burden.

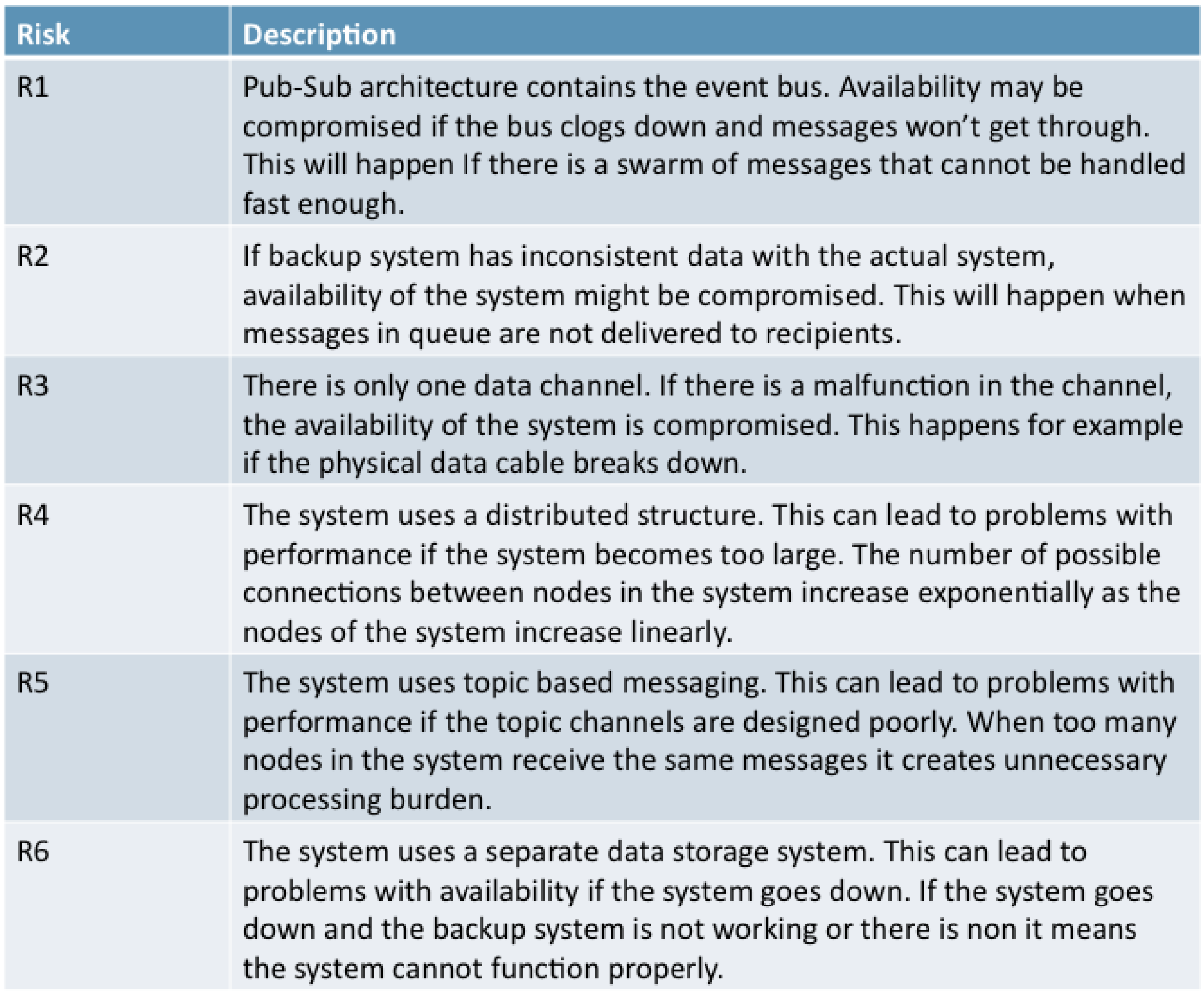


Scenario	Scenario (The system must use an asynchronous mode of operation)			
Attribute	Availability			
Environment	Normal operation			
Stimulus	System sends data that needs to be processed while system is processing data			
Response	Processing on new data is started instantaneously			
Architectural decisions	**Risk**	**Non-Risk**	**Sensitivity**	**Tradeoff**
Pub-Sub architecture	R1	NR1	S1	T1
Backup sub-systems	R2	NR2	S2	
Reasoning				
Pub-sub architecture helps to achieve availability in this case by providing possibility for loose coupling between subscribers and publishers.				
Backup systems can be used to provide additional processing capability, which is also asynchronous.				

Figure 15: Scenario (The system must use an asynchronous mode of operation)

b. Risks

Figure 16 illustrates the identified risks in the systems that are related to the architectural decisions that were made in the system design. Some of these risks are really manifestations of sensitivity points in another form. Often sensitivity points can turn in to risks in the system implementation.

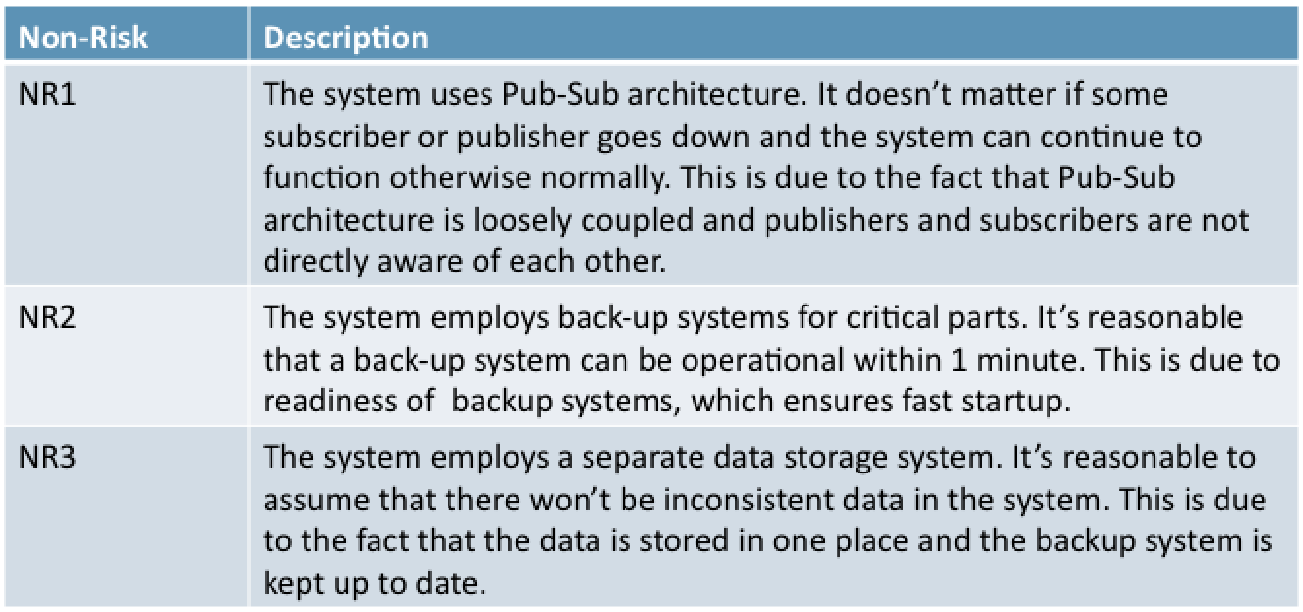


Risk	Description
R1	Pub-Sub architecture contains the event bus. Availability may be compromised if the bus clogs down and messages won't get through. This will happen If there is a swarm of messages that cannot be handled fast enough.
R2	If backup system has inconsistent data with the actual system, availability of the system might be compromised. This will happen when messages in queue are not delivered to recipients.
R3	There is only one data channel. If there is a malfunction in the channel, the availability of the system is compromised. This happens for example if the physical data cable breaks down.
R4	The system uses a distributed structure. This can lead to problems with performance if the system becomes too large. The number of possible connections between nodes in the system increase exponentially as the nodes of the system increase linearly.
R5	The system uses topic based messaging. This can lead to problems with performance if the topic channels are designed poorly. When too many nodes in the system receive the same messages it creates unnecessary processing burden.
R6	The system uses a separate data storage system. This can lead to problems with availability if the system goes down. If the system goes down and the backup system is not working or there is non it means the system cannot function properly.

Figure 16: Identified risks

c. Non-risks

Figure 17 identifies the non-risks associated in the design. While non-risks are not as important as risks they provide important pointers to what can be trusted on in the system design based on some particularities in the design.

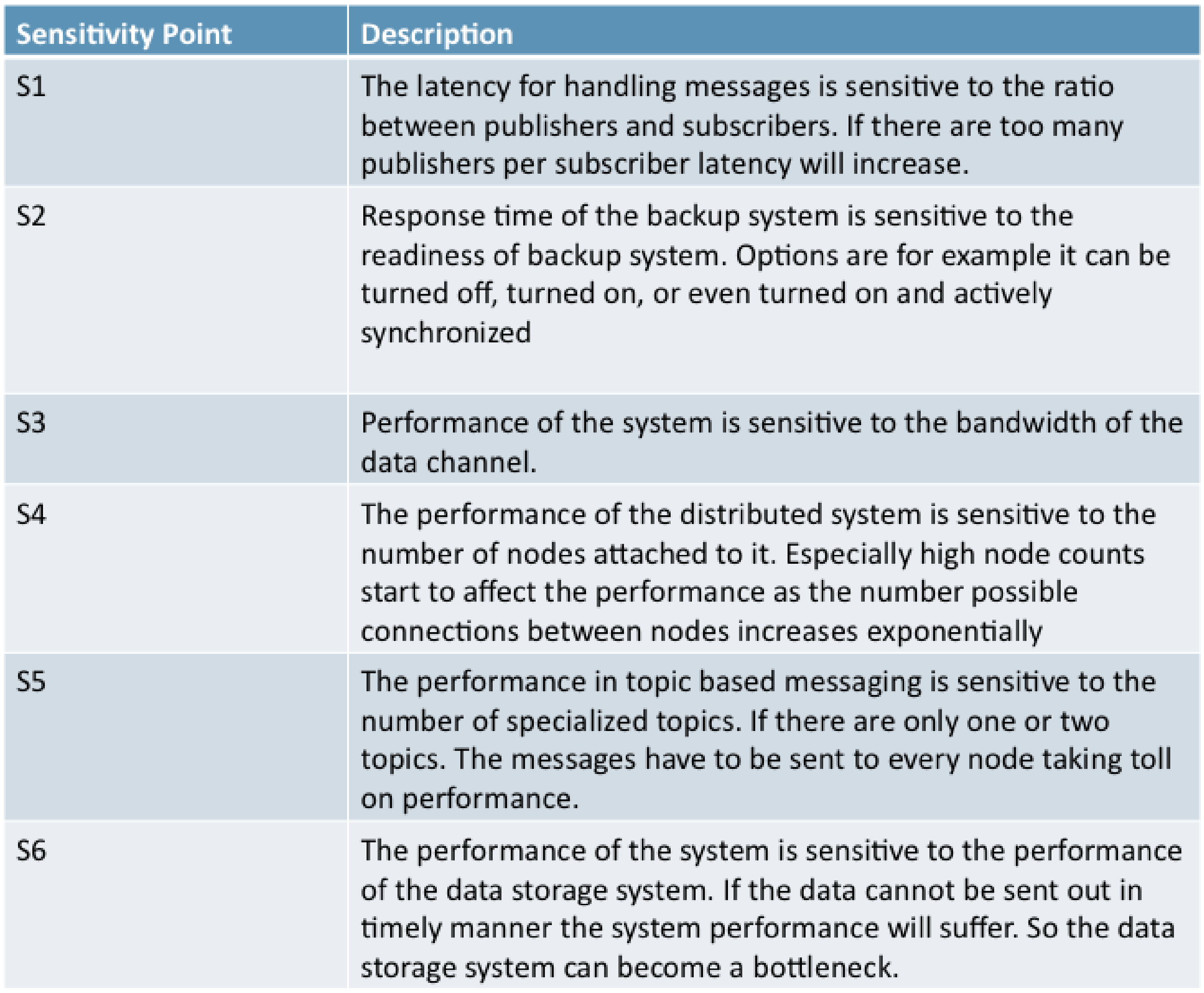


Non-Risk	Description
NR1	The system uses Pub-Sub architecture. It doesn't matter if some subscriber or publisher goes down and the system can continue to function otherwise normally. This is due to the fact that Pub-Sub architecture is loosely coupled and publishers and subscribers are not directly aware of each other.
NR2	The system employs back-up systems for critical parts. It's reasonable that a back-up system can be operational within 1 minute. This is due to readiness of backup systems, which ensures fast startup.
NR3	The system employs a separate data storage system. It's reasonable to assume that there won't be inconsistent data in the system. This is due to the fact that the data is stored in one place and the backup system is kept up to date.

Figure 17: Identified non-risks

d. Sensitivity points

Figure 18 illustrates the sensitivity points that were identified from the architectural approaches that were made in the system design. Some of these sensitivity points realize as risks in the actual design. They are important pointers to what should be carefully considered while implementing the architecture. If they are not paid heed to the system might not work as intended at least in regard to some specific quality attributes that are important to the system operation as specified by requirements, or the quality attribute three scenarios.

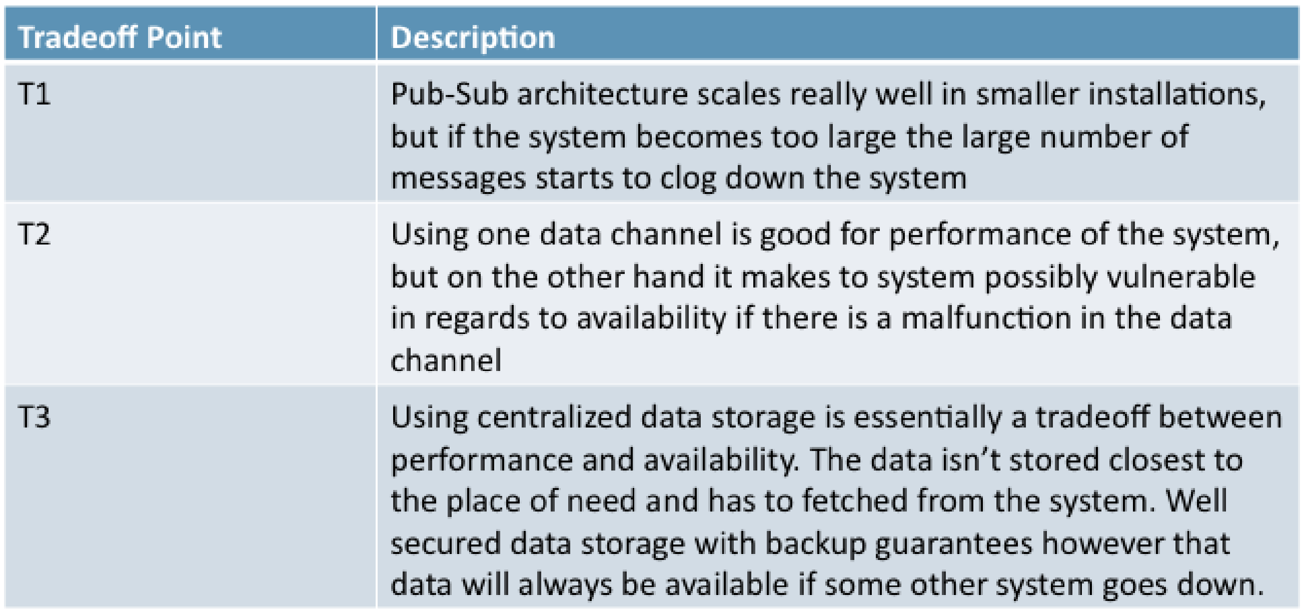


Sensitivity Point	Description
S1	The latency for handling messages is sensitive to the ratio between publishers and subscribers. If there are too many publishers per subscriber latency will increase.
S2	Response time of the backup system is sensitive to the readiness of backup system. Options are for example it can be turned off, turned on, or even turned on and actively synchronized
S3	Performance of the system is sensitive to the bandwidth of the data channel.
S4	The performance of the distributed system is sensitive to the number of nodes attached to it. Especially high node counts start to affect the performance as the number possible connections between nodes increases exponentially
S5	The performance in topic based messaging is sensitive to the number of specialized topics. If there are only one or two topics. The messages have to be sent to every node taking toll on performance.
S6	The performance of the system is sensitive to the performance of the data storage system. If the data cannot be sent out in timely manner the system performance will suffer. So the data storage system can become a bottleneck.

Figure 18: Identified sensitivity points

e. Tradeoff points

Figure 19 describes tradeoff points that were identified from the architectural approaches taken in the system design. Not all architectural decisions entail tradeoffs however. Some are purely made to achieve a purpose that benefits that architecture in regards to some specific quality attribute. In essence, trades off points are sensitivity points for multiple quality attributes in the architecture.



Tradeoff Point	Description
T1	Pub-Sub architecture scales really well in smaller installations, but if the system becomes too large the large number of messages starts to clog down the system
T2	Using one data channel is good for performance of the system, but on the other hand it makes to system possibly vulnerable in regards to availability if there is a malfunction in the data channel
T3	Using centralized data storage is essentially a tradeoff between performance and availability. The data isn't stored closest to the place of need and has to fetched from the system. Well secured data storage with backup guarantees however that data will always be available if some other system goes down.

Figure 19: Identified tradeoff points

Conclusion

In the purpose of applying and studying real case project for the *Design for Softwares and Systems* course, our team has been 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 step of this job was 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.

We have learned a lot about taking the good decisions to build architecture and justify the way we do it. Thanks to different methods like ATAM, we were able to start from the FDIR requirements and go through different steps of analyze to come to a proposed architectural design strongly justified and compared to others. We have made different choices of architectures family and non functional requirements. Rules and tools we used were the core of our design and the FDIR working system.

Also, we learned about analyzing and criticizing a self made architecture, comparing it with other alternatives. Even more, we build another type of architecture in order to compare the pros and cons of different solutions, and tried to make the best choices about it.

To conclude the design proposition and analyses, we brought and provide the output of architectural analysis facilitate communicating the architectural design to stakeholders.

Beside all these point, we focus on learning about ATAM process, ADL use and of course ADL tools in order to build and propose the FDIR architectural design. For constructing our architecture, we have used the ACME architectural description language. 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.

Both technical and theoretical learning points were really important and new for us. That was really interesting and necessary to focus on other part of computer science design. As we just knew about UML or other conceptual languages before, but this project was really formative because it helped us to understand why combining different techniques and methods is necessary and essential in building softwares and systems architectures.

References

Web Sites

1. http://en.wikipedia.org/wiki/Architecture_description_language
2. http://www.cs.cmu.edu/~acme/pub/xAcme/
3. http://en.wikipedia.org/wiki/Atam
4. http://en.wikipedia.org/wiki/Publish_subscribe
5. http://en.wikipedia.org/wiki/Client_server
6. http://en.wikipedia.org/wiki/Layer_%28object-oriented_design%29

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