Flowing Taks: Scalable Software Dependability and Performance

Sergio Montenegro [(1)], **Raffaele Vitulli** [(2)],

[(1)] *FIRST, Kekuléstraße 7, 12489 Berlin, Germany, Email: sergio.montenegro@first.fraunhofer.de*

[(1)] *ESA/ESTEC, Keplerlaan 1, 2201AZ Noordwijk, The Netherlands, Email: Raffaele.Vitulli@esa.int*

1. The Target

1. Dependability is a major challenge when designing space computing systems, both at software and hardware level. After 30 years of contentious research on how to achieve high dependability, not a single solution has been found. A huge effort has been invested to improve reliability, using reliable radiation hardened components. However, failures cannot be eliminated totally.

2. Nowadays, it is not imaginable the use of the same (general purpose) computer in a cube sat (10x10x10 cm^3, 1 Kilogram) and in a mini satellite. For each satellite, a special purpose computer configuration need to be assembled, consisting of board computer, payload for cameras, star tracker, power control, reaction wheels, etc.

2. Conceptual obtacles

From our experience we can identify at least 4 important very common conceptual obstacles when designing a fault tolerant system.

1) Any digital control system is a team work of software and hardware. Trying to solve any problem or challenge using only one of them means to loosing a lot of possibilities, like a team (Software-Hardware) where one player in software, but very few try to face the challenge in a team work of software and hardware.

2) Another wrong assumption is to think that to provide fault tolerance you just have to replicate resources. For example if you can control a water tank level opening and closing a valve, like in Fig. 1.

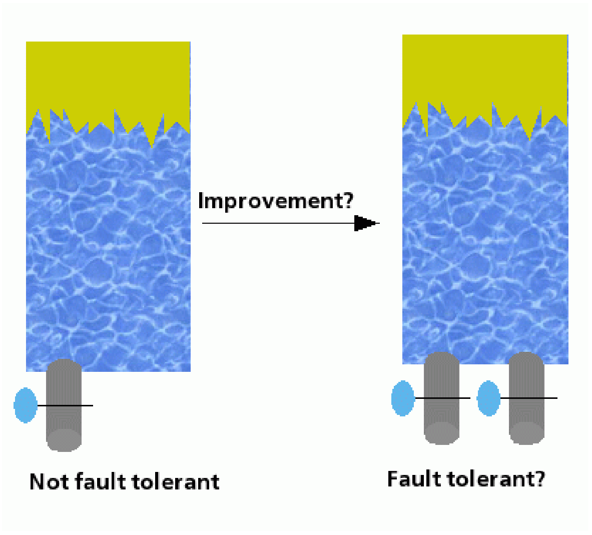

Figure 1: Replication alone is not fault tolerance

Then you may think: The valve can fail; let's add another valve (Fig. 1). Now one of the redundant valves fails in open position, you can not control the tank any longer. What you got is a higher possibility of failure and not more dependability. This was not right the solution!

3) Another wrong assumption is to think you can reach an absolute failure free system. Then an enormous effort is invested to reach what you can not reach. See Fig. 1. You can do what you want, but any system will crash some day.

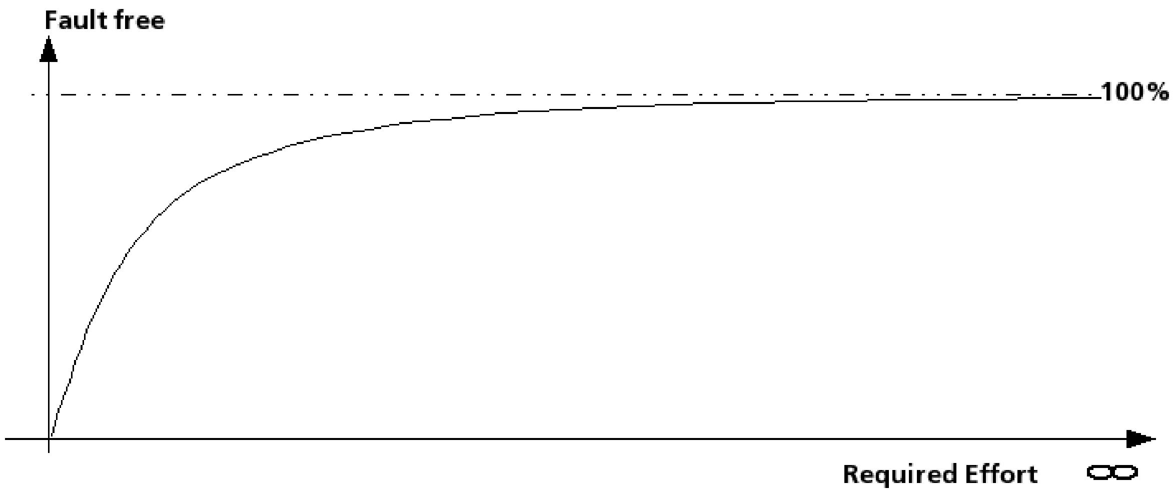

Figure 2: Infinite effort to reach a total fault free system

4) Another important consideration is what to do after so many resources are permanently damaged, that no normal operation is possible any longer.

3. A non conventional way to dependability

Let's consider (in this paper) 4 (from 0 to 3) stages of fault tolerance:

0: No fault tolerance. One failure is enough to cause a system failure or system crash

1: Any single failure can collapse some critical functions, but the system remains in a safe operation. (Gracefully degradation)

2: Like 1, but after a short time period the system can reconfigure itself to go back to normal operation (resilience)

3: No single failure can disturb the normal operation of the system.

Stage 3 sounds very nice, but it requires at least a 3 fold replication of (almost) all resources, including volume, mass, power consumption, heat production, etc. For some space applications this can be a prohibitive condition. Furthermore after the first permanent failure, the system will go down to stage 2 or 1. For long operations where permanent failures are expected, stage 3 will be possible only at the beginning of the mission.

It is therefore advisable to create a resources economical system which provides stage 2 for the whole mission even after several failures. Our target is to get the highest possible dependability using very limited resources. The system has to be operable always in a safe way, but temporally loose of advanced features is allowed. Thereby it was very important not to do again the common mistakes listed in chapter 1.

3.1. Step 1: Let it crash!

A realistic node computer operation can be represented like in Fig. 3.

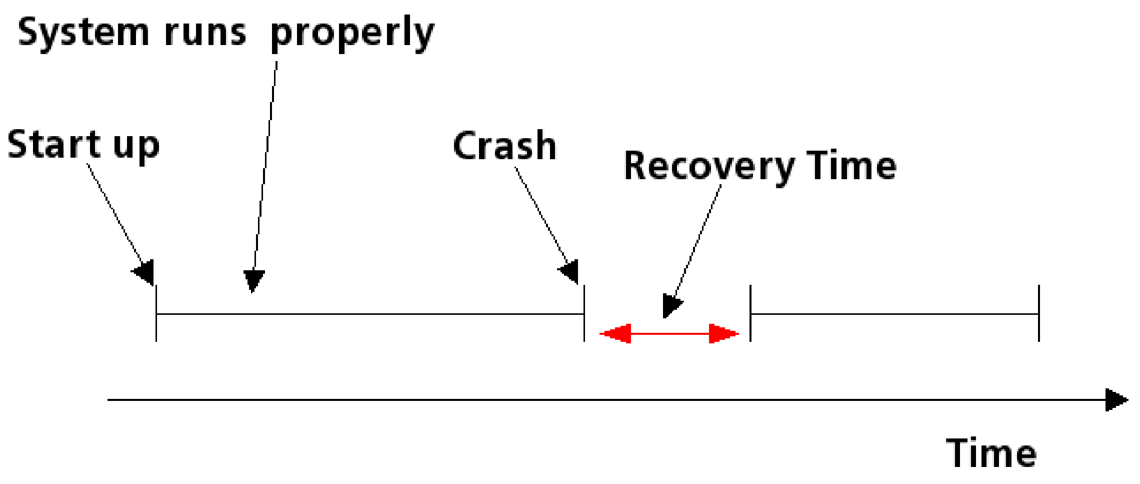

Figure 3 realistic node computer operation.

After start up a node computer will work properly for a time period (may be years) but some day it will crash and no one can impede this. Better than trying to create a (sub) system which shall never fail or where a failure is an extraordinary exception, we count with failures, they are not exceptions, but expected events which will be handles smoothly. The paradox situation is: to be safe, be ready to crash at any time. To increase the dependability of the system we can increase the reliability of each node (eg. mean time to failure MFTF) and reducing the recovery time after crashes (increasing Availability).

Much more important than try to avoid crashes (we can not) it to provide an ultra fast recovery. Recovery includes reboot, restore context of applications and update the context to the current situation. Having a very short recovery time, we can tolerate very easily multiple node crashes, and if we use some kind of redundancy, the probability of two redundant nodes to be down at the same time becomes lower. See Fig. 4

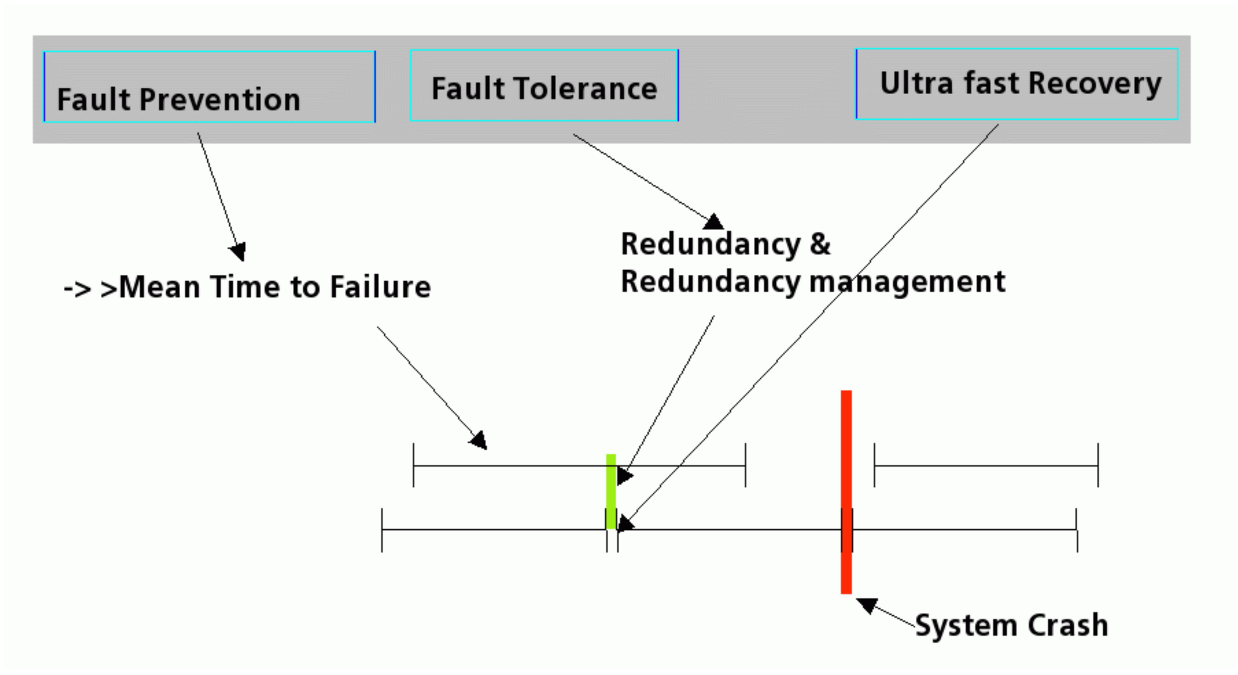

Figure 4: Increasing dependability

Today the recovery time is very long, depending on the operating system it can range from 10 seconds to some minutes. This long period increase the probability of a total system failure if redundant nodes are down at the same time (see Fig. 4). Or target is to have a recovery time of a few milliseconds. If so, we can tolerate several crashes per second without problems.

3.2. Step 2: Do not paddle only on one side, use both: Software and Hardware.

Better than trying to have fault tolerance in Hardware and to make it transparent to the software (e.g. Maxwell approach), we implements robustness as a harmonized team work of software and hardware. If you have a legacy or extreme complex software, which you cannot modify (or even understand) then you have to make fault tolerance transparent to the software, but doing so you lose an enormous potential from HW-SW team work: Hardware provides redundancy and software provides an intelligent redundancy management.

Using an intelligent dynamic redundancy management it is possible to adapt the redundancy degree to the current situation and mission phase. The resources usage can vary to provide high performance (no/low redundancy) or high dependability (high redundancy).

4. **Our Approach / Our solutions**

Contrary to most other approaches who try to minimise the possibility of failures, this approach considers that failures will occur and tries to handle them efficiently. Dependability is implemented using smart and efficient resource and redundancy management. The novel approach is to focus on efficient software solution instead on adding more hardware.

We aim an artificial brain computing model which controls the whole spacecraft. It consists of a network of computing nodes capable of executing different tasks and autonomous task re-distribution. The brain metaphor implies capability of self organisation and adaptability. If some nodes fail, their tasks will be automatically moved to other working nodes (imitating the brain feature of healthy cells being capable of overtaking the functions of damaged or decease-affected cells). This simplifies the spacecraft/satellite construction having only one computer with autonomous configuration and continuity of service.

On the Hardware side our approach is based on a network of computing nodes (more than 3). On the software side our Approach is based on the BOSS operating system (Originally: Bird Operating System) which was designed to support dependability. Two major objectives are the ultra fast task recovery and ultra fast task migration mechanisms. An application with multiple tasks running on BOSS is distributed over a set of node computers. The communication between tasks is performed using named ports with a producer/subscriber protocol. The number of tasks, ports and the topology are transparent to the application. This allows redistribution of tasks without any programmer intervention. Consequently, any task can be replicated for redundancy purposes making the running system highly reconfigurable.

Dependably is based on a team work of software and undependable and inexpensive hardware. The designer assumption is that a node computer may crash each second. No other approach can handle such an extreme high crash ratio. This is an enormous advantage for example for satellites which operate in environments with very high radioactivity and electromagnetic noise. This approach is ready to handle crashes at any time.

5. BOSS Kernel and middleware

The BOSS real time kernel and middleware offer an integrated framework (OO) interface. There is no need to differentiate what is a RT kernel and what is middleware functionality. Both together will be called the BOSS-framework. The BOSS framework aims to offer the most simple and small possible interface to users tasks, which still provides all required functionality and flexibility.

The BOSS-framework includes time management, resource management and communication functionality. Without an application the framework is inactive, it just reacts to interrupts doing nothing. An application can add actions to the BOSS-framework by inheriting classes and creating active objects. These objects will be integrated automatically in the framework. In this way the framework will be extended with user functionality.

5.1. Communicating tasks

Let’s take as example a task network to control 3 arms using an environment model. The task distribution is shown in Fig. 5.

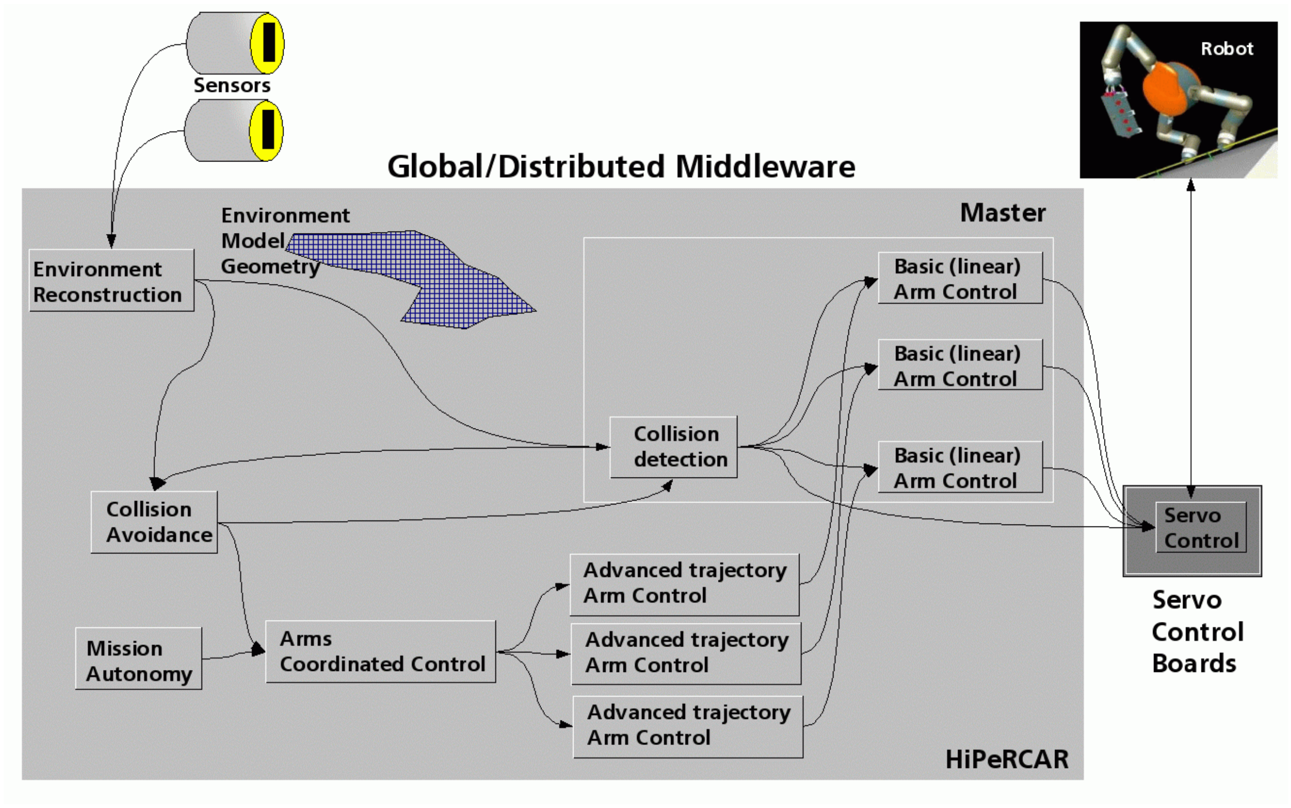
Sensors
Global/Distributed Middleware
Robot
Master
Environment Model Geometry
Environment Reconstruction
Basic (linear) Arm Control
Basic (linear) Arm Control
Collision detection
Basic (linear) Arm Control
Collision Avoidance
Servo Control
Servo Control Boards
Advanced trajectory Arm Control
Mission Autonomy
Arms Coordinated Control
Advanced trajectory Arm Control
Advanced trajectory Arm Control
HiPeRCAR

Figure 5: Hypothetical Example of Task Distribution

For the implementation there is no difference where the tasks will be deployed. The position of tasks can even change at run time, without requiring any explicit reaction of the other involved tasks. Task communicate using the BOSS Middleware, but its only interface are messages, the rest is transparent. Tasks just define input and output messages. The rest is some one else job (Middleware and real time kernel).

5.2. Structure of the BOSS framework

A BOSS controller is implemented as a collection of building blocks on the top of a middleware, which creates a virtual totally interconnection network for all tasks running on it. The middleware allows every task to communicate with any other task independent of their position. It imposes no limitations to communication paths, but the application shall use/create a meaningful, reasonable and efficient inter-task communication structure.

There may be different system configurations for BOSS applications. The system supports to migrate from any of these configurations to another without having to change the application. This will be achieved by implementing a layered system.

The application tasks run on the top of a virtual global middleware which cross node boundaries and interconnect all threads/tasks in the system. They do not need to know the details of lowered layers. Applications can interact without having to know about locations and hardware architectures.

The global middleware is implemented on local middleware running on each node. This local middleware runs on the top of a real time kernel or operating system which administrates the time and local resources in the node. The real time kernel is implemented using a virtual hardware abstraction, which is implemented by the hardware dependent layer. To move from any hardware platform to another one, only this small layer has to be re-implemented.

6. REFERENCES

Marra, et al. - *“HiPeRCAR: the High Performance Resilient Computer for Autonomous Robotics”* - DASIA 2006, Berlin.

Jameux, D. – *“Application of the Payload Data Processing and Storage System to MOSREM Multi-processor On-board System for Robotic Exploration Missions”*

Astrium/Scisys – *“UNIONICS Advanced On-Board Distributed Processing Architecture”* - Initial results from the Bench-Top Demonstrator”

M. Johnson, B. Green, O. Eman - EADS Astrium *“Improving the efficiency of spacecraft systems with the Unionics Advanced On-board Architecture”* – SDSS 2005

Guy Estaves, Alcatel Alenia Space – *Super Computers for Space Applications*- SDSS 2005, 17-20 October 2005, ESTEC, Noordwijk

SpaceWire – Links, Nodes Routers and Networks, ECSS-E-50-12A at ECSS-E-50-12 A
ESA ITT AO/1-4497/NL/AG – *SpaceWire Remote Terminal Interface ASIC*.