Ultra Fast Recovery

Sergio Montenegro
DLR-RY
Robert-Hooke-Str 7
23359 Bremen
Sergio.Montenegro@dlr.de

Abstract / Motivation

since the beginning of computer science era, computer scientists are concerned with avoiding crashes to provide dependability. After years of research no solution to avoid crashes was found. Lets try the opposite: Let it crash! Provide dependability by ultra fast recovery and intelligent use of redundancy.

Figure 1 shows the expected execution of any man made dependable system.

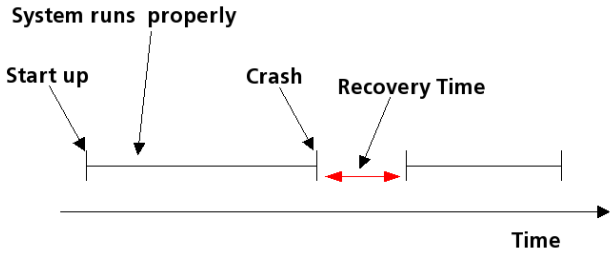

Figure 1: normal execution

After activation the system will work (more or less) properly for a (hopefully long) period of time. But some day it will crash. We can not avoid this. After a crash a recovery shall take place and the system shall continue its normal operation.
Using this normal behaviour (notice: crash is part of the normal behaviour) we create some parameters to quantify how good our system is:

Reliability: is a function of how long the system will work until the next crash, its unit is the mean time to failure (MTF). Notice: Failure is already expected and calculated!

Availability: is the provability the system is operational at any time point. It is a function of mean time to failure and recovery time: This is MTF / total time = MTF/(MTF + Recovery Time).
Notice: the case that we will need a recovery, is implied and calculated!

Dependability: is the mass in how much we may relay on the system. It is a function of reliability + availability + safety + security.

Safety and security are some times the opposite. For example a door which is very difficult to open (many keys) may be very secure, because unauthorized entry is difficult but it is very unsafe because in an emergency case (e.g. Fire) it may impede an escape. Similar is the case with an door which is very easy to open: safe but not secure. In our context, safety is more important, therefore we handle dependability as the function of reliability + availability + safety, but for this paper we will conciser only reliability and availability.

Our target is not just high reliability or high availability but the highest possible dependability. We conciser the key parameter for this is to reduce the recovery time and not so much to increase the mean time to failure. Even with a system with very short mean time to failure, we may get high dependability, if we have ultra short recovery time. Lets assume an (dreamlike) extreme case: Recovery time = 0. Note: recovery is not just reboot, but includes restore from previous context and all required status information to be able to continue operations. In this case even if we have a crash every second, the system is able to continue working instantaneously without loss of control. Any crash is invisible from outside. 0 time as recovery time is (now) not realistic, but lets assume a

recovery time far below the control cycle time (cycle time for example 500 ms) and a system which may tolerate one cycle without control. In this case a crash stays invisible for the system, because next control cycle will be normal again.

A second reason why to reduce the recovery time is to improve the redundancy management as fault tolerance means. Figure 2 show how a take over for redundant systems shall work.

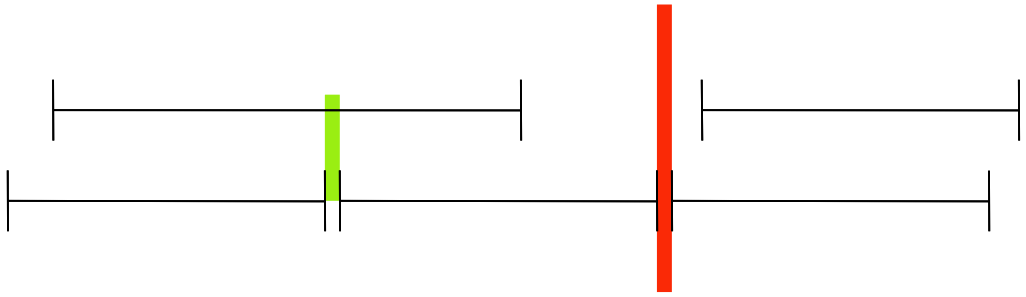

Figure 2: Redundancy as means for fault tolerance

Lets assume two redundant controllers, if one fails the second one shall take control as fast as possible, in any case faster than the control cycle. In the mean time the crashed controller shall perform a recovery and stay ready to take over in case of a second failure. If the recovery time is too long, then a crash of the still running controller, before the recovery is finish is quite possible. Then we would have a system failure and the redundancy as mean for fault tolerance was not effective. The same may apply to a triple module redundancy (TMR) even if the probability of a system failure is much lower.

Our approach: Network centric computing

Current space craft data handling systems are primary computer-oriented building computer-centric systems. In this model the central computer has to provide high computing power, large memory, high dependability, fault tolerance management, and many input/output connections to different devices. This makes the central computer development, very difficult, error prone and expensive. It makes the reuse of a board computer for different missions difficult because from mission to mission some IO devices may change.

In our approach we aim to build a network centric system, where the central element is not a computer but a powerful space craft area network (SCAN). The network is built using dependable intelligent switches. Figure 3 shows a typical "Network centric computing" System.

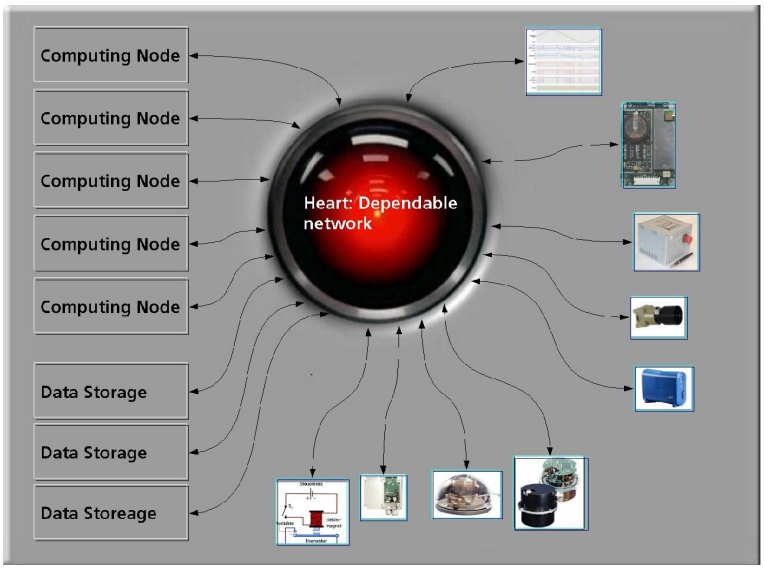

Figure 3: A typical network centric system

Any device or computer attached to the network is considered as non dependable and a failure is expected at any time. Non dependable components shall be redundant and the take over from one to a redundant module shall be as fast as possible. This is one or our primary development requirements; currently it is a few milliseconds. As long as the network stays operable the system is operable.

Computing nodes

A key parameter to speed up the recovery procedure of any component is to reduce the size of the context which have to be restored after a crash. For controllers we aim to implement cyclic control loops, in each cycle the controller gets first the sensor data and required context, then it performs some computations and then it sends the computed actions to the actuators. All context informations, which is needed for each control cycle, for example current system state or running system procedure are reduced to the minimum and distributed around all computing nodes. This context informations are distributed periodically from all nodes as broadcast in each control cycle. In this way, after a crash, a node will get all required information at the beginning of the next control cycle and at the end of the cycle it may control the system in a normal way.

Another important parameter is the boot time. For our concepts we have developed the corresponding operating system RODOS, which has a boot time of a few milliseconds. Many operating systems have to initializes IO drivers and devices before the user applications may be started. This consumes again a lot of preciously time. To avoid this, devices are not attached to the computing nodes but only to the network. Our devices support the same communication protocols like our computing nodes and are real network-devices.

RODOS operating system

RODOS (Real Time On board Dependable Operating System) is an open source building block execution platform/environment designed for space applications and for applications demanding high dependability. Simplicity is our main strategy for achieving dependability, as complexity is the cause of most development faults. The system was developed in C++, using an object-oriented framework simple enough to be understood and applied in several application domains. Although targeting minimal complexity, no fundamental functionality is missing, as its micro-kernel provides support for resource management, thread synchronisation and communication, input/output and interrupts management. The system is fully preemptive and uses priority-based scheduling and round robin for same priority threads. On the top of this kernel the RODOS middlware distributes messages locally and using gateways globally. The RODOS execution platform provides a (software) interconnection network between applications / building blocks (the middleware). A building block requires some services (incoming messages) in order to be able to provide other services (outgoing messages). The execution platform distributes such services (messages) from producer to consumers. (see Figure 4).

RODOS may be executed on the top of other operating systems or TSP (Time Space partitioning systems) or directly on the hardware in case no other operating system is running on the target hardware. In all cases the interfaces to the building blocks (or applications) remains the same, and a network of applications may be executed on different platforms and operating systems without modifications.

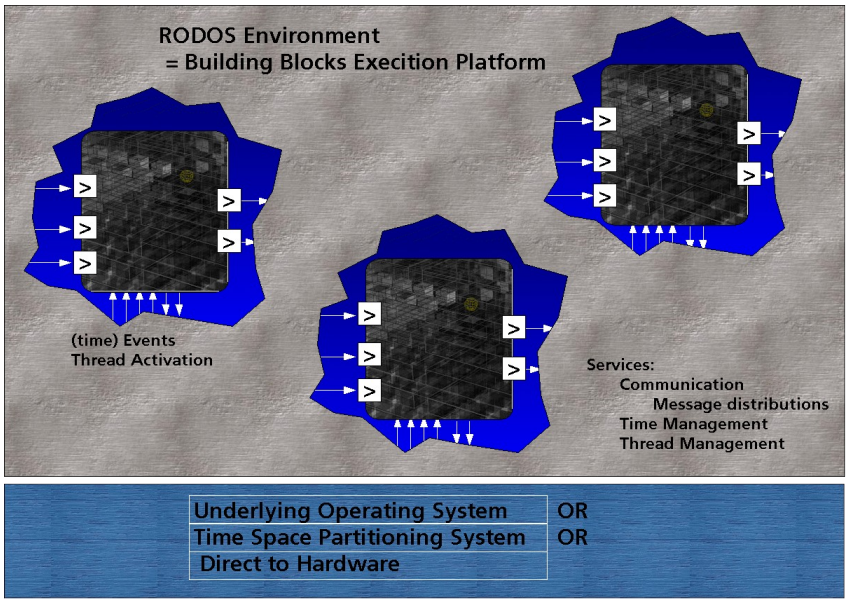

Figure 4: RODOS as Building blocks execution platform

Communication protocols

We use the same communication protocol for software applications, for the network and for the devices. Normally communication protocols store a big context information like for example link connections, connection paths, location of devices etc. Such usual protocols would imply a very long recovery time and (temporary) loss of information if a network switches crashes. To avoid this we use a connectionless communication protocol. All communications in the system are based on the publisher/subscriber protocol (in software and in hardware). Publishers make messages public under a given topic. Subscribers (zero, one or more) to a given topic get all messages which are published under this topic. To establish a transfer path, both the publisher and the subscriber must share the same topic. A topic is represented by a topic ID and a data type.

The services are distributed in the (Software/Hardware) network from producers (publishers) to consumers (subscribers). This does not depend on if the services are produced by software components or by hardware components. The same applies to the consumer of services.
The network is based on a publisher/subscriber protocol which is implemented in RODOS as a software middleware for the software tasks and in a FPGA as a middleware switch for hardware devices and to interconnect computing nodes. An ASIC implementation of the network is in work.

Network building blocks

The central component of a network centric system is the network which is the heart of the system. If it fails, the system fails. To create a dependable network we use redundant robust components, with ultra fast recovery time. The network stores no context information, each message will be routed independently of pass activities.

The building block of the network is the middleware switch, which logical view is way very similar to the software middleware (figure 5).

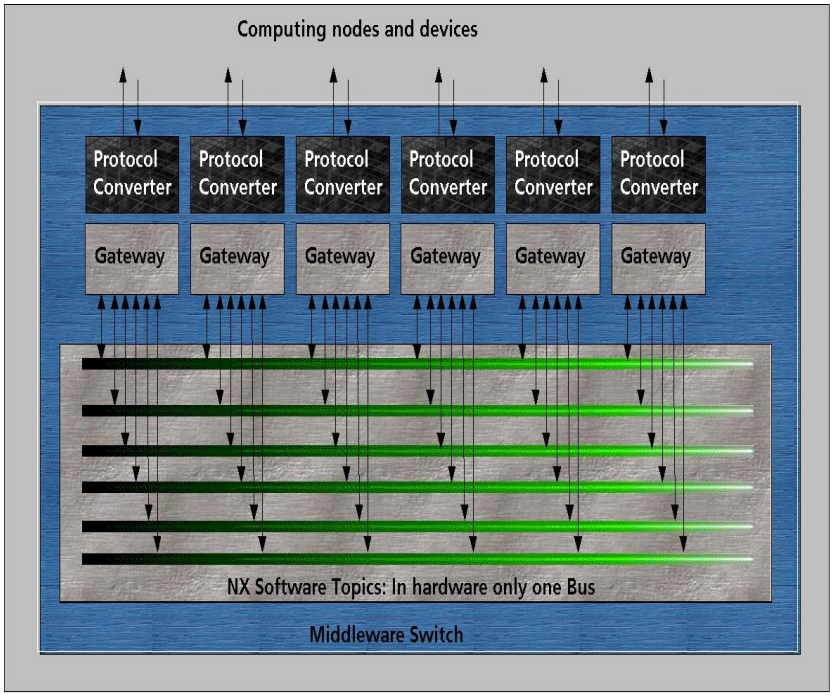

Figure 5: Logical view of the Middleware switch

The middleware switch implements (using internal software) an array of gateways connected to an array of virtual topic buses. The gateways distribute locally all incoming messages, whereas all other gateways select the messages which they will then forward, convert them to the corresponding protocol and send them using the associated link. It makes no difference what we find on the other side of the link, whether it is another network, a node computer, a device or a bus for devices.

Two or tree such switches may be connected parallel to implement redundancy. If a switch fails, the next can take over form one message to the next. By take over the interrupted message will go lost, but the applications are implemented in a way that the loss of one message can be tolerable.

References

[1] Fault-Tolerant Middleware Switch for Space Applications
Dr. Sergio Montenegro, Ebrahim Haririan;
IEEE Computer Society Washington, DC, USA
2009, SMC-IT, Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology -
ISBN:978-0-7695-3637-8

[2] Network Centric Core Avionics for Dependable Systems
Dr. Sergio Montenegro, CEAS 2009: Air and Space Confernce, Europe , October 26-29, 2009 - Manchester, UK

[3] **RODOS: Real Time Kernel Design for Dependability**
Dr. Sergio Montenegro, Frank Dannemann; **DASIA 2009 DAta Systems In Aerospace**
26 to 29 May 2009, Istanbul

[4] Let it crash: **http://blogs.teamb.com/craigstuntz/2008/05/19/37819/**

[5] Let it crash programming: http://akkasource.org/

[6] Crash-only software: More than meets the eye: **http://lwn.net/Articles/191059/**

[7] Operating system fault tolerance; Francisco Carlos Afonso; 2008, Phd Thesis, Universidade do Minho, Portugal

[8] Schoof, G., Kraemer, R., Jagdhold, U. Wolf, C., Fault-tolerant Design for Applications Exposed to Radiation, Proceedings of the Conference DASIA 2007 - Data Systems in Aerospace, 2007

[9] Crash-Only Software; George Candea and Armando Fox, *Stanford University; 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*