Deutsches Zentrum für Luft- und Raumfahrt e.V. in der Helmholtz-Gemeinschaft

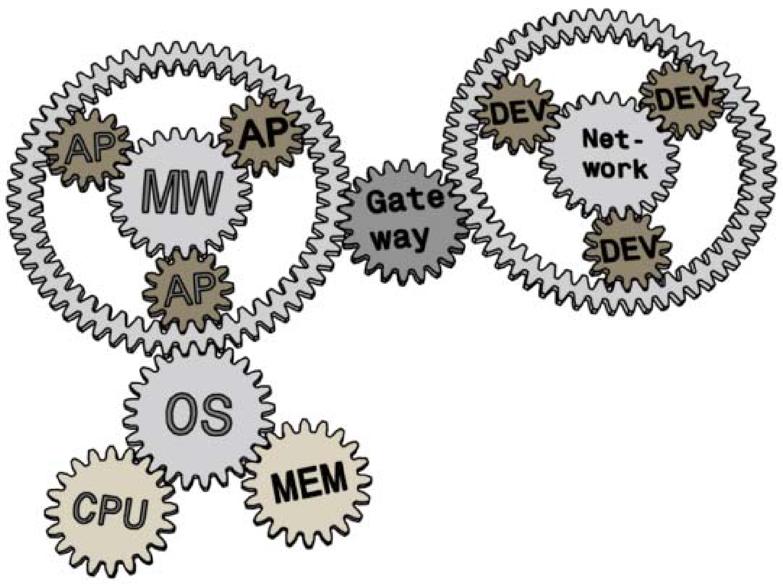
RODOS

Real Time Kernel Design for Dependability

Dr. Sergio Montenegro, Frank Dannemann
DLR, Institute of Space Systems, Robert-Hooke-Str. 7, 28359 Bremen, Germany
Web: http://www.dlr.de/irs
E-Mail: Sergio.Montenegro@DLR.de, Frank.Dannemann@DLR.de

Introduction

The NetworkCentric core avionics machine consists of several harmonised components which work together to implement dependable computing in a simple way.



The NetworkCentric Machine

Computing units (CPU +MEM) are managed by the local real-time kernel operating system (OS) RODOS. On top of the kernel runs the software middleware (MW) of RODOS and

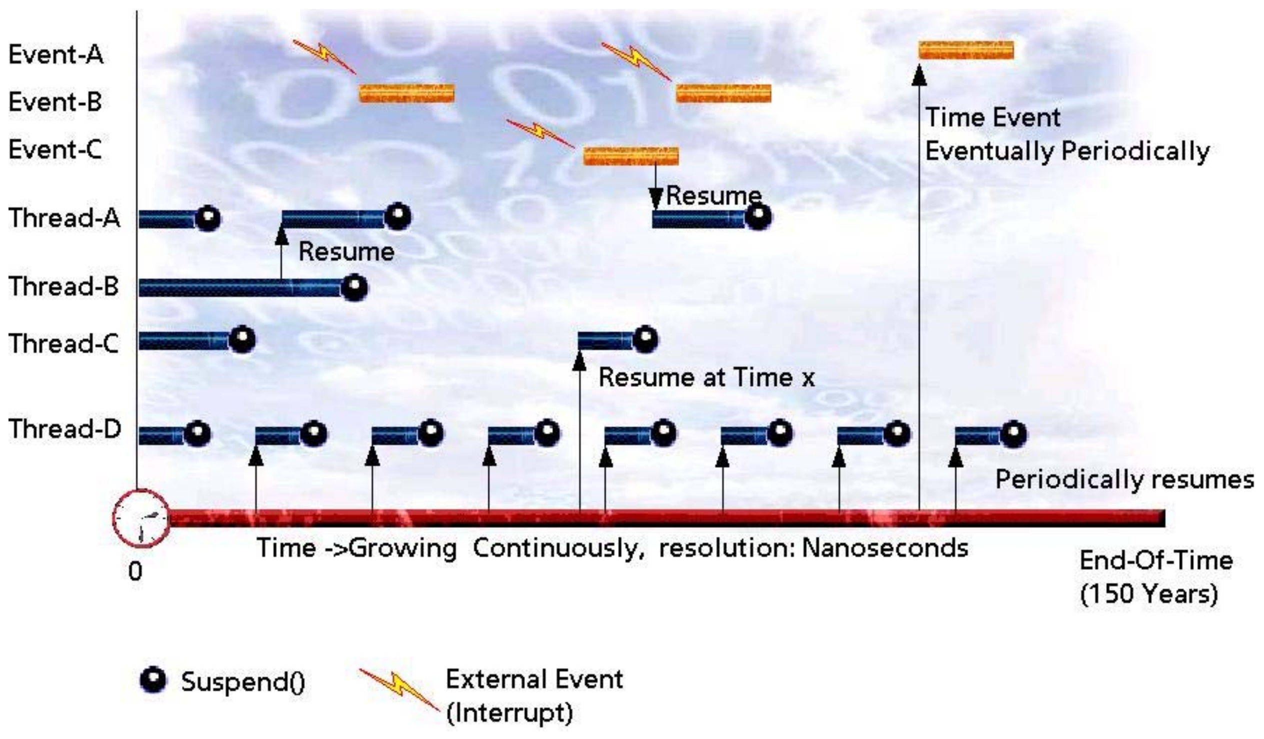
around this middleware the user can implement its applications (AP). To communicate with external units, including devices and other computing units, each node provides a gateway to the network and around the network's several devices (IO Devices and computing nodes) may be attached to the system.
RODOS is a real-time embedded operating system (OS) designed 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 microkernel provides support for resource management, thread synchronisation and communication, input/output and interrupts management. The system is fully pre-emptive and uses priority-based scheduling and round robin for same priority threads.

RODOS Core

RODOS was designed as a framework offering the following features:

- object oriented C++ interfaces,
- ultra fast booting
- real time priority controlled primitives multithreading,
- time management (as a central point),
- thread safe communication and synchronisation,
- event propagation

RODOS can be executed on embedded target hardware and on top of Linux. Applications can be moved from one host to another without modifications. The on-top-of-LINUX implementation helps developers to work locally on their workstation without having to use the target system. To move to the target, they have only to recompile the code. The behaviour is the same, except for timing requirements and time resolution, which on LINUX cannot be as exact as in the target systems.
Applications may be implemented by creating active and passive objects. Active objects may get CPU-time from the underlying core as reaction to time, to events, to message distribution and to requests from the object itself. To create an active object the user just needs to inherit and instantiate from the interface classes Thread, Event and/or instantiate Subscriber (Middleware Interface) objects.
The central element in the core is the time. The time begins at 0 (boot time) and increments continuously in nanosecond steps until "End-Of-Time" which is about 150 years into the future. This time controls almost all activities in the core.
Many threads may run (apparently) simultaneously. Each thread may run until it is suspended for a time period by itself (typically) or by other thread (not usual). To suspend a thread one may call explicitly the suspend() method, or access a synchronised object which may suspend its caller. Examples are entering a semaphore, reading from synchronised fifos, waiting for Messages, etc. Such synchronised objects just call the suspend() method of the caller, if it has to be blocked.



RODOS Thread Management

RODOS Middleware

The RODOS middleware was designed to support fault tolerance. All threads (and applications) running on top of the RODOS middleware can exchange messages asynchronously using a publisher/subscriber protocol. The RODOS middleware distributes (and replicates) messages locally in each computing node and using gateways it may cross node boundaries to reach all units in the network. Internally the middleware, gateways and hardware network (Middleware Switch) use all the same NetworkCentric protocol. Units attached to the network can be computing nodes and IO devices in the same way. The communication protocol is based on the most simple possible implementation (we were able to design) of the publisher/subscriber protocol. This gives us very high flexibility and users do not have to differentiate between local/remote communication and between any combinations of software/hardware/device communication.

Communication relationships can be very dynamic. Units may disappear or appear, tasks may be migrated, activated or deactivated at any time. The position of applications can even change (migration) at runtime, without requiring any explicit reaction of the other involved applications. There are no fixed communication paths. Each data transfer is resolved just in time using the registered communication topics.

The middleware imposes no limitations on communication paths, but the user shall use/create a meaningful, reasonable and efficient inter-task communication structure.

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 subscriber have to share the same topic. A topic is a pair of data-types and an integer representing a topic identifier. Both of the components -

software middleware and network switch (called middleware-switch) interpret the same publisher/subscriber protocol in the same way.

Applications and building blocks

To build a complex functionality and even perhaps many different functionalities in the same system, which is the typical case in satellites, it is advisable to encapsulate simple and clear cut related functions in building blocks – here called applications – and to plug such blocks together thereby building a network of applications/building blocks (not to be confused with the hardware network). Adding devices to the network we get a network of services.

Applications (or building blocks) may encapsulate threads, messages, event handlers, passive objects etc. From outside it shall not be visible to the innermost being of the application. The only interfaces to an application are messages which can be distributed/subscribed. Each application shall provide one specific service to the system.

Programming Interface

RODOS offers an integrated framework (OO) interface. Both core and middleware together will be called the RODOS-framework. The RODOS framework aims to offer the most simple and small as possible interface to user applications, which still provides all required functionality and flexibility.

The RODOS-framework includes time management, resource management and communication functionality. Without an application the framework is inactive. An application can add actions to the RODOS-framework by inheriting classes and creating active objects. These objects will be integrated automatically into the framework. In this way the framework will be extended with user functionality.

The framework technology is a further step following the object-oriented technology, where the functionality is provided by OO methods encapsulated in classes and other functionality by means of inheritance. A framework is composed of several classes in a structure with different relationships: inheritance, references and contention. The whole structure has a specific functionality. The user can adapt its functionality to his needs as follows: Some classes in the structure provide the adaptation interface (inheritance) for the user, while other classes offer just a function/method interface like the “normal” procedural interfaces.