NetworkCentric Core Avionics

RODOS

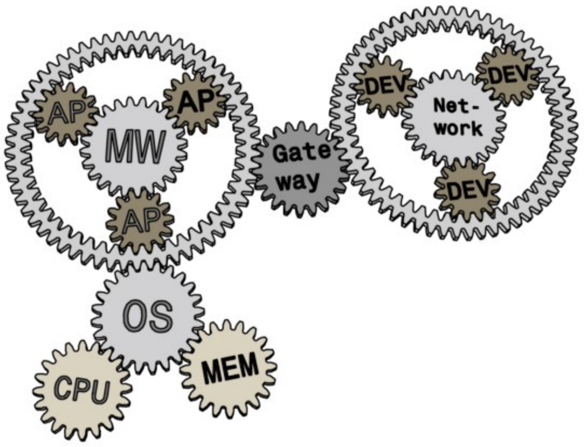
Version:	**05**
Date:	**01.11.2008**

RODOS
Real time kernel design for dependability

Sergio Montenegro
DLR-RY (Bremen)
sergio.montenegro@dlr.de

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."
-- Antoine de Saint-Exupery (1900 - 1944)

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 Devs 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 preemptive and uses priority-based scheduling and round robin for same priority threads.

RODOS provides a middleware which carries out transparent communications between applications and computing nodes. The messages exchange is asynchronous, using the publisher-subscriber protocol. Using this approach, no fixed communication paths are established and the system can be reconfigured easily at run-time. For instance, several replicas of the same software can run in different nodes and publish the result using the same topic, without knowing each other. A voter may subscribe to that topic and vote on the correct result. The core of the middleware distributes messages only locally, but using the integrated gateways to the NetworkCentric network, messages can reach any node and application in the network. The communication in the whole system includes software applications, computing nodes and IO devices.

All communications in the system are based on the publisher/subscriber protocol: 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. For this communication there is no difference in which node (computing unit or device) the publisher and subscribers are running. They may be in the same unit, or distributed around the network. They may be any combination of software tasks and hardware devices. To establish a transfer path, both the publisher and the subscriber must share the same topic. A Topic is a pair consisting of a data-type and an integer representing a topic identifier. Both the software middleware and network switch (called middleware switch), interpret the same publisher/subscriber protocol in the same way.

1. Terms and Definitions

1.1 General terms

TERM	Definition or meaning assumed in the NetworkCentric context
Criticality	System state where a failure may interrupt the global Mission; In the framework of the NetworkCentric project development the term “non-critical” corresponds to the system state where a failure of a processor will not create a mission critical risk or satellite safety risk.
Dependability	Definition: “The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers”. (From http://www.dependability.org/). Thus, “dependability” includes as special cases such attributes as reliability, availability, safety, and security.
Fault Tolerance	The ability of a system to respond gracefully to an unexpected hardware or software failure.
Module	Entities such that failure propagation among them shall be prevented; i.e. a failure affecting one module will cause, in the worst case, the loss of that module only.
Robustness	General definition: Property, which allows the system to perform in nominal mode with respect to external and internal perturbations.
Resilience	General definition: Ability of the system to re-distribute tasks on a subset of processing nodes. Definition applied in the context of the NetworkCentric applications: Capability of the system to recover lost functions (i.e. autonomous functions) after that robustness mechanism (graceful degradation, potentially down to “basic control mode”) was used. Resilience is possible when failures are software only or when the hardware failures are transient. Resilience is not possible when hardware is permanently damaged.

Graceful degradation	System mode, which happens after “robustness” mode if full resilience is not possible, i.e. if there were permanent hardware failure.
Safety	System state where it is not exceeded an acceptable level of risk with respect to: fatality, injury or occupational illness, damage to launcher hardware or launch site facilities, damage to an element of an interfacing manned flight system, the main functions of a flight system itself, pollution of the environment, atmosphere or outer space, and damage to public or private property.

1.2 Terms related to software design

If you are not an experienced programmer, then please read the following term definitions before reading the rest of this document, or else skip this chapter entirely.

TERM	Definition or meaning assumed in the NetworkCentric project
Class	The prototype for an object in an object-oriented language; analogous to a derived type in a procedural language. A class may also be considered to be a set of objects which share a common structure and behaviour. The structure of a class is determined by the class variables which represent the state of an object of that class and the behaviour is given by a set of methods associated with the class.
Super Class	(Or "superclass") The class from which another class (a "subclass") inherits, the class it is based on.
Sub Class	(Or "subclass") In object-oriented programming, a class that is derived from a base class by inheritance. The derived class contains all the features of the base class, but may have new features added or redefine existing features.
Inheritance	In object-oriented programming, the ability to derive new classes from existing classes. A derived class (or "subclass") inherits the instance variables and methods of the "base class" (or "superclass"), and may add new instance variables and methods. New methods may be defined with the same names as those in the base class, in which case they override the original one.
Override	When a subclass redefined a method of its super class
Specialise	To inherit, to create a sub class from a super class
Framework	In object-oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems.
Thread	Sometimes called “Runalbe”, is an object which may get CPU time. A Thread has own stack and context. Many threads may share the same CPU.
Method	The name given in object-oriented languages to a procedure or routine associated with one or more classes.
Application / Building Block	Own RODOS Definition: A set of Threads, Messages and passive objects to provide a user defined service, for example obstacle recognition. An Application is a (atomar) unit which can be moved as a whole from node to node.
Object	In object-oriented programming, an instance of the data structure and behaviour defined by the object's class. Each object has its own values for the instance variables of its class and can respond to the methods defined by its class.
Instance / Object from a class	An individual object of a certain class. While a class is just the type definition, an actual usage of a class is called "instance". Each instance of a class can have different values for its instance variables, i.e. its state.

Singleton	If there exactly one object for a class. Usually they shall have the same name, with the only difference: The class name begins in upper case, the object name begins with lower case. (RODOS directives)
Middleware	SW layer which facilitates communication and coordination of components in distributed systems. The NetworkCentric Middleware uses Topics to identify message distributions patterns.
Topic	A Topic is a pair: data-type and an integer representing a topic identifier. Topics are used as communication channels to distribute messages in the middleware and in the network. A messages will be published under a given topic and any subscriber to that topic will get a copy.
Service, Network of services	Services may be produced by software applications or by io devices. Services may be plugged together in a network of services distributed in different hardware units. Please do not confound a network of services and the hardware network to interconnect hardware units.

2. Introduction

RODOS is the gearwheel of the NetworkCentric core avionics machine, which controls activities in the computing units/nodes (processors, CPUs, Memory).

The RODOS real-time kernel and middleware provide an integrated Object-oriented (OO) framework interface to multitasking resources management and to the NetworkCentric communication infrastructure. The RODOS framework seeks to offer the simplest and smallest possible interface to user applications, while still providing all the required functionality and flexibility. It includes time management, CPU and memory management.

The RODOS middleware provides communication between applications, networks and all devices attached to the network. The fault tolerance support implemented in the middleware allows us to create dependable systems using unreliable components. In our concept, a hardware failure is not an exception, but a normal case, which can be expected and has to be handled. RODOS redundancy management supports different strategies to provide the highest possible dependability, our target, which is the ultra high dependability using a principle which the world has since forgotten: *Simplicity.*

Simplicity does not mean, however, lack of functionality. Real time scheduling, resource management, synchronization, middleware and simple communication and all the functions one can expect from a microkernel are implemented – just as simply as possible. An important RODOS design target is the irreducible complexity; this is the minimal possible complexity for a determined function. When it becomes no longer possible to implement it simpler without destroying the functionality.

RODOS is based on very few and simple basic functions. Applications running on top of RODOS are implemented using object-oriented technology, resulting in highly modular application software. Applications running on the top of the RODOS middleware are built using the schema of software building blocks. Several (simple) building blocks (called applications) can be distributed and interconnected in a computer+devices network using the NetworkCentric core avionics protocols, to build more complex functionality. Building blocks can be implemented and tested independently of each other. Building blocks can be interchanged without having to modify other blocks or interfaces.

3. RODOS Core

RODOS was designed as a framework offering the following features:

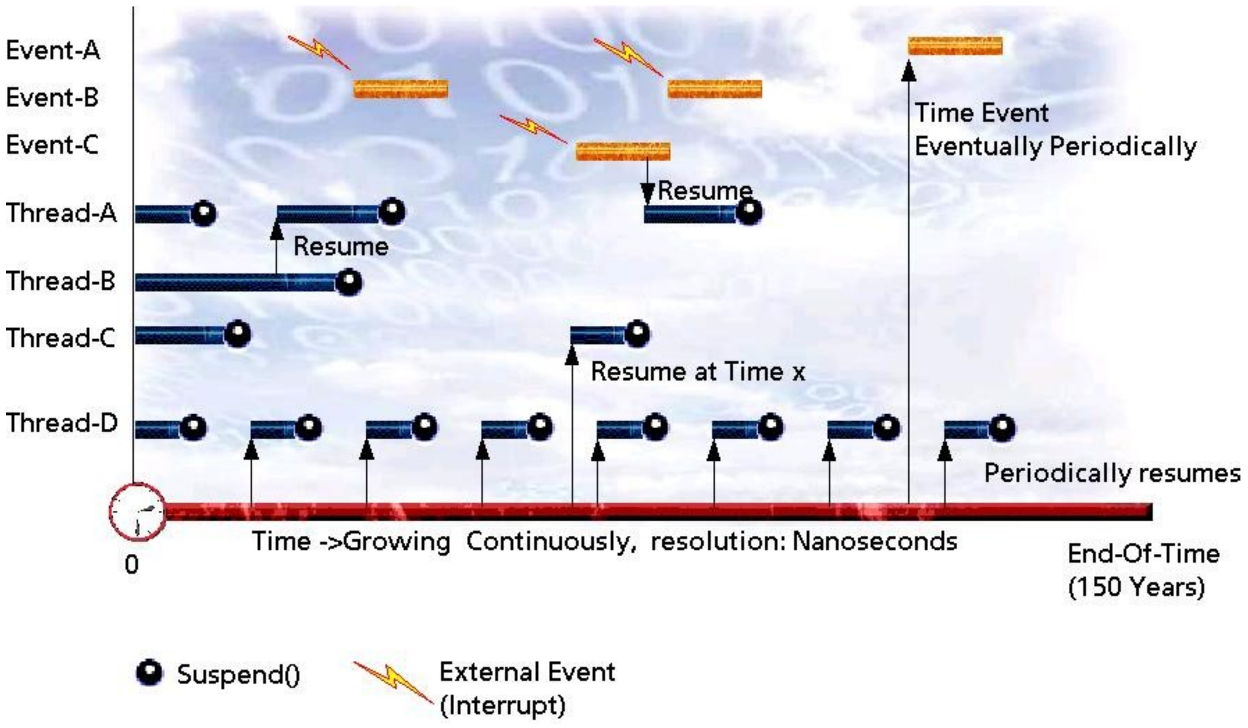
- object oriented C++ interfaces,
- Ultra fast booting
- real time priority controlled primitives multithreading,
- time management (as a central point),
- thread safe communication and synchronisation,
- Event propagation

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.



Suspending and resuming threads

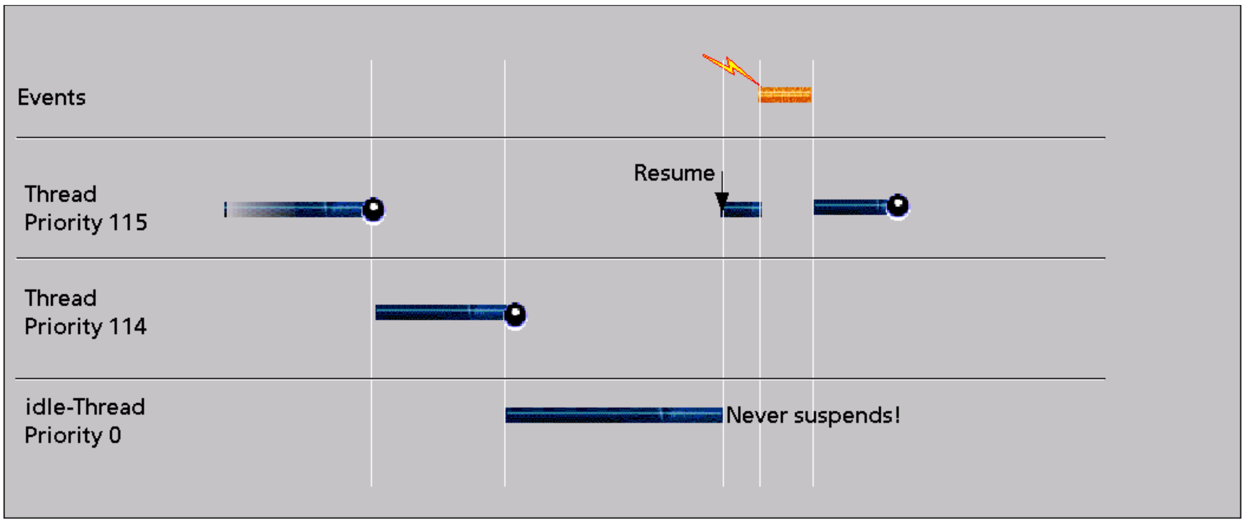
A Thread may be suspended (blocked) until a time point, then it will be resumed automatically when the corresponding time point is reached. If a Thread shall be suspended without limit, we use the time point “End-Of-Time”.

Any suspended thread may be resumed automatically at a time point or explicit at the moment by calling the resume() method. This may be done by other thread or by an event (time or interrupt server). Calling the resume() method may be explicit or may be contained in a synchronised object. For example when leaving a semaphore, the leaving thread will see if another thread is waiting to enter. If so, the corresponding resume will be called. This is implemented inside of the semaphore, the user just calls Semaphore::leave().

The core may activate event handlers too, to react to a time point or to an interrupt. Event handlers have to be very short, because they are executed in interrupt mode and while an event handler is being executed all other interrupts will be blocked. Event handlers may resume threads and manipulate data, like normal threads. Event handlers are usually used to implement IO-accesses, but in the NetworkCentric model, IO-Accesses are implemented as messages going and coming to the network where IO-Devices may be attached. Therefore Event Handlers will be scarcely used in RODOS programs.

As said before many threads and event handlers may be executed apparently simultaneously. If we have only one CPU then only one Thread or one event-handler will be executed at any time point. Whenever an event handler shall be executed any possible running thread will be shortly interrupted to execute the handler and then the interrupted thread may continue. We can say events have the highest priority in the system. When an event handler is executed the hardware interrupts are blocked. The next event handler may be executed only after the current one returns (terminates); events handlers can not be interrupted.

From all threads which are ready to run (they are not suspended at the current time point) the one with the highest priority will be executed first. If another thread with higher priority becomes ready to run the current thread will be interrupted and the one with higher priority will be executed (preemption). If a thread is suspended, then the scheduler will search for the next one with the highest priority. At the end of the list is the idle thread which is always ready to run, but has priority 0. If two threads have the same priority, a round robin procedure will be used. In one sentence: a “fair priority controlled preemptive scheduling”. Note: priorities are non-negative values from 0 to 2^31 (unsigned long int).



preemptives CPU Scheduling

4. RODOS Middleware

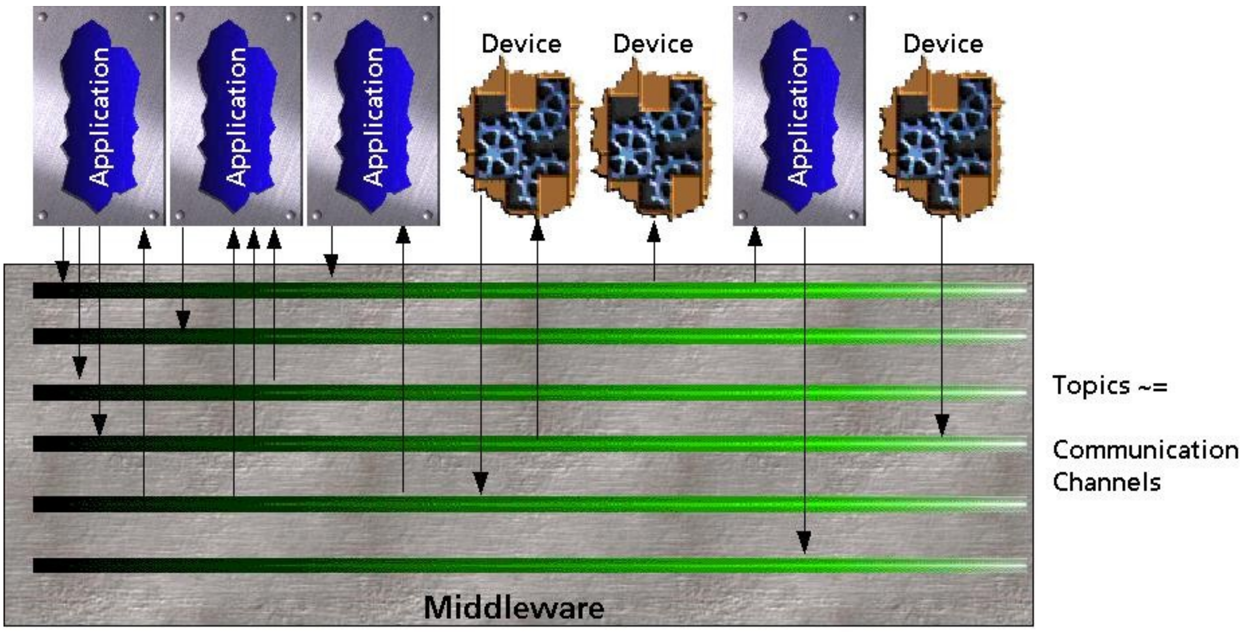
The RODOS middleware was designed to support fault tolerance. All threads (and applications) running on top of the RODOS middleware can exchange messages asynchronously using a publisher / subscriber protocol. The RODOS middleware distributes (and replicates) messages locally in each computing node and 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 combination 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.

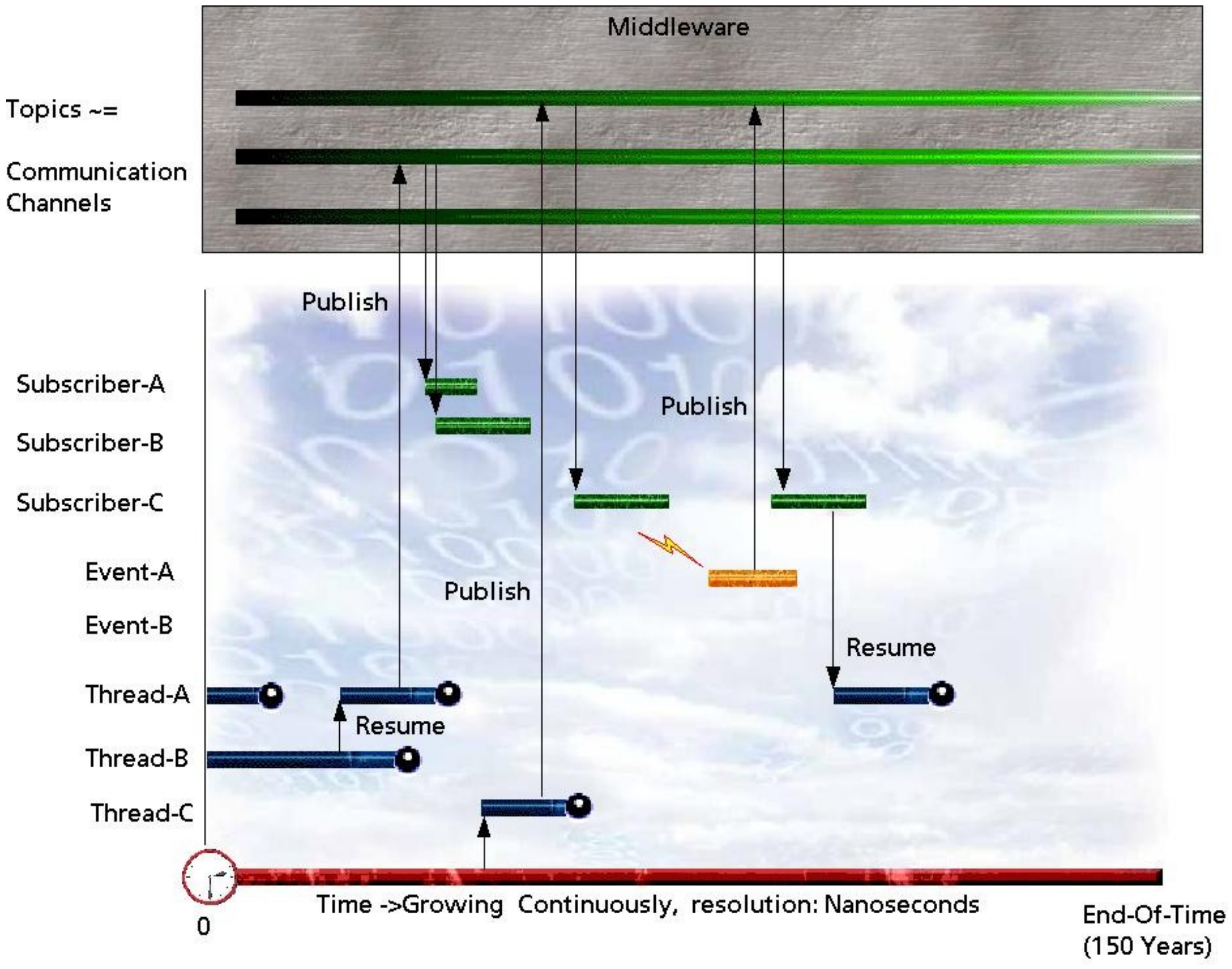


Topics in the Middleware

Software applications may access an arbitrary number of topics, both as subscriber and as publisher. Devices, on the other hand, publish or subscribe typically only one topic and, according to the NetworkCentric model, are attached preferably to the hardware network. For very small systems or in exceptional circumstances devices may be attached to a computing node (bu this is not recommended). In this case the device interface (IO) shall be encapsulated into a publisher/subscriber pair and attached to the local middleware.

Complex functionality may be implemented as a network service which is implemented into applications and which may be distributed using topics. A Topic may then be considered as a communication channel with an arbitrary number of writers and arbitrary number of readers and which may be distributed in different hardware units.

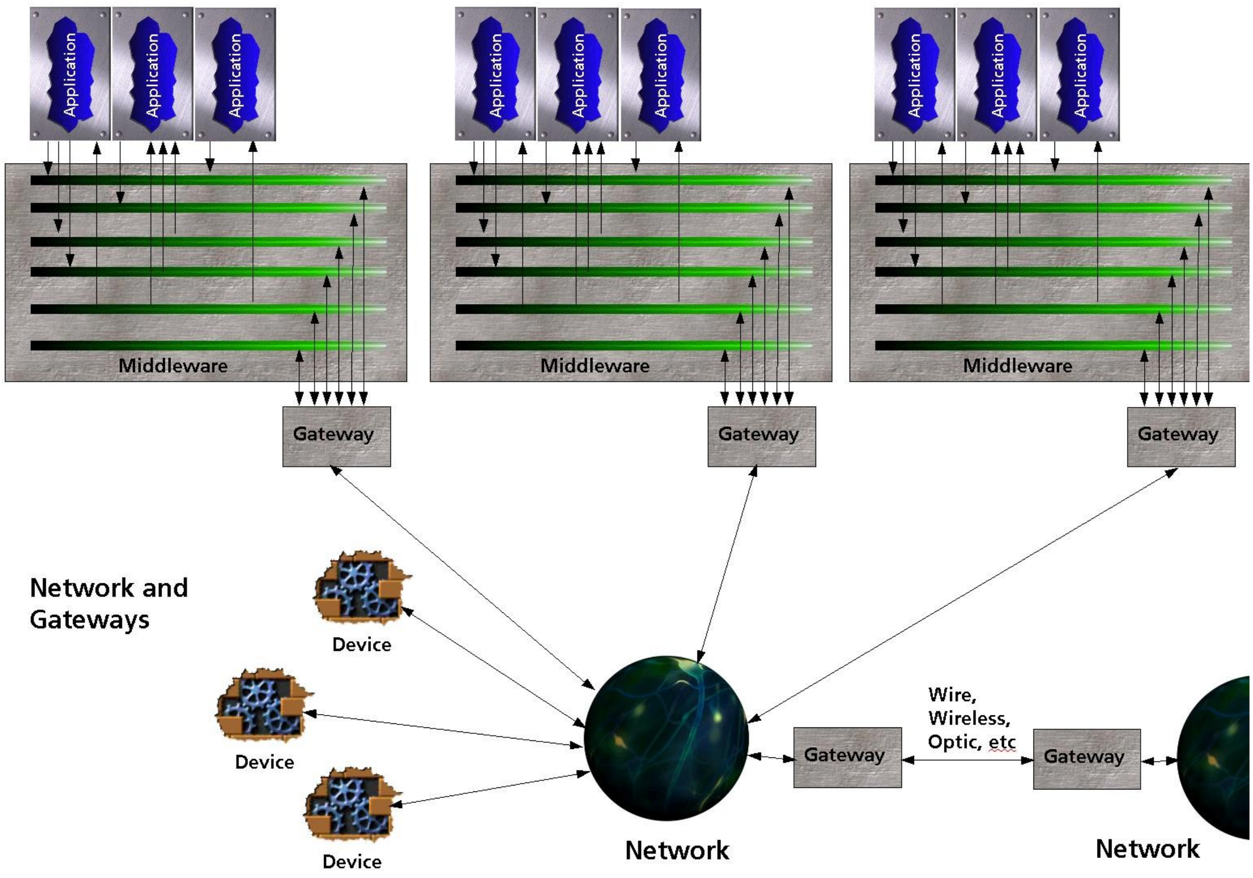
Threads and even event handlers may publish data in a non-blocking manner. For this purpose each topic provides a method Topic::publish(). As response to publish(), the topic will search its list of subscribers and notify each of them. Each subscriber provides a putter object which handles the delivery of messages. It may just register the data or it may resume a waiting thread.



Message distribution using topics

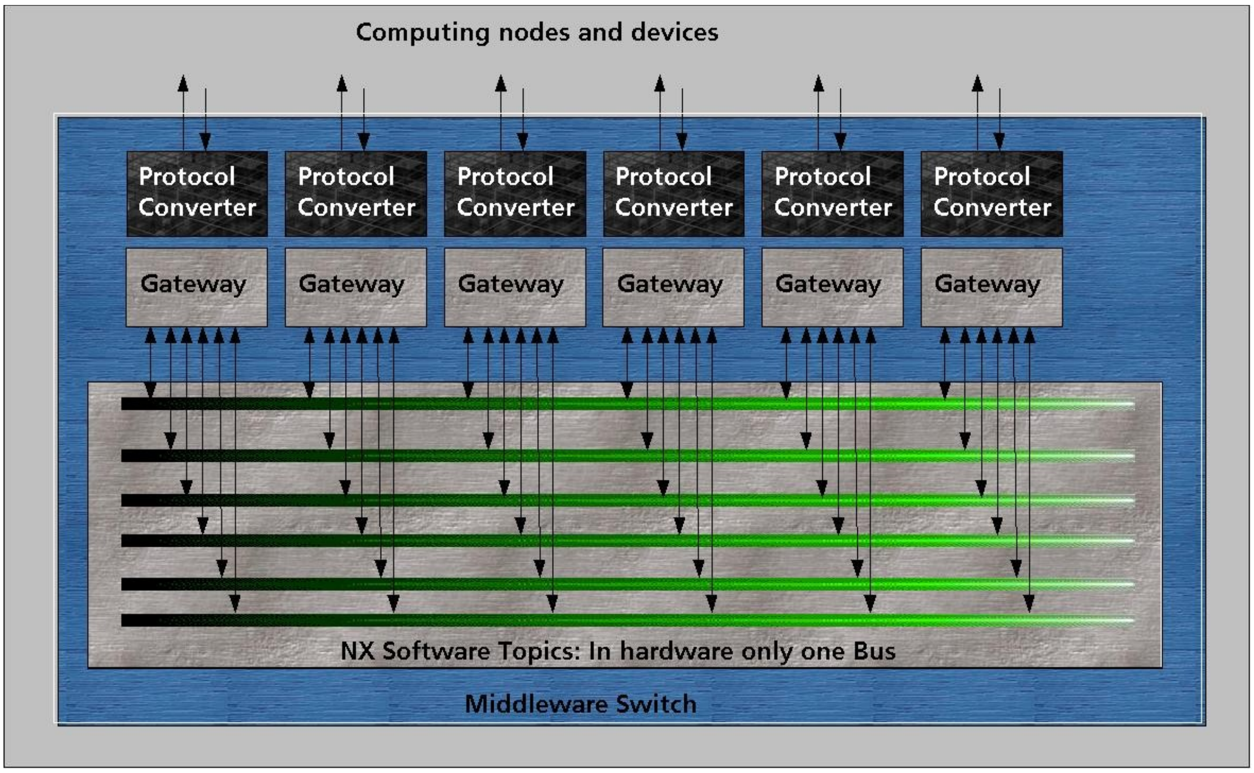
A local middleware distributes messages only locally. To access external units, computing nodes and devices one or more gateways have to be added to the local middleware. Gateways are smart publishers and subscribers which are not fixed to a single topic. They may listen to all topics and may publish messages in any topic. Gateways marshal internal middleware messages in an external format in order to send them to other hardware units. This external protocol may be, for example, a wireless protocol or Ethernet, SpaceWire, RS 322, MIL, radio communications, etc. The opposite, when the gateway gets messages from external units, it unmarshals the data and transforms it to the internal middleware format. Then using the corresponding topic it publishes the message locally.

Using this model the network of services may be distributed in several computing units, several satellites and even between satellites and ground stations. This network would allow us to substitute the typical command/telemetry procedures by a NetworkCentric network of services.



Gateways and networks

The network is implemented using intelligent middleware switches, which can interface different protocols in order to be able to access different devices. A middleware switch implements internally the same topic protocol just like the software middleware. Each port of the middleware switch implements a (different) protocol converter, like a gateway, to access external units like for example computing nodes and IO devices. In the case of IO-Devices, the protocol converter substitutes the typical IO-Driver of conventional systems.

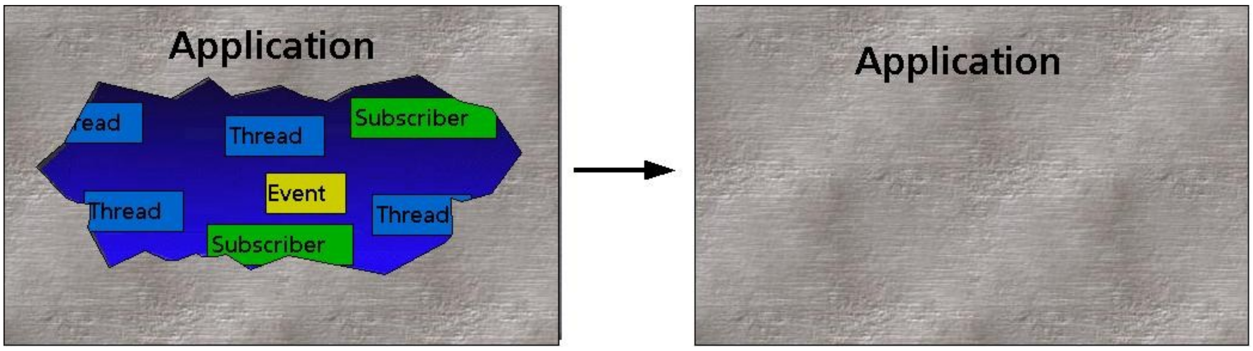


The Middleware switches (Hardware)

5. 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.

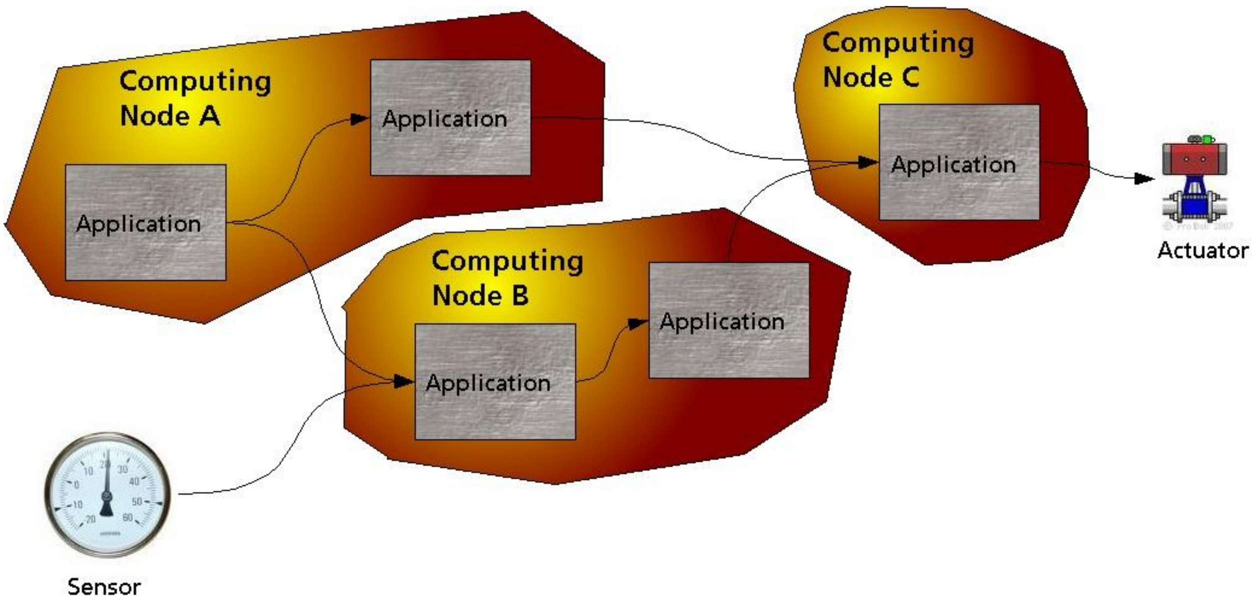
Applications (or building blocks) may encapsulate threads, messages, event handlers, passive objects etc. From outside it shall not be visible to the innermost being of the application. The only interfaces to an application are messages which can be distributed/subscribed. Each applications shall provide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



Applications encapsulate elemental components

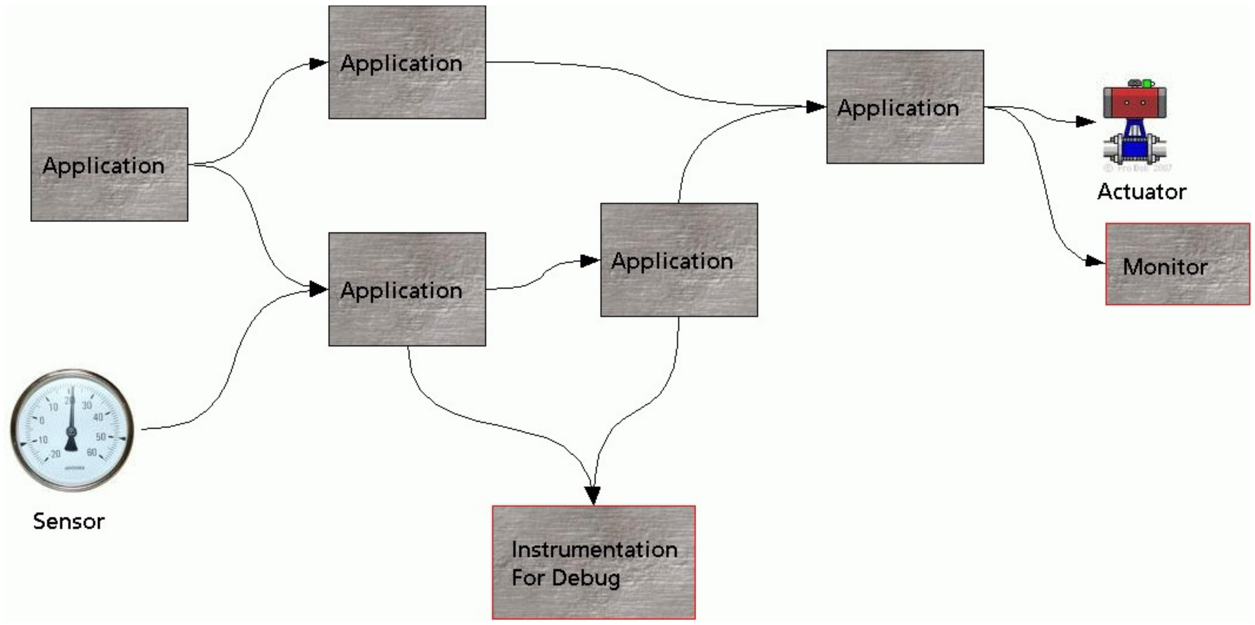
The network of services consist of applications and devices distributed in several computing units and hardware. Actuator devices do not produce information services but physical services. The opposite are the sensor devices, which build an interface from the physical world to the information world. Software applications stay (inputs and output) in the information world and the devices (sensors and actuators) are the bridge from the information world to the physical world.

There may be different system configurations for the distribution of applications in nodes and even all applications may run in the same node (typical configuration in small systems).



Example of distribution of applications

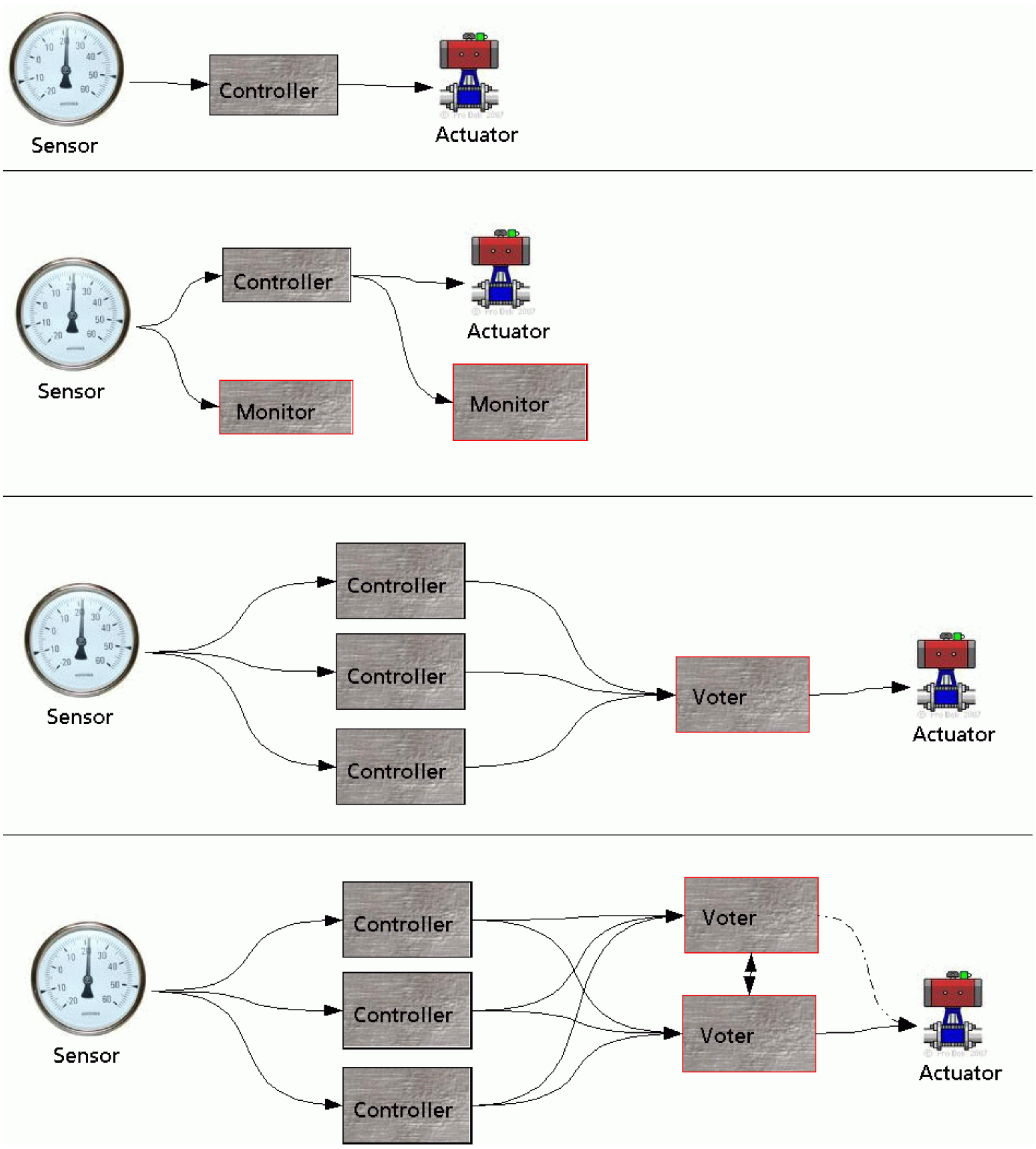
A services network may be extended (or modified) to provide more functionality without having to modify the individual building blocks. For example we may add instrumentation for debugging and a data logger by just adding new building blocks to the network.



Adding applications to the network of services

To build fault tolerance, both producers and consumers of services may be replicated. For the programmer there is no difference if one or more replicas are working in the same network. To merge outputs from several sources we need voters which may intercept messages, select the most probable correct one and forward it to the device. Even voters may be replicated.

Surely, we recommend the replication of sensors and actuators, too. But this step is not a part of the operating system and middleware description. This topic is handled in the NetworkCentric overview paper. Following some possible configurations which may be distributed in different computing nodes.



Adding fault tolerance by just plug and play

6. 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.

To adapt the functionality of the framework to his need, the user writes new classes, which inherit from the adaptation interface classes (subclasses). Some adaptation interface methods shall be overloaded with user methods and functionality in order to integrate the user functionality into the framework. The new (user) subclasses and its objects are integrated automatically (by inheritance) into the structure and the user does not need to register them manually. This makes the integration much simpler and reduces mistake sources.

A software system consists of a collection of passive and active objects. The passive objects just offers data and methods to be accessed by active objects. Some of these methods can, however, provide means for synchronisation for threads.

Active objects will be activated (executed) by the underlying system. They may get the CPU (time) as a reaction to time-points, events or requests. The most common example of active objects are the threads. Many threads may run (apparently) concurrently.

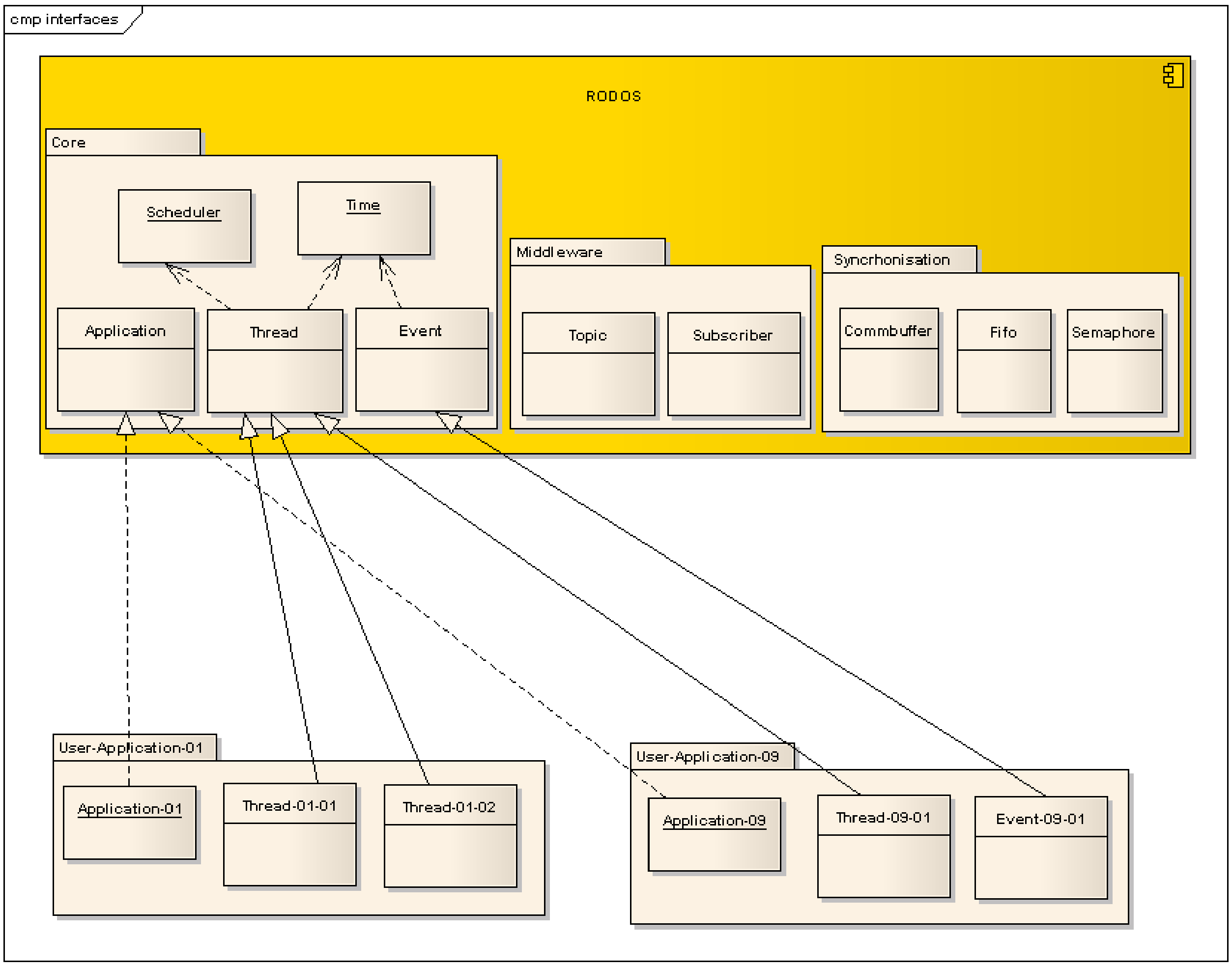
Time is a very central point for the RODOS framework. Execution of threads is mainly controlled by time. Each thread can define periodic execution or just time points when it shall be executed. The RODOS framework will try to satisfy all these time requirements. If more than one thread requests the CPU for the same time, then the conflict will be resolved according to priorities. The thread with the higher priority will be executed next.

There may be exclusive regions, where maximum one thread may execute a piece of code, or situations when a thread waits for a signal or data from another thread. RODOS provides means for such thread synchronisations, using semaphores, and synchronous data FIFOS.

Another class of active objects are the events. The user can define zero or more eventhandlers to react to events. The most popular example of events are time events, which produce a reaction when a time point is reached. Another example is the reaction to external events (interrupts). User eventhandlers shall provide a handle function which will be called when the corresponding event arrives.

And the last class of active objects are the middleware subscribers. A middleware subscriber may wait for a determined type of message (topic). When an expected message arrives, the corresponding subscriber will be activated.

The following picture is an example of user applications which extend threads and events to add functionality to the framework. The same can be done to subscribers. Passive objects like objects from the synchronisation classes may be instantiated and used without inheritance.



Framework interface to RODOS

For examples and explanations please refer to the RODOS tutorials: core, Middleware, Middleware_distributed, primefinder and support_programs. Read the corresponding readme.pdf files.