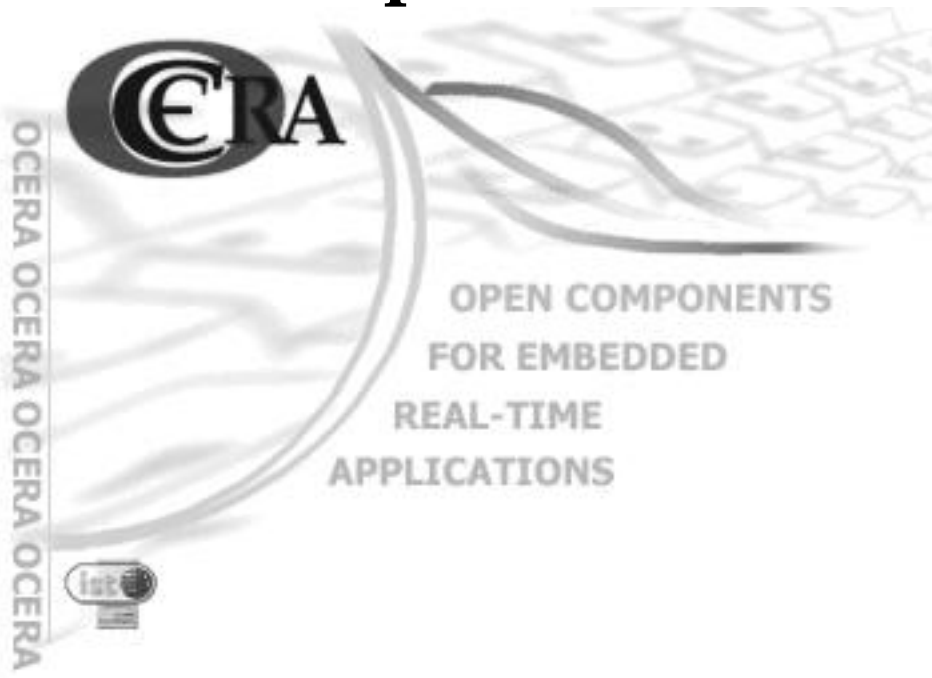
WP7 - Communication Components



Deliverable D7.1 - Design of Communication Components

WP7 - Communication Components : Deliverable D7.1 - Design of Communication Components

by Jan Krakora, Pavel Pisa, Frantisek Vacek, Zdenek Sebek, Petr Smolik, and Zdenek Hanzalek

Published November 2002

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Chapter 1. Design of Communication Components

1.1. The OCERA Real-Time Ethernet (ORTE)

1.1.1. ORTE design

The Ocera Real-Time Ethernet (ORTE) will be open source implementation of RTPS communication protocol. This protocol is being to submit to IETF as an informational RFC and has been adopted by the IDA group. Figure 1-1 shows the network stack layers. Non Real-Time applications, which are using standard protocols such as HTTP, FTP, DCOM etc., are running on top of standard TCP or UDP stack. ORTE is new application layer protocol, which is build on top of standard UDP stack. Since there are many TCP/IP stack implementations under many operating systems and ORTE protocol does not have any other special HW/SW requirements, it should be easily ported to many HW/SW target platforms. It doesn't use or require TCP, so it retains control of timing and reliability.

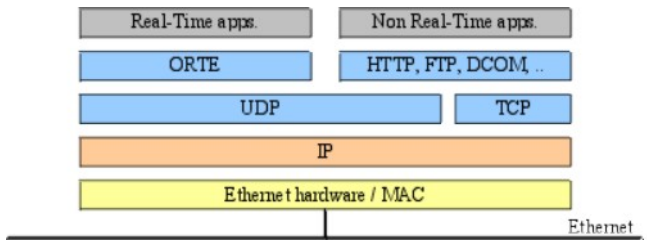


Figure 1-1. ORTE layers

The ORTE is composed from three main components, as shown on Figure 1-2:

Database: stores parameters describing both local as well as remote node's objects
Processes: perform message processing, serialization/deserialization and communication between objects
API: application interface

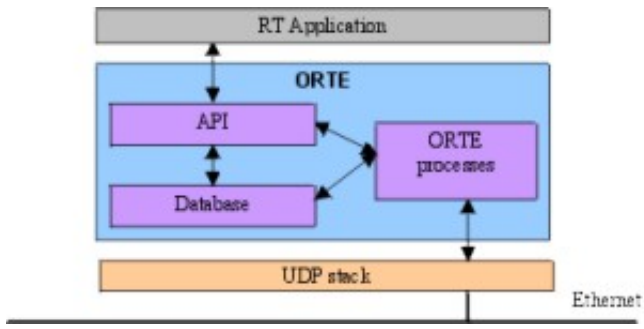


Figure 1-2. ORTE structure

The RTPS protocol is implemented as a set of objects. Objects are of the following types:

- Manager (M)

- ManagedApplication (MA)
- Writers (Publication, CSTWriter)
- Readers (Subscription, CSTReader)

A Manager is a special object that facilitates the automatic discovery of other Managers. There is one Manager on each participating network node. A ManagedApplication is an applciation that is managed by one or more Managers. The Publication is used to publish issues to matching Subscription. The CSTWriter and CSTReader are the equivalent of the Publication and Subscription, respectively, but are used solely for the state-synchronization protocol. Each object on network is characterized by GUID (Globally Unique Id).

The RTPS protocol uses five logical messages:

ISSUE: Contains the application's user data. ISSUES are sent by Publications to one or more Subscriptions.
VAR: Contain information about attributes of state of objects.
HEARTBEAT (HB): Describes the information which is available in a writer.
GAP: Describes information which is no more relevant to readers.
ACK: Provides information on the state of a reader to a writer.

Each of these logical messages are sent between specific readers and writers as follows:

Publication to subscription(s): ISSUE, HEARTBEAT
Subscription to publication: ACK
CSTWriter to a CSTReader: VAR, GAP, HEARTBEAT
CSTReader to a CSTWriter: ACK

1.1.2. Database

Status of all objects is stored in ORTE database. The database is set data structures containing parameters of all known objects. There are not only data describing local node's objects, but also description of all known remote nodes' objects. Since access to this database should be granted to both ORTE stack processes as well as to applications, there should be implemented some access control algorythm allowing concurrent access. Applications use an API call to access the database. The database will be implemented as a set of generic sorted arrays.

1.1.3. Manager

The Manager is an independent process, which is created during application startup. It is a special Application that helps applications automatically discover each other on the Network. Every Manager keeps track of its managees and their attributes. To provide this information on the Network, every Manager has the special CSTWriter writerApplications. The Composite State (CS) that the CSTWriter writerApplications provides are the attributes of all the ManagedApplications that the Manager manages (its managees). Whenever the Manager accepts a new ManagedApplication as its managee, whenever the Manager loses a ManagedApplication as a managee or whenever an attribute of a managee changes, the CS of the writerApplications changes. Each such change creates new instance of CSChange which has to be transferred to all network objects (Managers and ManagedApplications) by means of CST protocol.

1.1.3.1. Special objects in a manager

The manager is composed from five kinds of objects:

WriterApplicationSelf: CSTWriter throught which the Manager provides information about its own parameters to Managers on other nodes.
ReaderManagers: CSTReader through which the Manager obtains information on the state of all other Managers on the Network.
WriterManagers: CSTWriter throught which the Manager will send the state of all Managers in the Network to all its managees.
ReaderApplications: CSTReader which is used for the registration of local and remote managedApplications.
WriterApplications: CSTWriter throught which the Manager will send information about its managees to other Managers in the Network.

Each Writer communicates with a Reader on remote node. There is one local instance of RemoteReader object for each such remote Reader. Similarly there is one local instance of RemoteWriter object for each Writer object on remote node. These two types of objects represent status of remote Readers and Writers on different network nodes. These objects are not network objects, they do not have any GUID and it is not possible to communicate with them. Their purpose is solely to provide information on status of remote Readers and Writers.

1.1.3.2. Implementation

The Manager will be implemented as two threads - reader thread and writer thread. During its startup it opens two sockets, one for each thread, reads information about other managers in the network (list of their IP addresses) and initialize database. One a Manager is created, it starts to announce periodically its presence to other managers in the network.

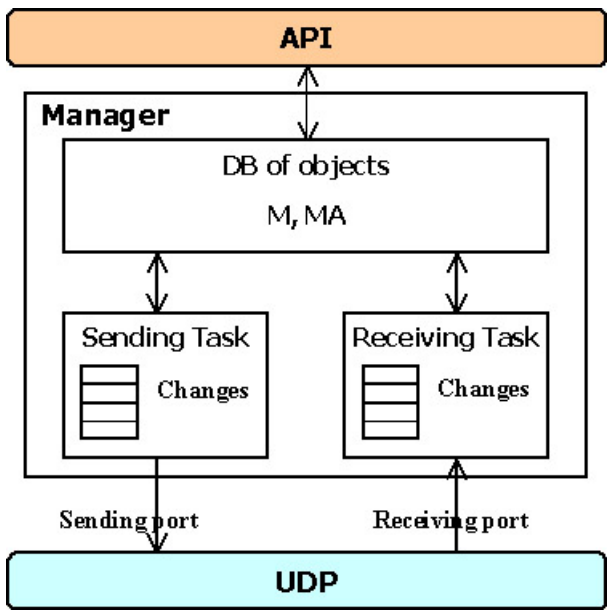


Figure 1-3. ORTE manager

Every change in the CS of the CSTWriter creates a new CSChange with a new SequenceNumber. The objectGUID of the new CSChange is the GUID of the NetworkObject that the change in the CS applies to. The attributes of that NetworkObject are represented as a ParameterSequence in the CSChange. The alive boolean is set to FALSE if the CSChange represents the removal of the NetworkObject from the set of objects in the CS. The CSTChangeForReader keeps track of the communication state (attribute

cS) and relevance (attribute relevant) of each CSChange with respect to a specific remote CSTReader. This relevant boolean is set to TRUE when the CSChangeForReader is created; it can be set to FALSE when the CSChange has become irrelevant for the remote Reader because of later CSChanges. This can happen, for example, when an attribute of a NetworkObject changes several times. In that case a later CSChange can make a previous CSChange irrelevant because a Reader is only interested in the latest attributes of the NetworkObject. It is the responsibility of the remote CSTReader to use this argument correctly so that the CSTReader can reconstruct the correct CS from the relevant CSChanges it receives. Figure 1-4 shows the Finite State Machine representing the state of the attribute cS of the CSChangeForReader. The states have the following meanings:

New: a CSChange with SequenceNumber sn is available in the CSTWriter but this has not been announced or sent to the CSTReader yet.
Announced: the existence of this SequenceNumber has been announced.
ToSend: it is time to send either a VAR or GAP with this sn to the CSTReader.
Underway: the CSChange has been sent but the Writer will ignore new requests for this CSChange.
Unacknowledged: the CSChange should have been received by the CSTReader, but this has not been acknowledged by the CSTReader. As the message could have been lost, the CSTReader can request the CSChange to be sent again.
Acknowledged: the CSTWriter knows that the CSTReader has received the CSChange with SequenceNumber sn.

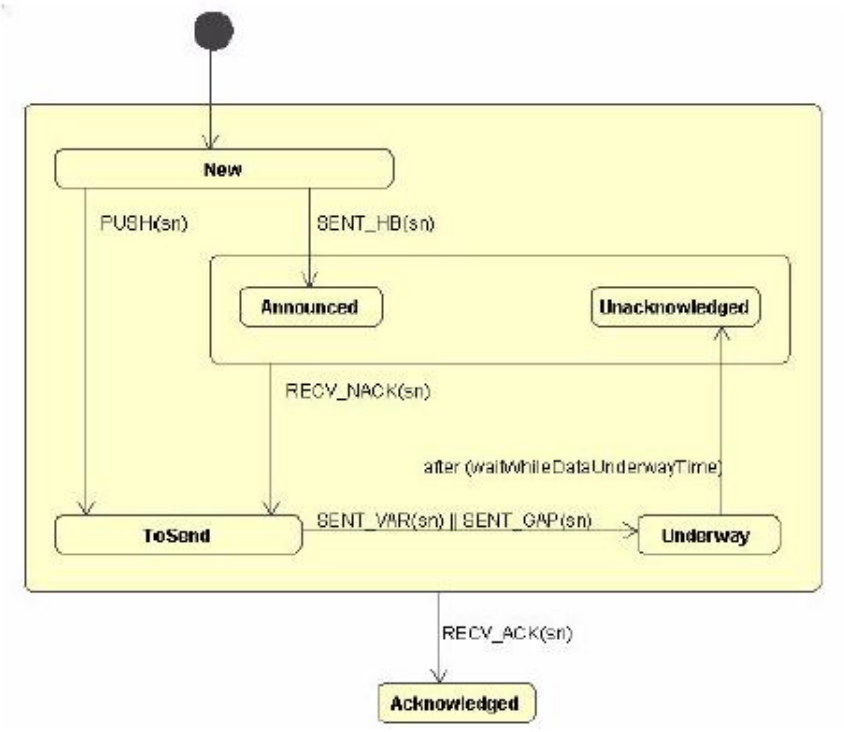


Figure 1-4. CSTWriter behavior

The reader thread is responsible for receiving of incomming CSTProtocol messages which are addressed to this manager. The CSTReader receives CSChangeFromWriters from the CSTWriter. In case a VAR was received for the CSChangeFromWriters, the CSTReader will store the contents of the VAR in an associated CSChange. The CSTReader should be able to reconstruct the current CS of a specific CSTWriter by interpreting all consecutive CSChanges. The Finite State Machine representing this process is shown of Figure 1-5.

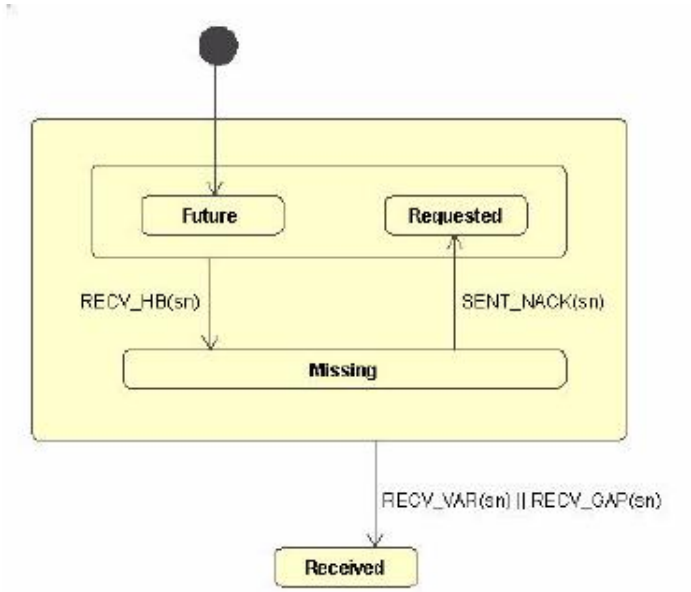


Figure 1-5. CSTReader behavior

Future: A CSChange with SequenceNumber sn may not be used yet by the CSTWriter
Missing: The sn is available in the CSTWriter and is needed to reconstruct the CS.
Requested: The sn was requested from the CSTWriter, a response might be pending or underway
Received: The sn was received: as a VAR if the sn is relevant to reconstruct the CS or as a GAP if the sn is irrelevant.

Each change in CS of an network object which is described in succesfully received CSChange is written into the database.

1.1.4. ManagedApplication

A ManagedApplication is an Application that is managed by one or more Managers. Every ManagedApplication is managed by at least one Manager. TheManagedApplication has a special CSTWriter writerApplicationSelf. The Composite State of the ManagedApplication::writerApplicationSelf contains only one NetworkObject - the application itself. The writerApplicationSelf of the ManagedApplication must be configured to announce its presence repeatedly and does not request nor expect acknowledgements. A Manager that discovers a new ManagedApplication through its readerApplications must decide whether it must manage this ManagedApplication or not. For this purpose, the attribute managerKeyList of the Application is used. If one of the ManagedApplication's keys (in the attribute managerKeyList) is equal to one of the Manager's keys, the Manager accepts the Application as a managee. If none of the keys are equal, the managed application is ignored. At the end of this process all Managers have discovered their managees and the ManagedApplications know all Managers in the Network.

The ManagedApplications now use the CST Protocol between the writerApplications of the Managers and the readerApplications of the ManagedApplications in order to discover other ManagedApplications in the Network. Every ManagedApplication has two special CSTWriters, writerPublications and writerSubscriptions, and two special CSTReaders, readerPublications and readerSubscriptions. Once ManagedApplications have discovered each other, they use the standard CST protocol through these special CSTReaders and CSTWriter to transfer the attributes of all Publications and Subscriptions in the Network.

1.1.4.1. Special object in a managedApplication

The managedApplication is composed from seven kinds of objects.

WriterApplicationSelf: a CSTWriter throught which the ManagedApplication registers itself with the local Manager.
ReaderApplications: a CSTReader throught which the ManagedApplication receives information about another ManagedApplications in the network.
ReaderManagers: a CSTReader throught which the ManagedApplication receives information about Managers.
WriterPublications: a Writer that provides issues to one or more instances of a Subscription using the publish-subscribe protocol and semantics.
ReaderPublications: a Reader throught which the Publication receives information about Subscriptions.
WriterSubscriptions: a Writer that provides information about Subscription to Publications.
ReaderSubscriptions: a Reader that receives issues from one or more instances of Publication, using the publish-subscribe service.

1.1.4.2. Implementation

Implementation of ManagedApplication is similar to implementation of the Manager. There will be one sending thread and one receiving thread. Each thread uses its own UDP port for communication. Algorithm of exchange of CSChanges is the same as in Manager. Aditionaly these two threads will be involved in exchange of application data by means of Public-Subscribe protocol.

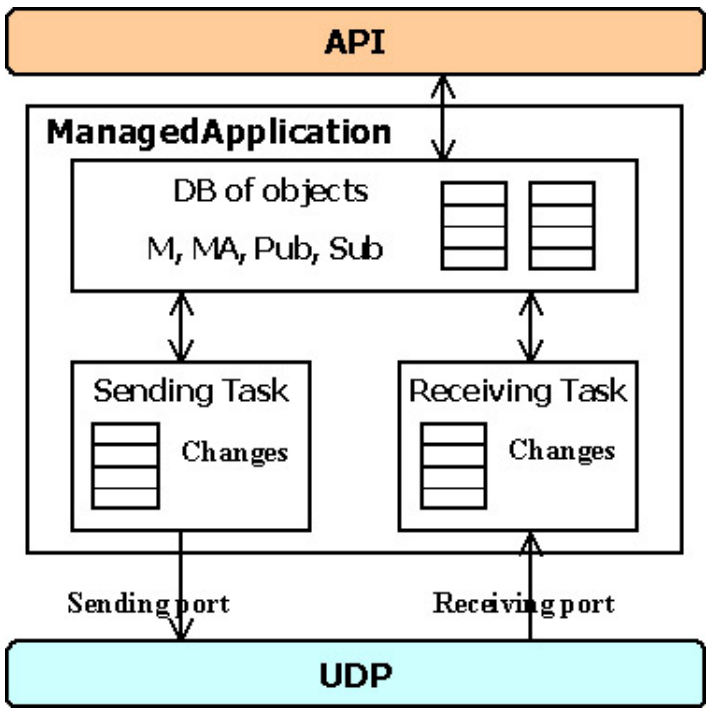


Figure 1-6. ORTE ManagedApplication

1.1.5. API

First implementation will be designed like one application running in user space using standard Linux 2.4 kernel. Since the main purpose of this version will be to test this

implementation against another commercially available implementation, there will be no any standard API provided. The real-time capabilities will not be focused during this phase too. The next version will be written as a Linux kernel module. Interface between this module and an application will use standard `ioctl` function and read/write operation. The function will be divided into three categories:

- Administration - create and destroy database (InitDB, DestDB, SetParamDB, GetParamDB,...)
- Publish - create and destroy publishers, sending data (CreatePub, DestroyPub, SndData,...)
- Subscribe - create and destroy subscribers, receiving data (CreateSub, DestroySub, RecDataPoll, @RecDataCallBack, ...)

There are two type of subscribers - polling or callback. Polling subscriberes have to ask ORTE layer by `RecDataPoll` API call whether new data are available. Callback subscriber creates an callback function, which will be passed to ORTE stack. ORTE stack will call this function every time when new data will be available.

1.2. The OCERA CAN component

1.2.1. Virtual CAN API (VCA)

The virtual CAN API is the interface used to connect the application threads either with the CAN hardware card or with other software layers substituting CAN bus. The application thread can live either in the Hard RT space or in the Soft RT space. In the the words we can say that VCA is a layer between the CAN driver and the application threads.

1.2.1.1. CAN driver

CAN driver in the OCERA project can be realized in two ways, as a standard kernel module or as a RT-Linux thread. The first approach is assigned for the soft realtime applications and the second one for the hard realtime ones. In both cases the driver is accessed via VCA using the same function calls.

The hard realtime CAN driver is the high priority RT-Linux thread which communicates with a CAN card and puts/gets CAN messages to the VCA queues. The soft realtime CAN driver is a standard kernel device driver accessed via /dev/can. If we want to use VCA in the soft realtime, we should use libvca.

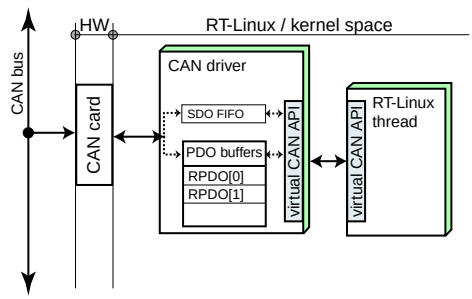


Figure 1-7. Hard real time CAN driver usage

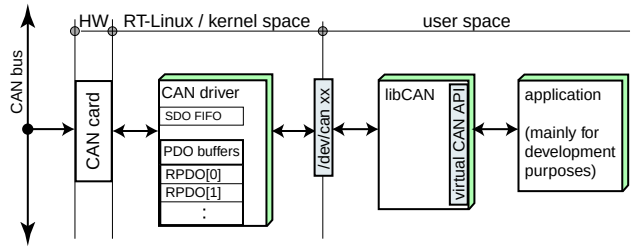


Figure 1-8. Soft real time CAN driver usage

1.2.1.2. VCA functions overview

*int vca_open_handle(vca_handle_t *vcah_p, const char *dev_name, const char *options, int flags)*

Function returns handle to opened CAN connection. Connection can be opened in read/write mode (options) and as blocking/nonblocking for reading (flags). Parameter *dev_name* is used only in Soft real time mode.

int vca_close_handle(vca_handle_t vcah)

Function closes formerly opened connection identified by handle *vcah* and frees system resources.

*int vca_send_msg_seq(vca_handle_t vcah, canmsg_t *messages, int count)*

Function sends a *count* of CAN messages to CAN bus.

*int vca_rec_msg_seq(vca_handle_t vcah, canmsg_t *messages, int count)*

Function provides a blocking read of CAN messages from CAN bus. Returns number of red messages.

int vca_wait(vca_handle_t vcah, int wait_msec, int what)

Function suspends a calling thread for *wait_msec*. Suspension can be awakened by occurrence of CAN message. The details of the awakening are specified by parameter *what*

1.2.2. CAN/CANopen monitor

1.2.2.1. Overview

CAN/CANopen monitor is a GUI tool written in Java designated for listening on the CAN bus and showing caught CAN messages.

1.2.2.2. Basic functionality

CAN/CANopen monitor can read/send the CAN messages from/to CAN bus. If some CANopen device EDS (Electronic Data Sheet) is loaded, the functionality of tool is extended. The CAN messages are translated according to loaded EDS and received SDO are shown in the EDS tree. You can also send SDO by editing field in EDS tree.

Necessary part of CAN monitor is the *canmond*. *Canmond* is a server connected to VCA, which sends received messages to all clients (ie. CAN monitor) connected to it through TCP socket. When some client send CAN message to the canmond, it resents it to CAN

bus. TCP connection allows clients to be placed wherever on Internet. One can also read/send CAN messages using a java applet on this HTML browser.

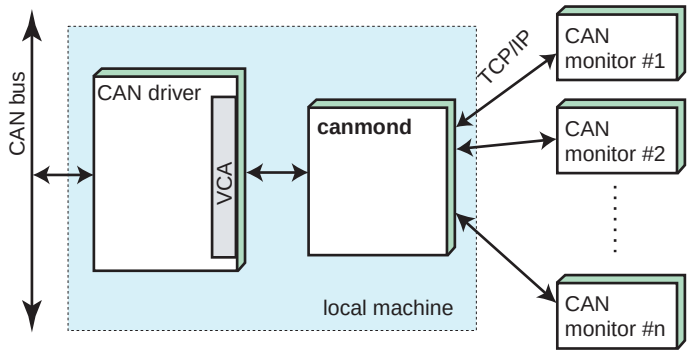


Figure 1-9. canmond server-client architecture

CAN monitor can serve as application showing all messages on CAN bus. You can also send a raw CAN messages to the CAN bus.

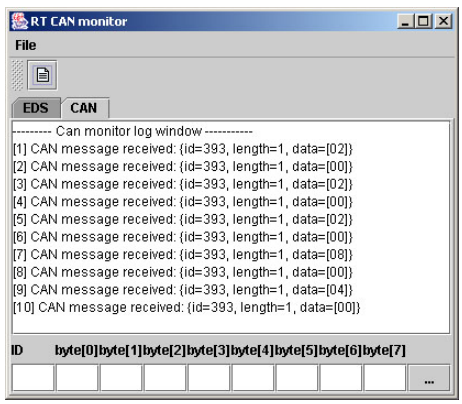


Figure 1-10. CAN monitor CAN messages window

With loaded EDS you can write/read values straight to the device object dictionary (OD).

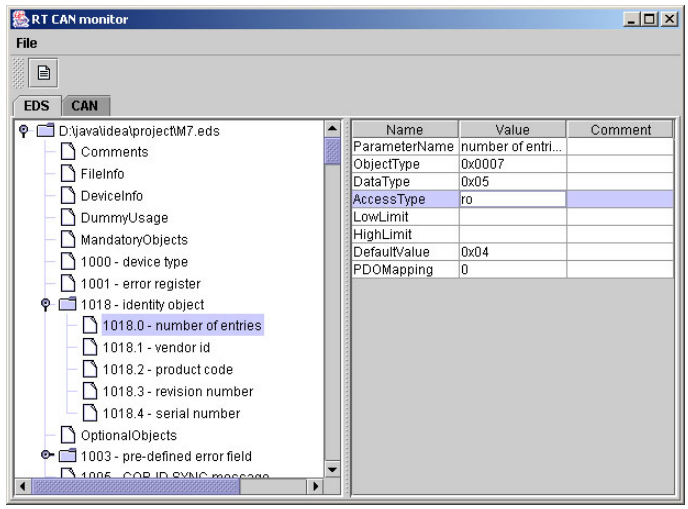


Figure 1-11. The Object Dictionary tree view

1.2.3. RT-CANopen device

1.2.3.1. Overview

RT-CANopen device (only *device* or *CANopen device* will be used instead in rest of this section) is the software solution based on the hardware, RT-CANopen FSM (Final State Machine) threads, EDS (Electronic Data Sheet) file and HDS (Handler Definition Sheet) file. Device can be configured to work as a CANopen master or CANopen slave.

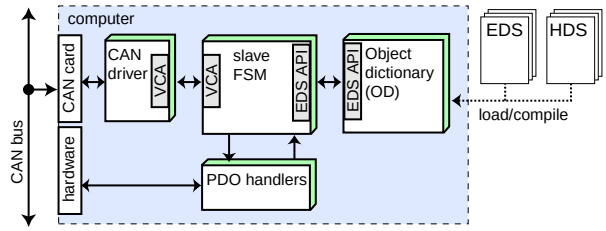


Figure 1-12. RT-CANopen device architecture

CANopen device components description

FSM: FSM (Final State Machine) means set of RT-Linux threads providing PDO and SDO communication via VCA VCA. Slave FSM also calls appropriate handler PDO communication and looks into the slave's object dictionary in case of SDO request.
PDO handlers: User written module containing handlers for reading/writing PDO mapped object data from/to hardware.
EDS: EDS means the *Electronic Data Sheet*, text file describing all objects in the slave object dictionary and its mapping tho the PDO. EDS is parsed in order to create the slave OD.
HDS: HDS (Handler Definition Sheet) means the *Handler Definition Sheet*, text file describing the linking PDO's COB-ID with required handler in order to grant correspondence between the CANopen object value and technological process data from the hardware. For example a thermometer with the analog output connected to PC A/D converter card needs handler which reads temperature from the card output port and gives it to the FSM. The slave designer have to write this handler code while the FSM source code remains always the same, OCERA written.

1.2.3.2. RT-CANopen device in the Soft real time space

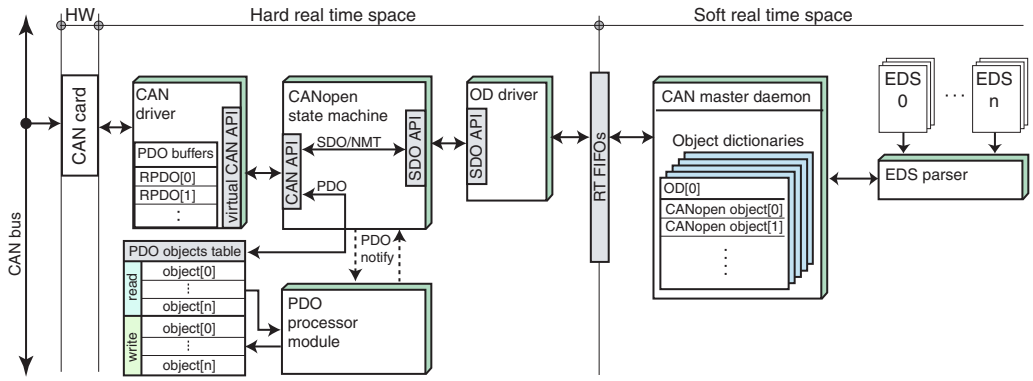


Figure 1-13. Soft real time CANopen device architecture

As can be seen on figure above CAN driver sends the CAN messages to CANopen FSM via VCA. FSM handles messages of two main categories, process data (PDO) and service data (SDO, NMT, SFO).

The process data (PDO objects) are handled separately of the SDO. Slave FSM exploits CAN driver message buffers as the buffers for the slave PDOs. This approach is necessary because some CAN chips have such buffers integrated. On the other hand this can speed up PDO object handling. Slave FSM role lies in updating this buffers after device specific event such is timer event or process object value change. The CAN driver sends objects from its buffers when needed (after SYNC object or as a response to RTR object). Consequently slave FSM has to read this buffers after WPDO object arrival.

PDO objects table is a memory mirror for transmitted and received PDOs. When a new RPDO comes, it is written into the table and PDO processor is notified by master FSM about this event. On the contrary, when PDO processor updates some object in the write part of PDO table, the FSM should be notified to allow it to transmit object change across the network.

The PDO processor module is used to synchronize PDO mapped objects values and real world data examined or set by computer hardware. Every PDO mapped object has assigned its reading and writing routine called PDO handler. These handlers are written by control system developer. Handlers placed in PDO processor share PDO table with FSM. In this table are stored/retrieved process data according to external events and occurrence of messages on the CAN bus. PDO processor is an user written set of functions designated for processing objects from read part of PDO table and generating new value of objects in write part of PDO table. New generated write objects can be sent across the CAN, if the processor notifies master FSM. This way the RPDO-TPDO mapping rules or control algorithms can be realized.

Some of SFO (Special Function Object) are handled directly by CAN driver. Such objects are the SYNC or RTR frames

Other objects (SDO, NMT) are sent through the SDO API to OD driver. OD driver is responsible for all object dictionary manipulations, that means getting and setting object values. If the PDO mapping change occurs due to SDO object processing, OD driver informs slave FSM (via SDO API) to correct PDO handlers table to reflect PDO mapping status in OD properly.

OD driver communicates with OD daemon, which resides in the user space, through RT FIFOs. OD daemon offers set of primitives to provide basic manipulations with OD like get/set object value, add object, delete object etc. OD daemon also owns slave OD in its memory space.

Slave OD can be loaded onto OD daemon memory by EDS parser. As EDS parser will probably serve CAN/CANopen monitor connected with it via Unix socket. This gives us the opportunity to control the daemon and slave OD remotely using TCP/IP. EDS parser is also responsible to read HDS and make appropriate changes mapping of handler functions in PDO processor module. This ensures that the proper handlers will be called for certain PDO objects.

The main difference between CANopen master and CANopen slave device is in OD and PDO processor module. We can say, that slave is more or less master with only one EDS in its OD and some restrictions on functionality (can't send NMT objects etc.). That means, that the slave device is a special case of the master one. Thanks to this generalization we can have also one code for the master and slave device and determinate final behavior by configuration and by loading different PDO processor module.

1.2.3.3. RT-CANopen device in the Hard real time space

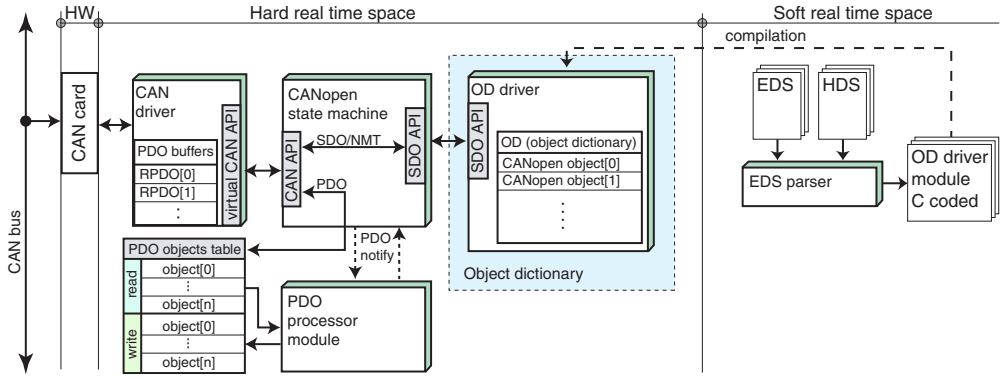


Figure 1-14. Hard real time CANopen device architecture

This architecture is very similar to the previous one. The main difference lies in OD position which is a part of OD driver module now. Every other part of slave remain the same. OD driver module is compiled from source code generated by EDS parser from the slave EDS, HDS and the empty OD driver template.

Benefits of the kernel space solution

- Faster SDO object processing.
- Slave does not need user space applications to work properly.
- Slave can be implemented to other POSIX compliant real-time OS like RTEMS.
- Suitable for CANopen slave realization in embedded systems.

Disadvantages of the kernel space solution

- OD is static, no objects can be added or removed.
- EDS parser can not explore OD any more. The diagnostic pipe has to be used for that purpose and all the information must be communicated through SDO API and slave FSM.

1.3. Verification of Distributed Systems

1.3.1. Problem statement

This chapter deals with a design conception of theoretical study offering methodology tool supporting analysis of distributed Real Time (RT) systems. Figure 1-15 illustrates mayor topic of verification of distributed systems . Figure shows a control system consisting of *n* independent processors and communication bus with general communication protocol. Let us consider the parallel machines running applications in the real-time operating system (RTOS) environment and further let us consider the communication protocol behaving in Real-time manner.

The crucial problem is whether the general real-time control system (RTCS) [ctu_ijk_bib_Buttazoo97] behaves in RT manner. This problem can be split into three subproblems that can be futher composed together:

- application SW (modeled by application developer)
- RTOS (study of preemptive and cooperative schedulers)
- RT communication - CAN (Medium Access Control modeling)

Coresponding three sub models can be futher combined to create RTCS model and it's possible behavior can be defined. On the other hand desired behavior of the RTCS has to be specified in the form of properties (e.g. deadlock, missed deadline, ...).

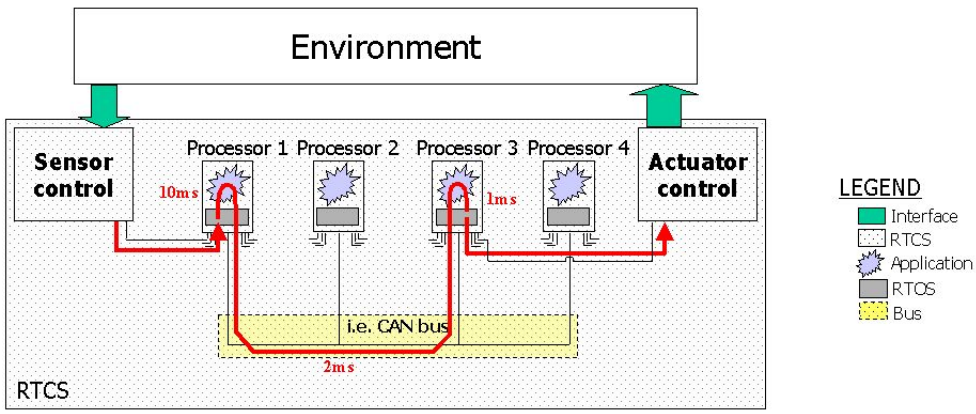


Figure 1-15. Real time control system structure with denotation of computation/communication times

OCERA goal is to provide

- model of RTOS and CAN
- develop examples of typical applications
- provide methodology for model checking of RTCS

To resolve the above mentioned problem we use a mathematical formalisms based on:

- system specification by means of PN or communicating automata
- design system behavior formulated by means of LTL
- verification algorithm

Especially we use the tools that have been already applied to solve similar problems (PEP tool, UPAAL etc.). These tools are able to model concurrent systems and mainly them.

The application developer willing to check to RT properties of the RTCS is required to model the application SW, to use properly models of RTOS and CAN and to specify correctly requested system properties (desired system behavior). While using the linear temporal logic (LTL) the application developer can verify his RT applications that are communicating via CAN by checking of properties like for example whether all task deadlines are satisfied or whether the message is received before another one. This approach is an alternative to the one known as VOLCANO [ctu_ijk_bib_volcano], and it offers more general framework for verification.

1.3.2. Modelling of communication protocol

When are modelling and verifying a communicating system we have to understand the idea of communication protocol design [ctu_ijk_bib_Holzmann91]. At the start of the design we must answer the next five questions:

- What service should be provided by the protocol?
- What assumptions are about the environment in which the protocol is executed?
- What vocabulary of messages should be used to implement the protocol?
- What encoding (format) of each message in the vocabulary should be used?
- What procedure rules are guarding the consistency of message exchanges?

If we satisfactorily answer the questions we can suggest concrete structure of communication protocol. It is evident that the most complicated design problem is the most difficult is his resolving. Partitioning the problem to small subproblems is reason that tell us common sense. The subproblems can be either easy to solve or they have been solved before. Some of them can be for example ISO/OSI model of protocol layers, encoding of messages (e.i. CRC) or access control at physical media (i.e. MAC).

In OCERA project we are interested in CAN bus communication protocol due to its real-time properties (see "Survey on RTOS"). The protocol will be modelled and it will be included to RTCS as communication part.

1.3.2.1. Example of CAN Medium Access Control model

CAN is based on the so-called broadcast communication mechanism. This broadcast communication is achieved by using a message oriented transmission protocol. CAN defines address of messages that are identified by using a message identifier. The identifier serves both as unique number as message priority. Bus access conflicts are resolved by bit-wise arbitration on the identifiers involved by each station observing the bus level bit by bit. The example deals with Medium Access Control of messages with different priority.

Model of the Medium Access Control (MAC) is realized by PN shown in Figure 1-16. There are three MAC of different priority messages. Places

- P1, P2 and P3 characterize MAC-H - MAC of high priority message
- P4, P5 and P6 characterize MAC-M - MAC of medium priority message
- P7, P8 and P9 characterize MAC-L - MAC of low priority message

Those MAC of the massages have the next three states:

- P1, P4, P7 - process in RTOS is computing its own task and it needn't communicate via the bus in this time
- P2, P5, P8 - waiting for yielding of communication on the bus. If there is a message with higher priority it can't be sent.
- P3, P6, P9 - message is sent on the bus. The message wins the CAN arbitration and can access the bus

P10 characterizes that the bus is free. P11 and P12 reflect the fact that the high priority (or medium priority respectively) message is willing to sent or not.

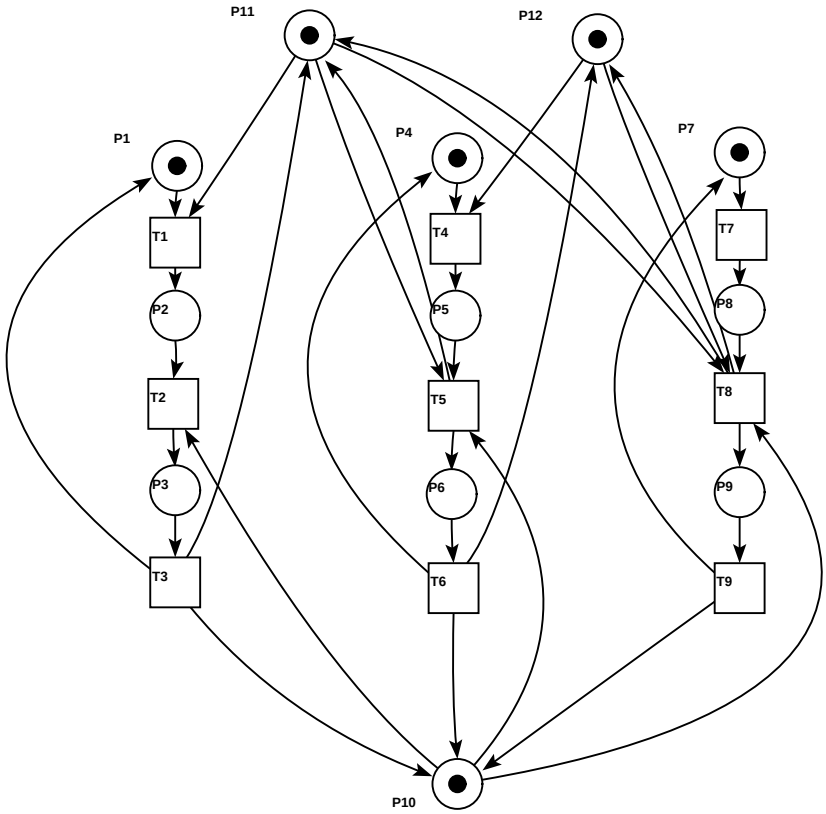


Figure 1-16. Model of CAN bus access method

Verification example of the model is shown bellow. Prior to the use of the verification tools the verification properties have to be defined. An examples of the properties are the next ones:

1. Does the system include deadlock?
2. Is there any state on which two messages are sent on the bus at the same time?

These properties have to be transformed to the verification tool formalism (the tools are described in follow section). The results of Analise are:

1. Deadlock-free analysis result: No deadlock
2. The state doesn't exist because there is no marking when two of P3, P6 and P9 are marked at the same time. The output of reachability tool is No.

PN can also contain for example time information which can influence behavior of application in global point of view. For example the computing time or ping-pong response times. Application developer can also check if the evolution of states is running in right way. For example:

- Does exist the sequence:

  1. send high priority message
  2. communication via bus for high priority message was permitted
  3. send low priority message
  4. communication via bus for low priority message was permitted too?

SPIN model checker tool is suitable for this kind of verification.

1.3.3. Model of RTOS

The term real-time is frequently used in many application fields. The definitions [ctu_ijk_bib_Buttazoo97] adopted by us is that the main difference between a real-time and non-real-time task is that the real-time is characterized by a deadline, which is the maximum time within which it must complete its execution. In critical applications, a result produced after the deadline is not late but wrong! RTOS is operating system in the manner of the definition.

We need RTOS model in this point of view. The model connects application part and communication part of RTCS verified by the verification tools. Since this problem is difficult to solve in general sense when preemption is allowed. We concentrate on cooperative schedulers with limited processor capacity assigned to interrupt handlers.

1.3.4. Tools

To modelling and verifying the models above we use a tools that are allowed do that with sophisticated mathematical formalisms. This tools are based on Petri Net or communicating automata and they proved their qualities in research community. Our intention is to use them in applications based on RTOS and fieldbus systems.

1.3.4.1. PEP tool

PEP tool (Programming Environment based on Petri nets) [ctu_ijk_bib_best98] [ctu_ijk_bib_peptool] is able to model concurrent systems and to verify them by partial model checking based on a compositional denotation Petri nets semantics. The language supported by the tools covers block structuring, parallel and sequential composition, synchronous and asynchronous communications and so on.

Modelling allows to create either graphical version of Petri net model or structured program code of the model in B(PN)^2 (Basic Petri Net Programming Notation) [ctu_ijk_bib_best93] or SDL (Specification and Description Language) [ctu_ijk_bib_peptool].

PEP contains those verification components:

- Deadlock-free tool - the tool checks whether or not a state can be reached in which no transition is enabled and displays corresponding transition sequence leading to such a state if one exists.
- Reachability tool - the tool checks whether a (sub-) marking of a PN is reachable. The names of the places and the number of tokens on these places can be and the results of the previous test are displayed.
- SMV - the tool serves Control Tree Logic (CTL) model checking.

- SPIN - it serves to invoke a fast model checking algorithm for Petri nets based on Linear Temporal Logic (LTL). The input is PROMELA formalism and property from PEP Formula editor.

1.3.4.2. UPPAAL

UPPAAL [ctu_ijk_bib_uppaal]allows modelling, simulation and verification of real-time systems. It is appropriate for systems that can be modelled as collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables.

Typical application areas of the tool include real-time controllers and communication protocols in particular, those where timing aspects are critical.

Bibliography

Gerard J. Holzmann, 1991, Prentice Hall, *Design and validation of computer protocols*.

Giorgios C. Buttazzo, 1997, Kluwer Academic Publisher, *Hard Real-time computing systems: Predictable Scheduling Algorithms and Applications*.

Eike Best and Bernd Grahlmann, 1998, *Programming Environment based on Petri nets: Documentation and User Guide Version 1.8*.

Eike Best and R. P. Hopkins, 1993, *B(PN)^2 - A Basic Petri Nets Programming Notation*.

UPPAAL tool: http://www.docs.uu.se/docs/rtmv/uppaal/.

PEP tool: http://theoretica.informatik.uni-oldenburg.de/~pep/.

Ken Tindell and A. Burns, 1994, *Guaranteeing Message Latencies on Controller Area Network (CAN)* .