CSP Channels for CAN-Bus Connected Embedded Control Systems *)

Bojan Orlic and Jan F. Broenink

Twente Embedded Systems Initiative,
Cornelis J. Drebbel Institute for Mechatronics and Control Engineering,
Dept. of Electrical Engineering, University of Twente,
P.O.Box 217, NL-7500 AE Enschede, The Netherlands
Phone: +31 53 489 2817   Fax: +31 53 489 2223
E-mail: B.Orlic@utwente.nl, J.F.Broenink@utwente.nl

Abstract – **Closed loop control system typically contains multitude of sensors and actuators operated simultaneously. So they are parallel and distributed in its essence. But when mapping this parallelism to software, lot of obstacles concerning multithreading communication and synchronization issues arise. To overcome this problem, the CT kernel/library based on CSP algebra has been developed. This project (TES.5410) is about developing communication extension to the CT library to make it applicable in distributed systems. Since the library is tailored for control systems, properties and requirements of control systems are taken into special consideration. Applicability of existing middleware solutions is examined. A comparison of applicable fieldbus protocols is done in order to determine most suitable ones and CAN fieldbus is chosen to be first fieldbus used. Brief overview of CSP and existing CSP based libraries is given. Middleware architecture is proposed along with few novel ideas.**

Keywords – **real-time, CSP, fieldbus, control, distributed, framework, fault-tolerance**

I. INTRODUCTION

Making embedded distributed fault-tolerant hard real-time systems requires substantial knowledge both in application specific domains, control engineering and in the software development field. There is lack of employees with sufficient knowledge in all mentioned areas and using application specialist and software specialist to write separate program modules and interconnecting those modules later in development process, generally does not yield satisfactory results.

Main objective of this project is to make a flexible framework that will enable control-system domain experts to build distributed fault-tolerant hard real-time systems without much help of computer software specialists. They should be able to easily use code generated from control-system modeling and design tools like Matlab, Simulink and 20-Sim. Unfortunately, although there is need for building parallel and distributed systems, most of the mentioned tools still generate sequential code. Reason for this approach is mostly the huge complexity associated with multithreaded programming. We want tools to assist designers in transparent and structured way of using multithreading for implementing inherently parallel control systems. On quasi-concurrent single processor systems this is solved by using our CT (Communicating threads) kernel library [1, 2]. This CT library is based on CSP (Communicating Sequential Processes) theory [3, 4]. Currently, we have C, C++ and Java versions of the CT library (CTC, CTCPP and CTJ). Since we want systems that are working really parallel and not just quasi-parallel, aim of this research is extending CT kernel library with real-time middleware operating over fieldbuses.

In existing control systems, there is often gap between timing constraints of control theory model and practical implementations, causing uncontrollable performance degradations [5]. Hard real-time systems are usually mission-critical, and thus they usually encompass some support for fault tolerance.

An overview of main timing, scheduling and fault-tolerance properties and requirements of real-time control systems is given in section 2. Section 3 is an overview of existing middleware solutions and their applicability in our framework. Since this middleware will be made using existing fieldbus protocols as lowest layers, a brief overview of fieldbuses, their history and comparison is given in section 4. As the CAN fieldbus is chosen to be first fieldbus used, separate subsection is dedicated to its properties and problems in real-time applications. Section 5 presents basic ideas and concepts of CSP, their appliance in CT kernel library and ways of formal check-

[*]) This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs an the Technology Foundation STW.

ing for synchronization problems. In the In the last section, it is presented how all requirements and design principles derived in previous sections are thought to be incorporated to form the new framework design.

II. DISTRIBUTED REAL-TIME CONTROL SYSTEMS AND REQUIREMENTS

Control systems typically function at number of levels (Figure 1).

Supervisory control ensures that overall aim is achieved by using monitoring functions, safety, fault tolerance and parameter adaptation algorithms. *Sequential control* produce sequences of operations, like in washing machine timed program.

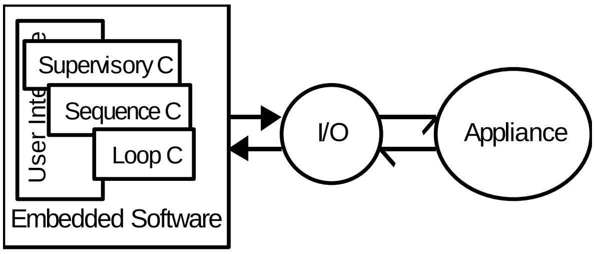

Figure 1: Embedded Control Systems Architecture

At lowest level there is *loop control* that periodically executes the following three phases: gather data from input sensors, calculate control signals according to chosen control algorithms and send them to actuators. Time between related sampling and actuation actions is called *control delay* [5]. In existing control systems, there is often gap between control theory model that assumes equidistant sampling with constant sensor to actuation delay time and practical implementations based on response time oriented scheduling models where it is not important when is task executed as long as it finished before its deadline [5, 6].

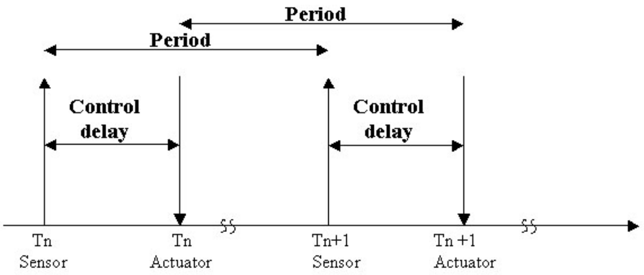

Figure 2: Equidistant sampling with constant control delay

III. TIME PROPERTIES OF PROCESSES AND MESSAGES IN CONTROL SYSTEMS

Parallel systems are made by dividing system into indivisible schedulable elements called *tasks*. Considering regularity of execution task can be: *periodic, aperiodic* and *sporadic*. Periodic tasks are required to execute exactly once every period. Aperiodic task can be triggered at any time and at any rate. Sporadic tasks are special kind of aperiodic tasks where maximum inter-arrival rate is predefined. Sporadic tasks can be handled in time predictable way by using periodic servers to handle them and reserving time slots according to defined minimal inter-arrival times.

Real-time systems are usually described as systems where not only the logical correctness of the results is important, but also time at which results are obtained. Therefore tasks and messages can also be classified according to changes in their utility functions over time (Figure 3). If results, arriving too late, have catastrophic implications, systems are characterized as *hard* real-time systems. If violated timing requirements affects performance but can be tolerated, systems are characterized as *soft* real time systems. *Non real-time* tasks have flat utility function. There are also tasks that need to be done *precise on time* (bounded jitter is allowed but usually influences system's performance).

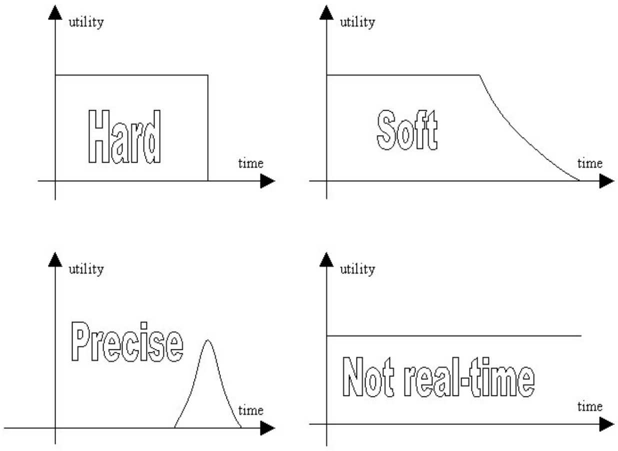

Figure 3: Utility functions for different tasks and message types (adapted from [7])

So far we mentioned two classifications of tasks and messages in real-time systems: one based on time-regularity and other on time-utility properties. All combinations of those two classifications are possible. In control systems, especially important are *precisely periodic* tasks, i.e. periodic tasks that must execute (not just to be released) exactly one period apart in predefined time.

Distributed scheduling techniques are described in detail in [6-8]. The kernel should group all scheduling specific details in separate modules to allow ease of replacing scheduling techniques. In general, task scheduling and bus scheduling in distributed systems are not independent. For instance, late delivery of messages postpone the release time of receiving task and therefore influence task scheduling on receiving node. Also, late arrival of sender task can influence bus communication if this communication is time-triggered. A holistic approach to task and bus scheduling was proposed in [9]. Nevertheless, the framework should decouple bus scheduling and task scheduling in a way that algorithms used to implement them can be changed independently. In distributed systems, clocks must be synchronized and *time master* is processor or dedicated hardware with reference real time clock. To tolerate for failures, several nodes should be potential time-masters. Highest priority task of time-master can be used to periodically produce clock synchronization and strobe messages.

Periodic messages are usually sent and received by periodic tasks (sending message periodically can also be triggered directly in hardware) and most often in control systems they directly or indirectly represent state variables. If a control algorithm can tolerate vacant sampling by reading old values, channels for periodic messages can be realized as *overwrite buffers*. Otherwise *synchronous rendezvous buffers* must be used. When using synchronous buffers, it is possible that processes blocked on channels can form a cycle. This is called deadlock. In CSP-based systems, deadlock situations can be detected and eliminated during design phase using some CSP based formal checking tool (etc. FDR tool developed by Bill Roscoe). To eliminate risk of deadlock, channels could encompass notion of timeouts and both sender and receiver should define how much they are prepared to wait for rendezvous.

Aperiodic messages (etc. alarms) are event triggered and they usually result in waking up some aperiodic event handling task. Calamities are mostly detected by more than one sensor, resulting in an avalanche of error messages. This state is known as *alarm shower* [10] and in such cases aim of utmost importance is to maintain temporal ordering of events such that the event causing the malfunctioning can be determined. Sensors sending alarms during an alarm shower can be on different nodes and in case they send messages in same time, the message with the highest priority will be sent first. The temporal order can be maintained either by including time-stamps in event messages or by relating the time of the event with the priority of the message. The former approach introduces additional overhead for every event message, while with the latter approach we loose the possibility to prioritize alarm events according to their urgency.

Timing problems in real-time control systems and their influence on the control performance are not yet a totally explored area, but merely a subject of research [5, 6, 11]. Although modern control theory is making some advances towards control algorithms that handle time varying delays, most of the contemporary control systems are built based on control theory that assumes constant control delays and equidistant sampling. Translated to the task domain, this means that sampling and actuation tasks should be precisely periodic. Otherwise, the control performance of the overall system will detoriate and in some cases even the stability of system can be jeopardized.

Computational algorithms of each subsystem usually consist of periodic tasks dependent of each other and executing in some sequence, which can be described by *precedence graphs* (an example is shown in Figure 4).

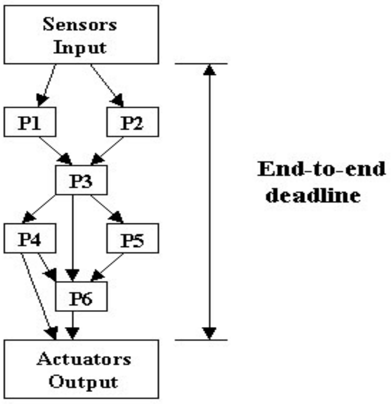

Figure 4: Possible precedence graph for one of loop control subsystems

This subtasks can execute on different processors, near data sources they use or because some of them need to use special processors or hardware. They start after the precisely periodic sampling tasks finish and must produce results before arrival of precisely periodic actuation task. Often, instead of specifying explicit deadlines to every intermediate task forming a precedence graph of computational algorithm, this behavior is defined only through *end-to-end deadlines*.

A. Fault-tolerance requirements

Fault tolerance is the ability of a system to keep providing specified services in spite of faults. Fault tolerance can be implemented in the framework, in the application or in both. Establishing fault tolerance facilities inside the framework make total fault tolerance support more efficient. Applications based on this framework are expected to use rather low-price COTS (commercial off the shelf) components then expensive specialized hardware. Therefore, software methods for fault tolerance are preferred choice.

Vacant sampling is situation when sampling data or output data from some computational task is not delivered in time to input of some computational task. This problem can arise from latency in overload situations or from failure of node, task or communication network. Vacant sampling or incorrect values can be experienced for longer amount of time and that situation is known as *temporary blackout*.

Often, after node failure and replicas activation, there is simply not enough processor power in system for all services. Answer to this challenge is in implementing support for *graceful degradation,* meaning that we make dynamical on-line reconfiguration trade-off between resource usage and performance level. In case of transient overload, way to make tradeoff between resource usage and precision of results is to use *imprecise computing.* This means that hard real-time tasks are divided in obligatory and mandatory part and mandatory part is performed only when there is enough time.

IV. OVERVIEW OF EXISTING MIDDLEWARE COMMUNICATION MODELS AND SOLUTIONS

A. Remote Method Invocation vs. Publisher/Subscriber communication model

What kind of communication model do we really need in distributed control systems: Remote Method Invocation (RMI) or the Publisher/Subscriber communication model? RMI as object oriented approach could at first sight seem to be far better. But in control system described through set of cooperating processes, most of the time we are interested just in passing data from one process to another and not in calling functions of remote objects. Besides that, RMI is based on point-to-point communication. If more than one process needs same data (etc. input from sensors), it must be separately sent to each of them. Hard real-time systems are usually mission critical and they often must contain fault tolerance mechanisms, which are usually made by replicating hardware and/or software modules. If there are replicated processes on different nodes, all replicas must get same input values and must agree on output values. Broadcasting is much more efficient for this kind of communication [12]. The Publisher/Subscriber method is far more simpler to implement and thus can be made far more efficient considering time overhead it will bring in control loop.

B. Why don't we use existing solutions?

Of course, at beginning of every project there is always a question: do we really have to make everything from scratch, or we can just (re)use something that already exists? First of all, commercially available and open-source middleware exhibit serious lack of fault-tolerance and real-time properties.

Most famous and most widely used middleware is CORBA. In embedded systems we aim to keep nodes as small and as cheap as possible, and CORBA is resource demanding middleware not tailored for embedded systems. It is based on RMI and not on the preferred publisher-subscriber communication model. Besides CORBA does not have appropriate mechanisms for guaranteeing real-time constraints. Maybe, we could use the CORBA reference model, cut out performance bottlenecks and try to make it real-time. This was actually done at Washington University in TAO project [13], but still the resulting middleware is too resource- intensive to fit in embedded systems. Anyway, the TAO project yielded many communication design patterns and some of them might be applied in our middleware.

From other existing middleware solutions, Jini is offering some interesting services. For instance, services can discover other services in the system; they can signup to be notified of other services/nodes appearing/disappearing. Furthermore, it is possible to download code. First problem is that Jini is implemented on Ethernet and therefore inherits its unpredictable message delivery time. One of the major goals of Jini was to "raise the level of abstraction of distributed programming from the network protocol level to the object interface level" [14]. Based on this, we could try to port this Ethernet-only middleware to work on some real-time fieldbus, for instance on CAN. This was done [14], at Carnegie Mellon University as part of RoSES project [15], but they were not satisfied with the achieved results. It appeared that in spite of the proclaimed intentions, some TCP/UDP specific features like host names and port numbers had crept into Jini's object interface level specifications. After discovering services, Jini communication between two services is based on RMI communication model as in CORBA. Therefore, that gives rise to problem of achieving efficient multicast and broadcast transfers. Besides, since Jini uses some functions not supported by KVM (embedded version of Java Virtual Machine), the whole Java Virtual Machine had to be used, resulting in a too large memory footprint. However, it is possible to tailor KVM and Jini to be compatible.

Reusing is not all about reusing existing components and source code. What we can also reuse is experience in solving some general and often recurring problems and usually this experience is best encapsulated in shape of various design patterns. Valuable directions for using OOP and design patterns in embedded hard real-time framework can be found in the AOCS (attitude and orbit control systems on satellites) framework [16]. Although the AOCS is a domain specific framework, it demonstrates a design approach that could be used in any embedded hard real-time systems.

V. FIELDBUSES

A. *Historical overview*

Historically, fieldbuses are end product in evolution of process control system architectures. In 1960s there were central mainframe computers (constituting single point of failure for system) with costly point-to-point link to each field device. In early 1970s, supervisory and control functions were delegated to several controllers still located near mainframe computer and connected with it and with field devices by point-to-point connections. In mid 1970s controllers moved from control rooms closer to field devices, using serial network to communicate among themselves and point-to-point connections for its own sensors and actuators. In early 1980s, serial networks known as fieldbuses are used for communication among field devices. Field devices became smart equipment with built in automatic calibrations, correction of offsets/drifts, executing local control and fault-monitoring functions [17].

A fieldbus is serial bus network that provides communication among field devices (sensors, actuators, controllers, regulators). It is usually based on layered structure made omitting few highest layers in OSI reference model. A fieldbus offers set of communication services and protocols. Currently there are more then 30 different kinds of fieldbus protocols: CAN [18, 19], Profibus [20], WorldFIP [21], LonWorks [22], Interbus, P-NET, EIB, DeviceNet, Hart, etc.

In the rest of this section properties of various fieldbuses are compared, but emphasize is put on CAN bus, because it has been chosen to be first one on which CT libraries will be ported.

B. *Comparison, classifications and CAN basic properties*

There is no generally accepted standard concerning fieldbus protocols. Therefore, in order to choose fieldbus most convenient for implementing CT library channel concept, some theoretical comparison studies between existing fieldbuses protocols concerning their application in hard real-time systems had to be done. Based less on this comparison and more on fact that it is already present in our lab, the first choice was the CAN fieldbus.

Many of fieldbuses are tailored to satisfy some specific request, thus forcing the system designer to be very careful when choosing appropriate fieldbus solution for its application. Some of most important differences like available bandwidth, message delivery times (through priorities), and robustness are often consequences of different optimizations in the underlying MAC (Media Access Control) implementation. For instance, CAN is tailored for automotive networks and thus it provide deterministic, reliable communications with short prioritized messages and extensive error detection, but a price is paid in limited length and speed (up to 1 Mbit/sec) of the bus. The Profibus has different variants tailored for factory automation, process automation, motion control and safety-relevant applications. LonWorks is optimized for flexibility to allow broad variety of applications. WorldFIP employs ‘producer/consumer’ mechanism allowing easy construction of distributed real-time databases. Besides, in WorldFIP aperiodic traffic is handled using an *external control* communication model, where the application processes explicitly trigger transaction and *autonomous control* is employed for time-triggered periodic traffic through static table-based scheduling. This makes it more resilient to *babbling idiot* faults and *event shower* situations. Most of other fieldbuses implement only *external control* based on priorities.

Classification of fieldbuses can be based on different properties. If we consider the approach used to identify source and destination nodes (addressing), there are two categories: fieldbuses that specify address of receiving node(s) directly in the message header and content oriented (or source addressing) fieldbuses, where receiving nodes filter message IDs and take only messages of types they are interested in. While direct addressing is more widely used, content based addressing is applied in WorldFIP and CAN fieldbuses. Advantages of content oriented approach on CAN fieldbus are: flexibility in adding nodes (no change of software on existing nodes is needed), guarantee that message is accepted either by all nodes or by no node, efficient support for multicasting. Multicast communication is needed when data collected by a sensor or intermediate results of control algorithms are shared among few nodes and is also useful for achieving group communication among replicas in fault tolerant systems. Many fieldbuses support one-to-many communication (multicast), but at the direct addressing approach, extra hardware is needed at every receiver to filter the broadcast address. In content-based fieldbuses, receivers filter message ID anyway. This property makes implementation of Publisher/Subscriber model easy and efficient.

If we base classification on the way fieldbuses resolve bus conflicts there are: Token based, Master/Slave, TDMA and carrier-sense (CSMA/CD, CSMA/CA and CSMA/DCR) fieldbuses.

Token based buses prevent bus conflicts through the use of special signals-tokens. Token is passed from one node to next in virtual or real ring structure and only holder of token has access to the bus. Token based buses (PROFIBUS, MAP, P-NET, IEC Fieldbus, FDDI, ARCNet, PLC-192) are predictable, but they suffer from large token-managing overhead under light traffic.

Master/Slave token passing (MS/TP) fieldbuses (USB, MIL-STD-1553B, MIL-STD-1773) minimize communication work of slave nodes having a centralized master periodically polling the slave nodes (can be seen as master giving a token to slaves) for information. A centralized master constitutes a single point of failure. This fault tolerance problem is in Profibus solved using a multi-master approach.

In *Time Division Multiple Access* (TDMA) systems each node transmits during its own offline predetermined time slot. TDMA systems (TTP-C, TTP-A, DATAC…) give guarantee of deterministic behavior, but there is lack of flexibility.

Carrier-Sense bus means that nodes transmit only after detecting the idle bus. Still collision can occur if two or more nodes detect idle bus and start to transmit in same time. There are three different mechanisms for resolving conflicts on bus and thus three different kinds of carrier-sense buses: Carrier Sense Multiple Access with Collision Detection (CSMA/CD), Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and Carrier Sense Multiple Access with Deterministic Collision Resolution (SMA/DCR).

In CSMA/CD if two or more nodes collide, they back-off and try to retransmit message after randomly determined time period. CSMA/CD is non-deterministic and thus buses based on it (Ethernet) are not appropriate for hard real-time systems.

Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) combines CSMA/CD efficiency under light traffic and token based efficiency under heavy traffic. Each node has local timer pre-set with a unique value. After a start signal sent periodically by leader node, all nodes release their timers. Node with lowest value in timer gets the bus first. All other nodes sense bus activity and reset their timers. In one cycle, a node can transmit one periodic message, one urgent aperiodic message and one not urgent aperiodic message. After sending those messages, node waits for leader to signal start of next period [17].

The CAN fieldbus belongs to CSMA/DCR category. Identification field of CAN message beside its role in content based addressing, constitutes priority for bitwise arbitration in case of collision. In fact, the CAN bus behaves as a wired AND gate with each node seeing all output. The *dominant value* (‘0’) on the bus always overwrites the so-called *recessive bit* (‘1’). All nodes competing for access to the bus transmit bit-by-bit their identification field and listen to the output value. The node that sent a recessive value and perceived dominant output looses arbitration and backs off. The node transmitting the highest priority message gains immediate access to the bus and can transmit without any delay.

Fieldbuses can be compared based on maximal speed, maximal bus length and support for working in harsh environments (error detecting and recovery mechanisms). One such comparison [23] for three most widely used fieldbuses is given in Table 1.

With a maximum speed of 1Mbit/s CAN can not be qualified as a high-speed bus. The maximum length is inversely proportional to the used CAN bus speed and ranges from only 10 m for highest speed to few hundred meters for much more moderate speeds. Reasons for this length limitation originate from the CAN error detection concept, where every node must listen to every bit on the bus.

Some fieldbus protocols have built-in support for extending networks. LON identifies each node by a hierarchical address composed of domain, subnet and node address and allows routing of messages. PROFIBUS allows several segments interconnected by repeaters. P-NET contains several segments interconnected through gateways. CAN does not have built-in support for extending the network, but it can be done building higher level layers on top of the protocol.

The CAN protocol has fixed-format messages of limited length shown in . CAN-B has an extended message ID field, namely 29 instead of 11 bits. By sending a *remote frame* message, a node may require another node to send the corresponding *data frame*.

Error detection is based on CRC checking and bit stuffing. Any node detecting an error sends an *error frame* consisting of six consecutive dominant bits, thus violating bit-stuffing rules. Faulty messages are retransmitted automatically.

In CAN, fault *confinement* (restricting influence of faulty nodes) is implemented by counting transmit and receive errors per each node. When each of the counters reaches 127, node enters error-passive state where it can signal errors only while transmitting and is obliged to have extra eight bit waiting period after transmitting, before it is allowed new transmission.

Fieldbus	CAN	PROFIBUS	WorldFIP
Max Speed	up to 1 Mbit/s	500 kbit/s	5 Mbit/s (with Optical fibres)
addressing	source	direct	source
Max # nodes with/ without repeaters	N/A /30	127 /32	256 /64
Max distance with / without repeaters	N/A / 40m-1Mb/s 1km-20 kb/s	800 m/ 200 m	10 km/ 2 km
Arbitration	CSMA	Token passing	Bus Arbiter
Header/Data size	8 bytes fixed	250 bytes	1 to 128 bytes
Primary applications	In sensors / actuators Automotive	Inter-PLC commun. Factory automation	Real-time control Process/ machine Distributed data base

Table 1 Fieldbuses comparison

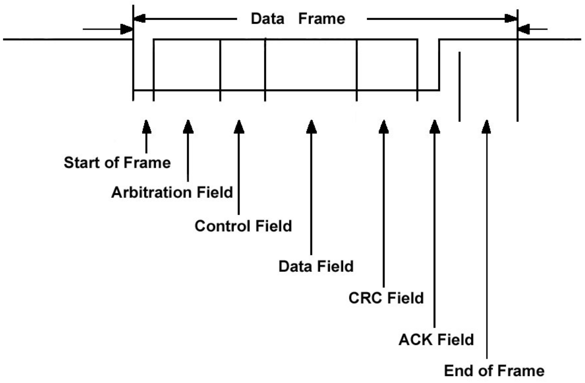

Figure 5: Format of a CAN message

C. CAN problems and protocols to solve those problems

Real-time fieldbus should have bounded response times of messages. Bitwise arbitration used in CAN enable a node transmitting the highest priority message to gain immediate bus access and transmit without any delay. Thus, it is possible to calculate worst case response time for highest priority message. Longest time node waits for the bus to become idle is equal to longest time to transmit CAN message. Problem is that latency for messages with lower priority seems to be unbounded. At University of York [24] analysis for calculating message response times on CAN has been done.

Although most failures are handled consistently by all nodes and there is general belief that CAN native mechanisms provide atomic broadcast, it was shown in [25] that there is possibility for inconsistent message delivery and generation of message duplicates in case of network errors. This scenario happens when two last bits of seven bit end of frame are faulty, receivers do not agree on correctness of message and sender crashes before retransmission. Mean time of such error is $3.98*10^{-8}$ which might be unacceptable for some safety critical applications. To solve this problem reliable broadcast protocol was made [25] as simple software layer on top of CAN. Complete set of atomic multicast and consolidation protocols as well as replication management framework have been proposed in [26].

Dynamic bus scheduling for CAN was investigated at the University of Michigan [27], resulting in proposed Mixed Traffic Scheduler (MTS) that was supposed to have better utilization then fixed-priority schemes and less overhead then dynamical earliest-deadline (ED) scheduling.

So far, several research groups have, with more or less success, ported CORBA on CAN [28]. But CORBA is RMI based and not real-time and therefore not suitable for hard real-time systems.

CAN fieldbus is particularly suitable for implementation of Publisher-Subscriber model, as shown in [29]. Same research group also invented new dynamic bus scheduling technique, where soft real-time activities are scheduled with EDF approach and deadlines of hard real-time communication is guaranteed by calendar-based resource reservation [12].

VI. CSP ALGEBRA AND ITS APPLIANCE IN THE CT LIBRARY

CSP [3, 4] is notation for describing patterns of communication by algebraic expressions that allows formal checking for undesired conditions like failures, divergences, race hazards, deadlocks, livelocks and starvation.

A number of concurrent programming languages like occam and Ada are based on CSP. In CSP each separate flow of control that executes sequential code along with all associated passive objects is called *process*. Process can communicate with other processes only through usage of special synchronization primitives called *channels*. Channels serve to hide location and internal structure of communicating processes from each other. Each process has separate address space and because channels have pass-by-value semantics, there is no way that one process can change state of some other process. Processes never access hardware directly and are therefore hardware independent. The channel communication model allows changing allocation of processes on processors without changing program code, which makes this solution scalable.

Process execution is organized hierarchically using sequential, parallel and alternative constructs. The *sequential* composition construct mean that its subprocesses are executed one at the time and in order. The construct finish when all subprocesses are finished. In *parallel* construct subprocesses are executed in parallel. If construct is executed on one processor then subprocesses are executed in quasi-parallel way by sharing processor time. Constructs end when all its subprocesses are finished. *Alternative* construct wait till one of the processes guarded by inputs, outputs and timeouts becomes ready, execute that process and finish. Although not described in CSP theory it is common to make two extra constructs by adding notion of priority to *parallel* and *alternative* constructs. Constructs derived in this way are *priparallel* and *prialternative*. One of possible CSP representations of precedence graph from Figure 4 is given on Figure 6.

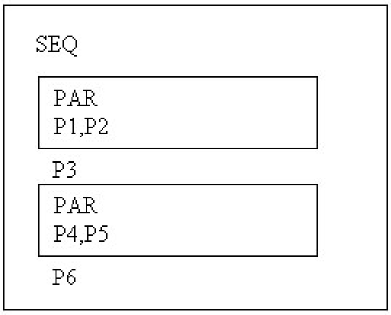

Figure 6: Transformation of precedence graph to CSP constructs architecture

There are two existing libraries providing process oriented design patterns for dealing with concurrency in Java: Communicating Sequential Processes for Java (JCSP) [30] and Communicating Threads in Java (CTJ) [1, 2]. Detailed comparison of two libraries is given in [31]. JCSP passes local messages by reference and only external messages by value. JCSP networking [31] solution is based on T9000 transputer communication mechanisms [32].

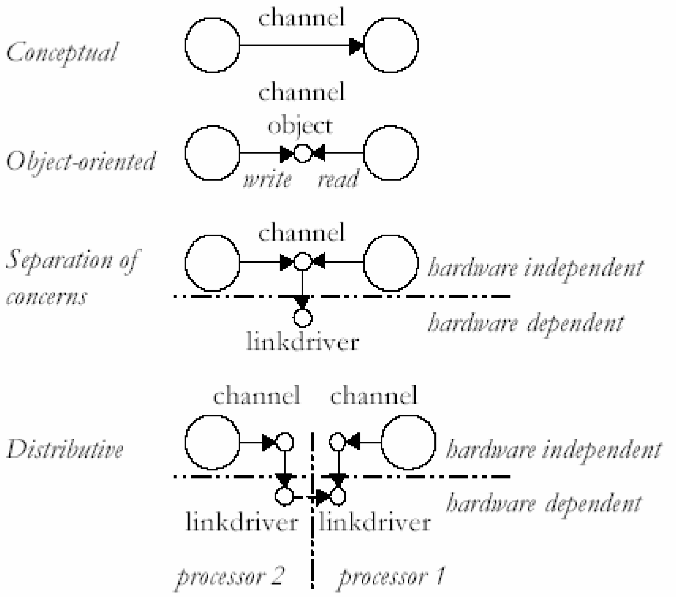

Figure 7: Channel Framework in CTJ

CTJ is intended to be the framework library for real-time systems. Therefore, instead of relying on different and often non-deterministic scheduling mechanisms of underlying operating systems, CTJ itself does thread scheduling. The Java garbage collector has non-deterministic behavior which makes it inconvenient for real-time systems. CTJ passes all messages by value, giving the same semantics to remote and nearby channels and encouraging reuse of objects, which results in less or ideally no activity of the garbage collector. In CTJ, channels are hardware independent, but can have plugged-in link-driver objects which encapsulate hardware-dependent device-driver code (Figure 7). Real-time control systems usually operate over fieldbuses. Thus making real-time library like CTJ distributed assume building in support for communication over various fieldbuses (Figure 8).

Ideally instead of having separate framework version for each network protocol, CTJ should encompass ability for distant processes to communicate over shortest route in heterogeneous network environment. Putting CTJ to work on CAN fieldbus is just a first step in making this library truly distributed in real-time control environments.

VII. PROPOSED ARCHITECTURE FOR REAL-TIME MIDDLEWARE

Since working on any framework is a process, especially while implementation is still far a way from being finished, I want to emphasize that proposed solutions to some of recurring problems in control and distributed

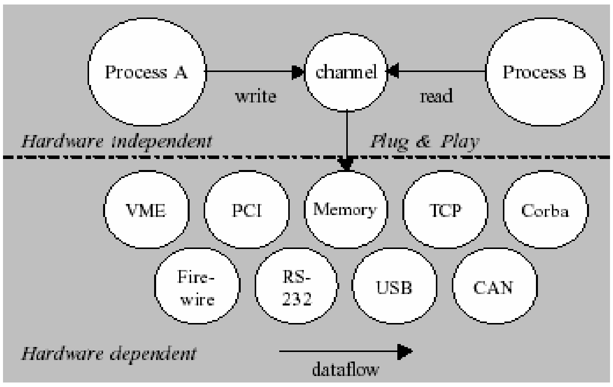

Figure 8: Plug and play framework for devices

systems are not definitive and are likely to be changed as soon as better and more elegant solutions came on surface. Consider following lines only as some of possible ideas being currently under consideration for building into framework.

A. Realizing end-to-end deadlines

Consider now subsystem with precedence graph shown on Figure 4. One possible scheduling approach is to derive release times and deadlines of all intermediate processes. First logical conclusion is that each process should have such a deadline that if it finishes before its deadline, all following processes and all following communication must have enough time for worst case execution. But missing this deadline based on worst case assumptions of all following processes and communications does not mean that there is no way to satisfy end-to-end deadline. On contrary, we can expect that in most cases end-to-end deadline requirements will still be satisfied. Possible solution is to define two kinds of deadlines for intermediate processes: soft and hard deadlines. Soft deadline would be calculated to leave enough time for sum of all worst case execution times of following processes and communications. This deadline should be forced by scheduling subsystem. Hard deadline would be derived based on minimal execution time of all following processes and communication. Error handling mechanisms should not be used until this deadline is missed, that is as long as there is slightest possibility to satisfy end-to-end deadline.

Expressing this in utility functions, yields diagrams as in Figure 9. Sensor and actuator tasks are precisely periodic and utility of their execution drops when jitter compared to fixed sampling and actuation points increase. Other tasks into precedence graph are constrained through order of execution and end-to-end deadline. Task utility is constant until it reaches soft deadline point where achieving end-to-end deadline can still be guaranteed even in case of worst case execution times of all following subtasks. After that point utility value of task execution drops until hard deadline point. Further from that point there is no way to achieve end-to-end deadline and further execution of task is useless, or we can say its utility value is zero.

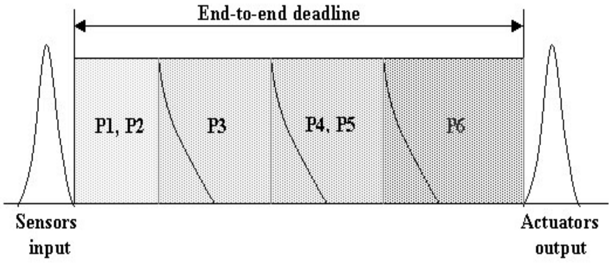

Figure 9: Utility functions for processes from Figure 4

B. Fault tolerance layer

Designer should be able to specify what should be done when vacant sampling is encountered: should some default value be used or should value be predicted from history values and how (value same as in previous period would be special case of this), or should the system enter some safe-state or even be shut down. Designer should also be able to specify allowed amount of lost information before registering temporary blackout situation and forcing system to shutdown or safe-state.

First we must make distinction between time criticality and utility criticality of tasks and messages. Functionalities essential for system might contain some soft and not real-time tasks considering time criticality. On the other hand, functionalities that are optional additions to system’s essential behavior can encompass hard real-time loops [15]. We can imagine each of those functionalities as separate *subsystem* consisting of mutually dependent tasks related through precedence graph and optionally constrained with end-to-end deadline. Furthermore these subsystems can be nested, thus enabling that only one part of functionality have hard real-time end-to-end deadlines. Subsystems are units of replication, meaning that internal outputs of processes inside subsystems need not to be agreed upon, while subsystem outputs consolidation is needed for replicated subsystems used by other subsystems. Sensor tasks and actuator tasks are grouped in separate sensor and actuator subsystems. Essential subsystems must be replicated, while optional subsystems may or may not be replicated depending on value of their criticality attribute. This replication can be either active with synchronous agreement protocol among replicas or passive where only one replica is executed at any time and is substituted only if it crashes. Note that, to tolerate for n faults, 2*n+1 active replicas must be used. Although process code downloading is desirable property, embedded networks usually have not enough bandwidth and therefore pre-positioning of replicas is preferred in first phase of framework implementation. Here we can follow Jini plug-and-play ideology by allowing higher level subsystems to reveal their presence to other subsystems on network. Subsystems should be characterized with two attributes: one specifying criticality and other specifying type of subsystem. Subsystems offering same functionality (having same type of service attribute) can be interchangeable. In case one of nodes fails, not replicated subsystems from failed node can't be recovered. Replicated subsystems need to agree on outputs and thus need to communicate through special kinds of channels-*group consolidation channels.*

Graceful degradation can be done by deciding to omit execution of some optional subsystems. Another way would be choosing to execute version of subsystems that require less resources. Third solution would be to intentionally change period for periodic subsystems. This is possible since period in control algorithms is generally not determined based only on Shannon theorem, but also based on required performance demands. Control algorithm can allow range of possible periods, where larger periods implicate somewhat worse but still acceptable control performance. [5, 6] Also, different subsystems implementing different control algorithms can be used for different ranges of sampling period value. So every periodic subsystem should define period for optimal control performance and period for achieving minimum allowable performance. Or we can even allow designers to make alternative subsystems and structures connecting period ranges with best suited alternative for that range.

General framework for building hard real-time control systems should not enforce neither one of proposed graceful degradation solutions, but should enable designers to easily and in transparent way implement each one of them.

Fault tolerance layer can be separate subsystem organized as priparallel construct consisting of Reconfiguration Manager, Replica Manager and user defined alarm handlers. They are all working on top of lower communication management layer and should be independent of underlying network protocol. When failed node is detected, Reconfiguration Manager is released to activate passive replicas and implement support for graceful degradation as described previously. Replica Manager is assisting group communication channels to achieve replicas determinism.

C. Scheduling integrated with CSP constructs

Slack (laxity) time is defined as the maximum time task can be delayed to complete within its deadline. In Figure 10 total slack time is sum of slack1 and slack2 times. But we are working with preemptive systems, where task execution can be interrupted and in classical scheduling approaches worst case execution time is fixed and slack time can't be updated properly.

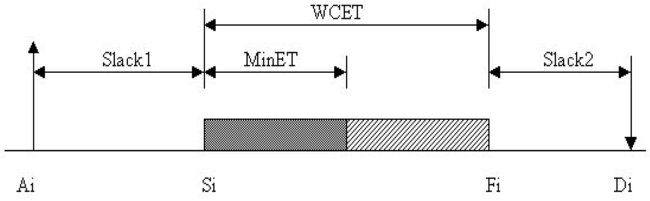

Figure 10: Usual way to calculate slack (laxity) time

Consider system consisting of many subsystem processes described with precedence graph as one on Figure 4 and optionally with attached end-to-end deadline. Specify worst case execution times of process Pi as Ci. If we have process P as sequential construct consisting of processes P1 and P2, worst execution time of P is C = C1 + C2. If P is parallel construct of same processes C = C1 + C2 in quasi-parallel realization (P1 and P2 are on same node) and C = max (C1, C2) in real parallel realization. Scheduler can be implemented to consist of three prioritized FIFO queues: hard real-time, soft real-time and not real-time queue. Each subsystem is sorted in one of those queues. Subsystem processes without end-to-end deadline go to not real-time queue, mission critical subsystems goes to hard real-time queue and all the others in soft real-time queue. Whenever one process in some construct finishes, all minimal and worst case left execution times of its parent constructs are updated. Adding new process dynamically also updates those times. For hard real-time subsystems we want to guarantee bounded execution times and thus we allow adding process to periodic hard real-time subsystem only before starting new job for that process. Scheduler must test schedulability of system with updated times of that subsystem. If system doesn't pass schedulability test, process is not accepted and case is transferred to Reconfiguration Manager to solve it.

The Scheduler process periodically gains the processor and sort ready queues according to their slack times (Least Laxity). Note that now minimum, slack and worst case execution times left, can be determined much more precise because their values are updated regularly. We can set thresholds for preemption to avoid context-switch related trashing, when two subsystems have close values of times used for scheduling. Each construct has an embedded scheduler that distributes processor time in the specified order for sequential constructs, fairly for parallel constructs and according to assigned priorities for priparallel constructs. When a subsystem gets processor time, it distributes it following its own hierarchy of nested schedulers (Figure 11).

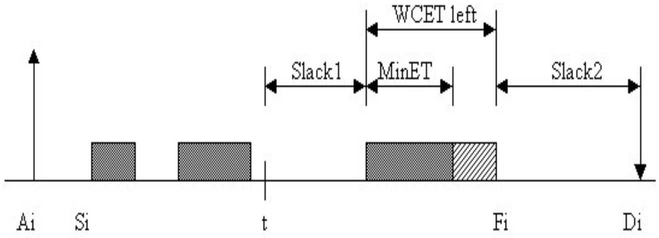

Figure 11: CSP construct based organization enable calculating worst case execution time left and slack times more precisely

This is a novel way to naturally integrate task scheduling with CSP construct-based application architecture. Since CTJ already has solid scheduling, proposed method for scheduling integrated with CSP constructs based architecture might be implemented in some later version of library.

D. Achieving constant sampling period and actuation delays

As it is elaborated in section II, control systems should have constant control delay and equidistant sampling. This is not easy to implement in practice, especially in heterogeneous distributed control systems where network communication is part of control loop and where multiple subsystems with different sampling frequencies and phasing are employed. Obviously, some kind of time-trigger is needed. Each subsystem can contain multiple sensors possibly on different nodes that should be sampled in same moment of time. Therefore, same reference real-time clock must be used to trigger all time-triggered actions.

Main stream in practice (and in CT library development team) is to use interrupt service routine triggered by local timers. Interrupt service routine (ISR) can read data from sensor, write it to buffer and signal some semaphore to release computational task waiting for input data. In CSP architecture this would mean that we realize sampling and actuation in passive channel objects and we do not need special processes for those tasks. But I find this approach to be potentially dangerous. Local clocks may be synchronized, but there is always some jitter. Sampling is a process that has some duration from triggering A/D converters until sampled data is ready on output of converter. Because it is not possible to block ISR, this time is wasted for processor. Long execution of one ISR can delay execution of other ISR that should also be precisely periodic. Even if we could bound execution time of ISR it is still complicated to implement scheduling algorithm that would take into account those times. After all, we can expect that in complex control system might exist some control process more urgent then current ISR. Furthermore, in custom based scheduling systems like CTJ, ISR doesn't have its own context, it is working in a context of interrupted process. Therefore, ISR should never write to buffer, because buffer implementation always encompasses mutual exclusion and potential blocking of current process. Conclusion is that interrupts should be used only to signal sampling moment and release waiting sensor tasks. Similar analysis can be drawn for actuator tasks. [7, 33, 34]

More efficient would be to have precisely periodic, high (not necessarily highest) priority, sensors tasks to transfer data from peripheral short-term memory to appropriate channels and their remote proxies. Note that multiplexed A/D converters can be used, but as usual their inputs must be from different subsystems with distant enough phasing of sampling moments. T*ime master,* or local timers if satisfying level of clock synchronization can be achieved, must periodically send *strobe message* (it must have very high priority) for each group of sensors. Strobe signal should release separate process, which will select proper inputs on appropriate multiplexers and initiate start of conversion in all A/D converters from that subsystem. Same broadcast address for start of conversion of all A/D converters would be particularly convenient in this case. Separate sensor processes are used for each sensor to collect data. Sensor processes are blocked and wait for interrupt from converter. In meantime other processes can execute. When conversion is finished, sensor tasks are released to collect data from peripheral short term memory. Alternatively, sensor tasks can be released upon strobe and send to sleep for predefined conversion time. After wake up, they poll status of converter. If number of available interrupts is to limiting, scheme with only one interrupt per subsystem can be used. Interrupt can poll converters and release appropriate sensor tasks. To reduce number of bus cycles, additional simple hardware can allow that each nearby converter from same subsystem use one bit in joint message to expose its status. Sensor processes are not all of equal importance, so they can be grouped in sensor subsystem and organized as priparallel construct.

Similarly, constant control delay can be achieved by employing actuator processes.

Often we want to use same sensor data in more then one computational task. This is one-to-many communication, not with altering readers like is usually done, but more like a broadcast transmission to all readers. Currently in CSP this is done through usage of "delta" process. This process reads data from one channel and writes the same data to channels of all processes interested for that data. Alternative solution could be *half-rendezvous channels*, that from outside have same semantics as delta process meaning that applications based on them can still be checked with CSP based checking tools. Main difference is that half-rendezvous channels are realized as passive objects and not as processes (active threads). Therefore main advantage is that this approach does not increase context switching times. The disadvantage is that this channel is not a compositional element like a delta process; rather it is customized for single but often recurring situation. The name *half-rendezvous* is given because readers wait for rendezvous and writer engage in rendezvous only if there are readers waiting. This means that the sensor process writes data to buffer inside the channel and releases all computation processes waiting on that channel. We do not want process that enter channel after sensor process and are still able to produce outputs on time, to be blocked until next sensor data arrive. Thus, what we can do is to make all processes interested in sensor data subscribe to channel. This will give channel some knowledge of processes and therefore is not totally in spirit of CSP, but is very useful concept. So, channel will maintain lists of all processes and ones that already used current data Process performing read function is blocked only if it wants to take same data twice. If some processes have not used current data and new data arrive, error is encountered and reported to fault tolerance layer. If this channel is built as remote, all proxies receive new data through broadcast transmission.

To achieve optimal control performance in control systems, the sampling period is often few times less then required by stability issues. This leaves room for tolerance to vacant sampling. Anyway, because constant computational time is also very important, often actuator should have some input in precise time. Thus, they are also precisely periodic processes waiting on channels to be released by time event. If, after released in precise times, actuator tasks would have to wait on rendezvous channels, their precise timing would be jeopardized. Therefore *asynchronous channels* are needed. To allow notification of actuator task of vacant data situations we can put a simple counter in the channel. Every write resets counter to zero, and every read increment counter. Read function returns data value and data status information. Status can have one of following values: *ok* if counter=1, *vacant* if counter>1 and *blackout* if counter exceeds value specified by actuator task. In case of *vacant* and *blackout* data status, actuator process will have to decide whether to use old value, some default value, to trigger entering safe-state (etc. shutting down all motors) or release some emergency process. If we want to enable replacing vacant data with some interpolated value we can introduce *asynchronous channels with history.* They can keep specified number of last received values as well as number of periods (number of reads) this values spent in status of last received and use some actuator specific mechanism for calculating interpolated values.

Note that still in some situation rendezvous communication for actuators and sensors tasks could be more suitable then proposed solutions. Choice between those two approaches is left to designer of system based on this framework.

E. Middleware organization

Computation tasks connected through precedence graph should use *rendezvous channels.* It would be convenient if both sender and receiver processes could specify amount of time they are ready to wait for rendezvous – timeout. For receiving process amount of time can be derived from minimal and/or worst case execution time. When receiving process gets data after minimal execution time of following processes expires, there is no use to do any computation with that data. Vacant sample should be encountered and fault tolerance layer should be notified of error. For sending process timeout is set according to need to handle next input data and can be set to be equal to period of whole subsystem.

Since running the Java version of the CTJ library is not possible without JVM and underlying operating system, on microcontrollers with low memory, C and C++ versions of library have to be employed. This basically means that library should enable nodes with different language versions of CT to communicate. Unlike C and C++, where size of some data types and Big/Little Endian order can be different on different machines, JAVA has same representation of data for all platforms. Therefore, decision is to send data in Java formats and systems with CTC or CTCPP libraries can have additional OSI *presentation layer* process to transform this data to local representations they use. JVM is solving different operating systems support for CTJ, but for CTC and CTCPP minimal differences might exists in versions of framework for different operating systems.

To have a scalable solution, we should allow processes to communicate over channel without knowledge where the other side is, addressing it just by its name. There must be some global base of channel names and route to channel expressed with its Channel ID, and Node ID. During its creation channel registers with Global Identification Broker, sending its name, type and Node ID and getting his unique Channel ID and route. Broker must be replicated to avoid single points of failure. Generally, it would be good to enable the designer to make a tradeoff between minimal time overhead when channels are created in initialization phase, convenient for hard real-time systems, and enhancing reconfigurability by enabling dynamical construction of channels Figure 12.

In *remote synchronized channels*, a message that is to be sent over network should have a criticality value

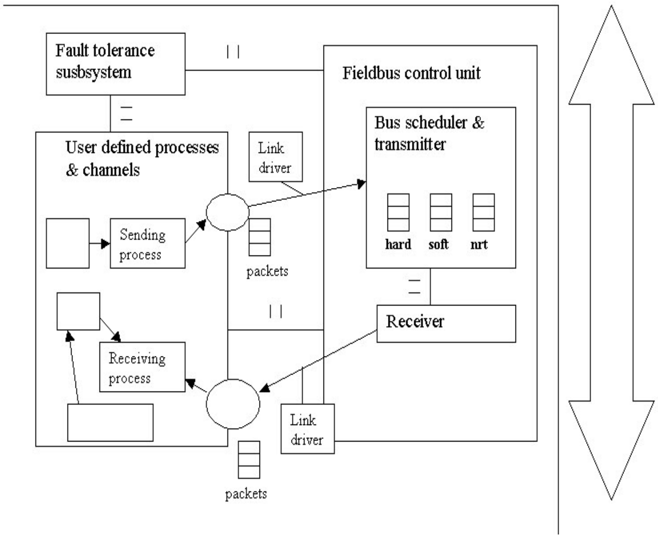

Figure 12: Main subsystem processes in middleware

(hard, soft, non real-time) and deadline defined. This value is used to dynamically determine priorities during bus scheduling according to dynamic bus scheduling technique. Message inherits criticality value from subsystem of sending process. A deadline value can again be derived from worst case execution time of following processes specified by receiver or in simplest case can be set equal to end-to-end deadline or period of whole subsystem. Node can be connected to many heterogeneous networks. Low level message shaping details are done in passive, link driver objects in context of sending process before it is blocked. Data shaping means that object to be sent must be flattened to stream of bytes and divided into packets of size limited by underlying network protocol. The header of the message contains a reference to its channel, size, deadline and criticality attribute. For each network on node, there is fieldbus control process constructed as parallel construct of two separate processes: one for bus scheduling and transmitting packets and other one for receiving packets Figure 12.

Multiple channels use the same fieldbus control process, concurrently writing message headers. Bus schedulers have separate FIFOs for every criticality attribute (hard, soft, not real-time). In each FIFO, message headers are prioritized by their deadlines. A bus scheduler takes packet from the channel pointed to by the first message header from the most critical FIFO and sets its Message ID used for bitwise arbitration and content addressing. The first part of the message ID is the priority field and the first two bits of the priority fields are used to distinguish between hard, soft and non real-time messages. The rest of the priority field can be predetermined in the channel creation phase or determined dynamically based on message deadlines. The second field of the message is the channel ID. For *one-to-X* channels this is enough for a unique message ID value across messages from different nodes competing for network. If *many-to-X* channels are required in system either node ID must be included as field of message ID, thus decreasing range of possible priorities, or more efficiently Channel IDs should be assigned separately for each involved sending process. Many-to-one communication can be triggered either by replica agreeing on group communication channel, or on multiple processes producing data for same consumer in turn. Each packet is sent separately over network and reception of first packet, as well as reception of whole message, has to be acknowledged. Since CAN has powerful error detection mechanism and automatic message retransmission, packets in between don't have to be acknowledged. Rendezvous process synchronization is ensured by delaying the acknowledgment of first packet until receiver task is ready and blocking sending task until last packet is acknowledged. If timeout expires and receiving process is still not ready, message transmission is aborted. If this is the case, obvious advantage is more efficient use of network bandwidth, since only first packet is sent.

At receivers end, packets are sent to FIFO inside input channel. When all packets arrive, receiving process is released and is using passive link driver objects to join them into stream, which is then unflattened to object. If needed, conversions to local format of C respectively C++ data types is done.

F. Ideas for future work

Next step is to implement the CT kernel on other chosen fieldbuses and make hard real-time industrial-like demonstrators. This should result in further improvements in reconfiguration and scalability issues, in minimizing communication overheads and better adoption to hard real-time systems needs.

Remote method invocation can be implemented through *call channels*. They accept function ID of service and all parameters, block client process and initiate operation at server side. When operation is executed return value is given to client process and that process is released.

Efficient ways for encapsulating location knowledge, finding shortest route and timely and reliable multicast in heterogeneous networks, where message from channel to its remote proxy can travel over several fieldbuses, should be investigated.

While load balancing is desirable especially in overload situations, mobile processes are not so easy to implement. Here we can differentiate between periodic processes with h-state (history) and periodic processes without h-state. While latter can be easily downloaded on other nodes, appropriate channels and channel-proxies can be easily made, making first category mobile involves problem of saving process h-state. Furthermore if we want to use passive replication this h-state must be recorded on node different from one where process is executing. Best time to capture h-state is on arrival of task, immediately before current job of task begin.

If we want to state that library is deadlock-free then higher layers of library should be built on same CSP construct based organization principles and CSP based formal check of whole library should be done.

Timed CSP theory should be investigated in details and applied if possible.

From implementation point of view, we have to investigate possible advantages of porting library to work under Real-time Java and KVM. KVM is version of Java Virtual Machine tailored for embedded systems by minimizing memory footprint and reducing API.

REFERENCES

[1] G. H. Hilderink, J. F. Broenink, and A. W. P. Bakkers, “Communicating threads for Java,” presented at Proc. 22nd World Occam and Transputer User Group Technical Meeting, Keele, UK, 1999.

[2] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, “A distributed Real-Time Java system based on CSP,” presented at Proc. Third IEEE Int. Symp. On Object Oriented Real-Time Distributed Computing ISORC'2000, Newport Beach, CA, USA, 2000.

[3] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

[4] S. Schneider, *Concurrent and Real-Time Systems: The CSP approach*: Wiley, 2000.

[5] N. J. Wittenmark B., Torngren M., “Timing problems in Real-time control systems,” presented at American control conference, Seattle, 1995.

[6] A. Cervin, “Analyzing effects of missed deadlines in contol systems,” presented at ARTES Graduate Student Conference 2001, Lund, Sweden, 2001.

[7] G. C. Buttazzo, *Hard real-time computing systems: Predictable Scheduling Algorithms and Applications*. Pisa, Italy: Kluwer Academic Publishers, 2002.

[8] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems, EDF and related algorithms*: Kluwer Academic Press, 1998.

[9] K. Tindell and J. Clark, “Holistic Schedulability Analysis for Distributed Hard Real-Time Systems,” *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, vol. 40, pp. 117--134, 1994.

[10] H. Kopetz, *Real-Time Systems, Design principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, 1997.

[11] O. Redell, “Global Scheduling in Distributed Real-Time Computer Systems, An Automatic Control Perspective,” Dept. of Machine design, KTH 1998.

[12] M. A. Livani and J. Kaiser, “EDF Consensus on CAN Bus Access for Dynamic Real-Time Applications,” presented at 6th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS ´98), Orlando, FL,USA, 1998.

[13] D. C. Schmidt, D. L. Levine, and S. Mungee, “The design of the TAO real-time object request broker,” *Computer Communications, Elsevier Science*, vol. 21, 1998.

[14] M. Beveridge, “Jini on the Control Area Network (CAN): a case study in portability failure,”: Carnegie Mellon University, 2001.

[15] -, “RoSES” http://www.ece.cmu.edu/~koopman/roses, 2002.

[16] A. Pasetti, *Software Frameworks and Embedded Control Systems*. ETH-Zentrum, Zurich, Switzerland: Springer-Verlag, 2002.

[17] F. Singhoff, “Fieldbus Networks: An Overview” http://beru.univ-brest.fr/~singhoff/DOC/PAPIER_A_TRIER/tese-c3.pdf, 2002.

[18] -, “CAN at Bosch” http://www.can.bosch.com, 2002.

[19] -, “CAN in Automation” http:///www.can-cia.de, 2002.

[20] -, “Profibus” http://www.profibus.com, 2002.

[21] -, “Worlfip” http://worldfip.org, 2002.

[22] -, “LONworks” http://www.echelon.com, 2002.

[23] R. Pietruszkiewicz, “An Introduction into Fieldbus Networks” http://www.maintenance.org.uk/Intranet/Index.html: The University of Manchester - Fieldbus Team, 2002.

[24] K. Tindell, A. Burns, and A. Wellings, “Calculating Controller Area Network (CAN) message response times,” presented at IFAC Workshop on Distributed Computer Control Systems, Toledo, Spain, 1994.

[25] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues, “Fault-Tolerant Broadcasts in CAN,” presented at Symposium on Fault-Tolerant Computing, 1998.

[26] L. M. R. d. S. Pinho, “A Framework for the Transparent Replication of Real-time Application,”: University of Porto, 2001.

[27] K. M. Zuberi and K. G. Shin, “Design and implementation of efficient message scheduling for Controller Area Networks,” *IEEE Trans. on Computers*, vol. vol. 49, pp. pp. 182-188, 2000.

[28] K. Kim, G. Jeon, S. Hong, T.-H. Kim, and S. Kim, “Integrating Subscription-Based and Connection-Oriented Communications into the Embedded CORBA for the CAN Bus,” presented at Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), Washington, D.C., 2000.

[29] J. Kaiser and M. Mock, “Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN),” presented at 2nd Int'l Symposium on ObjectOriented Distributed Real-Time Computing Systems,, San Malo, 1999.

[30] P. H. Welch, “Java Threads in the Light of occam/CSP,” presented at Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21, Amsterdam, 1998.

[31] P. H. Welch, J. R. Aldous, and J. Foster, "CSP networking for java (JCSP.net)," presented at Computational Science - ICCS 2002, 2002.

[32] P. H. Welch, M. D. May, and P. W. Thompson, "Networks, Routers and Transputers: Function, Performance and Application" http://www.cs.ukc.ac.uk/pubs/1993/271, 1993.

[33] D. Milicev, "Object-oriented programming and real-time (in Serbian)," University of Belgrade 1998.

[34] S. Vranjes, "Programming and Real-time (in Serbian)," University of Beograd 2000.