Patterns for High Availability Distributed Control Systems

Johannes Koskinen

Department of Software Systems
Tampere University of Technology
Finland

{firstname.lastname}@tut.fi

1 Introduction

The patterns presented in this paper are part of a larger pattern language that is currently formed in collaboration with large global machine control companies. The patterns have been collected from the real-life systems using architecture evaluations and interviews. The previous version of the language is available in [3].

A control system is a device, or set of devices to manage, command, direct or regulate the behavior of other devices or system[1]. In this paper by an embedded control system we mean a software system that controls large machines such as harvesters and mining trucks. Such systems are often tightly coupled with their environment. For example in case of a harvester, harvester head hardware needs special-purpose applications to control it. In a distributed control system, the system is divided into subsystems with each controlled by one or more controllers. These controllers are connected by networks.

2 Patterns

In this section a set of patterns from the pattern language (refer Fig. 1) is presented. The selected patterns for the paper are VOTING, STATIC RESOURCE ALLOCATION and STATIC SCHEDULING. The pattern language graph could be seen as a designer's map for solving design problems. The design process begins so that the first pattern to be considered is CONTROL SYSTEM in the middle of the graph. After the designer has made the design decision to use the pattern, she may follow the

[1] http://en.wikipedia.org/wiki/Control_system

arrows to the next patterns. An arrow means that the following pattern can be applied in the context of the resulting design from the old pattern. In other words, the subsequent pattern refines the design. However, a single pattern may be used regardless of the usage of previous patterns if the context of the pattern matches the current design situation.

The patlets for the patterns as well as the referenced patterns not included to this paper are presented in Table 1.

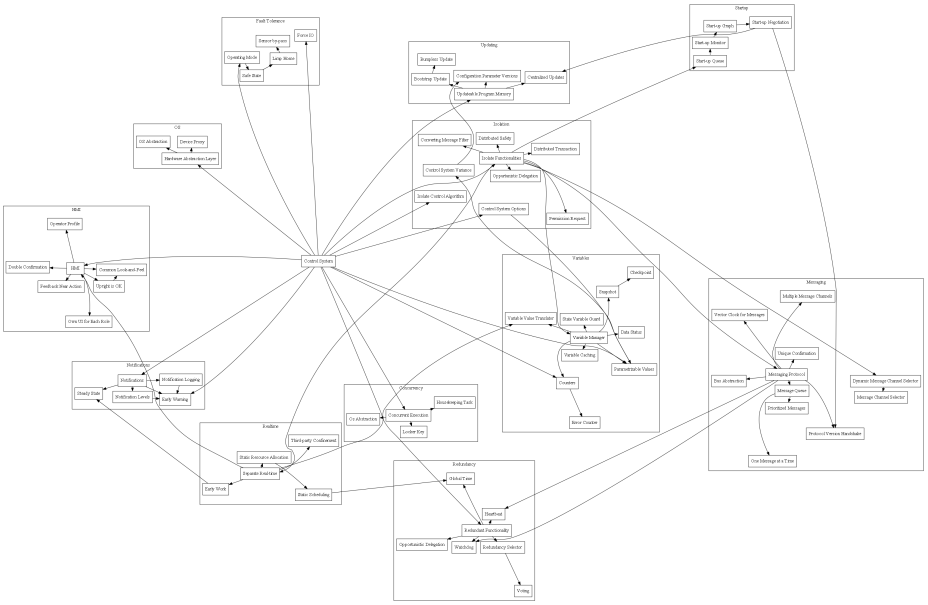


Fig. 1. Pattern language for distributed control systems.

Table 1. Patlets for the included and referenced patterns.

Pattern	Patlet
Control System	How to implement a work machine that offers interoperability between systems and is highly operable with good performance? Therefore: Implement control system software that controls the machine and can communicate with other machines and systems.
Global Time	How to prevent different nodes on the system from getting out of sync? Therefore: Use external clock, e.g GPS or atom clock, to synchronize all the nodes.
Early Work	How to execute resource-greedy tasks which may require more resources than are available after normal operating mode is established? Therefore: The processes should be prepared (like pre-calculating values to be used later on) in system startup, because there are usually easier to dedicate CPU time, memory, etc for heavy calculations than later on.

Error Counter	How to distinguish substantial faults from false alarms or transient faults? Therefore: Create a counter which threshold can be set to certain value. Once the threshold is met, an error is triggered. The error counter is increased every time a fault is reported. The counter is decreased or reseted after certain time from the last fault report has elapsed.
Limp Home	When a part of the machine is malfunctioning, how to still operate the machine to some extent? For example, to drive the machine from forest to the nearest road. Therefore: Divide the sensors and actuators into groups according to the high level functionalities, such as drive train, boom operations etc.. The groups may overlap. Malfunctioning device only disables the groups it belongs to and the other groups remain operable.
Notifications	How to inform operator or communicate to subsystems that something worth of noticing has occured in the control system? Therefore: Communicate noteworthy or suspicious state changes in the system using notifications. Implement also Notification service that is used to create, handle and deliver notifications.

Redundant Functionality	How to ensure availability of a functionality even if the unit providing it breaks down or crashes? Therefore: Clone the unit controlling a critical functionality. One of the units is active and other is in a hot standby mode. If the controlling unit fails, hot standby mode unit takes over controlling the functionality.
Redundancy Selector	How to remedy the failure of a redundant unit when activation time of the hot standby unit is too long? Therefore: Design the system so that all redundant units are active at the same time. Add a monitoring component that takes outputs of all redundant controlling units as inputs and examines the output and chooses which unit's output is forwarded as a control signal.
Separate Real-time	How to offer high-end services without violating real-time requirements? Therefore: Divide the system into separate levels according to real-time requirements: e.g. machine control and machine operator level. Real-time functionalities are located on the machine control level and non real-time functionality on the machine operator level. Levels are not directly connected, they use bus or other medium to communicate with each other.

Start-up Graph	How to determine an optimal start-up sequence when the system setup may vary? Therefore: Design the start-up graph of devices and their dependencies, start-up times and resources requirements during start-up and during normal use. The start-up monitor component determines the system setup and constructs the correct system start-up sequence.
Static Resource Allocation	How to make sure that the critical services will always have the resources needed? Therefore: All the resources needed for critical services should be pre-allocated during the system startup. The resources are never deallocated afterwards (i.e. the resources are fixed for the service).
Static Scheduling	How to schedule real-time processes efficiently with low overhead? Therefore: Divide the system into executable blocks. The executable blocks can be e.g. applications, functions, or code blocks. Scheduler has time slots of fixed lengths. The developer or compiler divides the blocks into these time slots, thus scheduling the program statically.

Watchdog	How to make sure that a node crash does not go unnoticed? Therefore: Add a watchdog component to each node to monitor the behavior of the application(s). If the application does not react within a given time limit, the watchdog deems the node to malfunction and starts remedying actions, such as reboot.
Voting	How to make reliable decisions in a distributed system where high correctness is required? Therefore: Create redundant nodes calculating the control output.

2.1 Voting

…there is a CONTROL SYSTEM with high availability and correctness requirements. However, there might be situations, when the functionality of a node may fail. REDUNDANT FUNCTIONALITY pattern is used to ensure high availability of functionality even in case of broken node. Still, we need to ensure correctness of the decision made by the control nodes. For example, a failure in single sensor should not affect node's output value.

How to make reliable decisions in a distributed system where high correctness is required?

The value of the control output should not be affected if a node providing it malfunctions or crashes.

There is no time available for off-line fault diagnosis and recovery actions, the other units should be used to mask and detect a fault in one of the hardware unit.

Decisions should be based always on the correct information. Potentially dangerous situations can occur if the information is not correct because of the failure in a node or a sensor. Usually, a fault tolerance in a control system is achieved through redundancy in hardware.

It should be possible to add redundant units later on, for example, if the requirements for the correctness are changed. In addition, the system should be scaled up and down depending on the current target usage.

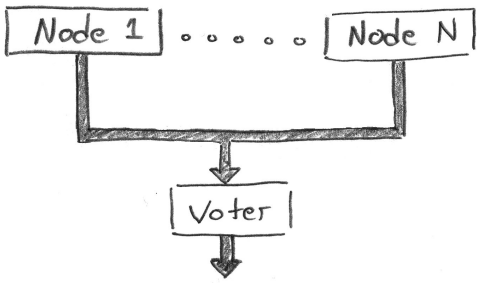
Therefore:

Create redundant nodes calculating the control output.

✣ ✣ ✣

In addition, a unit that collects the outputs from these redundant nodes should be added to the system. The outputs of the nodes should be comparable. The collecting node then deducts which output value has got the most "votes" and gives that forward as control output of the (sub)system. Simple disagreement of the output indicates a fault, but cannot identify the faulty node nor the correct output.

To achieve continuous operation and correct output, the majority vote of the outputs of three or more identical nodes are used. Simple two out of three votes masks single-module failure. The collecting unit (Voter) accepts all N control outputs as inputs and uses these to determine the correct, or best, output if one exists. In addition, if the faulty node can be identified, NOTIFICATIONS or ERROR COUNTER can be used to inform the rest of the system about the node.



A significant problem with voting is that the output values must be comparable. With switches this is easy as the switch is either open or closed. On the other hand, with analog signals, inputs of the Voter can be regarded as the same if the input values are close enough. The output of the Voter is then generated based on the inputs, for example, using average of the correct inputs.

If the input of the output is incorrect, the output is also incorrect, even if the node is fully functional. Thus, the sensors should also be

replicated to get correct inputs for the redundant units. It is also possible to have different versions of the software in each unit, so that the failures in software can also be masked. Moreover, the Voter can take the reliability of each node into consideration and determine the most likely correct answer. However, this will require knowledge of each node and/or additional acceptance or self-checking tests. For example, it might be known that the output of the value should be inside some predetermined range.

The voter component creates a single point of failure and WATCHDOG should be used to monitor a voter node. LIMP HOME can be used to select an output value from one of the redundant nodes if the voter node is broken. REDUNDANCY SELECTOR pattern can be used instead of this if there is no need for voted values. The selector selects one of the input values to be an output value. The selection criterion is simpler than with voting, for example, the first input with some kind of value is always selected.

Failure in one unit will not cause an erroneous output signal. More than one node is generating the output signal and faulty values are left out by the voting mechanism.

Because of the redundancy in the system, the output is always available, even if one of the nodes fails.

Self-checks can be used to improve voting process. If the nodes indicate some kind of failure in their software or components, the information can be take into consideration when the output value is decided.

With voter, it is possible to detect the malfunction in a single unit and replace the broken unit with a good one. On the other hand, redundant units and sensors increase system costs and take more space.

Synchronization of the units may be difficult and communication between redundant units increase complexity of the system.

The voter does not communicate directly with the nodes. In this way, it does not affect the operation of active unit. However, the voter component creates a single point of failure. Therefore the voter component should be robust and simple enough in order to be reliable.

A machine control system has its own vehicle identification number (VIM) 123. The identification number of the system is stored to the

nodes. After service operation, a node from some other machine (with VIM 234) is moved to our machine. The node contains still the vehicle identification number 234, when the system is started up. During startup, the voting procedure is used to identify the current VIM of the machine. Each node tells its understanding of the machine´s VIM and the VIM having the most of the votes (in this case 123) is selected. The identification number is updated to all the nodes.

2.2 Static Scheduling

…there is a CONTROL SYSTEM where SEPARATE REAL-TIME has been used to divide the functionality to high end functionality and real-time control functionality. With scheduling, the processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs. The processes have strict real-time constraints with hard deadlines. Execution times of the processes and timing constraints are known beforehand. In real-time systems, a process's response time to a certain event must be met, regardless of system load, otherwise the execution of the process fails. In other words, real-time computations can be said to have failed if they are not completed before their deadline. In case of hard deadlines, missing the deadline will cause the whole system to fail.

How to schedule real-time processes efficiently with low overhead?

With limited processor power usually found in embedded systems, the scheduling process should be lightweight and run-time overhead of scheduling should be small. Selecting next process to run should cause as little run-time overhead as possible.

The most important requirement of a hard real-time system is predictability as missing the deadline will cause the whole system to fail.

The number of the real-time processes is known beforehand.

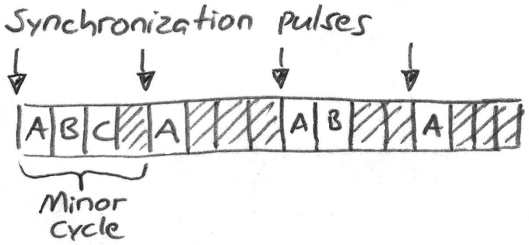
Therefore:

Divide the system into executable blocks. The executable blocks can be e.g. applications, functions, or code blocks. Scheduler has time slots of fixed lengths. The developer or compiler divides the blocks into these time slots, thus scheduling the program statically.

If the timing properties of all the processes are known a priori, the schedule of the processes can be built statically during compile time based on the prior knowledge of execution times, precedence and mutual exclusion constraints, and deadlines. Once a schedule is made, it cannot be changed in run-time. On the other hand, run-time overhead of scheduling is very small as the scheduling decisions are made beforehand. In addition, when the scheduling is done based on worst-time estimations of the execution times, the processes cannot be overrun. The compiler should give an error message, if the scheduling is not possible.

The processes are divided into executable blocks, which has their own execution period time (e.g. every 10 ms). The schedule has a main cycle (e.g. 100 ms), which is divided into minor cycles (e.g. 10 ms). The compiler divides the blocks into minor cycles so that each minor cycle executes code from the execution blocks based on their period time. This is done by calculating the worst-case execution times for every execution block using prior knowledge of the execution times for the operations carried out in the block. The scheduling is synchronized using synchronization pulse (like external clock interrupt), which starts each minor cycle.

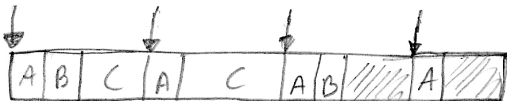
For example, let us have processes A, B, and C. The process A has a period time 40 ms, B 80 ms and C 160 ms. The main cycle is 160 ms and it is divided into four 40 ms minor cycles. Thus, the starting minor cycle will have code from all the processes and rest of the minor cycles contains code from the processes depending on their period time. As the process A has period time of 40 ms, its code is executed in each minor cycle. Process B has code in every second minor cycle and C only in the first cycle. If all the CPU time is reserved for the processes, there cannot be other external interrupts except synchronization pulse, but the CPU-usage is very high.



If the system should have interrupts, the worst-case execution time for the interrupt handling should be known beforehand and for each

minor slot, the corresponding free time for the interrupt handling should be reserved. On the other way, the interrupt can set a flag and the actual interrupt processing can be done periodically in pre-reserved time slot.

If the scheduling is preemptive (i.e. one process can interrupt the other) and based on priorities, the execution time of the process can longer than one minor cycle. In this case, the processes with shorter period time have higher priority and they will interrupt other processes. For example, if the execution time of process C is 50 ms, it will be distributed into two minor cycles. In the beginning of the second minor cycle, process A interrupts process C and it is executed first.



In similar way, it is possible to schedule dynamic, non time-critical processes in addition to time-critical, statically scheduled processes. The static scheduler is run in a single high priority task, which preempts the lower priority dynamic tasks. In each minor cycle, there should always be some time available for dynamic processes assuming that no interrupts has been occurred. On the other hand, CPU utilization is usually worse than pure static scheduling.

One can use GLOBAL TIME to synchronize execution between multiple units for increased accuracy.

✣ ✣ ✣

When the processes are scheduled statically, run-time overhead is small as all the scheduling decisions are done beforehand. In addition, there is no need to make preparations for unsuccessful scheduling, as there should always be enough CPU time.

As the scheduling is based on worst-case execution times, the process will never overrun.

Scheduling is predictable as the time slots are fixed and the number of the process is always same.

On the other hand, with pure static scheduling, dynamic processes cannot be created during run-time. However, high CPU utilization is easy to achieve.

It might also be impossible to calculate execution times for operations in all cases. If the worst-case estimations are used, it will lead lower CPU utilizations.

✣ ✣ ✣

Processes are scheduled statically in a power plant control system. For each process, the total execution time is calculated based on execution time of each instruction and measured timing information for operations. When the processes are compiled, the scheduling is defined and the code of the processes is divided into execution blocks, which worst-case execution time is the same as the minor cycle time. Each minor cycle executes code from one block. During one major cycle, all the blocks get executed. In this way, all the processor time is used to execute processes and CPU utilization is very high.

2.3 Static Resource Allocation

…there is A CONTROL SYSTEM with a node having several processes. At least some of these processes provide critical services that are essential for the system functionality. Such a service should always be available and could be e.g. emergency message handling, which sends an emergency EMCY message to the main node via bus in case of failure. Critical services usually background processes, which are triggered by a certain event like a failure in the system, thus making the exact execution moment unpredictable. However, the critical services should always be available. In addition, these kind of services usually have strict real-time response time constraints and thus the real-time part is separated with SEPARATE REAL-TIME.

How to make sure that the critical services will always have the resources needed?

In embedded control systems, there are usually limited amount of the resources (like memory, bus bandwidth, processing power) available to be shared for all the processes. Still, there must always be required resources for the critical services.

The resources required by the critical services should be available immediately and there might not be time for waiting other processes to free the desired resources. As the critical services are usually triggered

by an event, there usually is no time for resource allocations or initializations.

Nondeterministic timing of the dynamic resource allocation makes it hard or even impossible for a service to meet strict real-time constraints as the allocation can take more time than available for the service.

Therefore:

Pre-allocate all the resources needed for critical services during the system startup. The resources are never deallocated afterwards (i.e. the resources are fixed for the service).

Basically, static resource allocation is simple to implement. The critical services are started using START-UP QUEUE or START-UP GRAPH. All the necessary allocations and initializations can be carry out during system startup, maybe with EARLY WORK. When entering in normal operating mode, the services will not allocate any additional resources nor they will free any pre-allocated ones.

It is very important that static resource usage is minimized to be as low as possible. EARLY WORK can be used to decrease statically allocated resources if all the preparations are carried out during the system startup and only the resources required by the core functionality of the service are statically allocated.

Larger services should be divided into two smaller ones, if possible. One of the parts will provide the critical service with static resource allocation and the other containing rest of the service. The latter part will not require any fixed resources for its functionality. This is even more important as resources required by more than one service cannot be reserved statically just for one critical service. For example, message handling service should be divided into two new services, one for emergency messages and the other for regular messages. As the emergency message service should always be available, it has all the resources (such its own messaging slots and queues) pre-allocated.

To statically allocate CPU-time for the critical services, one may want to use STATIC SCHEDULING to share processor time statically for each service. To allocate memory statically, FIXED ALLOCATION [1] or STATIC ALLOCATION PATTERN [2] can be used.

When all the required resources are fixed for a critical service, it can never run out of the resources.

As the resources allocated by the critical services are not available for the other services, fixed resource allocation means usually increased resource requirements.

The response times are faster as the service will not have to wait the resources to be deallocated by the other processes.

Allocating resources statically increases also speed of the critical services as the time required by the allocation and deallocation is done in the system startup. When cost of having additional resources (such as memory) is relatively small, this pattern can be used for all services to increase speed of the whole system. In addition to speed, also predictability of the run-time execution is increased as the nondeterministic timing of the resource allocation is left out.

✣ ✣ ✣

In a control system, nodes send emergency messages (i.e. EMCY messages) to other nodes via bus in case of fatal error. The bus capacity is divided into time slots, each containing one message. Before the message can be sent, a sender allocates one slot for the message. To ensure messaging capacity for EMCY messages, one slot is statically reserved for such a critical messages. As the slot is used only for critical messages, it is always available and immediately ready to use in case of fatal error.

3 Acknowledgements

I want to thank my colleagues Ville Reijonen, Marko Leppänen and Veli-Pekka Eloranta for their help. In addition, I want to thank all industrial partners for their valuable cooperation in our pattern mining process: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D.

4 References

[1] Noble, J., Weir, C.: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)

[2] Douglass, B.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley Professional, USA (2003)

[3] Eloranta, V-P., Koskinen, J., Leppänen, M. and Reijonen, V.: A Pattern Language for Distributed Machine Control Systems, ISBN 978-952-15-2319-9, Tampere University of Technology, Department of Software Systems. Report, vol. 9, Tampere University of Techology, pp. 108, 2010.