Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 9
Tampere University of Technology. Department of Software Systems. Report 9

Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen & Ville Reijonen

A Pattern Language for Distributed Machine Control Systems

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos
Tampere 2010

ISBN 978-952-15-2318-2 (printed)
ISBN 978-952-15-2319-9 (PDF)
ISSN 1797-836X

Foreword and Acknowledgements

An embedded system is devised to control, monitor or assist the operation of equipment, machinery or plant [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, harvester head hardware needs special-purpose applications to control it. Tight coupling implies specific requirements – such as distribution, real time, and fault tolerance – to software, to be taken into account in the design of software architecture. As the embedded control systems have become larger, the software architecture of these systems plays a crucial role in the overall quality of the products. Yet, there is little systematic support for designing such architectures. It was noted that a pattern language specifically targeted for this domain could significantly assists software architects in designing high-quality systems.

During years 2008 and 2009, we have visited four sites of Finnish machine industry to identify design patterns specific to this domain. The target companies are global manufacturers of large machines and highly specialized vehicles intended for different branches of industry. During the visits, patterns were identified in the context of an architectural assessment of machine control systems provided by the companies. This work is a collection of all discovered patterns, presented in a form of a pattern language. Because the fundamental goal was to create a pattern language for this domain, essential solutions were captured and documented as patterns regardless of their prior existence in other pattern collections. The pattern collection process in described in more detail in [2].

We would like to thank our industrial partners Areva T&D Finland, John Deere Forestry, Kone, and Sandvik Mining and Construction. They provided all the necessary information to make this pattern language possible. We would like to thank also VikingPLoP 2008 and EuroPLoP 2009 workshop members for valuable feedback. Special thanks to our professors Ilkka Haikala, Hannu-Matti Järvinen, Kai Koskimies, and Tommi Mikkonen. Additionally, we would like to thank our colleagues Tero Ahtee, Jari Peltonen, Outi Räihä, and Mikko Vartiala who gave us valuable input. Finally, our appreciation goes to Jim Coplien who guided us to the path of patterns.

Table of Contents

1 Pattern language

The pattern language for distributed machine control systems consists of 45 patterns that are presented in a graph form in Fig. 1. 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 ISOLATE FUNCTIONALITIES in the middle of the graph. After the designer has made the design decision to use the pattern, she may follow the arrows to the next patterns. An arrow means that the following pattern can be applied in the context of the resulting design from the previous 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 language construction was carried out so that the patterns were at first grouped loosely together based on their target area. Patterns affecting the design on the higher level were grouped together and patterns with smaller impact were put together depending on which part of the system they improved. This gave an overview of the pattern relationships. The relationships between patterns were refined iteratively as the language grew. The resulting language has seven separate branches which hold patterns for special issues of the target domain. These branches are: messaging related patterns, patterns dealing with separation of real-time parts of the system, fault tolerance patterns, patterns concerning redundancy, system state related patterns, patterns for system configuration and updating, and operating system related patterns. Each branch is presented in its own section with a short descriptions of the patterns, i.e. pattern thumbnails, and separate sublanguage figure. Additionally, Appendix 1 contains all the pattern thumbnails in alphabetical order.

Following template is used to present the patterns. First, *Context* where the pattern can be applied is introduced. The next section is the description of *Problem* that the pattern will solve in the given context. In *Forces* section the motivation or goals that one may want to solve by applying this pattern are described. Furthermore, *Solution* is given and argument for it in *Consequences* section. *Resulting Context* will be the new context after applying the pattern. Finally, *Related Patterns* are presented and in *Known Usage* one known case of usage is given.

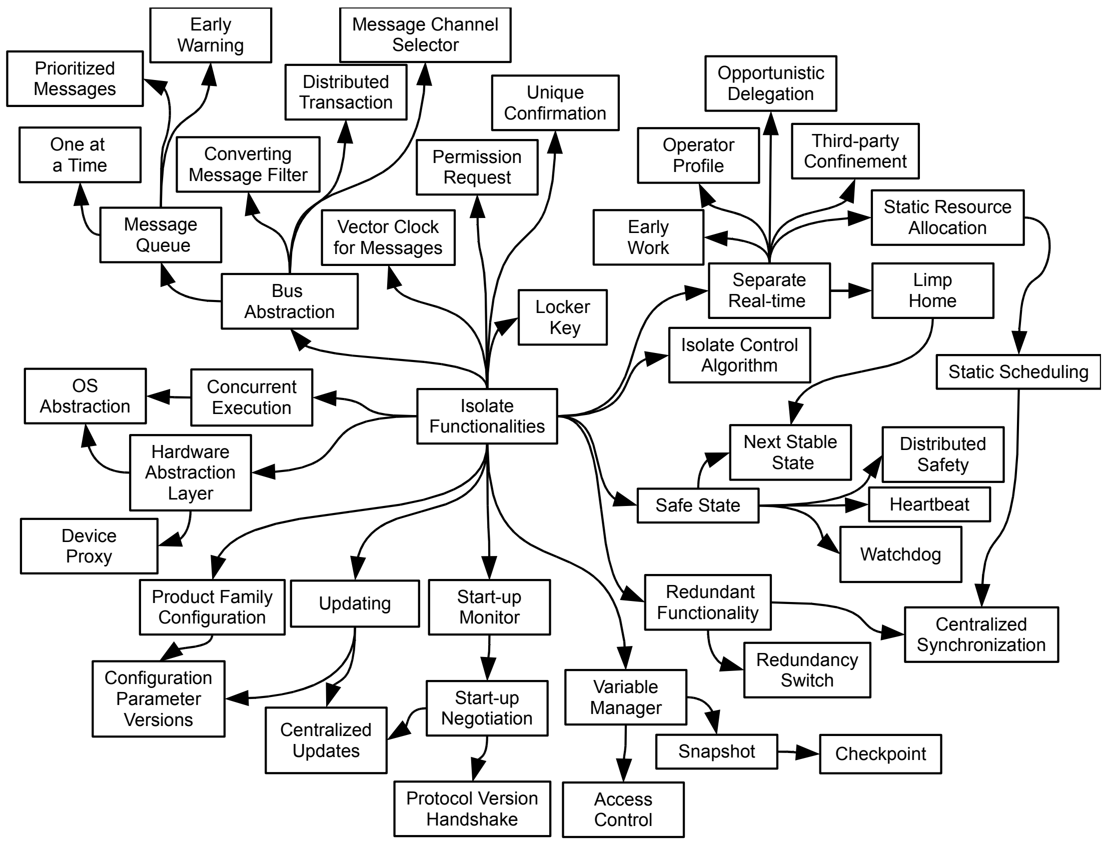


Fig. 1. The pattern language for distributed machine control systems in a graph form.

2 Patterns for Messaging

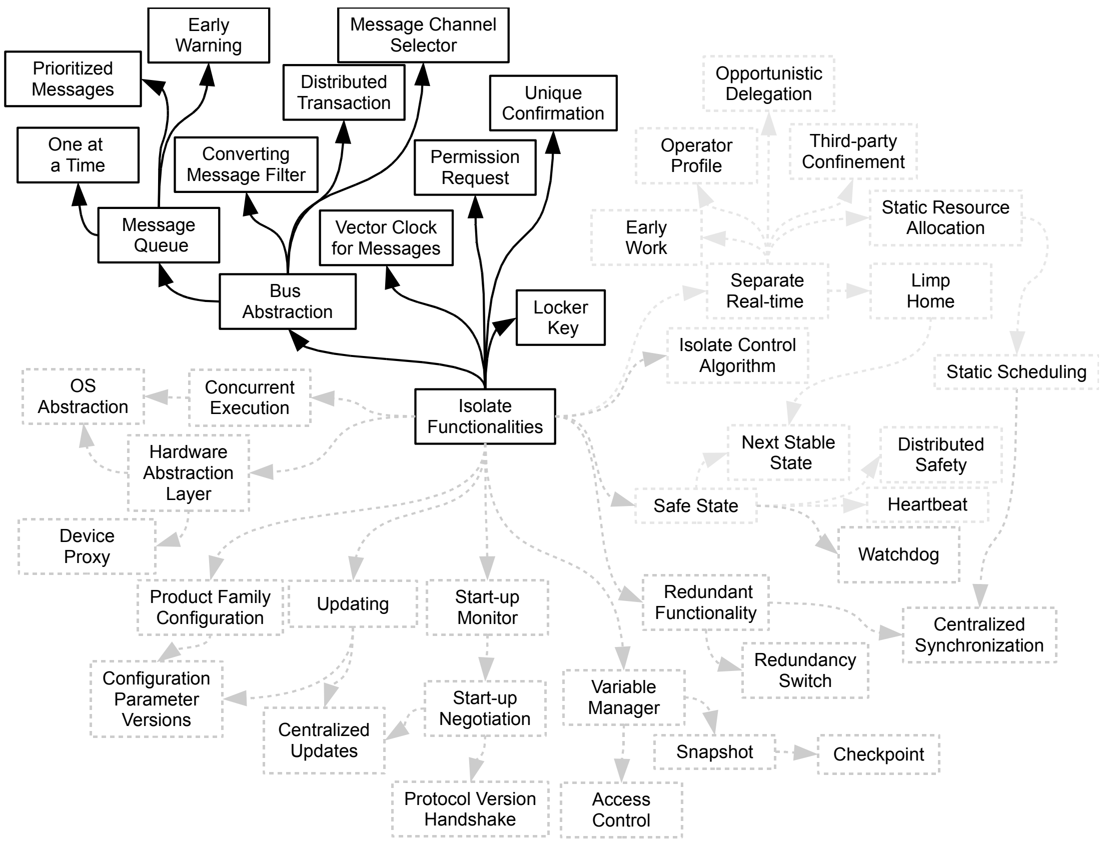


Fig. 2. The sublanguage for messaging in distributed machine control system.

Table 1: Pattern thumbnails

Pattern Name	Description	Page
ISOLATE FUNCTIONALITIES	The solution divides a system into subsystems according to functionalities. The subsystems are connected with a bus.	9
BUS ABSTRACTION	Nodes communicate via a bus using messages. The message bus is abstracted so the bus can be changed easily and the sender does not have to know the actual location of the recipient.	11
MESSAGE QUEUE	Nodes communicate asynchronously and the amount of messages that can be simultaneously processed is limited. Therefore, the rest of the messages have to be queued.	13
ONE AT A TIME	A node is prevented from flooding the bus with babble by limiting the amount of messages on the bus.	15
PRIORITIZED MESSAGES	Messages can have varying importance and hence some messages need more urgent attention. Therefore, important messages should be handled first.	17
EARLY WARNING	The solution introduces a way to make sure that possible buffer overflow situations can be detected before they occur.	19
CONVERTING MESSAGE FILTER	In a system with multiple messaging channels the messages are filtered according to the channels where the interested recipients reside. In some cases this might need conversion from a message format to another.	21
DISTRIBUTED TRANSACTION	An action requested over a bus consists of several logically connected command messages and it should be executed successfully as a whole whenever possible.	23
MESSAGE CHANNEL SELECTOR	There are several different message channel types in the system. A separate component selects the appropriate message channel for each message.	26

Continued on next page...

Table 1: (Continued)

Pattern Name	Description	Page
VECTOR CLOCK FOR MESSAGES	The order of events can be deduced by using a vector clock to timestamp events instead of normal timestamps which have clock skew in a distributed system.	28
PERMISSION REQUEST	To prevent conflicting actions in a distributed there must be a component that decides which actions can be taken.	30
UNIQUE CONFIRMATION	The default confirmation response must not be same twice, in order to eliminate the change of accidentally repeating confirmations causing unwanted actions.	32
LOCKER KEY	The solution provides an efficient way for communicating between processes while avoiding dynamic memory allocation.	34

2.1 Isolate Functionalities

Context

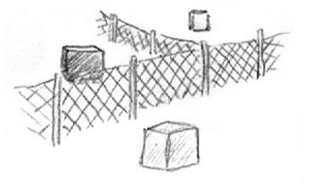
An embedded control system is needed to control a large machine which consists of different kinds of sensors, actuators and controlling devices.

Problem

What is a reasonable way to design embedded control system for a large machine?

Forces

- *Resource utilization*: Available resources should be used efficiently.
- *Analyzability*: It is easier to understand smaller and less complex entities.
- *Extendability*: It should be easy to add new functionalities.
- *Variability*: It should be possible to create different products from the same base system.
- *Testability*: Smaller entities are easier to test.
- *Reusability*: Same components can be reused in different products.
- *Subsetability*: The implementation of different kinds of functionalities may need different kinds of special expertise.
- *Cost efficiency*: Extensive wiring is expensive.
- *Cost efficiency*: High end components cost significantly more than low end components.
- *Fault tolerance*: Extensive wiring is more likely to break.
- *Fault tolerance*: Monolithic software in single device creates a single point of failure.



Solution

The system is divided into subsystems according to functionality. These subsystems can be placed on separate devices. For example, there can be separate controller for drive engine, frame, hydraulic pressure, boom, etc. As a controller takes care only its functionality, the hardware requirements are not so high. Therefore, a suitable hardware can be also chosen from low end components. In this way every functionality has just enough resources without wasting them. The controlling device should be located close to the actuators and sensors it is using. This results in less extensive, cheaper and less error prone wiring.

The devices are interconnected and they function as nodes on the bus, abstracting the physical location of the devices. The bus allows easy extendability as new nodes can be added without additional wiring. One of the most commonly used bus standard in large machines is CAN [3]. The communication on the bus takes place using a common message format. For example, CANopen [4] and J1939 [5] are commonly used standards for messaging on top of CAN bus. The throughput of the bus limits the amount of nodes and messages that can be on the bus at the same time. Standard bus components are cost-effective and well available. A bus is usually implemented with single well shielded wire that is less probable to break than a set of wires.

It is possible to test the device as a standalone subsystem, as it implements only certain functionality. This also makes the system easier to understand for developers. Implementation of some functionalities such as boom kinematics needs special expertise. The experts do not need to worry about the whole system as they can concentrate only on the problem area. When the same functionality is needed in some other product, the same device and software can be reused.

Consequences

+ Functional division makes system more understandable and manageable.
+ New nodes can be added easily within the limits of the message bus capacity.
+ Nodes do not depend statically on each other, but only on the message format.
– The capacity of the message bus limits the amount of nodes.
– Throughput may be compromised due to heavy message traffic.
– It may be difficult to know where functionality resides because the location is abstracted by the bus.
– Changing the bus implementation may require changes in several devices.
– A single bus wire can still break, even if the possibility is smaller than with a set of wires.

Resulting Context

The pattern results in a distributed and scalable embedded machine control system where nodes communicate with each other via a bus using a common message format. This pattern forms the base for a distributed embedded control system for a large machine.

Related Patterns

MESSAGE CHANNEL [6] and MESSAGE BUS [7] describe similar mechanisms for decoupling nodes but in different domains.

Known Usage

Programmable Logic Controllers (PLC) are used as controlling units in a drilling machine. These controlling units are connected to sensors and actuators required for the functionality such as measuring drill hole depth. Every unit is a node on the bus that interconnects them. In the drilling machine CAN bus is used as it is common bus solution for this domain. High-level communication protocol is provided by CANopen, removing the need to design it in-house. In addition, COTS (commercial off-the-shelf) components supporting CANopen are readily available. The system is divided so that each functionality has its own controller. For example, there are separate controllers for boom and drill.

2.2 Bus Abstraction

Context

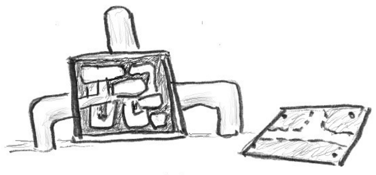
A control system has been distributed using ISOLATE FUNCTIONALITIES pattern. The system consists of multiple device nodes. There is a bus connecting the nodes together.

Problem

How to change bus standard or bus topology so that main application code does not need to be changed?

Forces

- *Scalability*: The system may incorporate many nodes and a lot of communication may emerge between them.
- *Flexibility*: Bus topology should be hidden from the developers.
- *Flexibility*: Developers should not need to know where certain services reside.
- *Modifiability*: The system configuration of nodes may change, even during the run-time of the system.
- *Reusability*: The same bus abstraction could be used by other nodes.
- *Portability*: During the life cycle of the product, the physical bus may change.



Solution

Over long life cycles it is often necessary to change the topology or used bus. When every application communicates directly to the bus, it will lock the developers to the selected bus and also to the bus topology. Changes in topology may require changes in various places in the code of every node. Additionally, network topology or bus standard complexities have to addressed in multiple times. In all, the situation will create unnecessary redundant code. To remove this extra code, a common application programming interface (API) is constructed to provide uniform messaging functionality. When API is taken into use, the application has to provide a hook function for the API. API will use the given function to give received message to the application.

The main task for API is to hide how the bus actually works from the applications and application developers. Therefore, as API hides the actual bus implementation, the bus standard or bus topology can be changed while API will stay the same. A change may require modifications to the API abstraction code, but not to the interface. Afterwards this same code can be used everywhere else in the system. When the bus standard and topology is abstracted, the addresses and locations of different services can not be known. Therefore, it is necessary name the services so they could be used. Developers will use these names to point to required services.

As names are used instead of addresses, it may be difficult to know for the developer if service is local and therefore fast in responding or remote and therefore slower with responses. Remote node may also fail to respond due to malfunction or some other error. In this location question hides design issue with responsibility division: is it API's task to retry and announce failure or applications? In this case what should be done with new messages? API developer should remember that there is no reply or receipt for a sent message on all bus types or for all message types.

The names may be fixed inside the API if the system nodes consist from a standard set. Alternatively API may use a naming service to resolve corresponding node and its address. Additional bus traffic should be avoided, so the API implementation should not query names over the bus too frequently. The reason for avoiding bus traffic is twofold. First, sending an query, waiting and finally receiving an answer will delay sending the message to the recipient. For second, extra bus traffic may add congestion to the bus and delay other messages. Therefore, available names could requested during start up to minimize bus load during normal activity or the query results should be cached.

Consequences

+ Nodes do not depend statically on each other, but only on the names, thus the system is easy to expand and modify, even at run-time.

+ The bus presents an abstraction of the physical world, understandable to software developers.
– Abstraction adds latency to the messaging.
– It may be difficult to know if requested service is local or remote.

Resulting Context
The result is a distributed control system where the bus standard or topology can be changed and the change will not affect the main application code.

Related Patterns
The pattern can be viewed as a MESSAGE DISPATCHER [7], applied to an embedded machine control system with a physical bus or connectivity. The abstraction can be enhanced with MESSAGE QUEUE which will address the problem of multiple simultaneous messages. MESSAGE CHANNEL SELECTOR discusses the case where multiple routes are available for message delivery. CONVERTING MESSAGE FILTER suggest a method to filter unnecessary traffic.

Known Usage
A forest harvester uses CAN bus with J1939 for communication. The bus might be changed to use some other bus standard in the future and therefore the bus is abstracted behind an interface. All possible device names are listed with destination addresses in the interface code, during start-up the current list of devices is received and is used to mark which destinations are available on the system. When an application needs to send message to some other service, it is done through the messaging interface. To send a message recipient name and message is required. The interface will deliver the message asynchronously. When message is received in the other end, the hookup function is called giving the sender and the message as parameters.

2.3 Message Queue

Context

There is a distributed embedded control system uses BUS ABSTRACTION to communicate between the nodes. The nodes should be able to send and receive messages regardless of the load on other nodes. This means that an asynchronous messaging scheme must be used.

Problem

How do you give time to both ends of a message channel to process all messages?

Forces

- *Fault tolerance*: All messages should be processed, because important data should not be lost.
- *Predictability*: The amount of messages sent or received is not known beforehand.
- *Predictability*: Usually the processing order of messages is important. The correct chronological order should be preserved.
- *Resource utilization*: All messages should be processed as soon as possible to avoid congestion.

Solution

The messaging architecture is designed to contain dedicated messaging buffers in the receiving and sending parts. Thee buffers are in effect containers which hold the messages in the order they are queued. The actual implementation may vary, but arrays or linked lists are common solutions.

A mechanism to processes the buffer with the messages must implemented in addition to the buffers. Its duty is to handle the queuing for the messages. The component must have some kind of event or interruption mechanism to notice that there are messages ready to be served. The message queue works in "‘first in - first out"’ fashion (FIFO), so that the oldest message in the queue gets processed first.

In the sending end the processing component fills the buffer with the messages. The message queue processor takes the oldest message in the queue and outputs it to the bus whenever possible. The receiving component reads the messages from the bus in the order they were received as soon as it has time to process them. The receiving end queue processor reads the message from the bus and puts it in the received messages queue. The designer should take care in designing the message queue size limits, as too short queue causes message losses and too long queue may waste resources and cause latencies.

It may be helpful to add some overriding system to the queue, if an emergency situation requires immediate response. The emergency messages can be processed as soon as they are received as the contents of the message queue usually have no use in the emergency situation. However, this causes problems if the system should return to normal operation as soon as the emergency situation is over. This could be remedied using PRIORITIZED MESSAGES patterns.

Consequences

+ The pattern gives the messaging components time to process the messages independently from the physical bus rate.
+ The resource utilization of the bus is improved as congestion peaks can be leveled during the quieter times.
+ The message order is not changed as the queue holds the messages in the order they were put in it.
– There is no easy way to prioritize the messages in FIFO.
– Message queue adds latency if the messaging rate is slower than the processing rate.
– It may be difficult to calculate the correct message buffer size, resulting in either resource waste or buffer overflows.

Resulting Context

The pattern results in a system that has ability to communicate asynchronously and the sending not constrained by the message bus rate as the data is queued.

Related Patterns

EARLY WARNING improves the queue handling properties. This pattern also resembles closely Producer/Consumer idea.

Known Usage

MESSAGE QUEUE is used in a heavy machinery, where nodes communicate via CAN message bus. The CAN hardware holds a buffer that queues the messages that are to be sent or received from the bus. When the application code wants to send a message, it gives it to the driver component which places the message into the queue. The application code does not have to care about the details of the sending. In the receiving end, the driver notifies the application with an interrupt that there are messages in the queue and some action should be taken to process the messages before the queue overruns.

2.4 One at a Time

Context

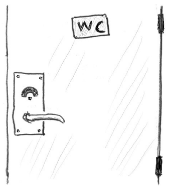
There is a distributed system where Bus Abstraction has been applied. Nodes communicate over bus using some mechanism, for example provided by Message Queue. Nodes may malfunction and start to send a large amount of (unnecessary) messages to bus, i.e. to babble. This kind of malfunction can be hard to detect with e.g. Heartbeat. The babbled messages can also contain erroneous data that may propagate unnecessary actions in other nodes.

Problem

How to ensure that malfunctioning device does not fill up the bus by repeating unnecessary messages? The problem is also known as babbling idiot's problem [8].

Forces

- *Throughput*: Bus load should not have high peaks as they may compromise priority message throughput.
- *Fault tolerance*: Malfunction of one node should not disable the whole bus.
- *Safety*: Critical messages should get through bus in all situations.



Solution

A bookkeeping mechanism is added to all components that communicate over bus. This mechanism keeps track of all messages that are sent to bus, and makes sure that only a certain number of messages of a certain type can be sent to the bus. Number of messages that can be sent to bus can be configurable, so that larger data sets spreading over multiple messages can be sent to the bus. Nevertheless, the number of messages that can be sent should small enough so that the bus will not choke on the amount of messages.

The bookkeeping mechanism must keep track of Time To Live (TTL) of each message and make sure that no more than certain number of messages are sent to the bus. In addition, nodes can give acknowledgement to sender node, so that the bookkeeping mechanism can allow sending of next message of the same type. Sending acknowledgement to broadcast messages can results in problems in form of acknowledge message flood, so it should be avoided. Furthermore, state of the bus should be monitored in some way, for example using Heartbeat, so that the bookkeeping can be sure that no acknowledgement messages are lost. If a node is connected to multiple buses, there should be separate bookkeeping for each bus, i.e. the message sent to one bus does not affect the number of messages that can be sent to another bus.

Consequences

+ Because only limited amount of messages per type at any given time can exists on bus, the bus can not be flooded by babbling nonsense to it.
+ Malfunctioning unit can not send conflicting information continuously, amount of erroneous state changes in other nodes are reduced. Nevertheless, the pattern does not solve this problem completely.
+ Critical messages get through even if there is a node malfunctioning and trying to flood the bus.
– The solution reduces bus throughput, as replying to a message is needed or messages must have TTL which have to be waited before sending next message.
– The bookkeeping may add delay to throughput if multiple messages of certain type are sent in a row.
– The solution adds complexity as bookkeeping mechanism is required.
– The bookkeeping forms a single point of failure.

Resulting Context

The system results in a distributed embedded control system where the possibility of babbling node to prevent normal operation is reduced.

Related Patterns

BUS-GUARDIAN [9] solves babbling idiot's problem by introducing time slots for each node where they can send their messages. If some node sends message out of this time frame other nodes notices that the babbling node is malfunctioning.

Known Usage

A mining drill consists of multiple nodes that use a CAN bus with CANopen protocol to communicate with other nodes. The controllers in the drilling unit are more likely to malfunction as rocks flying from the drilling process may physically damage controllers. Sometimes damaged nodes may start to send emergency-specific EMCY messages to the CAN bus repeatedly. The pace of sending the messages can be so high that if the number of messages sent is not limited with ONE AT A TIME, the receiving end will choke on the messages. This may result in situation where the receiving node does not get its own messages sent to the bus as it has to handle all messages received. This may even result in situation which makes the machine to stop operating. When this pattern is applied the problem is solved.

2.5 Prioritized Messages

Context

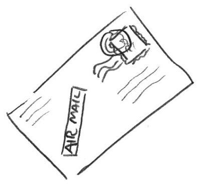
A distributed embedded control system uses BUS ABSTRACTION to communicate between the nodes. The MESSAGE QUEUE pattern has been applied to enable asynchronous communication. There are some messages that are more important than others.

Problem

How to ensure that important messages get handled before other messages?

Forces

- *Data integrity*: Regardless of the importance, all messages should be processed because no data should not be lost. Thus a message that needs immediate action may not discard other messages.
- *Response time*: Important messages, e.g. emergency messages, should be processed as soon as possible.
- *Response time*: All messages should be processed as soon as possible, but taking into account their relative importance.
- *Safety*: Emergency situations should be taken care of as fast as possible, but the system should also be able to return to normal operation as soon as the exceptional situation is over.



Solution

The nodes communicate with each other using a message queuing service. This consists of message queues and some kind of message dispatcher that serves the messages in the queue. The messages are prioritized according to their importance. The importance is a relative concept so the designer must decide which messages are consider more important than the others. Usually error messages and other exceptional communication is given a higher priority. In most simple case, there is only two levels of priority: messages of high importance and normal messages. The message dispatcher places messages in different work queues based on their priority level.

When the message processor begins to serve the queue, it begins with the highest priority queue and after it is empty, it starts to serve the lower priority level. After every processed message it must check whether a higher priority queue has new messages to process. The designer must be careful that no starvation on any message priority level shall occur. Starvation means that high priority messaging is so frequent that the lower priority messages can not be served at all.

The message priority must be somehow recognizable for the message processor component. It may be explicit from the message type, for example all emergency messages have high priority, or the message itself may carry additional meta-information about its priority. In this case, the sender must set the priority information correctly to the message and make sure that high priority messages are used only when needed. Otherwise, it may congest the message channel. In some cases, the priority may be deduced by the sender or from the context. For example, some node in the network sends always important messages.

Consequences

+ The most important messages can be processed regardless of the amount of the normal messaging.
+ The order of same priority messages is not changed as the queue holds the messages in the order they were inserted.
– The correct chronological order of messages is lost, because high priority messages are served first.
– Due to chronological discrepancies, priority inversion may occur. This means that a high priority message starts some action, that needs information from a lower priority message that is not served yet.
– Congestion of high priority messages can cause starvation of the normal messages.

Resulting Context

The pattern results in a system where messages are handled in the order of importance.

Related Patterns

ONE AT A TIME solves message prioritization problems using another approach. [10] describes a similar messaging concept. EARLY WARNING helps in designing proper message queue lengths. VECTOR CLOCK FOR MESSAGES can be used to keep track of the chronological order of the messages.

Known Usage

A drilling machine has controllers that communicate via CAN bus. Basic situations are handled with normal messages, but if an emergency occurs, for example a controller crashes, a special kind of message is used. These EMCY messages have higher priority as the normal messages, so other controllers can react to the situation quickly. The message queues are presented in Fig. 3. The message dispatcher serves the high priority EMCY messages first.

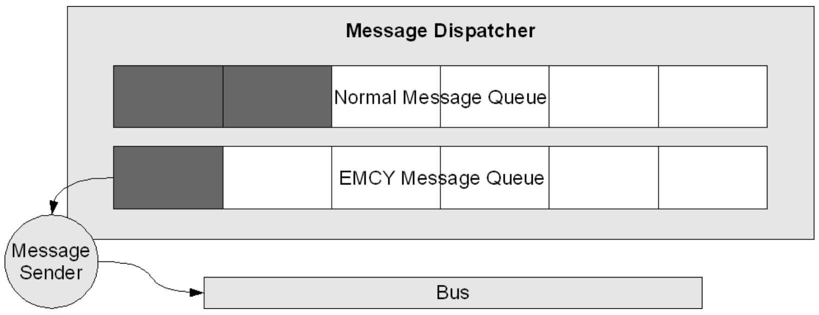


Fig. 3. Priority queues for normal and EMCY messages. The EMCY queue is served first.

2.6 Early Warning

Context

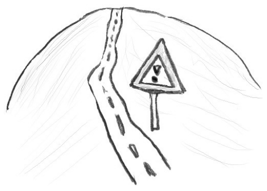
There is a distributed embedded control system that uses MESSAGE QUEUE or some other message handling mechanism such as event dispatcher. There is message queues in the system which size should be determined, but one can not be sure which buffer size is enough.

Problem

How to be sure phase that static buffers in the system are large enough to store messages also in the worst case scenario? How to make sure that system will not crash when the amount of messages exceeds the expected worst case scenario?

Forces

– *Fault tolerance*: There should not be message buffer overflows during the runtime as overflows may crash the whole system.
– *Predictability*: Only an estimate of the amount of messages sent or received during run time is known beforehand. Therefore the size of the buffer needed in the worst case must be estimated.
– *Testability*: A buffer overflow is not acceptable during testing phase, as it may cause harm to the environment, to the operator or to the machine itself.
– *Resource utilization*: Message buffer should not be unnecessarily large.
– *Efficiency*: No dynamic data structures can be used since they may cause another kind of problems.



Solution

The suitable buffer length can not be known before the system is tested. However, when testing the software with real hardware, there should not occur any buffer overflows as they may cause harm to the environment. Therefore a mechanism that warns the user or a tester when a certain filling level of buffer is met, is added to the buffer(s). The required buffer size of the worst case situation is evaluated and the filling level triggering the alert is set to the worst case situation. The buffer is designed to be approximately one third larger than the worst case situation requires. Usually, alert is given when the estimated worst case level plus a few additional slots of the buffer are used. The buffer usually is designed so that it has one third of its slots free in the alert situation. In this way, in testing phase the estimated buffer size adequacy can be evaluated in real use without crashing the whole system.

The correct filling level can be adjusted during the testing. If the initially set level is too low, it should be raised. On the other hand, if the level is never met during testing, the architect should try lowering the level to find optimum alert level. When the alert is given there should still be empty slots left in the buffer, e.g. one third or two thirds of the buffer left. Extra space of the buffer depends on the criticality of the system and it should be defined during the development phase. When the system is taken into production use, the mechanism is left in place. If the buffer should run out of space in production use, the mechanism warns the user with according error message and shuts the system down gracefully. Criticality of the buffer defines if the shut down is done automatically or left to the user.

Consequences

+ The pattern helps to verify that message buffer is sufficiently large in the testing phase. This increases reliability of the system.
+ Early warning mechanism can be used to shutdown the system gracefully in production use if the buffer overflow is about to occur.
+ Early warning makes it possible to test the system on actual hardware since possible buffer overflow situations can be recognized beforehand. In this way, dangerous situations are prevented even when testing with actual hardware.
– The warning mechanims may decrease system performance as the limiting value has to be checked for every message.
– The solution makes the buffer larger than it would need to be as additional reserve space to buffer has to be added.

Resulting Context

The result is a system that has buffers that should be large enough even in the expected worst case situation. Even during the worst case situation the system should have empty slots in the buffer, so it will not crash if the expected worst case situation is exceeded. In the production use the extra space in buffer is used if the expected worst case situation is exceeded and when the early warning limit is met, the system is shut down gracefully. The resulting system can also notify the machine operator or a tester that message queue overflow is about to happen and can shut down the system gracefully.

Related Patterns

Pattern can also be applied when PRIORITIZED MESSAGES has been applied. The ONE AT A TIME pattern can also be used to limit the number of simultaneous messages on bus.

Known Usage

EARLY WARNING can be used for example in buffers which store events that are received from the CAN bus. In the development phase, the amount of periodic messages is known, but the amount of sporadic messages can not be defined exactly. The architect assesses that system has 30 periodic messages and in the worst case there could be 15 sporadic messages simultaneously. The early warning level of CAN bus message queue is set 50 and the buffer size is set to 75. During the testing phase, stress tests results in the situation where the early warning level is met. It is found out from the system logs that the amount of the messages were actually 55 at maximum. The early warning level is set to 60 and the buffer size is increased to 90. Now the tests pass without any warnings from the early warning mechanism.

EARLY WARNING is applied so that in the production use the system will inform the machine operator when the early warning level of message queue used to stored received messages from the bus, is met and the system will automatically shut down the system gracefully. Now, for example if the boom controller node crashes and starts to babble to the CAN bus with sporadic messages, the system will not crash. The extra space in the MESSAGE QUEUE is used store the messages that the node sends and when the early warning level is met the system starts to shut itself down automatically. In this way, the machine malfunction will not cause danger to the operator or environment.

2.7 Converting Message Filter

Context

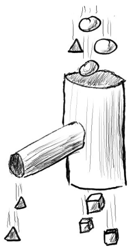
There is a distributed embedded control system where communication takes place over multiple communication channels and BUS ABSTRACTION has been applied. Communication channels are connectionless and they can be different or same type, e.g CAN and Ethernet or two CAN buses. Messages can be targeted to one specific node, to group of nodes or to all nodes meaning basically unicast, multicast or broadcast.

Problem

How to efficiently route and convert messages between message channels and filter unnecessary message traffic from the message channel?

Forces

- *Resource Utilization*: Only the message channels needed for the delivery of the message should be used. In other words, extraneous messages should not forwarded to the message channel where there are no recipients interested in the message.
- *Interoperability*: It should be possible to use different types of message channels in parallel.
- *Testability*: There should be an easy way to replace a message channel with a stub that can send automatic replies to messages or only monitor traffic.
- *Interoperability*: Some nodes may only function with a specific communication channel. It should be possible to use such nodes with other nodes that only function with another communication channel.
- *Interoperability*: Some message channels may work only with certain types of messages, e.g. periodic messages, whereas some other message channel uses only sporadic messages. There should be a way to convert messages from one type to another.



Solution

A software component, message filter, is added to the system between message channels. This component filters message traffic between message channels according to specific criteria defined in system configuration. In addition, the component can handle translation from a messaging type to another, if needed. The message filter should know which nodes are connected on which channels the nodes are connected to so that it can determine if the message needs to be delivered to another message channel. Location information of the node should be available in system configuration meaning that system configuration must be modified when a new node is added to the system. For example, in Fig. 4 there are two message channels M1 and M2, and three nodes A,B and C connected to channels. When node A sends a broadcast message to channel M1, node B receives the message and processes it if node B is interested in the information A sent. In addition, the message filter examines the message and notices that node C is interested in the message and therefore bridges the message forward to channel M2. Otherwise message filter will block the message and it is never sent to M2.

The message filter can be used to convert a message to another format if there are different types of incompatible message channels. For example, one message channel may use only asynchronous messages whereas another message channel uses only synchronous messages. The message filter can do the conversion between channels. The message filter must know message format of both channels as it needs to examine the messages to find out the receiver so that it can determine if the message should be bridged to another channel. In addition, if broadcast messages are used, the filter must know which kind of broadcast messages should be bridged and which ones should be blocked.

The message filter also helps to divide the system into logical partitions that can be tested separately. Message filter creates a logical division point for testing as filter can be quite easily stubbed. In this way, message filter can be replaced with a PC in testing phase. The PC can be used to monitor traffic or to create dummy replies. Sometimes message filter can also be used to divide one lengthy and noisy communication channel to logical parts. This solution helps to balance channel load as all messages are not send from one end to another.

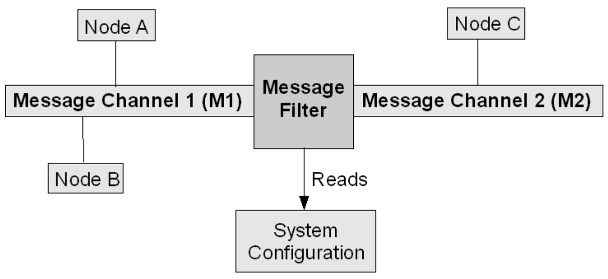


Fig. 4. Example usage of CONVERTING MESSAGE FILTER: two buses M1 and M2, and three nodes A, B and C.

Consequences

+ Load on a channel can be decreased by filtering out unnecessary messages arriving from another message channel.
+ Different kinds of message channels can be used together.
+ Message filter can be stubbed so that testing will be easier.
+ If a system has lengthy message channels it can be divided into separate parts by adding message filters to it. This may help to balance the message channel load.
– Message filter must know location of nodes and filtered message types. This makes the component complex and error prone. In addition, the system configuration is likely to change during the life cycle of the product. Therefore there should be a way to configure the message filter easily.
– The filtering may add additional delays to message delivery time.

Resulting Context

The result is a distributed system which can use inter-channel communication. In addition, there is filtering in inter-channel communication in order to minimize undesired message traffic between channels.

Related Patterns

MESSAGE CHANNEL SELECTOR provides a solution to decide which communication medium should be used for each message type. MESSAGE FILTER [7] describes similar mechanism to filter messages.

Known Usage

The machine control system uses two CAN buses, one inside the cabin (cabin bus) and another that connects all other machine control nodes (machine bus). A message filter is used between these two buses to filter out undesired messages from bus to another. In cabin bus there is a lot internal messages that the cabin PC sends to indicators and displays in cabin, etc. The message filter is used to filter these out from machine bus. Only messages that are related to steering the machine are bridged to the machine bus.

2.8 Distributed Transaction

Context

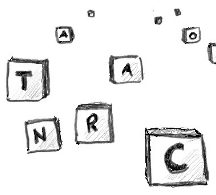
There is a distributed embedded control system where ISOLATE FUNCTIONALITIES have been applied. There are multiple logically connected nodes that are used coordinately to implement a functionality. Actions can consist of multiple command messages. However, a node can fail independently to commit to a given action request and other nodes should not execute the given action request either in this case.

Problem

How to ensure that nodes commencing an action cooperatively are in the proper state to proceed?

Forces

- *Resource utilization*: An implementation of a functionality may require multiple co-operating nodes. Co-operation might be needed, for example, because there is not enough I/O channels on one node.
- *Interoperability*: A node realizing a sub-action needs to be sure that other nodes can execute successfully their corresponding sub-actions, i.e the action can be realized as a whole.
- *Safety*: A safe realization of an action requires that all participants can execute their part of the action.
- *Distribution*: System has functionalities that have to work concurrently. Therefore, the system must be distributed.
- *Consistency*: The data integrity of the system should endure in all situations. The data integrity is lost if actions are not carried out as a whole.
- *Atomicity*: The action that is distributed to multiple nodes should be carried out as a whole or not at all.



Solution

Some functionality in the system requires co-operation of multiple nodes. This functionality is triggered by a request message or an event. If only a subset of nodes react to the request, while other nodes can not carry out the requested action, the result may be uncontrolled actions. These may cause severe harm or danger. Therefore, a distributed transaction is required to guard operations.

A distributed transaction consists of several sub-transactions which each run on a different node. Each node has a right of veto and can decide to abort the requested operation. In addition, there is a transaction controller in the system, usually situated on a high end node (introduced by SEPARATE REAL-TIME pattern). The transaction controller can be implemented inside interface created by BUS ABSTRACTION.

The transaction component takes the request for the action and divides it into sub-transactions which are delivered to corresponding nodes for execution. Each node replies to transaction controller within a predefined time limit. This reply is a vote whether the action should be executed. Nodes simulate the execution up to commit command to make sure they can execute the requested operation. Node votes "commit" if its sub-transaction can be executed. It means that the node informs the transaction controller that it is currently in a state where the transaction is possible. If the node malfunctions after sending the vote, the situation may change. If a node can not commit to the transaction it votes "abort" and does not wait for the results of the vote. If this is the case, it may continue its own execution flow. The transaction controller makes a non-reversible decision if the sub-transaction should be executed or not. If there are no failures and all nodes voted for commit, the transaction controller decision is "commit". Otherwise all sub-transactions will be aborted and the whole transaction will fail as well. When the decision is made the transaction controller will inform the decision to all nodes. If the decision was to commit, then all nodes will execute their sub-transactions.

In more detail, the transaction protocol functions as follows:

Voting Phase

1. The transaction controller sends a query to commit message over bus to all nodes and waits until each node has answered or a certain time window has passed.

2. Nodes execute the transaction up to the commit command (not executing commit). Each node writes an entry to the undo log. At this point the node creates locks for required resources.
3. Each node replies with a vote, if the transaction should be committed or not. The answer is yes if step 2 was successful.

Completion Phase

1. The transaction controller sends either a commit or rollback message to all nodes according to the results of voting. If any of the nodes has answered "abort" in voting phase, the rollback message is sent.
2. If the message was commit, then each node completes the operation. Otherwise nodes use their undo log to rollback. Afterwards the node releases all the locks and resources held during the transaction.
3. Each node sends an acknowledgement to the transaction controller.

Fig. 5 illustrates the communication between transaction controller and nodes in both phases. This kind of protocol achieves its goal even in many cases of system failure (involving either process, network node, communication, etc. failures), and is thus widely utilized [11]. However, it is not immune to all possible failures, and in some cases user intervention might be needed to resolve the outcome. In addition, if there are commands that result in changes of the physical state of the machine, the outcome of command might need to be simulated in some proper way instead of actually executing them. If a unit participating to voting malfunctions after the commit message, the protocol does not guarantee successful outcome. In some situations it might be reasonable to change the node which acts as transaction controller in order to avoid transaction controller from being a single point of failure. However, this requires that multiple nodes must implement the transaction controller logic. Usually a single transaction controller is used as it is easier to design and use.

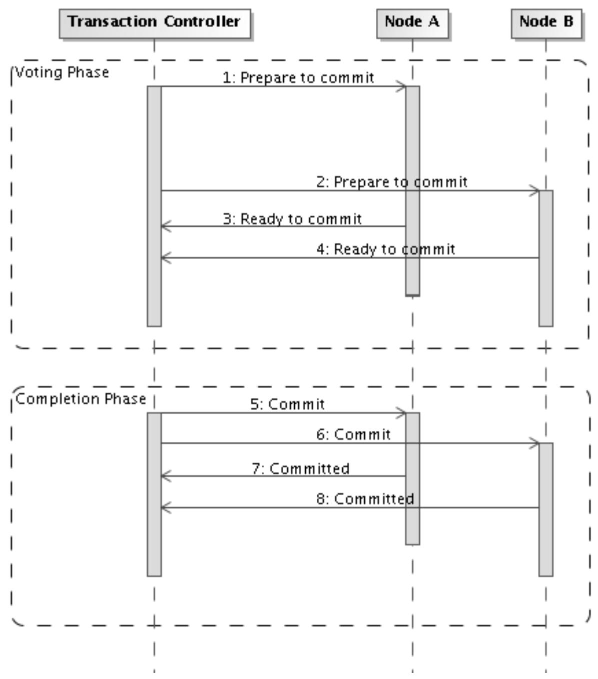


Fig. 5. Illustration of interaction between transaction controller and two nodes in both phases.

There may be multiple nodes in the system that implement the same functionality and are used in parallel. In this case, in voting phase it is enough that one of these nodes votes for commit. For example, in elevator system this could mean that when a call is allocated, it is enough that one of the elevator cabins commits to the request. Furthermore, if there are several elevator cabin groups that needs to commit to the service request, it is enough that one of the cabins from each group commits to the request. A variation of the solution presented here is that a node may forward the commit message from transaction controller to another node.

Consequences

+ The solution allows coordinated decision making for commands requiring co-operation of multiple nodes.
+ The implementation of a functionality may be divided into multiple nodes. For example, in a situation where a node does not have enough I/O this might be very useful.
– A big downside of the message transaction is that nodes must block while waiting confirmation to commit from transaction controller after the voting phase. This adds latencies to executing the operations and therefore this approach is not suitable for systems having strict real-time requirements.
– Transactions add bus load as additional messages are needed to execute an operation.
– Poor implementation may result in possible deadlock situations that can be hard to find in testing phase.
– Transaction controller may create a single point of failure.
– Starting time of the command execution can not be synchronized using this pattern alone. However nodes can synchronize their actions with other nodes using traditional synchronization mechanisms.

Resulting Context

This pattern results in a system where a functionality can be distributed over multiple nodes and can still be executed in coordinated manner.

Related Patterns

Solution introduced in this pattern is close to two-phase commit (2PC). Transaction controller introduced in this pattern can be located in high end node if SEPARATE REAL-TIME has been applied. Transaction mechanism can be abstracted from the developer if transaction controller is implemented behind the interface introduced by BUS ABSTRACTION. REDUNDANT FUNCTIONALITY can be used to create a backup for the transaction controller. A mechanism described by OPPORTUNISTIC DELEGATION can be used to determine if a node can commit to a service request or not.

Known Usage

In an elevator system all elevator calls are delivered to elevator cabins using mechanism provided by CONVERTING MESSAGE FILTER. In addition, there is a system that controls a group of elevators and gives floor calls to cabins which will serve these floors. Calls are presented to the cabin as a query to commit. Then each elevator cabin informs if it can commit to the list of calls. If not, all cabins rollback their list of calls to serve and the system that controls whole group reallocates all calls again using different cabins. When all cabins can commit to calls which are allocated to them, the serving order is fixed, i.e. the request is committed. When a new call arrives, then a new round of queries are sent.

2.9 Message Channel Selector

Context

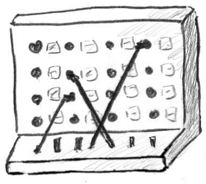
There is a distributed embedded control system where communication takes place over bus and Bus Abstraction has been applied. There are more than one message channels in parallel. Buses can be of the same types, e.g two CAN buses, or of different types, e.g. a CAN bus and an Ethernet bus. A variety of data types requiring different properties from the message channel needs to be transferred.

Problem

How to choose appropriate communication channels for each message sent?

Forces

- *Throughput*: Bus properties may vary by type and suitable channel should be used. For example, updates may require more bandwidth and should be transferred using faster communication channel.
- *Resource Utilization*: It should be possible to balance bus load between communication channels.
- *Fault tolerance*: There should be a mechanism to avoid broken communication channel.
- *Scalability*: Parallel communication channels offer higher bandwidth.
- *Extendability*: If an additional communication channel is added to the system, it should be still possible to use the same interface for sending and receiving messages from the channel.
- *Availability*: Multiple communication channels increase availability in case of malfunction of the channel.



Solution

There may be several different buses, such as CAN and Ethernet, or different service level channels of the same bus type, i.e. CAN bus for data messages and another CAN bus for control messages. Therefore, we need a mechanism that decides which bus to use in each situation. For example, the mechanism could be used to balance the load between message channels or deliver updates through faster communication channels. Therefore a selector component should be added behind the interface of bus abstraction (Bus Abstraction). In this way the the user of the interface does not need to care about using the correct channel as the selector component will decide that.

In Fig. 6 example usage is illustrated. Node A sends a message to Node B. Node A uses interface introduced by Bus Abstraction to send the message. Behind this interface message channel selector component is implemented and it chooses appropriate message channel for the message. Message is delivered using selected channel and received through interface on Node B.

The selector component examines type of the message and decides which channel to use according to the type. In this case the component is configured to send certain type of messages through a certain channel. Configuration can be given for example as an XML file which describes message types and channels and mapping between them.

The second option is that the selector component makes decisions depending on the size of the message to be sent or load of the channel. In this case, the component should be configured to use certain channels for certain sized messages. For example, it could use CAN bus for relatively small messages and Ethernet to transfer large messages, e.g. software updates.

Furthermore, there may be need to send messages through different channels because of messaging scheme. Some message channels offer only synchronous communication whereas some channels offer asynchronous communication. This may set limits to the channel selections and should be taken care of in the configuration file.

If bus load balancing is needed, it requires more sophisticated approach. The channel selector component should keep database of message traffic at different times on the bus and use that information to select appropriate communication channel. An adaptive system, which learns peak times, may be required in order to implement this.

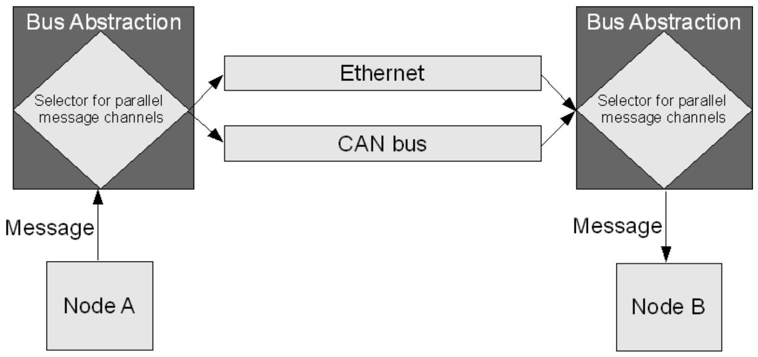

Bus Abstraction
Selector for parallel message channels
Ethernet
CAN bus
Bus Abstraction
Selector for parallel message channels
Message
Node A
Message
Node B

Fig. 6. Example usage of selector for parallel message channels.

Consequences

+ The system can have multiple communication channels which are as easy to use as one communication channel.
+ When using advanced approach the communication channel load can be balanced between multiple buses.
+ Selector chooses appropriate communication channel, the decision making is centralized.
– The solution adds another layer to abstraction and therefore may limit performance .
– The selector component needs configuration. Finding the right threshold values may take some experimenting before optimal operation mode is found out.
– the solution may create a single point of failure.

Resulting Context

The pattern results in a system where multiple communication channels can be used in parallel through single interface. Channel user does not need to care about which channel should be used as the routing service takes care of that.

Related Patterns

CONVERTING MESSAGE FILTER can be used to filter messages between two different buses. For example, if node A is connected to bus 1 and message from node C in the same bus is sent to A there is no need to deliver the message to D in bus 2 in the system.

Known Usage

In an elevator system, there is a screen that shows the passenger which elevator to take from sky lobby in order to reach the destination floor. In such system, other communication channel, e.g. Ethernet can be used to transfer the video to elevator cabin and CAN bus can be used to inform elevator cabin when to show the video to user.

2.10 Vector Clock for Messages

Context

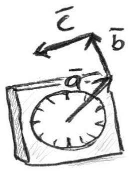
ISOLATE FUNCTIONALITIES pattern has been applied to create a distributed control system. The system communicates over a bus using messages. A message may contain commands and event information. There is a need to know the order of events. As the systems is distributed every node on the bus has its own clock. A small clock skew between two nodes is probable and therefore the exact time of an event in the system can not be determined using individual clocks.

Problem

How to find out the order of events in distributed system?

Forces

- *Testability*: It should be possible to replay the messages in the time order in testing.
- *Traceability*: The order in which the messages were triggered should be determinable.
- *Accuracy*: Clocks in nodes will show different time in distributed system due clock skew.



Solution

Every event is given an vector clock timestamp. The timestamp consist of separate counter values for every node. The counter for a node is increased by every message received which has higher counter value for that node. Following example is illustrated in Fig. 7. For example, a two node system consisting from nodes A and B starts with timestamp "A:0 B:0". In system the timestamp for first message sent by A would be "A:1 B:0" meaning that by the time event occurred and the message was in construction, B had sent zero messages and that this is A's first message. When B receives the message it will update its counter values for those values which are higher than the values it has. If B would now send a message, it would have timestamp "A:1 B:1" meaning that one message from A has been seen and this is first message from B. If A would have an event about the same time, it would have timestamp "A:2 B:0". When the counter values increase crosswise, we know that the events occurred parallel but unfortunately which one was exactly the first can not be said.

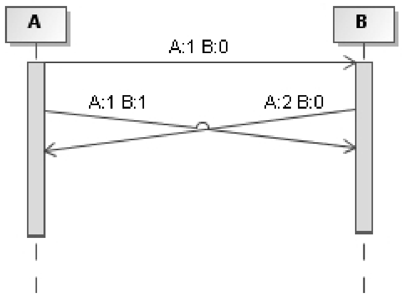


Fig. 7. A sequence diagram example of two node system using vector clock.

If all the events are not communicated, the message counter value can be replaced with event counter. In this case there can be larger changes between counter values, they just tell that events have occurred. The event counter are synchronized the same way as in previous example when the nodes communicate. If the nodes do a lot of work between communication, the order of events in a node can be observed but they can not be compared to some other nodes internal event times. In any case, if every node logs their own events and vector clock values, or the bus traffic is logged, the order of events can be determined from the timestamps. For nearly parallel events the parallelism can be also seen. The granularity increases when there is more communication.

Consequences

+ It is possible to put events in order.
+ With vector clocked messages, occurred situations can be replayed.
– The data has to be logged in every node or a separate observer is needed.
– Timestamps use more bandwidth than messages without them.
– Exact order of nearly parallel events can not be determined.
– If there is no communication for every event, the internal order of events of a node can not be compared to another nodes internal event order.

Resulting Context

The result is a system where the relative order of events can be determined later on from the time stamps.

Related Patterns

A vector clock is a classic solution for a distributed system, one example of this is Lamport timestamps [12]. FAULT OBSERVER [13] can be used to create an entity to the system which can log the order of events.

Known Usage

In a mining drill all messages were timestamped with a node clock time. This didn't work as well as it was thought to work. This was due to the relative clock skew between the nodes and prioritized messages which would pass the normal sending queue. Therefore, messages even from single node could arrive in disorder and the timestamps for two or more nodes could not be compared as they had clock skew. The problem was solved by changing timestamps to a vector clock.

2.11 Permission Request

Context

A distributed embedded control system has been divided into nodes with ISOLATE FUNCTIONALITIES. The nodes make independent decisions about their functionality. However, there may be situations where the autonomous functionality can not be executed as other nodes may have conflicting needs preventing the action. For example, the cabin controller has knowledge that parking brake is engaged, in this case the transmission controller should not allow driving. As there are multiple dependencies, albeit simple as such, but as a whole they form complex combinations which are difficult to synchronize.

Problem

How to ensure that an independent action of a node is not conflicting with some other nodes goals?

Forces

- *Distributability*: Information required for making a decision for execution may reside on some other node.
- *Resource utilization*: A node may require information from multiple nodes to make a decision. This may cause a lot of bus traffic.
- *Performance*: All information can not be shared to all nodes, as it causes latency and compromises bus throughput.
- *Safety*: Individual node may not take action autonomously as it may compromise safety. For example, the machine should not move if the cabin door is open.
- *Scalability*: When the number of nodes increases, the dependencies between nodes grow exponentially.
- *Decoupling*: A node does not need to mind of functionalities out of their responsibilities. For example, the transmission controller does not need know that harvester head even exists. However, harvester head operation may prevent the driving.

Solution

A node is responsible for some functionality and it has relevant information for its own responsibility area. However, different functionalities in different nodes may interfere with each other. Additional information, which is not locally available, may be needed for decision whether to execute functionality. Typically actions which need permission are irreversible. There has to be a way to either get information for decision making or externalize the decision.

As the system may consist of different set of nodes, it is difficult to know beforehand what functionalities might interfere with each other. Therefore, it is simpler to centralize the decision making to separate permit authority. It needs information from nodes to make a decision. The permit authority may collect the information on demand or it has collected them from the announcements of different nodes. Usually the permit authority is located in high-end node, as decision making may require processing power.

A device does not need to ask permission for internal actions separately, because it has permission to execute the whole functionality. For example, if harvester head option for prevention for fungal disease is activated, after cutting the tree down the root will be sprayed automatically. However, for the cutting functionality a permission has to be asked. If something occurs that interferes with the cutting functionality during operation, the action has to be aborted with separate abort message.

Consequences

+ As the decision making is externalized, less processing capacity is required in the node.
+ The permission is always asked from the same entity. This makes the system more understandable.
+ The details do not need to be transferred, because permit authority processes the information to a single decision.
+ The permission requesting decreases possibility of single node to cause harm.

+ External dependencies of single node are reduced.
– Permit authority may be a single point of failure.
– Designing the permit authority may be challenging as it depends on all the information and therefore from all the nodes.
– Different combinations of devices present different constraints for decision making.

Resulting Context

The solution results in a system where a node can check from the permit authority if it can execute its functionality.

Related patterns

DEVICE PROXY and HARDWARE ABSTRACTION LAYER can be used to abstract devices, so that if the hardware implementation changes, the permit authority is not affected. SOMEONE IN CHARGE [13] presents a paradigm that there has to be always someone responsible for decision. VARIABLE MANAGER can be used to provide a place to store the information required for decision making.

Known Usage

One node of the system is a dedicated authority node. This node uses VARIABLE MANAGER to store system state. For example, the drive controller receives a command from the bus to move the machine. Before the controller executes the command, it checks from the authority node if the command can be executed. For example, if the machine operator has pressed emergency stop, the command can not be executed.

2.12 Unique Confirmation

Context

There is a distributed embedded control system where ISOLATE FUNCTIONALITIES pattern is applied. The system has some actions that need confirmations from other entities. There might be multiple confirmations waiting for processing. The confirmations must not be confused with each other in order to avoid false positives.

Problem

How to handle situation where some actions need confirmation and there is risk to have false positives?

Forces

- *Correctness*: There should be no false confirmations as they may have severe consequences.
- *Resource utilization*: Complicated confirmation protocols may be cumbersome.
- *Performance*: Communication faults or latency may cause confirmations to be sent several times or in incorrect order.This may lead to confusion.

Solution

A situation may rise, where there are multiple pending actions waiting for a confirmation. A simple "OK" message may be interpreted by the acting unit as confirmation for any of the pending actions. Therefore, the messaging logic must be redesigned so, that the confirmation messages are different for every action to be confirmed. This may be achieved by using unique identifiers. For example, the request messages may be identified with unique identification numbers (e.g. "‘Request #1"’ and "‘Request #2"’). The confirmation messages can use this number to identify the correct request (e.g. "‘Confirm #1"’ and "‘Reject #2"’). In this way, there is no risk of confusing the replies. In simpler cases this may be achieved by just using different confirmation actions in different phases of confirmation sequences (e.g. OK1, OK2). This simpler mechanism is illustrated in Fig. 8.

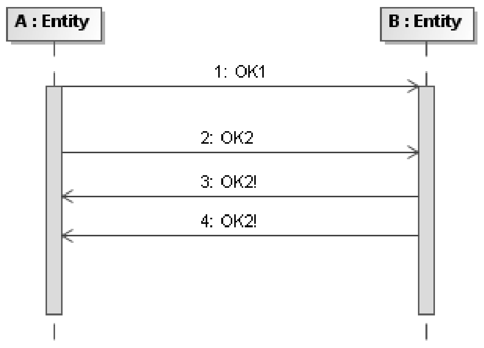


Fig. 8. The unique confirmations for request messages.

Consequences

+ The messages can be sent asynchronously, as the confirmations are always unique.
+ The system debugging is easier as the message confirmations are easily linked to their corresponding requests.
+ The pattern allows confirmations be duplicated without fearing false positives.
– The unique confirmations require bookkeeping of the sent messages and their corresponding confirmations.
– The identifiers make the messages longer, thus wasting the bandwidth.

Resulting Context

This pattern results in a system where replies to multiple requests are not mixed as all requests have unique confirmation responses.

Related Patterns

VECTOR CLOCK FOR MESSAGES addresses the problem associated with finding out the correct order of events in a distributed system. ONE AT A TIME can be used to prevent a situation where multiple confirmation messages of same type need attention.

Known Usage

UNIQUE CONFIRMATION is used in many communication protocols, where all confirmations hold the identification number of the message they are a response to. For example, TCP/IP has on TCP level a special sequence number in each of the messages that binds replies to a known message.

2.13 Locker Key

Context

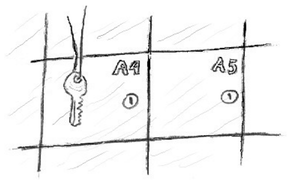
There is a distributed embedded control system where ISOLATE FUNCTIONALITIES has been applied. A controller has shared memory that multiple processes can access. Processes communicate with each other using some kind of messaging scheme. Intense communication between processes may occur. Direct interprocess communication scheme involves memory allocation for messages, copying large memory blocks from address space to another and requires active participation of both, the sender and the receiver. Therefore, this kind of scheme should be avoided.

Problem

How to efficiently communicate between processes avoiding dynamic memory allocation?

Forces

- *Efficiency*: Dynamic memory allocation consumes processing time. It takes time to find and reserve free memory and copy information between memory areas.
- *Throughput*: Direct interprocess communication should be minimized.
- *Safety*: Dynamic memory allocation may fail and such a situation may cause drastic consequences in a real time system.
- *Interoperability*: Interprocess communication should be carried out in a uniform way regardless of programming language.



Solution

An embedded controller has memory that can be shared among the controller's processes. All the processes can access this shared memory space and it has enough free space to accommodate all the communication needs of the system. This memory can be used for communication by allocating part of it as message lockers. A locker is a predefined sized memory area that can be used to store message content. The locker size is usually determined from the largest message size. By pre-allocating the lockers, dynamic memory allocation is not required later on. Therefore, the interprocess communication may not run out of memory and is faster.

Every locker is has a key which can be used to access the locker content. The key can be index, i.e. locker number. However, different key types such as Universally Unique ID (UUID) or result from a hash function can be used. Message sender requests a locker (i.e. a space in the shared memory) by using a reservation function. The function returns a key to the caller. The message sender uses a function with the key to insert data in the acquired locker. The key is sent to the receiver with direct interprocess communication scheme. The receiver uses the key to access the message from the shared space. The locker is freed after the access so it can be reused for other messages.

Consequences

+ No dynamic memory allocation is needed for messages. This is fast and prevents memory exhaustion.
+ The transferred data is minimal as only a key is delivered.
+ the solution offers a possibility for the receiver of the key to fetch the message data at suitable moment.
+ Common messaging mechanism makes the system more understandable.
– Normal indexes do not provide any kind of memory protection and therefore the locker integrity is compromised. By using UUID or hash, the key protection level is higher but also more processing is needed.
– In the situation where there is memory management provided by the operating system it may be costly and error-prone to map a single physical memory block to different logical addresses on different processes.

Resulting Context

The solution results in an embedded controller with fast interprocess messaging capabilities.

Related Patterns

EARLY WORK and STATIC RESOURCE ALLOCATION can be used to allocate lockers. EARLY WARNING can be used to find out the optimal amount of lockers and also during runtime to warn that critical amount of lockers are already in use.

Known Usage

In an elevator controller a part of the shared memory is reserved for an array that is used as lockers. The array element size is determined from the largest possible message payload. The lockers are used through an interface. The message sender can store the message payload to the locker using an interface function. The interface then returns the key for the particular locker. The sender delivers the key to the message receiver. The receiver uses the key to retrieve the message payload through the interface. When the key is used the locker space is freed. In other words, the same key can be used only once.

3 Patterns for Real-time Separation

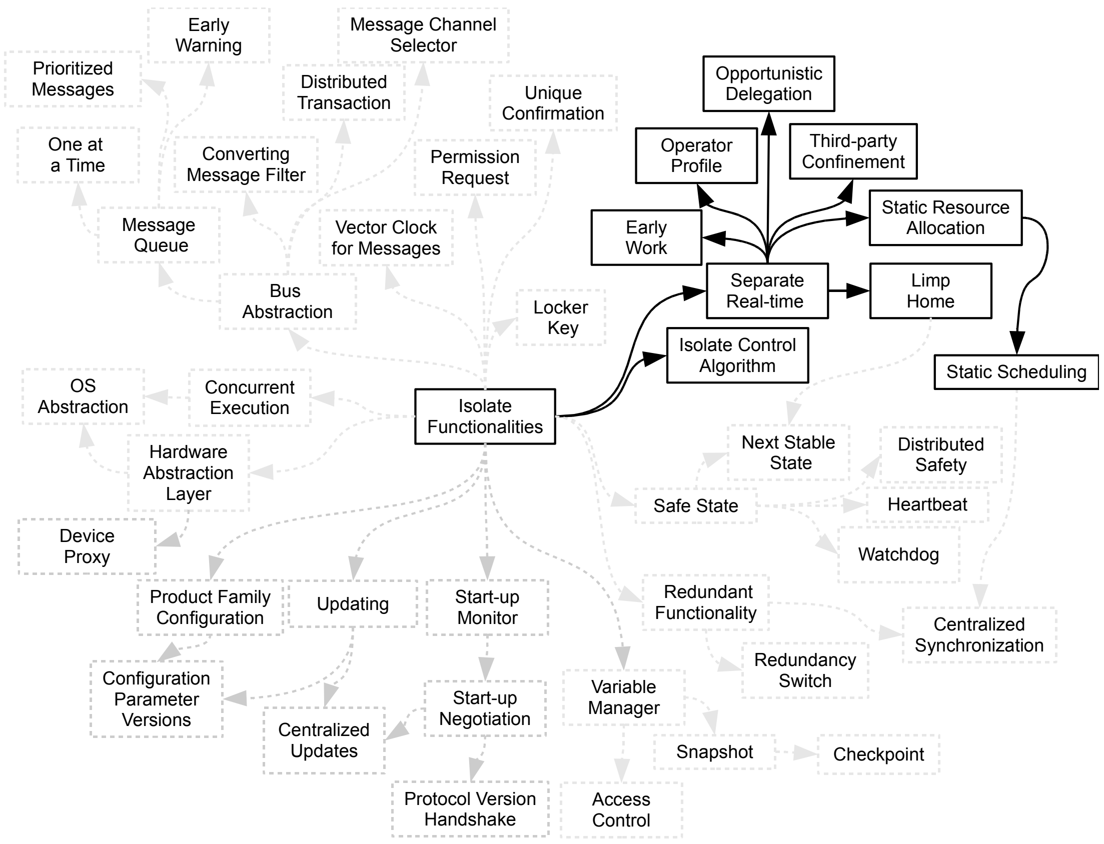


Fig. 9. The sublanguage for real-time separation in distributed machine control system.

Table 2: Pattern thumbnails

Pattern Name	Description	Page
SEPARATE REAL-TIME	The system is divided into separate levels in order to separate real-time functionality from the functionality which time behavior is not predictable.	37
EARLY WORK	Processes are initialized and all resource intensive tasks are carried out at startup. This frees resources for the main functionality during run-time.	40
OPERATOR PROFILE	The machine operator may use preferred settings regardless of the current machine in use.	42
OPPORTUNISTIC DELEGATION	In transient system tasks are delegated to a subsystem with available resources. This results in higher resource utilization.	44
THIRD-PARTY CONFINEMENT	The pattern results in a system with a capability to provide a restricted environment for third-party applications.	46
STATIC RESOURCE ALLOCATION	Critical services are always available when all resources are allocated when the system starts.	48
STATIC SCHEDULING	All processes are scheduled statically during the compiling time. This gives predictability to execution times.	50
LIMP HOME	If a sensor or a controller malfunctions the machine should still be at least partially operable.	52
ISOLATE CONTROL ALGORITHM	To improve changeability, control algorithms are abstracted in a way that they or their parts can be easily changed.	54

3.1 Separate Real-time

Context

The context is a distributed embedded control system where ISOLATE FUNCTIONALITIES has been applied. Part of the system has real-time requirements but there are also applications offering high-end services whose time consumption is not predictable. These applications need to interact with real-time functionality where time behavior should be predictable to some extent.

Problem

How to offer high-end services without violating real-time requirements?

Forces

- *Safety*: Real-time functionality must not be interfered by other functionality of the system, i.e. high-end services. High-end services may cause the real-time part to function in unspecified way, e.g. real-time requirements are not met due to long processing of large messages.
- *Testability*: It should be possible to test real-time and non-real-time functionalities separately.
- *Understandability*: It is easier to develop high-end applications when the developer does not need to take the real-time requirements into account.
- *Response time*: Real-time part must meet its timing requirements as it is used to control the machine and the delay should be minimal.
- *Resource utilization*: High-end applications, which may have graphical user interface, may require more processing power than there is available on embedded controllers.
- *Subsetability*: Real-time applications may have different developers than high-end applications. It should be possible to divide the work into independent parts as different competencies are required in implementation process.
- *Fault tolerance*: High-end services are not required for basic machine control. In the case of a failure of high-end application, the machine should still be operable.
- *Fault tolerance*: Typical high-end hardware is more error-prone than embedded controllers.

Solution

The system has different kinds of services, some services need to meet strict real-time requirements and there are also services that do not have real-time requirements. Additionally, legislation may set certain limits to implementation of a service. Therefore, the system should be divided into separate levels: machine control level, machine operator level, and emergency stop level. The machine control level takes care of real-time functionality of the system and the machine operator level runs applications providing high-end services. The emergency level is a fail-safe mechanism that is usually used to stop the real-time functionality. These separate levels should be decoupled so that they will not interfere with each other. Therefore, the levels communicate with each other using only communication channels such as CAN bus. In this way, the levels become decoupled. The emergency stop level can use the same communication channel as two other levels but with higher priority messages. Alternatively, it may have its own communication channel. In some systems, it might be required to implement the emergency stop level using only hardware.

The machine control level handles all lower level operations which are used to control the machine. Services typically residing on the machine control level are steering control, transmission control, frame controller, and all intelligent sensors. Strict response times that machine control have can be met by implementing aforementioned functionalities on the separate machine control level using controllers such as Flexbox or PLC. In this way, the real-time functionality is separated from the software that does not have real-time requirements. This division makes the testing process easier. All machine control related low-level services, except for emergency stop, should be implemented on this level.

The machine operator level implements all high-end services such as graphical user interface for machine operator, diagnostics and communication with remote systems. Usually the machine operator level is implemented using a PC which has enough processing power required by high-end services. Another benefit of using a PC in the machine operator level is that it makes the usage of commercial off-the-shelf (COTS) components possible. Third party applications can also be run at the machine operator level as COTS components are used.

The machine operator level communicates with the machine control level only through communication channel such as CAN or Ethernet. No critical services should be implemented on the machine operator level as it runs applications that may cause situations where real-time requirements can not be met. For example, third party applications are run on the machine operator level to ensure that they do not interfere with the machine control functionality.

The system should be designed in a way that even if the machine operator level is malfunctioning, the machine can be still controlled, although it might require more interaction from the machine operator. For example, a service that assists the machine operator to control the boom with a joystick can be implemented on this level. However, if the machine operator level malfunctions, the machine operator should still be able to control boom valve-by-valve, but it can not be anymore controlled with a joystick.

The third level, the emergency stop level, should only implement emergency stop functionality. This level is used to stop all nodes or controllers in the machine control level when an event occurs requiring to stop all functions immediately. Additionally, it may need to stop some high-end functionality. In other words, the third level overrides the other two levels. The emergency stop level can be implemented with a separate bus, but in many cases this is too expensive. If this is the case, the emergency stop level can use the same communication channel with other levels to deliver emergency stop messages. These emergency stop messages must have higher priority than all other messages, i.e. there should be minimal queuing delay for emergency stop messages.

Consequences

+ By isolating real-time functionality of the system in a separate level, basic functionalities are easier to manage and high-end services can not interfere with them.
+ It is easier to test if the real-time requirements are met as the real-time functionalities are implemented as separate level.
+ Higher abstraction level programming languages can be used on the machine operator level as it has more processing power.
+ Development becomes faster as different aspects of the system, e.g. the machine control level and the machine operator level, can be developed in parallel.
+ Basic functionalities of the machine can still be used when the machine operator level software is malfunctioning.
– Division into different levels can be challenging. Which functionalities can be placed on machine control level?
– Solution adds latencies as a separate communication channel needs to be used between different levels.

Resulting Context

The pattern results in a system with distinct levels: machine control level that handles all machine control functionalities and machine operator level where higher level functionalities are implemented. In addition, emergency stop has either its own level or uses high priority messages to communicate with other levels.

Related Patterns

LAYERED ARCHITECTURE [14] describes similar division to layers. LAYERS [15] describes also how to divide system into different levels. In LAYERS the amount of layers used is not limited.THIRD-PARTY CONFINEMENT pattern refines this pattern by adding a platform, which can be used to run third party applications on the machine operator level. LIMP HOME pattern can be applied to ensure that some of the functionalities in the machine control level are available even if a part of the machine control level is malfunctioning. EARLY WORK and STATIC RESOURCE ALLOCATION can be applied to the machine control level to improve real-time behavior. OPPORTUNISTIC DELEGATION also defines how tasks can be delegated in a system which is in transition.

Known Usage

Independent power plant control system has strict real-time requirements. Therefore the system is divided into two levels: machine control level and machine operator level. Fig. 10 depicts the division into levels. The control system has local controls at machine control level that can be used to control the monitored process locally. The machine operator level is used to deliver the status information to a remote location which is used to monitor multiple process controls. System has emergency stop messages that have highest priority and are served first when they are received from the bus.

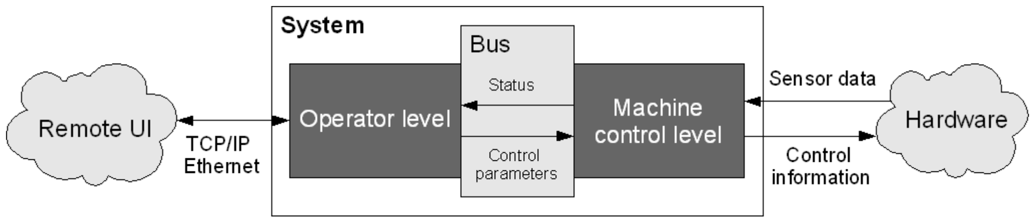


Fig. 10. Example of division into the levels.

The machine control level receives sensor information from hardware and processes it to present the system status. System status is presented in local UI and used to control the hardware. Status information is also sent using the bus to the machine operator level. The machine operator level receives this status information and sends it forward to remote monitoring application using Ethernet and TCP/IP. In addition, remote monitoring system may send a control command to the machine operator level which then delivers them to the machine control level using the bus. The communication to remote location must be implemented on the machine operator level as it would compromise the deadlines that must be met by the real-time part. Furthermore, even if the machine operator level malfunctions, the machine control level can be used to control the process. Although this means degraded service mode as the process can not be controlled remotely.

3.2 Early Work

Context

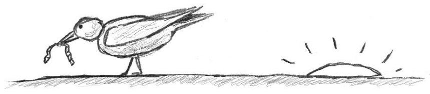
There is a distributed embedded control system where SEPARATE REAL-TIME has been applied. The system has hard real-time requirements. There are limited amount of processor or memory resources available after start-up. Thus, the resource usage should be minimized after start-up.

Problem

How to execute resource greedy tasks which may require more resources than are available for normal operation in run-time?

Forces

- *Real-time behavior*: The system has hard real-time requirements.
- *Resource utilization*: Some operations may need more resources than there are available after start-up.
- *Predictability*: Resource needs of process startup may be unpredictable.
- *Real-time behavior*: Because of limited amount of CPU time to use in run-time, there might not be enough CPU time for complex calculations.
- *Safety*: There must always be enough processing power for the main functionality.
- *Cost effectiveness*: It is not reasonable to have more hardware than needed during normal operation.



Solution

Allocating resource and starting new processes may take plenty of CPU time. Additionally, there may be a need for complex calculations that can not be pre-calculated during compile time. For example, values for some run-time variables are calculated based on the current configuration settings. Furthermore, in a system with hard real-time requirements, processes usually can not wait for needed resources (such as memory) to be available. In addition, initialization time may not be known beforehand for processes. Therefore, one should prepare the processes (like pre-calculating values to be used later on) in system startup, because there are usually more resources (such as memory and CPU time) available than later on. In addition, it is not so critical, if the startup takes longer as long as the execution is predictable after startup (i.e. in normal operation). This kind of pre-work frees the resources for the main functionality and makes the execution times of the operations more predictable.

In case of a large number of similar objects, a memory pool can be used to reduce memory consumption and improve response time in memory allocations. However, this technique can not be used for all kind of resource allocations. In addition, when using pooling, the resources must be allocated even if they are needed in very rare cases. Thus, allocating resources early can lead to unnecessary allocations and the system will actually need more resources.

Consequences

+ When all resource greedy tasks are carried out at startup, there are more resources for the main functionality in run-time.
+ The execution times are more predictable as the time consuming calculations are pre-calculated at startup.
+ Resource requirements are smaller as pre-work frees the resources for the main functionality.
– System startup time may not be predictable due to pre-work tasks.
– There are functionalities that can not be carried out at system startup.
– Pre-allocating resources may actually rise the resource consumption of the system.
– It might be difficult to find out which parts of the process can be moved to be executed at startup.

Resulting Context

The result of this pattern is a embedded control system where the resource greedy tasks are carried out at startup.

Related Patterns

STATIC RESOURCE ALLOCATION can be used to pre-allocate resources for critical services. START-UP MONITOR can be applied to control early work at system startup. POOLED ALLOCATION [16] describes a pattern for pooled allocation to allocate efficiently a large number of similar objects.

Known Usage

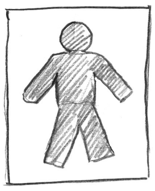
In a process control system, PLC language interpreter for hard real-time applications with very strict response time requirements (100 μs) is implemented in a CPU card. The interpreter process is started at start-up sequence and no other processes are created after that. In addition, all required memory is allocated before starting the interpretation to make sure that the execution times of the different operations are always known.

3.3 Operator Profile

Context
A distributed control system offers some high-level functionality for the operator using SEPARATE REAL-TIME pattern. This functionality is presented to the operator using some kind of graphical user interface (GUI), that has configurable settings for the user. In addition, there might be some physical control devices, that have adjustable parameters to accommodate the system for human operators, who have different preferences.

Problem
How the machine operator may use her personal preferences for UI and controls regardless of the current machine in use and the adjustments other operators have made?

Forces
- *Operability*: All operators should feel comfortable with the UI and controls.
- *Usability*: When an operator ends the work shift, the switchover should be quick for the next operator.
- *Attractiveness*: The operator should have an interface, that is easy to use and is modifiable to suit personal preferences.
- *Attractiveness*: Different kinds of devices performing the same function should have uniform controls. For example, a cabin chair should adjust to user preferences as closely as possible regardless of its adjustment axes.



Solution
The operator needs personalized settings in order to make the system more usable for him/her. This personalized information forms the profile for the operator. The settings that operator consist of multiple adjustable things in the operator environment. The adjustable things that form the profile may include:

1. the driving parameters, for example joystick sensitivity etc.,
2. the parameters for embedded functionality, for example valve response times etc.,
3. the third party software with their corresponding settings, and
4. GUI settings, such as gauge types (e.g. digital or analog gauge) and localization issues (e.g. language of GUI).

Optionally, all embedded software may be included in the operator profile. This ensures that all functionality is familiar for the user, but this may be difficult to achieve as the underlying hardware may be too different to control exactly in the same manner. The software package may be also too large to be conveniently used and updated.

All personalized information should be packaged in a single easily movable archive. The archive may reside on the operator PC hard disk. This creates an automatic backup of the operator environment settings. To make the system more user-friendly, the profile may be stored to a removable media, for example an USB memory stick. The USB stick may have a dedicated slot in the dashboard, so that the operator may access it easily and use the stored profile conveniently. In more sophisticated environments, the operator profile may be even fetched over-the-air, e.g. by using 3G data connection. The operator PC reads the profile during the system boot or when the operator logs into the system. The PC processes the profile information, makes the necessary adjustments and the system is ready to use. The profile may be stored for example as an XLM file. The file may hold all system parameters organized in a convenient manner. If the systems evolve, the operator profile may need some conversion tools between different versions of the operator environment.

If operator productivity is a traceable issue, the operator profile may also collect productivity data during the operator's work shift. This information may used as a basis for incentives or to train people off from ineffective work techniques.

Consequences

+ The system is easier to use as operator can import own familiar settings.
+ Familiar settings are transferrable to another machine.
+ Settings are not lost, if another operator makes modifications to the operating environment.
+ The settings are stored in two separate places. This creates automatic backup mechanism.
– The systems are hard to design so, that the settings are completely detached from basic operator-independent functionality.
– It is hard to make uniform controls for different kinds of devices that perform same functionality, for example different kinds of operator chairs.

Resulting Context

The pattern results in a system where personal settings for an operator can be stored and restored. These profiles are easy to change, when the operator shift ends.

Related Patterns

START-UP MONITOR provides a mechanism that can be used to make the setting adjustments during the system boot. CONFIGURATION PARAMETER VERSIONS pattern may be used to differentiate between parameter versions. Same kind of mechanism for web applications is introduced in Adaptive web applications [17].

Known Usage

The Windows operating system has different users. Every user has a default template for the user environment, which creates the basic usable environment. The user environment is customizable. The customizable elements include desktop style, background image etc. All setting changes are personal and do not affect other users. These settings are transferrable to other Windows PCs using for example an USB memory stick.

3.4 Opportunistic Delegation

Context

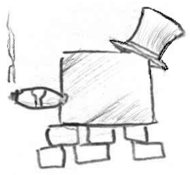
There is a distributed embedded control system where the ISOLATE FUNCTIONALITIES pattern has been applied. There are multiple autonomous subsystems executing similar tasks in the system, and they have constantly changing state such as location, acceleration or direction of movement. Tasks should be delegated to these autonomous subsystems but it is difficult to follow all the state transitions occurring in the system. Therefore, a already delegated task might be something that can not be done immediately by the subsystem. For example, it might require unsafe de-acceleration.

Problem

How to delegate tasks in a system where the situation can be in transition?

Forces

- *Response time*: The system as a whole should serve its users as fast as possible.
- *Throughput*: The system as a whole should optimize the execution so that as many tasks as possible are executed concurrently.
- *Scalability*: It should be possible to add new autonomous subsystems to ease the task load in the system. The adding process should not require major modifications to the system.
- *Resource utilization*: Resources should be utilized efficiently so that the task load is divided equally for all autonomous subsystems.
- *Testability*: The system task delegation process should be traceable.
- *Fault tolerance*: Error in one autonomous subsystem should not prevent the system as a whole from functioning.



Solution

A traditional way for delegating tasks is first to ask who can do the task, ponder on the answers and then give the task to a free party. When the state is changing continuously, the answer which was acquired by the question is already old by the time it is received. Decision making takes additional time and therefore makes the information even more antiquated. For example, an elevator going down would be able to stop on certain floor when it is first asked about it, but due to the slow decision process, it has passed the floor by the time the request to stop is received. This makes the whole system work slower than what it would be capable of.

In addition, in if a subsystem is given a task, which it will not be able currently to do, it will have to do it later. At the same time there might be some other subsystem, which could do the task immediately without any problems. If the subsystem can decline from taking the task, a new ask-decide-delegate cycle has to be run and this consumes additional time and the system status may change again during this cycle. In heavily loaded system this might lead to vicious cycle where only a few requests are served as the situation changes all the time.

As a solution in a system where the state changes constantly, decision power and tasks should be delegated. This is done by combining concept of task ownership and opportunistic task delegation. The basic idea is to give a task straight to a subsystem, hoping that it can take care of it. The subsystem knows what is its state and if it can do the task. Therefore, asking for this information separately is just waste of time. If the subsystem is able to do the task, it will take the task ownership and do it. If the subsystem is not able to do the task based on its state, it will decline to take the task ownership. This typically occurs when "point of no return" has passed, for example, an elevator is not able to stop for given floor due its speed. If the delegation failed or task was not accepted, the task can be given to some other subsystem.

Task delegation can always be carried out randomly but for more efficient system smarter delegation is required. Such a system is illustrated in Fig. 11. Information such as last accepted tasks, periodically sent data by the subsystem or, if nothing else is available, polled state information can be used. Available information should be used in as smart way as possible. One solution is to divide the subsystem states

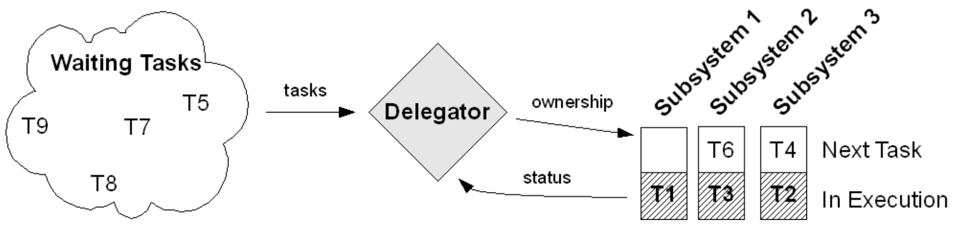


Fig. 11. Three independent subsystems with a delegator component which assigns task to them.

into three groups: states where it was, states where it might be and states where it will or could go to. When this division is known, those subsystems which are able to fulfill certain requests are known.

It might be possible to queue tasks to the subsystems, but if the system state is in constant transition just-in-time allocation might be more optimal. Instead the process could be further enhanced by anticipating future requests and allocating resources accordingly. This kind of system can be build to support different kinds of subsystems, some faster than others. Without task delegation, the subsystems would not have anything to do. Therefore, the delegator components is a single point of failure in the system.

Consequences

+ Resources are not wasted when a task is not given to a subsystem which currently can not execute the task.
+ Optimized concurrent operations are possible.
+ The task ownership makes handing over the responsibilities clear.
– It might not be easy to build smart and optimized concurrent delegation process.

Resulting Context

The pattern results in a distributed system where autonomous subsystems receive opportunistically delegated tasks.

Related Patterns

LIMP HOME describes how a system can still function in degraded mode – in this case this could be used to ensure operations in case of delegator component failure.

Known Usage

There are multiple elevators in the same building. A central component is responsible for elevators allocation. An elevator is going towards street level near the second floor. A button is pressed on the second floor and the controller assigns a pickup for the elevator for this floor. The elevator declines as it is not able stop for the floor. The controller then delegates the same request for another elevator which is passing fourth level. This elevator is able to stop for the second floor and therefore picks up more passengers before continuing to street level.

3.5 Third-party Confinement

Context

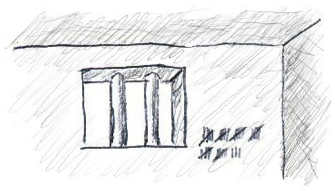
There is a distributed embedded control system where SEPARATE REAL-TIME has been applied. Real time part is separated from high-end functionality. There are third-party applications, such as fleet management and navigation which require a lot of processing power. The third-party applications should not compromise the system functionality.

Problem

How to cost-efficiently provide a generic and safe platform to run third-party applications?

Forces

- *Efficiency*: Third-party services may need a lot of processing power.
- *Safety*: There is a need to run third-party applications that can not be trusted as they may interfere with the machine functionalities.
- *Maintainability*: Development and maintenance of third-party services should be possible on COTS hardware.
- *Extendability*: It should be simple task to add new third-party applications to the system.
- *Reusability*: It should be possible to use same third-party applications in different products.
- *Installability*: The machine operator should be able to install preferred third-party applications.
- *Maturity*: There should be a place to run immature applications without risk of interfering with the well-tested machine functionalities. These applications may be third-party or in-house developed.



Solution

There is usually a need to offer third-party applications, such as mapping, to improve service quality for the machine operator. These applications rarely have real time demands but may require a lot of CPU time. In many cases such third-party applications are readily available and therefore make rapid development possible. Often their behavior in all situations can not be guaranteed and they should be isolated. Thus, a component providing a platform for the third-party application is added to the system. This component is placed at a high-end node to isolate it from the machine control functionality. Third-party applications reside inside this component in order to isolate them from machine specific high-end functionality.

The third-party applications communicate with the rest of the system via machine vendor provided interfaces. These interfaces should be mature and reliable. The application platform should have enough processing power to support resource intense third-party applications. The other option is to limit the available resources. A common solution to provide this service is to use PC as the high-end node.

Consequences

+ Third-party applications improve user experience for the machine operator.
+ New third-party applications can be easily installed.
+ It is easier to create third-party applications for COTS hardware as COTS operating systems, application development tools, and libraries can be used.
+ Third-party applications do not interfere with other machine functionalities.
− The high-end node creates a single point of failure.
− Confinement may cause latency that may not be acceptable for some third-party applications.
− The interface between machine and third-party application is challenging to design. It may cause maintainability issues and it may unnecessarily restrict third-party functionalities.

Resulting Context

The pattern results in a system with a capability to provide a restricted environment for third-party applications.

Related patterns
QUARANTINE [13] presents an error confinement system.

Known Usage
Machine vendor provides a subcontractor an interface that is used to implement remote diagnostics application. Remote diagnostics application uses the interface to acquire diagnostics data, such as oil pressure and diesel consumption. Subcontractor creates an user interface that is implemented on PC using QT. This user interface can provide different sensor values and production information for the machine operator. Remote access on PC is used to provide work plans in the beginning of the shift and send production reports to organization's database after the operator's shift.

3.6 Static Resource Allocation

Context

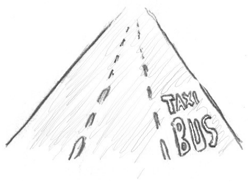
There is a distributed embedded control system where SEPARATE REAL-TIME has been applied. There are some critical services with strict response time requirements in machine control level. However there is a limited amount of resources (e.g. memory, message channels, processor time) available. The resources must be allocated when needed.

Problem

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

Forces

- *Resource utilization*: There are limited amount of the resources available and thus the resources must be allocated before using them.
- *Safety*: There must always be enough resources for the critical services.
- *Availability*: Critical services should be available immediately.
- *Response time*: Because of strict response time requirements, critical services can not wait for free resources.
- *Predictability*: Maximum amount of resources needed for critical services are known in advance.



Solution

Frequently, there are services in the system that can be considered as critical for the system functionality. These critical services are usually related to safety of the system (e.g. for sending emergency EMCY messages in case of fatal errors). As the system handles resource allocations in a dynamic way, it is not always guaranteed that a resource (such as memory, time slots for processes, bus resources) is available when needed. Therefore, all the resources needed for critical services should be pre-allocated during the system startup and not deallocated afterwards (i.e. the resources are fixed for the service). In this way, the resource allocation for the critical services is isolated from the dynamic resource allocation for the regular services. In addition, the critical services can not never run out of resources in run-time.

Fortunately, it is often possible to interleave resource utilization for regular services so that the shared resources are deallocated before some other service is allocating them. However, a critical service may be triggered by an unexpected event, thus making the exact execution moment unpredictable. Because of strict response time requirements, the critical service can not wait for resources to be free to be allocated.

Fixed resource allocation means usually also larger resource consumption as the resources allocated for critical services can not be utilized for other functionality. Thus, the amount of fixed allocated resources should be as small as possible. In addition, the resource allocation is based on worst case scenarios, so the number of critical services should be minimized. The mechanism can also be applied to all processes, so the resources are not shared. This increases amount of resources needed for the system, but increases response time for all the processes.

Consequences

+ Critical services will not run out of resources.
+ Critical services will not have to wait resources to be available.
– Less resources are available for regular processes as part of them are fixed for critical services.
– It might be difficult to know in advance all the resources needed for a critical service. Usually, the resources must be allocated for worst case scenarios.
– It might be difficult to identify critical services.
– Fixed resource allocation means usually increased resource requirements.

Resulting Context

The result is an embedded control system where all the resources needed for critical services are allocated at startup to ensure that they will not run out of resources.

Related Patterns

EARLY WORK can be applied if resource allocation process in run-time makes the real-time behavior unpredictable. STATIC SCHEDULING can be used to schedule pre-allocated processes. START-UP MONITOR can be applied to control resource allocation at system startup. FIXED ALLOCATION [16] describes similar way to statically allocate memory. It can be used to ensure that the system will never run out of memory.

Known Usage

In a machine control system, child nodes send emergency messages (i.e. EMCY messages) to the main node via bus in case of fatal error. To ensure messaging capacity for EMCY messages, a messaging slot is 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.

3.7 Static Scheduling

Context

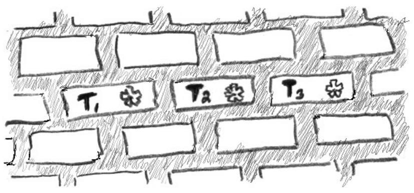
There is a distributed embedded control system where STATIC RESOURCE ALLOCATION has been applied. The system functionality requires several processes. The scheduling can not be pre-emptive because of strict real-time requirements. However, some tasks may need more frequent repetition and some tasks may require more processing time.

Problem

How to make sure during compile time that every task has enough CPU time and processes are scheduled with a predictable time behavior?

Forces

- *Predictability*: Due to interrupts and varying latencies, pre-emptive scheduling makes timing calculations hard.
- *Accuracy*: To operate accurately, there must always be enough CPU time for all functionality.
- *Real-time behavior:* A scheduling fault should never occur as it may lead to system malfunction.
- *Real-time behavior:* A node may have hard real-time requirements, thus requiring very accurate scheduling.



Solution

All resources are reserved during system startup using STATIC RESOURCE ALLOCATION. The system is divided into executable blocks, which have their own execution cycle (e.g. every 10 ms). During the compilation, the compiler divides the blocks into the scheduling time levels (e.g. 1 ms, 10 ms, or 100 ms). This is done by calculating the execution times for every block using known execution times of the CPU instructions and given execution times for the operations used. Each scheduling cycle (e.g. 1 ms) executes code from the lowest and from one other execution cycle. The scheduling is synchronized using synchronization pulse, which starts each scheduling time slot (see Fig. 12). In other words, the compiler schedules the program statically.

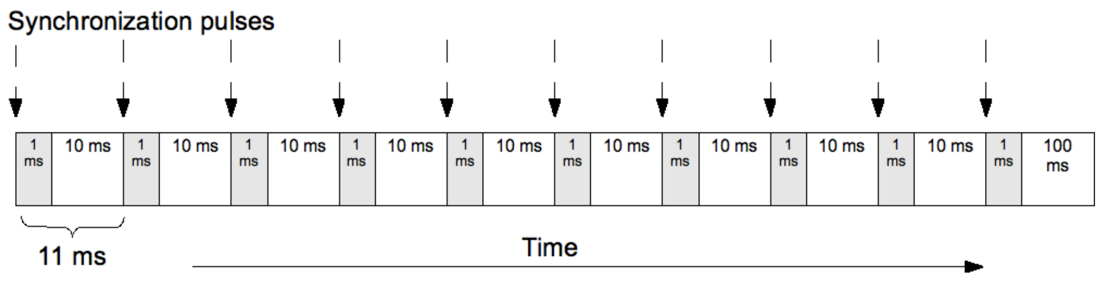


Fig. 12. Example of static scheduling with 1, 10, and 100 ms block size.

As there are no other external interrupts except synchronization pulse, the scheduling is always accurate. In addition, there is no need to make preparations for unsuccessful scheduling as the compiler schedules the system. However, dynamic processes can not be created during run-time as the scheduling has been fixed even before the system starts. Altogether, the static scheduling makes it possible to implement hard real-time software as all execution times are predictable.

Consequences

+ When all processes are scheduled statically, execution time is predictable.
+ There is no need to make preparations for unsuccessful scheduling. Preparing for unsuccessful process scheduling would require more memory space and processing power.
+ The pattern makes it possible to implement hard real-time software.
– Dynamic processes can not be created during run-time.
– It might be impossible to calculate execution times for operations in all cases.

Resulting Context

The pattern results in a distributed embedded control system where all processes are scheduled with predefined time slots without task overruns.

Related Patterns

CENTRALIZED SYNCHRONIZATION can be used to synchronize execution between multiple units for increased accuracy.

Known Usage

Processes are executed using an interpreter in an power plant control system. When the processes are compiled, the scheduling is defined based on execution time of each instruction and measured timing information for operations. For each time level, the total execution time is calculated. The total time is divided into time slots. The interpreter executes each process for one time slot.

3.8 Limp Home

Context

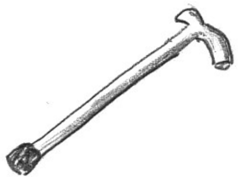
There is a distributed embedded control system where SEPARATE REAL-TIME has been applied. There are sensors and actuators in the system that may malfunction. The malfunction can be caused by physical damage or software bug, etc.

Problem

How to be able to operate the machine even when a part of it is malfunctioning?

Forces

– *Availability*: The basic functionality of a machine can be enhanced with special add-on devices. The basic operations should be available to some extent even if the add-on devices malfunction.
– *Fault tolerance*: If sensors malfunction, it may still be possible to use the machine. Therefore a malfunction in nonessential sensor or actuator should not disable the whole machine.
– *Operability*: A malfunction in a not-important sensor or actuator should not make the machine completely inoperable. The operator should be able to e.g finish the shift or drive the machine from the field to a service station or to a nearest road.
– *Safety*: In some cases of malfunction it is more convenient or safer to continue operation in a degraded service mode than to stop all operations.
– *Robustness*: The machine should adapt to situation where all devices are not available for use.



Solution

Most of the functionalities in the machine are controlled in computer-aided manner. This automatization requires information from various sensors on machine control level. Some functionalities can be controlled manually without the information from sensor. Therefore sensors and actuators should be divided into groups according to the criticality of the device. Some device groups are mandatory to a functionality and some are optional as they only enable the operator to use enhanced computer-aided systems. When a sensor in a group malfunctions, the machine makes a state transfer to limp home mode if group was optional. If device group is mandatory, the functionality is disabled. This may or may not disable whole machine, depending on the criticality of a group.

In limp home mode the responsibility of controlling a functionality, which was automatically controlled using sensor information or actuators malfunctioned, is transferred to machine operator. For example, if a sensor in a sensor group that is used to automatically control the drill in a mining drill malfunctions, the responsibility of controlling the drilling process can be transferred to machine operator. In this way, experienced machine operator can operate drill just by listening to the sounds from the motor. Of course, the decision whether to continue working with risk of breaking down the drill is left to the machine operator. It should be noted that a malfunction in the sensors that affect the drilling does not affect other functionalities of the machine.

Consequences

+ The machine does not become completely inoperable when some sensor or actuator malfunctions.
+ The solution increases safety as the machine can be operated to some extent even if a part of it is malfunctioning, e.g drill can be driven out of mine tunnel when the tunnel is about to collapse.
+ The solution reduces costs as the machine can be driven to nearest repair point, instead of repairing it in the field.
– Deciding appropriate device groups can be challenging. Device groups should not be too large or small.
– Deciding which parts of the machine can still be operated in each situation can be hard as the possibility to continue may depend on other sensors that could be damaged as well.

Resulting Context

The distributed embedded control system is not completely inoperable when a device malfunction occurs as they are divided into functional device groups. The system may be operable in a degraded service mode even if there is a malfunction in sensors or actuators.

Related Patterns

NEXT STABLE STATE pattern can be applied after this pattern to provide a stable state where the machine can automatically limp to. WATCHDOG and HEARTBEAT can be used to detect a malfunction in sensor or actuator. In addition, when LIMP HOME can not be used as critical device is malfunctioning, the system could change to SAFE STATE.

Known Usage

In a elevator system of a skyscraper there are call buttons on each floor. If a malfunction occurs and call buttons are disconnected, the passengers are unable to call the elevator to desired floor. The elevator should notice this malfunction and start round-robin stopping to all floors. Round-robin stopping means that the elevator cabin drives from a floor to the next floor and stops, opens doors and lets passengers in and out. When the lowest or the highest floor is reached the cabin changes direction. If there are more than one cabin serving floors, the elevator cabins can negotiate to which floor each of them will stop. In this way the elevator system can still offer service for the passenger even if it is operating in degraded service mode.

3.9 Isolate Control Algorithm

Context

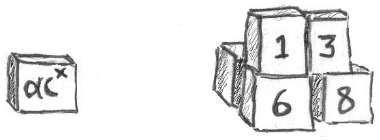
There is an embedded control system where ISOLATE FUNCTIONALITIES have been applied. There are functionalities which are controlled with special control algorithms. For a single functionality there could be a variety of choices for control algorithm. One algorithm is better than others depending on the situation. In addition, a single control algorithm may consist of multiple parts. These parts of a control algorithm may need to be changed during the life cycle of product or in different use cases.

Problem

How to make the control algorithm easily changeable?

Forces

- *Testability*: It should be possible to test the control algorithm separately or run the control algorithm on a PC while other parts of the software run on the target hardware.
- *Testability*: It should be possible to replace the control algorithm with a stub.
- *Analyzability*: The control algorithm should be located in one place to make it understandable and to make it easily modifiable.
- *Changeability*: Control algorithms or parts of them should be replaceable, even at the run-time. The optimal way of controlling of a functionality can change depending on the use situation.
- *Maintainability*: If a sensor is changed to another, it should require only minimal or ideally no changes in the control algorithm.
- *Variability*: Different devices and products may require different control algorithms. The algorithm should be easily changeable to support different kind of hardware environments.
- *Reusability*: The same control algorithms can be used in many products or in different product generations. Therefore the control algorithm should be easily reusable.



Solution

There are devices in the system that need to be controlled with algorithms. An algorithm can be monolithic or it can consist of multiple parts that should be separately changeable. Therefore, the control algorithm or its parts should be isolated in a single place, so that the algorithm can be easily changed - even at run-time. Furthermore, this principle can be applied to different subparts of an algorithm as well. For example, in forest harvester parts of the feeding algorithm may need to be changed according to the tree type. Algorithm isolation is also beneficial when the hardware environment changes and the change propagates modifications to the control algorithm. It is easy to make changes when the algorithm resides in a single place.

The control algorithm should be abstracted with an interface and there should be a way to parametrize the algorithm. Parametrization is important as there can be different hardware configurations that need to be supported. In many cases, the control algorithm needs to read information from various sensors. It should be noted that the control algorithm should not read the value directly from the sensor as it will make the algorithm to function only with that sensor type. Therefore control algorithm should use abstraction layer to read the sensor values.

Consequences

+ Control algorithm or parts of it can be changed at run-time.

+ Control algorithm can be tested separately as it is isolated from the rest of the system.

+ Different kinds of control algorithms are easy to import to the system. Algorithms can be imported, for example, from simulation environment such as Matlab. This enables easy prototyping with different kinds of algorithms.

+ The solution minimizes the required changes in control algorithm when hardware environment changes.

+ An algorithm controlling a functionality can be located in single place in the code. This makes it easy to follow and modify the algorithm implementation.

– It can be hard to find minimalistic interface for the control algorithm that offers everything that is needed but nothing else. The future changes are likely to propagate changes in the interface. Same applies for the interface towards hardware.

– It can be hard to design the algorithm so that it is hardware independent.

– Abstractions add delay and therefore decreases performance.

Resulting Context

The pattern results in a distributed control system where algorithms are isolated behind interfaces. Isolated control algorithms are easier to modify and replace than algorithms that are not separated from the rest of the system.

Related Patterns

The interface to hardware can be implemented for example using using VARIABLE MANAGER or HARDWARE ABSTRACTION LAYER.

Known Usage

Boom control algorithm is isolated using appropriate interface. All sensor data can be accessed by using solution introduced by VARIABLE MANAGER, so direct communication with sensors is not required. Different kinds of control algorithms for the boom are designed. Machine operator may want to use different control geometry, e.g. spherical, cartesian and controlled valve-by-valve, for boom depending on the situation. When control algorithm is isolated, different kinds of control algorithms for the boom can be implemented. In this way, the machine operator may change to controlling logic of the boom during run-time by pressing a button from dashboard.

4 Patterns for Fault Tolerance

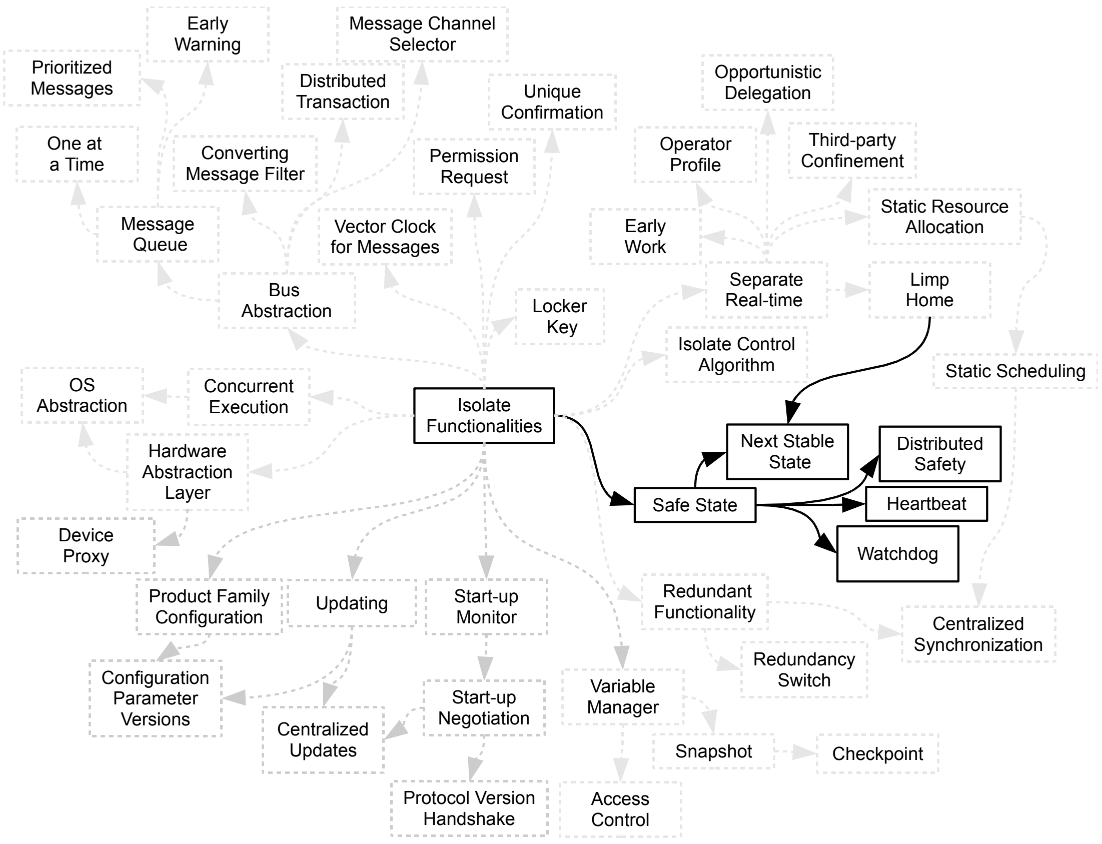


Fig. 13. The sublanguage for fault tolerance in distributed machine control system.

Table 3: Pattern thumbnails

Pattern Name	Description	Page
SAFE STATE	If something potentially harmful occurs all nodes should enter a predetermined safe state.	57
NEXT STABLE STATE	When a potentially harmful situation occurs, the system can still provide a last transition to the next stable state.	59
DISTRIBUTED SAFETY	Potentially harmful functionality is divided into multiple nodes in order to prevent a single node from creating havoc.	61
HEARTBEAT	The node uses periodic messages to ensure proper communication with other nodes through bus. This enables recovery actions to be taken as soon as possible.	63
WATCHDOG	The health of a node is monitored. If the node malfunctions, a recovery action can be taken.	65

4.1 Safe State

Context

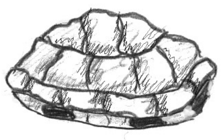
A machine control system has been distributed using ISOLATE FUNCTIONALITIES pattern. A single node can malfunction and prevent the safe operation of the machine. Other nodes should react to the malfunction of the single node by stopping functions which could compromise the safety. Furthermore, sometimes halting all functions may compromise safety as well.

Problem

Minimize the possibility that operator, machine or surroundings are harmed when some part of the machine malfunctions?

Forces

– *Response time*: Possible protective (in)action should be immediate.
– *Analyzability*: Functionality in safe state should be understandable and traceable.
– *Testability*: Emergency, fault, etc. situation should be testable.
– *Safety*: Operator, machine or surroundings should not be harmed.
– *Fault tolerance*: When a fault occurs, there should be mechanism that prevents further damage.
– *Recoverability*: It should be possible to continue operation after emergency situation occurs in predictable way.
– *Stability*: Safe state should be stable.



Solution

When a machine with moving parts malfunctions, it can cause harm when let to move uncontrolled. Therefore, it is needed to prevent the harm by entering safe state, for example, by stopping the moving parts. The safe state is device and functionality dependent and it is not necessarily the same as unpowered state, i.e. working brakes prevent movement to downhill and stable hydraulic pressure keeps lifted objects in the air. Sometimes determining the correct safe state can be challenging. Each actuator should know at any given moment which is the safe state that should be entered if a malfunction occurs. Transition to safe state can also be made when machine operator presses the emergency stop button.

When a malfunction occurs, every device is commanded by its controller to enter predetermined safe state. Malfunction can be detected for example, by using solution provided by the HEARTBEAT pattern. Sometimes the node can also detect itself that it is malfunctioning. Recovery from the malfunction can be initiated by machine operator who repairs the malfunctioning part and restarts the system. In addition, maintenance personnel should be able to start recovery from safe state from maintenance user interface.

Once the malfunction is solved, return to normal operation from safe state should be also handled in safe fashion, for example, so that a grappler does not open and drop lifted objects. When device enters the safe state it should inform other nodes about the situation. This can be done for example, by sending emergency message to the bus. Other nodes should react to this by entering their safe state. In this case, other nodes does not necessarily need to send another emergency message to the bus. Furthermore, sometimes it is required that multiple controllers enter the safe state in proper order so that the safe state can be achieved. This can make the solution more complex.

Consequences

+ The predetermined safe state can save lives, prevent accidents and material losses.
– The safe state is not necessarily the same in all cases and therefore it can be hard to determine which is the safe state. For example, safe state of frame controller can depend on the location of the machine. If the machine is in slope, the controller should not release hydraulic pressure. However, on flat surface, the hydraulic pressure can be released.
– There can be dependency chains between controllers complicating the transition to safe mode.

Resulting Context

The result is a distributed and scalable machine control system where controllers can enter the devices into safe state.

Related Patterns
By using HEARTBEAT and WATCHDOG, situations where safe mode might be needed can be determined. Partially working machine can continue in degraded mode in some cases with LIMP HOME in order to reach state described by NEXT STABLE STATE.

Known Usage
An object grabbed by clamps in a boom moves in high speed towards the cabin of the machine. The feed controller detects this potentially dangerous situation and enters its safe state and feeding is stopped. The feed controller sends an EMCY message to the CAN bus. Other nodes gets the message and enter their safe states. For example, boom controller does not release hydraulic pressure, but keeps the boom in the air. If the boom would be released it could also cause harm for the environment or the machine itself. The machine operator is informed about the situation on the screen in the cabin. Operator can evaluate the situation and may continue working by restarting the system.

4.2 Next Stable State

Context

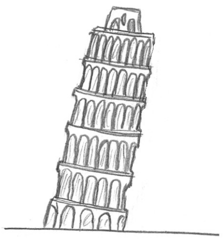
There is a distributed embedded control system where Safe State has been applied. There are devices in the system which can malfunction and make a state transfer to safe state introduced by the Safe State pattern. Nevertheless, it is not convenient in every situation to stop operations as end user may find additional value if the machine continues operating to some extent.

Problem

How to offer some level of service to the end user even if the system or part of it is malfunctioning?

Forces

- *Suitability*: In all cases it is not convenient to stop all operations completely when a malfunction occurs as there is a change that some services can still be provided.
- *Operability*: If it is possible, all operations should not be stopped if a part of the system is malfunctioning. It should be possible to drive the machine to a location where repairing is possible or repairing can be carried out safely. For example, forest harvester can be driven out of the forest for repairs, even if it is partly malfunctioning. This could mean that the driving speed is limited.
- *Usability*: All operations should not be stopped immediately as it may cause inconvenience to users. For example, an elevator cabin should be driven to next floor if it is safely possible.
- *Adaptability*: System should know how to react to changes in the operation environment. If characteristics of normal or safe operation range are exceeded, the system should know to which extent it can still offer services.



Solution

When a sensor or actuator malfunctions or system exceeds expected operation range, it may not be safe or convenient to continue operating. A system that uses solution introduced by Safe State will enter the safe state in such a situation. This usually stops operations. Nevertheless, it is not always the best thing to do. In some cases it may be more convenient for end users, or for some other reason, to offer still some services. For example, if elevator malfunctions between floors, it is more convenient for passengers the elevator to drive to next floor and let passengers out. In this case, the possibly required repairs are more convenient to carry out in the floor level as well. However, it is not trivial to deduce if the operation can be continued, or should the system move to a safe state when a malfunction occurs.

In safe state, the system then can decide if further measures should be taken (Fig. 14). If the system notices that it may still offer some services without causing any danger, it can make transition to the next stable state. Next stable state is more desirable than the safe state where all movements and operations were initially stopped when a malfunction was detected. However, the system should be know for sure, that transition to next stable state is safe. System can use Limp Home to realize the state transfer from safe state to next stable state. For example, if drilling controller in mining drill malfunctions, it enters the safe state immediately. This safe state transfer is propagated to other devices in the system as described in Safe State pattern. This will result the machine to stop all operations. The system may then inform the end user and the end user can deduce that only the drilling unit is malfunctioning. The operator then overrides the malfunction and the machine offers operator a possibility to drive the machine out from mine shaft. Fig. 14 demonstrates the state transfers of the machine. The system can make system wide decisions in the safe state that will result the state transition to the next stable state.

Consequences

+ System may still offer some convenience services even if a part of it has malfunctioned.
– The solution adds complexity to system design as the system must always know which is the next stable state.
– Reasoning logic, if the state transition to next stable state can be made, may be complex.

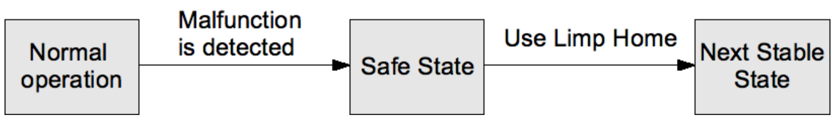


Fig. 14. State diagram of how next stable state is reached.

Resulting Context

The result is a system that can offer some some level of service for the end user even if a part of the system is malfunctioning.

Related Patterns

SAFE STATE defines a state that system should enter when a malfunction occurs. This state can be used to get time to decide which actions are done in next stable state. LIMP HOME can be used to move the machine to next stable state from safe state.

Known Usage

High skyscraper elevator has sensors which inform the elevator system how much the wind is swaying the building. The elevator cabin always knows the next stable state (the next floor) where it can be stopped if the building start to sway too much. The elevator would stop to the next floor also if the sway sensor malfunctions. In addition, if the speed sensor of the elevator cabin malfunctions, the cabin enter limp home mode and moves very slowly to the next stable state, i.e. next floor. When the next floor is reached the elevator lets passengers out and stops all operations. In this way, no inconvenience to the passengers is caused, i.e. passenger are not locked in the elevator cabin, even if the elevator malfunctions.

4.3 Distributed Safety

Context

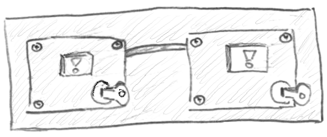
There is a distributed embedded control system where ISOLATE FUNCTIONALITIES has been applied. There is a functionalities in the system that move physical parts of the machine. If such functionality is not controlled properly, it may cause harm or damage to the environment, people, or to machine itself.

Problem

How to minimize the risk that a failure of a single node causes harmful situations?

Forces

- *Safety*: A malfunctioning node should not cause harm or damage to environment, people or to machine itself.
- *Testability*: A fail-safe mechanism is needed because all possible failure situations can not be covered in testing.
- *Redundancy*: Backup nodes or units may not be economically plausible or might not solve all risks as hot unit, i.e. the unit controlling a functionality, can cause harmful situations alone.
- *Redundancy*: Voting whether to take action is not plausible as there is not multiple units implementing the same functionality.



Solution

In order to prevent a single node from causing harmful situation, the potentially harmful functionality is divided into multiple nodes communicating with each other. Each of these nodes, implement a part of the functionality. For the realization of a functionality harmonious functioning of all nodes is required. There can be multiple harmoniously functioning nodes, if required. The division principle of actions required for the functionality should be decided so that a single node can not cause harmful situations. Usually the division needs to be decided case by case depending on the functionality.

A functionality can only be used if all nodes function harmoniously together. For example, if two nodes are used to control a boom. The first node is used to control the valves related to the functionality and the second node is used to control the hydraulic pressure of the boom. The first node initiates the action. If the first node is malfunctioning, the second should not give hydraulic pressure. If the second node is malfunctioning, the first should use valves to keep the boom steady. Malfunctions of other nodes can be detected for example with the HEARTBEAT pattern.

One of the nodes implementing distributed safety takes the action when the functionality is requested. Then the node communicates with other required nodes to orchestrate the execution of the functionality. If all nodes agree, the action is taken. Otherwise the nodes may need to communicate more to find out that all nodes are sane, if WATCHDOG pattern is not used. This means that the other node tries to find out if the other node is malfunctioning. If the node is malfunctioning, the node should stop operations and enter safe state which is described in SAFE STATE pattern. In addition, the malfunction can be reported, for example, to the machine operator or through remote diagnostics to the product vendor. If WATCHDOG is used, it detects if a node is malfunctioning and restarts it. If the communication channel between nodes implementing a functionality together malfunctions, the functionality is not executed as all parties involved in the functionality can not be reached.

Distributed safety mechanism requires that there are multiple nodes for implementing a functionality. In many cases this does not increase the costs as a part of a node required for some other functionality can be used to implement distributed safety. Redundant units are not necessarily needed. For example, controller A is responsible for implementing a functionality 1. A critical part of functionality 1 is distributed to controller B, which is responsible of functionality 2, in order to implement DISTRIBUTED SAFETY. Now if functionality 2 requires distributed safety as well, then a part functionality 2 can be distributed to controller A. In this way, no additional controllers are required.

Consequences

+ A single node can not cause harm or damage to surroundings because potentially harmful actions taken need cooperation with other nodes.

+ In many cases, the solution can be implemented using existing hardware.
+ In case of failure, a fail-safe mechanism will prevent the potentially harmful situation.
– A functionality divided into multiple nodes is more error-prone. The risk of losing functionality increases because probability of the hardware failure increases.
– The system may become less efficient as communication is needed between nodes implementing distributed safety.
– Understandability of the system decreases as a functionality is spread over multiple nodes.
– If additional nodes are required in the system, solution of this pattern may cost more than normal solution.

Resulting Context
The result is a distributed embedded control system where a harmful action of a single node is prevented because it needs cooperation of multiple nodes.

Related Patterns
WATCHDOG (another version of the pattern can be found from [13]) pattern can be used to monitor malfunctions of nodes which implement distributed safety. HEARTBEAT (another version in [13]) can be used.

Known Usage
Boom control is divided into two separate nodes. Boom controller does computing that is related to functionalities such as boom kinematics calculations. When boom is controlled to move, the boom controller asks for hydraulic pressure from another controller. In other words, the controller can not move the boom by itself, since it needs hydraulic power which is controlled by another controller. If the controller of hydraulic power does not respond, no actions are taken as there is no power. If the controller gives too much pressure, the boom controller can stop the movement. Hydraulic power controller can determine the sanity of boom controller from information on how much pressure was asked for. If this is the case, then the hydraulic power controller should not give any power.

4.4 Heartbeat

Context

There is a distributed embedded control system where the ISOLATE FUNCTIONALITIES pattern has been applied. The system consists of multiple autonomous nodes, cooperating with communication over some kind of channel. The communication channel or the nodes may malfunction, compromising the functionality of the whole system.

Problem

How to notice that the nodes or the bus will not silently fail?

Forces

- *Fault Tolerance*: A node may crash and other nodes should notice the situation as soon as possible to prevent the fault from propagating.
- *Fault Tolerance*: The communication channel may break and the end nodes should notice the situation as soon as possible.
- *Availability*: If a node or a channel malfunctions, it should be noticed quickly to minimize downtime.

Solution

To prevent the situation where other nodes do not notice a failure of a single unit or the communication bus, the nodes must communicate regularly. The messages act as a proof that the communication channel and the sender is healthy. Therefore, the nodes should send messages of their health at predetermined and regular intervals. This message is called the heartbeat of the monitored system. The messaging interval depends on the system that has to be monitored. For example, transmission controlling unit may have higher messaging rate than a boom controller. The information of the sent message may be just to indicate that the node is alive. Sometimes the reports may be part of the normal activity, if the node has to send cyclic messages anyhow.

There are two variants of the reporting mechanism: the monitoring node may request for the heartbeat or the system may autonomously send a notification. The listening end may be a centralized node or the heartbeat signal may be used point-to-point. If the listening node does not get any reports it will assume that the monitored node has failed and start actions that try to normalize the situation or prevent further damage. For example, system may boot itself or notify the user to repair the situation. During the failure, the whole system may enter some safe state (see SAFE STATE) in order to prevent the failure from propagating.

There are two variants of the reporting mechanism: there may be a node that may request for the heartbeat or the system may autonomously send a notification. The listening end may be a centralized node or the heartbeat signal may be used node-to-node. If the listening node does not get any reports it will assume that the monitored node has failed and start actions that try to normalize the situation or prevent further damage. For example, system may boot itself or notify the user to repair the situation. During the failure, the whole system may enter some safe state (see SAFE STATE) in order to prevent the failure from propagating.

Consequences

+ Noticing malfunctioning nodes or communication channels early results in higher error recovery.
– It is not always clear whether the bus or the monitored node has failed. This may affect to the corrective actions.
– The heartbeat messages add more traffic to the bus, if it is not piggybacked to the normal messages.

Resulting Context

The pattern results in a distributed system with a monitoring system that helps nodes to notice if a node or the bus crashes.

Related Patterns

Hanmer et al. have the HEARTBEAT [13] pattern which is a description of similar pattern. The WATCHDOG [13] pattern and the WATCHDOG pattern in this pattern language describe a mechanism that can be used within units to monitor crashes and restart them to increase availability. SAFE STATE or LIMP HOME are possible solutions for following actions after a failure has been detected. In environments that use REDUNDANT FUNCTIONALITY, HEARTBEAT may be used to decide which node is the active one. REDUNDANT FUNCTIONALITY also may provide a backup channel for communication in case of failure.

Known Usage

Internet communication utilizes a tool named Ping that can be used to test nodes or the communication channel. The sending node requests the other node for a response with Internet Control Message Protocol echo request packet. If a reply ("pong") is received, the communication channel and the other node are healthy. The protocol is illustrated in Fig. 4.4. The host A sends ping messages to the host B, and the host B must reply to each of the ping messages with a corresponding pong message.

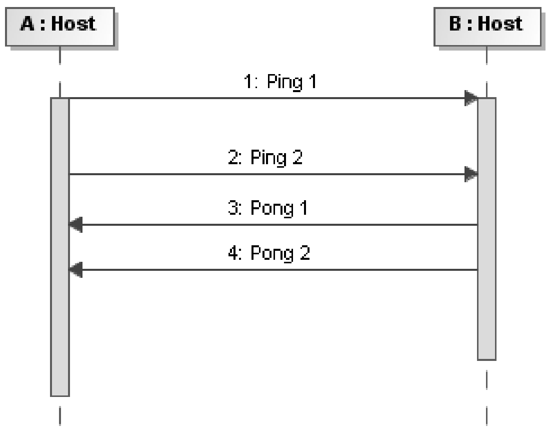


Fig. 15. Illustration of Internet Control Message Protocol echo request between two nodes.

4.5 Watchdog

Context

There is embedded control system nodes which have some functionality. The nodes are result of using ISOLATE FUNCTIONALITIES. The node may crash or deadlock, thus compromising the system integrity.

Problem

How to make sure that a node crash does not go unnoticed and a remedying action, such as reboot, is started as soon as possible?

Forces

- *Recoverability*: A node may malfunction because of programming errors. There is no way to ensure that no errors happen, so the system must be ready for them.
- *Response time*: A countermeasure for crash should be started as soon as possible.
- *Fault Tolerance*: There are some actions which may lead to a deadlock situation.
- *Availability*: All downtime should be minimized.
- *Cost Efficiency*: Redundant units adds cost. This may not be reasonable for all systems.

Solution

A watchdog component is added to the system to monitor the behavior of the node(s). The watchdog keeps track of the predetermined actions of the nodes. If the node reacts within a given time limit, the watchdog deems the node alive and working. If a node does not act within the timer threshold, it is supposed to be dead. After a node is pronounced dead, then the watchdog starts a remedying action to resume the normal activity of the task, for example, by rebooting the controller.

The watchdog may use nodes normal output to reset its timeout counter or the node may proactively reset the watchdog. Thus, if the node does not do the necessary actions to appease the watchdog in predetermined time, the watchdog tries to normalize the situation. This could in practice be implemented as a flag that the watchdog rises. It is the duty of the node to reset this flag in the given time limit. If it does not do this, the watchdog tries to rise the flag again and notices that the node has not changed it. It may then reboot the node.

Consequences

+ Quick restart results in higher availability.
– It may be hard to design the correct timing threshold for watchdog. Too short interval causes unwanted behavior and too long threshold hinders the response time for malfunctions.
– The watchdog must be completely independent from the main functionality, thus adding to the complexity.
– The correct remedying action for the watchdog may be hard to decide as simple reboot may lead to a boot loop. For example, broken hardware causes the system to continuously reboot.

Resulting Context

The result is a embedded control system node which has a mechanism for automatic recover after a malfunction occurs.

Related Patterns

Hanmer et al. have the WATCHDOG [13] pattern which is a description of a similar pattern. The Hanmer et al. HEARTBEAT [13] pattern and the HEARTBEAT pattern in this pattern language describe a mechanism that may be used to monitor system health. The remedying action for watchdog may be to enter some safe state, see SAFE STATE.

Known Usage

Almost all microcontrollers have a hardware watchdog. The watchdog has a timer register that causes a controller self-reboot when the register overflows. If the watchdog feature is activated, the software must actively reset the watchdog timer periodically to prevent the reboot.

5 Patterns for Redundancy

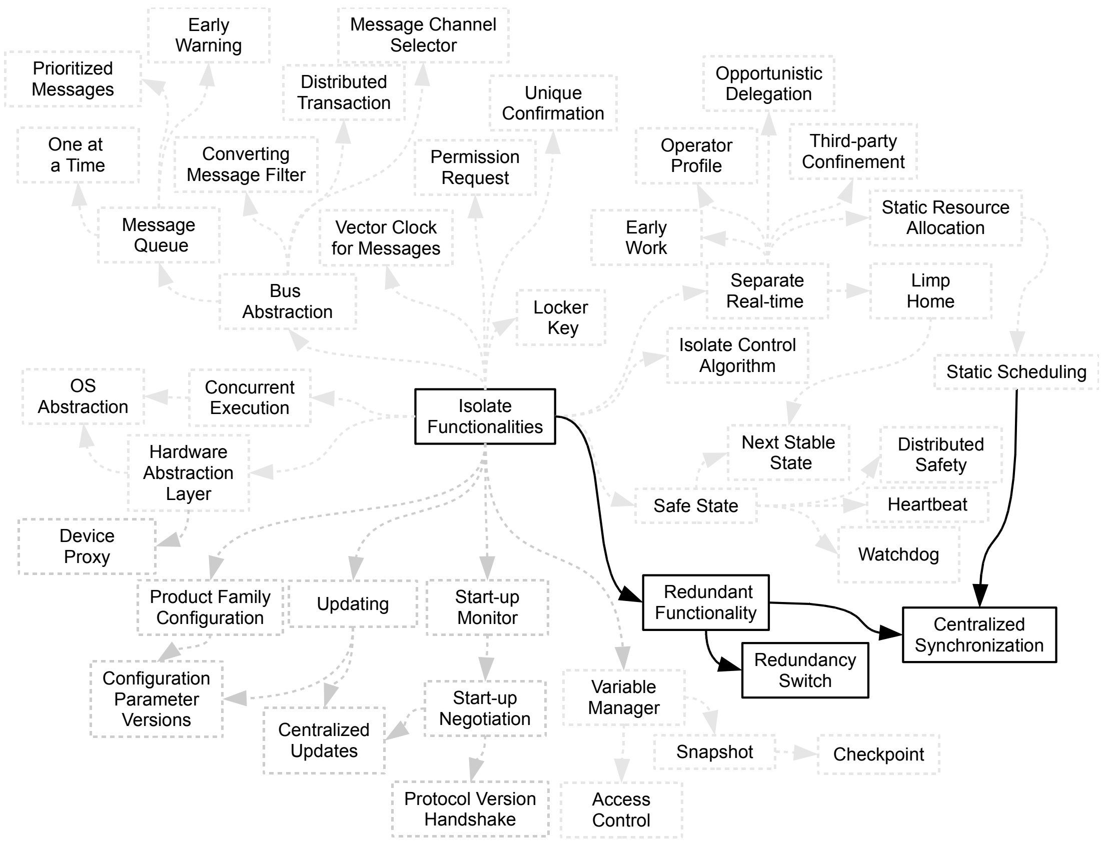


Fig. 16. The sublanguage for redundance in distributed machine control system.

Table 4: Pattern thumbnails

Pattern Name	Description	Page
REDUNDANT FUNCTIONALITY	A node controlling a functionality is multiplied to ensure the high availability. If the node malfunctions, a secondary unit will take over control.	68
REDUNDANCY SWITCH	There is one active unit and multiple secondary units in the system. When a active unit malfunctions a separate component decides which secondary unit takes over the control.	70
CENTRALIZED SYNCHRONIZATION	The program is scheduled statically during the compile time. During run-time an external clock signals the nodes to stay in schedule.	72

5.1 Redundant Functionality

Context

The system is distributed control system where the Isolate Functionalities pattern has been applied. The unit providing a service having high availability requirements. However, the hardware is likely to be less reliable than the availability requirements expect.

Problem

How to ensure availability of a functionality even if the unit providing it breaks down or crashes?

Forces

– *Availability*: Service should be available in all situations with only minimum downtime.
– *Fault Tolerance*: Quality of the service should not be affected if a unit providing it malfunctions or crashes.
– *Robustness*: There should always be someone controlling a functionality or service to prevent harm for environment or the system it self.

Solution

A unit monitors or controls functionality that needs to be controlled at all times or consequences can be harmful. Therefore a unit controlling such functionality needs to be duplicated or multiplied. In this way, the availability of the service is also increased. One of the multiplied units is responsible for controlling the functionality and other units are in a hot standby mode, i.e. those units are running the same process, but the output is not used to control the functionality. When the active unit fails, the inactive unit must be switched to be active so that the execution can continue on the redundant element. Switching over must occur as fast as possible to minimize the downtime caused by the switch over. The switch over should happen so fast enough, so that the functionality can not cause a harmful situation. Number of redundant units should be deducted from the malfunctioning frequency of the units.

The redundant units can monitor each other, e.g. with Watchdog or with Heartbeat. For example when redundant unit notices that the active unit is not responding, it can take over control immediately and boot the other unit. In addition, the active unit can boot the redundant unit if a malfunction is noticed. If the booted unit does not respond even after the boot up, some other measures can be taken in use, for example notifying system administrator that the unit must be replaced with new one. It is important that the monitoring between redundant units does not form single point of failure. In ideal situation the switch over does not cause any glitches to the control output. The controlling unit should synchronize the parameters, that are used to control the functionality or service, to redundant units in order to minimize downtime during switch over.

Consequences

+ Redundant unit results in higher availability.
+ Redundant unit can take control within short response time.
– Redundant unit increases system costs as there is redundant units in the system.
– Redundant units take more space, consume more electricity and need more maintenance. This increases costs.
– Depending on the criticality of the provided service, redundant unit may require synchronization with controlling unit to minimize downtime. This may require more processing power.
– Detecting the malfunction in the active unit may be challenging.
– Communication between redundant units increase complexity of the system.

Resulting Context

The result is a distributed system where a unit controlling a functionality has redundant units. If the active unit fails a redundant unit can be taken in use. In this way, the high availability requirements can be met and a functionality that may cause harmful consequences has always someone controlling it.

Related Patterns

REDUNDANCY SWITCH can be used to select the active unit from the group of redundant units. WATCHDOG and HEARTBEAT patterns can be used to detect malfunction in redundant units. In addition, Hanmer [13] describes a collection of error detection patterns that can be used to detect the failure of redundant unit. FAILOVER [13] pattern is a similar pattern as it describes how error-free execution can continue when the active unit of redundant units fail. REDUNDANCY [13] describes similar solution to increase availability.

Known Usage

A system that is controlling unit that compensates the deviations in the quality of electricity in electric network is duplicated in order to ensure the high availability of the service. The requirement for the availability is that the system must be available 99,999999 % of the time. Therefore there are two units that are both hot - one is controlling the functionality and the redundant unit is running as well, but the output is not used to control the functionality. If the active unit malfunctions the second unit takes over immediately. The active unit synchronize the control parameters with the redundant unit once in a minute, so that the redundant unit is ready to take over with no glitch in the service produced. There is a redundancy control unit (RCU) which is connected to two redundant control systems. The RCU is an intelligent optical switch which chooses the currently active controller according to the operating status of control systems.

5.2 Redundancy Switch

Context

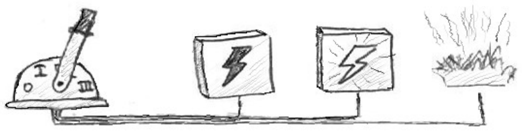
There is a distributed embedded control system where hardware is replicated using the REDUNDANT FUNCTIONALITY pattern to meet availability requirements. Redundant units are relatively error-prone. The system needs to recognize that the active unit's output is within acceptable limits. If the limits are not met, the unit is considered faulty and the system needs to choose a new replicated unit as active. The mechanism described by HEARTBEAT can not be used to detect a failure of a unit as it generates extraneous message traffic between units and may be too slow to detect the failure of an active unit.

Problem

How to respond rapidly to a failure in a unit and choose the active one from redundant units?

Forces

- *Availability*: A backup unit should take over in predetermined strict response time when the active unit malfunctions.
- *Reliability*: A unit can not make sanity checks for itself as self-tests may not be reliable enough or consume too much resources.
- *Reliability*: In a system with even numbered amount of units, voting is not an option as it may result in draw.
- *Resource utilization*: A unit may not make sanity checks for other units, because it may consume too much resources.
- *Scalability*: It should be possible to add new backup units to the system and use the same fault detection mechanism.
- *Resource utilization*: Fault detection can not be implemented using mechanism such as HEARTBEAT as it is not fast enough and the heartbeat messages from multiple units may congest the bus.
- *Fault tolerance*: Malfunction of monitoring mechanism should not interfere with the output of the active unit.
- *Response time*: Switching the active unit should be as fast as possible.



Solution

The system has redundant units that communicate with each other to keep their state consistent. One is used as active unit to control the process and others are hot standby units that can take over control when needed. A special monitoring unit or component is added to the system. The monitoring component examines the output of replicated control units. A monitoring component is configured so that it knows the acceptable limits of the output of controlling units. It takes outputs of all controlling units as inputs and examines the output and chooses which unit is used as active unit. The monitoring component then switches the output of active unit as system's control output.

By default monitoring component uses the active unit for output. When it detects that output is not within the acceptable limits, but outputs of redundant units are, it chooses one of them to produce output. The output can be changed using a hardware switch. After the control output is switched it can boot the unit which malfunctioned, in order to get it back online as a backup unit. If the rebooted unit does not produce output which is in acceptable limits, the monitoring mechanism takes the unit offline and may notify the system operator or administrator.

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

Consequences

+ Monitoring component makes the fault detection more reliable.
+ Monitoring component makes the fault detection and unit switching fast enough to meet high availability requirements.
+ System operator or administrator can be notified of unit malfunction.

– Monitoring component creates a single point of failure. In addition, the mechanism limits the possibilities to make a redundant monitoring component.
– If the active units output is in acceptable limits, the monitoring component does not recognize a malfunction.

Resulting Context
The result is a system where units have redundant units and the controlling which one of them is active unit is placed on separate component. Active unit can be switched within predetermined response time.

Related patterns
WATCHDOG can be used to detect malfunctioning units and restart them.

Known Usage
In an embedded control system, a redundancy control unit (RCU) is connected to two redundant control systems. The redundancy control unit is an intelligent switch, which chooses the currently active controller according to the operating statuses of the control systems. Alarm from the sanity checking system signals a malfunction and RCU is notified. RCU tells to backup unit to go active.

5.3 Centralized Synchronization

Context

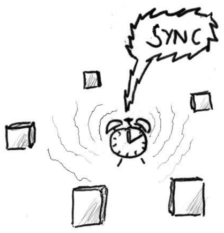
There is a distributed embedded control system where STATIC SCHEDULING and REDUNDANT FUNCTIONALITY has been applied. System has a functionality implemented with a node that uses static scheduling. This functionality is critical or has strict availability requirements and is therefore duplicated using solution offered by REDUNDANT FUNCTIONALITY. The redundant unit is used in hot standby mode, i.e. it is running the same process as the active unit both the output is not used for controlling the system. Individual clocks of units may drift which could result in clock skew between units.

Problem

How to keep the schedulers of distributed nodes synchronized as closely as possible?

Forces

- *Interoperability*: Separate units running their own instances of the same statically scheduled process must function synchronously.
- *Real-time behavior*: The process or processes running on duplicated hardware must be executed according to the strict timing requirements.
- *Distribution*: Units running the same process may be physically distributed to prevent downtime in case of failure caused by the environment, e.g. fire. Distribution to different locations may prevent reliable synchronization between units.
- *Accuracy*: Local clock of a unit might not be accurate enough to keep the processing synchronized.



Solution

There may be units in the system which are used to control a functionality or there is a functionality that has active unit controlling it and a hot standby unit, i.e. REDUNDANT FUNCTIONALITY has been applied. In these units, processes are scheduled statically during compile time. At run-time the units synchronize the process cycle using synchronization pulse. This synchronization pulse can be timed using local system clock. However, the system clock might not be accurate enough for systems having strict timing requirements such as power plant control systems. Furthermore, if multiple units must be synchronized, each unit can not use their local system clock to synchronize scheduling as those may have clock skew between them. In addition, if one units clock is used to synchronize all units, there is a risk that the unit malfunctions.

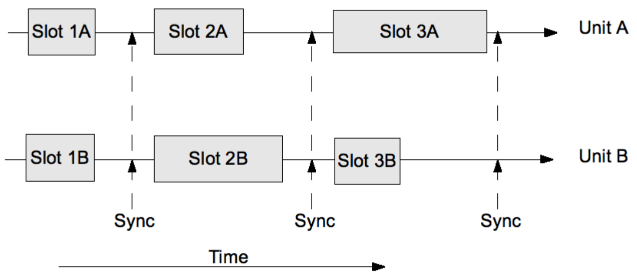


Fig. 17. Illustration of two units A and B using synchronization pulse after execution of each slot.

Therefore all units should use external clock, e.g GPS or atom clock, to synchronize their scheduling. Tasks are executed in statically scheduled order and if a task finishes early, the next task does not start before the synchronization pulse from the external clock source is received. Synchronization pulse from external clock should trigger the execution of next slot. For example, Fig. 17 illustrates how synchronization pulse is used in scheduling. Unit A and B both execute slot 1 and if units finish early, they still wait for synchronization pulse before executing slot 2. Amount of executable code in each slot may vary, but should always remain under reserved time slot before synchronization pulse. However, if task overruns its slot, a fault is generated. In this case, the system may switch to backup unit or boot itself depending on the purpose of the system. If task overrun happens on a backup unit, it is booted and meanwhile active unit continues to function. After the backup unit has booted up, there should be some kind of synchronization between units. In some cases it might also be sensible to use this pattern to synchronize the scheduler of one system.

Consequences

+ Multiple units running the same statically scheduled process can be synchronized.
+ External clock is not affected by the operation of the system and therefore it provides precise timing.
– The solution increases the system complexity as synchronization pulse originates from external system.
– Delays from the external clock source to all units using the external clock should be same. Sometimes this might be challenging to implement.
– Synchronization with external clock may create a single point of failure.

Resulting Context

The pattern results in a system where different units are scheduled synchronously using clock pulse from external source.

Related Patterns

If REDUNDANT FUNCTIONALITY pattern is applied, the synchronization is carried out between redundant units. Otherwise, this pattern can be applied to systems where multiple units are controlling single functionality.

Known Usage

Power network controller has a redundancy unit. The units are physically located in different locations in the same site. Controller uses static scheduling and synchronization pulses to pace the execution. In order to provide very high availability, the backup unit must be able to take control in a few microseconds. The system can be run a year without any maintenance breaks or boots. System clocks can not be used to synchronize backup unit and active unit as they are not accurate enough. Instead GPS clock is used to synchronize scheduling of the both units. GPS clock generates an interrupt on regular intervals. This interrupt triggers the execution of the next code block.

6 Patterns for System State

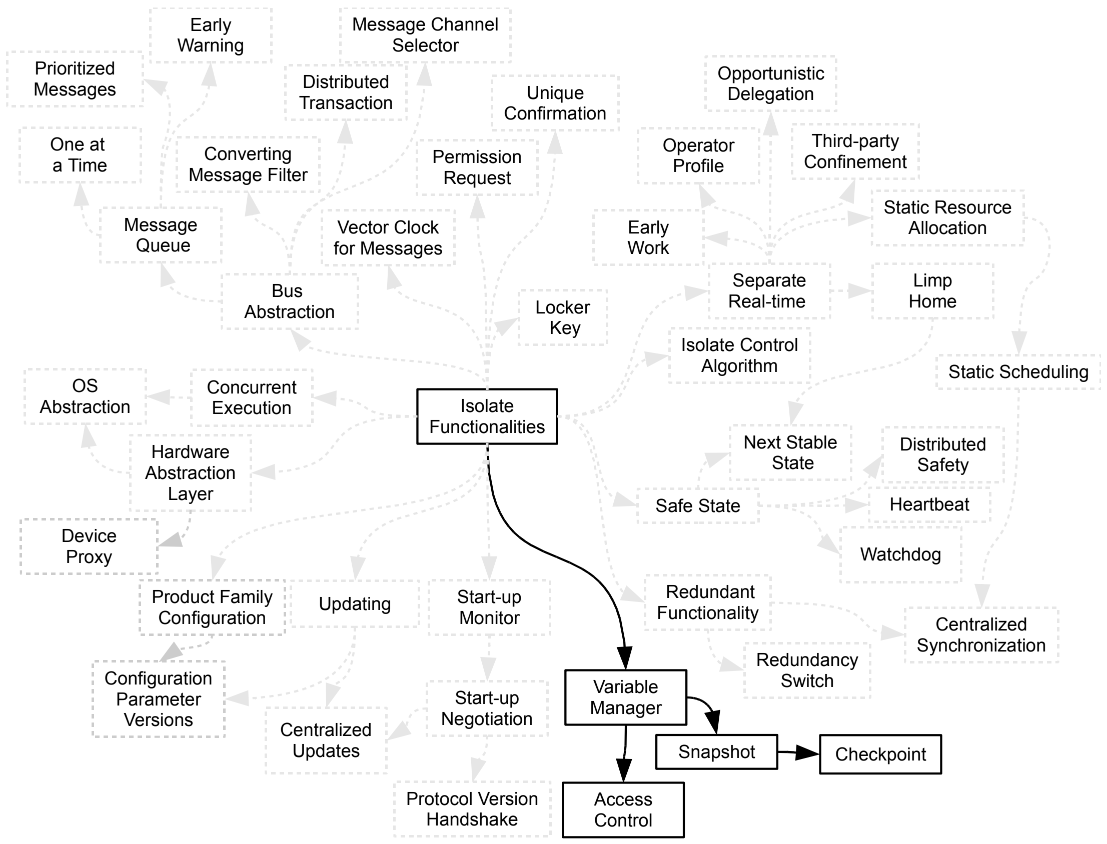


Fig. 18. The sublanguage for system state in distributed machine control system.

Table 5: Pattern thumbnails

Pattern Name	Description	Page
VARIABLE MANAGER	The complex system state information is efficiently shared and stored by using a separate component.	75
SNAPSHOT	The complex system-wide state information can be stored and restored for further usage.	77
CHECKPOINT	Snapshots are taken periodically so that information is not lost due to a fault.	79
ACCESS CONTROL	Unauthorized modifications to the common state information are prevented using access control component.	81

6.1 Variable Manager

Context

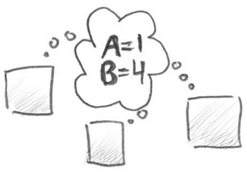
ISOLATE FUNCTIONALITIES has been applied, resulting in a distributed embedded control system with consists of several autonomous nodes. These nodes must share system state information to take proper care of their responsibilities. The node does not need to know the source and location of the information, and all data should be accessible independently of the provider.

Problem

How can you efficiently share systemwide information in the distributed embedded system?

Forces

– *Accuracy*: Data is volatile, so the local caches need to be flushed frequently.
– *Efficiency*: Data updating causes message traffic that should be minimized.
– *Scalability*: System must be scalable in terms of nodes.
– *Extendability*: It should be possible to add new units and they should have easy access to the system state information without changes to the rest of the nodes.
– *Adaptability*: It should be easy to change the location of the state information source.
– *Usability*: It should be easy to find the desired information about the system state.



Solution

A common state information module is added to every node. It contains the information that is relevant to operation of the corresponding node. This information is presented as separate state variables. For example, the current hydraulic pressure in kilopascals may be stored as one integer variable in the component. The values are received from and sent to other nodes using the bus. The local value of the remote variable can be updated when a message containing the changed value is noticed on the bus. All variable changes are sent as broadcast messages so the actual location of the information is not needed. The values of the variables can be sent and updated using different strategies: by-request, periodically, or as a side-effect, for example, when another variable is updated.

A variable can have an associated status or age. This information can be used to check whether the stored value of the variable is valid. If the value is not valid, the node should send a request for newer data to the bus. In this way, the node does not need to know about the origins of the data. The system may present even its internal state information as state variables. Its responsibility is then to send the notifications about the changes in the relevant variables to the bus. As there is a lot of information, which is partially not crucial for the whole system functionality, a care should be taken on what variables are updated via the bus. Otherwise, the bus capacity might be exceeded.

Consequences

+ Assuming that the updating frequency of the shared variables is high enough, each node gets sufficiently accurate state information concerning the other parts of the system.
+ Nodes can change information location transparently.
– This solution does not prevent the nodes from modifying the common system state so that the information becomes inconsistent.
– The solution may result in a large number of variable names that is difficult to manage.
– Variable broadcasting may cause a lot of bus traffic.

Resulting Context

The result is a distributed embedded control system where common state information is shared between the nodes.

Related patterns

The pattern can be thought of as a special kind of PROXY [18] pattern. BLACKBOARD [15] uses similar kind of data sharing. If the state information sharing mechanism is distributed, it can been seen as a BUS ABSTRACTION. On the other hand, the implementation may use BUS ABSTRACTION internally.

Known Usage

A heavy machinery system such as forestry machine uses multiple controllers to steer the machine. The controllers must share information in order to co-operate. This is achieved by every node keeping track of the needed parts of the system state information. The nodes update their system state information by receiving messages from the bus. Some information-carrying messages such as messages containing sensor data are sent periodically, some are sent whenever information is available.

6.2 Snapshot

Context

The system is distributed embedded control system that uses some mechanism, for example, provided by VARIABLE MANAGER, to share the system-wide state information. This information is volatile, but in some cases it would be usable. For example, a replacement unit for a faulty unit has no information of the system state.

Problem

How to store the state information for restoring and further usage, such as testing, debugging and performance analysis, in systems that have a complex system-wide state information?

Forces

- *Testability*: The system has many states, so the testing of all possible combinations is difficult. Therefore, it should be possible to restore the system to any certain state for testing purposes.
- *Analyzability*: The state information of the system should be restorable in laboratory for further analysis.
- *Recoverability*: If a node malfunctions and gets replaced with new hardware, the system state should be recoverable from the backup. For example, machine learned data could be uploaded to the replaced node from the central storage.
- *Reusability*: Machine learned data, for example, neural network state information, should be able to be copied to a new system.

Solution

Normally, tracing a problem the system may be difficult, as the system history depends on all actions taken. This makes achieving the exactly same starting state for testing difficult. Also, it may take too long to perform all previous actions to enter the wanted starting state. If the system has a mechanism that holds all vital state information for the system functionality (see VARIABLE MANAGER, it can be used to represent accurately the current state of the whole system. This information may be used to avoid problems with the testing, as it is possible to implement a mechanism to save the current state information as a snapshot and later to upload it back. In practice, the saved state may be a XML file that holds all variables and their corresponding values.

The stored information can be used in multitude of ways in addition to the testing. It may be taken under inspection in laboratory environment to collect information about performance, programming errors, etc. The saved state may be used to restore a crashed unit or to inform a replacement unit about the current system state. If the system state is uploaded to a replacement unit, it may continue from the point where the system was during the save.

In some cases, the physical world needs some adjustments to correspond to the assumed state of the system. For example, some switch sensor must be closed to correspond to the parking brake state. The system state information may consist of large quantities of variables, so it may be cumbersome to save and upload the whole state information in one piece. If this is the case, the system state information should be divided into more manageable units. For example, one could save state information separately for local and system-wide functionality. Local information should not be as crucial as the system-wide information, so it can be discarded if storage space is full.

Consequences

+ The testing becomes deterministic as the start state for corresponding test cases may be always restored.
+ The system state information may be used to detect programming errors after a system crash.
+ The system state information may be used to analyze the system in the laboratory.
– There may be some information that is not contained in the Variable Manager, thus compromising the testing.

– The observable real world state must correspond to the saved system state.
– The system state information may be too large to be saved conveniently.
– The system state saving and restoring typically momentarily halts the normal operation of the machine and it can not be done when the system is running.

Resulting Context
This pattern results in a system with ability to save and restore its current state, for example for testing, debugging, analysis etc. purposes.

Related Patterns
VARIABLE MANAGER can be used to store the system-wide state information. The designer should consider using CHECKPOINT pattern after implementing this pattern in order to enable periodic saves.

Known Usage
The elevator group control system has a neural network that learns patterns from the people flow of the building. This learned structure is saved as a state information. The state information may be saved as a snapshot for testing purposes and analysis in the laboratory. The snapshot also works as a backup for this valuable information.

6.3 Checkpoint

Context

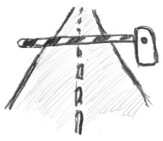
A distributed embedded control system uses some mechanism, for example created by VARIABLE MANAGER, to share the system-wide state information. The system has an ability to store the full state information, as described in SNAPSHOT pattern. Because the snapshots are taken on-demand, they may be obsolete or completely non-existent.

Problem

How to keep a continuous record of complex system state, so that information is not lost due to a fault?

Forces

- *Recoverability*: The system should be able to restore a certain state if something goes wrong.
- *Testability*: The saved system state should always be current enough, so that it can be used in system debugging.
- *Recoverability*: The system should be able to restore a very recent saved state, in order to make fault recovery fast.



Solution

The system should save its state as snapshots periodically in order to keep up the state information. The periodic save creates a checkpoint where the system can return. The reason for returning may be for example a system fault. The system may be up and running when the checkpoint save is done or it may be taken when the system is starting up. The checkpoint save may be triggered by different kinds of events. For example, the save may be done periodically, when ever a new device is added or when the system starts up. In start-up phase, START-UP MONITOR pattern may help in detecting if a new checkpoint save should be triggered.

The state could be dumped every time as whole, but that method consumes a lot of system resources as the system state may be very complex. In order to lighten the load of saving, it is possible to save the modifications in between periodic saves as incremental information. All state changes are saved as delta information, so using a known base situation and all delta information the system should be able to reconstruct the latest state. This enables the restore process to return to same point where the system was before the fault and no data is lost during the recovery. In this case, the restore process is more taxing to the system. The base state information has to be restored first and after that all delta information must be taken into account.

Because checkpoint data provides insight to the system state, it may help in debugging. If the system crashes, the checkpoint data may be inspected in order to find out what the system state was when the error occurred. If the periodic saves are made frequently enough, the progression of the problematic situation can be backtracked.

If REDUNDANT FUNCTIONALITY has been applied, the checkpoint data may be migrated to a backup unit in order to quickly restore functionality, if the active node fails. A saved state is uploaded to a replacement unit so that unit is immediately able to continue from the latest checkpoint.

Consequences

+ The system always has a restore point if anything goes wrong.
+ Checkpoint mechanism creates an automatic backup mechanism.
+ The checkpoint information may be used to detect programming errors after a system crash.
– The system state information may be too large to be saved conveniently and transfer times may be long. This may impose severe restrictions to the saving frequency.

Resulting Context

The pattern results in a system where the system state is saved periodically to avoid total loss of gathered information.

Related Patterns
CHECKPOINT [13] is a description of similar periodic save mechanism.

Known Usage
Most database systems make periodic saves of the whole base. The modifications after the checkpoint are saved as a incremental delta information. This enables the system to be always in a consistent state regardless of any faults. No information is lost if the database crashes.

6.4 Access Control

Context

There is a distributed embedded control system that uses some mechanism, for example, provided by VARIABLE MANAGER, to share the system-wide state information. The state information is modifiable for every application in the system.

Problem

How to prevent either malicious or unintentional modifications to the common system state information?

Forces

– *Data integrity*: The system state should be consistent with the real world.
– *Data integrity*: The system state should be protected from malicious changes.
– *Analyzability*: The state changes should be traceable to help debugging.
– *Understandability*: All applications do not need a full access to the common system state.

Solution

The state variables of a system are usually stored in a centralized mechanism (see VARIABLE MANAGER). In the basic case, all vital state information for the system functionality is stored so that all interested parts of the system may read and write the state whenever necessary. This compromises the reliability of the state information as a node may due to programming error or hardware malfunction update wrong state variable(s). Furthermore, there may be applications in the system that are provided by a third party. These applications can not be always trusted, as they may do some malicious modifications to the system state.

Some kind of mechanism to guard the state variables should be designed. This mechanism is the only component in the system that can access the state variables directly. It must demand authorization for other system parts to submit their own state changes. The access control component knows which applications or nodes of the system are allowed to alter the state information. It is easy to make the mechanism even more fine grained, as some parts may be allowed to make only modifications to a subset of the common system state. Furthermore, providing a read-only access to more critical parts is also an option.

The access control system should keep some kind of access log, so in all situations some kind of traceability of the changes may be instantiated. The access log must naturally be protected, so that users may not tamper with it.

Consequences

+ All system state protected against unintentional or malicious updates.
+ If something goes wrong, the part that caused the malfunction may be identified.
– The access control makes the common state access slower.

Resulting Context

The pattern results in a system with ability to protect the common system state from unauthorized modifications.

Related Patterns

VARIABLE MANAGER can be used to store the system-wide state information.

Known Usage

The database systems hold connection manager that gives different user accounts different rights to modify the database tables. For example, the administrator account may modify all tables freely, but the normal user may only modify the user account information.

7 Patterns for System Configuration and Updating

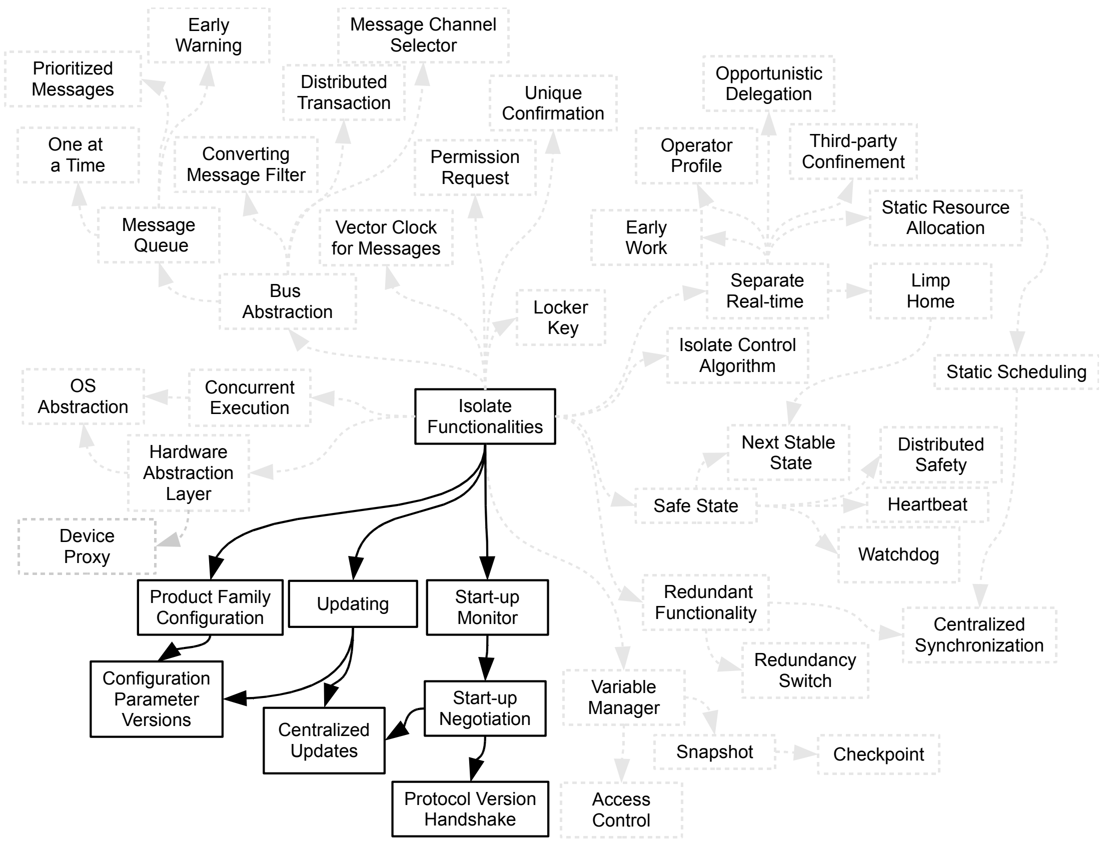


Fig. 19. The sublanguage for system configuration and updating in distributed machine control system.

Table 6: Pattern thumbnails

Pattern Name	Description	Page
START-UP MONITOR	During start-up all devices are started in certain order and with correct delays. Additionally, care is taken that there is no malfunctions.	83
START-UP NEGOTIATION	The system device setup may change. Therefore, available features are detected during start-up and core functionality is confirmed to be functional.	85
PROTOCOL VERSION HANDSHAKE	A common protocol is used to detect that components having different software versions can function together.	87
UPDATING	Software and configuration of a device should be replaceable.	89
CENTRALIZED UPDATES	A single unit takes care that software versions on all nodes are compatible with each other.	91
PRODUCT FAMILY CONFIGURATION	A common pool of software components is used with a configuration file to vary product families and products.	93
CONFIGURATION PARAMETER VERSIONS	Different software components use the same base parameters. Newer software version may have some more advanced parameters which are separated from the older software version parameters.	95

7.1 Start-up Monitor

Context

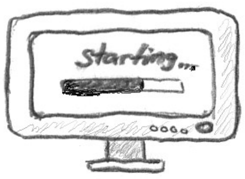
A control system has been distributed using ISOLATE FUNCTIONALITIES pattern. The system has a fixed set of devices but all the devices in the system are not necessary in use. Devices may have damage, and damaged devices might prevent the system from working or they might cause additional damage when used. The devices in the system have to be started in certain order and with certain delays to prevent malfunctions.

Problem

How to ensure that the devices are functional and they start in right order with correct delays without breaking anything?

Forces

- *Resource utilization*: Starting may be heavy process and scarce resources should not be overtaxed during start-up.
- *Analyzability*: The starting order of devices should be known and always the same.
- *Testability*: It should be easy to test correct start order.
- *Suitability*: Some devices might need undisturbed warmup period.
- *Predictability*: Resource utilization during start-up can be predetermined from historical values.



Solution

Every machine has to be started before use, but the start-up procedure is not always so simple as switching power on. A device might need some resource such as hydraulic pressure or high voltage during start-up. Devices needing certain resource might have to take turns as the resource might be scarce. Sometimes a device start-up might take considerable amount of some resource so that nothing else is left to others, even for normal operation, until the device has finished the resource hungry start-up. For some devices the start-up procedure might take longer time as, for example, the device is heated before use. There might also be functional dependencies so that, for example, device A needs services from device B.

Clearly there is a need to put devices in starting order. The problem is solved by introducing a new component, start-up monitor, to the system to control and guide the system start-up process. The monitor component can also do additional tasks if necessary. For starters, the monitor component wakes up first in the system. It makes some basic system tests on the core functionality such as does to the bus work. After this, it will start waking up the required services to make the core of the system functional. During this process it will test that the devices are intact and working. The monitor will enable devices in such an order that there should not be any problems with dependencies and resource deficiencies. Additionally it will provide sufficient time for slow devices to make themselves available.

Start-up monitor has either to know the dependencies, delays and limitations of the devices and resources; or it has to follow some fixed preset order. In the first case the start-up order will be deduced from a graph with some suitable algorithm. Which ever is the case, the monitor takes care of the start-up order and delays between different devices. If any problems occur during the process, the device is then put to best possible state where external diagnostics can be run or the servicing of the machine is easier. Usually for many machines this means that they are shut down.

Consequences

+ Dependencies between devices can be handled in a single place.
+ Different startup sequences can be tested easily.
+ Devices can be started in a controlled way.
+ Devices do not need to know about each other to start up.
– This solution may create a single point of failure unless duplicated.
– Start-up monitoring slows down the start-up as safety-limits will be followed.

Resulting Context

The result is a distributed control system where start-up order is handled by a separate entity.

Related Patterns

START-UP NEGOTIATION describes how this pattern can be enhanced, so that there can be changing set of devices available during start-up. This could be useful in case where the same core system is used many product variations. SYSTEM MONITOR [13] discusses on a component whose task is to monitor what is happening in the system.

Known Usage

When a new satellite starts up, its systems are cold. It is the responsibility of boot-up routine to wake up the system so that nothing breaks. At first, the essential subsystems are loaded with new patches to fix previous software problems. After that, the essential systems are put online. One of these is heating. Energy has to be conserved until energy resources are observed to be safe for use. For example, a new satellite has to open the solar panels before they can be used. After the system is in minimal and stable working state, different devices are woken up in order. In some cases devices may not be used immediately, for example, a camera in -270 Celsius temperature would break when used cold. Therefore, it has to be warmed up before any other device or program is allowed to use it. Such an accident is prevented by keeping the camera disabled. If some unexpected happens, the system is moved to a safest possible state where it basically just is keeping itself alive so that the control station personnel on earth can decide what to do.

7.2 Start-up Negotiation

Context

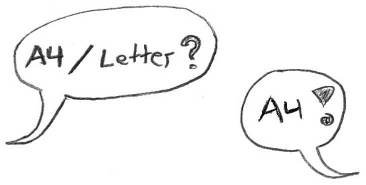
The start-up order of a distributed control system is handled by a dedicated component created with START-UP MONITOR. There might be different set of devices available on each start-up. Depending on what devices are available and what is the system configuration, different features maybe be available. Certain set of devices are required as a core of the system functionality or the system can not be used.

Problem

How to verify that required devices are available during start-up and find out what features can be enabled?

Forces

- *Extendability*: It should be possible to add optional new devices to add new functionality to the system.
- *Configurability*: Devices can be used with different options and settings depending on the machine configuration.
- *Interoperability*: Different device combinations need to be able to work together.
- *Safety*: It should be verified that the minimal configuration for operations is present.



Solution

At system start-up, all nodes send information of their existence to the bus. A central node, created with START-UP MONITOR and amended with this pattern, collects this information and by a deadline builds a list of nodes and devices available. A verification is done to guarantee that at least minimal configuration is available so that the machine can be operated. If minimal configuration is not available the machine operator will be informed on the situation and all available diagnostics will be gathered. Additionally, if there is some device which is incompatible with the system it can be disabled.

The list of available devices is compared to an previous list from last start-up to see which devices are now inactive and which devices are new. Based on the list of available devices, the start-up order may be revised. Additionally, depending on the system settings and inserted or removed devices, some devices might need to be updated with new configurations before the system can be used. For example, new attached drill might not be able to handle high pressure the same way as the old one did, so the compressor should not create as much pressure as before.

When a dedicated component takes care of the feature negotiation, it is possible to have many system setup variations based on the same machine core by just adding and removing devices. The different system setups can be then sold for different uses with varying options. This also makes it possible to add or change features to the system after sales, even on field, just by adding a new device. In some situations this can be beneficial as the same machine can be used on different tasks. For example, there is different kinds of forest harvester heads for different kinds of trees and the head can be changed on site.

Consequences

+ This pattern can be used to make sure that required devices are available.
+ This pattern can be used to make sure that the machine wide configuration is sound.
+ This pattern can be used to test devices with different parameters.
+ New devices can be attached and the control system can detect and use them after reboot.
+ Devices can be removed and the control system can handle the changes.
+ The start-up negotiation facilitates the variation of the product for different uses.
− The pattern may create single point of failure unless duplicated.
− It maybe difficult to test and verify all device and software version combination.
− The negotiation slows down the start-up.

Resulting Context

The result is a distributed control system where required device presence can be verified, the start-up order is flexible and available feature are negotiated at start-up.

Related Patterns

CENTRALIZED UPDATES describes a system which takes care that different devices in the system are using compatible software versions by updating everything from same software bundle.

Known Usage

The operator of a forest harvester starts the machine. During start-up an hydraulics controller, an essential component for the system functionality, is not announcing itself to the CAN bus. When it does not announce itself before the deadline of 200ms, it is assumed either to be removed or broken. The machine operator is alerted on the failure and the machine will be shut down, as it can not be used.

7.3 Protocol Version Handshake

Context

There is a distributed control system start-up and feature negotiation is handled by a dedicated component created with START-UP NEGOTIATION. If a device in the system is replaced, the replacement part might differ from the original. Therefore, also the software on the replacement part could be different than on the old part. Consequently, the device might use different protocol version for communication than the other nodes.

Problem

How to communicate with a device that uses different protocol version than the rest of the system?

Forces

- *Changeability*: The protocol must adapt to changes in the implementation of device hardware and software with the new features those might bring along.
- *Adaptability*: The whole system could have much longer life cycle than what is availability for some certain types of device hardware.
- *Interoperability*: Different software and hardware versions should work together.
- *Predictability*: A device attached to the system should do actions which conform to the task which is given to it.

Solution

Without same language there is no communication. Therefore, to communicate all nodes on the bus need to speak similar language to each other. When a node starts up first time, it will announce itself to the bus for the start-up negotiator component. After a while the negotiator knows all active nodes in the system and based on this information, it will initiate a short discussion with each of the nodes to find out their capabilities. This discussion is called handshaking. During the handshake, from the oldest nodes with the oldest software to the newest nodes with the newest software all have to work together. This means that even by the oldest possible negotiator component should be able to deduce the knowledge level of each node in the common language.

The handshake procedure is something that can not be changed after the product is launched. Otherwise some of the older negotiators components might not know how to handshake with newer nodes and vice versa. At minimum during a handshake, a node announces what is the highest protocol version it knows. The handshake can also contain more information than just the protocol version. For example, a device could announce what kind of device hardware group it presents, what is its hardware version, what kind of services its software offers and what version its software is. This information can be used by the negotiator component, for example, to decide in which order system components should be activated.

When the lowest common nominator has been found for the common language across the system, further negotiations will be done at this level. This level, of course, could be too low for any meaningful work from viewpoint of a newer node. In this case, the protocol version will reveal incompatibility between nodes. In such a case, the node can not be used in the system. Such a problem might arise from the fact that the newer software has to know always all the older languages versions and this might build up as considerable burden over the years. Alternatively, even if the hardware functionality is the same, its actual implementation might have totally changed. If older implementation has been visible to the other nodes (abstraction leakage), this might have created really strong ties between implementation and nodes, and this will prevent from using any new designs.

If a system life cycle can be counted in tens of years, it is probable that some of the components will be replaced during this time. Furthermore, as the life cycle of the electronic component is comparatively short, the new component will be quite certainly different than the original one. Still, the software can work similarly even on different hardware and therefore it should not create a big barrier for communication. Therefore, the biggest problems with compatibility can be found from the evolution of the software and from the new features it has. It is impossible to predict what kind of new features the new device has with new software and design such a common language that these features would be usable before they even exist. Therefore, old system can only use features which were available during its design time.

Consequences

+ All devices can be recognized.
+ Different devices from different eras should be able to work together.
+ The information gathered during handshake can be used later on during the start-up.
− A large set of different protocol versions will create a burden for maintenance.
− Possibility to use different protocol version with different hardware versions adds complexity.

Resulting Context

The result is a system, where a common language can be negotiated.

Related Patterns

Using UPDATING pattern, it might be possible update nodes so that they would have capabilities and higher level of common language. CONFIGURATION PARAMETER VERSIONS pattern can be applied to make sure that different generations of software are able to work together with the shared set of system parameters. Older hardware can be abstracted with HARDWARE ABSTRACTION LAYER.

Known Usage

A forest harvest is taken into yearly maintenance. Diagnostics inform that the boom controller should be replaced. A new part directly from the factory is used to replace the old part. When the system is started, the new boom controller has much more capabilities than the old controller had, but as the rest of the system use older version of the common language these capabilities will not be used. Therefore, from the system side the new part transparently replaced the old part.

7.4 Updating

Context

A control system has been distributed using Isolate Functionalities pattern. Some of the devices in the system might need modifications during the machine life cycle.

Problem

How to ensure that software of the device is replaceable?

Forces

- *Extendability*: Add-on devices might need additional software.
- *Updateability*: The device should be able to run newer versions of the software.
- *Configurability*: Device configuration may change and therefore it should be updatable.
- *Access Security*: The updating functionality should be shielded from unintentional or malicious use.
- *Fault Tolerance*: Device should stay usable even if updating process fails or is aborted for some reason.
- *Safety*: Updating process should not cause any harmful side-effects.

Solution

If the hardware and software is implemented as unchangeable combination, the whole device must be replaced if there is a need for changes or optimizations. Therefore, it is often desirable that the software is updateable. This, of course, requires rewritable persistent memory on the device. The updating can be done either offline by using a separate tool which will overwrite the persistent memory or online by running update process. It should not be possible to run the update by mistake, the device should be shielded with access rights or different usage mode should be required for the update functionality. When the update is commencing, the code under update should not be used as it would probably lead to unknown errors. The device update may fail or the device might break during the update. Therefore, after writing process, the code should be compared to verify that the writing was successful.

If the update is run online from the random-access memory (RAM), it is often desirable to have a recovery capable bootloader or a separate recovery section in the persistent memory which is not overwritten during the update. If the update fails, for example, due to power failure, the device should still be able to boot so that the update could be continued. It is convenient to have some small area on the persistent memory for update status storage. The bootloader can check this update status area during boot to recognize, for example, if the program section update failed and the update should be retried.

Sometimes it is necessary to update the bootloader but if this update fails the whole device is unusable. To solve this problem the bootloader can be divided into two stages. The first-stage bootloader's job is to find a working second-stage bootloader and run it. The second-stage bootloader's responsibility is to set up the device so that the actual software can be run. The second-stage bootloader can be found for example by using a fixed address, a memory address containing the real starting address or a unique token that is searched to find the address where the code starts.

The second-stage bootloader can be duplicated so that there is always one functional second-stage bootloader even if an update fails. In Fig. 20 example layout is illustrated with first-stage bootloader and duplicated second-stage bootloaders. The recovery second-stage bootloader would be used, for example, after hard reset or after reading failure bit from update status area. Still, the same updating problem applies to the new first-stage bootloader but usually this is not a problem as it rarely needs updating as it is quite small piece of code needed only in the beginning of the boot process.

When part of the bootloader are duplicated, only first duplicated section is updated and the other part is kept as is. After the update the section is compared to a checksum to verify that the update was successful. If the update was successful, the unique token or address pointer is changed. After this the second duplicated section can be updated. The layout of the persistent memory can change when sections change in size. If the sections get smaller during update, the updated can commence in same order as the the system would start up. If the sections get larger, a reverse order should be used so that there will be always at least one working bootloader which is not party overwritten.

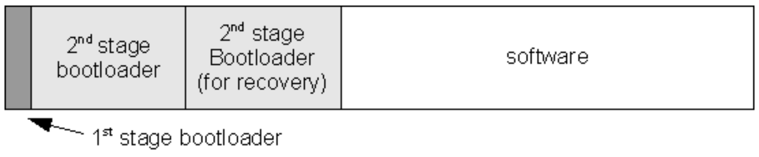


Fig. 20. Persistent memory layout for first-stage, second-stage and recovery second-stage bootloader.

Consequences
+ There might be no need to replace the device, if the software can be updated.
+ The update can be done more quickly than a device could be replaced.
+ The solution lowers the maintenance costs as the software can be updated.
– Updating adds production costs as more expensive or additional components may be needed.
– Updating adds complexity to system design and updating process.
– Update may fail or break the device.

Resulting Context
The result is an embedded control system device with a functionality that can be updated.

Related Patterns
SOFTWARE UPDATE [13] discusses also on the same topic. REDUNDANT FUNCTIONALITY discusses how duplication could be arranged. CENTRALIZED UPDATES provides a central point where multiple nodes are updated in coordinated way. CONFIGURATION PARAMETER VERSIONS presents a way to make sure that configuration parameters are compatible between different software versions.

Known Usage
In harvester's boom a grappler module is updated to new software version. The machine is put to stop and the device is deactivated. The service person connects an updating tool to the device maintenance interface. The updating tool sends the new program to the module's flash block by block. After writing the code is read to verify that its cyclic redundancy checksum (CRC) is correct. If the update was successful the device is activated.

7.5 Centralized Updates

Context

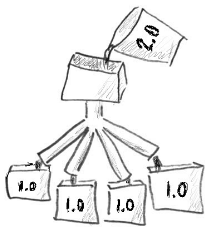
Updating patterns have been applied to distributed control system to create updateable distributed functionalities on separate nodes connected by a bus. Additionally Start-up Negotiation has been applied so that changes in the system could be detected automatically. The nodes communicate with each other to function as a whole. Therefore, if the versions and age of the software on different nodes differ much, there might be compatibility issues and the system might not work.

Problem

How to be sure that every node has been updated to compatible software version across the system?

Forces

- *Interoperability*: Nodes should have compatible software to function together.
- *Analyzability*: It should be easy to deduce that the whole system is using the same software release package.
- *Updateability*: There should be a simple way to deploy system wide software updates.
- *Adaptability*: New hardware should be detected and updated automatically to make it compatible with the rest of the system.



Solution

Updating every device in the system by hand in cumbersome. In manually updated system there might be some nodes having really old software and other nodes that have quite a new software. This may cause malfunctions as there might be compatibility bugs and added functionality in new new software that the older software does not support. To combine compatible and tested software together, all the software for the system and devices is combined into single release package. Basically a release package is tested snapshot of all the system software from certain date. Still, it should be noted that as the software release package has software for many devices, it is possible that not all device combinations are tested and therefore there might device setups which do not work together.

One way to guarantee that the system has the same software release package is to have one node in control which will check this fact during start-up before the system can be operated. This node will function as central clearinghouse for version information gathering and the node update delivery. If a node is detected during start-up to have software version which is not from the same software release package as the rest of the system, it shall be updated with software from the release package to make it compatible. This kind of software version disparity could be either caused by a device which has been replaced or by a new device which is added to the system. This requires the update server to contain software for every known device which can appear in to the system.

To make the updating functionality to work all the devices answer in uniform way to software version queries and update the software through similar interface. This requires that the interface is fixed and will not change between software package releases. If new hardware is added, it is noticed during the machine start-up by the machine start-up negotiator. Before the machine can be used, the operator is queried if the added node should be set-up or disabled. It might be possible that there is no support for the new device in the currently used software release package. In this case, the whole system has to be updated to a package version where support is available before the new device can be taken into use.

Sometimes there is a new software release package for the whole system available. When a service personnel loads the new software release package from a media to the update server, the package is verified as valid before it is stored. When this package is stored to the update server the whole system can be updated with the new package from single point simplifying the task of updating. The new packages should not overwrite the old ones, so that any update package can be used if necessary. When there is multiple software release packages available on the update server, the operator can choose which package version is used for the machine. This means that the whole system can be updated to any available software release package that has software for all the devices in the system. This is for the convenience of the machine operator as it might be inconvenient time to do an whole system update or the operator might prefer the way the system operates with the older software release package.

During a whole system update the nodes in the system are updated one by one. If the update for a node fails it will be retried. If, after couple retries, the update can not be uploaded, it has to be determined if the node is functional at all. If the node is functional there is three recourses to the situation: get a new node which can be updated, disable the node or use software release package which is compatible with this unupdateable node. If the last option is selected, then the rest of the system has to have compatible with this node which could not be updated. Basically this means, that the system has rolled back to the version which it had before the failed update. The service personnel should be notified on failure.

Consequences

+ System is always in compatible state if the whole update package was deployed.
+ Central update mechanism is easy to understand and use.
+ Older software release packages can be used, if newer update results in dysfunctional machine.
+ All devices can be handled with same update interface.
+ It is easy to deploy an update to the system.
+ If a device is added to the system and it has support in the package, it will be automatically usable.
– Single failing device may prevent the whole system update.
– Single repository can be a single point of failure, but fortunately it used only for updating.
– It is difficult to test all device combinations for all existing environments and therefore there can be combinations which do not work together.
– Old software release might not support some new device and for the support the whole system software has to be updated.

Resulting Context

The pattern results in a system where single unit takes care that device software versions are compatible with each other.

Related Patterns

DISTRIBUTED TRANSACTION discusses how commence with actions which involve multiple nodes so that all the nodes commit to the action or the action is not done.

Known Usage

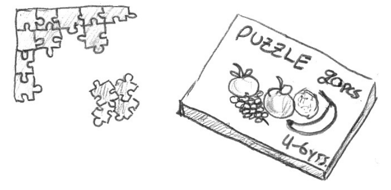
A drilling machine is stopped and set to stable state. A service person insert an USB stick to the cabin computers USB port. The system notices the inserted media, find the update package and verifies it as a valid package. After this the package is extracted to the update server. The system notifies the service person of the new available system update and queries if it should be installed. When the service person confirms the installation the system starts to update. The updated software is sent from the update server to each device in the system one by one over bandwidth limited CANopen bus. After the last device is updated and success is verified, the machine is started with new software.

7.6 Product Family Configuration

Context
There is a distributed control system platform which is the core for different product families. There is many product variations inside a family for different uses and different price categories. The software for the core system is shared for all machines but all other software components depend on product family and product variation.

Problem
How to manage families and variations of the same base product from larger software component pool?

Forces
- *Changeability*: Depending on the target, suitable software and parameters should be chosen.
- *Variability*: It should be possible to efficiently make variations of the product from the same component pool.
- *Reusability*: Subsets of a larger software component pool should be usable in different products.
- *Maintainability*: There should be only one codebase, as its management is simpler.
- *Configurability*: Same software components should function same way in different machines.



Solution
When a system has been designed, the base system of a machine has been designed to be common for all the platforms and variations. This base system is made from certain components which are shared between all the products and a common software controls this base system. To create a machine, application area specific components are added on top of this base system. For example, a harvester and forest tractor have similar transmission mechanisms but harvester has specialized boom for cutting trees. A specialized software controls these hardware components. Because different customers want different setups of the same base machine, there has to be a possibility to create variations of the product family design. For example, forest harvesters can have many different kinds of harvester heads to cut different kind of trees.

To achieve this differentiation to product family and product variations, the system setup is described in a configuration file. This file is used to select the required software components and their parameters for the desired system setup. During compilation or integration phase suitable components are included into the setup so that their parameters are compatible and their functionalities support other included components. For example, gearbox controller has to behave in different way depending on how large motor the machine has and what kind of gearbox is used.

It may be difficult to find these variation points unless the domain is known really well. Even then different needs may appear long after the platform has been designed and these need may require new variation points. A new point may be almost the same as an old point but there is such a difference that same point can not be used. Sometimes totally new variation points are needed and software redesign is required as the old design does not support variation in that place. It is hard to test that the configuration will result in valid set of components. Additionally, the parametrization combinations of different software components may not work together either. To confirm compatibility, testing has to be done.

Consequences
+ For known product families well-defined variation points can be established.
+ Same component set can be used in multiple setups.
+ Different combinations of software components can be chosen compile time to create software for different products.
– It may be difficult to find variation points.
– It is difficult to test all possible software component combinations.

Resulting Context
It is possible to variate product families and product using a common pool of software components.

Related Patterns
COMPONENT CONFIGURATOR [6, p. 490] presents a configuration system which can be used at run-time to select the correct configuration. START-UP NEGOTIATION detects different system components during start-up and sets the system up. This can be used to add new features on the system after it has left the factory.

Known Usage
The product component setup for a forest harvester is XML based. For each product family member, a family specific XML file generator is first created with base configuration XML file. After compiling, a family specific XML file is created. This can be edited to create such a machine variation as the customer wants. With this file a correct component setup is then combined on top of the base system as a final product.

7.7 Configuration Parameter Versions

Context

Variations of the product are created on top of the same platform. The system configuration depends on the variation and the configuration contains only devices and software components which belong to the variation (PRODUCT FAMILY CONFIGURATION). The devices in the system are updatable (UPDATING) but even when compatible, some devices might present different phases of the product's evolution. The devices are configured using parameters across the products and software versions.

Problem

How to ensure that configuration parameters are compatible between different software versions and on different product variations?

Forces

- *Changeability*: It should be possible to have compatible parameters across different software versions and different product variations.
- *Interoperability*: Configuration parameters should work the same way on different products or with new devices.
- *Extendability*: In some cases, more detailed parametrization is needed to make a device work correctly.
- *Updateability*: Type and size of a parameter may change due to a software update.
- *Updateability*: Large changes in software can make some parameters irrelevant and require set of new parameters.

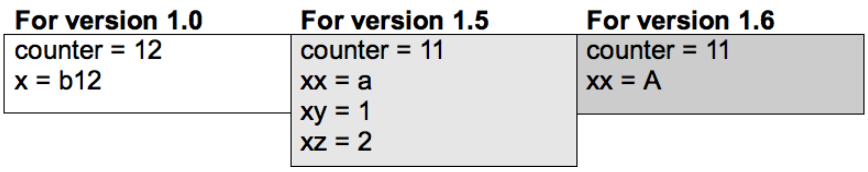
Solution

One can not be sure if a parameter is compatible with a new software version just by looking its size and type. For example, enumeration values could have changed, and this can not be noticed . Using the new value as old might cause obscure problems. For example, an engine might start to knock when given wrong tuning value. Therefore, it is necessary to have some way to deduce if the parameter is compatible or not. One way to verify compatibility is to separate different parameter versions to separate sets. The device reads parameter values from the sets which have the same or lower version number marker that what it is capable of reading.

Some parameters might be present in different parameter version sets. They are basically duplicated for devices which understand different parameter versions. In use, the information (values) in a newer block will always overrule the same information of the older one. Old parameter version set can not be obsoleted unless it is certain, that there is no devices which would use it. Therefore, if there is even one device which could be attached to the system and it only supports older set, the older set can not be removed. Consequently, all newer devices have support the lowest common version set. The old parameter versions can be only removed after update where newer software version will replace all older ones and therefore there would not be any devices which would use the older set.

When a parameter in one version is adjusted, all corresponding parameters in other versions have to be adjusted so that all the other "older" devices will know too that the value has changed. When a device updates a value and is not capable of handling the higher version sets, the updated set values might in conflict with these other sets. One solution to notice this problem is have a counter for each set, this is illustrated in Fig. 21. For every set, the counter value should be the same. If it is not, then parameter versions have to be updated to match the set with highest counter value. In the example, version 1.0 has highest counter value and therefore sets for 1.5 and 1.6 should be updated accordingly. A device which updates a counter value should always update all the counter values in those sets which it can work with. Before reading a parameter, the device should check that none of the other sets have larger counter number and if they do, the device should update all the sets it can so that they match with the modification in the older set.

In some cases, the parameter expands to multiple parameters (x to xx, xy, xz) or becomes obsolete over time and versions. In those cases more work is needed to support the older versions. If only part



For version 1.0	For version 1.5	For version 1.6
counter = 12	counter = 11	counter = 11
x = b12	xx = a	xx = A
	xy = 1	
	xz = 2	

Fig. 21. Example of three parameters set for three different software versions.

of the information can be saved to the old parameter, additional new parameters will be needed to save the information which does not fit in the old space. The old device will work correctly with the old parameter but the other newer device might require more information so they require additional parameters. Therefore, two sets of parameters need up-keeping. Similarly, if the parameter has become obsolete but there are old devices which use the parameter, it has to be updated.

Updating corresponding values in different sets is not always trivial. The new settings might be so different that complex or complicated logic is required to make the translation from one set to another - if it is possible at all. Additionally, when value in older set has changed, there might be additional parameters in the new set which can not be set based on the information in the older set (Fig. 21, "a" becomes "A" for xx). A simple solution to this is to use default values. This might not work in every case, for example, a few parameters which do not exist in older set have been optimized in the newer set. These parameters have one associated parameter which exists in both sets. This one parameter is changed in older set, what should be done with the associated parameter values in the new set?

Consequences

+ Parameter type can be changed.
+ New parameters can be added.
+ Older software can be used with other new software.
– Additional work is needed to create translator components for older parameter version.
– In case of parameter expansion or reduction the up-keeping is more complex.
– It is much more complicated to read and write many parameter version sets which are required for compatibility.
– The parameter namespace could become hard to follow and understand.
– In might be impossible to do some translations from version set to another.

Resulting Context

The pattern results in a distributed embedded control system where parameters are compatible with device software version combinations.

Related Patterns

START-UP NEGOTIATION may provide the information which version sets are used in the system.

Known Usage

One well know case of parallel information is the case of MP3 files. Originally an MP3 file could contain information on the song name, artist, genre etc. in ID3 field. When the format was released the information fields were fixed length. Later on, support for fields such as album art, lyrics, etc. was required, but the old fixed length fields could not support the request. Therefore a new information was added to the end of the file and it was called ID3v2. The old players can not read the new ID3v2 tags so it is up to the file encoder to write both ID3 and ID3v2 fields if support for old player is wanted.

8 Patterns for Operating System

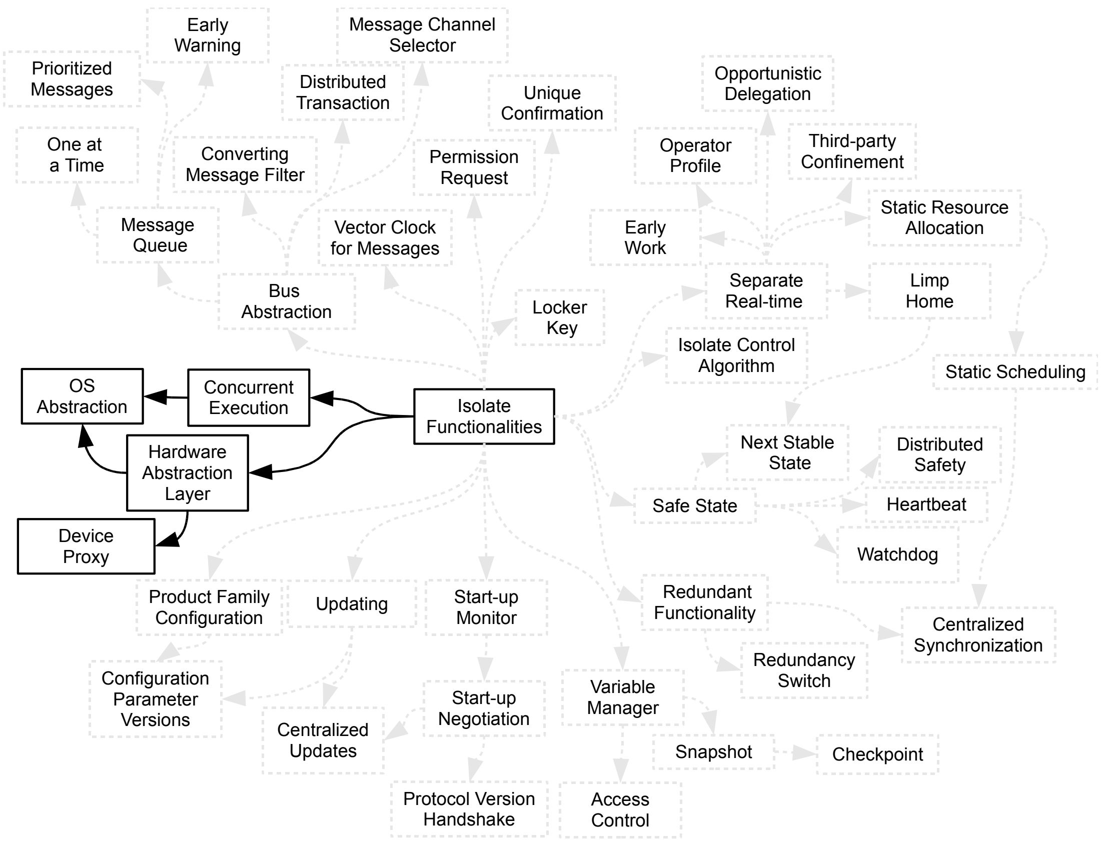


Fig. 22. The sublanguage for operating system in distributed machine control system.

Table 7: Pattern thumbnails

Pattern Name	Description	Page
HARDWARE ABSTRACTION LAYER	The system is designed so that application programmers do not need to care about details of the actual hardware.	98
DEVICE PROXY	System has subsystems that are manufactured by different vendors. Different kinds of subsystems should be controlled in uniform way.	100
CONCURRENT EXECUTION	To provide concurrency, multiple tasks are run on a single processor.	102
OPERATING SYSTEM ABSTRACTION	The underlaying operating system is abstracted so that it can be changed with only few modifications to the application.	104

8.1 Hardware Abstraction Layer

Context

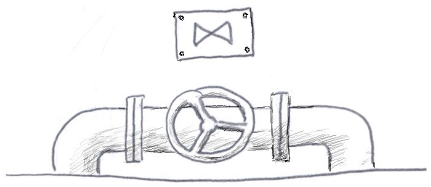
The system has embedded controllers, which are used to operate different kinds of devices. These devices perform varying actions, such as control hydraulic oil stream with a valve, sense the engine coolant temperature, measure the rotating speed of the cooling fan, etc. These devices may differ by vendor or by features, but in general, they are classifiable by device types.

Problem

How to enable the application programmers to discard the details of controlling of the hardware devices?

Forces

- *Scalability*: System should be scalable in terms of the number and type of hardware devices.
- *Extendability*: It should be easy to add new hardware devices to the system.
- *Changeability*: It should be as easy as possible to change the hardware components regardless of the implementation details.
- *Changeability*: It should be easy to write new software controlling the devices.
- *Variability*: The application software should be unaffected by the details of hardware.



Solution

Because the devices can be classified into functional types, such as temperature sensors, valve coils, pulse counters, etc. they should be controllable in an uniform way, regardless of their hardware implementation details. For instance, the application programmer is interested in temperature, in degrees of Celsius, of the engine coolant liquid or the ambient air. A programmer should not need to be interested if this information is conveyed by voltage between bimetallic strips or by thermistor resistance. These kinds of implementation details should be abstracted under a generic interface. These interfaces form a layer between the application software and the controlling mechanisms of the devices in the hardware.

This kind of separation makes it easier to control the devices without extensive knowledge of the hardware details. This also makes it possible to change the hardware without affecting the control software. All modifications are made in the abstraction layer. However, the abstraction layer is a difficult design task, and all future changes to the hardware may be impossible to predict in design. This makes the system maintenance more difficult and sometimes the application programmer must circumvent the hardware abstraction layer in order to achieve something the layer does not allow. For example, the device interface may offer only 5 ms sample rate for measurement. If the programmer needs more precise results for high resolution calculations, the device must be read directly. This causes abstraction leaks and makes the software unportable to other hardware platforms. Therefore the layering should not be breached.

Consequences

+ Hardware devices that are similar or of same type may be used through an uniform interface.
+ The device driver may be changed under the interface, without any changes to the actual application
+ New device types can be used by implementing only a new interface.
+ The uniform application interface makes application porting easier.
– It may be difficult to design an interface that is generic enough. If the hardware implementation changes drastically, a new interface may be required. This may make the maintenance of the application difficult.
– If the application programmer must circumvent the interface, the design-by-contract [19] is violated and abstraction leaks may be introduced to the design.
– The performance may be compromised, because the abstraction layer brings an additional level of indirection to the system.

Resulting Context
After this pattern has been applied, similar hardware devices can be controlled via an abstraction layer, which makes possible to change the underlying hardware.

Related Patterns
LAYERED ARCHITECTURE [15] describes a method to add layers to increase abstraction level. Our pattern is a specialization of this.

Known Usage
All modern desktop operating systems have a hardware abstraction in use. This normally includes typing hardware devices by type or by connection. For example, a sound card is attached via USB bus. The application developer needs not to know anything about the basic sound card connection, as the hardware is abstracted. The operating system takes care of the vendor-specific details if the sound card is changed to another, for example a PCIe sound card.

8.2 Device Proxy

Context

The distributed embedded control system uses functional blocks or subsystems to provide a functionality or a service. These functional blocks usually consist of hardware and software. The are manufactured by a variety of different vendors. During the long life cycle of the product, subsystems or functional blocks may get replaced with new versions or systems from other vendors. Therefore, the system should offer support for different kinds of functional blocks or subsystems. In addition, these blocks should be used in a way that the location of the block does not affect the usage. Furthermore, solution offered by HARDWARE ABSTRACTION LAYER is not enough as a whole subsystem is replaced instead of a single device.

Problem

How the system could offer a unified way for a developer to control different kinds of functional blocks or subsystems without knowing the details of the device?

Forces

– *Extendability*: It should be possible to extend the system with a new subsystem or a functional block.
– *Variability*: Product variants can have different subsystems installed in their system configuration. There should be a way to control different kinds of subsystems in a unified way.
– *Changeability*: It should be possible to use different kinds of subsystems in a product. There might be different variations of the product which have different kinds of subsystems. For example, the forest harvester can have different kinds of harvester heads depending on the model.
– *Reusability*: Same subsystems can be used in different products.
– *Testability*: It should be possible to replace a subsystem or functional block with a stub during testing phase.
– *Replaceability*: Subsystem should be easily replaceable during the long life cycle of the product.
– *Interoperability*: System should function properly with different subsystems created by different vendors.
– *Distribution*: It should be possible to use the subsystem regardless of actual physical location of the subsystem.

Solution

The subsystems in a product should be easily changeable in order to prepare for future changes or to provide support for wide range of subsystems. However, when the subsystems or the functional blocks are replaced, the system should not require major changes on code level. Therefore the subsystems used in a product should be abstracted. The solution is to developed subsystems in a proxy-server fashion. The server, i.e. the subsystem, implements the actual service while a proxy provides access to the respective server hiding the actual location and implementation of the service. In this context, by proxy we mean an interface hiding the location of the device and providing access to the services offered by the server.

The interface is designed by the product vendor and devices that are compatible with the interface can be used in the product. It is challenging to develop an interface that supports wide range of products and has required functions to control the device. On the other hand, the interface should not contain anything extraneous.

During the system startup the system creates a proxy and a server component for each subsystem present in the system. When a service from a subsystem is needed, a proxy handle is requested from the system and it is used to access the service provided by a subsystem. Proxy handle grants a right to use the proxy. The handle can be given for one subsystem at a time. In this way, the whole subsystem, e.g. harvester head, is abstracted from the system's point of view. Additionally, proxies can be stubbed and those stubs can be used in the testing phase instead of actual devices.

Device abstraction can be challenging, especially when the devices that are abstracted use common resources. This means that if an interface function of device A is used, it might block some functions of device B. Poor implementation of such interface may result into deadlock situations. Therefore one should be really careful when implementing such an interface.

Consequences

+ Different kinds of devices or subsystems can be controlled using the same proxy interface. This results in better support for different kinds of devices.
+ A device can easily be replaced by a dummy proxy during the testing phase of the system.
+ The system becomes more understandable as it is easy to divide it into separate functional blocks.
+ The subsystem can be used in a uniform way regardless of actual physical location of the subsystem.
– Abstraction level becomes very high. This can cause unexpected problems as the subsystems may have common resources which need to be mutually exclusive. This may result in unexpected deadlock situations.
– Designing the subsystem usage in proxy-server fashion can be challenging. How to make sure that the interface does not limit too much the subsystem usage?
– Higher abstraction level usually requires more processing power.

Resulting Context

The result is a system where a proxy interface is used to control a complete subsystem. This subsystem is also easily replaceable by another subsystem which uses the same proxy interface.

Related Patterns

HARDWARE ABSTRACTION LAYER describes a solution to abstract different hardware components. In one sense, the device proxy takes this abstraction further by abstracting whole subsystems. START-UP MONITOR could be used to detect components that are present at start up and need a proxy to be created.

Known Usage

In an elevator group there can be elevator cabins from different vendors that are controlled by one system. Therefore the elevator system uses lift cabin proxies to abstract the cabin from the system's point of view. At the start up, system detects which kind of cabins are present in the system and which proxies for the cabins need to be created. Proxies that can be created are pre-defined by the system vendor. In this way, the system can control cabins from different vendors easily. If compared to solution provided by HARDWARE ABSTRACTION LAYER, the proxy solution offers the abstraction of whole elevator cabin.

8.3 Concurrent Execution

Context

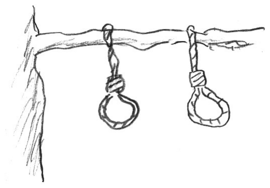
The system is divided into nodes by using ISOLATE FUNCTIONALITIES. Some of the nodes need to run multiple simultaneous tasks with a single processor. There may be some interaction between the tasks.

Problem

How to run tasks concurrently on a single node?

Forces

- *Response Time*: Tasks may have different response time requirements.
- *Scalability*: A new task may be added to a node or existing tasks may be divided into multiple tasks.
- *Extendability*: There might be multiple tasks which should run in parallel.



Solution

A single core can do semi-parallelism by switching between tasks quickly so that to the observer it seems that multiple tasks are running concurrently. This requires an additional scheduler component which allocates time slices for different tasks. Depending on the scheduling algorithm, the tasks may be interruptive or non-interruptive. With non-interrupting scheduling, a task may finish what it is doing before the switch. If task takes a long time, then other tasks have to wait until the primary task is finished. If tasks are not evenly sized this may lead to long waits. Interruptive scheduling on the other hand may create strong feeling of concurrency as the tasks are switched after certain time slices. As the task is probably in the middle of its work, it might have some resources in use which some other tasks also require. If a certain task has to be done before the deadline, interruptive scheduling may prevent from doing that. Additionally, task switching requires time for context saving and loading. These both add to overhead which is away from the actual tasks.

If the scheduler and the operating system running on the processor is advanced enough, there can be two types of tasks running. The most common type for a task is a process. Contents of different processes are shielded from each other and therefore one should not be able to modify memory space of an other process. The second task type is a lightweight process called a thread. One way of looking at threads is to think them as subtasks inside a task. Threads inside a process are not shielded from each other and as a result collaboration between threads is easy as they can share the same variables. On the other hand, one misbehaving thread may crash a process and its threads. Interrupts can be used also for scheduling if the scheduling needs are not extensive.

Parallelism, even interruptive semi-parallelism, will have it own problems with resources. The resource access has to be safeguarded with locks so that simultaneous writes and reads to same area do not create chaos. When locks are used, by contract some memory area can be only accessed through the lock. Depending on the need, there is either a single lock or two locks called read lock and write lock. If a single lock is used, whoever has locked the resource is the only one who can read and write to it. If two locks are used, the write lock prevents from reading or writing to the value as it is going to updated by someone else. Read lock on other hand prevents from updating the value as it is being read.

A lock is usually provided done by using a semaphore which is a protected variable which can be read, modified and written in single operation. Semaphore can be either binary (yes/no) or counting (a queue). A mutex (mutual exclusion object) is an object which contains a semaphore and provides the locking, queuing and unlocking services. Locks and any similar access control system may create additional problems such as deadlocks where multiple participants wait each other to release a resource in circular chain or resource starvation where the participant never gains access to the resource.

Consequences

+ Concurrency may raise the system utilization rate.
+ It might be easier to design separate tasks than create one single task.
− Dependencies between tasks may create problems.
− Additional scheduler might be needed to manage the tasks.

Resulting Context

The pattern results in a system where tasks can be run simultaneously.

Related Patterns

STATIC SCHEDULING and CENTRALIZED SYNCHRONIZATION provide tools for process time allocation. SEPARATE REAL-TIME provides a solution for those system which have strict real-time requirements.

There is extensive amounts of literature available on parallelism, for example, see [20]. Additionally, there exists a book on parallel programming patterns [21], there is a pattern language for parallel programming [22], and at least one conference has been organized for parallel programming patterns called ParaPLoP [23].

Known Usage

Forest harvester head grappler controller does multiple concurrent tasks when a tree trunk is cut to standard sized pieces. When a section of a tree trunk is moved under the saw, a grappler has to hold the tree with adequate pressure, move it though the grappler with high speed, measure the length of the tree moved past the saw and stop the movement when required length has been moved.

8.4 Operating System Abstraction

Context

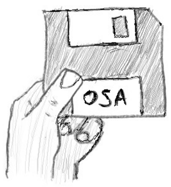
There is an embedded control system, where the software platform uses an operating system (OS) to provide the basic services as HARDWARE ABSTRACTION LAYER is not enough. The operating system may be a commercial off-the-shelf product (COTS) or an in-house product.

Problem

How to make sure that the operating system can be changed with only few modifications?

Forces

- *Adaptability*: It should be possible to choose a suitable operating system for the underlying controller.
- *Scalability*: The underlying controller should be able to be updated for more performance.
- *Reusability*: Same application should be able to run on multiple platforms.
- *Usability*: The developer should not need to know all possible OS APIs the application runs on. The OS APIs should be therefore be hidden from the developer.
- *Changeability*: The OS must therefore be changeable, because the product may have longer lifespan as most commercial operating systems.
- *Testability*: There should be known interfaces for testing the application behavior. In embedded environment the application/OS -interface is among the most important ones.



Solution

The architect must wrap the operating system services in such way, that they are usable using a generic interface. The basic services of an operating system usually include

- interrupt handling,
- memory management,
- task execution with some kind of multitasking,
- mutual exclusion mechanisms,
- device drivers,
- storage interface and
- communication services.

In addition to these features, more advanced operating systems may provide resource protection for different processes, virtual memory and disk access services. All these services must be presented via such interface that no operation system specific parts are visible to the application developer. The interface may be implemented as a library, it may be a statically linked part of the application or it may even act as a platform to run the developed applications.

In an ideal case, the operating system abstraction level simulates the more advanced services the particular underlying operating system can not provide, for example, mutual exclusion mechanisms for parallel tasking. On the other hand, the OS abstraction should make sure that different capabilities of operating systems do not cause harm. For example, if some supported operating system has a filesystem that does not support lowercase filenames, it should not be possible to create a situation where filenames become ambiguous because of using lower case names. Sometimes the interface turns out to be unsuitable for the intended use or the circumstances have changed in an unpredicted way. In such cases, the developer must circumvent the interface and use the OS services directly. This causes portability issues as the abstractions leak from the lower layer to an upper layer.

Consequences

+ Productivity increases as all applications that comply to the OS abstraction interface are runnable on all other platforms.
+ The common platform makes the software development easier as the developer need not to know all

peculiarities of certain platforms. It is sufficient to know the OS abstraction API.
+ The OS may be changed if for example the support for the chosen OS ceases. The system that has a long product lifespan is not stuck with one commercial OS vendor.
– It is difficult to design a interface that covers all OS dependent services and is compatible with all possible operating systems.
– Supporting legacy operating systems may prevent using most advanced features of other operating systems.
– Supporting legacy operating systems may also require implementing much of the advanced features of the OS in the abstraction layer, thus making it cumbersome and error-prone.
– The OS abstraction API adds latency and takes resources as all operating system services must be accessed through an additional layer. To make things worse, the OS services are used frequently and may be sensitive for timing.
– The abstractions are prone to leak because of bad designs and changing requirements.
– The OS abstraction interface creates an additional artifact to maintenance.

Resulting Context
The result is a system that isolates its underlying operating system under a common interface. The interface makes the application porting easier.

Related Patterns
The HARDWARE ABSTRACTION LAYER pattern helps in making device drivers more portable. The operating system should present the CONCURRENT EXECUTION pattern to provide multitasking capabilities. BUS ABSTRACTION is a similar kind of an abstraction layer to provide the developer an uniform interface to a common resource. If the OS abstraction is developed in-house, it should be implemented using patterns for operating systems.

Known Usage
A top hammer drilling machine features a PC to provide higher end services. This PC runs an operating system. In order to achieve portability, a POSIX compliant OS is used. POSIX is a common IEEE standard for making Unix software portable across different operating system variants. If an operating system complies to the POSIX standard, all POSIX compliant software written for it should be able to run on all compatible platforms. This makes all drilling software more easy to port to other OS platforms. The principle is illustrated in Fig. 8.4.

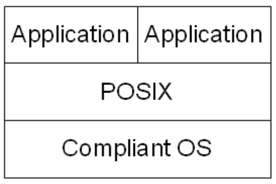


Fig. 23. The POSIX layer abstracts the underlying OS

References

1. IEE: Embedded systems and the year 2000 problem guidance notes. Embedded Systems and the Year 2000 Problem: Guidance Notes., IEE Technical Guidelines 9:1997 (1997)
2. Leppänen, M., Koskinen, J., Mikkonen, T.: Discovering a pattern language for embedded machine control systems using architecture evaluation methods. In: 11th Symposium on Programming Languages and Software Tools (SPLST'09), Tampere, Finland (August 2009)
3. ISO 11898: Road vehicles – Controller Area Network (CAN). ISO, Geneva, Switzerland. (2003) http://www.iso.org/.
4. CiA: CANopen Specification. CiA, Nürnberg, Germany. http://www.can-cia.org/.
5. SAE: J1939 Standard. SAE International, Warrendale, USA. http://www.sae.org/.
6. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. Wiley (May 2007)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (October 2003)
8. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications. Norwell, MA: Kluwer Academic Publishers (1997)
9. Herzner, W., Kubinger, W., Gruber, M.: Triple-T (Time-Triggered-Transmission) - A System of Patterns for Reliable Communication in Hard Real-Time Systems. Proceedings of EuroPLoP 2004, Hillside Europe (2004)
10. EventHelix.com: STL design patterns II. http://www.eventhelix.com/realtimemantra/Patterns/stl_design_patterns_2.htm#Priority%20Message%20Queue%20Pattern
11. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc. (1987)
12. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM **21**(7) (July 1978) 558–565
13. Hanmer, R.: Patterns for Fault Tolerant Software. John Wiley & Sons (2007)
14. Coplien, J.O., Schmidt, D.C.: Pattern Languages of Program Design. Addison-Wesley Professional (1995)
15. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons (August 1996)
16. Noble, J., Weir, C.: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
17. Koch, N., Rossi, G.: Patterns for adaptive web applications (2002)
18. Vlissides, J.M., Coplien, J.O., Kerth, N.L.: Pattern Languages of Program Design 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
19. Meyer, B.: Applying "design by contract". Computer **25**(10) (1992) 40–51
20. Stalling, W.: Operating Systems: Internals and Design Principles, 6th Edition. Pearson education international (2008)
21. Mattson, T.G., Sanders, B.A., Massinqill, B.L.: Patterns for Parallel Programming. Pearson education international (2004)
22. OPL Working Group: A pattern language for parallel programming ver 2.0. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.
23. ParaPLoP 2009: Workshop on parallel programming patterns. http://www.upcrc.illinois.edu/workshops/paraplop09/program.html.
24. Eloranta, V.P., Hartikainen, V.M., Leppänen, M., Reijonen, V., Haikala, I., Koskimies, K., Mikkonen, T.: Patterns for distributed embedded control system software architecture. Technical report (2009) Presented in VikingPLoP 2008.
25. Eloranta, V.P., Koskinen, J., Leppänen, M., Reijonen, V.: Software architecture patterns for distributed machine control systems. In: Proceedings of EuroPlop '09. (2009)

Appendix 1 - Pattern Thumbnails in Alphabetical Order

In Table 8 the pattern thumbnails are presented. Thumbnails consist of pattern name and short problem description. The patterns in the table are arranged to an alphabetical order. Earlier versions of some patterns are presented in [24] and [25]

Table 8: Pattern thumbnails

Pattern Name	Description	Page
ACCESS CONTROL	Unauthorized modifications to the common state information are prevented using access control component.	81
BUS ABSTRACTION	Nodes communicate via a bus using messages. The message bus is abstracted so the bus can be changed easily and the sender does not have to know the actual location of the recipient.	11
CENTRALIZED SYNCHRONIZATION	The program is scheduled statically during the compile time. During run-time an external clock signals the nodes to stay in schedule.	72
CENTRALIZED UPDATES	A single unit takes care that software versions on all nodes are compatible with each other.	91
CHECKPOINT	Snapshots are taken periodically so that information is not lost due to a fault.	79
CONCURRENT EXECUTION	To provide concurrency, multiple tasks are run on a single processor.	102
CONFIGURATION PARAMETER VERSIONS	Different software components use the same base parameters. Newer software version may have some more advanced parameters which are separated from the older software version parameters.	95
CONVERTING MESSAGE FILTER	In a system with multiple messaging channels the messages are filtered according to the channels where the interested recipients reside. In some cases this might need conversion from a message format to another.	21
DEVICE PROXY	System has subsystems that are manufactured by different vendors. Different kinds of subsystems should be controlled in uniform way.	100
DISTRIBUTED SAFETY	Potentially harmful functionality is divided into multiple nodes in order to prevent a single node from creating havoc.	61
DISTRIBUTED TRANSACTION	An action requested over a bus consists of several logically connected command messages and it should be executed successfully as a whole whenever possible.	23
EARLY WARNING	The solution introduces a way to make sure that possible buffer overflow situations can be detected before they occur.	19
EARLY WORK	Processes are initialized and all resource intensive tasks are carried out at startup. This frees resources for the main functionality during run-time.	40
HARDWARE ABSTRACTION LAYER	The system is designed so that application programmers do not need to care about details of the actual hardware.	98
HEARTBEAT	The node uses periodic messages to ensure proper communication with other nodes through bus. This enables recovery actions to be taken as soon as possible.	63
ISOLATE CONTROL ALGORITHM	To improve changeability, control algorithms are abstracted in a way that they or their parts can be easily changed.	54
ISOLATE FUNCTIONALITIES	The solution divides a system into subsystems according to functionalities. The subsystems are connected with a bus.	9
LIMP HOME	If a sensor or a controller malfunctions the machine should still be at least partially operable.	52
LOCKER KEY	The solution provides an efficient way for communicating between processes while avoiding dynamic memory allocation.	34
MESSAGE CHANNEL SELECTOR	There are several different message channel types in the system. A separate component selects the appropriate message channel for each message.	26
MESSAGE QUEUE	Nodes communicate asynchronously and the amount of messages that can be simultaneously processed is limited. Therefore, the rest of the messages have to be queued.	13
NEXT STABLE STATE	When a potentially harmful situation occurs, the system can still provide a last transition to the next stable state.	59
ONE AT A TIME	A node is prevented from flooding the bus with babble by limiting the amount of messages on the bus.	15
OPERATING SYSTEM ABSTRACTION	The underlaying operating system is abstracted so that it can be changed with only few modifications to the application.	104
OPERATOR PROFILE	The machine operator may use preferred settings regardless of the current machine in use.	42

Continued on next page...

Table 8: (Continued)

Pattern Name	Description	Page
OPPORTUNISTIC DELEGATION	In transient system tasks are delegated to a subsystem with available resources. This results in higher resource utilization.	44
PERMISSION REQUEST	To prevent conflicting actions in a distributed there must be a component that decides which actions can be taken.	30
PRIORITIZED MESSAGES	Messages can have varying importance and hence some messages need more urgent attention. Therefore, important messages should be handled first.	17
PRODUCT FAMILY CONFIGURATION	A common pool of software components is used with a configuration file to vary product families and products.	93
PROTOCOL VERSION HANDSHAKE	A common protocol is used to detect that components having different software versions can function together.	87
REDUNDANCY SWITCH	There is one active unit and multiple secondary units in the system. When a active unit malfunctions a separate component decides which secondary unit takes over the control.	70
REDUNDANT FUNCTIONALITY	A node controlling a functionality is multiplied to ensure the high availability. If the node malfunctions, a secondary unit will take over control.	68
SAFE STATE	If something potentially harmful occurs all nodes should enter a predetermined safe state.	57
SEPARATE REAL-TIME	The system is divided into separate levels in order to separate real-time functionality from the functionality which time behavior is not predictable.	37
SNAPSHOT	The complex system-wide state information can be stored and restored for further usage.	77
START-UP MONITOR	During start-up all devices are started in certain order and with correct delays. Additionally, care is taken that there is no malfunctions.	83
START-UP NEGOTIATION	The system device setup may change. Therefore, available features are detected during start-up and core functionality is confirmed to be functional.	85
STATIC RESOURCE ALLOCATION	Critical services are always available when all resources are allocated when the system starts.	48
STATIC SCHEDULING	All processes are scheduled statically during the compiling time. This gives predictability to execution times.	50
THIRD-PARTY CONFINEMENT	The pattern results in a system with a capability to provide a restricted environment for third-party applications.	46
UNIQUE CONFIRMATION	The default confirmation response must not be same twice, in order to eliminate the change of accidentally repeating confirmations causing unwanted actions.	32
UPDATING	Software and configuration of a device should be replaceable.	89
VARIABLE MANAGER	The complex system state information is efficiently shared and stored by using a separate component.	75
VECTOR CLOCK FOR MESSAGES	The order of events can be deduced by using a vector clock to timestamp events instead of normal timestamps which have clock skew in a distributed system.	28
WATCHDOG	The health of a node is monitored. If the node malfunctions, a recovery action can be taken.	65