Software Architecture Patterns for Distributed Machine Control Systems

Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen, and Ville Reijonen *
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

Abstract. In distributed machine control system the software architecture is typically a weak spot because developers lack good design practices. Software architecture design patterns have been found useful for improving the software design. However, there is no comprehensive collection of patterns for distributed machine control systems even though many patterns and pattern languages can be applied to this domain. We carried out architecture assessments in Finnish machine industry and this gave us a possibility to collect recurring solutions as patterns for this domain. The resulting pattern language constitutes a comprehensive collection of solutions for distributed machine control systems. In this paper, we suggest a pattern language for embedded distributed control systems and introduce seven representative patterns from this language.

1 Introduction

A distributed machine control system consists of multiple embedded controllers which communicate with each other. These embedded controllers are devised to control, monitor or assist the operation of equipment, machinery or plant [1]. In this context, a distributed machine control system is a software system that controls large machines such as harvesters and mining trucks. Environments of such systems impose special requirements for the software architecture such as distribution, real time, and fault tolerance. For example, harvester head software architecture has to meet certain requirements because productivity demands fast operation and high measurement precision. As machine control systems have become larger and complex, 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. We present a pattern language specifically targeted for this domain. This pattern language assists software architects in designing high-quality systems.

Although there is a plethora of design patterns for software architectures, there are fewer patterns for high level distributed embedded control system design. The identification of patterns in this domain is hard due to several reasons. Typically, the software in many embedded systems has been poorly documented, and the hardware architecture design tends to dominate development of the software architecture. Designers of such systems have often different area of expertise than software system design and are more familiar with the hardware technologies than with modern software engineering. However, there are proven solutions, which are

informally communicated among the architects. The purpose of this paper is to document this folklore as patterns.

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. The process for architectural assessment was derived from ATAM [2]. Because the fundamental goal was to create a comprehensive pattern language for this domain, essential solutions were captured and documented as patterns regardless of their prior existence in other pattern collections. The collection process of pattern language is documented in more detail in [3].

In this paper, we introduce a pattern language consisting of found patterns for embedded distributed control systems. In addition, we describe in more detail seven patterns typical for the domain that we consider to be the most interesting. The full set is available online [4].

The patterns were formulated and written down by the members of the assessment team. As the pattern mining process was associated with an architectural assessment, it was easy to find the quality attributes the patterns are related to. This is visible in pattern descriptions in the *Forces* section. Each force is prefixed with the related quality attribute. Quality attributes make it easy to see which aspects of the system design the pattern affects.

A figure is sketched to describe the idea of the pattern. These figures are intended to give a quick intuitive idea rather than a technically sound solution model. A example application of a pattern, abstracted from the actual occurrences, is presented in the pattern description as *Known Usage* section. In addition, *Known Usage* may incorporate a technical diagram.

2 The Pattern Language

Our pattern language consists of 45 patterns in its present form (Fig. 1). The patterns with gray background color are presented in this paper. When constructing the language, the patterns were at first grouped loosely together based on their area of effect. Patterns which were affecting the design of larger parts of the system were grouped together and patterns with smaller impact were put together depending on which part of the system they improve. This gave an overall view of the pattern relationships.

In the second phase we connected the isolated groups with arrows. The semantics of an arrow pointing from pattern A to pattern B in our language is "pattern B refines pattern A". This means that if the architect has solved some design problems with pattern A, the design context is now compatible with the required context of pattern B. The designer might look at all refining patterns if there are still some unsolved problems in the context.

The connecting of the patterns simulated the thinking process of an imaginary architect, who is tackling the problems in the design one by one and advancing from more general problems towards more detailed design. The key question was to try to understand which is the problem the architect tries to solve and which patterns are related to this problem. As the patterns change the system design, the design problems will change. New patterns can be applied to solve, if necessary, the new design problems.

The simulated use of the pattern language helped to see some "blind spots", solutions that emerged in the software architecture assessments but were not yet identified as patterns. Often there were separate groups of patterns that were difficult to link with the rest of the pattern

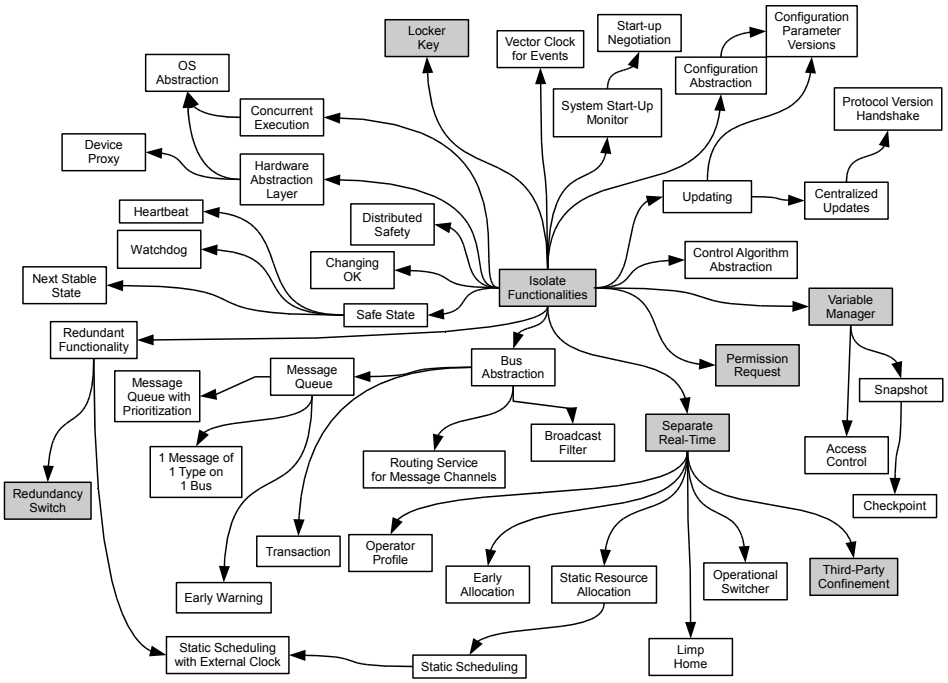


Fig. 1. Pattern language for machine control systems. Grayed patterns are presented in this paper.

language. After adding a pattern solving a problem in a more general context, the orphaned pattern group could become children of this pattern. For example, there was a group of system updating related patterns that was not linked with the language. This was solved by adding UPDATING pattern that describes the basic mechanisms needed for updating. Therefore, the pattern connecting is very efficient approach to recognize patterns.

3 Patterns

In this section, we present the selected patterns that are most interesting and typical for the domain. The selected patterns are grayed in Fig. 1.

We use the following template to describe our patterns. First, we present *Context* where the pattern can be applied. The next section is the description of *Problem* that the pattern will solve in the given context. In *Forces* section we describe the motivation or goals that one may want to solve by applying this pattern. Furthermore, we give *Solution*, argument it in *Consequences* section and give *Resulting Context* that 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.

3.1 Isolate Functionalities

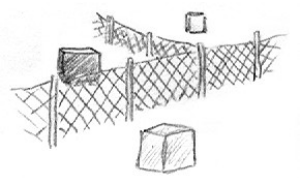
Context
An embedded control system is needed to control a large machine which consists of different kinds of sensors, actuators and controlling devices.

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 expertises.
- *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 [5]. The communication on the bus takes place using a common message format. For example, CANopen [6] and J1939 [7] 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

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 [8] and MESSAGE BUS [9] 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.

3.2 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 in conflict in 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.
+ 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

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 [10] presents a paradigm that there has to be always someone responsible for decision. PROCESS IMAGE or 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.

3.3 Third-Party Confinement

Context

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?

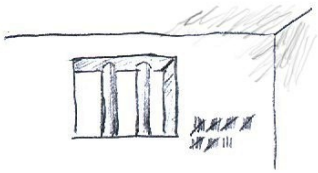
Forces

- *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.
− 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
A system with a capability to provide a restricted environment for third-party applications.

Related patterns
QUARANTINE [10] 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.

3.4 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

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 [11] pattern. BLACKBOARD [12] 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.

3.5 Redundancy Switch

Context
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?

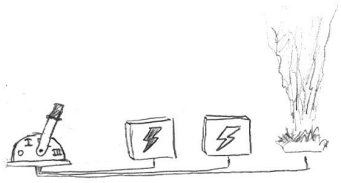
Forces
- *Availability*: A backup unit should take over in predetermined strict response time when the active unit malfunctions.
- *Reliability*: A unit can not make sanity checks for itself as self-tests may not be reliable enough or consume too much resources.
- *Reliability*: In a system with even 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

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.

3.6 Locker Key

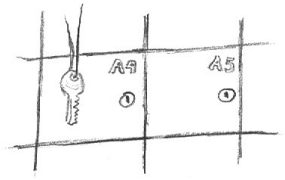
Context
A distributed embedded control system where ISOLATE FUNCTIONALITIES has been applied. A controller has shared memory that multiple processes can access. Processes communicate with each other using some kind of messaging scheme. Intense communication between processes may occur. Direct interprocess 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.
+ 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

An embedded controller with fast interprocess messaging capabilities.

Related Patterns

EARLY ALLOCATION 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.

4 Conclusions

In this paper, we have introduced a pattern language that was collected during architectural evaluations in Finnish machine industry. We have illustrated this language with seven example patterns. These patterns reflect the some characteristics of evaluated embedded systems such as distribution, real time and fault-tolerance.

Although we have a pattern language the pattern collection work is expected to continue. Therefore, pattern language is constantly evolving, leading to a more comprehensive and more systematically organized pattern language of embedded machine control systems. Even though there could be some gaps, the pattern language covers the most important discovered solutions.

5 Acknowledgements

We like to thank our shepherd Jorge L. Ortega Arjona for his insight. Also we would like to thank for valuable feedback our EuroPLoP 2009 workshop members Anjali Das, Arto Juhola, Farah Lakhani, Martin Wagner, Stefan Sobernig and Tim Wellhausen. Special thanks to our professor Kai Koskimies and colleagues who gave us valuable input. Additionally, our appreciation goes to Jim Coplien who guided us to the path of patterns.

References

1. IEE: Embedded systems and the year 2000 problem: Guidance notes. IEE Technical Guidelines 9:1997 (1997)
2. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional (2002)
3. 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). (2009)
4. Eloranta, V.P., Koskinen, J., Leppänen, M., Reijonen, V.: A Pattern Language for Distributed Machine Control Systems. Technical report, Tampere University of Technology (2010) http://practise.cs.tut.fi/publications.php?project=sulake.
5. ISO 11898: Road vehicles – Controller Area Network (CAN). ISO, Geneva, Switzerland. (2003) http://www.iso.org/.
6. CiA: CANopen specification. CiA, NÂ¸rnberg, Germany. http://www.can-cia.org/.
7. SAE: J1939 standard. SAE International, Warrendale, USA. http://www.sae.org/.
8. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. Wiley (May 2007)
9. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
10. Hanmer, R.: Patterns for Fault Tolerant Software. John Wiley & Sons (2007)
11. 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)
12. 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)