Patterns Related to Software Updating for Distributed Control Systems

Ville Reijonen
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

1 Introduction

In this paper we will present two patterns for distributed machine control systems. A distributed machine control system is a software system that is specifically designed to control a certain hardware system. This special hardware in turn operates some work machine, which can be a forest harvester, a drilling machine, elevator system etc. or some process automation system. Some of the key attributes of such software systems are the close relation to the hardware, real time requirements, safety issues, fault tolerance, high availability and long life cycle. Distribution plays a major part in the control systems as the different parts of the machine are physically far from each other and they must communicate with each other in order to perform their functionalities.

The patterns in this paper are part of a larger collection which was collected during years 2008-2011 in collaboration with industrial companies. Some real products by these companies were inspected during architectural evaluations and whenever a pattern idea was met, the initial pattern drafts were written down. These draft patterns were then reviewed by industrial experts, who had design experience from such systems. After these additional insights, the current patterns were written. The published patterns are a part of a larger body of literature, which is not yet publicly available. All these patterns together form a pattern language, which consists of more than 70 patterns at the moment.

2 Patterns

Other patterns are referenced in this paper using SMALL CAPS. Patlets of the patterns that are referenced in this paper are presented in Table 1.

Table 1. Patlets of Patterns referenced in this paper but not present

Pattern name	**Description**
CONTROL SYSTEM	How to implement a work machine that offers interoperability between systems and is highly operable with good performance? Implement control system software that controls the machine and can communicate with other machines and systems.
REDUNDANT FUNCTIONALITY	How to ensure availability of a functionality even if the unit providing it breaks down or crashes? Clone the unit controlling a critical functionality. One of the units is active and other is in a hot standby mode. If the controlling unit fails, hot standby mode unit takes over controlling the functionality.
START-UP NEGOTIATION	How to verify that required devices for base configuration are available and find out which features can be enabled based on the other found devices? Make all nodes of the system to announce themselves by sending a short information package to the bus after self check. In the package, the node tells of its existence and also what it is, what it requires and what it can provide and in what limits. Design a central node that gathers this information and by a deadline builds a list of nodes and thus devices available.

2.1 Updateable Program Storage

... you have distributed CONTROL SYSTEM consisting of many nodes which each have software. Nodes in the system are connected over a bus. A shipped system might be in service for tens of years, sometimes located in hard to access locations. Due to long service time the original requirements rarely cover the needs of the future. There is often needs to modify and update the system during its lifetime.

* * *

How to support changing requirements for the software of a system with long life cycle?

While the time passes, software of a node may evolve by receiving new features and bug fixes. Not only the node but the system setup might go through changes that move it out of the capabilities of the original software. This is almost a certainty in systems with long planned life cycles up to thirty years. Thus, a system who's parts cannot be updated will literally be stuck to past. If renewal is not an option, the systems value will diminish greatly by every new requirement which it cannot fulfill.

Usually cost is a big issue when systems consisting of multitude of nodes are being built. This usually means decisions which lock the system design constraints to known needs. One way of saving in costs is to put the software on read only memory (ROM), but this also makes it difficult to replace the software with never version. If the software has been written to ROM chip, the chip has to be replaced for the software update; the chip is the software. Consequently, making chips and the required logistics for delivery does not make ROM chips flexible and cheap option.

In working machine one would prefer to cover well all easily damaged components such as electronics. To be manually changeable the software chip would need to be easily accessible which it will not be when well covered. A node might be embedded to such a location in the system that it is hard to reach or detach. Attaching separate cable for flashing or replacing the chip might require extensive amount of work and require expertise which is not always available. If multiple nodes are being updated this could be time consuming. Changing the software should be easy.

When a system or part of it is changed it involves a risk of failure. This risk should be minimized but as it is not always possible, there should be ways to mitigate aftereffects. It should be possible to retry any operation and even reverse it when required. It would be even preferable if this could be tried without any external help as such is not always available or even anywhere close to the location where the system resides.

It is difficult to fathom all possible future needs in current version of the software as those needs have not yet materialized. The only way to enable the flexibility needed by the future is to be able to change the software to meet the new requirements. These requirements might lead, for example, to faster operations, better accuracy, better energy efficiency, support for new devices, etc. None of these can be met unless the software is changeable.

Therefore: **For each node place the software on rewritable persistent storage. It should be possible to update the memory without its removal over the bus. Any update failure should not prevent from trying again.**

* * *

The usual way for the update to commence is to set the node to separate update mode. The program which does the updating could be either reside on separate ROM space, be part of the program residing on the persistent storage or be loaded to the memory of the node as first step of a update process over the wire via the bus. Depending how much memory the node has the update can be either transmitted as whole or fed in suitable sized blocks over the wire. When the update commences an under-update flag is set. The update program rewrites the persistent memory with the data it has received. When all the data has been transmitted, update is verified, under-update flag is removed and the node is rebooted.

When the update is commencing, the program code under update should not be used as it would probably lead to undefined behavior and unknown errors. It should not be possible to run the update by mistake and the updating functionality should be shielded from unintentional or malicious use. Updating software over the bus can be simple what comes to the update operation but it requires additional safeguards. The node should be shielded with access rights or different usage mode should be required for the update functionality.

There should be a way to flag the device as being under update. There might be partition in the memory or separate memory area reserved for flags. After successful update the device should be flagged again as active. This way device would not be used even by mistake if the update fails. The device update may fail or the device might break during the update. Therefore, after writing process, the result or functionality should be verified to know that the update was successful. The verification can be done either by counting checksum from the written result and comparing it to expected result or doing byte by byte comparison. Only if the verification is successful under-update flag may be removed.

If a node boots and under-update flag is set, the node software should try to enter update mode. This is one way to start update process that ensures that no program code is in use. This can also be handy method for resuming update, for example, in case of power failure. If updating program resides on separate ROM space it cannot be updated, but the device can always boot to updating mode. If the program resides on persistent storage the update program may also start, but only if previous update has not overwritten the update program with garbage. Therefore, to be on safe side, the area of the update program should not be updated unless it is really necessary. If the updating program has been loaded over the wire, it might so that it cannot be loaded again. This might happen if the node is not able to boot to required state where program can be loaded.

Update can always fail due to power loss, bad connection, faulty hardware, etc. In any case a device which is in transitory state should not be used and under-update flag guarantees that the device does not become active. In some cases the update may continue, and in other cases it might not. If there is a risk that disturbed update might render the node unusable or there could be unrecoverable memory errors, it might be good idea to apply REDUNDANT FUNCTIONALITY pattern and duplicate the memory so that there is always one complete working copy of update program or contents of persistent storage available.

There should be more storage available that what is required by the first version of the software as new versions usually gains new features which consume additional space. How much more space is needed depends on the purpose of the device and its development path. If nodes are shipped out to the world in large quantities it might be reasonable to plan future requirements to find suitable storage size. Consequently, if only few nodes are shipped, making such a plan probably costs more than what can be saved by using smaller storage. If the node does not have enough storage for the envisioned functionalities, squeezing the software to smaller size will make programming much more complicated and costly.

Nodes in larger system could be all updated simultaneously to consistent state by applying CENTRALIZED UPDATES pattern. This requires that the bus speed is high enough for fast software delivery.

* * *

Rewritable persistent storage in suitable quantities adds the necessary flexibility needed to support needs of the future. Meanwhile it requires also safeguards to prevent unintentional or malicious updating. This adds design costs but in a machine which consists of multitude of nodes which are all updateable, the same design can be used all over. When larger run of machines is produced the cost per node should be not be an issue. Still, rewritable storage is not as cheap as ROM is, but gained flexibility should be more than adequate to offset the costs. Cost for replacing buggy ROMs cannot even be compared to price of running software update.

Updateability rises the value of the system over its lifetime. When the system can be updated, new features, bug fixes and other enhancements may be received. Some of these might be covered by service contract while others might be add-ons to existing system. By providing easy way of updating, less is wasted on non-value adding operations to get the update available. Still, an easy mean to service software should not be seen as a permission to use the end users as software testers.

* * *

In harvester’s boom a grappler module is updated to new software version. The machine is put to stop. The service person connects usb-stick containing update for the grappler to cabin pc usb-port. The updating tool sends the new software to the grappler module’s updater program over the bus 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 machine can be operated with functional grappler module.

2.2 Centralized Updates

... you have and distributed CONTROL SYSTEM where each node can be updated over the bus due to UPDATEABLE PROGRAM STORAGE. The bus has reasonable throughput so that it is sensible to update nodes over the network. START-UP NEGOTIATION is in place and provides information what features are available based on existing nodes. Updating each node one by one is tedious manual task. It is also potentially error prone as the operator may forget to update a node or nodes.

* * *

How to update the entire system conveniently with a single action?

The nodes in the system communicate with each other to function as a whole. Therefore, each node should have software versions which are designed to work together. If nodes have mixed untested set of software, there might be unknown compatibility issues. Certain versions of software are developed and tested together and there is better knowledge of their compatibility than for others.

If new node is added to the system, it might have different aged software than rest of the system. It should possible to update the node or the whole system to compatible version level so that the new hardware support would be guaranteed. Even better would be if the updating would be doable on side of installation or regular maintenance.

Updating system node by node manually is time consuming and cumbersome. As a humane lapse may cause problems as described previously, such a task should not be left to be handled by a person. Therefore it would be preferred if the software update delivery would be automated and but the operator could decide on activation.

Therefore**: Deliver compatible software together bundled in single package. and create a centralized update service component that handles updates. The update service distributes the software from the bundle for each node when an update is requested.**

* * *

Start-up Negotiator is enhanced so that it gathers on start-up information on software versions from each node. Start-up Negotiator will function as central clearinghouse for version information gathering. This version list is reported to Update Service which in turn will check if nodes have software from same software bundle. If they have, it informs Start-up Negotiator that everything is ok. Otherwise it informs the operator that update should commence. If the operator accepts, the update service distributes the updates to the nodes and each node updates itself.

During complete system update the nodes in the system are updated one by one. During the update the system should not be operational. A sensible order of updating would be doing in the order of dependencies, if those are known as it would guarantee that at least basic services are compatible if there is some major problem with the update process. If the update for a node fails it will be retried using the retry mechanism provided by the node. If, after couple retries, the update cannot be uploaded, it has to be determined if the node is functional at all. If the node is functional there are three recourses to the situation: get a new node which can be updated, disable the node or try to use other software bundle for the whole system. It might be that there could be multitude of hardware versions and just some do not work with all the software. The service personnel should be notified on failure.

If a node is detected during start-up to have software version which is not from the same software bundle as the rest of the system, it should be updated with software from the software bundle to make it compatible. Software version disparity could be either caused by a node which has been replaced, or by a new node which is added to the system. This requires the update service to contain software for every known device which can appear in to the system. If new hardware is added, it is noticed during the start-up by the Start-up Negotiator. Before the new hardware 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 bundle. In this case, the whole system has to be updated to a newer version of the software bundle where support is available before hardware may be activated.

Sometimes there is a new software bundle available for the whole system. When a service personnel loads the new software bundle from a media to Update Service, the bundle is verified as valid before it is stored. After the bundle has been stored to Update Service the whole system can be updated from single point with the contents of the bundle. There can be multiple bundles available at the same time, so that any software bundle can be used if necessary. When there are multiple bundles available on the update service, the service personnel can choose which bundles should be used for the machine. This is for the convenience of the machine operator as it might be inconvenient time to do a whole system update or the operator might prefer the way the system operates with the older software bundle.

To combine compatible and tested software together, all the software for the system and potentially used devices are combined into single software bundle. Basically software bundle is a compatibility tested snapshot. It has all the software from certain date for all hardware which the system may have. Still, it should be noted that as the software bundle has software for wide range of hardware, it is possible that not all combinations are tested and therefore there might setups which do not work together.

* * *

Now it is possible to update whole system to use software which should be compatible. The whole update process is doable from single point. If newer hardware is available but there is no support for it in older software bundle, the hardware cannot be taken into use unless the software is updated for the whole system. With this limitation older software bundle can always be used instead of older if, for example, newer software has some issues or operator prefers the older software.

* * *

A drilling machine is stopped. A service person insert an USB stick to the cabin computers USB port. The system notices the inserted media, selects the software bundle and verifies it as a valid. After this the bundle is extracted to the update service. The system notifies the service person of the new available system update and queries if it should be installed. After the service person confirms, the system commences with the update. The software is sent from the update service to each node in the system one by one over bandwidth limited CAN bus. After the last device is updated and success is verified, the machine is restarted with updated software.

3 Acknowledgements

I want to thank my colleagues Johannes Koskinen, Marko Leppänen and Veli-Pekka Eloranta for their help. In addition, I want to thank all industrial partners for their valuable cooperation in our pattern mining process: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D. As last but not least, I thank for my better half Ewa, who let me have these writing sessions required, when free time is limited, to write this prose down instead of doing diaper changing duties.