2010 13th International IEEE
Annual Conference on Intelligent Transportation Systems
Madeira Island, Portugal, September 19-22, 2010

Approaches for Increasing Availability of Component-based Traffic Management Software

Christoph Stögerer and Wolfgang Kastner

***Abstract*— For today's traffic management and control systems, availability (defined as the degree to which a system is operable at any given point in time) is of increasing importance. This concerns on one hand hardware and on the other hand software. While in case of a failure in a hardware circuit, a watchdog resets the faulty control-unit, for the monitoring of software, patterns like the Watchdog Pattern, the Safety Executive Pattern or improvements to them might be applied. As these patterns primarily target the monitoring of a single software component running on one node, extensions are necessary that allow supervising complex, multi-threaded components and, thus, fully distributed applications. This paper presents two approaches applicable to the domain of Variable Message Signs (VMS). Both solutions make use of the Web-Based Enterprise Management (WBEM) technology to acquire status information of the underlying multi-threaded components, and to start-up the system whenever necessary. For a proof-of-concept implementation the Windows Management Instrumentation (WMI) has been applied.**

I. Introduction and Motivation

Traffic Control Systems (TCS) [6] are responsible for information propagation from a *Traffic Control Center (TCC)* via a control hierarchy down to the traffic participant. TCS follow a common structure but may differ in terms of (1) numbers of nodes connected, (2) levels of hierarchy involved, and (3) functional complexity. As shown in Fig. 1, at the highest level, a *TCC* collects data from *Sub-Stations (SS)* and provides it to its users for global strategies concerning road traffic monitoring and control. SS are responsible for intermediate data collection for the reasons of (1) controlling specific motorway lines by means of VMS on the basis of collected data, and (2) aggregating data of a specific area and communicating it to the TCC [4]. Apart from collecting traffic data, gathering of environmental data is relevant.

SS are interconnected to one or more *Local Control Units (LCU)* that are in turn wired to *sensors* (e. g. loop detectors, radar detectors) and *actuators* (e. g. gates, traffic lights, VMS). They are responsible for data processing and autonomous control tasks. Several detection-sites may be assigned to an LCU. As depicted in Fig. 1, in our use-case the LCU is integrated into the VMS on the gantry.

An example for the LCU's software architecture was introduced by Kulovits et. al. [7]. This architecture provides a rapid integration of VMS into the general framework of existing traffic control systems and their infrastructure. It consisted of three independent and self-contained components, namely *PLC Service, Controller* and *Communicator* (Fig. 2). All of them communicate using the Distributed Component Object Model (DCOM). The *PLC Service* component is responsible for interfacing with the underlying hardware (a programmable logic controller). The *Controller* component has to manage the communication to a configurable quantity of different sign controllers belonging to one or more VMS. Finally, the *Communicator* component provides the interface to the next level in the traffic management hierarchy.

The minimum system configuration of a TCS can be composed of a autonomously acting *Remote Control Unit (RCU)* controlling a single VMS. The RCU, may be located in a control room or in the gantry distribution cabinet (Fig. 1) in case of bigger systems. It hosts a so-called *Sign Control Application* that provides information about sign status, and identifies errors and defective parts. Furthermore, information about current brightness settings or dimming mode (manual or automatic dimming), service door status or controller temperature can be retrieved. New content can be uploaded to the LCU as well as activated and displayed via a simple user interface. Logging history and errors that occurred during regular communication with the sign are also reported. The maximum system configuration may consist of multiple levels of control and monitoring facilities, where each level is designed for autonomous operation as a kind of fallback in case of breakdown of a higher order level facility.

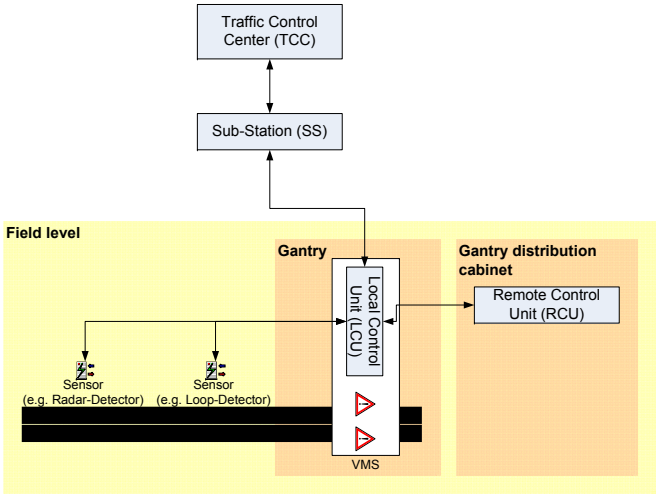


Fig. 1. LCU and RCU in the context of a Traffic Control System

C. Stögerer is Head of Research & Development, SWARCO Futurit GmbH, Mühlgasse 86, 2380 Perchtoldsdorf, Austria stoegerer.futurit@swarco.com

W. Kastner is with the Automation Systems Group, Vienna University of Technology, Treitlstrasse 1/4, 1140 Vienna, Austria k@auto.tuwien.ac.at

978-1-4244-7659-6/10/$26.00 ©2010 IEEE

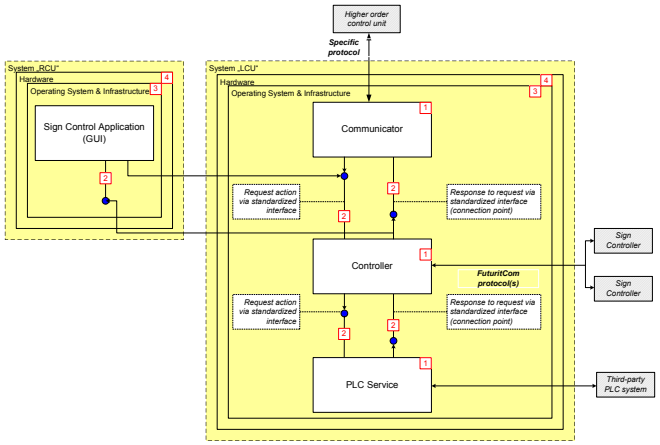


Fig. 2. LCU as an example of a distributed application

As we can see from the example outlined, the system's software components may be distributed over different nodes (e.g. the aforementioned LCU and RCU) depending on the application's specific demands leading to more flexibility.

Today system operators often demand for a high availability of the TCS. For this reason, system vendors need to guarantee a certain availability of the overall system and its components to fulfill the requirements. As the overall availability depends on the availability of each component in the system, monitoring cannot be neglected for the single node LCU software-architecture as well as for the distributed LCU/RCU architecture.

In the remainder of this paper, first, a minimum configuration for enhancing the LCU software architecture with supervising features for a single node architecture and underlying fundamentals are explained. Next, prerequisites for improved monitoring (i. e. support for multi-threading and distribution) are shown. Based on the minimum configuration, extensions for thread monitoring and distributed component monitoring are shown. At the end of the paper, a practical estimation of the different approaches and an outlook are given.

II. Basic Monitoring Instrumentation

When designing traffic management applications, a high degree of reliability and availability is of key concern. As we are faced with a concurrent system with distributed cooperating components, where one process might consist of multiple threads, naive approaches to recover from failure are not adequate. Attempts by individual cooperating processes to achieve backward error recovery can result in the problem which has come to be known as the *domino effect* [8].
As Douglass [3] outlines, functional safety is a system issue and can be addressed through software, hardware, or more usually, a combination of both. If faults occur, they are normally handled in one of the following ways:

- The operation should be retried (feedback fault correction).
- Information redundancy should be built into the data itself to correct the fault (feed-forward error correction).
- The system should be put into a fail-safe state.
- Supervising personnel should be alerted.
- The system should be restarted.

The LCU software architecture realizes the possible types of actions by

1) retrying an operation in case of internal communication error or timeout until a defined number of retries is exceeded,
2) going to a fail-safe state (i. e. the VMS is switched off) in case the operation still fails after a defined number of retries, and finally
3) alerting monitoring personnel when the system is going to shut down by sending an alarm message to the next level of hierarchy.

But still it lacks possibilities to cover the following failures:

- In case of failure of a single component (see 1 in Fig. 2), other components might not be able to detect the cause of error or do not even note that a problem occurred.
- Threads of a single component might crash without being detected.
- In case of failure in the intra-component communication infrastructure (see 2 in Fig. 2), components might not even recognize that there is an error.
- In case of missing system resources (see 3 in Fig. 2), components might crash without a notable reason.
- In case of failure of the underlying hardware (see 4 in Fig. 2), components might crash without any mechanism to recover.

To overcome these drawbacks, the architecture needs to be enhanced. As the solutions mainly targets the goal that the internal computational processing of the components involved is proceeding as expected, the *Watchdog Pattern* [3] was selected as a starting basis for instrumentation. Even though the pattern itself has only low coverage of failure (compared to, for instance, the *Sanity Check Pattern* [3]) it can be easily extended and enhanced to the domain's needs. On the other hand it can be rapidly integrated with the components to be guarded.

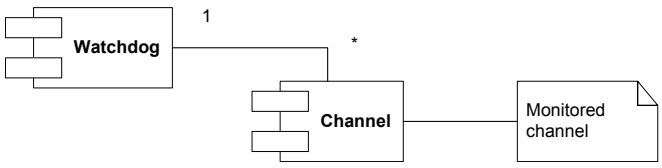


Fig. 3. Watchdog pattern

As shown in Fig. 3, the pattern is built up of several *Channels* (i. e. guarded components) which operate independently of a *Watchdog* (WD). Main task of the channels is to trigger the watchdog accordingly. Thus, the guarded components have to send liveness messages to the watchdog in a predefined time.

Next to continuously checking the liveness of the system, the WD is also responsible for starting-up the systems components in the right order, and shutting down and restarting

the whole software system in case of failure. When the system is restarted, the components take care of putting the system into a fail-safe state before it is available for further requests. In case of a VMS, the fail-safe state is defined by a blanked sign or a ”standard”-aspect to be shown.

During system start-up, the WD parses an XML-based configuration file where the starting order of the guarded processes and further system parameters are defined. Listing 1) shows the XML configuration for the LCU software.

```
<WDConfig>
   <LogConfig file="WDLogCfg.xml"/>
   <Settings ServiceTimeout="15000"
         MaxMemLoad="80"
         MaxPageFileLoad="80"
         MaxPhysicalMemLoad="80"
         MaxVirtualMemLoad="80"/>
   <Services>
         <Service ID="1" Name="PLCService"
                  Monitor="false"/>
         <Service ID="2" Name="Controller"
                  Monitor="true"/>
         <Service ID="3" Name="Communicator"
                  Monitor="true"/>
   </Services>
</WDConfig>
```

Listing 1. XML configuration for the LCU software

The parameter *Settings* incorporates attributes for setting the limits of system resources, such as *MaxMemLoad*, *MaxPageFileLoad*, *MaxPhysicalMemLoad* and *MaxVirtualMemLoad* indicating the maximum allowed memory loads in percent. The parameter *Services* contains information about services to be guarded. Each *Service* element consists of the attributes *ID* (specifying a unique identifier for the starting order), *Name* (the name of the service to be supervised) and *Monitor* which indicates if the service is only going to be controlled (i. e. started and stopped) or also to be monitored. This is necessary as we might be faced with components needed in the system architecture, which for some reason cannot be extended for sending liveness messages to the WD and therefore cannot be included into the monitoring infrastructure (e. g. third party components with no source code available). Data between the WD and guarded components is again exchanged using Microsoft's Distributed Component Object Model (DCOM). For this reason, the watchdog component provides a lean interface.

III. Fundamentals for Improved Monitoring

The instrumentation introduced in the previous section provides basic functionality for supervising the LCU's software architecture (i. e. simple components on a single host). In case of more complex systems where multi-threaded and distributed components are concerned (i. e. the aforementioned LCU/RCU architecture), additional measures are needed.

As outlined in [10], *Web-Based Enterprise Management* (WBEM) is well suited for the domain of traffic management. This is due to the fact that WBEM

1) provides built-in classes for querying performance information about the system, processes and threads,
2) supports dynamic creation and destruction of objects (e. g. for registering threads/components to be monitored),
3) can be extended to the application's specific demands,
4) can be accessed from remote, and
5) is platform independent and hence suitable for heterogeneous distributed systems.

The basic WBEM infrastructure consists of a *WBEM-client* acting as network management client and a *WBEM-server* responsible for managed objects (Fig. 4).

Each WBEM-server comprises of the following main components:

- An *HTTP-server* acting as the communication interface between the WBEM-server and its clients. It forwards the requests to the Common Information Model (CIM) Object Manager for processing.
- A *CIM Object Manager* (CIMOM) is the heart of the WBEM-server. It delegates the received requests to a dedicated Provider based on the information stored in the *CIMOM Repository*.
- The *Provider(s)* (also called WBEM-Instrumentation) can be seen as plug-in module(s) to the CIMOM. They act as interfaces to the specific resource and offer the functionality to translate a WBEM-request into resource-specific commands. Providers are not necessarily software components located on the same node. The standard specification for the interface between the WBEM-server and its providers explicitly allows for a remote connection to the providers [5].

Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows based operating systems. Scripts or applications can be written to automate administrative tasks on remote computers. WMI also supplies management data to other parts of the operating system, for example the System Center Operations Manager, the Microsoft Operations Manager (MOM), or the Windows Remote Management (WinRM). In this way, WMI provides an easy to use framework for monitoring events such as detecting service stops, unavailable threads or system bottlenecks (e. g. memory or disk load). Furthermore

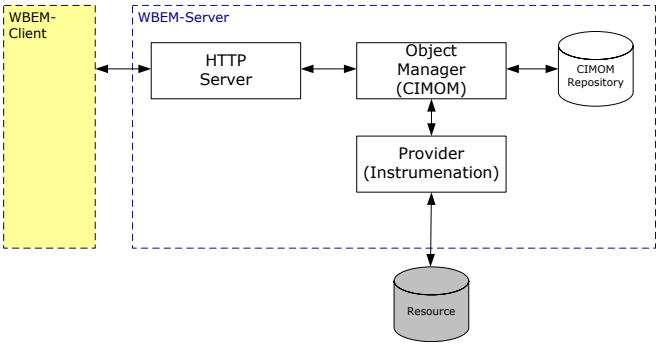


Fig. 4. WBEM infrastructure

WMI also includes capabilities needed to start up the system components via the *Win32_Service*-class, a built-in class of WMI for service control [2].

Summing up, WMI (and WBEM as a technology in common) is a well suited base-technology for extending our watchdog solution. Coupling of WMI to our existing WD solution is depicted in Fig. 5.

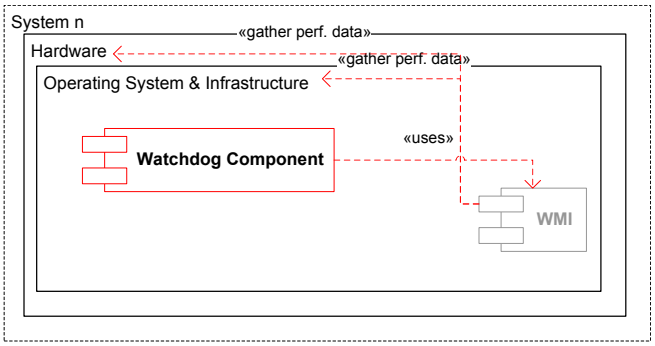


Fig. 5. Watchdog component coupled with WMI

IV. Extensions for Local Thread Monitoring

For monitoring the threads of a component, two different approaches are available. In the *endogenous* approach [9], existing source code is adopted in a way that specific worker-threads (i. e. the threads created by the application's main thread performing a particular – normally time-consuming – operation in the background) are derived from a base-class providing a publisher/subscriber and notification mechanism to the WD. As our solution shall also cover applications where the source code of worker-threads cannot be changed, the endogenous approach is not considered further in this work.

The *exogenous* approach (shown in Fig. 6) on the other hand is targeting the following goals:

- It shall be possible to monitor the worker-threads of a component. This shall be made possible independently how much threads are created or destroyed at runtime.
- Since the components know best about the dynamic behavior of worker-threads' creation and destruction, the *main part* of the component which is controlling (i. e. starting and stopping) the threads shall register and unregister them with the corresponding WD if needed.
- It shall be possible to upgrade existing multi-threaded applications without changing the worker-threads' code.

To fulfil these goals, functionality provided by WMI is used to check the worker-threads' state without changing their code. After the hosting component registered its threads for being guarded, the check is handled through WMI's performance counter class *Win32_PerfFormattedData_PerfProc_Thread* which can be accessed using the *IWbemServices* interface.

Properties provided by this class (e. g. *ThreadState*, *PercentProcessorTime*) allow deriving the current status of a thread or having a look on its dynamic behavior and the WD can react to a critical state accordingly. The state of the LCU's worker-threads is checked once during a watch-interval. Thus, it might take up a full watch-interval for the WD to detect a failure state of a registered worker-thread.

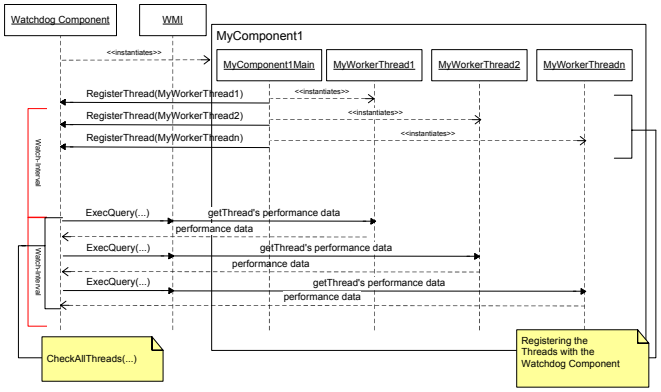


Fig. 6. System start-up with three monitored worker-threads

V. Extensions for Distributed Component Monitoring

Applying WMI as an open framework for interfacing between the WD and distributed guarded components is basically accomplished by creating, destroying and manipulating objects in the CIM repository. The WMI class definition fitting the WD's architectural demands was modeled following the guidelines listed in [5]. It incorporates the component's process ID, a minimum and maximum timespan when the WD must be triggered by the guarded component and a timestamp when the WD was last triggered. Since no adequate component information class definition could be found in the core- or common-model, an extension schema was defined resting upon *Win32_PerfRawData*. *Win32_PerfRawData* is derived from *CIM_StatisticalInformation* which provides a collection of statistical data or metrics applicable to one or more managed system elements. The resulting Managed Object Format (*MOF*) file is shown in Listing 2 below.

```
#pragma namespace("\\\\.\\ROOT\\CIMV2")

[Dynamic, Provider("MyProvider")]

class WD_ComponentInfo : Win32_PerfRawData {
  [key,read,Not_Null] uint32 dwProcessID;
  [read] uint32 dwMinTriggerTime;
  [read] uint32 dwMaxTriggerTime;
  boolean bActivated;
  uint32 dwTriggerTime;
};
```

Listing 2. MOF definition for WD_Component_Info class

Before the class is ready for further use, it must be registered within the CIM repository [1]. After defining the Watchdog class and registering it, a proper way how the client components register and unregister with our Watchdog for being guarded has to be selected. Using WMI, this is accomplished by creating a WMI-class object in the CIM-repository. The WD is registering for an instance creation event for our particular class *WD_ComponentInfo*. When a component

registers with the WD by creating a new *WD_ComponentInfo* object, the watchdog's *IWbemObjectSink::Indicate()* method is called by WMI indicating the object's creation.

In order to allow the WD to control (i. e. start-up and shut-down) components located on a remote machine, the existing configuration is extended by the following parameters:

- The *Host address* or *Host name* where the targeted component resides on.
- A *Username* and *Password* pair for authorization on the target host.
- A *Domain name* for authorization validation (in case a domain-controller is acting as the security master).

As different services might be located on different target nodes, the *Service*-element(s) entry seems to be the reasonable entity for adding the information needed.

```
<WDConfig>
   <LogConfig ... />
   <Settings ... />
   <Services>
       <Service ID="1" Name="Watchdog"
            Monitor="false"
            Server="B"
            Domain="MyWDDomain"
            User="personnel"
            Pwd="xy%982Ax"/>
   </Services>
</WDConfig>
```

Listing 3. Exemplary XML configuration

As shown in Listing 3, the attributes *Server* (specifying a unique identifier by means of address or name of the target machine), *Domain* (specifying the name of the domain acting as the security master) and *User*/*Pwd* which indicate the username and password to be used for access validation on the target node are added to the *Service*-element.

VI. Approaches for Centralized and Decentralized Monitoring

Monitoring a system consisting of distributed components can either be done in a centralized or decentralized way.

In the *centralized* monitoring-approach, the WD acts as a singleton-instance residing on a particular node of the distributed system. Components configured must be started from this central location. All components are sending their liveness indications over the network to the one and only watchdog. Following this approach, all distributed control and monitoring information is passing the network. When communication infrastructure fails, components on remote nodes cannot be put into a fail-safe state. Referring to the requirements for error-handling stated in Section III, the centralized approach must be discarded because a consistent fail-safe state for the distributed system cannot be guaranteed in case of failure in the intra-component communication infrastructure.

In the *decentralized* monitoring-approach shown in Fig. 7, a WD resides on every target node involved. One dedicated WD acts as the so-called "master" whereas all other act as "slaves". Each configured slave WD is remotely started by the master WD. Next, all components are started by their local WD. All control and monitoring information of components on a single node is exchanged locally. The slave WDs then indicate the liveness of the whole (local) subsystem via heartbeat messages passing the network to the master WD. When communication infrastructure fails, components on the remote nodes can be put into a fail-safe state by the local watchdog.

During system start-up, the master as well as the slave WDs register for the creation-event notification in order to be automatically informed by the WMI-instrumentation in case a component puts an instance of the *WD_ComponentInfo* class. Afterwards each WD instantiates its guarded components on the local and/or remote machine. The process of forwarding such heartbeat messages is depicted in Fig. 8.

Depending on the possibilities of instrumenting the target-components (e. g. adoption of source-code), we again differentiate between *exogenous* and *endogenous* instrumentation. With *exogenous instrumentation* we summarize mechanisms that do not require any changes to the existing source-code of the guarded components. In contrast, *endogenous instrumentation* means mechanisms that need to be integrated with the existing source-code of the guarded components.

Endogenous measures are applied by using static WMI-data for liveness indication from the guarded component to its watchdog. As no specific provider implementation is needed in case of static WMI-data, the so-called *intrinsic*-events provided by WMI are used for implementing the mechanisms needed. The intrinsic events provided by WMI are:

- __InstanceCreationEvent
- __InstanceModificationEvent
- __InstanceDeletionEvent

As shown in Fig. 9, the guarded component registers with the WD by creating a *WD_ComponentInfo* instance in the WMI repository. The watchdog on the other hand registers for the *__InstanceCreationEvent* of the *WD_ComponentInfo* class being dynamically informed about the creation of a

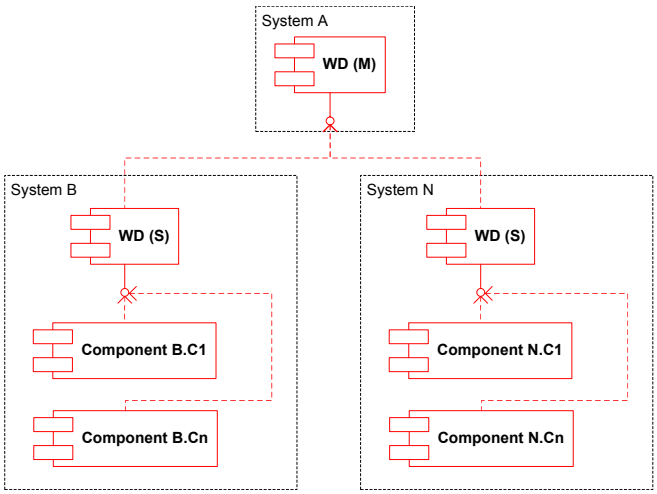
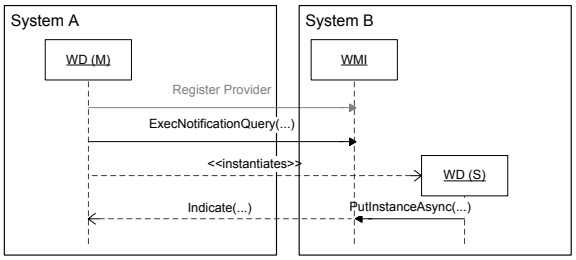
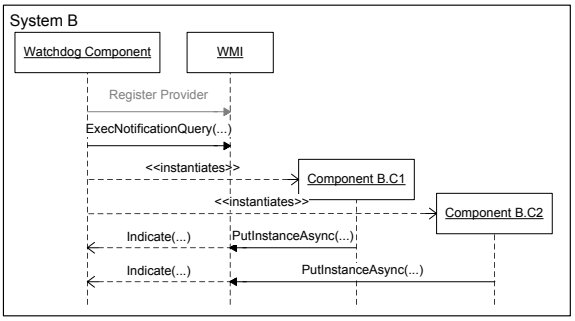


Fig. 7. Decentralized monitoring



(a) Master



(b) Slave

Fig. 8. Starting up the system using WMI

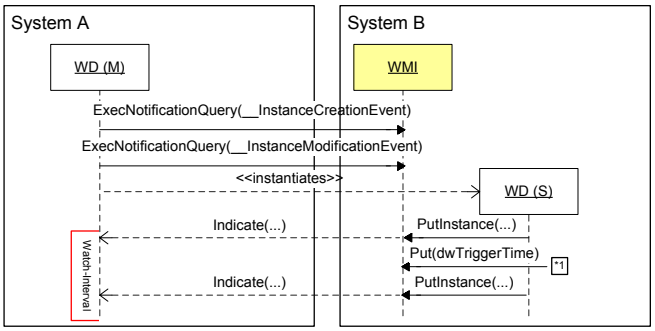
new instance. On instance creation, the WD stores information about the component to be guarded. During operation, the component triggers the WD by incrementing the *dwTriggerTime* property and updating the corresponding *WD_ComponentInfo* instance. Again the WD is informed of the change that occurred inside an instance by registering for an event mechanism provided by WMI called *__InstanceModificationEvent*.

The advantages and disadvantages of this approach can be summarized as follows:

+ Easy to implement (no custom provider implementation needed).
+ Critical system-states can be detected accurately.
− Source code of the component to be guarded must be available.
− Resource consuming.

VII. Conclusion

The paper presented approaches for increasing availability of distributed and multi-threaded applications of the traffic management domain by applying monitoring mechanisms. Introducing the watchdog extension for multi-threaded applications reduces MTTR times by about 80% [9]. After applying basic watchdog-mechanisms and thread monitoring already in-field, we will implement the distributed approach in our application in order to evaluate its resource consumption and scalability. Also the opportunity of adding manageability [10] to our components using the same base-technology will lead to new possibilities in future.



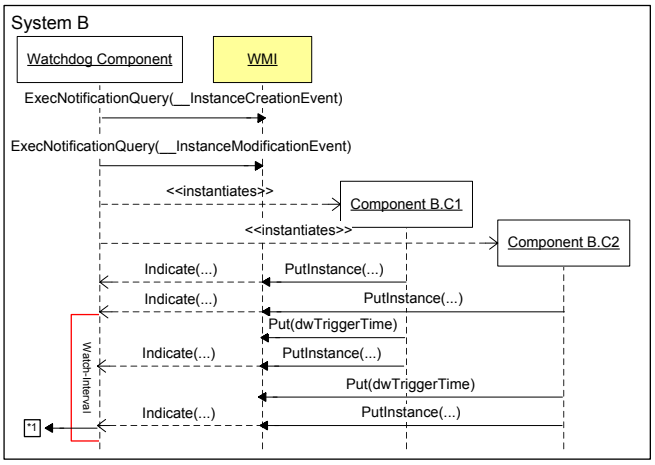


Fig. 9. Non-provider approach

References

[1] Microsoft Corporation. *WMI Administrative Tools*, 2002. http://www.microsoft.com/downloads/details.aspx?familyid=6430F853-1120-48DB-8CC5-F2ABDC3ED314&displaylang=en.

[2] Microsoft Corporation. *Win32_Service Class*, 2009. http://msdn.microsoft.com/en-us/library/aa394418(VS.85).aspx.

[3] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-time Systems*. Addison Wesley, 2003.

[4] Kirschfink H. Collective traffic control on motorways. TUTORIAL on Intelligent Traffic Management Models. Heusch/Boesefeld GmbH, August 1999.

[5] C. Hobbs. *A Practical Approach to WBEM/CIM Management*. Auerbach Publications, 2004.

[6] H. Kirschfink, J. Hernandez, and M. Boero. Intelligent traffic management models. In *Proceedings European Symposium on Intelligent Techniques*, pages 36 – 45, 2000.

[7] H. Kulovits, C. Stögerer, and W. Kastner. System architecture for variable message signs. In *Proceedings 10^{th} IEEE International Conference on Emerging Technologies and Factory Automation*, pages 903 – 909, 2005.

[8] P. A. Lee and T. Anderson. *Dependable Computing and Fault-Tolerant Systems Vol. 3*. Springer, 1990.

[9] C. Stögerer and W. Kastner. Extending the watchdog pattern for multi-threaded windows based traffic management and control applications. In *Proceedings 13^{th} IEEE International Conference on Emerging Technologies and Factory Automation*, pages 854 – 860, 2008.

[10] C. Stögerer and W. Kastner. System management standards for traffic management systems. In *Proceedings 14^{th} IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1 – 8, 2009.