A Synergistic Approach to Monitoring Distributed Software Components

Aleksander Laurentowski, Krzysztof Zieliński
al@ics.agh.edu.pl, kz@ics.agh.edu.pl
Institute of Computer Science, University of Mining & Metallurgy, Kraków, Poland

1 Introduction

Component oriented programming is currently advertised to be the future of enterprise distributed software. With its growing complexity, monitoring of software components appears a natural requirement. The following study has been inspired by our cooperation with an US-based company active in the sector of Computer Telephony and call centers, that placed a research grant at our university. We had a chance to compare the industrial view on software components and distributed monitoring systems with our previous ideas [3, 2] grown on pure academic ground, and to work out a consensus between our knowledge and industrial experts' expectations.

This paper is an attempt to propose a synergistic approach to monitoring distributed software components, i.e. to combine existing distributed monitoring paradigms (like SNMP, ISO CMIS, CORBA Notification Service) into one comprehensive framework. It results in a prototype system providing various monitoring data collection and dissemination schemas, open monitoring data structures, introspection, dynamic registration/deregistration of observers in run-time, configurable transport of event notifications with elements of QoS, and others.

2 Requirements for a Monitoring System's Model

The most common approach to monitoring distributed systems is based on building and analysing a trace of inter-process (inter-object) communication events (remote invocations). However, in industrial practice this approach is often too heavy and not interesting from the application ("business process") point of view, i.e. in many application domains the very monitoring of state of components and objects is more important than monitoring a full sequence of actions leading to a particular overall system's state. This simplifies the architecture and construction of a monitoring system a great deal and eliminates the overhead incurred by tracing interactions between components and building a consistent global snapshot of the monitored system. In consequence, the desired monitoring system could deal with changes of state of monitored components and objects, including object attributes' value changes as well as internal structure of components.

In our system the monitoring information inside a component is organized as a tree of monitored objects, i.e. objects selected to be monitored. Each of these objects may have a collection of monitored attributes, i.e. attributes selected to be monitored. Both attributes and objects generate events on changes of their state. An important part of each monitored object's state are also its relationships with other monitored objects: its parent and children in the tree of monitored objects. Our approach is, in fact, an intentional attempt not to "re-invent the wheel", but to adapt the SNMP and ISO CMIS models to monitoring distributed object software systems. Consequently, we call the tree of monitored objects in a component its Monitoring Information Base (MIB). Such a model provides additionally a simple and clear naming schema, somehow similar to SNMP MIB names.

Another general requirement is related to the choice of a notification model, like e.g. synchronous on-request (pull) or asynchronous callback (push) notification. None of these models is suitable for all purposes, so it seems reasonable to produce both or provide the user with a framework to compose the chosen one with the rest of monitoring system components.

The usage of the system for complex software means that it should be self-descriptive at run-time, which requires support for introspection. To aid ease of use, it should also provide a user-friendly navigation in monitoring data for the client.

However, the basic objective in industrial applications' monitoring is to maintain their performance by means of reducing overhead incurred by the monitoring system wherever possible.

3 Monitoring Infrastructure

To represent the objects under investigation and intercept events concerning those objects, the designed monitoring framework defines two instruments: Monitored Object and Sensor, notion of which is based on the ISO Management Information Model (ISO 10165-1, X.720 standard). These instruments are implemented as classes adding to or wrapping the original application object classes. Each object the programmer wishes to make monitoring-enabled should inherit from the *MonitoredObject* class. We call such object a Monitored Object (MO). Each MO may contain a set of Sensors and a list of "child" Monitored Objects. Sensors represent attributes of monitored objects selected for monitoring, but only those of simple data types. The programmer should replace types in attribute definitions with corresponding Sensor classes, e.g. "IntegerSensor" instead of "int". The Sensor classes implement the same functionality as original data types, provide identical interface as those data types and behave exactly in the same way. They only additionally provide methods to access current value of an attribute and its type and name.

3.1 Component Monitor

An access point for the clients to the component's internal monitoring infrastructure is a Component Monitor (CM). Each monitored component is equipped with a CM, which may provide one or more of the three monitoring interfaces: (1) Introspection, (2) Pull Monitoring and (3) Push Monitoring. The only mandatory interface is Introspection. Pull and Push are optional, i.e the user is free to require one or both interfaces, according to the needs and available network and processing resources. Such approach enables higher flexibility and composition possibilities that the client may employ to customize the monitoring infrastructure and its influence on the system's performance.

3.1.1 CM Introspection Interface

The designed monitoring framework does not assume any particular architecture or type of component – it should handle any newly created component. In order to cope with this challenge, in the absence of a repository describing components (e.g. similar to Interface Repository in CORBA), introduction of introspection mechanisms was a must. The tree-like Monitoring Information Base is the foundation for functioning of this mechanism on run-time. CM Introspection Interface provides the client with introspection of the "static" part of the component, i.e. the part that does not change during execution of the system, e.g. the type of a MO and a list of its attributes (Sensors).

3.1.2 CM Pull Monitoring Interface

CM Pull Monitoring Interface provides monitoring of the dynamically changing part of the component, with which the client can on-request get:

- the current value of an attribute,
- a list of current values of a set of attributes,
- a list of Monitored Object's children,
- a sub-tree of Monitored Objects.

Pull Monitoring enables requesting current values of attributes (wrapped by Sensors) and current snapshots of the MIB tree. The initiator of Pull Monitoring invocations is always the client. One consequence of this is that to notice subsequent changes of attributes' values and tree structure, the client must periodically poll the Pull Monitoring Interface. This is, however, the basic model provided by SNMP (with an exception of a - very limited - mechanism of traps), which proved to be sufficient in a wide range of applications.

3.1.3 CM Push Monitoring Interface

CM Push Monitoring Interface provides the client with asynchronous notification about events taking place in the monitored component. The concept of Push Monitoring is based on the ISO Common Management Information Service - Notification Service (ISO 9595, ISO 9596 standards) and

OMG CORBA Notification Service. Its implementation is in turn based on the well-known "Observer" design pattern. On the contrary to Pull Monitoring, the active role is played by the observed component which notifies interested clients of events that occurred.

With CM Push Monitoring the client can be notified of selected events when they occur:

- *attribute change event* - generated by a Sensor at the change of its attribute's value,
- 2 structure-related events: *add* and *remove*, generated by a MO when a child is being added to or removed from that object,
- *delete* - generated by a Monitored Object when it is being deleted.

Clients express their interests in particular events by means of run-time registration in sources of these events, i.e. in Monitored Objects and Sensors. They can always de-register later on. Some Sensors and Monitored Objects may produce events that are of special importance to several clients. These clients may choose an urgent mode of events' delivery from those Sensors and MOs.

Therefore, client registration is parametrized by:

- the mode (urgent or regular) in which events should be delivered from a given Sensor or Monitored Object to the registering observer,
- the kind of structure change event the observer (client) is interested in (child addition, child removal or both those events) in the case of registration in a Monitored Object.

The client may also choose to receive extra the addition and/or removal events originating from the descendants of the given Monitored Object - these we call propagated events. If a given MO that is being deleted happens to have a subtree of descendant MOs, one *delete* event issued by this object may be sufficient for the client to delete the whole subtree, and thus save on network traffic (a number of *remove* events concerning descendant objects should have been issued otherwise).

Figure 1 depicts the internal structure of a monitored component, with its MIB, CM interfaces, and the notification infrastructure pertinent to each client: two notification queues and a proxy.

The client of our monitoring system is equipped with a GUI similar to the MS Windows Explorer or other tools supporting tree-like directory management.

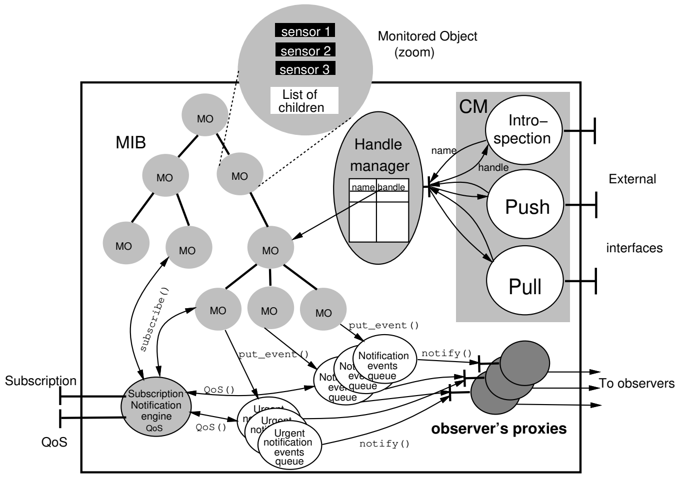

Figure 1: Internal structure of a monitored component

4 Design for Optimal Performance

An important decision at the phase of design of a monitoring system for distributed software components is how the system's functionality should be split between the monitored component's process and other resources (processes) of the underlaying system. This should be carefully designed as a trade-off between performance and ease-of-use. Every monitoring system provides three fundamental functionalities dealing with monitoring data: (1) generation, (2) dissemination, and (3) processing [1]. We argue that only monitoring data generation and dissemination mechanisms should be implemented within the monitored software component. That means any CPU-consuming event processing or filtering should be distributed over separate (possibly remote) process(es), otherwise they could be devastating for the original component's performance.

In our model, the user can only register within selected Sensors or Monitored Objects for a "raw" stream of events concerning a single component. What about complex events, i.e. events involving several attributes, from different objects (possibly even from different components), what about evaluating and matching expressions and filter rules, that are an indispensable part of an effective monitoring system? Our idea is to put all these mechanisms into a separate layer of observers, registered in original monitored components, evaluating complex events and expressions, and subsequently notifying their "private" observers. Clients interested in complex events may therefore register in these first-level observers and get notified when the rules and expressions are satisfied. These do not necessarily have to be end-user clients, but another level of observers evaluating even more complex expressions, and so on. The key idea is that the original application components (processes) are free from additional processing and the direct monitoring overhead is reduced to notification and replying to introspection (rarely) and pull requests, while the event processing is distributed over a hierarchy of observers situated elsewhere.

The second performance issue we focused on was access time to MIB elements (Monitored Objects and Sensors) – searching down their hierarchy on the basis of a path name given as a parameter may be time consuming as it requires parsing the name (a string!) each time the request is serviced. We came up with an idea of handles, which are a kind of safe pointers, used as shortcuts to MIB elements and avoid excessive usage of names in programmer's practice. Handles are created by a Handle Manager object which is a part of Component Monitor. Its role is to provide bi-directional mapping between handles and names.

Another issue are interactions of monitored application's and monitoring infrastructure's threads – these should preferably be separate pools of threads. Activity of application threads related to monitoring should be reduced to a minimum. In our design, all that the application threads have to do is putting event messages to relative queues as soon as events happen. There are two queues for each observer, one per each delivery mode (regular and urgent). The queues are served by the pool of monitoring infrastructure threads, which eventually deliver event notifications to observers.

4.1 Quality of Service Mechanisms

One of the main problems with monitoring is constraining the monitoring data stream, which, as widely accepted, can be done in the most efficient way at its source.

One way of doing this in our system is management of data sources. Monitored Objects and Sensors are equipped in a registry of all interested observers. Each MO and Sensor notifies its clients as long as they are registered. If all observers of a MO or Sensor de-register, it becomes silent. Notification can be also switched off and back on by the monitoring system manager - there are relevant operations in the interfaces of MOs and Sensors.

Another instrument offered in our system is a Quality of Service interface. Inspired by the OMG Notification Service specification, we propose providing the Push monitoring interface with elements of Quality of Service (QoS), to enable tuning the communication channel between the client (observer) and monitored component. Before first registration in any object or Sensor, the observer must create a communication channel with the investigated component. A special object is being created to manage that channel. The observer may then set relevant QoS properties and rules governing delivery of events over the channel to customize the traffic to available resources and possibilities to reduce the costs of remote invocations of notification operations.

Example Quality of Service contracts include: multicasting of selected streams, periodic notification (with settable period), packaging of event notifications (QoS may implement a scheme similar to the Nagle algorithm known from TCP) or an urgent events handling mode for monitoring threads.

Acknowledgements

We would like to thank Andrzej Uszok and Tomasz Haduła who worked with us on the described monitoring system's design.

References

[1] M. Mansouri-Samani and M. Sloman. Monitoring Distributed Systems. In Morris Sloman, editor, *Network and Distributed Systems Management*, chapter 12. Addison-Wesley, 1994.

[2] Jakub Szymaszek, Aleksander Laurentowski, and Krzysztof Zieliński. Instrumentation of CORBA-compliant applications for monitoring purposes. In *Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 129–134, Zinal, Switzerland, March 1997. BROADCAST, Operating Systems Laboratory, Swiss Federal institute of Technology, Lausanne.

[3] Krzysztof Zieliński, Aleksander Laurentowski, Jakub Szymaszek, and Andrzej Uszok. A tool for monitoring heterogeneous distributed object applications. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 11–18, Vancouver, Canada, May 1995. IEEE CS Press.