Architecture for an Automated Management Instrumentation of Component Based Applications

Rainer Hauck

Munich Network Management Team,
University of Munich, Dept. of CS, Oettingenstr. 67, D-80538 Munich, Germany
hauck@informatik.uni-muenchen.de

Due to the ongoing trend towards application service provisioning (ASP), the monitoring of user-oriented quality of service (QoS) parameters (e.g., transaction duration) is gaining more and more momentum. However, most techniques currently used for application management cannot provide this kind of user-oriented information. At present, only instrumentation techniques are suited to provide the required information. On the other hand instrumentation techniques cause considerable efforts due to the need to insert code into an application to be monitored. Therefore they are hardly used today. The architecture proposed in this paper greatly reduces the efforts of management instrumentation by utilizing the component based structure of future applications. Additionally, a means for automated correlation of transactions based on control flows is introduced. Thus, the proposed architecture lays the foundation for a widespread use of application instrumentation.

Keywords: Service Monitoring and Reporting, End-to-end QoS Management, Application Monitoring, Management Instrumentation, Component Based Applications

1 Introduction

In recent years a trend towards service orientation can be observed especially in the area of network and internet connectivity. Tasks outside the main focus of an enterprise have increasingly been outsourced to external service providers instead of accomplishing them internally. However, service orientation is by no means limited to the area of network connectivity and nowadays can also be found in different areas like e.g., business applications. More and more enterprises refrain from running their business applications themselves and rather prefer to outsource them to application service providers (ASP).

Along with the outsourcing of services comes the need to precisely describe not only the functionality of the service to be provided but also the quality of service (QoS) a customer can demand from a service provider. This is usually done in so-called service level agreements (SLAs). Obviously, simply defining QoS parameters is in no way sufficient. Instead, means to monitor the agreed QoS are necessary as well as means to guarantee proper fulfillment.

However, QoS monitoring for application services is still an unsolved problem. Due to the huge differences between existing application services, not even a common understanding about the parameters to monitor has been established yet. Furthermore, only a few tools are available to support the monitoring of ”application-level parameters” and their use is cumbersome and costly.

As application services typically provide transactions that can be triggered by the service user, we believe that the most important requirement to prove proper SLA-fulfillment is the ability to monitor these transactions. If such a user transaction fails, additional means are necessary to quickly identify the root cause of the problem. This can be done by subdividing the user transaction into subtransactions and extending the monitoring to these subtransactions. Of course current application instrumentation techniques like the

Application Response Measurement API (ARM) [ARM98] (see section 2) can provide this kind of information. However, these techniques are hardly used today, because of the huge efforts required to instrument applications.

This paper introduces an architecture that greatly alleviates the task of instrumenting applications. This is accomplished in two ways: On the one hand, the management instrumentation of an application is completely automated based on the component based structure of future applications. On the other hand, the correlation of measurements is completely automated based on the control flows the application is running in. As these are the major drawbacks of current application instrumentation techniques, the proposed architecture lays the foundation for a widespread use of application instrumentation.

In case of component based application development, two roles must be distinguished: **Component developers** implement components that can be used later by **application developers** during assembly of an application. It must be stressed that our approach does not simply shift the efforts from the application developer to the component developer, but substantially reduces the overall efforts of application instrumentation by automation. Thus, the requirements our solution should satisfy are to provide a means to monitor the actual user transactions of an application service as well as their subtransactions. Furthermore, the additional development effort caused by the solution must be reduced to a minimum for both the application developer and the component developer.

The paper is organized as follows: Section 2 shows related work by giving an overview of techniques currently available for application management. The ARM API, which is the technique most closely related to the proposed architecture, is introduced in some more detail. In section 3, our architecture for the automation of management instrumentation is described. After a detailed explanation of our approach in section 3.1, the runtime architecture as well as the development architecture of the solution are introduced. A prototypical implementation of the architecture is shown in section 4. Based on the experiences gained with the prototype, a comparison of the proposed architecture with the ARM API is done in section 5. A short summary and a description of ongoing work conclude the paper in section 6.

2 Related Work

As was already shown in [HR00a] there essentially exist four basic techniques for application management. Two of them, the **monitoring of network traffic** (e.g., [BMR99], [App00]) and the **monitoring of system-level parameters** (e.g., [KS98]) simply cannot deliver the user-oriented information required. The third technique, the **client-side application monitoring** can deliver this kind of information, but only shows the client-view of the application and thus fails to provide detailed information required for a root cause analysis in case of an error. Examples for client-side monitoring are use of *synthetic transactions* like done by *Geyer & Weinig's GW-TEL INFRA-XS* [Wei00] or *GUI-based approaches* like *Candle ETEWatch* [Gro98]. Only the fourth technique, the **application-wide monitoring**, is suited to completely deliver the required information. Application-wide monitoring comprises *application description*, like done by the *Application Management Specification* (AMS) [AMS97] and the *DMTF Application Model* [DMT00], and *application instrumentation*, like the *ARM API* [ARM98]. The application-wide monitoring techniques suffer from high efforts posed on the application developer and thus are likewise not in widespread use today.

A lot of researchers are dealing with the topic of automating application management: [BH97] describes an approach for the automation of fault management of component based applications. However, the approach is constrained to so-called *store-and-forward* architectures and requires great efforts of the component developer because an instrumentation of all components is necessary. [KKC00] in contrast focuses on automatically determining the dependencies of applications and their underlying infrastructure. While this is a promising approach for the area of configuration and fault management, it does not cover user-oriented performance or accounting management. To achieve this, the authors again refer to classical instrumentation techniques. The approach introduced in [HMMT99] suggests generating and mapping events to transactions instead of calling measurement agents. Thus, the transactions to be monitored can be defined more flexible. However, manual insertion of the code generating the events into the source code of the applications is still required.

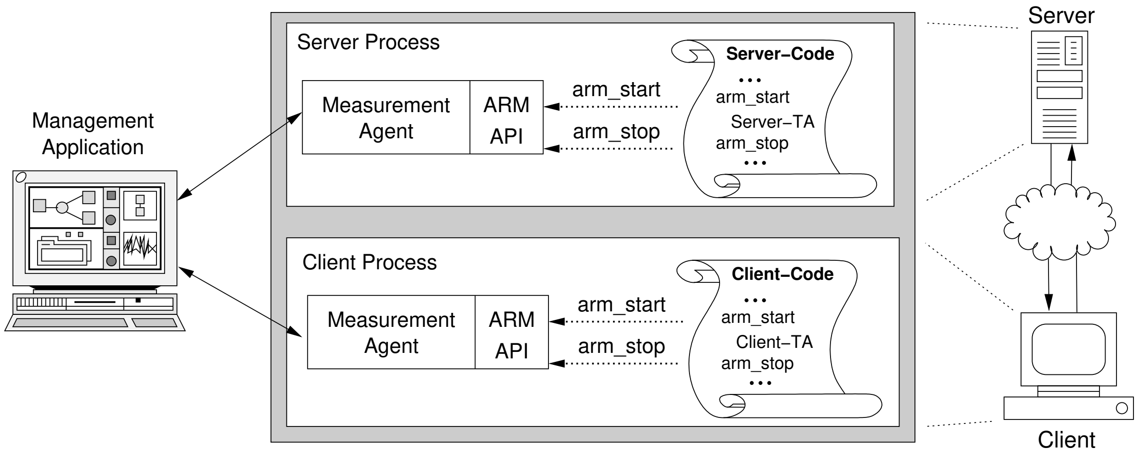

Fig. 1: ARM API: Architecture

As already mentioned, application instrumentation is required to provide the information necessary for service-based application management. The instrumentation approach most widely used today is the ARM API. ARM defines a very simple API that can be called from the application to inform a management agent whenever a transaction begins or ends. Figure 1 shows the usage of the API. An application developer inserts calls to the API into the source code of the (possibly distributed) application. At runtime, a measurement agent integrated in the application's process is called, which determines the time of the call and typically forwards the information to a management application. To achieve correlation of measurements, ARM uses the following mechanism: Whenever an application informs the measurement agent about the start of a transaction, a unique identifier is generated and returned to the application. This parameter has to be passed through the application and back to the measurement agent when the transaction ends. For correlation of subtransactions a second identifier (called a correlator) can be requested and is used analogously. The basic problem of this approach is, that these identifiers must be passed through the entire application as additional parameters for each method invocation. Practical instrumentation experience shows that this mechanism is inconvenient for the developer and can lead to a great number of errors (especially when instrumenting code developed by a third party). Even worse, in case of component based application development, this approach would require an extension of all the methods of all components to allow passing of the respective identifier to the component which cannot be expected in practice.

3 Architecture

The following section describes our proposed architecture for an automated management instrumentation of component based applications. The introduction starts with a detailed description of our approach in section 3.1. The architecture itself is divided in two parts along the lifecycle of an application. The first part covers the development of an application and describes the automated insertion of management code into the source code of the application. The second part covers the runtime of the application and describes the measurement interface and the automated correlation of measurements gained from an instrumented application. For presentation purposes first the runtime part is introduced in section 3.2. Then, the development part is illustrated in section 3.3.

3.1 Approach

The approach introduced in this section overcomes the two major drawbacks of ARM API: First, the problem of placing measurement points in the source code of an application is eliminated by extending the development environments. Thus, placement of measurement points can be completely automated. Second, no manual passing of correlation information is needed as information about the control flows the application is running in is utilized.

3.1.1 Placement of measurement points

The following section describes how measurement points can be inserted into the source code of an application without requiring manual intervention of the application developer. This is done for both the monitoring of user transactions and the monitoring of subtransactions.

Monitoring of user transactions: As already mentioned, the monitoring of user transactions is of great importance. A general model of a user transaction is depicted in figure 2. It starts with a user interaction (UI) initiated by the user (e.g., pressing a GUI button or the `ENTER`-key) followed by some kind of activity. At the end of the user transaction, the result is presented to the user.

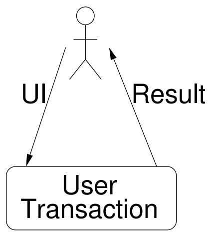

Fig. 2: General model of a user transaction

At development time, the application developer must identify the interactions that start and end the user transactions to be monitored (laid down in the SLA) and must uniquely name these user transactions. As in the case of component based application development no source code access is available for the application developer, this can only take place during component customization. This means, every component providing user interactions must be (manually) instrumented by the component developer to allow this kind of identification. As only a small number of components provide interactions and as GUI components achieve a particular high degree of reuse, this additional effort can easily be tolerated.

Thus, every input component must provide a means to declare each of its user interactions as the beginning of an arbitrary user transaction. To distinguish different user transactions, a means to uniquely name transactions is further required. Whenever such an interaction takes place, the respective component must inform a measurement agent about the beginning of a new instance of a user transaction. Analogously, a way to declare the end of a user transaction is needed. The correlation of the two measurements can be done automatically as described in section 3.1.2.

Monitoring of Subtransactions: While the monitoring of user transactions still requires manual intervention by the application developer, monitoring of subtransactions can completely be automated. In case of component based application development, a call to a single component represents the suitable level of detail to be monitored. Thus, a measurement must take place, whenever a component is called and whenever a call to a component returns.

Particularly in case of component architectures that use development environments to generate adapters to wire the components (e.g., JavaBeans [Jav97]), slightly extending the development environments is sufficient to achieve an automated instrumentation. When creating an adapter, the development environment must insert calls to the measurement agent into the adapter just before calling the target component and immediately after returning from the call. In case of exception-based error handling, catching (and rethrowing) exceptions even allows to distinguish between successful and failed subtransactions. A detailed description of the required extensions for the Java *Beanbox* is given in section 4.1.1.

3.1.2 Correlation of Measurements

As multiple user transactions, either of the same or of different types can be executed in parallel, all measurements must be correlated to their corresponding transaction. Instead of manually passing some kind of generated identifier through the monitored application, the information about the control flows the application is running in can be used to achieve this correlation.

A single control flow can only execute one user transaction at a time. Each user transaction definitely starts in one single control flow (the one, the GUI of the application is running in). Then – depending on the type of the transaction – it might completely be executed in this control flow or new control flows might be added. Typically, only short running transactions will be executed in the control flow of the GUI, because otherwise no more user transactions can be started while the transaction is running (the GUI is "frozen").

If the complete user transaction is executed within a single control flow, a correlation can very easily be achieved: The measurement agent simply determines the identifier of the current control flow at the time the transaction starts. Every subsequent measurement taking place in the same control flow then must belong to the same transaction until the end of the transaction. When control returns to the GUI, the measurement agent must be informed, that the transaction is not running anymore.

However, there are many reasons, why a single control flow is not suited to realize an application service. As already mentioned, this would allow only one user transaction to be executed at a time. Additionally, for performance and distribution reasons or ease of programming new control flows might be started. Figure 3 shows the different ways, user transactions (UTAs) can be executed.

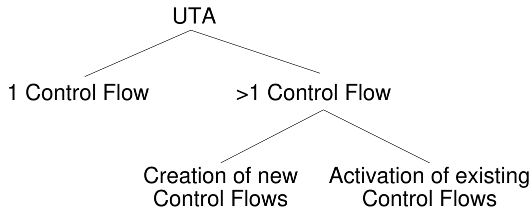

Fig. 3: Different ways of UTA execution

As can be seen from the figure, there are essentially two ways to add a new control flow to the execution of a user transaction: A new control flow could be started or an existing one could be activated. The following paragraphs explain, how a correlation of measurements can be achieved in these two cases.

Creation of new control flows: For the creation of new control flows, some kind of system mechanism is used. Usually there exists a library that shields the details from the developers. By instrumenting this library to inform the measurement agent whenever a new control flow is started, the correlation can be achieved. As the call to the library is executed within the initiating control flow, the library has knowledge about both the initiating and the created control flow. If it provides this information to a measurement agent, it can correlate the new control flow to the transaction the already existing control flow was executing.

Activation of existing control flows: The situation is slightly different in case of activating existing control flows. Essentially, there are two ways to achieve this: Either, a system mechanism can be used to activate the control flow (e.g., in case of remote communications) or a job is placed in some kind of queue that is regularly checked by a component executing in a different control flow.

When using a system mechanism for communication, an automated correlation can be achieved as follows: The communication mechanism must be instrumented to inform a measurement agent about the newly added control flow. In case of remote communications, a unique identifier for the invocation must be created and transmitted to both the local and the remote measurement agent thus again allowing correlation. This should be done transparently to both the application developer and the component developer.

When using a queue for communication, a correlation can only be assured if the component implementing the queue has been instrumented by the component developer to analogously provide the required correlation information.

3.2 Runtime architecture

This section introduces the runtime architecture of the proposed solution. After a brief overview of the components, the measurement interface is described.

The runtime architecture is depicted in figure 4. Essentially it consists of a – possibly distributed – instrumented application, a measurement agent per process (called a measurement correlator), instrumented libraries of the execution environment as well as management agents and management applications.

Instrumented application: The instrumented application calls the corresponding measurement correlator whenever a transaction (either user transaction or subtransaction) starts or stops. For this purpose, the measurement interface described in the next section is used. The automated development of an instrumented application is explained in section 3.3.

Instrumented libraries: To allow monitoring of user transactions executed in more than one control flow, an instrumentation of the appropriate libraries is necessary. This means, the libraries responsible for starting and stopping control flows as well as the libraries for the activation of existing control flows must be instrumented to provide the required information to the measurement correlator.

Measurement correlator: The measurement correlator is responsible for correlating measurement to user transactions. It is called from both the instrumented application and the instrumented libraries via the measurement interface. To avoid large amounts of interprocess communication, the measurement correlator is executed in the same process the application to be monitored is running in. The gathered information is regularly forwarded to the system's management agent.

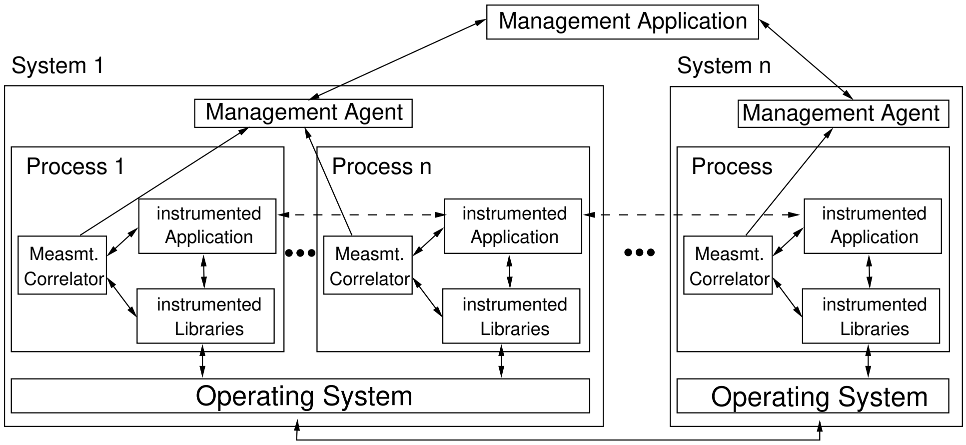

Fig. 4: Runtime architecture

Management agent and management applications: The information gathered by the measurement correlator is regularly forwarded to a management agent running on the same system. The management agent runs outside the process of the measurement correlator (and thus of the application to be monitored). The management agent forwards the information to all registered management applications.

The management applications collect the information of the various management agents, correlate information in case of distributed applications and can do any kind of further processing.

3.2.1 Measurement interface

The measurement interface is the interface provided by the measurement correlator which is called by both the instrumented application and the instrumented libraries. Figure 5 shows the specification of this interface (where UTA and STA mean user transaction and subtransaction resp.). As can be seen from the figure, no explicit correlator (like necessary when using ARM API) must be passed as parameter. Instead, correlation of measurements is completely automated based on the approach described in section 3.1.2. The following paragraphs briefly explain, how the measurement interface is expected to be used:

In any case, measurement of a user transaction starts with a call to `startUTA`. If the complete user transaction is executed within a single control flow, any number of `startSTA`/`stopSTA`-pairs might follow (to measure individual subtransactions). At an arbitrary point during execution, a call to `stopUTA` informs the measurement correlator that the result of the transaction has been presented to the user.

If additional control flows are started during execution, the measurement correlator is informed via `addControlFlow`. Analogously, a control flow leaving execution of a transaction causes a call to `removeControlFlow`.

<<interface>> **Measurement Interface**
startUTA(utaName: String, userName: String, componentName: String, componentID: OID) *stopUTA(status: String, info: String)* *startSTA(componentName: String, componentID: OID, parentTA: UUID=0)* *stopSTA(status: String, info: String)* *addControlFlow(newControlFlow: ThreadID, target: OID)* *removeControlFlow(status: String, reason: String)* *initiatedTA(isAsynchronous: Boolean):UUID* *logInfo(info: String)*

Fig. 5: Specification of Measurement Interface

The most complicated case is activation of existing control flows. As an example, figure 6 shows a sequence diagram of a remote procedure call (RPC). The local application uses a communications mechanism to transparently call the remote server. The communications mechanism issues an `initiatedTA`-call to receive a unique identifier for the current invocation from the local measurement correlator. This identifier is then transparently transmitted to the remote system and given to the remote measurement correlator as a parameter in a `startSTA`-call. Then the remote server is activated. While executing, the remote server again uses `startSTA/stopSTA`-calls (to the remote measurement correlator) in order to provide detailed information about its further subtransactions. Before returning to the calling system, the communications mechanism issues a `stopSTA`-call to the remote measurement correlator to inform it about the end of the subtransaction. Using the exchanged identifier, management applications later can easily correlate the remotely executed subtransactions to the appropriate user transaction.

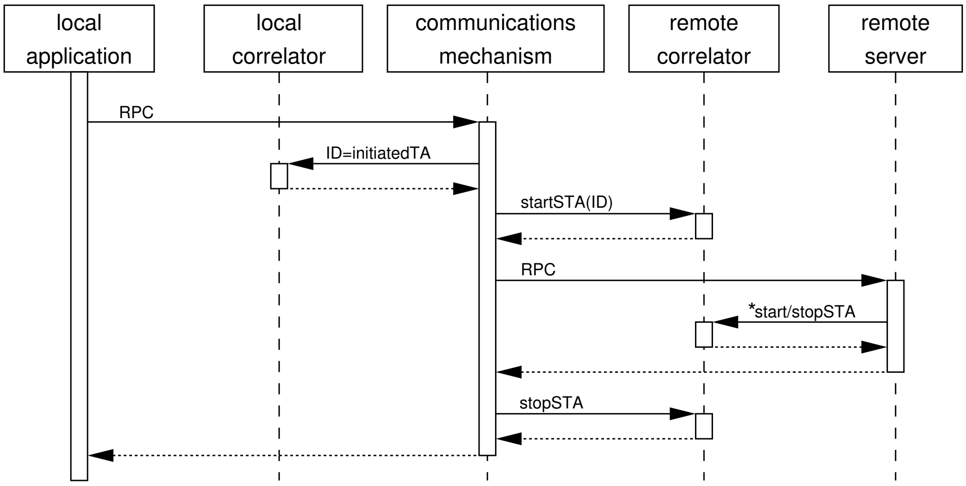

Fig. 6: Example: Sequence Diagram of a remote procedure call

3.3 Development architecture

The instrumented applications mentioned in the previously described architecture can easily be generated by using the development architecture described in the following. It essentially consists of components and an extended development environment which cares for the automated instrumentation. The following paragraphs briefly explain the elements of the architecture.

Components: Most of the components can be used without any change. However, two classes of components, GUI-components and so-called active components, require special instrumentation.

For every user interaction provided by a GUI component, an instrumentation is required (by the component developer) that allows the application developer to specify whether this user interaction serves as the start or stop of a user transaction of the developed application. In addition, the application developer needs a means to uniquely (within the application) name the user transaction to be started. In our approach this is done during component customization, which essentially means configuring a component's parameters (e.g., via the GUI of the development environment) during application development.

Similarly, so-called *active components* require special treatment. By *active components* we mean components which have a queue and a control flow of their own. These components require an `initiatedTA`-call whenever a task is placed into the queue as well as `startSTA`- and `stopSTA`-calls respectively surrounding the processing of the task.

Extended development environment: The most important part of the development architecture is the extended development environment. It is used to build applications from prefabricated components. Our architecture builds on component architectures wiring components by generating adapters. By slightly

```
public class ___Hookup_16e8ce750f implements java.awt.event.
     ActionListener, java.io.Serializable {
...
    public void actionPerformed(java.awt.event.ActionEvent arg0) {
       Measure.startSTA(target.getClass().getName(),String.valueOf(
            System.identityHashCode(target)),"startSort");
       try {
            target.startSort();
       }
       catch (Error e){
            Measure.stopSTA("Failed",e.getMessage());
            throw e;
       }
       catch (RuntimeException e){
            Measure.stopSTA("Failed",e.getMessage());
            throw e;
       }
       Measure.stopSTA("Success",null);
    }
...
```

Fig. 7: Adapter generated by extended BeanBox

extending the development environment, calls to the measurement correlator can be inserted automatically into these adapters just before the target component is called and immediately after the call to the target component returns. By catching exceptions possibly thrown by the target component, even an automated distinction between successful and failed calls can be done.

4 Implementation of the architecture

To allow a thorough evaluation of the proposed architecture, a prototypical implementation has been done. The implementation is based on *JavaBeans* [Jav97] and uses the *Java BeanBox* [Sun99] as a development environment. Again the description is divided in two parts, one focusing on the implementation of the development architecture and the other one focusing on the implementation of the runtime architecture. The following sections give an overview of both parts of the prototype.

4.1 Implementation of development architecture

The implementation of the development architecture essentially consists of an extension of the *Java BeanBox*, a simple and freely available development environment for *JavaBeans*-based applications. Additionally, a number of components have been instrumented.

4.1.1 Extension of the Java BeanBox

The instrumentation of two classes of components does not require any extensions of the *BeanBox*. The configuration of these components can transparently take place during component customization. Only for the insertion of `startSTA`- and `stopSTA`-calls into the generated adapters some minor extensions are necessary.

The *BeanBox* generates adapters by writing into a file using calls to the `print`-method. Thus, the extension could easily be achieved by adding further calls. An example for an instrumented adapter automatically generated by the extended BeanBox can be found in figure 7. To distinguish the additionally generated code, the ”normal” code is depicted in grey. Details about the performed extensions can be found in [Hau01].

As can be seen from the figure, a call to `startSTA` has been inserted just before the method `startSort` of the target component is called. The call to the target component is now done within a `try/catch`-environment to catch possible exceptions thrown by the target. Depending on the success of the call, a `stopSTA`-call is done with the appropriate status parameter.

```
public void mouseReleased(MouseEvent evt) {
   if (!isEnabled()) {
       return;
   }
   if (down) {
      Measure.startUTA(utaName, System.getProperties().getProperty(
                "user.name"), "StartButton", String.valueOf(
                System.identityHashCode(this)));
      try{
            fireAction();
      }
      catch (Exception e){
           Measure.removeControlFlow("Failure", "Thread returned to GUI");
           throw e;
      }
      Measure.removeControlFlow("Success", "Thread returned to GUI");
      down = false;
    }
```

Fig. 8: Instrumentation of a GUI-Button

4.1.2 Instrumentation of components

As an example, the instrumentation of a simple GUI-button is shown in figure 8 (again showing the original code of the button in grey). This button can be used to trigger arbitrary user transactions. When the user releases the button, the method `fireAction` is called which essentially calls all registered event listeners.

Immediately before `fireAction` is called, a `startUTA`-call to the measurement correlator is inserted. As a parameter, this call provides the name of the user transaction (configured by the application developer during customization). When `fireAction` returns, `removeControlFlow` is called to inform the measurement correlator that the current control flow no longer executes this instance of a user transaction (This must not be confused with a call to `stopUTA` which indicates that the result of the user transaction has been presented to the user).

4.2 Implementation of runtime architecture

In order to prototypically implement the runtime architecture, an implementation of the measurement correlator, an instrumentation of some *Java Virtual Machine (JVM)* libraries as well as an implementation of a management agent and management application has been done. The following sections briefly introduce these components of the architecture.

4.2.1 Measurement correlator

A measurement correlator that provides the measurement interface described in section 3.2.1 was implemented in *Java*. To allow easy access to the measurement correlator from both the instrumented application and the instrumented libraries without having to pass a reference, an additional class was implemented, providing the measurement interface using static methods.

The measurement correlator uses a hash table to map *Java* threads to instances of user transactions. Every time the measurement correlator is called, the appropriate user transaction can be determined from the hash table using the thread of the caller. Current system time is determined and stored in combination with further information in a vector. Regularly, a low priority thread is activated, sending the information to the management agent.

4.2.2 Instrumentation of JVM Libraries

The proposed architecture requires the identification of every start and stop of a thread. To allow correlation of distributed transactions, the communication mechanisms must be instrumented as well. However, in our protoype, no instrumentation of inter-process communication has been done yet (it is planned to do so in future releases).

For identification of start and stop of threads, an instrumentation of the class `java.lang.Thread` was done. We used the freely available source code edition [Sun01] of the JVM for this purpose. Essentially, the method `start` of `java.lang.Thread` was instrumented to call `addControlFlow` whenever a new thread is started. As `start` is executed in the initiating thread, a correlation can easily be achieved.

Similarly, the method `exit` from `java.lang.Thread`, which is executed by the system whenever a thread ends, was instrumented to inform the measurement correlator via a call to `removeControlFlow`.

4.2.3 *Implementation of management agent and application*

A management agent was built, which receives the information from the measurement correlator. It stores the information and forwards it to arbitrary management applications. To achieve independence from the underlying platform and programming language, the information is coded in XML.

Our management application gets the information from the management agent and visualizes it. Figure 9 shows a screenshot. It illustrates an instance of an exemplary user transaction (called `QuickSortUTA`) that simply does a delay of six seconds and then starts sorting an array.

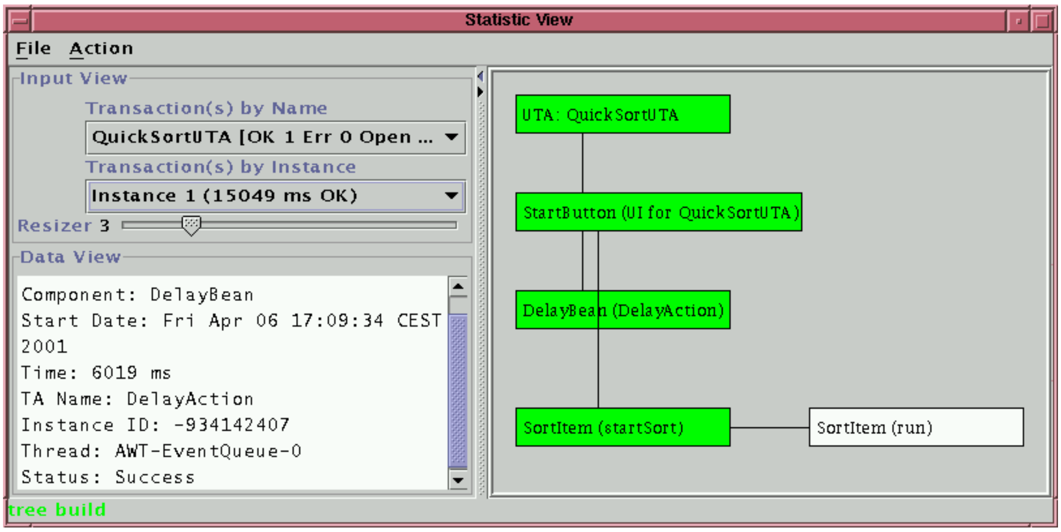

Fig. 9: Prototypical management application for the visualization of transaction instances

Using the fields ”Transaction(s) by Name” and ”Transaction(s) by Instance”, an instance of a user transaction to be visualized can be chosen. The right side of the window shows the transaction instance as well as all of its subtransactions. As can be seen from the figure, the `QuickSortUTA` starts with a user interaction in the start button, followed by sequential calls (visualized by the two vertical lines) to a `DelayBean` and a bean called `SortItem`. The `SortItem` forks a new thread (visualized by the horizontal line) that actually does the sort. By different colors, the status of each of the transactions is illustrated.

The lower left part of the windows shows detailed information about subtransactions that can be requested by clicking on the appropriate rectangle. In the example, information about the `DelayAction` executed by the `DelayBean` is shown. Both the start time and the duration of the execution can be seen. Further information concerning the component, the name of the transaction or the status of the transaction is likewise available.

5 Evaluation

Based on the experiences gathered with the prototype and on experiences gathered with the ARM API in former projects (see e.g., [HR00b]), the following paragraphs give a comparison of our approach in contrast to ARM:

Available information: Our experiences show that both approaches can provide similar information. In contrast to the ARM API, our approach provides a platform- and implementation-independent representation of the information available to arbitrary management systems, while ARM offers no standard mechanism for management systems to retrieve the management information from the measurement agent.

Instrumentation effort: The ARM API does not provide any kind of tool support to provide automation. It simply offers an API and leaves it up to the application developer to do the entire instrumentation manually. Not even a methodology is given to aid the application developer in finding the places suitable for instrumentation. Thus the development effort is high and impedes widespread use of this promising technique.

Our approach in contrast delivers a high degree of automation. The management instrumentation of an application is done almost automatically. Only the user interactions that start and end user transactions have to be declared by the application developer. This is done during customization and does not require the application developer to insert any kind of code into the application. The instrumentation of subtransactions is automated entirely.

The additional effort posed on component developers can easily be tolerated, as only two classes of components require special instrumentation and a high degree of reuse can be expected for these components (especially for GUI components).

Correlation of measurements: ARM uses a costly and cumbersome technique to achieve correlation of measurements. The application developer must care for passing identifiers of transaction instances throughout the application. Thus, correlation of subtransactions is hardly ever used today and instrumentation of component based applications is almost impossible.

Our approach completely automates correlation of measurements. By using the information about the control flows the application is executing in, no care at all must be taken by the developers about how the measurements correlate with each other. As the correlation of subtransactions thus requires no additional effort, by far more detailed measurements can be expected.

6 Conclusion and future work

The paper proposed an architecture for automation of management instrumentation of component based applications. Based on the experience that traditional techniques for application monitoring cannot deliver the information needed for service-oriented application management and that current instrumentation techniques require too much effort by far, a new approach was introduced.

This architecture greatly reduces the instrumentation effort for both the application developer and the component developer by automatically placing calls to the measurement correlator into the application. Therefore an extension of the development environment was necessary. A second major drawback of current instrumentation techniques, the cumbersome correlation of measurements, was overcome by utilizing the information about the control flows the application is executing in.

Our current work focuses on extending the solution to component architectures that do not use adapters for the wiring of the components. Especially *Enterprise Java Beans* [EJB00] and *Corba Component Model* [OMG99] will be taken into account. As in these architectures every call is intercepted by the container, an instrumentation of the container provides the information required. Also, the control flow based correlation of measurements will be examined for use in non-component environments. A further focus of our current work is to develop management applications that allow a user-based mapping from component failures to the probability of service outages based on statistical information about past component usage.

Acknowledgment

The author wishes to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of the paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of

Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its webserver is located at `http://wwwmnmteam.informatik.uni-muenchen.de`.

References

[AMS97] Application Management Specification. Version 2.0, Tivoli Systems, 1997.

[App00] Apptitude. MeterFlow Network Decision Data Engine. Technical White Paper, January 2000. http://www.apptitude.com/pdfs/mfwhitepaper.pdf.

[ARM98] Application Response Measurement (ARM) API . Technical Standard C807, The Open Group, July 1998.

[BH97] V. Biaggiolini and J. Harms. Toward Automatic, Run-Time Fault Management for Component-Based Applications. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)*, number 5 in TUCS General Publication, pages 5–12, Jyväskylä, Finland, September 1997. Turku Centre for Computer Science.

[BMR99] N. Brownlee, C. Mills, and G. Ruth. RFC 2722: Traffic flow measurement: Architecture. RFC, IETF, October 1999.

[DMT00] DMTF Application Working Group. Application MOF Specification 2.5. CIM Schema CIM_Application25.mof, Distributed Management Task Force, December 2000.

[EJB00] Enterprise JavaBeans Specification Version, Version 2.0 – Proposed Final Draft. Specification, Sun Microsystems, October 2000.

[Gro98] Hurwitz Group. Candle captures the "user experience". Technical White Paper, September 1998.

[HAN99] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[Hau01] R. Hauck. *Architektur für die Automation der Managementinstrumentierung bausteinbasierter Anwendungen*. Dissertation, Ludwig-Maximilians-Universität München, 2001. To be published.

[HMMT99] J.L. Hellerstein, M.M. Maccabee, W.N. Mills III, and J. Turek. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 152–162, Austin, TX, USA, June 1999. IEEE Computer Society.

[HR00a] R. Hauck and I. Radisic. Monitoring Application Service Performance – Classification and Analysis of Existing Approaches. In *Workshop of the OpenView University Association (OVUA 2000), Santorini, Greece*, June 2000.

[HR00b] R. Hauck and H. Reiser. Monitoring Quality of Service across Organizational Boundaries. In C. Linnhoff-Popien and H.-G. Hegering, editors, *Trends in Distributed Systems: Torwards a Universal Service Market. Proceedings of the third International IFIP/GI Working Conference, USM 2000*, number 1890 in Lecture Notes in Computer Science (LNCS), Munich, Germany, September 2000. Springer.

[Jav97] Javabeans. Specification, Sun Microsystems, July 1997.

[KKC00] G. Kar, A. Keller, and S. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. In J. W. Hong and R. Weihmayer, editors, *NOMS 2000 IEEE/IFIP Network Operations and Managment Symposium — The Networked Planet: Management Beyond 2000*, pages 61–74, Honolulu, Hawaii, USA, April 2000. IEEE.

[KS98] C. Krupczak and J. Saperia. RFC 2287: Definitions of system-level managed objects for applications. RFC, IETF, February 1998.

[OMG99] Component Spec - Volume I. TC Document orbos/99-07-01, Object Management Group, July 1999.

[Sun99] Sun Microsystems, Inc. Java Beans – Downloading the BDK 1.1, April 1999. http://java.sun.com/products/javabeans/software/bdk_download.html.

[Sun01] Sun Microsystems, Inc. Welcome to the Java Community Source Developers' Area, 2001. http://developer.java.sun.com/developer/products/java2cs/index.html.

[Wei00] Geyer & Weinig. Portofolio – INFRA-XS, 2000. http://www.gwtel.de/.