Monitoring of Distributed Component Interactions

Nikolay K. Diakov
CTIT
P.O. Box 217, 7500 AE
Enschede
The Netherlands
Tel.: +31-53-4893747
Fax: +31-53-4894524
diakov@ctit.utwente.nl

Harold J. Batteram
Lucent Technologies
P.O. Box 18, 1270 AA
Huizen
The Netherlands
Tel.: +31-35-6871378
Fax: +31-35-6875954
batteram@lucent.com

Hans Zandbelt
Telematica Instituut
P.O. Box 589, 7500 AE
Enschede
The Netherlands
Tel.: +31-53-4850445
Fax: +31-53-4850400
zandbelt@telin.nl

Marten J. van Sinderen
CTIT
P.O. Box 217, 7500 AE
Enschede
The Netherlands
Tel.: +31-53-4893677
Fax: +31-53-4894524
sinderen@ctit.utwente.nl

1. Background

Distributed applications are becoming one of the major types of software used nowadays. At the same time, the fast-paced software market demands for rapid development of distributed applications. This leads to reshaping of the software development methodology towards the usage of off-the-shelf components for quick assembly of applications of arbitrary complexity.
Unfortunately, ability for quick manufacturing of software using assembly and configuration of available components does not guarantee correctness of the solution nor quality of the product.
One way to enhance quality is through thorough testing. However, testing of applications that run in a distributed environment is not an easy task. Distributed computing environments usually require several physical machines with different hardware configurations, having installed different operating systems and middleware software, and different characteristics of the network connections between them. Thus, testing in distributed environments can be very different from the single-computer case (see [2]).
In order to contribute to solutions in this area, we set ourselves the following goals:

1. Provide a basic *monitoring* framework for dynamic analysis of distributed component applications, enabling tracing of flows of control through the system as it executes in a normal mode.
2. Integrate the monitoring into the development process, so that the developer is not burdened with monitoring issues.
3. Prove that the approach is realistic by implementing a prototype and suggest particular technology solutions.

2. Problem analysis

Our major assumption is that the targeted family of applications is built from units of distribution called components [1] that communicate with each other through well-defined interfaces. When we started this work, we had a distributed software component framework [6] available, with component model (similar to the CORBA component model [5]), allowing design and development of distributed components.
As a first approach we consider components as a black-box, and observable events happening to a component to be its *lifecycle* events (create, destroy, suspend, resume, etc) and its *interaction* events. This set of events forms the information model of the investigated monitoring scheme.
Components interact with other components through the CORBA DPE in a synchronous and asynchronous manner. In CORBA terms, an interaction is a procedure call to a CORBA object implementing an IDL interface. An invocation consists of transmission of request objects encapsulating operation names, parameters, results and other information.
A flow of control is a sequence of component interactions that share common *context*. In order to construct a flow of control two separate problems have to be solved:

- Sending *context* from one component to another, in a generic way;
- Propagating *context* through the custom component implementation, in a generic way;

3. Use of reflective technology

The demand for generality naturally brought our attention to reflective technology. Reflective technology would allow us to isolate the monitoring specific code into separate facilities and libraries, that will leave component

developers free of concerns about the monitoring during design and implementation phases. In our approach we investigated three technologies:

- CORBA Interceptors;
- Reflection on the thread model through Java;
- CORBA Portable Object Adapter (POA);

CORBA provides the *interceptor* mechanism that reflects on the invocation model. Our monitoring scheme uses interceptors(message and process level) that provide low-level access to the CORBA request/reply.
The Java 2 platform offers a simple API for reflection on the thread model [7] that allows propagation of per-thread *context* through a Java application in a generic way.
POA standardises the way CORBA IDL interface implementations are being executed, which is essential to a reflective approach.

4. Solutions

At some point before the actual execution of a CORBA request (or reply) we need to insert the monitoring *context* into the message. This information is essential to the ordering of events and the tracking of control flows. In a similar way we need to analyse the monitoring *context* that was propagated back from serving to calling object at the return of the request.
In order to achieve our goals in tracking the flow of control we need a generic way for 'peeking' into the black box of the component implementation at run-time. We use this generic scheme for passing *context* information that will help reconstructing causal relations between component interactions.
We rely on the CORBA Portable Object Adapter (POA) while assuming consistency of the execution of interface implementations, e.g. no collocated call optimisations for components sharing the same ORB instance. Furthermore, we need to be able to tag threads of execution that are currently processing the code of a particular component, with information about the request. Typical example of thread dependency is the '*forks*' in the execution of the current Thread. We track '*forks*' using the Java 2 API for passing the thread context between the child and parent threads. Another example of context exchange between threads is a synchronisation point. The later is not yet supported in our solution because Java does not provide reflective interfaces for managing its synchronisation scheme.

5. Current prototype status

The latest release of the prototype includes fully functional component monitor that delivers a comprehensive visual representation of component interaction events in form of message sequence charts. This format has been specially extended for depicting run-time events. Interactive interface offers the tester option to assign trail labels to component interaction points thus *labelling* the flow of control passing through this interaction point.

6. Conclusions

We have described a generic monitoring approach that can be used to enhance the quality of distributed component software. The generality of the approach is achieved through using reflective technology, i.e. CORBA Interceptors, CORBA POA and Java 2 features.
A drawback of using interceptors is the fact that their interfaces have not been standardised so far. However, the process of standardisation is ongoing and has recently resulted in an OMG Joint Revised Submission called Portable Interceptors [4].
Although the Java 2 API enables discovery of '*forks*' in the execution, we find it lacking functionality for the purpose of discovering synchronisation points (e.g., '*joins*') within multithreaded components.

Acknowledgements

This work has been partially supported by the research projects FRIENDS and AMIDST [3].

References

1. Szypersrki, C., *Component Software: Beyond Object Oriented Programming*, Addison-Wesley, 1998.
2. Krawczyk, H., Wiszniewski, B., "Analysis and Testing of Distributed Software Applications", Research Studies Press Ltd., Baldock, Hertfordshire, England, 1998
3. The AMIDST web site: http://amidst.ctit.utwente.nl/
4. Portable Interceptors Specification (orbos/99-12-02), http://www.omg.org/
5. CORBA Components resource page: http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html
6. Bakker, J.L., and H.J. Batteram, "Design and evaluation of the Distributed Software Component Framework for Distributed Communication Architectures", Proceedings of the 2nd Intl. Workshop on *Enterprise Distributed Object Computing* (EDOC'98), Nov. 1998, pp. 282–288.
7. http://java.sun.com/products//jdk/1.2/docs/api/java/lang/InheritableThreadLocal.html