

Design and Implementation of a Framework for Monitoring Distributed Component Interactions

Nikolay K. Diakov¹, Harold J. Batteram², Hans Zandbelt³, Marten J. van Sinderen¹

¹ CTIT, P.O. Box 217, 7500 AE, Enschede, The Netherlands
{diakov, sinderen}@ctit.utwente.nl

² Lucent Technologies, P.O. Box 18, 1270 AA, Huizen, The Netherlands
batteram@lucent.com

³ Telematica Instituut, P.O. Box 589, 7500 AE, Enschede, The Netherlands
zandbelt@telin.nl

Abstract. This paper presents a framework for monitoring component interactions. It is part of a larger component framework built on top of the CORBA distributed processing environment that supports development and testing of distributed software applications. The proposed framework considers an OMG IDL specification as a contract for distributed interactions and allows precise monitoring of interaction activities between application components. The developer is not burdened with monitoring issues because all the necessary code instrumentation is done automatically. The tester is given the opportunity to use monitoring facilities for observing interactions between distributed component applications. This paper explains the monitoring framework and reasons about its expressive power, accuracy and applicability. The approach is validated in a platform for design, development and deployment of on-line services.

1. Introduction

Distributed multimedia 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 multimedia 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, the ability to quickly manufacture software through 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 environments usually consist of 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 introduces additional aspects to the single-computer case. This paper presents a framework for monitoring component interactions. It is part of a larger component framework that is built on top of the CORBA distributed processing environment, and that supports development and testing of distributed software applications. The proposed framework considers an OMG IDL specification as a contract

for distributed interactions and allows precise monitoring of interaction activities between application components. The developer is not burdened with monitoring issues because all the necessary code instrumentation is done automatically. The tester is given the opportunity to use monitoring facilities for observing interactions between distributed component applications. Monitoring a distributed system requires both observing system behavior at specific observation (or monitoring) points and effectively representing the observed information in a graphical or textual manner. In this paper, we only address the first aspect.

The remainder of this paper is organized as follows. Section 2 gives some background and motivates this paper. Section 3 identifies and analyses basic issues regarding the design of a monitoring framework. Section 4 outlines the architecture of the framework. Section 5 discusses related work. Section 6 presents our conclusions and exposes our plans for future work in this area.

2. Background and motivation

New technologies, in particular broadband and multimedia technologies, open many possibilities for the telecommunications market. In this dynamic environment, new and innovative services are being invented and introduced at a high speed. These services exist in a large variety of application domains, such as on-line entertainment, tele-conferencing, tele-education, tele-medicine, CSCW, and electronic commerce. In general, service providers, service developers, and service users struggle to keep up with the rate of change and constantly have to adapt their way of work.

Producing high quality software becomes a serious challenge, since development and testing of distributed applications requires new methodologies and more powerful tools. Quality testing involves diverse static and dynamic analysis techniques, which can handle a large number of monitoring states, visualize program behavior, support off-line log analysis, e.g., for the purpose of conformance testing, and others [5].

An important aspect of system design using (de-) composition is that the system must be verifiable and testable at the abstraction level of its design. For a component based design this means that not only the behavior of each separate component must be testable, but also the interactions between the components. For complex systems, language level debuggers are not very suitable for testing because they have no knowledge of components and manage distribution in proprietary ways. This means that tools and techniques are needed that allow a tester to verify and test distributed systems at higher levels of abstraction, e.g., at the component level. Once a component has been identified that does not behave as designed, implementation language level debuggers can be used to pinpoint the problem within the component. The majority of software development processes consider software quality during a testing phase and often alternate testing with the implementation phase. Exercising the program code in order to observe its real behavior is the ultimate test for every software product. Better tools and techniques are needed to help the tester carry out the observation activities through the cyclic development process of distributed applications.

Our approach allows a tester to test individual operations on component interfaces and to generate a sequence diagram as an invocation propagates through the system.

The monitoring system observes invocations at component interface levels and generates all the necessary information needed to identify the component, the involved interface, the operation, the parameter values, etc. The collected monitoring information can be used for generating graphical representations of complete sequence diagrams of the component interactions. During system testing, the output of the monitoring process can be used for conformance tests with the system design specification.

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 distributed component interactions through the system at runtime.
2. Automate any code instrumentation needed for monitoring in the development process, so that the developer will not be burdened with monitoring issues.
3. Prove that the approach is realistic by implementing a prototype of the monitoring system.

The monitoring framework has been designed and implemented within the cooperating research projects FRIENDS [10] and AMIDST [11]. A monitoring prototype has been developed within the FRIENDS project. This prototype has been integrated with the Distributed Software Component (DSC) [1, 2] model used in the FRIENDS integrated services platform.

3. Problem Analysis

This section identifies and analyses the following basic requirements, which are essential to our goals:

- Accuracy - relates directly to the granularity of the observed information, the frequency of the measurements and the information model of monitoring events;
- Expressive power - enables precise verification techniques like conformance testing using formal methods. The expressive power of the monitoring system rests on, for example, the ordering and causality information that can be retrieved from the generated monitoring events;
- Applicability - has to do with the software development process, how easy it is to incorporate, manage and use the monitoring framework. Another important issue is the performance and flexibility of the system at runtime.

We use the following terminology to further explain and discuss these requirements.

The monitoring system operates on applications - the *targets* that have been modified to facilitate a monitoring framework. A target consists of software units of distribution called *entities*. The framework monitors application *behavior* and as a result produces *monitoring events*. With application *behavior* we refer to the interaction patterns between the entities of an application. A *process* in this paper is equivalent to a *thread of execution*. An *execution environment* corresponds to the operating system running on a computer that is wired to a communication network.

3.1 Interaction contracts

We assume that the monitored distributed application consists of *entities* that interact with each other via a communication network. Further on in this paper, we specialize entities into DSC components for the purpose of the prototype of our framework.

The contemporary middleware platforms employ interaction contracts for standardizing the interactions between software entities. An example of such a contract is the OMG IDL specification. An example of a middleware platform is CORBA. The monitoring framework we offer is built around the notion of an interaction contract. For the purpose of building our prototype we chose CORBA to provide the contract for interactions between the entities of the distributed application.

In Fig. 1, entity A interacts with entity B using an interaction contract. The contract contains a set of services (IDL operations) that one of the entities explicitly provides (implements) and the other entity uses (invokes). In CORBA, the definition of interactions through a contract is unidirectional, unlike, for example, a bi-directional communication channel. Referring again to Fig. 1, Entity A has the role of initiator that calls an operation on the implementation of the contract in entity B.

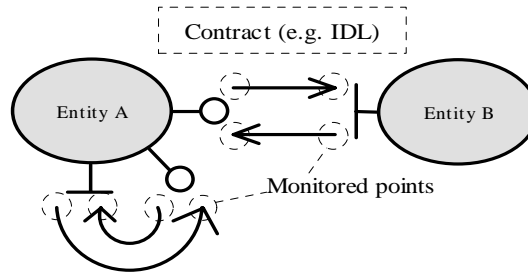


Fig. 1. Points of observation. Entity A interacts with B and itself.

We define the monitoring framework to observe entity interactions at the level of the contract. Every service in the contract is monitored and every value related to this service is recorded. The frequency of the measurements performed by the framework is determined by the monitored entities, which invoke services using the contract. Fig. 1 depicts how the framework observes entity interactions at monitored points.

Since the design and the implementation of the application entities follow exactly the contract definition, the monitoring framework will be able to monitor any application designed and implemented with this middleware. The prerequisites are such that the implementation of the monitoring framework conforms to the computational model of the middleware and the developers conform to the interaction contracts used with the particular middleware.

3.2 Execution model, order and causality relations of events

At any moment, distributed applications consist of a number of asynchronous processes that execute different tasks. Each process generates a sequential stream of

events. The processes communicate by passing messages using the available communication environment [8]. The exchange of a message also generates events in each one of the participating processes. Without restrictions we assume that process execution and message transfer are asynchronous, and the communication environment is finite and unpredictable. The processes, messages and events are main actors in the execution model of the distributed application.

The computational model of the middleware may influence the execution model of the distributed application. For example, a particular middleware product may allow usage of different policies for assigning of processes to interactions between entities.

In general, an interaction between two entities can be mapped to a message exchange between two processes within these entities. Nevertheless, there are cases, in which entity interactions cannot be mapped to process message exchange, as for example when two entities share the same execution environment.

Processes are called collocated if they act on the same entity. Entities are called collocated if they share the same execution environment. Normally, an execution environment corresponds to the operating system running on a physical node, however, in some cases, e.g. as for the Java Virtual Machine, it is possible to run several execution environments on one physical node.

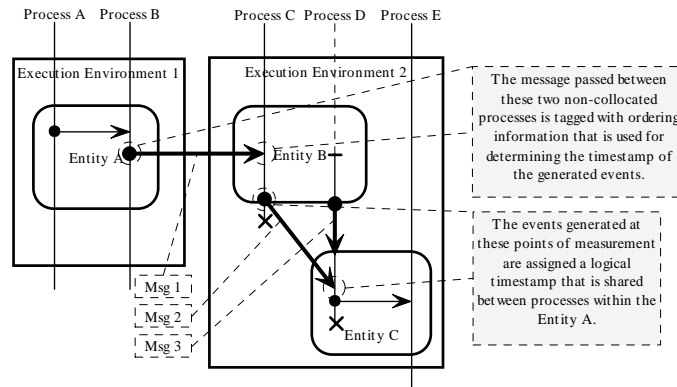


Fig. 2. Entities and processes deployed over execution environments and processes.

We take collocation into account as it relates directly to sharing common resources, e.g., memory space. Fig. 2 shows how several entities interact. Processes A and B are collocated processes, as are C, D and E, whereas entity B and C are collocated entities. Process C finishes after sending "Msg 2" to process D, which is created at the time the message is sent to it and dies after sending a message to process E. Process D is shared between entities and performs a direct method call ("Msg 3") across the boundaries of entities. This example comes from our experience using ORB implementations that optimize interactions between collocated entities, so that the usage of the communication environment is circumvented. Note that only the thick arrows depict entity interactions that are observed by our monitoring framework.

The monitoring framework captures its measurements into standalone events that can be stored to persistent media or can be sent to a consumer application (monitor) that further analyses them. In the monitored system, these events are generated in a

particular order. Distributed systems often operate in an environment of network delays and other unpredictable factors that may lead to receiving events in a different order than the order in which they were generated. There are different approaches that deal with this ordering problem. We employ solutions using a logical timestamp that is recorded in the event (See [8])

The monitoring of causality is useful to assess the correct working of applications. For example, it helps the tester to reason about the actual source of runtime errors during the testing phase of the software development process.

In an execution environment of asynchronous processes, the order relation provided by logical clocks is also a causal relation. However, the shared memory solution to retrieve ordering information within an entity does not allow one to keep track of causal relations anymore, since delays are introduced as a result of applying a scheme for sequential access to the shared logical clock.

After investigating several particular technologies, like the Java platform and the CORBA Interceptor standard, we have developed a technique for propagating the ordering information between collocated processes within the same entity. This technique allows us to restore causality information in a generic way, without entering a conflict with a black-box approach.

4. Technology solutions

In this section we investigate available technologies that can be used to build the monitoring framework, and we outline the software architecture of our prototype.

4.1 Black-box approach

An entity can be instantiated and configured as a part of an application. Deciding on what is an entity in the application influences the granularity of the monitoring system. Granularity does not really put restrictions on the monitoring system, as the *entity*-based observation model does not have a notion of what is inside the entity. Entity candidates in our system are CORBA objects (each one implementing a single interface) and software components (based on a particular component model [4]).

When we started this work, we had the DSC framework [2] available. DSC offers a component model similar to the CORBA component model [7], allowing rapid development of distributed components. Although components can be large entities, we have chosen granularity for three reasons: (1) component technology enters the software industry at a fast pace; (2) we had an advanced component framework available for experiments; and (3) components can be approached as black boxes [4].

DSC components use the CORBA middleware as distributed processing environment. In CORBA, interaction is done through invoking operations on an interface implementation. Furthermore, invocations can be synchronous and asynchronous.

The process semantics behind the synchronous and asynchronous invocations is not explicitly defined in the CORBA standard. Fig. 2 shows that sometimes messages passed between entities are not messages between processes but rather method calls in the same process and this depends only on the deployment of the entities, e.g. whether

they are collocated within the same execution environment or not.

We consider a component as a black-box designed and implemented by a third-party following a particular component model. Observable events, which occur in a component, are *lifecycle* events and *interaction* events. Lifecycle events relate to creation and destruction of a component instance, announcement of explicit dependencies and implemented interfaces, suspend or resume of component instances, etc. Lifecycle events occur within one process and are not related to the communication between processes. These events receive their timestamps by using the shared memory scheme. Interaction events are a result from an invocation of an operation from one component on an IDL interface implemented by another component.

To order the monitoring events, the monitoring framework relies on propagating monitoring information along with the invocations between components.

4.2 Context Propagation

In order to trace the propagation of an invocation through the system, each invocation is transparently tagged with context information, which is updated at each monitoring point. The context is a data structure that encapsulates value of a logical clock, causality information, etc. The context is propagated between interacting components.

Two particular problems have to be solved with respect to the context propagation:

- Sending context from one component to another, in a generic way;
- Propagating context through the black box of the custom component implementation, in a generic way.

Reflective technology allows 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 have investigated CORBA Interceptors, reflection on the thread model through Java and CORBA Portable Object Adapter (POA).

CORBA provides the *interceptor* mechanism to reflect on the invocation model by offering low level access to the CORBA request/reply object. Our monitoring scheme uses interceptors (message and process level) to pass monitoring context between components.

The black box approach interferes with the mechanism for propagating context inside the components of the distributed application. For example, when an invocation enters a component, it may be assigned to a process executing custom code that creates a number of parallel processes, each one following arbitrary interaction patterns with other components (Fig. 3). This custom code may be built by a third-party or in general is encapsulated in an off-the-shelf component.

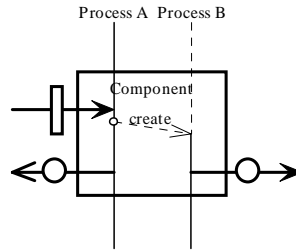


Fig. 3. Process assigned to an incoming invocation creates a second process.

We use the `ThreadLocal` Java class to assign context to processes (threads of execution) in a generic way [9]. The advance of the logical clock follows the original rule. We also use the `InheritableThreadLocal` Java class to propagate the monitoring information when a process creates another process (typical 'fork' in the execution) inside the black box.

Nevertheless, a generic scheme for tagging messages between threads of execution within the Java Virtual Machine is not available. For example, reflection on the message exchange (i.e. synchronization) between threads of execution allows us to propagate context within the component implementation. The current prototype of our monitoring framework can only partially restore causality relations between monitoring events.

The CORBA POA specification defines the execution semantics of the CORBA invocation. ORB implementations that conform to POA always call the stubs in skeletons. This allows us to correctly generate interaction events when the ORB performs invocation optimization for collocated components, i.e. one process handles the component interaction.

4.3 Configurability

Our implementation of the monitoring framework employs techniques like intercepting and tagging interactions between application components, generating and sending monitoring events to consumer applications (Fig. 4), and others. This behavior translates into high CPU utilization and frequent network transfers of monitoring events. Thus depending on the number of application components being monitored per execution environment and in the whole system, the overall performance of the target application is lower and the system may degrade to an unacceptable level.

We believe the solution to the performance problem is in a flexible monitoring framework, that does not employ bottleneck solutions. Configurability of the monitoring architecture in the component can be used to lower the CPU utilization in the physical node where components execute.

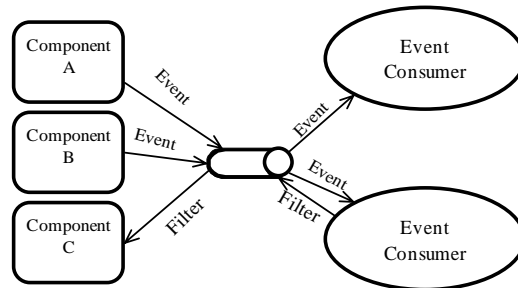


Fig. 4. Events and Event filters are delivered through the notification service.

Inside the component, we define two types of configurability: static and dynamic. Static configurability comprises reconfiguration activities that require stopping, reconfiguring (e.g. recompiling) and restarting of the component instances. The process is supported by monitoring tools. Depending on the IDL and a static description of what should be monitored inside, such tools determine whether component code will be modified to produce monitoring events or not.

Dynamic configurability does not require stopping and restarting of the component instances. Instead of this, the component implements a special interface that allows dynamic reconfiguration of the internal monitoring architecture, e.g., what for message types will be produced, turning on/off interception of particular interfaces, etc.

Dynamic configuration is preferable, because it does not put restrictions on the lifecycle of the components. Nevertheless, we choose to split configuration into static and dynamic, because some of the technologies (particular ORB implementations, Java, C++ compilers) allow only a static approach to some of the reconfiguration schemes, as for example the activation of interceptors. Moreover, when we choose not to monitor a system, a performance increase may be achieved with the help of static reconfiguration by removing all monitoring hooks. Another way of reducing workload at the components is to decouple components that act as event producers from event consumers. Decoupling can be achieved by using an event service like the CORBA notification service, such that components do not have to hold references to the event consumers anymore. Instead, a reference to the event channel is held and events are sent only once to the channel, compared to the solution where several consumers must explicitly be notified of the same event. Decoupling also increases the scalability of the monitoring framework and thus of the applications facilitated by the framework.

The CORBA notification service standard defines event filters. These filters encapsulate the request of the consumer application for particular events. Furthermore, event filters are propagated from the consumer through the notification service, to the sources of events - the components (Fig. 4). The monitoring framework can use the event filters for dynamic reconfiguration at the components.

4.4 Modular architecture

The architecture of the monitoring framework is modular and scalable. Inside the components, the monitoring architecture is based on the Portable Stub and Skeleton architecture as defined in [12].

The IDL compiler generates a pair of *Stub* and *Skeleton* classes for each interface type encountered in the IDL specification file. The Skeleton class is the base class from which the user-created implementation class must inherit. The Skeleton class contains an `_invoke()` operation that has the following signature:

```
public org.omg.CORBA.portable.OutputStream
_invoke(String method,
org.omg.CORBA.portable.InputStream input,
org.omg.CORBA.portable.ResponseHandler handler) throws
org.omg.CORBA.SystemException;
```

The body of the `_invoke` operation tests the value of parameter `method`, extracts the method parameters from the `input` parameter and invokes the method in the implementation class.

The Stub class is used as a local proxy for a (possibly) remote object. It implements all the operations of the target CORBA object. Within user-created implementation code a reference to a Stub object is obtained by narrowing a CORBA object reference to the appropriate type. When an operation is invoked on the Stub object, a CORBA request is assembled which contains the name of the operation, its parameter values and the destination. The request is further translated into an IIOP request and transmitted over the network by the ORB. The Skeleton and Stub classes mark the entry and exit points of an invocation on a component interface. Custom monitoring interceptor code executes just before and after each invocation, generates context information and prepares a monitoring event. Once the system is sufficiently tested, the monitoring code can be removed by recompiling the IDL files without using the monitoring IDL compiler tool.

Local monitor objects encapsulate functionality for processing the monitoring events like, assigning the proper context to an event, queuing and scheduling events for delivery to the consumer applications (CentralizedMonitor), switching between different event delivery policies (direct send, using notification service, logging to local persistent media) and others.

The CentralizedMonitor is an event consumer application that is capable of analyzing, storing, and presenting monitoring events to the component tester. The CentralizedMonitor component maintains several analysis tools (ObserverGUI objects) that analyze and represent event information properly to the component tester.

4.5 Information model

A monitoring event is a persistent object that contains a number of data fields.

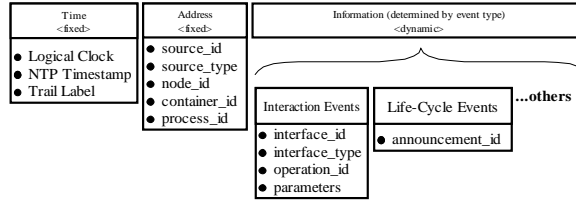


Fig. 5. Monitoring event structure.

The information model consists of three groups of fields: time, address, and information (Fig. 5). The time group contains a fixed set of fields that captures ordering and causality information. The address group contains a fixed set of fields with information about the source of the event. The information group contains variable fields, depending on the event type. For example, an interaction event captures information about interface (interaction contract) id and type, operation name, and parameter values.

4.6 Development process and test cases

A distributed application can be a large composition of components, interacting with each other through their CORBA interfaces in a distributed environment.

Fig. 6 shows the DSC component development process.

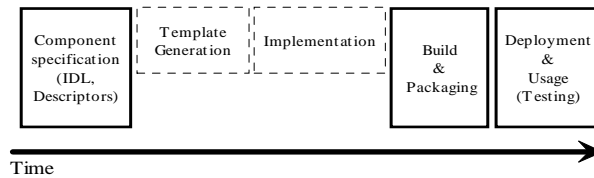


Fig. 6. Phases of the component development process essential to the monitoring framework.

DSC components are described in component specifications. A component specification is the input for the automated monitoring tools. Explicit dependencies and IDL types are the most important entries from the component specification, used during the component build and packaging phases for generation of the augmented with monitoring code stubs and skeletons (Fig. 6).

Once the components are packaged they can be deployed in an execution environment where monitoring tools are executed to observe the behavior of the distributed component application. Application testers follow a common scenario: starting of all centralized monitor components, running the application and usage of the analysis tools provided by the centralized monitoring application.

The central monitoring application is an event consumer specialized in collecting all events from the monitored system. From a GUI at the central monitor, the tester is

able to select entry or exit (interaction) points of the component test subjects and to assign them textual label values.

The label is then propagated with every interaction at each interaction point.

The monitor records the events it receives to a persistent storage. One of the graphical representations of the collected events is a Message Sequence Diagrams (MSD) viewer. Note that our notation is an extension of a subset of the ITU notation for message sequence charts [6]. We have also developed a visual tool allowing the application tester browse the complex parameters of the CORBA invocations.

4.7 Prototype

This section reports on different technology specific issues encountered during the implementation of the monitoring framework prototype.

The prototype currently does not use a CORBA Notification Service because no suitable commercial or freeware implementations were available to us.

The monitoring framework supports the following ORBs: Orbacus for Java 3.3, JacORB 1.0 beta 15, JavaORB 2.2.4, Visibroker for Java 3.4.

The monitoring framework makes use of the POA and the Interceptors interfaces. We encountered many problems with incorrect implementations of the POA and Interceptor specifications resulting in a submission of a number of reports to the vendors (Visibroker, JacORB, JavaORB, Orbacus).

The mechanism for propagating context between the components has been implemented using CORBA Interceptors. The specification allows generic access to the assembled request object of the CORBA invocation. The CORBA request format contains a specific field for this purpose: the service context field. This field can hold an array of service contexts that is transparently passed from client to server and vice versa. Below is the Java declaration of the service context field in a CORBA request:

```
Public org.omg.IOP.ServiceContext[]
service_context;
```

Each service context consists of a numeric identifier and a sequence of bytes. In case of a request from client to server it is used to propagate information about the execution context of the client object to the server; in case of a CORBA reply it contains information about the execution context of the server object that may be examined by the client.

Below is a part of the Java source code of the ServiceContext class:

```
package org.omg.IOP;
public final class ServiceContext implements
org.omg.CORBA.portable.IDLEntity
{
    public int context_id;
    public byte[] context_data;
```

A unique number was chosen to identify the DSC monitoring context in a sequence of service contexts.

At some point before the actual execution of a CORBA request (or reply) we need to insert the service context into the message. In a similar way we need to analyze the

service context that was propagated back from server to client in the reply to the request. The custom CORBA interceptors we provide, insert the context in the request at the outgoing points and retrieve context at the incoming points. For a detailed explanation of interceptors the reader is referred to the ORB manuals or the OMG documents about interceptors [14].

DSC monitoring uses message-level interceptors, which provide low-level access to the CORBA request/reply at the following four points:

- at the client side - before sending the request and after receiving the reply;
- at the server side - after receiving the request and before sending the reply.

Our framework performs instrumentation of the Stubs and Skeletons that can be replaced by the recently accepted Portable Interceptors submission [13], once implementations become available.

5. Related work

In [8], Raynal gives an overview of the methods for capturing causality in distributed systems. He defines an execution model of processes that pass messages between each other. In this environment Raynal describes the logical clock schemes for capturing order and causality of messages. Our monitoring framework uses Raynal's technique, however, our execution environment has constraints such that not all messages can be tagged with the necessary information. After applying a shared memory solution for propagating context between the processes within an entity, our framework is capable to employ the logical clock technique and restore order. Nevertheless, because of the shared memory approach, the causality relation in the execution model is broken and a separate solution is sought in order to track causality in the system.

In [15], Logean describes a method for run-time monitoring of distributed applications that supports a software development process. His approach employs ordering technique deriving from logical clocks. The level of entity granularity is the CORBA object. Our framework enhances this approach with introduction of the entity.

The configuration of the monitoring code in the Logean's method can be dynamic and supported by tools. By applying notification service, our approach introduces additional scheme for decoupling event sources from event consumers. Additionally, the monitoring framework supports several event types, ordering within multithreaded components, a tester tool for labeling invocations, support for different ORB implementations, and support for optimized invocations between collocated entities sharing one execution environment.

6. Conclusions

We succeeded to define a framework for the monitoring of component interactions, which has sufficient expressive power, allowing formal analyses to be performed. The information generated by the framework accurately follows the interaction contract, which allows monitoring of most applications implemented using contracts.

The monitoring framework seamlessly integrates in the component development

process, by providing automated tool support for all activities related to monitoring. The technologies used to implement the prototype are standard, the framework is flexible and configurable. The emphasis of the proposed monitoring architecture is on dynamic reconfiguration of the monitoring output (through event filters) as opposed to static reconfiguration broadly suggested by previous work done in this area.

The prototype of the monitoring framework is used for testing of services developed for a large application framework [10]. The component tester can make use of several facilities to track invocations, like the centralized monitor and the Message Sequence Diagram viewer.

The current framework supports only partial capturing of causality. In order to improve the opportunities for performing formal analysis, we need to extend the causality support in the framework.

Provided a formal model of the behavior of a distributed system, conformance testing can be done assisted by a tool. For this purpose, a mapping has to be created from the execution model of the monitoring system to the formal model describing the system behavior [16].

The monitoring framework can be specialized to support extension of distributed applications with generic monitoring functionality. The ultimate goal is to define a flexible framework that allows introduction of new application components into legacy distributed systems at reduced cost. All observation-related functionality necessary for the operation of the new application components will be defined as reconfiguration of the monitoring code in the legacy system. For example, using our framework, introduction of an accounting facility can be achieved by reconfiguring the legacy components to produce only events related to the accounting domain.

We intend to develop a generic architecture for development of monitoring application components. Such components use the basic event information model, combine it with a static data mapping and translate the monitoring information to a specific application domain. For example, monitoring events delivered to an accounting application components can be translated through static mapping to the terminology of the accounting domain.

References

1. Batteram, H., H.P. Idzenga. A generic software component framework for distributed communication architectures. *ECSCW'97 Workshop on OO Groupware Platforms*, 1997, Lancaster, UK, 68-73.
2. Bakker, J.L., H. Batteram. Design and evaluation of the Distributed Software Component framework for distributed communication architectures. *2nd Intl. Workshop on Enterprise Distributed Object Computing (EDOC'98)*, San Diego, USA, Nov. 3-5, 1998, 282-288.
3. The MESH project, see <http://www.mesh.nl/>
4. Szyperski, C. *Component software: Beyond OO programming*. Addison-Wesley, 1998.
5. Krawczyk, H., B. Wiszniewski. *Analysis and testing of distributed software applications*. Research Studies Press Ltd., Baldock, Hertfordshire, England, 1998.
6. *Message Sequence Chart (MSC)*. ITU-T Z.120, ITU-TSS, Oct. 1996.
7. *CORBA component model*. RFP orbos/97-06-12, Revised Submission, Feb. 2000.
8. Raynal, M., M. Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer* 29, Feb. 1996, 49-57.

9. *Java 2 platform*. Standard Edition, API spec., <http://java.sun.com/products/jdk/1.2/docs/api/>
10. The FRIENDS project, see <http://friends.gigaport.nl/>
11. The AMIDST project, see <http://amidst.ctit.utwente.nl/>
12. *IDL to Java mapping. Java ORB portability interfaces*. OMG TC Document formal/99-07-53, June 1999.
13. *Portable interceptors*. OMG TC Document orbosl/99-12-02, December 1999.
14. *CORBA 2.3 chapter 21, Interceptors*. OMG TC Document formal/99-07-25, July 1999.
15. Logean, X., F. Dietrich, H. Karamyan, S. Koppenhoefer. Run-time monitoring of distributed applications. *Middleware'98, IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*, Springer-Verlag, London, 2000, 459-473.
16. Quartel, D., M. van Sinderen, L. Ferreira Pires. A model-based approach to service creation. *7th IEEE Comp. Society Workshop on Future Trends of Distributed Computing Systems (FTDCS'99)*, IEEE Computer Society, 1999, 102-110.