Instrumentation of CORBA-compliant Applications for Monitoring Purposes*

Jakub Szymaszek, Aleksander Laurentowski, Krzysztof Zieliński
e-mails: {jasz, al, kz}@ics.agh.edu.pl
Institute of Computer Science, University of Mining & Metallurgy, Kraków, Poland

Abstract

This paper presents object-oriented instrumentation techniques that can be applied to CORBA-compliant applications for the purposes of monitoring, visualization or debugging. We propose low-intrusive instrumentation of source code automatically generated by the IDL compiler. The methodology has been employed in the MODIMOS monitoring system[2, 1] for IONA's Orbix 1.3 and the introduced overheads have been measured.

1 Introduction

The advances and growing expansion of CORBA-compliant systems put forward an issue of monitoring, visualization and debugging of the inherently complex distributed CORBA applications. A proper instrumentation technique is one of the basic issues in implementation of every monitoring system. The primary goal of instrumentation is to intercept events being in scope of the monitor. We understand instrumentation as a process of adding a set of intercepting sensors to the monitored application. Augmenting application code with monitoring facilities can concern various layers of the monitored software system, i.e. the the user-defined code, the run-time support modules, the libraries or some supporting mid-layer supplying information useful for the monitoring purposes. Instrumentation can be performed automatically (with usage of a preprocessor), semi-automatically or manually.

In the MODIMOS monitoring system[2, 1] development we have chosen the instrumentation of source code automatically generated by the IDL compiler. This solution relieves the user from modifying its (client and/or server) application code. The instrumentation scheme we propose is low intrusive, little change is required to the instrumented code itself while the sensors code is contained in separate modules which are bound at link time. Our solution can be easily performed automatically and we have implemented simple preprocessors supporting the proposed instrumentation process. The sensors are defined in form of C++ classes, representing the entities being monitored. The MODIMOS monitoring model includes the following entities (in order of descending levels of hierarchy): environment (the ORB run-time system), processes, objects, activation of operations (methods), and invocations of operations. Invocation is an abstract entity representing the state in which a client invokes an operation from the server object. Operation activation is an abstract entity representing the state in which an executing thread is passing a particular operation code at the server side. MODIMOS system monitors and visualizes events concerning instances of the above mentioned entities. There are two events pertinent for each such instance: its creation (appearance) and destruction (disappearance).

*This work was sponsored by the Polish State Committee of Research (KBN) under grant no. 8T11C01210
A. Laurentowski was partially sponsored by the Foundation for Polish Science

2 Description of Sensors

The instrumentation sensors are "shadows" of entities: they emerge when the entities appear (immediately reporting that fact to the monitor) and vanish when they disappear (previously sending a report message to the monitor). The sensors' code is therefore contained in two methods of each sensor class: init() and final(), and every effort is made to assure their invocation at the very beginning and end of the entity's (thus also the sensor's) life, respectively. Therefore, wherever possible, these methods are called implicitly in sensors' constructors and destructors. These two methods do most of the sensors' job, each one gathering the information needed to formulate an appropriate report message and sending that message to the monitor. The examples of that information can be: an object interface name and reference, a network address, ORB environment location, server activation mode, etc. Sensors form a hierarchy of levels, ordered as the entities listed in the previous section. "Parent" sensors can keep some information needed by "child" sensors, and a "child" can ask its "parent" about it.

2.1 Identifier Distribution

In every monitoring system there is a need of unique identification of monitored entities. Hence, an important task performed by MODIMOS sensors is assignment of unique identifiers for monitored entities. Each identifier has a hierarchical structure, and is a concatenation of the identifier of its parent entity and a unique child number within the parent. For instance, assuming that the identifier of a process is "1:3", the identifier of the first object in that process is "1:3:1". Therefore, in order to produce its entity identifier, the entity sensor has to ask its parent for the parent identifier and the unique child number.
An interface used for communication between sensors follows.

```
typedef sequence<octet> Identity;
interface Id_Distributor {
    unsigned long assign_child_num();        // Assigns a unique child number for a child entity.
    Identity get_id();                       // Returns the id of the entity monitored by the sensor.
};
```

An identifier distributor is a part of each sensor.

2.2 Environment Sensor

An environment sensor is activated in a separate server, called orbix_sensor. This sensor reports Orbix environment start-up and shut-down. Among other actions, it obtains a list of nodes where Orbix daemons are active (from the Orbix locator service) and sends it to the monitor.

```
class Environment_Sensor : public Id_Distributor_Impl {     // Base assigns child numbers for processes
  public:
    Environment_Sensor() {... init(); ...}
    ~Environment_Sensor() {final();}
  private:
    void init();     // Produces an environment id, formats a node list, reports environment creation.
    void final();                                              // Reports environment destruction.
    ...
 };
```

2.3 Process Sensor

A process sensor monitors creation and destruction of Orbix client or server process. It should be instantiated within a process as a global static C++ object. This guarantees that the sensor will be created after the process starts and it will be destroyed before the exit or return. Sensor constructor and destructor call the init() and final() method, respectively. Information about the monitored process are sent within the reporting message, as the event attributes. The information, such as executable name, application name, process identifier, address, server activation mode, server marker, etc., are obtained from constructor arguments, system calls, or from the Basic Object Adapter (BOA).

Another role of the process sensor, is to hold data like the application name, a reference of the monitor (being an Orbix object), the operation sensor for the currently active operation, and the invocation sensor for the current invocation, that are commonly used by other sensors residing within the process's address space.

```
class Process_Sensor : public Id_Distributor_Impl {          // Base cl. assigns child numbers for objects
  public:
    Process_Sensor(const char* container_name, const char* application_name) {... init(); ...}
    ~Process_Sensor() {... final(); ...}
    Local_Monitor *get_lm();              // Returns monitor reference used by all sensors in the process.
    char *get_application_name();             // Returns application name sent to monitor by all sensors
    Operation_Sensor *put_current_operation();        // Stores sensor for the currently active operation.
    Operation_Sensor *get_current_operation();        //Returns sensor for the currently active operation.
    Invocation_Sensor *put_current_invocation();              // Stores sensor for the current invocation.
    Invocation_Sensor *get_current_invocation();            // Returns sensor for the current invocation.
    static Process_Sensor *instance();                      // Returns the instance of the process sensor.
  private:
    void init();      // Formats process id, gets process data from the system & BOA, reports creation.
    void final();     // Reports process destruction, releases resources allocated for monitoring purposes.
    ...
};
```

2.4 Object Sensor

Object sensors monitor Orbix objects, that are always instances of IDL C++ classes derived from the CORBA::Object class and generated by the IDL compiler. However, some instances of IDL C++ class do not represent Orbix objects, i.e. proxies on the client side, which are out of our interest. Real Orbix objects are instances of TIE or BOAImpl classes, which are derived from IDL C++ classes. Therefore, the initial and final actions of an object sensor should be called (directly or indirectly) by the constructor and destructor of TIE and BOAImpl classes. Another requirement is that a sensor has to access information about an Orbix object stored in the CORBA::Object class and this information should be available at that time. We present one of the solutions to these requirements below.

The object sensor class is introduced as a base class of IDL C++ class. The initial action of an object sensor is called explicitly by constructors of TIE and BOAImpl classes, when the initialization of CORBA::Object is complete. As an argument of init() a pointer to the sibling CORBA::Object class should be passed. The sensor destructor and final() are called automatically, when the Orbix object is destroyed.

The object sensor sends to the monitor its object's and process's identifiers, a reference, an implementation definition, etc. which are obtained from the process sensor (accessible via Process_Sensor::instance invocation) and the monitored object.

The object sensor declaration and a sample instrumented IDL C++ and BOAImpl classes follow (the instrumentation of TIE class constructors is similar). All the code needed to be inserted during instrumentation is underlined.

```
class Object_Sensor : public Id_Distributor_Impl {       // Base cl. assigns child numbers for operations
  protected:
    Object_Sensor();
    ~Object_Sensor() {_MODIMOS_final();}
    _MODIMOS_init(const CORBA::Object* obj);                    //Called by BOAImpl or TIE ctors.
  private:
    void _MODIMOS_final();                                           // Reports object destruction.
     ...
};
```

```
//grid.hh
class grid: public virtual CORBA::Object, public virtual __MODIMOS__::Object_Sensor {...};

class gridBOAImpl : public virtual grid {
public:
    gridBOAImpl (const char *m="", CORBA::LoaderClass *l=NULL) {
        if (CORBA::PPTR::isOK (m_pptr, grid_IR))
            m_pptr = new grid_dispatch ( (grid*)this,
                (CORBA::Object*)this, m, l, grid_IR, this);
        __MODIMOS_init((CORBA::Object*) this);          // init invocation added to the ctor
    }
    ...
};
```

2.5 Operation Activation and Invocation Sensors

Operation activations and invocations can be quite easy monitored in Orbix, if you take advantage of process filters that allow to monitor all the requests coming in and going out of a process.

When a monitored process starts, its process sensor creates an instance of the Reporting_Filter class. The Reporting_Filter class overrides four methods of CORBA::Filter, that are invoked by Orbix when a request containing all the arguments comes into a server (operation activation event), goes out of a server (operation termination event), goes out of a client (operation invocation event), and returns to a client (end of invocation event).

```
class Operation_Sensor : public Id_Distributor_Impl {     // Base assigns child numbers for invocations.
  public:
    void init(CORBA::Request& r) {...caller_id.decodeOp(r);...}  // Reports operation activation.
    void final(const CORBA::Request&);           // Gets data from a request, reports op. termination.
};

class Invocation_Sensor {                        // The lowest level sensor does not assign child numbers.
  public:
    void init(CORBA::Request& r) {... Process_Sensor::instance()→get_current_operation()→get_id();
        ... my_id→encodeOp(r); ...}  // Inserts invocation id, reports op. invocation
    void final(const CORBA::Request& r);          // Gets data from request, reports end of invocation
};

class Reporting_Filter : public CORBA::Filter {
  public :
    ...
    unsigned char inRequestPreMarshal(CORBA::Request& r, CORBA::Environment&) {
        ...
        Operation_Sensor *ms = new Operation_Sensor;             // Creates a new operation sensor.
        ms→init(r);                                                        // Calls initial action.
        Process_Sensor::instance()→put_current_operation(ms);          // Stores the operation sensor.
        ...
    }
    unsigned char outReplyPostMarshal(CORBA::Request& r, CORBA::Environment&) {
        ...
        Operation_Sensor *ms = Process_Sensor::instance()→get_current_operation();
        ms→final(r);
        delete ms;
        ...
    }
    unsigned char outRequestPostMarshal(CORBA::Request& r, CORBA::Environment&) {...}
    unsigned char inReplyPreMarshal(CORBA::Request& r, CORBA::Environment&) {...}
};
```

When a request comes into a server or goes out of a client, the filter creates an operation or invocation sensor, calls its initial action (passing the request as an argument), and stores the operation or invocation sensor in the process sensor. When a request goes out of the server or returns to the client, the filter gets the operation or invocation sensor pointer stored in the process sensor, calls its final action and destroys the sensor.

Operation and invocation sensors analyze request packets and send reports about operation activations and invocations to the monitor, respectively. Moreover, each sensor sends along its entity identifier (part of which is assigned by the parent sensor).

An additional information reported to the monitor by every operation sensor is an identifier of invocation which caused activation of that operation. Clients of the monitoring system, such as visualizers, can use this identifier to recover relationship between invocations and operations, and e.g. draw an arrow between the figures representing them. This invocation identifier is inserted into a request by an invocation sensor on a client side and it is then removed by an operation sensor on the server side.

Note that the presented method of monitoring inter-process communication allows to avoid instrumentation of any user-written or IDL-generated code. To do this without filters, one would have to modify a signature (pass an additional parameter) invocation and implementation of each IDL operation.

However, using filters has some drawbacks. It does not allow to monitor interactions between collocated objects. We also found out that filters do not catch properly DII requests, i.e. replies for DII invocations are not caught by filters on the client side. Moreover, it is not possible to retrieve (and report to the monitor) the actual operation parameters without causing the application to crash. This is due to the fact that it is not possible to retrieve arguments from a request without removing them. This problem could be solved if the CORBA::Request class provided a copy constructor, but it did not.

Another major disadvantage of the presented method is that it works correctly only for single-threaded processes and synchronous invocations. The solution is based on assumptions that there might be only one instance of an operation active in a process at a time, and that the active operation can make only one invocation. Sensors for the current operation and invocation are held in the process sensor.
In multi-threaded applications it is each thread to store the sensor for the active operation. In such a case using filters is not possible because Orbix filters process each request in one separate thread and have no access to the state of other threads. Hence, changes of multithreaded server and client code are necessary for proper instrumentation.

3 How to Instrument an Orbix Process

Instrumentation of Orbix applications requires no changes in user-written code. The user has to perform the following instrumentation steps: 1) write, compile and link to the executable a file containing solely a declaration of the static process sensor object; 2) compile all IDL interfaces and change the generated header files as described in Sec 2.4, to achieve instrumentation of object creation and destruction; 3) link all objects and libraries together with the orbix_instr library containing the sensors code (definitions of all sensor classes).

4 Performance Evaluation

The key question concerning monitoring systems usually refers to the overhead introduced by the instrumentation. In MODIMOS the most time-consuming operations are: reporting events and formatting their parameters. Selected results of a preliminary performance study for Orbix 1.3 running over a network of SUN SPARCstations 5 have been shown in Table 1a, 1b, and 2. The client, the server, and the monitor were all running on separate machines. The table 1 left shows that the instrumented sample applications run nearly twice as long as the original ones. The largest percentage overheads are introduced by instrumentation of Orbix objects (table 1 right). However, absolute values of object creation and destruction time are relatively small and have little effect the global execution time.

For operation invocations, the relative overhead defined as a ratio of the instrumented to non-instrumented invocation time is particularly important. The results presented in Table2 show that

Application name	Non-instrumented time [s]	Instrumented time [s]	Overhead [%]
grid	0.23	0.39	172
timer	16.05	20.02	125
bank	1.57	2.82	180

Application name	Non-instrumented time [s]	Instrumented time [s]	Overhead [%]
account	0.49	3.93	804
bank	0.46	3.86	831
currentAccount	0.55	4.00	766

Table 1: left: Execution times and overheads for some demo applications; right: Creation times and overheads for objects in the bank demo application

this relative overhead decreases when complexity of operation raises. In the case when an operation is enough complex to justify its remote processing, the instrumentation overhead seems to be acceptable. The overhead is low, and could be even lower in large applications, where substantial amounts of data are transfered.

Application name	Number of bytes	Non-instrumented time [ms]	Instrumented time [ms]	Overhead [%]
Oneway::test_no_param	0	0.67	3.55	526
Oneway::test_prim_args	25	0.76	3.67	480
Oneway::test_struct	2500	0.74	4.11	551
Oneway::test_prim_seq	2500	5.57	6.34	114
Oneway::test_struct_seq	2500	5.56	7.19	129
Oneway::test_struct_array	2500	6.23	7.19	115
TestReqReply::test_prim_args	100	9.55	14.84	155
TestReqReply::test_struct_args	100	7.92	13.73	173
TestReqReply::test_prim_seq	10000	30.65	36.32	118
TestReqReply::test_struct_seq	10000	43.06	47.95	111
TestReqReply::test_struct_array	10000	38.61	43.73	113

Table 2: Remote operation invocation times and overheads for timer demo application

5 Summary

This paper presented shortly an instrumentation technique developed for Orbix 1.3 applications for the needs of MODIMOS monitoring system. We assumed that the technique should be simple and suitable for both manual and automatic instrumentation, as well as that it should not require any changes of user-written code. The result of our research was the concept of entity sensors and object-oriented instrumentation. A preliminary performance study was presented at the end of the paper.

References

[1] Jakub Szymaszek. Monitoring and Visualizing Software-Heterogeneous Distributed Object Applications. In *Proceedings of European Research Seminar on Advances in Distributed Systems (ERSADS'94)*, pages 138–144, France, April 1995. IMAG & INRIA Rocquencourt.

[2] K. Zieliński, A. Laurentowski, J. Szymaszek, and A. 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.