High performance Non-intrusive Distributed CORBA Monitoring

Brecht Vermeulen, David De Reu, Piet Demeester
Department of Information Technology
Ghent University, Belgium
brecht.vermeulen@intec.rug.ac.be
Tel.: +32 9 267 35 86 Fax.: +32 9 267 35 99
Abstract—Due to a vast emergence of distributed software based on CORBA, a clear need arises for tools to ease the development, debugging and profiling of such complex distributed software. Existing tools were not satisfactory for our intentions because they show only a sequential packet based view of the CORBA communication, they need instrumentation of the software under test or they only provide a single sniffing point. As a result, we developed our own tool for CORBA monitoring which supports at the user-end a GUI which gives a clear overview of the present objects and the communication between these objects and which is at the low level end supported by distributed sniffing of network traffic on different points in the network. Furthermore, sniffing is done by (high end) Linux PCs with the MIT Click Modular Router package which makes it possible to let the network cards work in a polling mode instead of in interrupt mode to avoid the receive livelock problem. As a result, fully loaded gigabit ethernet links with a minority of CORBA traffic (which is e.g., the case if CORBA is used in network management) can be sniffed without dropping any GIOP packet. Future work includes further investigation of the profiling part by letting the monitor automatically create statistics about the recorded communication and in a further step this information can then be used for automatic object distributing and load balancing of CORBA objects based on monitoring input.

Keywords—CORBA, monitoring, distributed, non-intrusive, management

I. INTRODUCTION AND MOTIVATION

CORBA is gaining more and more field in various areas of software development, a.o. network management [1], distributed applications [2], even in not distributed applications as the GNOME window-manager [3] which profit from the concept of language independent IDL (Interface Definition Language) interfaces. Reducing the time-to-market is an important constraint on modern software development and as such, tools are needed to help with the complex development, debugging and profiling of CORBA based applications. Existing debugging and profiling tools are not aware of the distributed aspects of the applications and cannot help in e.g. improving the spreading or collocation of objects. Demonstration of such distributed frameworks is also an important need.

Different approaches are possible and exist (see also Section V), but no existing tool fullfills all our requirements. Our most important requirement is the strictly non-intrusive monitoring of CORBA components. Also solutions which use portable interceptors or adapt the IDL compiler to instrument the CORBA components under test are not satisfactory as they cannot be used on all CORBA implementations, need an ORB specific implementation and need a recompilation of the components. A second need is to use sniffing on different points in a network or multiple networks, because distributed software is per definition distributed over a network and sniffing on one point in the network may not be sufficient to see all communication. The gathered information has to be used then to visualize the communication on a Graphical User Interface, but also with a modular design, so that other back-ends (e.g. other GUIs or components which use the gathered information for statistics or load balancing) can be easily added to the framework. As a last requirement, the performance must be optimized so that high speed links (say gigabit ethernet) with a minority of CORBA traffic can be monitored. The developed CORBA Monitoring Framework fulfills all of these requirements.

The paper is further structured as follows: Section II describes the developed framework and the design decisions in detail. The measured performance of the tool is depicted in Section III while possible applications and future work are mentioned in Section IV. Related work and other tools are listed in Section V, finally Section VI concludes the paper.

II. MONITORING FRAMEWORK

A. Coping with the requirements

Sniffing CORBA GIOP traffic (General Inter-ORB Protocol) on the wire is the only way to avoid instrumentation or recompilation of the CORBA components under test. Instrumentation of the ORB dynamic libraries would maybe be an alternative to prevent recompilation of the components, but this would need an ORB specific implementation and would limit the use to ORBs and operating systems based on dynamic libraries. Of course, sniffing only has also its limitations: only shared networks or switches which support port mirroring can be used and not all needed data may be available by sniffing GIOP packets. For our needs however, these were not major issues.

Sniffing CORBA messages on different points in the network or networks is solved by simply using distributed sniffing and gathering the information to a central point to process it further and to filter e.g. duplicate information. Of course, new problems as e.g. how to transfer the information to the central point and synchronization issues come into play here, see further.

To support a modular design and provide a clean interface for various back-ends, CORBA and IDL interfaces for the monitoring framework itself seem ideally suited and also make it possible to run the back-end transparently remote or even as a JAVA applet from a browser.

The performance is an actual issue as the standard interrupt driven Linux operating system suffers from receive livelock [4]: network tasks are scheduled driven by interrupts of the network cards and as such in extreme conditions of much network traffic (and they do not have to be that extreme) all time is spent in the interrupts and no other work is done resulting in e.g. not processing any packet. See the performance results for examples of this. The solution for this is using polling instead of interrupt driven networking. Therefore, the MIT Click Modular Router software [5] was used to provide within Linux polling of the sniffing network interface and which supports also the filtering of the CORBA GIOP packets from the other traffic in kernel space and has as a consequence that only GIOP packets have to be transferred to the monitor processes in user space. A disadvantage is of course that the CPU is fully loaded because of the polling, but this was solved by using dual processor machines which have one CPU fully loaded for the polling and in which the user space monitor processes run on the second CPU.
B. Architecture

As a logical consequence, the architecture as depicted in Figure 1 meets all requirements. Monitors sniff GIOP packets at different places in the network, feed their data to a central Correlator which removes duplicates and stores the CORBA messages in a queue until a back-end, e.g. a GUI, fetches them.

The communication between these three types of components makes also use of CORBA, see Section II-D. The Monitors push their data into the Correlator while the back-end GUI pulls its data from the Correlator. As such, the Monitors and Correlator are time critical to avoid losing too much sniffed packets while the back-end can now and then, before the queue of the Correlator is filled up, pull the data from the Correlator in a much less time critical manner. The Monitors and Correlator are carefully designed and developed in C++ while the back-ends can e.g. be developed in Java. The Correlator can also contact an Interface Repository to gain better knowledge of the IDL interfaces in the CORBA communication under test as will be described in the next section.

C. What can be sniffed?

As already said before, by sniffing GIOP packets [6], one has only access to limited information. The tool restricts itself to the sniffing of IP packets which contain IOP messages (Internet Inter-ORB protocol, the specialization of GIOP to the TCP/IP protocol). Those packets can be recognized because the first four bytes of the TCP payload are “GIOP”. From those packets the following data is known. Of course the source and destination IP address and TCP port can be extracted from the IP and TCP headers. In the GIOP header, amongst others the GIOP version, the CORBA message type (Request, Reply, LocateRequest, MessageError, . . . ), the endianness of the messages (needed for further analysis of the content) and the message size can be found.

On its turn, each message type has its own header, and optionally its own payload. For the two most important message types (Request and Reply) these contain the following. A Request message contains in its header a request_id which is used to track down the corresponding reply, the object_key of the destination CORBA object and the name of the called operation, while the body contains the CDR encoded (Common Data Representation) in and inout parameters of the request. A Reply message header on the other hand has a request_id and a reply_status field, which describes the status of the reply (NO EXCEPTION, USER EXCEPTION, SYSTEM EXCEPTION, . . . ). The Reply body holds then data regarding the status of the reply and this can be the CDR-encoded return value, out and inout parameters or more information about a thrown exception.

The most important shortcomings are that in a Request the destination CORBA object is perfectly known by the unique object_key, but the client is only known by the source IP address and no distinction can be made between processes on that host (even the TCP source port cannot be used as this can vary if various connections are set up). The CDR-encoded parameters are also not self-describing which means that the IDL definition has to be known to decode these parameters. As a solution for this, the Correlator component provides the possibility to contact an Interface Repository where the IDL definitions of the CORBA objects under tests can be registered.

D. Communication between the components

As described earlier, the different components in the architecture also use CORBA to communicate between themselves. For the time critical communication between the Monitors and Correlator however, a performance study was done to discover the best alternative. Choices were: plain sockets, CORBA or the CORBA event service. The last option is from an architectural viewpoint the best as it provides automatically the concept of multiple producers (the Monitors) and consumers (one or more Correlators e.g.) while they don’t have to know each other. Considering also the performance, this solution is not adequate as it introduces one more component which fulfills more or less the role of the Correlator, but of course no extra functionality can be introduced in this standard service. Internal performance tests show also that the Event service is not that performant.

Remains the choice between sockets and CORBA: the first have the advantage of speed, possibility to use UDP or TCP while CORBA has the well-known advantages of transparent distribution, platform and language independence and provides the transport of high-level datatypes for which the marshalling is done by the ORB. On the other side, if one has to program this marshalling himself (one side has always to interpret what the other side sends and the communication between Monitors and Correlator is not that basic that it is a simple byte stream as e.g. a video-stream), this cannot be done much more efficient than it is implemented and optimized already since years in ORB implementations.

The speed offered by UDP (which doesn’t acknowledge its data and which doesn’t retransmit lost packets) however is a big advantage. Standard CORBA calls are especially costly because first a Request packet is sent, followed by
a Reply packet in the other direction even if no return value is expected. CORBA offers however also a UDP like technique, called oneway calls, which are embodied by a single Request packet without a Reply packet. Of course, these are still sent using TCP/IP, but the big advantage is that, because there is no Reply message expected, several oneways can be sent in one TCP segment instead of the Request and Reply each consuming a TCP segment. These oneways are UDP like because a fully compliant ORB can drop all of the oneway calls.

So, a performance measurement urged itself to investigate the performance and characteristics of these oneway calls. Figure 2 shows the results for omniORB 3.0.4 [7] and Orbacus 4.0.3 [8]. CORBA benchmarking at the Charles University of Prague [9] showed already that omniORB is one of the fastest ORBs around and the figure shows the same. As parameter a struct containing a long, a double, a boolean and a string of 71 characters, similar to the IDL interface of the Correlator, was used in all tests. Tests were done over a dedicated 100 Mbit/s link between two Linux PCs (Debian 2.2 with 2.2.19 kernel). The choice to use omniORB as ORB is very easy (although our framework is developed in a portable way and can be compiled with both ORBs) and also the oneway calls seem very performant while sniffing the network showed indeed that multiple oneway requests are bundled in one TCP segment. In terms of bandwidth, the oneway calls could be sent at approximately 75 Mbit/s which should be enough if the CORBA communication under test is the minority of traffic (see also Section III). The reason why the Orbus performance of oneway is very poor, can be ascribed to a poor implementation: a client performing, say, one million oneway calls, as used in our benchmark, queues all oneway calls in its memory without sending one oneway call onto the wire, using huge parts of memory even using swapping memory which is of course very slow. In the limit, our machines crashed by lacking memory and they had 256 MB RAM and 256 MB swap space. OmniORB didn’t suffer of this and had a constant throughput leading to very good results. These results lead to the implementation choice of using omniORB oneway calls between the Monitors and the Correlator.

E. Detailed functionality of framework

The components of the monitoring framework depicted in Figure 1 have the following functionality:

Monitors. The Monitors are basically running two working threads. One sniffs the packets via the libpcap library [10] (which on its turn receives the packets from the Click kernel module to avoid livelock) and puts a copy of the packets in a synchronized circular queue (of e.g. 1000 packets which can be configured as command line option). Another thread does the interpretation of the GIOP message and calls a message type specific oneway operation on the Correlator transferring the timestamp of the message and the necessary information. Note that optionally (specified by the back-end when the monitoring is started) also the body of the message containing the parameters can be transferred to the Correlator (the interpretation of these parameters is not time critical and because of the needed processing power is not feasible in the Monitors, this is however not yet implemented).

Correlator. The Correlator stores the received messages in a time sorted queue, while neglecting duplicate messages, until a back-end fetches the queued messages. Optionally the Correlator can also obtain the most derived Repository Id of the CORBA objects. Therefore the CORBA object servers under test should be able to reach an Interface Repository (typically via a system wide configuration file) where its IDL interfaces should be registered. If the Correlator then detects a new object_key (this is done only once for each object) an object reference based on the IP address, TCP port and object_key is created and the get_interface operation is called on the object, which on its turn contacts the Interface Repository to return a reference to an object of type InterfaceDef on which the Correlator can call the describe_interface operation to get a full interface definition. The goal of this is two-fold: firstly a GUI e.g. can show the type of the objects on the screen and secondly to analyze the parameters of the messages, the IDL definition has to be known.

Back-end. We have implemented a GUI to be used as back-end but also more intelligent back-ends (see e.g. Section IV) could be developed. Therefore an IDL interface is specified on the Correlator which supports starting/stopping the Monitors, getting information on the number of received packets and fetching the messages. The back-end itself has to build an overview of the present objects and receives only the sniffed and analyzed messages in a time-sorted order. The back-end pulls its information from the Correlator, because also implementing CORBA objects in the back-end would limit the flexibility (e.g. the use of applet clients wouldn’t be possible).

III. Performance

To evaluate the monitoring framework, extensive performance tests were done testing various performance characteristics. The tests were done on gigabit ethernet where one port of a gigabit switch was mirrored and to which one monitor was connected. The mirrored port was loaded with real CORBA traffic of various bitrates between two

![Fig. 2. Performance of oneway calls vs. in/out/inout parameters](image-url)
hosts (AMD Athlon 1 GHz and 750 MHz) and also with background non-CORBA traffic generated by a Smartbits 2000 device with a gigabit interface. The Monitor PC was a high-end dual 1.266 GHz Pentium III PC with a motherboard supporting a 64 bit/66MHz PCI bus which contained an Intel Ether Pro 1000/F gigabit ethernet card. The CORBA traffic sent was a simple Hello World example with a string of 600 characters as parameter and the same string returned as return value.

A first test was considered as a normal use case (Figure 3): low CORBA traffic (approx. 45 calls/s, totally 540kbit/s load for the requests and replies) with heavy background traffic. Background1 is a load of 191.718 pps (packets per second) of 600 bytes resulting in 957 MBit/s while background2 is a load of 1.077.586 pps of 60 bytes resulting in 724 MBit/s. The figure shows that the framework with standard interrupt-driven Linux sniffs only a minority of the CORBA messages (vertical axis), even worse, not a single message is being sniffed for higher packet rates (background2), illustrating the receive livelock problem described earlier. If however the Click module is used to turn polling on on the interface, not a single message is missed!

With and without body indicate if the message body is transferred from Monitor to Correlator.

![Fig. 3. Normal usecase: low CORBA traffic (540kbit/s, 45 calls/s) with background traffic](image)

To test the throughput of the monitoring framework itself (Figure 4), no background traffic was used and the CORBA traffic under test was varied in bandwidth and calls/second. Here, Linux without the Click module is performing slightly better, and this is because the polling fully occupies one CPU and as such the remaining processing power of the host is lower of course.

Finally also the memory use in the Correlator was measured in relation to the number of messages queued because they weren’t fetched by a back-end (Figure 5). Averaged per message, about 230 bytes per message is needed without body and around 830 bytes is needed with the body. This is of course very logical as the string which was used in the tests as parameter was 600 bytes long.

![Fig. 4. Throughput of the monitoring framework without background traffic](image)

![Fig. 5. Memory use in the correlator](image)

IV. Applications and further work

A couple of applications of a CORBA monitoring architecture were already shortly mentioned before. Debugging and demonstration of CORBA based distributed software was our primary goal and as such a GUI was developed as back-end (Figure 6). The next step is to add the analysis of the parameters of the calls to make debugging still easier.

However, the use of the tool with our existing software made it clear that profiling is also possible: one sees all distributed traffic and as such one can optimize code by being more efficient regarding remote calls. As such a whole range of other back-ends could be developed: a Message Sequence Chart GUI could be interesting to see the calls more in real-time (our current GUI shows one message a time and as such the user has to manually run through the messages). A profiler GUI could show a picture with all objects and clients and draw lines between them which vary in thickness according the number of calls. This information could then lead to e.g. collocation of some objects.

Another topic which we will research in the future, will be the active load balancing of objects based on information gathered by the monitoring. This is related to the profiler back-end, but instead of manually moving objects, it should also be possible to do this more dynamically and automatically.
V. Related work

A lot of related work can be found in the literature which can be divided in two big categories: intrusive and non-intrusive monitors. Intrusive monitors include e.g. Object-Monitor/VISCO [11] developed at the University of Wiesbaden which is based on the filters provided in Orbix (which is a kind of predecessor of Portable Interceptors) and the Monitoring Framework [12] which is considered as non-intrusive by the authors. However, they have adapted the IDL compiler to include some instrumentation and as such the components under test have to be recompiled. This imposes indeed no burden on the programmers, but they have only adapted the IDL compiler of JacORB so it is not usable with all ORBs. As described in the requirements these tools were not sufficient for our needs because they are ORB specific and need instrumentation of the software.

Amongst the non-intrusive tools that can be found are tcpdump [10] and ethereal [13], both based on libpcap which provide a view on the sequence of sniffed packets. Ethereal offers also some analysis of GIOP packets but both are not relating any messages to objects or client-server communication. They also only snoop packets on a single point in the network.

IIOP-tracer [14] is another tool developed at the University of Wiesbaden and is based on sniffing and comes close to our framework but there is no GUI back-end available, it is not freely available and as such couldn’t be tested and if multiple sniffers are used, the back-end has to cope with this. Also, no information is known on the performance of the framework.

VI. Conclusion

The CORBA Monitoring Framework proposed in this paper was developed because no existing tool could answer all our needs, which include non-intrusive monitoring by sniffing on different points in the network, a modular design which makes it possible to use different back-ends and as a last requirement the performance had to be optimized.

The new tool does indeed cope with all these requirements and the performance is even better than we had hoped and as such new applications as e.g. active load balancing and profiling become possible and will be studied in the future. The framework will also be made available as Open Source software in the near future.

ACKNOWLEDGMENTS

Part of this work has been supported by the European Community through the IST research program in the IST-1999-11719 project HARMONICS.

REFERENCES