Approach for Debugging in a Distributed System

M. Riedl∗ R. Schneckenhaus∗ M. Meier∗ Ch. Diedrich∗∗

∗ Institut f. Automation und Kommunikation, Magdeburg, Germany (Tel: +49 391 9901460; e-mail: matthias.riedl@ifak.eu).
∗∗ Institut f. Automatisierungstechnik (IFAT), OvGU Magdeburg, Germany (Tel: +49 391 67 18499; e-mail: christian.diedrich@ovgu.de)

Abstract: The concept of distributed systems has proven itself in modern automation concepts. However, efficient debugging of such systems is still a field of research. Different design principles, e.g. SOA, allow the definition of distributed systems. Such systems provide the ability to design complex but flexible applications that consist of several distributed objects or services, executed on different nodes and communication interfaces between them. Specific implementations offer varying capabilities. Up to now such systems mostly supply built-in support only for process logging, securing of node communication and error handling. The paper describes an approach to trace the interaction of the components or to debug the whole distributed application from a central point.

1. INTRODUCTION

Nowadays distributed systems, e.g. based on the Service Oriented Architecture design principle as investigated in research projects SOCRADES [2009], have been established in many industrial sectors. The same performances are reachable at lower costs by using a cluster of low-end computers instead of employing single high-end computers. In addition, the technique of parallelisation can be deployed to compute expensive tasks in less time or to split up the network load of many clients. Distributed systems are easy to extend by adding additional computers in contrast to upgrading a single mainframe.

Particularly with regard to safety-relevant usage it is important that systems do not completely crash if something goes wrong. Distributed applications are able to keep running if components cause an error or are unreachable. Moreover it is possible to bring in some other nodes to take over the tasks of the absent component in some distributed systems.

Distributed systems are also used to enable the shared usage of resources. In this case a resource is an abstract term that can be comprehended as any reasonable hardware or software component such as hard disks, printers, databases or any desired data.

Aside from all the advantages of distributed systems there are also a few drawbacks that should be considered.

One central problem of distributed systems is the synchronisation of all sub-processes. Events can be triggered in any order and at almost any time. However shared resources only can be accessed by one process at the same time. So-called deadlocks can be the consequence. This behaviour also implies another difficulty that concerns the troubleshooting in a distributed application.

The global application status is represented by the sum of all sub-process states and the states of the communication channels as mentioned in KsSi [2008]. Each of these processes is able to change its state independently at any time and in a way that is not transparent for the whole application in all cases. Therefore the process-local states have to be gathered and controlled to ensure valid and actual application states.

To detect run-time errors in programs, a debugger is the instrument of choice. Debuggers for distributed applications have to provide extra functionality that exceeds the normal amount of the debugger’s tasks. Such a debugger has to gather debug information from any sub-process and make it accessible at a central point. In addition the concept of breakpoints has to be reconsidered. Triggered breakpoints have to stop the whole application at a specific state.

The remainder of this paper is organised as follows: The second section briefly presents the model of the distributed system, wherefore the debugging approach is implemented. The third chapter briefly discusses different principles about distributed debugging, the fourth chapter analyses the requirements of the distributed debugging and the fifth chapter introduces the approach. The sixth chapter presents the implemented solution and finally the last chapter summarises the benefits of the presented approach and gives an outlook about the next steps, improving the found solution.

2. OVERVIEW OF THE USED DISTRIBUTED SYSTEM

2.1 General Introduction

The used distributed system for applying the debugging approach is called Distributed Object Model Environment (DOME) and is described in Riedl [2005]. DOME applications consist of one or more network nodes that are represented by so called DOME Managers. Such managers administrate all running processes on a specific computer or...
network node. These processes are the parts that form the application in their entirety. A process is able to contain several DOME objects like groups, class objects and links. Groups can be understood as execution contexts that are logically separated. Basically the process consists of one or more groups which contain class objects. In addition link objects are present to enable object communication, see Figure 1.

Figure 2 shows an example of a basic DOME application, consisting of several processes and a special process for GUI tasks. The processes are connected via NodeConnections. DOME uses these NodeConnections for the real data transport of the link objects.

2.2 Links

Links are communication channels that connect I- and O-ports. In DOME the O-ports are called require ports and the I-ports aliased as service ports, see Figure 3. Basically a link will call the service port routine of the target object when the associated require port of the source object is triggered. Links can be implemented as remote links that will operate between two DOME processes by means of the NodeConnections or as local links that will simply call process-local service functions.

2.3 Class Objects

Class objects are the user-defined components of a DOME process. These objects will provide several ports and user-defined methods. Communication channels between class objects will be established by links that connect the object’s I/O ports. The main logic of a DOME application is embedded in the class objects and the communication structure between them. Ports are used as an interface to communicate with other class objects. The communication between ports can be either synchronous or asynchronous, depending on the port definition.

2.4 Groups

Groups represent the logical execution contexts, e.g. implemented as a thread. They contain, administrate and execute child objects like local links and class objects depending on their activation.

2.5 NodeConnections

NodeConnections represent the connections between DOME processes (see Figure 2). They are used frequently by the DOME runtime to exchange process information or for controlling purposes. All links that have been established between objects of two processes will also use the NodeConnection of these processes.

The debugger application will distribute the debug mode state also through NodeConnections by sending opcodes to all processes that are part of the DOME application. Therefore it is also implemented as a valid DOME process.

2.6 LogConnections

LogConnections can be established between a DOME LogConnectionServer and a LogConnectionClient. In general these classes are used for logging tasks but they also provide some features that are well-suited for the debugger. Namely a LogConnection can be used to trigger ports without the need of established links.

3. PRINCIPLES OF DISTRIBUTED DEBUGGING

3.1 The Snapshot-Algorithm of Chandy and Lamport

As stated in CoDK [2002] and TavS [2003] this algorithm can be used to retrieve global states of an application. Therefore an amount of states of the processes and transmission channels (called snapshot) is recorded. Special marker messages are transferred to control the snapshot recording. The global states are represented by consistent cuts of the execution history while the algorithm itself only records local states that are sorted by their synchronised execution times. To ensure such synchronisations the
concept of logical clocks and the associated "happened-before-relation" can be used as described in TavS [2003] and CoDK [2002]. In addition researches about clock synchronisations can be found in Kopetz [1997].

The snapshot algorithm postulates the following requirements:

1. Errors will occur neither for the processes nor for the transmission channels. Every sent communication message will reach its target in a finite time.
2. Transmission channels are unidirectional and support FIFO message processing.
3. The graph of the processes and transmission channels has to be a complete graph. Every pair of processes is connected by a communication path.
4. Every process is able to initiate the recording of a global snapshot.
5. Process execution and communication can be continued while a snapshot is recorded.

Nevertheless, DOME and other distributed applications consist of a graph of processes which is not a complete graph in the majority of cases. Therefore the snapshot algorithm is not suitable for such distributed applications. Furthermore, this algorithm does not include the real capture of global states but only the recording of local process and channel states. To evaluate global states, an additional central gathering process is necessary.

For the purpose of a distributed debugger, errors may occur and have to be detected. Therefore a monitoring of the global state should be possible even if a failure emerges or processes become inoperative. In addition, processes do not need to know anything about global application states if they are not debugged. Therefore an integrated snapshot initiation algorithm will produce unnecessary overhead inside distributed processes.

3.2 Existing Approaches

As mentioned above, the "Snapshot-Algorithm" can be used to record local process states and the states of the transmission channels. The sum of all these sub-states can be interpreted as a global application state.

While debugging distributed applications, the global state is the most important factor to understand the behaviour of the program execution. Therefore the recording and monitoring of global states are the central points for the debugging process.

CoDK [2002] gives an example on how to implement distributed debugging by recording global states that are created by the "Snapshot-Algorithm". Therefore two concepts are introduced:

1. Possibly $\phi$
2. Definitely $\phi$

$\phi$ is a global state predicate. The main goal of the described debugging approach is to detect cases where $\phi$ is possibly true or definitely true. For that reason all processes send their states to a central monitoring process in specific time intervals in the form of so-called state messages. The monitoring process will store those states in separate queues and will extract valid global states by detecting consistent cuts of the state histories.

Subsequently the monitoring process evaluates possibly $\phi$ and definitely $\phi$ by processing the hierarchy of reachable states beginning at the initial state.

This approach is quite expensive in the context of program speed and memory usage. The costs are stated as $O(kN)$ for the amount of comparisons as well as for the memory usage ($k$ is the maximum amount of events per process and $N$ the total amount of processes). Furthermore, the approach deals with specific state predicates while a universal usage is desired in this paper.

As the distributed debugging should also work for the DOME environment, the existing approach is not qualified. There is no fixed distributed system with suitable state predicates. Therefore a debugger for DOME and other distributed applications must be able to work for any applications based on such environments. In addition, the mentioned overhead is too large with regard to complex distributed systems that consist of a couple of processes.

4. REQUIREMENTS FOR DEBUGGING DISTRIBUTED SYSTEMS

4.1 Analysis

In contrast to normal debuggers, additional issues exist that have to be analysed and worked out and are summarised in Schneckenhans [2010].

One of these issues is the non-determinism of the transfer times between processes and the local process state alterations from the view of the whole application. Furthermore, the logging of the process and object communication is a central point to provide debugging information to the user.

The debugging of the process communication should not require any instrumented code. Therefore it will be possible to design other debuggers that will also work with the DOME environment without the need of additional instrumented code for the debugger.

The application execution should be influenceable in order to enable the user to react on triggered breakpoints or allow step-by-step program execution.

A debugger for a distributed application basically has to debug all the sub-processes like a debugger for non-distributed applications and additionally control the communication between these processes. Extended features like "port trigger detection" and "central information gathering" are also required.

4.2 Features of normal debuggers

Debuggers for normal (non-distributed) applications usually provide the following features:

1. A debug function that is executed prior to any statement
2. Possibility to browse any variable and change its value as appropriate
3. Attach and detach breakpoints to code lines of functions
(4) Enable and disable attached breakpoints
(5) Step through the program

The debugging approach for distributed applications shall at least provide options 1, 4 and 5. In addition option 3 is required for O-ports instead of code lines or functions. The debug function mentioned in option 1 is needed as a special O-port call that will provide information about the next O-port (which can be interpreted as a statement).

Option 2 is approximately required for the parameters of the called O-ports. However it is not possible to change the parameter values in such a message. This may be part of further research.

4.3 Requirements and constraints

The debugger for distributed systems should be developed to work with the DOME system as described above. Therefore components of the DOME run-time and specific compilers have to be extended or redesigned as well. Basic debugging features like breakpoints and step-by-step execution are self-evident and are also to be implemented along with the debugger application.

Debuggers should be designed in a way that they do not influence the debugged application too much. For that reason the network usage and process interruption must be reduced to a minimal level.

Of course distributed applications shall work as fast as usual if they are not debugged. Therefore a relevant demand is to avoid additional overhead if the debug mode is not active and to minimise the overhead while debugging.

Moreover it would be preferable if existing DOME applications and modules would be able to work with the new debugging system without rewriting or editing them. Nevertheless the recompilation of the target is sufficient.

An additional requirement is the portable implementation of the debugging system to support different operating systems. Furthermore the approach should be adaptable to other distributed systems, following the SOA paradigm.

5. DESIGN CONCEPT TO DEBUG DISTRIBUTED APPLICATIONS

While debugging distributed systems, the current state of the whole application at a certain time is of interest. For the reason that the states of the sub-processes are able to change at any time it is important to ensure a kind of synchronisation. However the application by itself should only be influenced in a limited way by the debugger. Therefore design goal is not to synchronise each process directly by the debugger but synchronise the gathering of the states instead.

To get information about the current state of a process it has to be passed to the debugger. In this context for a distributed application, the triggered O-ports play a major role because they execute service routines of other objects and cover a large range of the application’s program logic. Therefore it is important to notify the debugger about triggered O-ports.

In principle there are two approaches to detect those triggered ports.

(1) The debugger asks all processes permanently
(2) The process or the object informs the debugger

The essential advantage of the second option is that the network is not unnecessarily loaded and so the application and other network nodes will be less affected by the debugger. The object informs the debugger only once per called O-port, namely immediately before the port is really triggered.

Port-based breakpoints are also easy to realise with this approach. When the debugger is informed about the call of an O-port, it is able to check if a breakpoint is associated to this port. If this is the case the debugger can detect that the port is about to be triggered and leave it to the user to continue the port execution at an arbitrary time. This is possible because the debugger only receives the information that the port should be triggered, however the port itself is not really executed at this time.

In order to allow the user to explicitly determine the time of a port call (e.g. after a breakpoint was triggered), the port must not be executed right after the port call notification was sent. There are also two approaches for this behaviour.

(1) The debugger sends a STOP signal to all processes to avoid triggering further ports
(2) The object interrupts the port execution till a CONT signal from the debugger is received

The preferred approach is the second option. Objects will wait for the CONT signal (continue) after the port information was sent to the debugger. Up to that point they will not execute any port. The waiting time does not play a decisive role in this case, because the object will act as desired: If there is no CONT signal, the object will interrupt its execution and when the CONT signal is received the object will continue in a normal way until the next O-port is called. A CONT signal will only be sent to exactly one class object after it informs the debugger that a require port should be invoked.

Moreover the synchronisation of processes can be handled adequately by the CONT approach. If a breakpoint is triggered by an executed port it is obvious to stop the whole application and not only the involved processes and objects. The attempt to send the STOP signal to all processes of a large distributed application is inappropriate due to the fact that there is no way to ensure an exact synchronised process stoppage. For the reason that all processes wait for a CONT signal, the debugger can just stop sending it to achieve an application stoppage. Hence it is assured that no process will continue execution when a breakpoint is triggered. From the view of the port communication, the application will be stopped in a synchronous way as no associated process will be able to execute any require port and therefore change the application state from the moment on when the last CONT signal was sent.

What will happen if the execution is continued? This is the case when a CONT signal waiting for it is sent to the objects of the related processes by the debugger. These
signals will not be synchronous, because they are sent by request. However this circumstance is not relevant for a distributed application. Delays will arise for all process communication anyway, so the additional delays caused by the CONT signal are insignificant or rather will only change the waiting time between two processes.

Synchronization of the stop signal is important because the application state at the stoppage time is of interest. For CONT signals this is not critical for the reason that the time of the CONT signal arrival is unimportant, because the process states are not be changed while the CONT signal is received. For that reason each process, that receives CONT signals will not be synchronous, because they are sent by request. However this circumstance is not relevant for a distributed application. Delays will arise for all process communication anyway, so the additional delays caused by the CONT signal are insignificant or rather will only change the waiting time between two processes.

Unfortunately the CONT-approach also implies a problem: If there is no debugger or a process is not debugged by it, no CONT signal will be sent and all processes and objects will be frozen. Therefore the objects have to know if they are currently debugged or not and act differently in these cases. If the debug mode is not active, each object will not wait for a CONT signal but will call require ports immediately and omit the sending of the port information to the debugger instead. Therefore the additional overhead for O-port calls is reduced to a bool-flag check when the debug mode is not active. This way the indirection over the network becomes unnecessary and the object is able to act in a normal independent way. The O-port is simply called directly without generating and sending debug information.

To enable this behaviour the processes and their objects have to be informed if they are in debug mode or not. This task will be handled by the debugger since it has to know all processes that should be debugged anyway. The processes in turn know their objects and can forward the debug mode state to them. In initial state the debug mode is disabled but the debugger is able to enable or disable it as requested. When the debugger is terminated the debug mode is disabled for all processes automatically to ensure normal application behaviour.

The debug mode effects the whole distributed application instead of only single objects or processes. For this reason it is necessary to ensure that the debug mode is activated or deactivated for all processes that are part of the application. Any process that is connected to another process of the application is also a part of it.

Usually the debug mode is started only for one process. The process will then distribute the debug mode further to all processes that it is connected to. The following processes will do the same to distribute the debug mode finally to the whole application. This concept was developed under the name distributed debug mode state.

The considerations arise that only three pieces of information have to be transferred to enable debugging:

1. CONT signal
2. debug mode state
3. port information

The CONT signal is sent to the object which waits for it by the debugger, the port info on the other hand is sent from objects to the debugger every-time a require port should be executed. The debug mode state is distributed

### 6. THE IMPLEMENTED DEBUGGER

An important part of the debugger’s tasks is the starting and the stopping of the debug mode for a distributed application. Mainly the distribution message of the debug mode also contains the information about the debugger which initiated the debug mode change. In this way each process of the distributed application is able to establish a connection to the debugger if its debug mode is activated. This is shown in Figure 5.
Beyond the run-time of the considered distributed system of DOME, also the class objects have been extended to process the CONT signal and block require port calls until it is received. The following methods have been added to the class objects:

(1) virtual void __debug_send_info(dome::string host, dome::string process, dome::string object, dome::string port, dome::string port_info, dome::uint64 timestamp);

This method is actually a require port function that will be generated for every DOME class. It is implemented as a virtual method in the base class of all DOME classes (class class object) so that the concept of polymorphism can be used ([Stro [2000]]). This basic implementation will just throw an exception as it makes no sense to use objects of this base class.

(2) void require_dispatch_cont();

This method will be automatically called by each generated CONT port of a DOME class. In contrast to the above method, no polymorphism is necessary as the CONT functionality is always the same for all DOME classes. Namely a conditional object is notified. For further information about conditional objects see POSIX (peond [2008]) or Windows (MSDN [2010]).

Moreover the existing method dispatcher for the required port has been extended. This method is called when a require port is invoked. Therefore the following things have been added to that method:

(1) First a boolean flag is checked to determine if the debug mode is active or not. Only if the object is debugged are the following steps executed.

(2) The SendInfo signal will be sent before the require port is really executed. Therefore the port information is gathered and encoded for the network transferring. Finally it is sent to the debugger.

(3) After port information sending, the method dispatcher will wait for the conditional object. Only after the condition was triggered, will the method dispatcher continue its execution.

(4) At the end the require port is executed as in normal mode. This means that the connected service port of the target object is invoked with the actual port arguments.

7. CONCLUSION

The developed approach was validated in real applications, once controlling a robot and once controlling a manufacturing demo plant, consisting of four collaborating modules.

The developed debugger application was designed in a way that it fits to the functionality of normal debuggers as far as possible. For that reason features such as breakpoints, step-by-step execution and visualisation of the next executed step were implemented as well. In addition, the GUI provides several elements to control and visualise the debugging process.

All components of the debugging system, including the debugger GUI application, were developed in a portable way so that they will be applicable on all operating systems that are supported by the DOME environment.

Moreover all given requirements and constraints were met and verified. The costs of the debugging have been determined as well in order to allow an estimation of the capability in specific fields of application and the considered environment itself.

Additionally, the processes have to be debugged locally to allow the debugging of all parts of a distributed application. This purpose is planned to be realised in future work that will be based on the developed approach.

REFERENCES


