This paper was published in
Lecture Notes in Computer Science
(http://www.springer.de/comp/lncs/index.html)

The copyright for this contribution is held by
Springer-Verlag (http://www.springer.de)

Using Monitoring Techniques to Support the Cooperation of Software Components*

Roland Wismüller

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Technische Universität München, D-80290 München, wismuell@in.tum.de

Abstract. Many applications in computing could be implemented by re-using existing programs, if only it were possible to easily make them interact with each other. MPI or PVM do not support these applications, as inserting a direct communication makes the components depend on each other. Even middleware for distributed computing still requires that the components already implement the necessary interfaces and that these are sufficient for the required interactions. The paper shows how on-line monitoring techniques can be used to improve this situation. An extended version of the monitoring system OCM allows both to incorporate the necessary interfaces and to define interactions between components at run-time, without having to modify their source code.

1 Introduction

In parallel applications, several components (usually processes) tightly interact in order to perform a common computation. In distributed applications, on the other hand, client components use common services that are provided by usually remote server components. In both scenarios, the components need an infrastructure supporting an efficient communication. Such infrastructures have been available for quite some time with communication libraries like MPI and PVM or distributed middleware, such as Corba.

There is, however, an important scenario, which is not adequately supported by these infrastructures. Imagine a set of independently developed software components, which normally are used as stand-alone applications. In some cases, however, it may be desirable to use them in a cooperative manner in order to exploit a synergy. Examples for this situation are so called multi-physics codes [3]. E.g. to simulate air pollution, a climate code, which models air and water transport, can be loosely coupled to an atmospheric chemistry code, which accounts for chemical reactions. Other examples, which have been the motivation for this research, are tool environments to monitor and control program executions. E.g. a visualizer shows a coarse grained overview of an execution, while a debugger can provide ample details about a specific program state. However, the states shown by the visualizer are already in the past, while the debugger

* This work was partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1, and Research Grant WI-1346/1-1.

can only examine a program's current state. Combining these tools with a checkpointing system allows to transparently restore the execution to a state selected in the visualizer, which then can be examined in detail [9].

The distinctive properties of the scenario outlined in the previous paragraph are the following:

1. The components have been developed independently, usually by different institutes or companies.
2. The components are designed to be used stand-alone. They are not prepared for any cooperation with other components.
3. The source code of these components often is not available for modification. Even if it is, modifications are often undesirable for reasons discussed later.
4. The intended cooperation requires only infrequent communication with a comparatively low data volume.

When these properties are met, we will speak of *interoperating components* in the remainder of this paper.

In the following, we outline the requirements of applications based on interoperating components and show that they are not met by existing communication libraries and middleware. Section 4 presents a solution that emerged from our research in the area of monitoring tools. We propose to use on-line monitoring techniques in order to implement the cooperation between software components. In this approach, the monitoring system plays the role of an extended, event oriented middleware. Section 4 also provides some details on the implementation of the approach in the framework of the OCM project [10].

2 Example: An Environment for Efficient Debugging

To provide the following discussions with a concrete background, we will use the tool environment already mentioned in the introduction. The environment consists of three independently developed run-time tools for parallel programs based on PVM: a program flow visualizer, a debugger, and a checkpointing system[1] [9]. The visualizer VISTOP animates the program's execution behavior by displaying a sequence of snapshots. Each snapshot shows a global view of the program's processes and their interactions, i.e. communication and synchronization, as well as other global state information, e.g. the contents of PVM groups. The debugger DETOP allows to examine and modify the current state of all or a selectable subset of processes. The checkpointing system CoCheck finally can save and restore the complete global state of the program.

By using all three tools concurrently, a feature strongly required by users could easily be implemented: When a suspicious state is found in the high level view of the program execution presented by the visualizer, users want to be able to examine this state more closely with the debugger. This is not possible

[1] The real tool environment actually contains a fourth tool – a deterministic execution controller – which is not discussed here for simplicity.

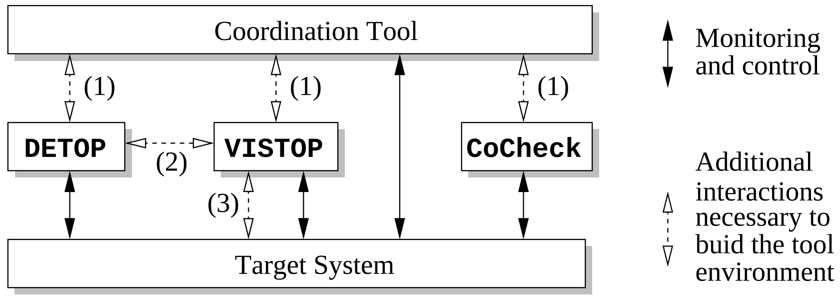

Fig. 1. Component architecture of the example environment

without a cooperation between all three tools, since the visualizer shows the history of the program's execution, i.e. the state in question has already passed. In order to implement this feature in the tool environment, the target program is checkpointed periodically. When the user wants to inspect a selected state more closely, the program is first reset to the most recent checkpoint preceding this state. Next, it is automatically re-executed from this point on until it reaches the target state, which then can be examined using the debugger.

Such a concurrent, cooperative use of run-time tools is only possible if (1) there are no conflicts between these tools, e.g. due to resources that can only be used exclusively [9], and (2) the tools can interact with each other and can be controlled by a higher-level coordination tool. We will only discuss the second issue in this paper.

Figure 1 indicates the interactions necessary to realize the feature outlined above:

(1) Certain actions of DETOP, VISTOP, and CoCheck need to be invoked by the coordination tool in order to reset the target program to the desired state. E.g. the coordination tool must be able to get information on the state currently displayed by VISTOP, to restore the proper checkpoint, or to print a notice in DETOP's output window.
(2) When certain event occur in a tool, actions in other tools must be invoked. For example, when VISTOP shows a global state that is in the past, DETOP should inactivate its state display, since it shows information on the current state, which is inconsistent with the one presented by VISTOP.
(3) Finally, the tools need to execute additional actions when certain events occur in the target program. E.g. when a target process is stopped by DETOP, VISTOP must update its displays in order to provide consistent information.

These classes of interactions are also representative for other applications based on interoperating components. While for the first two classes this is rather obvious, the third one seems to be specific to interoperating run-time tools. However, even in other applications the components often should react on events in their execution environment, e.g. node failure, low memory, etc.

3 Requirements of Interoperating Components

In the scenario of interoperating components, each component is implemented by a separate processes or a set of processes. Thus, the required interactions between these components must be implemented by some kind of message exchange. The attempt to use MPI or PVM for this message passing, however, results in severe problems:

- First of all, explicit send and receive calls must be inserted into the components. This means that the source code must be modified and the components need to be recompiled. Furthermore, the developers of the components should be convinced to incorporate these changes into their 'official' version. Otherwise, the changes must be re-applied to each new version.
  Developers are, however, rather reluctant to include such modifications, especially, since there may be different, possibly conflicting requests from users applying the component in different environments.
- Adding explicit communication also makes the components depend on each other [8]. For example, in the environment outlined in Section 2, VISTOP could send a message to DETOP asking it to inactivate its state display. This would lead to the following consequences: VISTOP needs to know that a DETOP process is running and must find its task ID or rank in order to send a message to it. When used in a different environment, VISTOP needs to locate and notify different components.
  Furthermore, VISTOP must know the message format expected by these components (or vice versa, they must know the format defined by VISTOP). If this interface is modified in one of the components, VISTOP has to be adapted accordingly.
  Without special measures, the components even are no longer usable in a stand-alone fashion, since e.g. the sending component will be blocked or buffers will overflow when there is no receiver. Such dependencies are usually unacceptable to component developers.

The situation is not much better when using object oriented middleware, like e.g. Corba. The components still have to be modified, unless they already provide Corba interfaces for *all* required interactions. In addition, the component acting as a client still depends on the component acting as a server.

A general principle in computer science to break dependencies is the use of indirection. In our example, instead of letting VISTOP send the notification directly to DETOP, it could always send it to a special *broadcast message server* (BMS). Components interested in this type of notification have to register with the BMS, which then forwards the proper messages to them. Thus, the components no longer need to know which other components exist in the current environment. This scheme might be implemented using PVM broadcasts and dynamic groups. A more specific BMS implementation is ToolTalk [5].

There is, however, still a dependency, as all the components need to agree on common message types and formats. This dependency can be removed by using *mediators* [8]. A mediator is an object that monitors events in some components.

When an event is detected, the mediator executes an associated piece of code that typically invokes actions in other components. The mediator is free to adjust the parameters when invoking the action; it can also call a sequence of actions in potentially different components. In general, a mediator implements an arbitrary behavioral relationship between two ore more components. While the mediator – of course – depends on all these components, the components themselves now are fully independent.

All current implementations of mediators, like e.g. the Corba-based MediatorService [7], still require the components to explicitly provide interfaces for events and actions. As a consequence, all components depend on the concrete implementation of the mediator approach and can no longer used purely standalone. E.g. components using the MediatorService cannot be used outside a Corba environment any more. Another drawback of current implementations is that we still need to modify the components' source code, if the interfaces are not yet available or are incomplete w.r.t. our needs. Sullivan, who originally invented the idea of mediators, states that "[anticipating] *the events needed in an object's interface* [...] *is unreasonably hard*" ([8], Section 8.10). This problem, however, only occurs if we view components as static, unchangeable entities. Modern monitoring techniques allow to dynamically modify programs during their execution. Thus, the idea is to insert the required component interfaces dynamically during run-time, as needed. In this way, we avoid source code modifications and can even use legacy applications as components. Stand-alone usage is no problem at all, since the static executable remains unchanged. Dependencies to other components or the infrastructure (like ToolTalk or Corba) are completely avoided.

4 The Monitoring System as Communication Middleware

If we examine the operation of a modern, distributed monitoring system like the OCM [10] more closely, we even see that its operation is already very similar to a mediator: the monitoring system waits for events in the target program and executes a set of actions when the event occurs. In addition, the OCM supports run-time programmable relations between the events and lists of actions to be executed. It transparently accounts for the distributedness of the target system by automatically routing event information and requests for actions to the proper nodes in the system. Thus, the OCM not only enables the dynamic insertion of component interfaces, but also supports the definition of behavioral relations between components that can be adapted on demand.

These features lead to the following idea: if the OCM attaches to all interoperating components in the same way as it attaches to its regular target processes, it can realize the mediator concept without a need to specifically prepare or adapt the components. There are, however, two requirements, which must be met before this idea can be realized: (1) the OCM must be able to detect all events of interest in the components, and (2) it must be able to invoke actions, which are executed inside other components.

4.1 Detection of Events

The OCM already can detect certain classes of events in its target processes without a need for an explicit source code instrumentation. These classes include the beginning and end of function or subroutine invocations in a target process. Currently, two different implementations are available: The first one is based on binary wrapping [2]. It requires to relink the target program, since it modifies the object code of programming libraries. Thus, this technique is mainly suited to monitor library calls. A recently finished second implementation uses the dyninstAPI [4] that allows to dynamically insert code into running processes.

In both cases, the inserted code notifies the OCM about the event. It first writes the relevant event information into a shared memory segment and then executes a trap instruction. This results in a signal being sent to the OCM. The event information usually consists of the values of parameters or the result of the monitored subroutine, but it can also include the values of global variables or other state information. The event information to be acquired for different subroutines is specified in a definition file – in analogy to an interface definition file used e.g. with Corba – and thus can easily be adapted if necessary.

When the OCM receives the trap signal, it reads the current event information from the shared memory segment and examines the currently defined event/action relations. If necessary, it sends messages to remote OCM processes requesting the execution of actions on their respective nodes.

4.2 Execution of Actions

In the OCM's normal use, actions are executed directly by the OCM processes. In order to support interoperating components, we have extended the OCM with means allowing to execute actions within a target process. Like in the regular case, the interface of such an action is specified in a special interface definition file, while the action itself is implemented as a piece of C or C++ code. Similar to standard middleware environments, an IDL compiler generates a stub routine from this file, which adapts the internal generic action interface of the OCM to the specific C/C++ interface of the action's implementation.

When such an action needs to be invoked in the context of a target process, two steps must be executed:

1. If not already done before, the stub routine and the code of the action's implementation must be loaded into the target process's address space.
2. The target process must be forced to call this stub routine and, thus, the action.

Step 1 is achieved by using dynamic linking techniques. The stub and the action's code are compiled into a dynamically linkable library. The target process then is forced to execute a call to the `dlopen` routine in the UNIX run-time system, using the same technique as described for step 2 below. This routine loads the library into the target process. Since the dynamic linker resolves symbol references in both directions, the action's implementation can access any public symbol in the target process, i.e. it can read and write global variables and can call subroutines in the target process's code. Thus, the action's implementation code usually just calls such an already existing subroutine, passing on its own parameters to it.

Step 2 is realized by manipulating the stack and the program counter of the target process using operating system services. The dyninstAPI already offers a routine for this so-called inferior RPC. A technical problem is that the inferior RPC is performed completely asynchronously w.r.t. the target process's execution. This can result in problems, if e.g. the inferior RPC interrupts the target process in a routine that is later called again by the invoked action. Most routines behave incorrectly when reentered in such a way. Thus, in the OCM the simple inferior RPC has been extended towards a synchronized scheme: when an action has to be invoked, the OCM installs a local event/action relation in the target process, where the event signals that the execution reached a point where it is safe to call the action.

4.3 Support for Interoperating Components

Due to the dynamic instrumentation of subroutine invocations supported by the OCM, events need not be explicitly announced by components, as long as they are related to the beginning or end of such an invocation. The experience with the tool environment presented in Section 2 suggests that this regularly is the case with reasonable well structured codes. While technically, dynamic instrumentation also enables to monitor the execution of arbitrary points in a program's code, the problem is to identify and to name them in a meaningful way.

The OCM's implementation of actions allows to invoke any subroutine in a component from the outside, possibly adapting its interface as needed. In addition, completely new actions can be added to a component. The component's source code does not need to be modified in any of these two cases. Thus, neither a dependency to other components nor to the OCM infrastructure is created.

Since the OCM loads the information on available events and actions at run-time, it can be adapted to different applications without a need to alter its code. Via the OCM's monitoring interface, a coordination "tool" can define the necessary relations between events and actions in the components of a specific application at run-time. In addition, the coordination tool can directly invoke actions in the components via this interface.

4.4 Performance

The OCM's enhanced flexibility asks for some price to be paid in terms of performance. On a Sun Sparc Ultra-10 running Solaris 2.6, the time between the occurrence of an event in a component and the execution of the associated action in another one[2] currently amounts to $1.7ms$. This compares to $0.33ms$ needed

[2] In order to enable the use of high resolution timers, both components have been located on the same machine.

by an equivalent standard PVM communication between two processes on the same node. The higher overhead mainly comes from two sources:

1. While with PVM a communication only requires two hops when the processes are located on the same node, the OCM always needs four: (1) The event in the component is notified to the local OCM process via a signal. (2) A message is sent to the OCM's routing process. (3) The message is forwarded to the local OCM process on the destination node. (4) The action is invoked in the destination component using an inferior RPC. This communication accounts for about half of the total time.
2. The other half is caused by the management of event/action relations and the marshaling and unmarshaling of the communicated data structures, which are more complex than in an equivalent PVM communication.

We are still in the process of optimizing the OCM. Since in the scenario of interoperating components, communication is typically of low-volume and infrequent, this is, however, not of crucial importance in practice.

5 Related Work

We have already discussed the relation of our work to communication libraries, middleware and broadcast message servers like ToolTalk in Section 3. The approach most closely related to our work is the idea of *mediators* [8]. We have extended this idea with dynamic code modification techniques, which allow to completely avoid dependencies between components. This extension is the distinctive feature of our work.

The *interactor* approach [6] is a different extension of the mediator idea, which addresses the case where components can also call external services rather than just creating events. In our OCM-based environment, this is possible, too. Just like the coordination tool, any other component can use the OCM's monitoring interface to call remote actions. However, since we are concerned about components that normally are used stand-alone, we do not need this feature.

The goal of coupling independent components finally is addressed by a large number of *coordination languages* [1], which allow to uniformly specify the interfaces and interactions of distributed components. These approaches are orthogonal to the one proposed here: while current implementations of coordination languages still require the interfaces to be statically compiled into the components, the OCM currently requires three different specifications for events, actions and their relationships, respectively. Thus, the integration of our approach with a coordination language will be an important issue addressed in future work.

6 Conclusions

The paper has shown that the interoperability between independently developed components is not adequately supported by existing middleware. In order to avoid undesired component dependencies, it is necessary to define the required interfaces and interactions separate from the components' code and to link them to the components only at run-time. We have shown that this is possible by using on-line monitoring techniques. Furthermore, the monitoring system can play the role of the communication middleware, since it already implements the necessary features. We have demonstrated the usability of this approach by composing a tool environment for advanced parallel debugging from existing tools. However, the approach can support any kind of loosely coupled components, like e.g. multi-physics applications or distributed simulations.

References

1. F. Arbab, P. Ciancarini, and C. Hankin. Coordination Languages for Parallel Programming. *Parallel Computing*, 24(7):989–1004, 1998.
2. J. Cargille and B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17–18, June 1992.
3. T. J. Downar, R. Eigenmann, J. A. B. Fortes, and N. H. Kapadia. Issues and Approaches in Parallel Multi-Component and Multi-Physics Simulations. In H. R. Arabnia, editor, *Intl. Conf. on Parallel and Distributed Processing Techniques*, volume II, pages 916–922, Las Vegas, NV, USA, June 1999. CSREA Press.
4. J. K. Hollingsworth and B. Buck. *DyninstAPI Programmers' Guide, Release 1.2.* Univ. of Maryland, College Park, MD, USA, Sept. 1998. `http://www.cs.umd.edu/projects/dyninstAPI/dyninstProgGuide.pdf`.
5. A. M. Julienne and B. Holtz. *ToolTalk & Open Protocols: Inter-Application Communication.* SunSoft Press, 1994. ISBN 0-13-031055-7.
6. G. Paul, K. Sattler, and M. Endig. An Integration Framework for Open Tool Environments. In H. König, K. Geihs, and T. Preuß, editors, *Distributed Applications and Interoperable Systems*, pages 193–200, Cottbus, Germany, Sept. 1997. IFIP, Chapman & Hall. `http://wwwiti.cs.uni-magdeburg.de/~sattler/papers/dais97.ps`.
7. G. Paul and K.-U. Sattler. MediatorService - Integration von verteilten Objekten durch Beschreibung der Interaktionen. In *Proc. Int. Workshop Trends in Distributed Systems*, Aachen, Oct. 1996. `http://wwwiti.cs.uni-magdeburg.de/~sattler/papers/trevs.ps`.
8. K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems.* PhD thesis, Dept. of Computer Sciences and Engineering, Univ. of Washington, USA, 1994. Techn. Bericht 94-08-01. `ftp://ftp.cs.washington.edu/tr/1994/08/UW-CSE-94-08-01.PS.Z`.
9. R. Wismüller and T. Ludwig. Interoperable Run-Time Tools for Distributed Systems - A Case Study. In H. R. Arabnia, editor, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Application, PDPTA'99, Volume IV*, pages 1763–1769, Las Vegas, NV, USA, June 1999. CSREA Press.
10. R. Wismüller, J. Trinitis, and T. Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, pages 1–9, Welches, OR, USA, Aug. 1998. ACM Press. `http://wwwbode.in.tum.de/~wismuell/pub/spdt98.ps.gz`.