CORAL - Online Monitoring in Distributed Applications: Issues and Solutions

IVAN ZORAJA, IVAN ZULIM, and MAJA ŠTULA
Department of Electronics and Computer Science
FESB - University of Split
R. Boškovića b.b., 21000 Split, CROATIA
{zoraja|zulim|kiki}@fesb.hr
www.fesb.hr

Abstract: - In this paper we describe and evaluate issues that come up in the development of online monitoring systems which connect software tools to a running distributed application. Our primary intension was to elaborate how to deal with complex middleware mechanisms that cater for the middleware functionality in a way transparent to the users and tools. Our current implementation, called Coral, manages DSM mechanisms that provide an abstraction of shared memory on loosely coupled hardware, and allows multiple tools to perform consistent yet efficient operations on the entities being monitored. Since our primary design choice with Coral was portability we will port Coral to distributed environments based on the SOA technology.

Key-Words: - Online Monitoring, DSM, Tools, Process migration, Performance analysis, Checkpointing

1 Introduction

Rapid technology transitions in conjunction with growing complexity of distributed applications make software development in such environments increasingly difficult mostly because the applications have become more parallel, distributed, and heterogeneous. In addition, such classes of applications provide complex usages scenarios and ways of interactivity.

While distributed applications are widely used in many business domains, sophisticated *tools* that are capable of observing and manipulating running distributed applications in a transparent yet consistent way have not been developed. This is primarily caused by the competitive pressures imposed on vendors which ship new systems prematurely without an appropriate tool support. The vendors usually prioritize development of new technologies and programming models, and develop either rudimentary tools or tools with limited functionality.

Online monitoring refers to a set of techniques and mechanisms needed to control the system being monitored, hence allowing software tools to gather information from the system as well as to manipulate its runtime behavior. It is contrasted to *off-line* monitoring [5] where the manipulation capability fails and the observation is done after the application finishes the execution.

Currently, online monitoring systems that can fully support parallel tools such as OMIS [2], OCM [3] and OCM-G [11] are implemented for the message passing programming paradigm and can not, without modifications, be used for distributed applications based on other paradigms. For example, they can not transparently manage the underlying mechanisms that provide an abstraction of shared memory in software for distributed applications built on the DSM (Distributed Shared Memory) middleware.

In this paper, we present our online monitoring system called Coral [1] [8]. It manipulates DSM applications thereby providing an abstraction of shared memory to the parallel tools via a transparent management of underlying mechanisms that cater for the DSM functionality. Particular emphasis is placed on the management of complex interaction patterns among DSM processes during process migration and checkpointing. We validate our implementation in terms of functionality, effectiveness, architecture and portability, and provide an insight into how Coral can be reused to support SOA-based applications.

2 Distributed Middleware

Distributed middleware systems are used to simplify parallel and distributed computing on diverse and heterogeneous computing platforms by reducing the complexity on benefit of both the application users and developers.

In the *message passing* programming paradigm, processes reside in different address spaces and communicate by explicitly sending messages to one another using two primitives *send* and *receive*, with several parameters that specify peers and interaction semantics. The message passing libraries such as PVM [7] and MPI are built for dedicated distribute memory machines and workstation clusters.

Distributed shared memory denotes both computing systems that provide an abstraction of shared memory on loosely coupled hardware. The DSM paradigm is aimed at hiding complex communication patterns provided by the message passing paradigm from the developer and, at the same time, reducing overall memory access latencies. The DSM libraries such as IVY, Orca, and TreadMarks [4] provide primitives in a conventional shared memory style for the allocation and release of shared memory as well as for the synchronization and coordination among shared accesses.

Distributed Object Computing (DOC) integrates object-orientation and the DSM paradigm providing functionality by means of *services* offered by *servers* or *containers*. Services are described by defining interfaces using an interface definition language such as IDL and WSDL. The basic communication mechanisms in DOC-s can be viewed as an interaction between the *proxy* and the *broker* patterns. Typical representatives in this category are: CORBA, .NET Remoting, EJB, Globus, and SOA-based approaches such as WCF and SCA.

3 CORAL Requirements

Monitoring systems can be implemented at various levels usually combining hardware and software solutions.

With reference to Fig 1, a software-based monitoring environment embraces a group of external actors that via a set of monitoring *actions*, provided by the monitoring system, can interact with one another. Via requests from tools, monitoring actions can be invoked *conditionally* and *unconditionally*. Unconditional actions are immediately executed after the request has been received while the conditional ones wait for other actors or the system being monitored to generate requested events.

Monitoring functionality is specified by a set of actions that are provided by the monitoring system and made available to multiple tools used for later phases of the software development process. Because monitoring actions can be invoked concurrently, the Coral is designed to resolve concurrency conflicts. In addition, Coral provides actions for all the tools that can observe and manipulate all the entities being monitored without deadlocks and races.

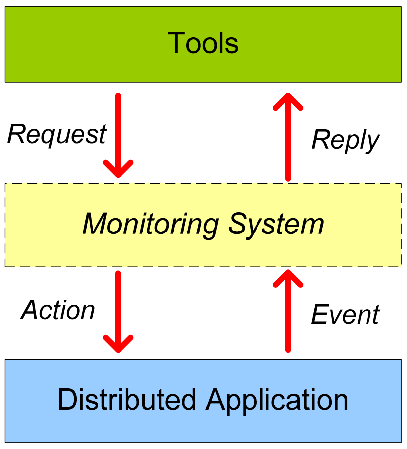

Fig.1. Online Monitoring Environment

Software tools could be able to dynamically combine monitoring actions and events and cooperate through Coral to gain improved functionality. Finally, Coral can transparently manage the underlying mechanisms that cater for the programming abstraction exposed by the middleware.

3.1 Entities and Actions

Coral can observe and manipulate the following categories of objects (*entities*) being monitored: processing nodes, processing activities, services, middleware mechanisms, and internal middleware entities.

Processing nodes can host processing activities. Active nodes represent nodes on which activities run and where monitoring components control application processes and measure load while passive nodes are without running activities and monitoring components only measure the load. Manipulation actions for those entities include adding and remove processing nodes while information actions return various hardware and OS parameters such as the length of the process running queue.

Processing activates include OS processes and threads running on active nodes. Manipulation actions for processing activities include actions for attaching and detaching to/from activities, stopping and resuming activities, migrating activities, and checkpointing and restoring activities. Information actions include information such as activity identifiers, scheduling parameters, and memory usage. Notification actions include information about changing in an activity such as activity stopped or migrated.

Services include entities that are transparently managed by the middleware. In DSM, to this category belong *shared data*, *data access routines*, and *synchronization routines*. Manipulation actions include information such modifications of shared data, information actions return information about shared data and routines, while notification actions inform about changes performed on shared data and routines. For instance, a notification can be "a particular lock has been obtained or released".

Middleware mechanisms are used to provide the transparency of the middleware to the application programmer. In DSM, to this category belong *communication* and *virtual memory* mechanisms usually implemented as handlers. Modification actions can install and remove handlers, information actions can obtain information about handlers such as frequency and number of calls, while notification actions deal with events from handlers.

Internal middleware entities refer to particular implementation aspects of the middleware that can not be accessed through its API. Coral does not support this type of entities since they can only be useful for the developers of middleware.

4 Coral Architecture

As shown in Figure 2, Coral consists of three logical parts: the *coordination component*, *local monitors*, and *intruders*. The actual monitoring code is divided between local monitors and intruders. Local monitors control activities on nodes. Intruders represent code injected into the middleware libraries and control middleware services. The interaction and coordination among monitoring activities is implemented via the coordination component.

4.1 Coral Components

The Coral coordination component (C^3) is a single process responsible for distribution and parallelism since it splits requests from tools and sends them to local monitors for further processing. Other main tasks of C^3 include enforcing consistency of issued requests, binding events to actions, gathering results from local monitors, and sending replies to the tools. The main monitoring loop waits for requests that can come up from two sources: tools and local monitors.

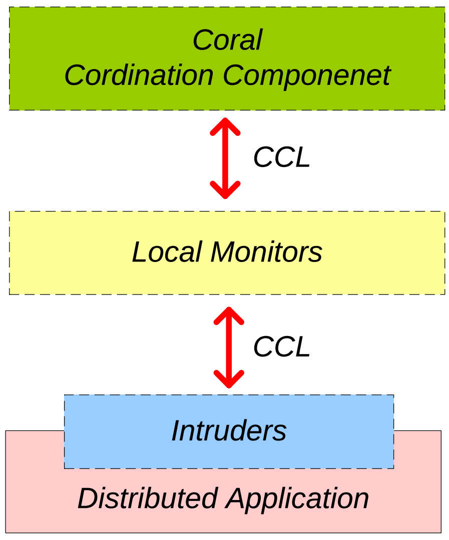

Fig. 2. Coral Monitoring Architecture

Coral local monitors are processes that run on each node being monitored and implement monitoring actions that can be applied to *processing nodes* and *processing activities*.

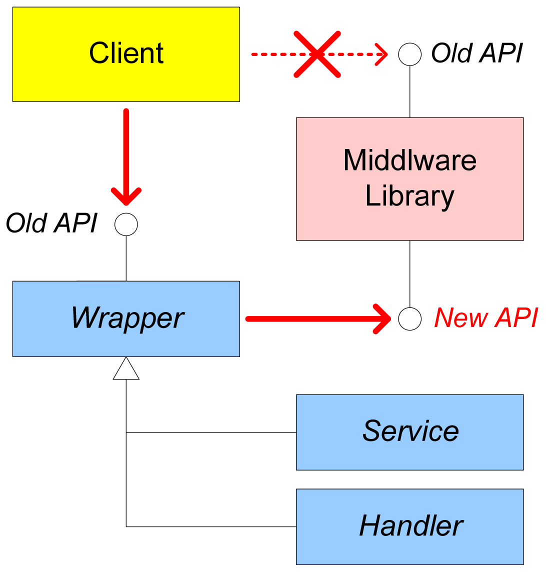

Fig. 3. Source Code Wrapping

Action to be performed on monitoring activities can not be implemented inside the process being monitored since we would not be able to detect requests and events independently of the state of the process being monitored. Some actions applied to *services* and *middleware mechanisms*, such as process migration, are spread among local monitors and intruders while the others, such as measuring performance of shared routines, are completely implemented in intruders.

Coral intruders are implemented by inserting monitoring code that intercepts client calls to the middleware. Coral wraps the DSM libraries at source code level and supports two types of wrappers: wrappers for *services* and for *middleware mechanisms*. With reference to Fig. 3, Coral wrappers provide the same interface to the clients and delegate calls to the new API added to the middleware. This approach allows monitoring code to be executed without recompiling clients – they must be linked again. We refrained from binary wrapping techniques since they produce too much name collisions for middleware of choice.

4.2 Coral Communication Library

In Coral, we do not use communication mechanisms provided by the middleware being monitored for internal monitoring communication because such sharing could be disruptive to the middleware being monitored and would potentially diminish both application and monitoring performance.

Therefore, we implemented a communication library called CCL (Coral Communication Library) which makes use of TCP/IP sockets and UNIX shared memory segments to exchange internal monitoring messages. To lesson the impact on local computing, potentially caused by execution of monitoring actions, and still preserve the effectives of the monitoring functionality, CCL supports interrupt-driven communication, buffering of early messages, and multiplexing among multiple senders.

To distinguish among different communication parties and various message types, Coral makes use of monitor and message identifiers.

5 Implementation and Testing

In this section we present implementation solutions as well as core Coral use cases and test scenarios including load management, process migration, performance measurement, visualization, checkpointing, and debugging. The potential of Coral monitoring approach has been explored using the *TreadMarks* DSM UNIX library and the application suite consisting of eight applications that accompany the library. The tested for the development and evaluations consists of a set of interconnected of Solaris machines.

5.1 Starting Monitoring

Since Coral utilizes source code wrapping and *processing nodes* for TreadMarks *processing activities* are statically specified at startup we first start the monitoring system and then the application being monitored which then attaches to the monitoring system.

The C^3 process is started manually. It reads the application configuration file and subsequently starts a local monitor on each specified node using a CCL routine. It also starts local monitors on other nodes since they will be used for load balancing purposes. Each local monitor prepares two types of communication routes: shared memory segments for communication with intruders and sockets for the communication with other local monitors. Information about both communication routes are sent back to C^3 which uses that information to connect local monitors to each others and to attach application being monitored – via intruders – to local monitors.

After the monitoring system is up and running the application being monitored can be started. Since the middleware linked with application is instrumented the intruder in the client reads information about communication routes and uses it to attach to local monitors.

After both the monitoring system and the application being monitored are started tools can be started and connected to C^3.

5.2 Load Measurement

Load management in Coral is implemented using the `/dev/kmem` file that contains an image of the kernel memory on the *processing node*. Local monitors perform the actual measurement in regular intervals and send the measured load data to the C^3 which forwards the data to the load balancer for evaluation. To economize messages, load balancers may also request direct communication over CCL to local monitors from the C^3.

5.3 Process Migration

Coral provides an even usage of computational resources via the process migration technique that takes care of (1) process states, (2) shared memory pages, (3) communication mechanisms, and (4) internal monitoring data structures about the entries being monitored. The state of single process is saved and restored utilizing the Condor [10] library.

To start the migration, C^3 sends a message to the appropriate local monitor which becomes the *migration manager*. For instance, to migrate process P_A from node N_i to node N_j, local monitor M_i

becomes the *migration manager*. M_i forwards the message to all other monitors (M_k) which wait for the application to reach *a safe point* for migration.

The safe point in Coral is reached when all synchronization routines have been completed. After the safe points in all intruders are reached, the intruder (I_i) in the process being migrated, stores the necessary information required for the resurrection while other intruders (I_k) wait in the safe point until the process is migrated. M_i then terminates P_A and sends a message to M_j to restore the process P_A in P_A' and rebuild the communication routes and shared pages. New communication points and information about shared pages are sent to all intruders I_k to update that information in their processes. After all processes are updated the application is allowed to continue execution.

In Fig 4., we show average times required to migrate the DSM mechanisms for the previously mentioned application suite varying number of processing nodes. The curve marked *Messaging* refers to the time spent to migrate communication routes and to transfer monitoring data while the curve marked *Mapping* shows times needed to migrate shared pages.

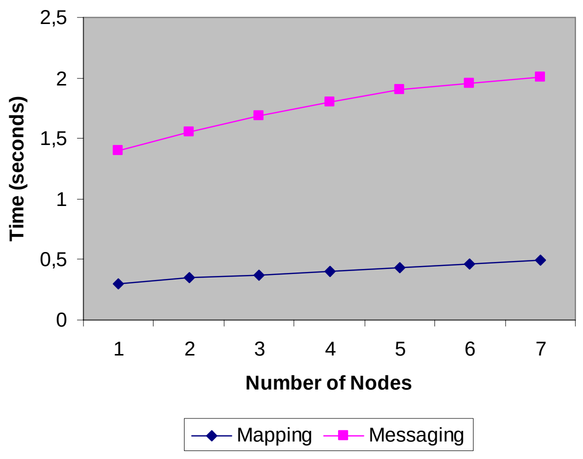

Fig. 4. Migration Times

5.4 Performance Measurement

We used the Coral measurement capabilities to find out where an application spends most of its running time. Total running time for a DSM application can be divided into a series of busy and idle intervals: t_c is time spent for useful computation to accomplish the user algorithm, t_r is time spent handling remote requests to ensure DSM coherence, t_l is time spent to detect shared accesses and handle local data misses, and t_s is time spent waiting when synchronizing. We defined the efficiency of a DSM middleware η_i as:

$$\eta_i = \frac{t_c}{t_c + t_r + t_l + t_s}$$

The application suite used to test the efficiency of DSM middleware consists of the following programs: *Barnes-Hut, FFT, QSort, IS, Gauss, SOR, TSP, and Water.* The *Barnes-Hut* is a gravitational N-body problem, the *FFT* solves partial differential equations, the *QSort* is a recursive sorting algorithm, the *IS* ranks an unsorted sequence of keys, the Gauss implements the Gaussian elimination, the *SOR* solves partial differential equations using successive over–relaxation approach, the *TSP* finds the shortest path, and finally the *Water* solves dynamic molecular solutions.

In addition to η_i, in Table 1, we show η'_i which does not take into account startup and initialization times.

Program	***η**ᵢ (%)*	***η'**ᵢ (%)*
Barnes-Hut	*85.0*	*64.4*
FFT	*57.6*	*1.89*
Quick Sort	*64.2*	*15.9*
IS	*87.2*	*67.3*
Gauss	*72.5*	*46.7*
SOR	*94.7*	*15.4*
TSP	*86.7*	*46.6*
Water	*75.8*	*44.3*

Table 1. Migration Times

For short running applications such as *FFT* and *SOR* the startup and initialization times dominate.

5.5 Visualizing

Coral supports lifetime events about the entities being monitored such as *a barrier has been reached* or *a process has been migrated*. Visualization tools [6] can register themselves to be notified when events of interest occur. The order of events is supported via vector timestamps.

5.6 Checkpointing and Debugging

Apart from process migration, the ability to save and restore the state of distributed applications is very useful for several other purposes: e.g. fault tolerance, rescheduling, and debugging. In contrast to the Coral process migration, checkpointing saves states of all processes and restores them on the same nodes where they had been running before the checkpointing. Since there is no guarantee for a process to obtain the same communication points as the ones used before the checkpointing, after restarting, all communication routes are rebuilt.

6 … to SOA Monitoring

While our results and experience from lessons learned during the designing and implementation of the online monitoring for DSM applications have proven to be very useful and enlightening we have started porting Coral to SOA [9] environments.

To fulfill the maturity level 4–*Value* that requires the usage of sophisticated tools; we will monitor *services* for the purpose of visualization and connections to the workflows and business processes. Performance analysis will be useful for checking the health of an application measuring the frequency of calls, the duration of calls, and amount of data exchanged. Execution replay will be useful to support distributed debugging and our load measurements techniques can be useful for balancing and redirecting the load in a farm. We will also monitor transactions, especially the long-running ones, and provide online security control.

Coral is currently being ported to the WCF (Windows Communication Foundation) and JAX-WS (Java API for XML-Based Web Services) middleware that support SOAP based and RESTful based distributed applications.

7 Conclusion

In this paper we elaborate the design and implementation decisions for an online monitoring system that supports DSM applications. The requirements for the monitoring functionality are driven by the perceived needs of the application programmers. To hide the complexity as well as the diversity of DSM design and implementation choices, Coral transparently manages resources that cater for the DSM functionality, giving the tools an abstraction of shared memory on loosely coupled machines.

During the course of this research we resolve several important issues concerning the development of online monitoring system for distributed applications and will use that experience to port Coral to the most prevailing and promising distributed technology – SOA.

References:

[1] I. Zoraja, *Online Monitoring is Software DSM Systems*, Shaker Verlag 2000.

[2] T. Ludwig, R. Wismüler, V. Sunderam, A. Bode. OMIS – Online Monitoring Interface Specification, Version 2.0, Technical report 9, LRR-TUM, München 1997.

[3] R. Wismueller, J. Trinitis, and T. Ludwig. *OCM – A Monitoring System for Interoperable Tools*. In the Proceedings of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT98, pages 1-9, ACM press, August 1998.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keheler, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, *TreadMarks: Shared Memory Computing on Networks of Workstations*. IEEE computer, 29(2):18-28, February 1996.

[5] Microsoft Corporation, *BizTalk Server 2006: Business Activity Monitoring*, Whitepaper, 2005.

[6] A. Bode. *Parallel Program Performance Analysis and Visualization*. In Dongara and Tourancheau, editor, Environments and Tools for Parallel Scientific Computing, SIAM, pages 246/253, 1994.

[7] V. S. Sunderam, PVM: *A Framework for Parallel Distributed Computing*. Concurrency: Practice and Experience, 2(4):315/339, December 1990.

[8] I. Zoraja, A. Bode, and V. Sunderam. *A Framework for Process Migration in Software DSM Environments.* In Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, pages 158-165, IEEE Computer Society, 2000.

[9] Hewlett-Packard Company: Operational Management and Monitoring of Business Processes in the SOA World, Whitepaper, 2006.

[10] M. Litzkov, T. Tannenbaum, J. Basney, and M. Livny. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. Technical report 1346, University of Wisconsin-Madison, April 1997.

[11] B. Balis, M. Bubak, M. Radecki, T. Szepieniec, and R. Wismüller. Application Monitoring in CrossGrid and Other Grid Projects. In Proc. European Across Grids Conference 2004, Nicosia, Cyprus, January 2004.