Final thesis

Resource Monitoring in a Distributed Self-Managing Middleware

by

Oskar Hermansson

LIU-IDA/LITH-EX-A--08/061--SE

2008-12-18
Final Thesis

Resource Monitoring in a Distributed Self-Managing Middleware

by

Oskar Hermansson

LIU-IDA/LITH-EX-A--08/061--SE

2008-12-18

Supervisor: Magnus Tränkner, Enea
Examiner: Prof. Petru Eles
Abstract

The next generation of automotive systems is self-managing. Concepts such as self-aware and self-monitoring are necessary parts of the system to be able to meet the requirements of self-configuration and self-optimization. DySCAS is a research project, funded by the European Union, which aims at developing such a system.

This thesis report describes and elaborates the theory for three topics: Built-in self-test, resource monitoring and power management, all with the focus on integration in DySCAS. Resource monitoring is selected to be further investigated with design and implementation in SHAPE, the reference implementation of DySCAS.

To test and verify the resource monitoring system, a load balancing scenario was created. The result from this scenario is presented in graphs for visualization and a discussion about the benefits and drawbacks of the implemented system. Finally the conclusions from both the literature study and the implementation work are summarized and some ideas on future work are presented.
List of Figures

2.1 Schematic overview of the DySCAS architecture .................. 6
2.2 Overview of the SHAPE architecture ............................... 7
6.1 Simple DySCAS system with three nodes and four applications .... 21
6.2 DySCAS system with resource monitoring enabled. Node load marked with darker color ................................. 22
6.3 DySCAS system where load balancing has been performed ........ 22
6.4 Logical view of the resource monitoring system ................... 23
6.5 Deployment view of the resource monitoring system .............. 23
6.6 Resource monitoring components inside SHAPE  

7.1 Implemented resource monitoring system with processes  

8.1 Sampling problem for CPU load when using the load application  

A.1 System CPU load with load 1m as parameter and ten seconds interval  
A.2 System applications state with load 1m as parameter and ten seconds interval  
A.3 System CPU load with load 1m as parameter and 60 seconds interval  
A.4 System applications state with load 1m as parameter and 60 seconds interval  
A.5 System CPU utilization with utilization as parameter and ten seconds interval  
A.6 System applications state with utilization as parameter and ten seconds interval  

List of Tables

7.1 Data structures defined in the resource collection interface  
7.2 Functions defined in the resource collection interface  
7.3 Local resource monitoring signal interface  
7.4 Global resource monitoring signal interface  
8.1 Processes mapped to nodes in load balancing scenario
List of Algorithms

7.1 Infinite loop that executes every second 
7.2 Main loop of local resource monitor
Chapter 1

Introduction

This chapter gives a short background and presents the purpose of the thesis as the methods used to achieve its goals. Finally, an outline of the rest of the report is presented.

1.1 Background

This thesis was done at Enea AB, a software and IT consulting company in Kista, Stockholm. Within the frames of a research project called DySCAS, which Enea has been involved in for two years, several master theses have been carried out previously. This thesis builds on the results from the previous work, but has explored new areas of the project.

1.2 Goal and Objectives

The overall purpose of this thesis is to improve the quality and extend the functionality of the reference implementation of DySCAS. The objectives are to answer the following questions that were raised in the assignment specification:

1. What can be tested with built-in self-test (BIST) and how can it be integrated in DySCAS?
2. How can resource monitoring be performed within a DySCAS system?
3. Which quantities in a DyCAS system are involved in the concept of power management?

1.3 Methods

The thesis was started by an introduction to the project and the problem statement. During this period an assignment specification was written, which contained the assignments to do and questions to answer. In the next phase a literature study was carried
out. This study consisted of investigating the topics concerned in the assignment specification, by reading scientific articles and doing some practical research in the area. Work has also been spent on reading the DySCAS project specifications to understand the use cases, requirements and architecture as well as the SHAPE reference implementation.

The literature study was followed by an implementation phase where one of the objectives was selected for being designed and implemented. The implementation ended in an integration phase, were the implemented system was integrated with the existing implementation of DySCAS. A video demonstration showing the implemented system in run-time was also created.

1.4 Outline

The next chapter contains an overview of the DySCAS project and its use cases and requirements. The architecture and the reference implementation SHAPE are presented. Chapters 3, 4 and 5 describe the results of the theoretical literature study. These chapters address the three questions raised in Section 1.2.

Chapters 6 and 7 describe the design and implementation respectively of a resource monitoring system. Chapter 8 contains a final discussion about the conclusions from the theoretical parts and the result from the implementation part.
Chapter 2

The DySCAS Project

DySCAS stands for Dynamically Self-Configuring Automotive System, and is a research project funded by the European Commission with several European companies and universities involved. Among the Swedish stakeholders, you can find Enea Services AB, Volvo Technology AB and KTH (Royal Institute of Technology). The project started in June 2006 and will end in December 2008. The objectives of the project are described on the project website as:

“The main objective of the DySCAS project is the elaboration of fundamental concepts and architectural guidelines, as well as methods and tools for the development of self-configurable systems in the context of embedded vehicle electronic systems. The reason is the increasing demand on configurational flexibility and scalability of the systems imposed by future applications which will include simultaneous access to a number of mobile devices and ad-hoc networking with the built-in devices”

2.1 Layers of Electronics

The electronics in a car can typically be divided into three separated layers. The lowest layer is called the safety layer. It contains the electronics for the engine, the breaks and other safety-critical components. The middle layer is called the body electronics layer and it contains the general electronic components in a car. These are components such as power windows, power seats and door locks. The upper layer is called the infotainment layer. This layer contains the information and entertainment components, such as car stereo, GPS device and LCD display. The DySCAS project is mainly concerned with the infotainment layer.

2.2 Use Cases

The Scenario and System Requirements document [6] contains the DySCAS use cases. The use cases are divided into four different groups, called Generic Use Cases (GUC).
Each GUC contains a number of Specific Use Cases (SUC). The four GUCs are:

**GUC1:** New Device Attached to the Vehicle  
**GUC2:** Integrating New Software Functionality  
**GUC3:** Closed Reconfiguration  
**GUC4:** Resource Optimization

These four GUCs describe the functionalities that the system provides in different areas. Two of these use cases are more interesting for this thesis. The first is **GUC2: Integrating New Software Functionality.** This use case involves a download of new software into the system. This could be either new application software or updates for the operational software (the middleware).

The second use case is **GUC3: Closed Reconfiguration.** This use case starts by the detection of an unexpected event, such as failure of a node. The system needs to be reconfigured to provide the best possible operation under the new circumstances. The strategy for calculating the new configuration could be based on some non-functional requirement such as minimizing power consumption. The outcome of a reconfiguration is a new stable configuration for the system.

### 2.2.1 Validate New Software

GUC2 contains two interesting SUCs, which are called **SUC2/1: Diagnosis and System Maintenance** and **SUC2/3: Software Update and Upgrade.** These both have the requirement that after successful software download, the new software should be validated by the system. This validation is done to ensure that the new software will work properly together with the rest of the system.

### 2.2.2 Load Balancing

GUC3 contains two SUCs that are especially interesting. The first is **SUC3/4: Load Balancing.** The purpose of SUC3/4 is to use the available hardware more efficiently by balancing the load of the nodes. If one node is overloaded it will have problems to complete its tasks on time and eventually it will not be able to provide the intended functionality. By moving functionality from an overloaded node to an underloaded node, the system will achieve a better balance between the nodes.

### 2.2.3 Graceful Degradation

The second SUC of GUC3 is called **SUC3/5: Graceful Degradation.** The purpose of this use case is to provide increased fault tolerance by having the system to degrade gracefully. In case of, for example, a node failure, a reconfiguration of the system should be triggered. If there no longer exist enough resources to provide full functionality, the system should prioritize the essential functionality and disable other "nice to have"
functionality. This strategy of the reconfiguration is called graceful degradation because it allows the system to be degraded step by step instead of having a failure that makes the system unusable.

An example for this situation could be when the battery in a car is running low. Instead of disabling the navigation system, the sound volume on the car stereo is reduced. If the battery level becomes even lower, the whole entertainment system might be disabled.

2.3 Requirements

The use cases described in the previous sections come with a list of functional requirements. Below are the requirements that are especially interesting for this thesis.

**REQ-A-29: Validate new software.** The system shall validate the new software when the execution starts and ensure that it works properly with the rest of the system.

**REQ-A-33: System state monitoring.** The system shall continuously monitor the system state. This includes information about error codes, software versions and middleware policies.

**REQ-A-44: Resource monitoring.** The system resource usage shall be monitored.

**REQ-A-39: Detection of potential resource imbalance.** The system shall detect imbalances in how the system utilizes its hardware resources such as CPU, memory, power and communication bandwidth.

These requirements will serve as a motivation for the topics that have been investigated and analyzed as well as what has been implemented in this thesis.

2.4 Architecture

The DySCAS project has developed a conceptual system architecture [2]. A schematic overview of the architecture can be seen in Figure 2.1. This architecture defines the instantiation interface, the middleware services and the application program interface.

The instantiation interface, also called the portability layer, contains services that interact directly with the underlying system platforms. These services provide portability, interoperability and transparency to the upper layer.

The middleware services are divided into three layers. The lowest layer is called the primitive operation layer. It provides the most basic core services in the middleware, such as detection of new devices. The middle layer is called elementary operation layer and it provides dependability and configuration handling. The upper layer is the task layer. It provides the autonomic decisions and self-management.

The application program interface is the interface for the applications running on top of the middleware. It provides a complete abstraction of the middleware service and its functionality.
2.5 Middleware

The role of Enea in this project has been to design and implement a reference implementation and a demonstration platform. This reference implementation is called SHAPE (Self-configuring High-Availability Policy-based system for Embedded systems). Currently, SHAPE runs on Linux, OSE and OSE Epsilon (OSE is a real-time operating system developed by Enea and OSE Epsilon is a lightweight version of OSE, aimed for embedded systems with very limited resources). Previously, a demonstration platform consisting of six nodes running OSE Epsilon, connected to a CAN bus network has been used, but in this thesis a network of Linux and OSE nodes connected with Ethernet will be used.

The architecture of SHAPE can be seen in Figure 2.2.
Figure 2.2: Overview of the SHAPE architecture
Chapter 3

Start-up Testing

Start-up testing is a procedure that takes place during the start-up of a (new) application, running on top of the middleware. Before the application is allowed to execute freely, it needs to pass the start-up test procedure. This procedure consists of several steps and covers several different areas. The main motivation for the start-up test procedure is to provide a method to satisfy the DySCAS requirement of software validation (REQ-A-29). However, this method is not limited to only perform software validation. It could also contain several other features that are implied by the DySCAS project specification, but not explicitly stated in any requirement.

3.1 Software Validation

Validation, and in particular, software validation, could mean several different things depending on the context. Validation is often defined as the process of checking if a system meets its specified criteria. In the software world, validation means checking that the software design fits the intended usage ("You have built the right product"). This is not to be confused with verification, which means checking that the software satisfies the requirements ("You have built it right"). [20]

In the context of our self-managing middleware and its applications, however, software validation means something like this: "Checking (during run-time) that an application will behave as expected (from both the middleware’s and the application’s point of view)".

A realization of this concept could include:

- Comparing the software version numbers of the applications and the middleware to ensure compatibility.

- Calculating hash numbers from the (downloaded) application binaries and comparing with known/expected values to ensure the integrity of the files.
3.2 Controllability

For a self-managing middleware it is important to be able to control the applications running on the system. Currently, applications are executed as independent processes and this reduces the possibility to manage them and their behavior through the middleware. To cope with this and increase the controllability, a control signal interface is proposed. This interface defines signals that all applications must listen and respond to.

The control signal interface could include the following signals:

**TERMINATE:** The application must immediately terminate itself.

**RESTART:** The application must restart itself.

**PING:** The application must respond with a pong to confirm it is alive.

**STATUS:** The application must respond with its current status (e.g. OK, ERROR, WAITING etc.)

To verify that all applications implement this signal interface, it should be tested during the start-up testing procedure. Verifying the PING and STATUS signals is straightforward, but to verify the TERMINATE and RESTART signals a little more effort is needed, because they require close interaction with the underlying platform and operating system.

3.3 Built-in Self-test

Built-in Self-test (BIST) is a general concept for electronic and computer systems which refers to a design methodology where the system is equipped with the ability to test itself and the test cases are included within the system. This makes it possible to test a system after it has been deployed and delivered, by making it run the BIST functions.

As software systems grow more and more complex, and with the introduction of self-managing software, the need for self-testable software has emerged [15]. Software BIST is a component of a system that contains functions and methods for performing self-test of the system. This is especially useful in component-based systems where each component can contain its own BIST module. The BIST module can either be designed to test the internal behavior of a component or to test a component’s external interface (the one used by other components). The latter case is often referred to as contract-based testing. [13, 1]

SHAPE (the middleware system) is not a true component-based system. Its internal modules and services are linked tightly together and they are not designed to be replaced by some third-party module. However, the application layer on top of the middleware can be viewed as a distributed component-based system, where the components are not known at design time or even at deployment time and third-party applications are not only possible, but assumed. This dynamic nature of SHAPE makes it particularly interesting to introduce the concept of software BIST into the system.
Because BIST modules are written for specific applications, it is the responsibility of the application developer to supply the application with its own BIST module. The role of SHAPE in this scenario would be to initiate the self-testing by activating the BIST module when the application is about to start.

### 3.4 Summary

To summarize this chapter it can be concluded that there are several possible approaches for integrating some kind of start-up testing in DySCAS. However, the requirements in DySCAS give little or no help at choosing one of these approaches.

Even though BIST is a useful tool when building larger component-based software systems, it is not suitable for integration in the DySCAS middleware, mainly because it would require the components to be redesigned. It might be a better fit in the application layer, but it would demand more requirements on the application developer. In a potential future, where DySCAS is a matured application platform for the car industry, and several third-party developers are supplying applications, integration of BIST for the applications could be very beneficial.
Chapter 4

Resource Monitoring

Resource monitoring is the activity of monitoring (by measurement, calculation and logging) the usage of system resources. The system resources consist of hardware resources with physical constraints. Among the most notable resources, in the context of a distributed embedded electronic system, are CPU, memory, and communication network. To be able to monitor a resource, first a quantity has to be defined. This quantity needs to have a magnitude and to be measurable.

4.1 Portability Concerns

A problem that arises for systems running on multiple hardware platforms and operating systems is to define quantities with portability and interoperability properties. It is crucial that the resource monitoring produces that same result independent of platform and operating system. Inconsistency between the numbers presented by the same resource monitoring activity ran on different platforms under the same conditions is not acceptable. To solve this problem, a standards-based approach to defining the quantities could be used. Unfortunately, such a standard does not seem to exist. The POSIX standard [12] for example, does not include anything about how to define resource monitoring quantities.

4.2 The Virtual Process File System - procfs

In many of the Unix-like operating systems a virtual file system called procfs (process file system) is implemented. It was first invented in the 8th edition of Unix [14] and was later on ported and evolved to other Unix-like operating systems. The file system is used for accessing process information. In Linux this file system is extended to also include non-process-related system information, such as CPU load, memory usage and other useful resource monitoring quantities. The problem with procfs is that there exists no standard specification and the functionality differs in the operating systems where it has been implemented. One example of this issue is a project that aims at porting the Linux procfs to GNU/Hurd [16].

Oskar Hermansson

Linköping University
4.3 Distributed Monitoring

The adaption of the resource monitoring discussed here will be in a distributed system. Therefore, it is not only necessary for each node to have a local view of its resource usage, but there should also exist a global view of the total resource usage. This view should be a collection of the local views to present a coherent image of the total resource usage for the complete system. This raises some questions regarding update frequency and synchronization issues among the local resource monitoring facilities.

4.4 Resource Quantities

This section describes a few common resource quantities that have been investigated. Their definition and usage is explained on a general level, and where it has been possible, implementation details for Linux and OSE5, respectively, are examined.

4.4.1 CPU Load

The CPU load is a quantity that can be used to tell if the CPU is currently overloaded. A high load will result in the system being less responsive. To calculate the CPU load, first a load number is needed. The load number is equal to the current length of the ready-queue for the CPU scheduler of the operating system. This number includes both processes that are currently executed by the CPU and the processes that are waiting for CPU time. Processes in other states, such as idle or sleep, are not included in the load number. For example, on a single-processor system a load average of 2.34 would indicate that on average there where 1.34 processes waiting to be scheduled. This could correspond to an overloaded system.

The load average $LA(t)$ is calculated as an exponential moving average (EMA) of the load number $LN(t)$ as

$$LA(t) = \alpha LA(t-1) + (1 - \alpha) LN(t) \quad 0 < \alpha \leq 1 \quad (4.1)$$

where $t$ is the discrete sampling time and $\alpha$ is the weighting/smoothing constant [11].

Linux

The Linux kernel has built-in support for calculating the CPU load average. This is implemented by calling a load calculating function every 5 seconds. Three different load averages are calculated. These correspond to the load average over 1, 5 and 15 minutes. The load average calculation in the Linux kernel looks like

$$LA_m(t) = \alpha_m LA_m(t-1) + (1 - \alpha_m) LN(t) \quad \text{for } m = 1, 5 \text{ and } 15 \quad (4.2)$$

where $LA_m(t)$ is the current load average over the last $m$ minutes, $LA_m(t-1)$ is the previous load average over the last $m$ minutes, $LN(t)$ is the current load number and $\alpha_m$ is the weighting constant, defined as $\alpha_m = e^{-\frac{5}{m}}$. [7, 8, 9]
Getting the current CPU load averages on Linux systems, is easily done by calling the commands `w` or `uptime`. The load averages are the three numbers presented on the right part of the output. The order of the numbers is: 1, 5 and 15 minutes. To get these load averages within a program, the `sysinfo()` syscall within the `sys/sysinfo.h` header file can be used.

**OSE5**

For monitoring CPU load in OSE5, there exists a signal interface for the run mode monitor (RMM) that can be used. This interface is found in `ose_spi/monitor_cpu.sig` and it defines several different profiling types. The one most useful here is probably the one for monitoring the total CPU load.

Before this interface can be used, a connection to the RMM is needed. This is setup with `MonitorInterfaceRequest` and `MonitorConnectRequest`. The total CPU load profiling is activated by sending the `MonitorSetCPUReportsEnabledRequest` signal. To get the current load, you send a `MonitorGetCPUReportsRequest` signal. [5]

### 4.4.2 CPU Utilization

CPU utilization is the quantity that shows how much time the CPU is currently spending executing tasks. Each tick, the CPU can be in one of several states. These states can for example be executing user program, waiting for I/O operation and being idle. By counting the amount of ticks the CPU has spent in these states since the last time, you can calculate how much time was spent in each state, in percentage. This gives a fairly good estimation of the CPU utilization.

Assuming the previous example, states are available for a CPU and $T_S(t)$ is the total number of ticks spent in state $S = \{\text{exec}, \text{wait}, \text{idle}\}$ since the system was started, then the difference in the number of ticks from the previous sample time can be defined as

$$D_S(t) = T_S(t) - T_S(t-1)$$  \hspace{1cm} (4.3)

and the CPU utilization $U(t)$ can finally be calculated as

$$U(t) = \frac{D_{\text{exec}}(t)}{D_{\text{exec}}(t) + D_{\text{wait}}(t) + D_{\text{idle}}(t)}$$  \hspace{1cm} (4.4)

**Linux**

In Linux, the virtual process file system, procfs (described in Section 4.2), holds information about the CPU ticks. In particular, the file `/proc/stat` stores the number of ticks the CPU has spent in each of eight different CPU states. These states are:

- User CPU time (Executing user process)
- Nice CPU time (Executing user process with reduced priority)
• System CPU time (Executing system process)
• Idle (CPU is idle)
• I/O wait (CPU is waiting for input/output operation to complete)
• Hardware IRQ (Executing hardware interrupt request)
• Software interrupts (Executing software interrupt)
• Steal time (Applicable for systems running on a virtual machine)

Since /proc/stat stores the total number of ticks in each state, the file has to be read twice and using equations 4.3 and 4.4 the CPU utilization can be calculated. If the file is only read once, and no difference is calculated, the total CPU utilization since the system was booted can be calculated.

OSE5
To enable monitoring of CPU utilization in OSE5, an approach using an idle process could be used. This idle process should run at the lowest priority in the system and its task is to count the number of CPU ticks elapsed when executed. This process is only executed when no other task is waiting for CPU time. Comparing the number of ticks the idle process was executed with the total number of elapsed ticks, the current CPU utilization can be calculated as

\[
U(t) = \frac{D_{exec}(t)}{D_{total}(t)} = \frac{D_{total}(t) - D_{idle}(t)}{D_{total}(t)} = 1 - \frac{D_{idle}(t)}{D_{total}(t)} = 1 - \frac{T_{idle}(t) - T_{idle}(t-1)}{T_{total}(t) - T_{total}(t-1)}
\]

(4.5)

where \(T_{idle}(t)\) is the current number of ticks for the idle process, \(T_{idle}(t-1)\) is the previous number of ticks for the idle process, \(T_{total}(t)\) is the current total number of ticks for the CPU, \(T_{total}(t-1)\) is the previous total number of ticks for the CPU and \(t\) is the discrete sampling time.

Using this idle process approach could however have impacts on, for example, power management facilities in OSE, because it will look like the CPU is never idle at all.

4.4.3 Memory Usage
Current memory usage could be defined as either the number of used (allocated) bytes, the number of free (unallocated) bytes, the percentage used of total memory size or the percentage free of total memory size. For two reasons it seems most intuitive to use the definition of used memory instead of free memory. First, since the resource quantity is called memory usage it should reflect the system memory usage and not the opposite situation. Second, for the other resource quantities, increased values mean higher system
4.4. RESOURCE QUANTITIES

load and decreased values mean lower system load. If the memory usage definition would be the amount of free memory, then the opposite situation would occur.

A significant issue with this quantity is that operating systems manage memory in very different ways. Memory management can differ in several respects, such as partitioning techniques and allocation strategies.

**Linux**

Memory management in Linux uses a limited version of the segmentation model. There are segments for kernel code, kernel data, user code and user data [19]. Information about memory usage is accessible through both the procfs file system and the `sysinfo()` syscall.

**OSE5**

OSE uses signals for communication and memory that is dynamically allocated is called a signal buffer. The kernel allocates the buffers upon request, and to avoid external fragmentation of memory, the kernel only allocates buffers of fixed (configurable) sizes. When a process frees a signal buffer, it is given back to the kernel, but the kernel never deallocates the buffer once it has been allocated. Instead the buffer is kept in a list of free buffers to be used later. This important fact makes memory usage in OSE systems behave a little bit different from other systems. The memory usage by an OSE system can never decrease with time, if the amount of allocated memory is considered. The memory usage will increase over time until it has reached its maximal point and will then stay at this level forever. [4]

In addition to this, OSE uses memory pools to separate the memory management between different blocks of processes. The system configuration might consist of one memory pool for the kernel and the core OSE processes and another pool for the application processes.

### 4.4.4 Network Throughput

Network throughput is the rate at which data is sent and received over a communication link. The network throughput is usually measured in bytes/s or bits/s. There are several different possible definitions of the throughput for networks and communication links. The general definition of throughput is the amount of data that is successfully transmitted, but it has to be decided in which layer the throughput is measured. If measured in direct connection to the physical layer the throughput includes a lot of overhead produced by header data from each layer above. If the throughput is measured at the application level (for example by sending a file between two nodes in a network and measure the time of the transfer) the throughput can never reach the theoretical bandwidth of the communication link (because of the overhead from the headers). Because of this, throughput at application level is often called goodput (i.e. the amount of useful data per time unit).

Oskar Hermansson

Linköping University
Linux

In Linux, information about network statistics such as sent and received bytes and packets can be found using procfs. The file `/proc/net/dev` stores the amount of sent and received bytes and packets from all interfaces since system start. Using a similar approach as with CPU utilization, this file can be read at regular intervals and the difference between each iteration can be calculated and this is the network throughput.

4.5 Summary

This chapter can be summarized by saying that the most important issue of resource monitoring in DySCAS is the problem with portability of the implemented resource quantities. When choosing and defining the quantities, portability and consistency between platforms and operating systems has to be taken into consideration. Another important conclusion on this subject is that there are no standards or specifications to follow when implementing resource monitoring.
Chapter 5

Power Management

Power management is the activity of managing the power consumption generated by a system. Depending on the context, power management can have somewhat different goals. In general, the goal of power management is to minimize power consumption while maximizing the system performance. In an embedded or a distributed system the goal is to minimize power consumption while not exceeding any deadlines and still satisfying potential quality of service constraints on the system. [17]

5.1 Techniques

There are in general two different kinds of techniques that can be used to apply power management in a system. The first is static power management, which is applied during design-time, and the second is dynamic power management, which is applied during run-time.

5.1.1 Static Power Management

Static power management represents the techniques that can be applied at the system design stage. During task mapping and scheduling, especially for distributed systems, consideration is taken to where the tasks are mapped and in which order they should be scheduled to reduce power consumption. Static power management also includes compile-time strategies such as instruction selection, data placement and register allocation. [18, 17]

5.1.2 Dynamic Power Management

Dynamic power management (DPM) is the technique of detecting idle devices or components of a system during run-time and putting them into a sleep or power-saving mode. The length of the idle-period must be long enough for the power-saving cost to weigh up for the cost of shutting down and starting up the device. To detect such idle-periods, a DPM policy has to be used. The design of such policies has been an active research
topic for a long time [10, 3]. The general approach is a policy based on either timeout, predictive or stochastic techniques. Timeout policy can be using a fixed, random or adaptive time length. Predictive methods predict the length of an idle-period for an idle device, to decide if it is worthwhile to put the device in sleep mode. With stochastic techniques, Markov models can be used to model the idle-period length [21].

5.2 Applied in DySCAS

There are several aspects that need to be considered to be able to successfully integrate power management in DySCAS. First of all, because DySCAS runs on multiple hardware platforms and operating systems, it probably only makes sense to use dynamic power management techniques. Static power management infers design-time and compile-time knowledge about the specific platform, which is not available when designing a DySCAS system.

On the lowest level, the hardware platform that runs DySCAS needs power management support. The hardware modules need to have the possibility to enter a power-saving or sleep mode. On the operating system level, support for some DPM policy is needed for detection and prediction of idle-periods. On a distributed level, global policy decisions can be taken to reduce power consumptions, while preserving QoS constraints. Such decisions can be to activate an inverted load balancing process. If several nodes are performing tasks that could be handled by fewer nodes, the tasks are rearranged to release one node and put it in power-saving mode.

To realize power management on a distributed level, resource usage parameters from the local nodes are needed, and this is yet another motivation for the need of resource monitoring in DySCAS.

5.3 Summary

To summarize this chapter it can be concluded that only dynamic power management techniques are possible to integrate in DySCAS. The hardware platform and the connected devices need to have built-in support for a power-saving or sleep mode. In addition to this, the operating system needs to implement a dynamic power management policy for identifying idle devices and predicting idle-periods. If these prerequisites are met, then power management on a distributed level can be implemented in a DySCAS system.

Oskar Hermansson
Linköping University
Chapter 6

Design of Resource Monitoring System

This chapter describes the design of the resource monitoring system that has been implemented in this thesis. The load balancing scenario which serves as a motivation is described here, followed by an architectural overview of the system. The architecture is presented with two different views. The last section describes the mapping of the implemented resource monitoring system into the existing SHAPE system.

6.1 Load Balancing Scenario

The main motivation for implementing a resource monitoring system is the load balancing use case described in Chapter 2. In this scenario, the DySCAS system consists of multiple nodes with (possibly) varying hardware resources and capacities. The system provides some functionalities or services to the user or the environment. These functionalities or services are implemented in several applications running on top of the system middleware. An example of a DySCAS system setup can be seen in Figure 6.1.

Now, if a resource monitoring system is integrated in DySCAS, the resource usage for all nodes can be continuously monitored and a global knowledge of the system status is present. This makes it possible to detect both nodes that are overloaded (having trouble to provide the intended service because of lacking physical hardware resources)

![Figure 6.1: Simple DySCAS system with three nodes and four applications](image-url)
and nodes that have a low load and could possibly be assigned more tasks without impairing the current services on that node.

Such a situation could occur either because of bad offline task scheduling, because of some unexpected event (such as node or service failure) or simply because of the dynamic nature of the DySCAS system that new functionality is dynamically requested during run-time. Figure 6.2 shows a system where the load of each node is monitored.

When the situation is detected by the resource monitoring system it can be used to trigger a load balancing. This means that the system uses parameters from the resource monitoring system to decide if, what and when some service or functionality should be moved from one (overloaded) node to another. Figure 6.3 shows the system after the load balancing has been performed.

There are a lot of implications when performing load balancing and a lot of parameters must be considered. For example, which services can be moved to which nodes (hardware and resource dependencies)? Which parameters from the resource monitoring system should be used to classify a node overloaded? At which rate should services be moved? As described earlier, the focus of this thesis has been to design and implement the resource monitoring system and not the load balancing, but a small and simple load balancing algorithm has been implemented in cooperation with another thesis worker which serves as a proof-of-concept. This load balancing is described in Section 7.5.
6.2 Architecture

The resource monitoring system consists of several components. In Figure 6.4 the components are presented as layered blocks of the logical architecture of the system. Each component uses only components next above or below. The arrows with smaller heads denote that the component uses a signal interface for communication. The arrows with thicker heads denote function calls. The components are described in detail in Chapter 7.

Figure 6.5 shows a deployment view of the system. In this example, the system consists of two nodes and the global resource monitoring process is located on node 1. It shows that local processes are duplicated for each node and that global processes are unique throughout that system.

Oskar Hermansson
Linköping University
6.3 Mapping into SHAPE

When the resource monitoring system is integrated into SHAPE the names of the components are changed to match the architectural style and conventions of SHAPE. Figure 6.6 describes the mapping between the components described in this chapter and the components inside SHAPE.

Block 1 contains a library called sys_qos.c. This library corresponds to the resource collection interface. The process called qos_collector corresponds to the local resource collection component. In block 2 the file sys_qos.sig defines the signal interface between the local resource collection and the local resource monitor. The file sys_qos.h contains the function interface towards the functions defined in the resource collection interface. Block 3 contains two processes, one local and one global. These correspond to the local and global resource monitoring components. Finally, in block 4 the file dyscas_qos.sig contains the signal interface used to communicate with both the local and global resource monitor.
Figure 6.6: Resource monitoring components inside SHAPE
Chapter 7

Implementation of Resource Monitoring System

This chapter provides descriptions of the implementation of the different components that constitute the resource monitoring system. The components are presented in a bottom-up approach, starting with the lowest level, closest to the platform and hardware, and ending with the global components. The chapter is ended with a presentation of the development environment that has been used.

7.1 Resource Collection Interface

The Resource Collection Interface component defines the interface that upper layer components can use to collect resource usage information from the system. It defines both the data structures used to pass information and the functions that can be called. The interface is platform and operating system independent and currently defines the data structures listed in Table 7.1.

These data structures are used for storing the information that is collected from the system. It is the responsibility of the upper layer component to allocate memory for these structures before calling any of the functions in this component. The functions defined in this interface are presented in Table 7.2. The description of each function is

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_qos_cpu_load_t</td>
<td>Structure for storing CPU load average</td>
</tr>
<tr>
<td>sys_qos_cpu_utilization_t</td>
<td>Structure for storing CPU utilization</td>
</tr>
<tr>
<td>sys_qos_cpu_states_t</td>
<td>Structure for storing CPU states</td>
</tr>
<tr>
<td>sys_qos_memory_usage_t</td>
<td>Structure for storing memory usage</td>
</tr>
<tr>
<td>sys_qos_network_throughput_t</td>
<td>Structure for storing network throughput</td>
</tr>
<tr>
<td>sys_qos_network_data_t</td>
<td>Structure for storing network data</td>
</tr>
<tr>
<td>sys_qos_node_info_t</td>
<td>Structure for storing node info</td>
</tr>
</tbody>
</table>

Table 7.1: Data structures defined in the resource collection interface
### Table 7.2: Functions defined in the resource collection interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int sys_qos_cpu_load(...)</code></td>
<td>Get the current CPU load averages for 1, 5 and 15 minutes.</td>
</tr>
<tr>
<td><code>int sys_qos_cpu_states(...)</code></td>
<td>Get the amount of time the CPU has spent in each of the three states exec, idle and wait as well as the total amount of time since system boot. These numbers can be used to calculate the CPU utilization.</td>
</tr>
<tr>
<td><code>int sys_qos_memory_usage(...)</code></td>
<td>Get the current amount of used, free and total memory (in bytes) for the system.</td>
</tr>
<tr>
<td><code>int sys_qos_network_data(...)</code></td>
<td>Get the current total amount of bytes and packets sent and received since system start. These numbers can be used to calculate the network throughput (bytes sent/received per second).</td>
</tr>
</tbody>
</table>

The implementation of these interface functions is highly platform dependent. The sections below describe the implementation for systems running the Linux 2.6 kernel.

**int sys_qos_cpu_load(struct sys_qos_cpu_load_t *cpu)**

This function uses the built-in syscall `sysinfo()` to calculate the three load averages for 1, 5 and 15 minutes. The `sysinfo()` syscall returns the load averages in unsigned binary fixed-point arithmetic on the format Q16.16. This means that the first 16 bits of the number represents the integer part and the last 16 bits represents the fractional part. To convert the number to floating-point arithmetic (with single precision) the whole number is casted to `float` and divided with $2^{16}$ (because the size of the fractional part is 16 bits).

**int sys_qos_cpu_states(struct sys_qos_cpu_states_t *cpu)**

This function uses `procfs` (the process file system) to collect information about CPU states. The file `/proc/stat` is opened and the first line of the file is read. This line contains eight integers which represent the number of ticks that the system (sum of all CPUs on multi-processor systems) has spent in each of the eight different states. These states are, as described in Section 4.4.2, `us` (User CPU time), `ni` (nice CPU time), `sy` (system CPU time), `id` (idle), `wa` (I/O wait), `hi` (hardware IRQ), `si` (software interrupt) and `st` (steal time). The states are merged into three general states according
to Equation 7.1.

\[
\begin{align*}
\text{exec} &= \text{us} + \text{ni} + \text{sy} \\
\text{idle} &= \text{id} \\
\text{wait} &= \text{wa} + \text{hi} + \text{si} + \text{st} \\
\text{total} &= \text{exec} + \text{idle} + \text{wait}
\end{align*}
\]

These general states represent the total distribution of CPU states since system start. To calculate CPU utilization, this function is called at regular intervals and the difference is calculated, but this is performed in the upper component called local resource collection, described in Section 7.2.

```c
int sys_qos_memory_usage(struct sys_qos_memory_usage_t *mem)
```

This function uses the syscall `sysinfo()` to return the system memory usage. `sysinfo()` returns the total amount of system RAM, the current amount of free RAM and the memory unit in which the previous numbers are given. The numbers are multiplied with the unit to present the result in bytes and the amount of used memory is calculated from the amount of total and free RAM. Finally, the fraction of used memory compared to total memory available is calculated and the result with all four parameters is returned.

```c
int sys_qos_network_data(struct sys_qos_network_data_t *net)
```

This function uses, similarly to `sys_qos_cpu_states()`, procfs to fetch network data from the system. By opening and parsing the file `/proc/net/dev` four parameters can be calculated. These are received bytes, received packets, sent bytes and sent packets. Just like with the function `sys_qos_cpu_states()` these values are relative to when the system was started. To calculate the network throughput, this function is called at regular intervals by the Local Resource Collection component.

### 7.2 Local Resource Collection

The local resource collection component is responsible for the collection and also in some cases the calculation of the system resource usage quantities. It uses the functions and data structures defined in the resource collection interface. The component consists of one process, called `local_resource_collector`. The process consists of one main loop which is executed at regular intervals (default is every second). The process calls all of the four functions `sys_qos_cpu_load()`, `sys_qos_cpu_states()`, `sys_qos_memory_usage()` and `sys_qos_network_data()`.

The data received regarding CPU load and memory usage are already calculated resource quantities, but both the CPU states and network data need to be processed before they represent valid resource quantities. This is also the reason why it is important that the main execution block of this process is fired once exactly every second (or at least with reasonable precision).
By storing the CPU states for the previous iteration a difference can be calculated. This difference represents which states the CPU has had during the last second. This result is called CPU utilization and the numbers are given as floating-points from 0 to 1, where the sum of the states exec, idle and wait always is 1 (100%).

The same technique is applied to the network data. The difference from previous iteration is calculated and this difference is divided with the interval length in seconds (which defaults to 1) which gives the network throughput: The current number of sent and received bytes (and packets) per second.

This information is packed into four signals and forwarded upwards to the next component which is called local resource monitoring. The execution block ends with the calculation of the execution time which is used to decide for how long the process should sleep before it is time to execute the next iteration of the main loop. These calculations are presented in pseudo-code in Algorithm 7.1.

**Algorithm 7.1** Infinite loop that executes every second

```plaintext
interval ← 1.0
loop
  start ← time()
  :
  execute code here
  :
  exec ← time() - start
  if exec < interval then
    sleep(interval - exec)
  else
    {execution time longer than interval}
  end if
end loop
```

### 7.3 Local Resource Monitoring

The responsibility of the local resource monitoring component is to keep track of the latest measured resource data and to reply to requests from other services in the system that need resource monitoring information. The component consists of one process which has a main loop like sketched in Algorithm 7.2. Whenever a signal is received it can be one of two types. If it is a request signal, a reply signal is created, the latest resource monitoring data of the requested type is entered in the signal and the reply is sent back to the sender (which is some local service or application). If the received signal is an update signal (this is what the local resource collector sends), the stored resource data is updated with the new information and the received signal is forwarded to the global resource monitor (if such process exists and is registered in the system).
Algorithm 7.2 Main loop of local resource monitor

```pseudo
loop
    sig ← receive()
    if sig ∈ request then
        reply to sender with latest data
    else if sig ∈ update then
        save data and forward signal to global resource monitor
    end if
end loop
```

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYSCAS_QOS_CPU_LOAD_REQUEST</td>
<td>Request for CPU load</td>
</tr>
<tr>
<td>DYSCAS_QOS_CPU_LOAD_REPLY</td>
<td>Reply with CPU load</td>
</tr>
<tr>
<td>DYSCAS_QOS_CPU_LOAD_UPDATE</td>
<td>Update with CPU load</td>
</tr>
<tr>
<td>DYSCAS_QOS_CPU_UTILIZATION_REQUEST</td>
<td>Request for CPU utilization</td>
</tr>
<tr>
<td>DYSCAS_QOS_CPU_UTILIZATION_REPLY</td>
<td>Reply with CPU utilization</td>
</tr>
<tr>
<td>DYSCAS_QOS_CPU_UTILIZATION_UPDATE</td>
<td>Update with CPU utilization</td>
</tr>
<tr>
<td>DYSCAS_QOS_MEMORY_USAGE_REQUEST</td>
<td>Request for memory usage</td>
</tr>
<tr>
<td>DYSCAS_QOS_MEMORY_USAGE_REPLY</td>
<td>Reply with memory usage</td>
</tr>
<tr>
<td>DYSCAS_QOS_MEMORY_USAGE_UPDATE</td>
<td>Update with memory usage</td>
</tr>
<tr>
<td>DYSCAS_QOS_NETWORK_THROUGHPUT_REQUEST</td>
<td>Request for network throughput</td>
</tr>
<tr>
<td>DYSCAS_QOS_NETWORK_THROUGHPUT_REPLY</td>
<td>Reply with network throughput</td>
</tr>
<tr>
<td>DYSCAS_QOS_NETWORK_THROUGHPUT_UPDATE</td>
<td>Update with network throughput</td>
</tr>
</tbody>
</table>

Table 7.3: Local resource monitoring signal interface

The reason that the local resource collection component and the local resource monitoring component is split into two components are because the latter needs to listen for incoming signals. Because the `receive()` function is a blocking function call (i.e. it will not return until a signal is received) it would be impossible to guarantee that an execution block with a call to `receive()` can be executed at regular intervals (which is required for the local resource collection component). An alternative is to use the non-blocking receive function `receive_w_tmo()` where a timeout is specified. If a signal is not received within the specified timeout length, the function returns and execution control is returned to the caller. Using non-blocking receive is, however, a big difference in theory. It turns the currently deterministic computation model of SHAPE (with non-blocking send and blocking receive) into a non-deterministic model.

The signal interface for this component is specified in Table 7.3. It specifies all signals that are received to and sent from the local resource monitor process. The signals in the table are grouped together with their resource quantity type. Each resource quantity comes with three different signals, one used for requests, one used for replies and a signal used when sending updates.
7.4 Global Resource Monitoring

The last component of the resource monitoring system is the global resource monitoring component. It consists of one process called global resource monitor. It is a global process, which means that there only exists one instance of the process in the system. It can be located on any node in the system. When the process starts, it registers itself as a global service. This makes it possible for other services to subscribe to this process and receive a global reference which can be used to communicate with it, regardless of which node it is located on.

When registered, the global resource monitor waits to receive signals. Just like with the local resource monitor, the signals can be one of two types. If the signal is an update signal (sent from a local resource monitor) the resource information is stored in an array of system information. The information is grouped with the id of the sending node. If this node was previously unknown to the global resource monitor, it recognizes this as a new node. If the node is previously known, the old resource information is overwritten. If the received signal is a request signal, the kind of request is identified and a reply with the resource information is sent back. Table 7.4 show the signal interface for the global resource monitoring component.

7.5 Load Balancing

The load balancing component makes it possible to load balance a system during runtime. One part of the load balancer contains the functionality for requesting and evaluating resource monitoring information from the system (this is requested from the global resource monitor). Based on this evaluation and information about migratable applications running on the system, a decision to migrate a specific application from one node to another is taken.
The other part of the load balancer takes care of the actual migrations of applications between nodes. It also keeps an updated list of which applications are running on which nodes.

This load balancing component has been developed by another thesis worker, but the focus of his work has been to enable migration of applications between DySCAS nodes. The decision algorithm of the load balancer is therefore very basic and simplistic. More results and conclusions from this can be seen in Section 8.1.

7.6 Global Resource Logger

For debugging and visualization purposes, an additional process called the global resource logger was developed. The process requests system resource usage information from the global resource monitor and running applications information from the global load balancer each second. The information from the two processes is combined and written to disk. Two files are used, one which only contains the current information (this file is truncated before written each second) and one file which contains the complete log history for the system since it was started (the content is appended at the end of the file each second).

The two files serve different purposes. The file with the current information only can be used for real-time analysis of the system during run-time. The log file can be used for static offline analysis of the system. The data is written in a file format called YAML. It is a human-friendly plain-text data serialization format. Using this standard for the data has the benefits that the files are both easily read by humans and can be parsed by most popular programming languages using YAML bindings for that language.

7.7 Development Environment

The whole resource monitoring system has been completely written in C. As far as possible, the ANSI C99 standard has been followed. The resource collection interface was written for Linux but the other components should be platform independent and run on any platform that runs SHAPE.

The development environment has consisted of two virtual machines running Debian 4.0 (with Linux 2.6.18). For virtualization Sun VirtualBox was used. VirtualBox is an open source virtualization engine available for most platforms. It also has the very important feature of simulating an internal network between the virtual machines. This put together gives an environment of two computers running Linux connected to each other through an Ethernet switch, but the complete environment is simulated and everything is running on a standard Windows workstation.

As for sharing the source code between the host and the virtual machines, VirtualBox supports shared folders. One folder on the workstation (the host) is selected to be shared on both virtual machines. Everything in this folder is accessible from the virtual machines (both read and write privileges) so any changes made to a source code file on the workstation is immediately accessible from within the Linux nodes. The new code

Oskar Hermansson  
Linköping University
can be compiled and run directly inside the nodes without any need to copy or move any files before.

### 7.8 Summary

The implemented resource monitoring system can now be summarized in Figure 7.1. The implemented function library, local and global processes are marked. The signal interface with request, reply and update signals can be seen, representing the data flow in the system.
Chapter 8
Discussion

This last chapter concludes the thesis report with a discussion about the results from the implementation of the resource monitoring system, conclusions from the literature study and the thesis in general, and finally some words about possible future work.

8.1 Result

To evaluate the implemented resource monitoring system, a load balancing scenario was simulated. The scenario consisted of two Linux nodes running SHAPE and the processes needed for resource monitoring. The mapping of processes to nodes can be seen in Table 8.1. To simulate the load on the system, an application has been developed for this purpose. It consists of one process with an infinite loop consisting of one busy-period and one sleep-period. When starting the application, a parameter is passed to specify the amount of load that the application should produce. The parameter is a number between zero and one that specifies the fraction of the time that should be spent in the busy block. For example, if the number 0.5 is passed, the application will spend equal time sleeping and being busy, which will produce a CPU utilization of 50%. If the number 0.2 is passed, the application will be busy 20% of the time and sleeping 80% of the time.

It is interesting to note how this application’s behavior affects the resource quantities that are measured. Ideally, with a parameter of for example 0.2, the CPU load should increase with the same amount, but because of how the CPU load is calculated (the

<table>
<thead>
<tr>
<th>Node</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Local resource collector, local resource monitor, global resource monitor, global resource logger, global load balancer and local process starts</td>
</tr>
<tr>
<td>2</td>
<td>Local resource collector, local resource monitor and local process starter</td>
</tr>
</tbody>
</table>

Table 8.1: Processes mapped to nodes in load balancing scenario
Figure 8.1: Sampling problem for CPU load when using the load application

length of the scheduler’s ready-queue is sampled every five seconds) the interval at which
the application’s busy and sleep periods are repeated is crucial.

Figure 8.1 shows that a large interval can make it look like an application that runs
at 50% rate is detected as running on 100%. By having a sufficiently small interval
(compared to the sampling period time, which is five seconds) this issue can be dealt
with. With a small interval the probability that a sample time will be during the busy-
period approaches the fraction of busy time for the application, which is the expected
result.

For CPU utilization, there exist no sampling issues because the quantity is not cal-
culated using sampling. The difference in CPU states that is calculated every second
takes all states into consideration. Of course, if the interval of the application is larger
than the interval of the resource collection (determined by the local resource collector)
the CPU utilization will not be correctly calculated.

Another aspect that differs between CPU load and CPU utilization when using the
load application is that the CPU utilization will only be correct as long as the load
applications do not exceed the physical constraints. If for example three load applications
are started, each with the parameter 0.5 the result will still only be a CPU utilization of
100% and the applications will actually only run as if the parameter 0.33 was specified.

When running the load balancing scenario, in addition to the previously listed pro-
cesses, five load applications, each with a parameter of 0.2 were initiated on node 1. If
the load balancer detects that the system is unbalanced (by asking the global resource
monitor for system resource usage information) it takes a decision to migrate (move) one
application from the overloaded node to the one with the lowest load. This is repeated
at regular intervals so the desirable behavior is that at least two applications are moved
to node 2.

During the first run of the system, the load balancer was configured to use the 1
minute CPU load parameter for comparison of the load of the nodes. The interval
between each decision point was set to ten seconds. The result, which can be seen in
figures A.1 and A.2 in Appendix A, was an oscillating behavior where all applications
were moved between the nodes. This behavior was caused because the parameter that
was used (1 minute CPU load) takes one minute to converge to its value, but, since the load balancer uses an interval of ten seconds, it will have taken five decisions before the parameter has fully reacted even to the initial state.

To cope with this problem the load balancer was reconfigured by increasing the decision interval to 60 seconds. The result from this second run can be seen in figures A.3 and A.4. The result this time is better but after some time the system starts to oscillate again. It is more difficult to determine the cause of the behavior this time, but it probably still has to do with the fact that the parameter used for the decisions takes very long to converge and is lagging compared to the actual state of the system.

The last approach that was used was to change the parameter from CPU load to CPU utilization. Because CPU utilization reacts almost instantly (there is no convergence time) the interval time between the decision points in the load balancer is reset back to ten seconds again. The result from this last run of the system can be seen in figures A.5 and A.6. Using this approach proved to be much more efficient for our scenario. The figures show that the CPU utilization parameter is in step with the system state and no oscillating behavior is seen (except for the fifth application which is constantly migrated between the nodes, but this behavior is to be expected due to a very naive implementation of the load balancer).

8.2 Conclusions

This section describes the overall conclusions from this thesis, with respect to the questions raised in the beginning. The conclusions are based on the results from both the literature study and the design and implementation work.

Q1: What can be tested with BIST and how can it be integrated in DySCAS?

In general, almost anything can be tested using BIST. It is a technique that is more concerned with how the testing is organized rather than what is tested and how it is implemented. By embedding the test cases inside the components together with the code, the testing opportunities are increased from only comprising development testing to enable production testing, installation testing and run-time testing. BIST is also a useful tool when developing component-based software systems because the self-testing approach can be used to verify and validate a component’s external interface towards other components.

For integration in DySCAS, BIST seems to be especially suitable in the application layer. Different software vendors and third-party developers that are supplying DySCAS applications can use BIST to ensure compatibility between software components.

Q2: How can resource monitoring be performed within a DySCAS system?

This question is answered in Chapter 4 as part of the literature study and in chapters 6 and 7 as part of the implementation work. In general the most important issue of resource monitoring in DySCAS is the problem with portability of the implemented
resource quantities. When choosing and defining the quantities, portability and consistency between platforms and operating systems has to be taken into consideration. Another important conclusion on this subject is that there are no standards or specifications to follow when implementing resource monitoring.

The experience from the practical work in this area has showed one major drawback with the implemented system: The lack of resource usage information on application level. If it would be possible to measure and show for example the CPU utilization that is used by a specific application it would allow a much more advanced load balancing algorithm and increase the possible use of the resource monitoring system.

Q3: Which quantities in a DySCAS system are involved in the concept of power management?

The current requirements and specifications of DySCAS do not specify any quantities related to power management that a DySCAS system needs to implement. The resource quantities that were implemented in the resource monitoring system, especially those regarding the CPU, are in many ways important to the concept of power management, but in addition to the resource usage it is also necessary to know the cost (in terms of energy consumption) of using these resources.

It can also be concluded that before a DySCAS system can implement power management, the hardware platforms and the connected devices need to have built-in support for power-saving or sleep mode and the operating system needs to support a dynamic power management policy.

8.3 Future Work

The implemented resource monitoring system is far from perfect. Due to time limitations, several simplifications were made to the system. This section describes some suggestions of topics to focus on in future work on the system.

Resource usage on application level

As concluded in Section 8.2, the major drawback of the current resource monitoring system is the lack of resource usage information for specific applications. Currently it is only possible to get information about the whole node and not for specific processes. If the system could be extended to include process-specific resource usage data it would be very beneficial. The main issue here is that the concept of processes differ greatly between the platforms and operating systems that are targeted by SHAPE. A flexible and platform-independent interface for process information needs to be created.

Porting collection interface to OSE5

The initial intention was to be able to implement the resource collection interface for both Linux and OSE. Due to time constraints and technical issues with SHAPE, nothing
was implemented for OSE5. The reason why this is needed, in addition to the fact that
the system needs to be available for both platforms, is that the portability of the system
needs to be tested and verified.

**Measuring network bandwidth**

One issue when using the network throughput is that the value cannot be compared
to anything. The system should be extended to include a parameter called network
bandwidth as well. This parameter represents the maximum capacity of the network
connection. It could either be defined as the theoretical maximal bandwidth (for example
100Mbit/s when using 100BASE-TX Ethernet) or it could be measured by sending a file
between two nodes in the system, for example during system start-up. The latter choice
gives a much more accurate value but is more complex to implement.

**Improved load balancing**

The last suggestion for future work is to implement a better load balancer. The current
load balancer is, as stated previously, very simplistic. Basically it always decides to
migrate applications from the node with the highest load to the node with the lowest
load. One desirable behavior from the load balancer is that it should try to estimate
the new load numbers that the nodes will get before the application is migrated to see
if it is worth the effort. One reason that the current load balancer does not behave like
this is because the resource monitoring system does not supply any way to monitor the
load caused by a specific application. Once this issue is resolved, the load balancer can
be more advanced. However, it should also be noted that designing a load balancing
algorithm is a topic large enough to have its own thesis.
Bibliography


Appendix A

Graphs

Figure A.1: System CPU load with load 1m as parameter and ten seconds interval
Figure A.2: System applications state with load 1m as parameter and ten seconds interval

Figure A.3: System CPU load with load 1m as parameter and 60 seconds interval
Figure A.4: System applications state with load 1m as parameter and 60 seconds interval

Figure A.5: System CPU utilization with utilization as parameter and ten seconds interval
Figure A.6: System applications state with utilization as parameter and ten seconds interval
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Oskar Hermansson