IT 09 014
Examensarbete 30 hp
April 2009

Policies and Checkpointing in a Distributed Automotive Middleware

Joakim Hägglund

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Policies and Checkpointing in a Distributed Automotive Middleware

Joakim Hägglund

The goal of this master thesis is to analyze different methods for checkpointing with rollback recovery and evaluate how these would perform in a distributed heterogeneous automotive system. The conclusions from this study lead to a design of a checkpointing system for the Selfconfigurable HighAvailability and Policy based platform for Embedded system (SHAPE) middleware together with an API to use this new functionality. This design has been implemented in SHAPE to prove that it works. The conclusion is that this design should be enough for most applications in SHAPE although it requires some extra work by the application programmer.

This report also covers some aspects of policy based computing and presents a design which aims to simplify the usage of policies. This results in a system with a policy manager and a context manager which is implemented in SHAPE.
This implementation was shown to greatly improve the usability of policies and contexts without removing any functionalities.

Handledare: Detlef Scholle
Ämnesgranskare: Ivan Christoff
Examinator: Anders Jansson
IT 09 014
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **5**
1.1 This Thesis 5
1.1.1 Purpose 5
1.1.2 Analytical Approach 5
1.1.3 Technical Approach and Goals 6
1.2 Delimitations 6
1.3 Method 6
1.4 Previous Work 6

2 DySCAS/SHAPE **7**
2.1 The Project 7
2.2 The Platform 7
2.3 Use Cases 8

3 Analysis **9**
3.1 Policies 9
3.1.1 Concept 9
3.2 Versioning Support 11
3.3 Checkpoints 11
3.3.1 Implementation Level 11
3.3.2 Checkpoint Consistency 12
3.3.3 Input/Output 13
3.3.4 Dynamic Systems 13
3.3.5 Piecewise Determinism 13
3.3.6 Checkpointing Techniques 13
3.3.7 Comparison 17
3.3.8 Other Uses 19

4 Design **21**
4.1 Introduction 21
4.2 Policies 21
4.2.1 Contexts 21
4.2.2 Application Policies 22
4.3 Checkpointing 23
4.3.1 Local Checkpoint Manager 24
4.3.2 Global Checkpoint Manager 24
4.3.3 Saving the State 24
4.3.4 Serialization 25

4.3.5 Restoring 25

5 Implementation **27**
5.1 Introduction 27
5.2 Policies 27
5.2.1 Context Manager 27
5.2.2 Policy Manager 28
5.2.3 Signal changes 29
5.2.4 Using policies 29
5.3 Checkpoints 30
5.3.1 Checkpoint API 30
5.3.2 Common List 30
5.3.3 New Signals 31
5.3.4 Using Checkpointing 31

6 Conclusions and Future work **33**
6.1 Conclusions 33
6.1.1 Checkpointing 33
6.2 Future Work 33
6.2.1 Policies and Contexts 33
6.2.2 Checkpointing 34

A Acronyms **35**

Chapter 1

Introduction

This master thesis is a part of the DySCAS[1] research project and is created by Joakim Hägglund at the Computer Science program at Uppsala University. The research project itself will be discussed in detail later in this report. This thesis and report are created at ENEA during the spring of 2008.

1.1 This Thesis

1.1.1 Purpose

This master thesis is divided into two different areas within the DySCAS project and aims at investigating the following subjects:

- Application Policies(AP), investigate how application policies should be implemented in a dynamically self-configuring automotive system.
- Checkpointing(CP), investigate how checkpoints and rollback recovery should be managed in a distributed dynamic automotive system.

1.1.2 Analytical Approach

In the literature study this master thesis shall investigate how application policies can be implemented in a distributed environment and analyze these methods to see which solution would be preferable to implement in a dynamically self-configuring system. It shall also investigate how sharing of a local context between processes within a distributed application can be performed. The thesis aims at answering the following question: How should application specific policies be implemented in a distributed system?

A second objective of this master thesis is to analyze different approaches of checkpointing and rollback recovery in distributed software and investigate how this could be implemented in a dynamically self-configuring system. It shall consider the concepts of saving and restoring states by using checkpoints as well as analyzing how policies of different versions and feature sets should be managed. The question is: How should checkpointing be implemented in the SHAPE demonstration platform?

[1] Dynamically Self-Configuring Automotive Systems

1.1.3 Technical Approach and Goals

This master thesis aims at fulfilling the following two goals in terms of implementations in the demonstrator platform:
The first goal is to implement application policies allowing the system to:

- Provide application level policies and a mechanism for processes within an application to use a common context. Also, an existing GPS application shall be adapted to use policies to demonstrate that it works and, if there is time available, develop a policy usage methodology.

The second goal is to implement checkpointing in order to get:

- A system capable of storing its current state in a checkpoint and restoring previously saved checkpoints during runtime.

1.2 Delimitations

Due to the limited time of twenty weeks, this master thesis will not be able to cover all aspects of the subjects discussed. Also because this thesis is based on the SHAPE reference platform some effort is required to study the previous work. The area of saving and restoring checkpoints is a huge project to implement and will only cover the issue of how a state should be saved or restored, not when.

1.3 Method

This master thesis is divided into four major parts. The first part contains this introduction and background information of Dynamically Self-Configuring Automotive System (DySCAS) and SHAPE to give the reader the information needed to familiarize with the system.

The second part is the theory chapter. This part contains the analytical section where theories and solutions to the different problems are discussed and compared against each other.

The third part consists of a description of the design of the planned implementations.

Lastly the fourth chapter covers the implementation where the practical part of this thesis is presented. Last in this report the results and future work will be discussed, but this is not considered as a separate part but rather a summary of the other parts.

1.4 Previous Work

Since this master thesis is a part of the DySCAS project, it is heavily influenced by work done by the DySCAS consortium members and by previous theses that have been made at Enea.

Chapter 2

DySCAS/SHAPE

The complexity of automotive systems are increasing at a high rate at the moment with mobile devices creating ad-hoc networks with the built-in devices. While the complexity increases, the user interfaces still need to be as simple and intuitive as possible, therefore a platform capable of hiding the complexity while retaining the functionality is most desirable. This calls for a scalable system that can be dynamically reconfigured. At the moment there are no vehicle electronic architectures capable of this, therefore the DySCAS project was started.

2.1 The Project

The DySCAS project is a collaborative research project involving the companies Enea, Volvo, Daimler Chrysler and Bosch and the universities Royal Institute of Technology, University of Greenwich and University of Paderborn[5]. The project started in June 2006 and will end in November 2008. The project is funded by the European Commission. ENEA's role in the project is to develop a reference platform, which will be discussed in more detail in section 2.2. The general goal of the DySCAS project is to develop core concepts and architectural guidelines for developing self-configurable automotive electronic systems.

2.2 The Platform

This is a description of the state of the reference platform called SHAPE before this master thesis started. The platform will later be modified in the implementation phase of this thesis.

The DySCAS architecture is a layered structure based on the conceptual architecture[4]. At the highest level is the application layer where applications and application services are running. Application services are processes that provide some sort of functionality to the system. Applications are processes that use the functionality provided by the services. The middleware is responsible for matching applications with the appropriate services.The Applications and services shall use the DySCAS API for all middleware functionalities and should not be aware of anything below it.

All communication uses message passing and no processes are allowed to share memory. The middleware makes communication between processes com-

pletely transparent and processes do not know if the process they are communicating with reside on the same node or not.

Below the application layer is the middleware layer which is the primary layer of the SHAPE platform. This is where all the SHAPE functionalities are implemented. The code in the middleware layer should be completely platform independent and use the system API to abstract away any platform dependencies.

The last layer is the instantiation layer and this is the only platform dependent layer, making SHAPE highly portable since this is the only code that needs to be modified to get the SHAPE middleware to run on a new OS or architecture.

2.3 Use Cases

The DySCAS project specifies four generic use cases[3] that form the common functionality of the system. These use cases are:

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

The Generic use cases consist of several specific use cases which describe concrete scenarios. One specific use case describes one scenario and belongs to exactly one generic use case. A specific use case could for example be "Load Balancing".

The next layer is the system functionality layer, which is a set of abstract system functionalities that are required to perform the use cases. Each functionality belong to exactly one generic or specific use case. Each use case may require many functionalities. E.g. the specific use case "Redundant ECU" requires the following functionalities among others "Determine when and how reconfiguration should occur" and "Mechanisms to perform the reconfiguration".

The functionalities maps to a set of system requirements that the middleware needs to support. Some functionalities maps to the same requirement and one functionality may also map to several requirements. While the system functionality can be redundant, each system requirement should be unique.

An overview of how the DySCAS use cases, system functionalities and system requirements are correlated is shown in figure 2.1.

Figure 2.1: The DySCAS Use Cases and Requirements

Chapter 3

Analysis

3.1 Policies

The idea behind policies is to be able to dynamically change the behavior of an application depending on external factors without the need to recompile and reload it. This type of behavior could also allow for applications to change the behavior of themselves and other applications. This could for example be used for self learning systems.

3.1.1 Concept

Policies works by separating parameters and logic from the actual implementation of an application. The policy system itself contains a *policy engine* that controls the system according to the *policy rules* and *context*. The policy rules are the logic and the context is a set of parameters.

Kepart and Walch[8] state that a system in state S can move to a state σ by taking the action a. The system will end up in different states σ_i depending on the action a_i, see figure 3.1. Executing a policy will perform one such action or a set of actions, but the way to decide what action to perform differs between different policy types. There are three types of policies, *action policies*, *goal policies* and *utility function policies*.

Action Policies An action policy is the simplest form of policy. It usually has the form IF(Condition)THEN(Action). Condition is a specific state or a set of states for which the action should be taken. The programmer writing the policy need to know what state will be reached by taking the action and determine how desirable it is since the policy itself has no knowledge about it.
Advantages: Action policies are easy to define and the system does not need to calculate what actions to take.
Disadvantages: The writer of the policies need to have low level knowledge about the system. There can be conflicting actions and the system needs to be able to detect and resolve conflicts.

Goal Policies This could be seen as the opposite of action policies, instead of specifying the action, goal policies describes a desired state or set of states

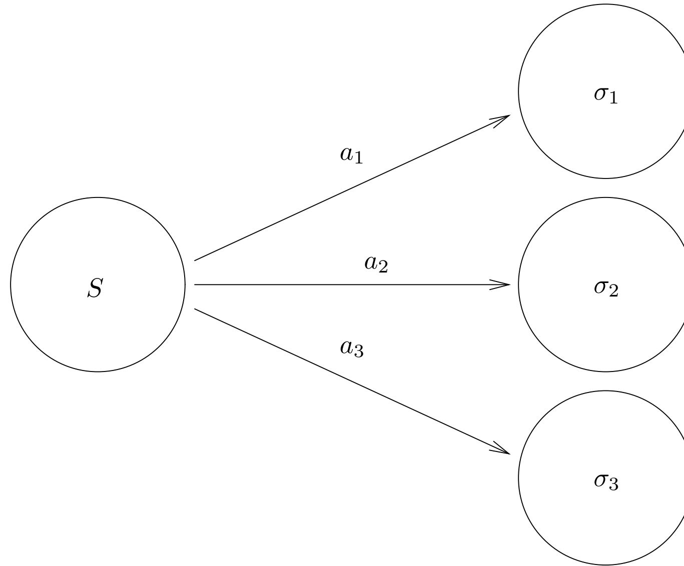

Figure 3.1: States and actions

σ. The system must then calculate what actions to take to get to one of the specified states from the current state S. All goals are equally preferable and the system does not differentiate between them, a state is either acceptable or not.
Advantages: The writer of the policies does not need any information about the system so the policy rules are general and easy to specify. The system will also be able to choose the best actions according to the current conditions, and the writer of the policy does not need to know anything about the context available for the system.
Disadvantages: The system must be able to calculate what actions to take to get to state σ from state S. If there are many goals and not all can be fulfilled at the same time, the system can not decide what goals to meet. The system can sometimes take unwanted actions to reach a goal state, therefore there needs to be a way to specify what actions the policy can use.

Utility Function Policies A utility function policy works much like goal policies, but with the difference that all goal states instead of being deemed acceptable or not, gets a weight. The state with the highest weight is chosen as the target state and the system decides what actions to take to get there. The weight is not predefined for the states, instead it is calculated based on the current state in the system.
Advantages: The utility function can determine how valuable a state is for the system and decide which state σ to try to reach. It is possible to implement goal policies with utility functions by only using a binary weight for the states. If no goal states are reachable, the utility function can choose the best reachable bad state.
Disadvantages: Calculating a correct weight value for a state is a very complex operation. If the weight value is not correct, the policy is likely to take unde-

sirable actions. The weight of state S might also be dependent of the weight of state σ.

3.2 Versioning Support

Versioning support is used to be able to keep multiple versions of applications and choose which one to use. An example of where this could be wanted is in an automotive system where a special version of an application with an extended feature set might be required when a trailer is attached instead of a simpler and less resource demanding version. The version support system needs to keep track of dependencies between different versions of different applications to assure compatibility between components in the system.

3.3 Checkpoints

Checkpointing is a technique that saves the state of a set of processes in a system to be able to restart the system from that state at a later time. The state is not necessarily a state that has occurred in the system, but it must be a legal state that could have occurred in the system before the checkpoint is taken. Different protocols have different rollback properties, some can restore to the latest consistent state that was seen in the system while some only restore to a state that is known to be legal, for definition of legality see section 3.3.2. This is often used to achieve fault tolerance, by taking periodic checkpoints and rollback to the latest checkpoint when an error is detected, the system can recover from a multitude of faults, including hardware failures. The action of restoring a checkpoint is usually called rollback-recovery.

While the main reason to use checkpoints is supply failure recovery, it can also be used for other purposes i.e. process migration. This and other uses will be covered briefly in section 3.3.8.

3.3.1 Implementation Level

According to Plank[10], checkpointing can be implemented at three different levels in the system, namely by the operating system, transparent user-level and non-transparent user-level.

OS-Level

On this level the checkpointing is performed by the operating system. Normally this can be done entirely without any special functionality in the process being checkpointed since the operating system has access to the memory area of the process and all its open file descriptors and such. The downsides with this approach is that the entire memory area of the process needs to be stored and also that the operating system is not aware of when it is an appropriate time to take a checkpoint.

Transparent User-Level

Here checkpointing is performed by the application itself, usually by using a special compiler or by rewriting the executable files.There are some things the programmer of the application needs to take care of, for example, process id's and file structures might change so the application must not assume that the environment is static.

Non-Transparent User-Level

In this approach the checkpointing is entirely performed by the application itself by letting it use special functions and libraries. This way the checkpointing can be precisely controlled by the programmer but it also places a large burden on the programmer. The programmer can also decide what data to store in the checkpoint so only the data necessary for recovery is written, thus minimizing the size of the checkpoint.

3.3.2 Checkpoint Consistency

A state is said to be consistent if and only if it is a state that could occur in the system, although it is not necessarily a state that actually has occurred in the system. A message that has been recorded as received in the checkpoint must also be recorded as sent by the sender process in the checkpoint for the state to be consistent. The opposite is not necessary though, a message that has been sent but not received can be seen as a lost message unless the system assumes reliable communication. In the case where the system does assume reliable communication, all such messages need to be recorded as in flight messages. System models are usually visualized as in figure 3.2 with horizontal lines representing the process executions, arrows between the lines represents messages sent between processes and dotted lines represents a "cut". In this model no arrow should cross the cut line from the right to the left in a consistent state.

If a process has received a message in a state where the message has not yet been sent, the receiver has to rollback to an earlier checkpoint to "unreceive" that message. This might cause other processes to have to rollback further until a global consistent checkpoint is reached, this is called a domino effect. In the worst case the system might have to rollback to the start of the execution. There are different ways to avoid this problem, as we will see in section 3.3.6

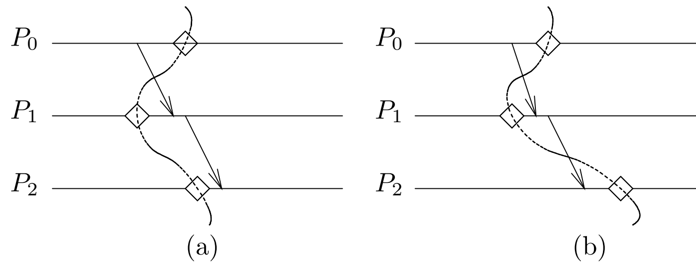

Figure 3.2: (a) shows a consistent cut while (b) shows an inconsistent cut.

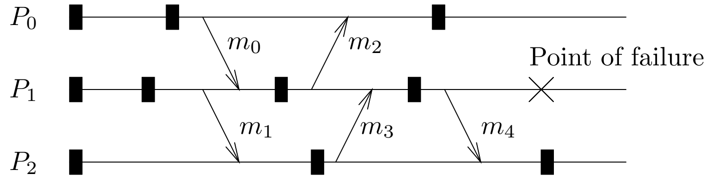

Figure 3.3: Domino effect, when P_1 crashes P_2 needs to rollback to unreceive m_4 causing P_1 to further rollback

3.3.3 Input/Output

With the introduction of Input/Output (I/O) in a system new obstacles appear. All input from a user should be recorded to avoid forcing the user to resend its input, and in some cases the input unit might not even be able to rollback and resend. Perhaps even more important is output commit [6], that is, all messages that have been sent to an output unit must be in the checkpoint, for example, a printer can not unprint a character and a user can not unread a message.

3.3.4 Dynamic Systems

In a dynamic system further complications occur during recovery. Since the configuration of the system might change between checkpointing and recovery, the recovery process can not assume that all processes can be recreated. Also if the system is a heterogeneous system no assumptions about architecture or network topology can be made. Most of this need to be handled anyway in such a system, so the system should be able to assist with functionality to solve these issues.

3.3.5 Piecewise Determinism

Most protocols assume Piecewise Determinism (PWD) in the system, meaning that the execution between nondeterministic events(receiving messages etc.) is deterministic. Piecewise determinism can be achieved in a nondeterministic system by either forcing the system to take a checkpoint after each internal nondeterministic event or by converting all such events to external events that the checkpointing protocol can track.

3.3.6 Checkpointing Techniques

The techniques used for saving and restoring states can be divided in to two different families, Checkpoint-based and Log-based recovery[6]. Checkpoint-based recovery techniques works by saving the complete state. Log-based protocols also save states but additionally also records messages and events which can be replayed at recovery.

Checkpoint-based Recovery

Uncoordinated Uncoordinated checkpointing is the simplest technique for saving states. In this method all processes takes checkpoints individually whenever they can. The benefits of this are that the processes can wait until it

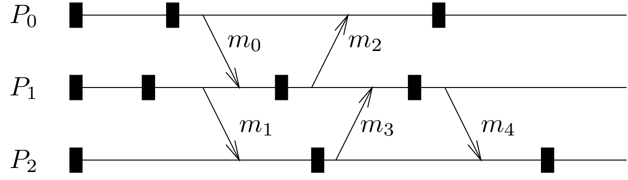

Figure 3.4: Uncoordinated checkpointing

reaches a good point in its execution to store the state. For example when the load is low. Also, this method does not cause any network traffic during checkpointing. The main problem with this is that because there is no interprocess synchronization, the probability of a domino effect is quite high. Since there is no guarantee that a checkpoint can be used, output can not be committed to a known consistent state.

Coordinated In coordinated checkpointing all processes synchronize their checkpointing to form a global consistent state as in figure 3.5. Since all processes start from the latest checkpoint and it is known to be consistent, there is no possibility of a domino effect. Garbage collection with this method is trivial since only the latest checkpoint needs to be saved. Coordinated checkpointing can be either blocking or nonblocking. Coordinated checkpointing can be initialized either by a request from a "master" or if the nodes share a common global time, it can be used to take checkpoints at certain time intervals. The disadvantage of coordinated checkpointing is that a new global checkpoint has to be taken every time there is interaction with the environment to guarantee that I/O-messages are stored, this incurs a very large overhead if I/O-communication is frequent.

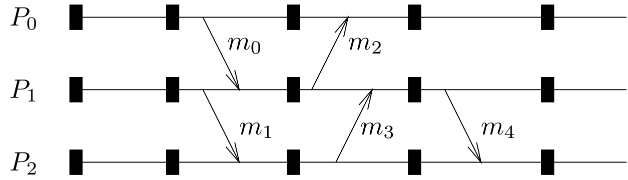

Figure 3.5: Coordinated checkpointing

Communication Induced Communication induced checkpointing does not use special checkpoint messages, but instead piggybacks enough information on the normal messages so the nodes can decide themselves if a checkpoint needs to be taken or not. It is possible to piggyback a checkpoint index on each message sent. When a message is received with a piggybacked index greater than the local index the receiver is forced to take a checkpoint before the message can be processed. The naive way to increment the indices would be to just increase the value by one each time a checkpoint is taken, but this can result in an excessive number of checkpoints to be taken. Another approach is presented by Baldoni, Quaglia and Fornara in [1]. In their solution the index is only increased when

certain conditions are met according to an equivalence equation to reduce the amount of checkpoints.

Log-based Recovery

Log-Based Recovery depends on piecewise determinism in the system where execution is a sequence of deterministic intervals. Each interval starts with a nondeterministic event, such as receiving a message. The protocol saves enough information about the nondeterministic events so they can be replayed on rollback. To reduce the work needed to rollback, local checkpoints are taken periodically in each process. There are mainly three different log-based rollback-recovery protocols, pessimistic logging, optimistic logging and casual logging.

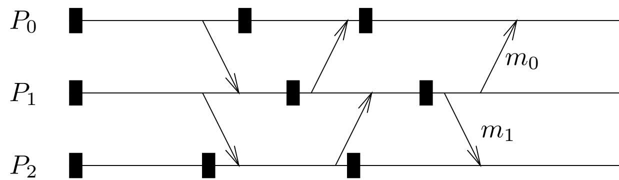

Figure 3.6: Log based recovery, messages m_0 and m_1 needs to be logged since they arrive after checkpoint Y

Pessimistic Logging In pessimistic logging all nondeterministic events are logged. A form of logging called *synchronous logging* is used, meaning that all nondeterministic events needs to be logged before they are allowed to affect the system. With this protocol the system will always be able to recover to its latest state. This gives pessimistic logged four advantages:

- Garbage Collection is simple since nothing before the latest checkpoint needs to be saved.
- Recovery is simplified because only the process that fails is affected. This is because the process will recover to a state that includes its latest interaction with the outside environment.
- Output to the outside environment can be committed without requiring any extra considerations.
- A Lot of flexibility is offered in terms of checkpoint frequency, it would be possible to use only the log replay and still recover. This can be desirable if the checkpoints are large and messages are small and infrequent, then the first checkpoint could be delayed.

But there is also a downside to this, it generates a rather high overhead since data needs to be written to stable storage before each nondeterministic event. Although, there are some optimizations to be found. The normal approach is to let the receiver store messages and all meta data needed to recover, the problem with this is that the node needs to access stable storage. Often the nodes own storage can not be considered stable since it would be lost if the whole node needs

to be replaced due to hardware failure. This means that the receiver would need to send all messages to stable storage possibly doubling the bandwidth usage on the network. One solution to this is to let the sender store the message, in this case only the meta data needs to be sent to stable storage[7] The problem with this is that recovery is only guaranteed for one failure. Multiple failures can be handled in special network topologies with extensions to this technique[12]. Another solution would be to use a special bus with logging capabilities. But with this technique all internal nondeterministic events need to be converted to external messages so that the bus can log them.

Optimistic Logging Optimistic logging protocols use *asynchronous logging* and assumes that the logging event will complete before a failure after each nondeterministic event. Because of this, optimistic logging does not need to block processes before logging and therefore has a lot better performance. The downside to this is more complex garbage collection, output commit and recovery compared to pessimistic logging. With this technique logs are stored in volatile memory and flushed to stable storage periodically. This means that if a node fails, the data stored in its volatile memory can not be used and all nodes that depends on it needs to roll back. Because of this optimistic logging may need to store more than only the latest checkpoint and therefore garbage collection must keep track of all dependencies.

Causal Logging Causal logging tries to retain as many advantages of both optimistic and pessimistic logging as possible. It has about the same failure-free performance[1] as optimistic logging but without making optimistic assumptions. It also isolates processes from the effects of failures in other processes and limits the rollback to the latest checkpoint on stable storage. This comes at the cost of more complex recovery.

The idea is that a process has access to information about each event that causally precedes the current state of the process. The causal order is determined using Lamport's *happened-before* relation[9]. Each process maintains enough information about all events that have affected its state causally to allow it to guide the recovery of a process. For example in figure 3.7(a) process P_0 knows in which order messages m_1 and m_3 should be delivered to process P_1 to allow it to reach state Y. The message itself should be locally available to process P_0 or logged to stable storage.

The issue of tracking causality can be solved by a graph where the nodes are the events and the edges correspond to the *happened-before* relation. Figure 3.7(b) shows how such a graph would look for process P_0 in state X in figure3.7(a). The graph is piggybacked on each message sent. This could cause a really large overhead in reality and should be handled in a better way. Each graph is a superset of the previous graph sent from the same process. This means that when a process p sends a message to process q only the differences between the new graph and the latest message sent between them.

Hybrids Rao et.al [11] has suggested a hybrid solution in which the logs are stored in volatile memory at the sender and also at the receiver in stable storage. The sender uses synchronous logging while the receiver uses asynchronous

[1] This is the overhead of the protocol during normal operation when no crashes occur.

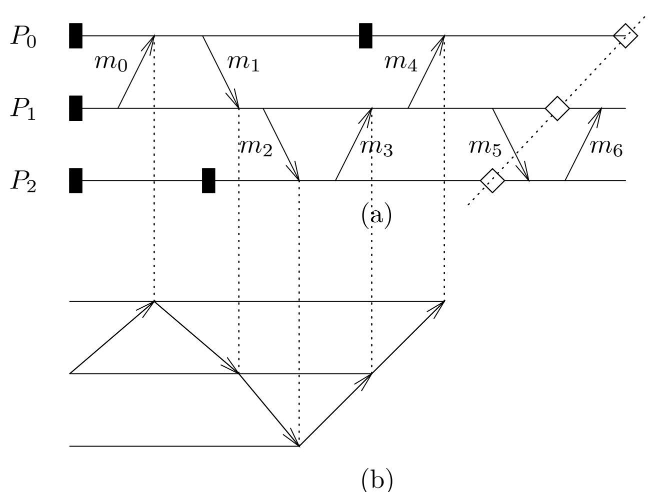

Figure 3.7: Causal logging. (a) Maximum recoverable state and (b) dependency graph ofP_0.

logging. This protocol keeps almost exactly the same performance as normal sender based logging during fail free operation, and maintains close to the same recovery speed as receiver based logging. Another version of this is to perform the same optimizations on causal logging, which gives even better performance.

3.3.7 Comparison

There is no solution that can be said to be better than all the other ones. All protocols have different strengths and tradeoffs concerning performance overhead, storage overhead, simplicity of garbage collection and recovery, freedom from orphans and domino effects, output commit latencies and extent of rollback. This section contains a comparison based on [1, 2, 6, 7, 11].

Overhead

When it comes to storage overhead, coordinated checkpointing is generally the best option, since it only stores the checkpoints and only one per process. Both optimistic and pessimistic logging requires slightly more overhead since the logs need to be stored, the increase in overhead depends on the checkpoint frequency and message sizes. Causal logging needs to store and send the causality tracking graphs, generating some additional overhead. The hybrid protocol produces some more overhead due to the double storing of logs until they are written to stable storage. Uncoordinated checkpointing can have an unbounded amount of checkpoints in the worst case scenario, requiring relatively large amounts of storage in comparison.

The performance overhead of uncoordinated checkpointing is rather low since the processes can decide to perform the checkpointing whenever it is appropriate.

The garbage collection can be quite complex on the other hand. Optimistic Causal The hybrid versions are only marginally slower than their optimistic and causal counterparts Pessimistic logging has the problem of blocking during storage of logs which causes large overhead.

Recovery Performance

Here the checkpoint based protocols can not really be compared with the log-based since those will not reach the same state as the log-based. Uncoordinated checkpoint recovery is very unpredictable due to the risk of a domino effect and there is always the possibility of returning to the start state. And even when no domino effect occurs, the recovery is quite complex. Coordinated checkpoint recovery is trivial because it only consists of loading the latest checkpoint for each process since all checkpoints are part of a global consistent checkpoint.

The following is based on [11] and it should be noted that they consider local storage to be stable, which, for example, means that the receiver-based logging does not need to access the network to retrieve the checkpoint. It should also be noted that the network they are using is not particularly loaded during operation, and other results may occur with a network with more limited bandwidth. Receiver-based pessimistic logging performs really well and the hybrid version of the sender-based version is just as fast. Optimistic logging performs surprisingly well and beats both sender-based pessimistic and causal logging. Though the hybrid version of the causal logging protocol performs as good as the sender-based pessimistic hybrid.

Conclusion

As noted earlier, no protocol can really be seen as a winner, it all depends on the situation.

	Checkpoint based			
	Uncoordinated	Coordinated	Index	
PWD Assumed?	No	No	No	
Domino Effect Possible	Yes	No	No	
Output Commit	No	Slow	Slow	
Overhead	Lowest	High		
Recovery Speed	Slow	Fast	Fast	
	Log-based			
	Pessimistic	Optimistic	Causal	Hybrid
PWD Assumed?	Yes	Yes	Yes	Yes
Domino Effect Possible	No	No	No	No
Output Commit	Fastest	Slow	Fast	Fast
Overhead	High	Medium	Medium	Medium
Recovery speed	High	Medium	Slower	High

Table 3.1: Comparison between some different rollback recovery protocols

3.3.8 Other Uses

The main use of checkpointing is normally to provide a failure recovery scheme, but there are other uses as well. In this section some examples of other potential use scenarios, suggested by Plank in [10], are presented.

Migration

Process migration can use checkpointing to transfer a process to another processor. But instead of writing the checkpoint to storage, the checkpointing processor sends it to the other processor, which then resumes computation from this checkpoint. After the checkpoint has been successfully sent, the first processor terminates its copy of the process. This can be used for load balancing to relax a heavily loaded processor. Another use of this could be to transfer running processes from a node scheduled for shutdown.

Hibernation

Hibernation could be applied on both process and system level. On a process level a process can be temporarily stopped by taking a checkpoint of its current state and writing it to stable storage and then terminating that process. The execution of the process can then later be resumed by restoring that checkpoint.

On a system level, this could be used to shutdown the system without losing any information. It works simply by storing a global checkpoint and then shutting down, later the system can be restored from that checkpoint.

Post-mortem and replay debugging

Checkpoints can be used to assist in debugging an application. One way of doing this is to create a checkpoint whenever the application exits abnormally, this checkpoint can then be analyzed with a debugger.

A more powerful debugging method, called replay debugging, is to store periodic checkpoints, the application can then be rolled back to any previous state to replay the execution from there. This will of course generate a quite large overhead in both performance and memory usage, especially in a parallel environment since the checkpoints will need to be coordinated and all checkpoints must be stored to be able to jump to any previous state.

Elimination of boundary condition errors

This is more like a special case of error recovery than an alternative use of checkpoints. Sometimes errors occur because of a rare boundary condition caused by a synchronization state that the developer could not foresee. These errors are usually very rare and hard to detect due to their low probability of occurring. This kind of error can be dealt with by using rollback recovery to get to a state just before the error and try again. Since it is statistically unlikely that the system will enter the exact same state again, the error will probably not appear in the second attempt.

Chapter 4

Design

4.1 Introduction

The reference architecture of SHAPE presented in 2.2 is reworked and extended with new functionalities. The new architecture is shown in figure 4.1.

Policies: This section contains a detailed description of how to extend the policy engine to allow applications in the system to use policies in a manageable way.

Checkpointing: This section discusses how Checkpointing is to be implemented in the middleware.

4.2 Policies

The old policy system in SHAPE is not available for applications and it is not very easy to use. Therefore an extension aiming to solve these issues is designed. It also includes a new way of managing contexts and also allows contexts to be stored between executions.

4.2.1 Contexts

Contexts are used to provide policies with information about the current state of the middleware and its environment. An example of this could be the temperature in the engine or the current battery level.

Context vs. Templates

Contexts and templates are essentially the same thing, with the difference that templates are stored between sessions while contexts are only valid during the execution. In the new design templates are handled by creating a context and loading default values from the repository. Hereafter *template* will refer to the

Figure 4.1: The new design of the SHAPE middleware.

default values stored in the repository and not the actual runtime context. Creation of a new template is done by creating a new context with the value to be stored in the template, this context is then stored in the template in the repository. A template can be updated by loading and updating the corresponding context and storing that.

Context Manager

In the previous architecture each process kept their own list of contexts, severely restricting the usability of them. In the new design a *context manager* is introduced. This is a two tier system where the contexts are stored locally in the same node as the process that creates it is located. When the Context Manager (CM) receives a request for a context which does not exist in the local CM, it sends the request to the global CM which in turn redirects the request to the CM that owns that context. That CM then responds with the context directly to the calling process. If a CM receives a lot of request for a certain context from another node(and not that many local requests), that context could be migrated to that node instead. Contexts are used through the interface *dyscas_context.h*.

Second design iteration of Context Manager

During the implementation phase the original design was reworked due to issues with consistency if the original creator is removed among other problems.

In this second iteration of the design of the CM all contexts are forwarded to the global context manager. The global CM stores the contexts in the repository but also locally in a list for faster access. If the list gets too large, the least recently used contexts should be removed from the list and only remain in the repository until needed again. When a context is updated in the global CM, that update is forwarded to all context managers that are subscribing to that context. The global CM also acts as a local CM in its node as in figure 4.2. The local CM's keep their own list of contexts to reduce the latency when an application requests a context. If the context list grows large, contexts that are not commonly used can be removed until they are needed again. When a local CM receives a context update from the global CM it pushes the update to all its subscribers of that context. If the update comes from a local application instead, it is sent on to the global CM.

4.2.2 Application Policies

The application policy system is an extension built on top of the existing policy engine already in SHAPE. Applications can use polices by including the *dyscas_policy.h* interface.

Policy Manager

Policies must first be added to a Policy Manager (PM) before they can be used. Each node contains a local PM which keeps track of all policies used in its node and also caches the most frequently used policies locally for faster access. When a policy is not available in the local PM, the PM tries to receive it from the global PM located in the master node. The global PM fetches the policy from the repository if it is not available locally. All policies get sent to the global PM

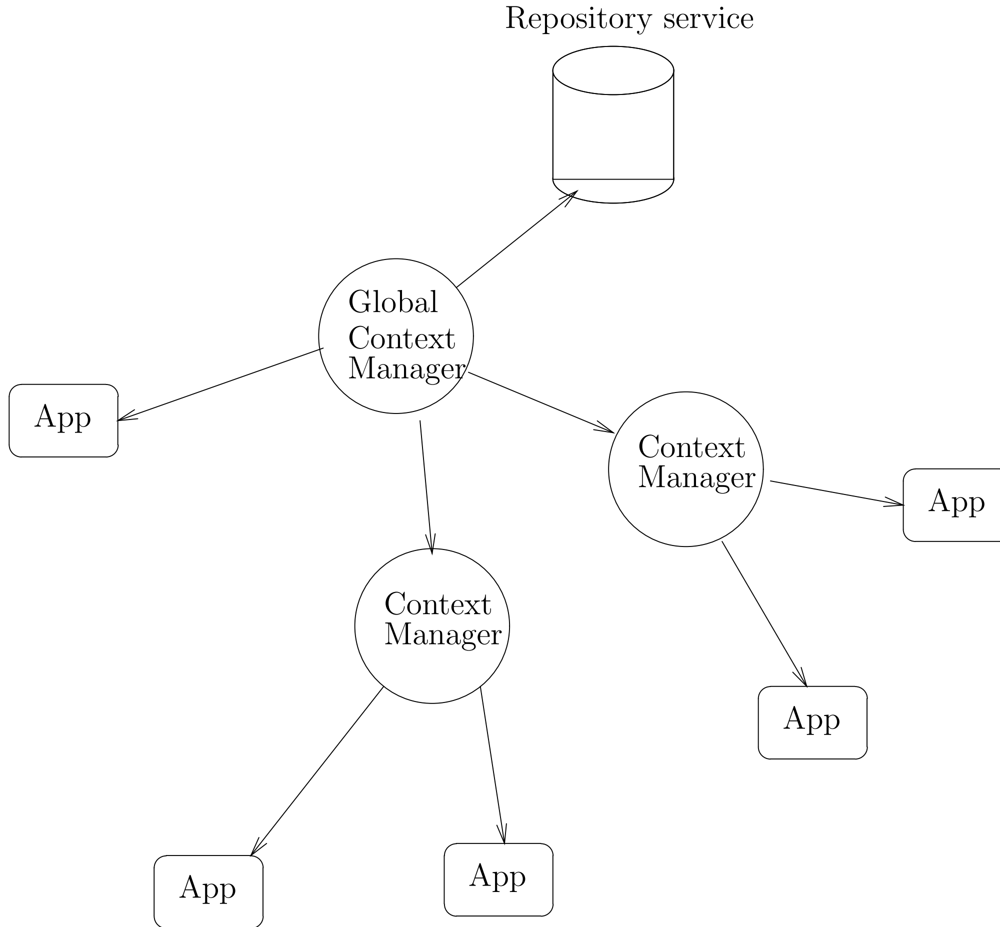

Figure 4.2: Hierarchy of the context managers

when created or updated, and the global PM pushes the policy to all relevant nodes and also saves it in the repository.

4.3 Checkpointing

The checkpointing model is a two tier system. Each node is a system in itself and its processes need to be checkpointed internally. These checkpoints form a global checkpoint for the whole node which is used as a local checkpoint in the distributed checkpointing for the entire system. Processes that want to be checkpointed need to include *dyscas_checkpoint.h*. The local checkpointing is a semi-synchronized model with pessimistic message logging. The checkpoint manager sends checkpoint signals to all registered processes and when a process receives such a signal, it begins its checkpointing procedure.

4.3.1 Local Checkpoint Manager

The checkpointing of each node is controlled by a local Checkpoint Manager (CPM). This manager keeps track of all processes that need to be saved in a checkpoint. It also notifies processes of when it is time to take a checkpoint. The actual checkpointing is controlled by each process itself, this is described in section 4.3.3. The CPM also handles the communication with the repository for the storage handling of checkpoints and logs.

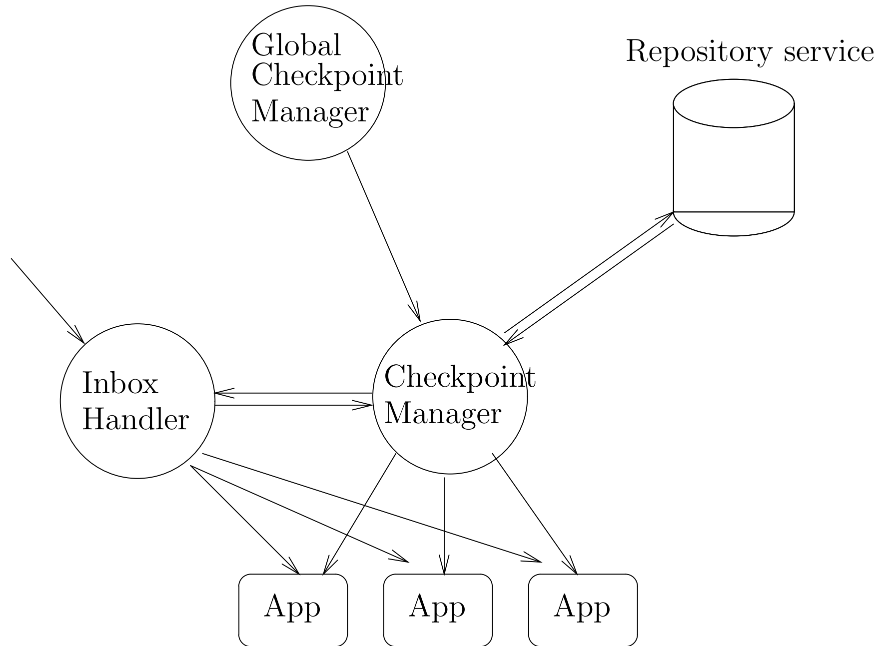

Figure 4.3: The interaction between a local checkpoint manager and applications.

4.3.2 Global Checkpoint Manager

The role of the global CPM is to keep track of the local CPMs. The local CPMs register themselves in the global CPM at startup. When a node needs to recover it is the global CPMs task to send the request.

4.3.3 Saving the State

Large parts of the middleware are mostly stateless and therefore they do not need to store much, but some components require some special attention. These are parts that play a vital role in the checkpointing procedure itself.

Inbox Handler

The inbox handler is modified to capture all messages and make sure they are logged by sending them to the CPM, which stores them in the repository. When

the inbox handler receives an “OK”-signal for the message, the inbox handler can forward the message as usual to the correct process.

Context Manager

The CM (see section 4.2.1) is extended with functionality to save and restore its complete set of contexts. When the context manager has recovered its state it shall verify its lists of subscribers. If a subscriber no longer exists it shall be removed from the list and if the list for a context gets empty, that shall be removed too.

Subscription Service

The subscription service needs to be able to save all the service subscriptions of the node. During recovery the subscription list must be verified to make sure that all subscriptions are still valid.

Saving the Applications

The processes will be responsible of storing its own data and sending it to the checkpoint manager. The process will start the checkpointing as soon as possible after receiving a *DYSCAS_CHECKPOINT_MSG* signal. The applications shall use the provided functions from *dyscas_checkpoint.h* to create the checkpoints.

4.3.4 Serialization

The checkpointed data needs to be serialized by the process before it is sent to the checkpoint manager. Also, the process must be able to restore the serialized checkpoint when recovering its state. This is performed by the supplied checkpoint functions found in the *dyscas_checkpoint.h* interface.

4.3.5 Restoring

Recovery of a node is started by sending a *DYSCAS_RECOVERY_START_MSG* signal to the checkpoint manager of that node from another node. The checkpoint manager then guides the recovery of all the processes in its node. The recovery sequence is as follows:

1. Start up
   If the system is not already running, it should be started in the normal way. During startup the system will send a *DYSCAS_CHECKPOINT_REGISTER* signal to the master node. If this node is already registered for checkpointing the master will assume that the node failed and force a recovery of that node. This is the normal way to start a recovery procedure.

2. Begin Recovery
   The recovery process of a node is initiated by sending a signal of type *DYSCAS_RECOVERY_START_MSG* to the Checkpoint manager of that node. The first step of the recovery is to stop all messages entering the system by halting the inbox handler. Only recovery related messages are allowed to be delivered.Also the outbox is disabled, since all messages that the node wants to send during the recovery phase should have been sent

already and also received by some other node in the system, the exception being the recovery related requests, which are filtered out and sent in the normal fashion. The checkpoint manager loads the meta data needed for the recovery from the repository, this contains a list of all processes to recover in the node.

3. Recovery of the Middleware
   Recovery of all middleware specific functions is initiated by the checkpoint manager, this includes reloading of policies and contexts for example. Before this is started the checkpoint manager starts to prefetch the actual checkpoints from the repository, which sends it directly to the recovering processes. When recovery has started for all processes, the checkpoint manager waits for confirmations from all of them, which they send when they have completed recovery.

4. Restore Applications
   When the checkpoint manager has received OK from all Middleware processes it sends *DYSCAS_RECOVERY_START_MSG* to all applications that need to be recovered.

5. Replay Messages
   While the applications are loading the checkpoints the checkpoint manager prepares the inbox handler for the message replay. When a process has finished the loading of its checkpoint it will start receiving messages and the inbox handler will start delivering them one at a time, in the order they where delivered during the real execution.

6. Resume Normal Execution
   When the message cue for a recovering process has been completely replayed, the inbox handler and the outbox handler are enabled for that process again and it can resume normal execution.

Chapter 5

Implementation

5.1 Introduction

This chapter describes the additions that have been made to the SHAPE reference platform during this thesis.

5.2 Policies

A new layer has been built on top of the existing policy system and some parts have been redone. The policy subscription is now a multi tier system with local caching of policies for faster access. The same has been done with the context management and a new context manager which resides in every node has been introduced. Along with this a complete API for using the new managers has been introduced, this will be described in 5.2.1 and 5.2.2.

5.2.1 Context Manager

The context manager hierarchy consists of a local context manager in each node which stores all contexts that are relevant to that node, together with a list of all local subscribers for each context. The context manager in the master node also acts as a coordinator for the context management and makes sure that all local CMs contains updated versions of all contexts. The following function calls where implemented in *dyscas_context.h*

Adding/Updating Contexts

dyscas_create_context: Sends a signal to the context manager with a request to create a new context with the supplied identifier and value.

dyscas_update_context: Sends a signal to the context manager with a request to update the value of the context with the specified identifier.

Requesting Contexts

dyscas_get_context: Requests the value of the context with the supplied identifier and returns the value when it is received from the context manager.

Un-/Subscribing Contexts

dyscas_subscribe_context: Sends a subscription request for a context, the context manager will then send updates for that context to the calling process when the context is changed.

dyscas_unsubscribe_context: Sends a request to unsubscribe a context, the context manager will then stop sending further updates to the requesting process.

Saving/Loading Templates

dyscas_save_template: Saves a context in a template in the repository.

dyscas_load_template: Loads a template from the repository and creates a new context from its values.

Remove a Context

dyscas_del_context: Removes a context from the system.

5.2.2 Policy Manager

Each local policy manager keeps a local set of policies relevant to its node and sends all updates to the global policy manager located in the master node. The master node forwards all updates it receives to all nodes affected by the update to make sure all PMs contains up to date versions of all policies. The following function calls where implemented in *dyscas_policy.h*

Adding/Updating Policies

policy_add: Adds a new empty policy with the supplied policy id to the policy manager.

policy_add_rule: Adds a new rule to an existing policy.

policy_update_rule: Replaces an existing rule with a new rule in a policy.

Preloading Policies

policy_register: This forces the policy manager to subscribe to the requested policy to make it ready for use.

Evaluating Policies

policy_evaluate: Evaluates a policy by using the Decision Evaluation Module (DEM) through the policy manager and returns the resulting value of the evaluation.

policy_evaluate_w_input: Evaluates a policy with input provided by the application by using the DEM through the policy manager. The call returns the resulting value of the evaluation.

Swapping Policies

policy_swap: This swaps the identifiers of two policies, thus making all applications that are using this policy identifier swap policy to evaluate.

policy_swap_version This loads a different version of the policy and replaces the previous one with this one for all applications.

5.2.3 Signal changes

The signals for contexts and policies remain mostly unchanged, the fields *context_node* and *context_process* have been removed since contexts do not belong to a certain process. Also the field *context_value* has been renamed and is now only called *value*. Policy and context signals have been separated in *dyscas_policy.sig* and *dyscas_context.sig*.

5.2.4 Using policies

A policy manager has been created and it is designed to make the usage of policies as transparent as possible for the application programmer. Policies can either be stored in the repository and then be loaded when needed(*policy_register*), or the application itself can build them and send them to the PM(*policy_add*, *policy_add_rule*).

When a new rule is added to a policy (*policy_add_rule*), the PM sends a subscription request for any contexts that are needed to evaluate that rule. One of the largest changes here is that the old system required the applications to keep its own list of policies and contexts instead of letting the policy managers and context managers handle that part.

When an application reaches a decision point it asks the PM to evaluate the policy (*policy_evaluate*,*policy_evaluate_w_input*)to decide which function block to execute. The PM then evaluates the policy using the DEM and responds to the application with the result from the DEM.

Policy API	
dyscas_policy.h	dyscas_context.h
policy_add	dyscas_create_context
policy_add_rule	dyscas_update_context
policy_update_rule	dyscas_get_context
policy_register	dyscas_subscribe_context
policy_evaluate	dyscas_unsubscribe_context
policy_evaluate_w_input	dyscas_save_template
policy_swap	dyscas_load_template
policy_swap_version	dyscas_del_context

Table 5.1: The new function calls in the policy api.

5.3 Checkpoints

Each node contains a checkpoint manager which makes sure that the concerned nodes takes checkpoints and restores them. There is also a global checkpoint manager which controls the local CMs and orders them to start a recovery in their node if required. Some new basic helper functions to build checkpoints have been created to lessen the effort needed by the application programmer to perform the checkpointing. The following function calls were implemented in *dyscas_checkpoint.h*.

5.3.1 Checkpoint API

sys_serialize_signal: Serializes an entire signal including its header and stores it in a checkpoint signal.

sys_rebuild_signal: Rebuilds a signal including header informs from a serialized signal stored in a checkpoint signal.

save_state: Takes a set of variables and serializes them and stores the serialized data in a checkpoint signal.

restore_state: Sets a set of variables according to the data from a previously serialized signal.

dyscas_checkpoint_register: Registers a process for checkpointing, meaning that the checkpoint manager will send out save requests to it and also restore it if a recovery occurs.

5.3.2 Common List

create_liststorage: Creates a storage room for storing lists in the repository.

create_subscrstorage: Creates a storage room for storing list subscribers in the repository.

save_subscribers: Serializes all elements in a subscriber list and stores them in the subscriber storage room in the repository.

restore_subscribers: Fetches all subscriber posts in the repository for a certain element and restores the subscriber list of an element.

save_list: Serializes all elements in a list and stores them in the list storage room in the repository. It also saves all the subscribers for each element using the *save_subscribers*-function.

restore_list: Fetches all elements belonging to a list from the repository and rebuilds the list including all elements subscriber sets.

Policy API
dyscas_checkpoint.h
sys_serialize_signal
sys_rebuild_signal
save_state
restore_state
dyscas_checkpoint_register

Table 5.2: The new function calls in the checkpoint api.

5.3.3 New Signals

The checkpointing only makes use of one new signal type, namely the:

DYSCAS_CHECKPOINT: This signal is used for all checkpoint related communication. To distinguish between different checkpoint messages an action field is used, the different actions are as follows:

- REGISTER_FOR_SAVE Used to signal that a process wants to register itself for saving.
- BEGIN_SAVE Tells the receiving process to start saving its state.
- SAVE_COMPLETE Tells the checkpoint manager that a process has completed its save procedure.
- START_RECOVERY Orders a process to start recovering. Also used to make a local CPM begin recovery of its node.
- RECOVERY_COMPLETE Informs the checkpoint manager that a process has finished recovering its state and is now ready to resume.
- CHECKPOINT This is a serialized state or signal packeted into a signal.
- SAVED_SIGNAL Indicates that a signal has been successfully logged in the repository.

5.3.4 Using Checkpointing

A checkpointing manager has been added to the middleware to control the state saving and restoring of applications. For an application to be saved it needs to register itself in the local checkpoint manager (*dyscas_checkpoint_register*) this will cause the checkpoint manager to save the process id in its list of checkpoint capable processes.

The checkpoint manager will periodically send save requests (*BEGIN_SAVE*) to all processes in its list, causing them to save their state as soon as the message is received. The applications saves their state by first saving its lists (*save_list* in *dyscas_mw_common_list.h*) and then serializing and saving its state using *save_state*. This function call takes a special string and all variables to save as arguments. The string identifies the type of the variables using the form

```
'b' = U8
's' = U16
'd' = U32
```

When the checkpoint manager receives a request to recover the state of the node(*START_RECOVERY*) it fetches its process list from the repository recovers that. After that it fetches all relevant checkpoints from the repository using the contents of this list and forwards the checkpoints to the correct processes. When a process receives a checkpoint to recover it starts by recovering any lists (*restore_list*) and then by calling *restore_state* with the checkpoint received as one argument and the same string and set of variables as used to *save_state* as the rest of the arguments. When this function has completed the recovery it notifies the checkpoint manager (*RECOVERY_COMPLETE*). During this time the checkpoint manager waits for all processes to finish before it resumes operation as normal.

When a node is started its checkpoint manager registers itself with the global checkpoint manager. If this checkpoint manager already exists in the list of registered checkpoint manager the global checkpoint manager assumes that the node has exited abnormally and forces it to recover.

Chapter 6

Conclusions and Future work

6.1 Conclusions

6.1.1 Checkpointing

For checkpointing to be really useful it should be considered and implemented at a much earlier stage of development of the middleware. Although checkpointing can still be implemented in a useful way, but this requires that all existing applications are modified. Despite this, adaptation of most of the current applications and middleware components should be quite simple. The reason for this is that most components contain rather small amounts of persistent state data to save. Also in most cases the components resides in the main loop waiting for new signals, and if checkpoints are only taken between actions caused by signals no program counter needs to be saved.

6.2 Future Work

6.2.1 Policies and Contexts

The caching of contexts and policies should use a more efficient data structure than lists to be able to scale properly in more complex systems. Also the policy rules inside a policy should be stored in a sorted manner, this way the DEM would not need to search the whole list for each rule since rules are always evaluated in increasing order.

Security and integrity has not been considered in this thesis, but there need to be a system for access control for both policies and contexts if this is going to be used in a commercial application. Preferably a system with separated read and write access.

One issue with policies is the added overhead compared with normal evaluation, this could be reduced by implementing a two stage evaluation function consisting of a non blocking call which evaluates the policy and stores the result until the application asks for it with a second call. The pre-evaluation could also be redone if a context or input that is affecting the result of the evaluation is

changed before the final result is delivered to the application. The policy manager could also be modified to forward all requests to another node to reduce the load on light weight nodes to allow the use of complex policies on any node. If this is properly done, it could not only reduce the overhead of using policies, but also provide a speedup compared to normal evaluation in some cases.

6.2.2 Checkpointing

There are still things to be done with the checkpointing system and adaptation of the rest of the platform to properly use the checkpointing. The checkpoints should be compressed before storing to reduce storage and bandwidth overhead. The checkpoint manager should check to make sure that all processes complete the checkpointing and recovery procedures and take action if something fails.

Appendix A

Acronyms

DySCAS	Dynamically Self-Configuring Automotive Systems
AP	Application Policies
CM	Context Manager
PM	Policy Manager
CPM	Checkpoint Manager
CP	Checkpointing
DEM	Decision Evaluation Module
DySCAS	Dynamically Self-Configuring Automotive System
I/O	Input/Output
PWD	Piecewise Determinism
SHAPE	Selfconfigurable HighAvailability and Policy based platform for Embedded system
VS	Versioning Support

List of Figures

2.1 The DySCAS Use Cases and Requirements 8

3.1 States and actions 10
3.2 (a) shows a consistent cut while (b) shows an inconsistent cut. . 12
3.3 Domino effect, when P_1 crashes P_2 needs to rollback to unreceive m_4 causing P_1 to further rollback 13
3.4 Uncoordinated checkpointing 14
3.5 Coordinated checkpointing 14
3.6 Log based recovery, messages m_0 and m_1 needs to be logged since they arrive after checkpoint Y 15
3.7 Causal logging. (a) Maximum recoverable state and (b) dependency graph ofP_0. 17

4.1 The new design of the SHAPE middleware. 21
4.2 Hierarchy of the context managers 23
4.3 The interaction between a local checkpoint manager and applications. 24

Bibliography

[1] R. Baldoni, F. Quaglia, and P. Fornara. An index-based checkpointing algorithm for autonomous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):181, 1999.

[2] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Symposium on Fault-Tolerant Computing*, pages 68–77, 1997.

[3] Lead contractor for deliverable: Daimler Chrysler AG (Viktor Friessen). DySCAS Scenario and System Requirements (D1.2). Technical report, Internal DySCAS consortium document, July 2007.

[4] et.al DeJiu Chen. DySCAS System Architecture (D2.1). Technical report, Internal DySCAS consortium document, 2007.

[5] DySCAS. www.dyscas.org, Februari 2008.

[6] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems, 1996.

[7] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The 7th annual international symposium on fault-tolerant computing*. IEEE Computer Society, 1987.

[8] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY*, pages 3–12, 2004.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[10] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, 1997.

[11] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The cost of recovery in message logging protocols. *IEEE Transactions on knowledge and data engineering*, 12(2):160–173, March/April 2000.

[12] S. Venkatesan and Tony T-Y. Juang. Efficient algorithms for optimistic crash recovery. *Distrib. Comput.*, 8(2):105–114, 1994.