FT-PAS – A Framework for Pattern Specific Fault-Tolerance in Parallel Programming

Gopinatha Jakadeesan

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April 2009

Your file Votre référence
ISBN: 978-0-494-63279-6
Our file Notre référence
ISBN: 978-0-494-63279-6

Abstract

FT-PAS – A Framework for Pattern Specific Fault-Tolerance in Parallel Programming

Gopinatha Jakadeesan

Fault-tolerance is an important requirement for long running parallel applications. Many approaches are discussed in various literatures about providing fault-tolerance for parallel systems. Most of them exhibit one or more of these shortcomings in delivering fault-tolerance: non-specific solution (i.e., the fault-tolerance solution is general), no separation-of-concern (i.e., the application developer's involvement in implementing the fault tolerance is significant) and limited to inbuilt fault-tolerance solution. In this thesis, we propose a different approach to deliver fault-tolerance to the parallel programs using a-priori knowledge about their patterns. Our approach is based on the observation that different patterns require different fault-tolerance techniques (specificity). Consequently, we have contributed by classifying patterns into sub-patterns based on fault-tolerance strategies. Moreover, the core functionalities of these fault-tolerance strategies can be abstracted and pre-implemented generically, independent of a specific application. Thus, the pre-packaged solution separates their implementation details from the application developer (separation-of-concern). One such fault-tolerance model is designed and implemented here to demonstrate our idea. The Fault-Tolerant Parallel Architectural Skeleton (FT-PAS) model implements various fault-tolerance protocols targeted for a collection of (frequently used) patterns in parallel-programming. Fault-tolerance protocol

extension is another important contribution of this research. The FT-PAS model provides a set of basic building blocks as part of protocol extension in order to build new fault-tolerance protocols as needed for available patterns. Finally, the usages of the model from the perspective of two user categories (i.e., an application developer and a protocol designer) are illustrated through examples.

Acknowledgements

I would like to express my special thanks to my supervisor Dr. Dhrubajyoti Goswami for all his support, guidance, and encouragement. His valuable suggestions, feedbacks and continuous evaluation throughout the course of this research, especially during difficult times, helped me greatly to progress with my work. I am thankful for his advice and the precious time he spent helping me amidst his tight schedule; without him this thesis would not have been completed.

My special thanks to my parents for their love and advice, continuous support and encouragement. I am grateful to them for their blessings and all they have given me throughout my life.

I would like to thank my family members, friends and colleagues for their help and support.

Thanks also to alma maters and all the staff at Concordia University and everyone who helped me in any way.

Finally, I am grateful to The Almighty for His blessings.

Table of Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 The Problem 1

1.2 Objective 3

1.3 Contribution 4

1.4 Organization of the Thesis 5

2 Background and Related Works 7

2.1 Various Fault-Tolerance Techniques 7

2.1.1 Replication Mechanism 8

2.1.2 Checkpoint Mechanism 9

2.1.3 Logging Mechanism 14

2.2 Fault-Tolerance in Parallel Systems 15

2.2.1 FT-MPI 16

2.2.2 MPICH-V 16

2.2.3 Charm++ 17

2.3 Fault-Tolerance in Pattern-Based Parallel Systems 17

2.3.1 Muskel 18

2.3.2 Persistent Fault-Tolerance for the Divide-and-Conquer Application 19

2.3.3 MPI Farm Library 20

2.3.4 CoHNOW – FT-DR 20

3 Pattern-Specific Fault-Tolerance 22
3.1 Pattern-Specific Fault-Tolerance Classification 22
3.1.1 Task Farm Pattern 23
3.1.2 Master-Slave Pattern 29
3.2 Protocol Discussion 33
3.2.1 Checkpoint Gradient 33
3.2.2 Checkpoint Dependency Graph 34
3.2.3 Gradient-based Checkpoint Protocol 35
3.2.4 Extended Protocol: Color-based Checkpoint Protocol 39
4 Introduction to FT-PAS 45
4.1 PAS Overview 45
4.2 User Categories and their Roles 48
4.3 Introduction to the FT-PAS Model 49
4.3.1 Overview 50
4.3.2 Specificity 51
4.3.3 Separation of Concern 52
4.3.4 Protocol Extension 53
4.3.5 Generic Group Definition 54
5 FT-PAS Design, Usage and Case Study 56
5.1 Framework Architecture 56
5.1.1 The FT-PAS Architecture 57
5.1.2 Framework Assumption 59
5.2 Design of the Framework Internals 60

5.3 Protocol Extension: Primitives, Usages and Case Studies 70
5.3.1 Overview of the Protocol Extension 70
5.3.2 Primitives for the Protocol Extension 72
5.3.3 Framework Usages and Case Studies 78
6 Evaluation 90
6.1 Environment 90
6.2 Experiences on the Framework Usages 91
6.3 Experimentation and Results 92
6.3.1 Framework Overhead 93
6.3.1 Comparison of the Different Fault-Tolerance Protocols 95
7 Conclusion and Future Research 99
Bibliography 101

List of Figures

Figure 1: Message types 10

Figure 2: Pattern-specific fault tolerance classification 23

Figure 3: Minimal state retention in a task-farm pattern 29

Figure 4: Gradient-based checkpoint protocol 36

Figure 5: Color-based checkpoint protocol 41

Figure 6: PAS skeleton 46

Figure 7: FT-PAS skeleton and its phases 49

Figure 8: FT-PAS skeleton and its components 51

Figure 9: Example of generic group mapping 55

Figure 10: General view of the framework 57

Figure 11: Architecture of the FT-PAS framework 58

Figure 12: High-level view of the framework 58

Figure 13: Conceptual view of the fault-tolerant parallel architectural skeleton 59

Figure 14: Framework internals 60

Figure 15: Failure detection monitor - post detection procedure 63

Figure 16: Interaction between dependency analyzer and other components 65

Figure 17: Three phase consistent checkpoint coordination 67

Figure 18: Recovery module and its action 69

Figure 19: High-level class diagram of gradient-based checkpoint protocol extension ... 79

Figure 20: Gradient-based checkpoint protocol - protocol behavior 80

Figure 21: Gradient-based checkpoint protocol - failure reactor 80

Figure 22: Gradient-based checkpoint protocol - marshaller 81

Figure 23: Gradient-based checkpoint protocol class 82

Figure 24: Gradient-based checkpoint protocol - service registration 83

Figure 25: Gradient-based checkpoint protocol - application developer's perspective.... 84

Figure 26: High-level class diagram - fault-tolerance protocol for iterative problem..... 85

Figure 27: AbstractIterator and State interfaces 86

Figure 28: Iteration-based application level checkpoint protocol class 87

Figure 29: Iterator with fault-tolerance actions 88

Figure 30: Fault-tolerant iterative application – application developer's perspective..... 89

Figure 31: Overhead due to logging 94

Figure 32: Overhead due to checkpointing 95

Figure 33: Fault tolerance overhead percent - varying message localization density 96

Figure 34: Overhead ratio - varying message localization density 97

Figure 35: Overhead comparison – application-level and system-level checkpoint 98

List of Tables

Table 1: Overhead incurred with and without the simple logging protocol 93

Table 2: Overhead incurred with and without checkpointing..... 94

Table 3: Fault tolerance overhead - varying message localization density 96

Table 4: Overhead comparison – application-level and system-level checkpoint..... 97

Chapter 1

Introduction

In this chapter, we describe our research objective and summarize what is achieved.

1.1 The Problem

The advancements in computer hardware and high-speed networks have revolutionized the concept of building powerful clusters using networks of workstations. The networked-workstation clusters are more popular and used as a common environment for parallel-computing. This is due to their cost-performance benefit, and their suitability for solving a vast range of computational intensive problems using their combined computing powers. This is in contrary to the high priced sophisticated parallel-computing environment which is made from special parallel computers.

This paradigm shift towards the networked-workstation cluster has triggered interesting challenges for researchers in various aspects. One such problem with the use of the networked-workstation cluster is lack of reliability as this is made from off-the-shelf components.

Fault-tolerance is essential for a long running parallel application in order to avoid computational wastage. It must be noted that achieving fault-tolerance in parallel application is complex. There are various reasons and challenges for the previously mentioned difficulty in achieving fault-tolerance in parallel applications.

- There is no clear standard defined regarding fault-tolerance support in the parallel programming environment. For example, in the message-passing parallel programming environment, the Message Passing Interface (MPI) standard [46] defines some general error handling mechanism intended mainly for resource clean-up action, rather than from the perspective to support fault-tolerance.
- In most of the existing MPI parallel-programming environments (e.g., LAM-MPI [4], MPICH-V [5] etc), the solution to tolerate fault is addressed in general and is not specific to a problem category. Such a solution can lead to a performance problem.
- Addressing fault-tolerance issues in specific at the application development phase is tedious. The parallel-programming environment such as FT-MPI [1] requires significant effort from an application developer to achieve fault-tolerance in a given parallel program. It deviates the application developer's objective from the application development.
- Moreover, the fault-tolerant solution provided in the existing system (such as MPICH-V [5], etc.) is fixed and rigid. Such solution experiences closeness issues, i.e., a given application is constrained to use the fault-tolerant strategy that is provided within. Hence, it is not possible to build and add a new fault-tolerance protocol.

Various researches have been conducted to resolve some of the above mentioned difficulties. Our research focuses on a specific approach that is based on patterns. We believe that patterns can indeed be used to provide fault-tolerance support for parallel

applications. All of these have motivated our research toward building a new model to address the above stated challenges.

1.2 Objective

Although separating fault-tolerance implementation concern is an important benefit, most of the existing parallel-programming system (LAM-MPI [4], etc.) leads to undesirable performance overhead due to its fault-tolerance solution's generic nature. Such generic fault-tolerance solution might fit well for certain problems but leads to bad performance for others. In general, most of the existing system does not support fault-tolerance in a problem category specific manner. Hence, it is necessary to provide such support in order to check undesirable performance overhead.

There also exists an extended MPI implementation (e.g., FT-MPI [1]), which provides a basic facility to implement application specific fault-tolerance but at the cost of considerable involvement on the part of the application developer, such as saving system-states, logging communication messages, etc. It is tedious to address such system-specific issues (i.e., fault-tolerance) at the application development phase. Hence, it is necessary to alleviate or liberate such burden during the application development phase in order to focus on the application development rather than on the fault-tolerance issues.

Most of the existing systems (MPICH-V [5], Muskel [24], etc.) support a limited and fixed set of fault-tolerance protocols. Consequently, there is no facility provided to add new fault-tolerance as needed for available pattern implementation. Thus, if an application demands a different fault-tolerance protocol which is not supported, generally

the application has no alternate choice but to use what is available or to abandon the idea of specifically providing fault-tolerance. Such system imposes limitations and thus leads to undesirable performance problems for applications. Hence, it is necessary to provide support to build a new fault-tolerance strategy as needed.

1.3 Contribution

Through this research, we have contributed patterns classification based on fault-tolerant strategies. This classification is novel to the best of our knowledge. We have designed and implemented a model – the Fault-Tolerant Parallel Architectural Model (FT-PAS) – to demonstrate and verify our concepts (separation-of-concern, protocol extension). The FT-PAS model contributes: (1) to assist in application-specific fault-tolerance in a programmer transparent/semi-transparent way, and (2) to provide a test bed in order to build new fault-tolerance strategies and to evaluate the performance overhead of the fault-tolerance strategies.

We believe different fault-tolerant techniques are well suited for different patterns. This pattern-specific fault-tolerance solution indirectly contributes to overcome undesirable performance overhead, which is incurred when employed with a non-specific solution. In FT-PAS, we achieve separation-of-concern by pre-packaging pattern-specific fault-tolerance strategy implementation in an application-independent manner. This notion facilitates in separating the fault-tolerance implementation issues and alleviating burden from the application developer. Lastly, we address the closeness issue by supporting a fault-tolerance protocol extension. In FT-PAS, we contribute a set of core facilities as building blocks in order to design and integrate a new fault-tolerance strategy for

available patterns. Thus, it provides greater flexibility to build new fault-tolerance strategy at ease, for use with available patterns.

The FT-PAS model introduces two user categories in order to support a fault-tolerance protocol extension: a protocol developer (responsible for implementing the new fault-tolerance protocol) and an application developer (responsible for using the available fault-tolerance protocol).

We evaluate the model implementation from two aspects: usage and performance. The usage of the framework is evaluated based on the easiness of implementing various fault-tolerance strategies. Subsequently, the performance of the framework is evaluated by measuring and comparing the overheads incurred from different fault-tolerance strategies.

1.4 Organization of the Thesis

The thesis is organized as follows: Chapter 2 provides background knowledge and related works. Chapter 3 discusses our novel classification of a few well-known patterns into sub-patterns based on fault-tolerance strategies. Subsequently, we discuss two protocols for a sub-pattern along with their correctness proof. Chapter 4 introduces our FT-PAS model with discussion on two user categories: the protocol developer and the application developer. The subsequent section discusses three important aspects which are addressed in the model related to fault-tolerance. Chapter 5 discusses the model architecture, design and usages from a two user group perspective. This chapter also includes a discussion on the framework internals. Then, the core facilities related to protocol extension and their primitives are discussed, along with case studies on several protocols and their usages.

Chapter 6 includes a discussion of the environment and the results that are observed from our experiments are evaluated. Chapter 7 concludes the thesis and introduces the future research direction.

Chapter 2

Background and Related Works

In this chapter, we introduce background knowledge and related works. This chapter is divided into three sections. In Section 2.1, we introduce background details related to various fault-tolerance techniques that are practiced in general. The following section provides a brief review on few existing MPI-based parallel-programming environments that support fault-tolerance. Section 2.3 describes some of the related pattern-based parallel-programming works that provide fault-tolerance.

2.1 Various Fault-Tolerance Techniques

In this section, we discuss various fault-tolerance techniques and their background. Fault-tolerance in a single process system is achieved by saving the current state for later recovery using a contemporary checkpoint/restart system implementation. Many checkpoint/restart libraries are available such as Libckpt [9], PSNC checkpoint library [10], Condor checkpoint library [11] and BLCR (Berkley Lab's Checkpoint Restart) [12]. These systems are different in various aspects. To mention a few: the amount of state saved is different, the medium of storage is different, the API is different, etc.

Achieving fault-tolerance in a parallel-distributed system is much more complex. This is due to the fact that such applications involve more than one process. These processes communicate and exchange information in order to solve a given problem. Using the

checkpoint libraries (e.g., PSNC checkpoint library, Condor library) alone would not be sufficient in order to provide fault-tolerance. There are various mechanisms that are proposed in [13] to achieve fault-tolerance for such distributed systems. At a higher-level, they are classified into three categories: replication mechanism, checkpointing mechanism and logging mechanism. There are other variances of these techniques available. They are derived either from one or more of the above mentioned mechanisms. Similar fault-tolerance techniques can equally be applied to the parallel programs with additional emphasis on performance and scalability.

2.1.1 Replication Mechanism

Replication is one of the well-known techniques used to achieve fault-tolerance. As illustrated in [14], it can be classified into two subcategories: active replication and passive replication.

In the active replication type, there exist one or more backup nodes running in parallel to a primary node. Each backup node receives all necessary inputs as received at the primary. Independently, each node computes and generates results. Consequently, all the nodes compare their results and take consensus regarding the correct output. They might as well execute the byzantine algorithm to handle byzantine faults [22]. Other than computation, each backup node monitors the primary node to detect failure.

Similarly, in the passive replication type a.k.a. primary-backup technique, there are backup nodes. But unlike the active replication, here the backups do not compute in parallel to the primary node. Instead, the primary sends all the necessary application

states to all the backups in order to keep them up-to-date. Thus, the backups which hold the necessary system software and application states are ready to take over from the primary in case of failure.

The replication scheme is costly as it nearly doubles the expenses without increasing the computational capacity [15]. Instead, it is possible to replicate the modules at a finer level rather than to replicate the entire machine as in [23]. This proves to be comparatively cost effective but at the expense of overhead due to hosting a replica in the processing node.

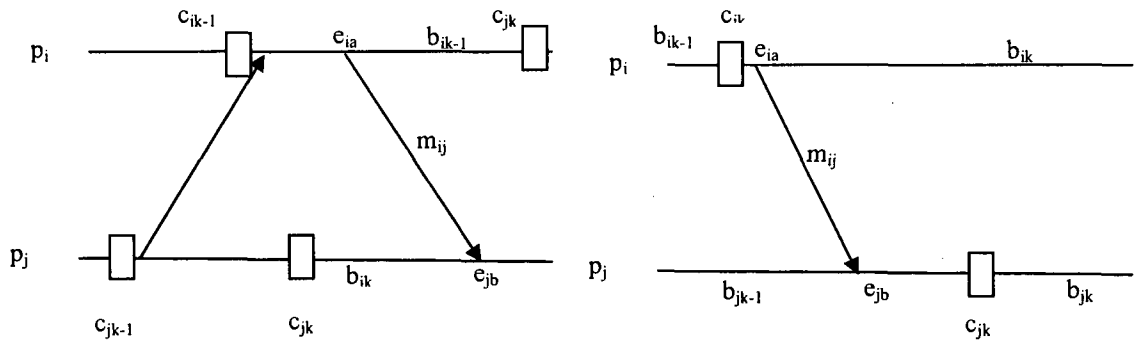
2.1.2 Checkpoint Mechanism

Checkpoint mechanism is a commonly used technique to provide fault-tolerance in the distributed and parallel systems. First, we discuss some general terms and definitions related to this mechanism, which are used in later chapters. Then, we present the classification of the checkpoint mechanism.

A global state is a set of process states, which represents the snapshot of the system at an instance [42]. In global checkpointing, the global states are recorded periodically as a set of checkpoints. A checkpoint refers to a process state saved during the failure-free operation. There are two concepts related to consistency, each represented as a message type: an in-transit message and an orphan message [42, 43, 44].

Definition 1 *An in-transit message can be defined as a message which is sent but not yet received in a given global state [43]. Formally, we refer a message* m_{ij} *from* p_i *to* p_j *as an in-transit message if* $e_{ia} \rightarrow e_{jb}$ *such that* $e_{ia} \in b_{ik-1}$; $e_{jb} \in b_{jk}$.

Where, (i) $\rightarrow$ denotes happened-before relation [17]. (ii) e_{ia} and e_{jb} are the send-event and receive-event of the message m_{ij}. (iii) The behavior of a process containing the events that occur between two checkpoints during the process execution is termed *behavioral fragment*. For example, the behavior of a process 'j' containing the events that occur after the checkpoints c_{jk-1} but before c_{jk} is denoted by the behavioral fragment b_{jk} associated with the checkpoint c_{jk}. Similarly b_{ik-1} is the behavioral fragment corresponding to the checkpoint c_{ik-1}. (iv) c_{ik-1} is the k-1th checkpoint at a process 'i' and c_{jk} is the kth checkpoint at a process 'j'.



(a) In-transit Message **(b) Orphan Message**

Figure 1: Message types

Definition 2 *An orphan message can be defined as a message whose receive event is recorded but not the send event in a given global state [43, 44]. Formally, we refer a message m_{ij} from p_i to p_j as a potential orphan message if $e_{ia} \rightarrow e_{jb}$ such that $e_{ia} \in b_{ik}$; $e_{jb} \in b_{jk-1}$.*

Where, (i) e_{ia} and e_{jb} are the send-event and receive-event of the message m_{ij}. (ii) b_{ik} and b_{jk-1} are the behavioral fragments corresponding to the checkpoint c_{ik} and c_{jk-1}

respectively, and (iii) c_{ik} is the k^{th} checkpoint at the member 'i' and c_{jk-1} is the $k\text{-}1^{th}$ checkpoint at the member 'j'.

An orphan message can occur in a system during recovery upon using an inconsistent global snapshot. A message can be prevented from becoming orphan when both its send-event and receive-event are placed in the same global snapshot during failure-free execution.

Definition 3 *A consistent cut is a set of checkpoints in which if a process's checkpoint state reflects a message receipt, then the checkpoint state of the corresponding sender reflects sending that message [13, 17].*

In specific, the orphan messages do not exist in the consistent checkpoint [42, 43]. In case of the in-transit messages, they are either nonexistent or exist to be replay-able during the recovery operation. In addition, all the determinants of the non-deterministic events should be replay-able during the recovery operation.

Definition 4 *Consistent checkpoint is said to be strong if it does not contain the in-transit messages [42].*

As mentioned earlier, one way to handle the orphan message is to take precautionary measures to prevent a message from becoming orphan. Similarly, all the in-transit messages and the determinants [45] of the non-deterministic events (if any) should be recorded to help in replay during the recovery operation.

Checkpoint mechanisms can be classified into three subcategories: uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing. They are discussed as follows.

(i) Uncoordinated Checkpointing

The uncoordinated checkpointing allows each process to decide independently when to take checkpoints. This protocol is also known as independent checkpointing. This autonomy allows processes to execute the checkpoint operation when their state information is small. The major drawbacks of this protocol are susceptibility to domino effect [45], useless checkpoints, and storage overhead due to multiple checkpoints.

This protocol constructs and maintains a graph to identify a consistent cut during the recovery operation. Two such graph models are identified here: the dependency graph [20] and the checkpoint graph [21]. They are constructed during the failure-free execution based on the message-send and the message-receive. In case of failure, these graphs help in recovering the failed process and rolling-back the dependent processes to a consistent recovery line.

(ii) Coordinated Checkpointing

The coordinated checkpointing requires processes to collaborate in order to form a consistent global state. In case of failure, all the processes are rolled-back to a most recent checkpoint during recovery execution. They are not subject to the domino effect [45]; hence, the recovery procedure is simplified. It reduces storage overhead as only one checkpoint is maintained on the stable storage. However, such protocol may incur

overhead due to the coordination action. The coordination can be achieved by different means: blocking and non-blocking.

A simple approach to the coordinated checkpointing is to block execution and communication of all the processes while executing the checkpoint protocol [16]. This protocol is called blocking checkpoint coordination. In the non-blocking checkpoint coordination, the protocol does not block the communications. Instead, the coordination is achieved by sending an explicit checkpoint-request message preceding the first post-checkpoint message on each link. This way each process is forced to take a checkpoint upon receiving the first checkpoint-request message. A well-known example of such non-blocking checkpoint coordination protocol is the distributed snapshot algorithm proposed by Chandy and Lamport [17].

Some protocols use marker [18] or checkpoint indices [19], which are piggybacked along with the post-checkpoint message in order to achieve coordination.

(iii) Communication-Induced Checkpointing

The communication-induced checkpointing mechanism avoids the domino effect without requiring the coordination action. This protocol generates two types of checkpoints: local and forced. Local checkpoints can be taken independently, while the forced checkpoint is taken to guarantee the progress of the recovery line. In specific, the forced checkpoint avoids creation of the useless checkpoints. Here, no explicit coordination message is exchanged. Instead, the coordination message is piggybacked along with the application

message. The receiver decides with the piggybacked information whether to take a forced checkpoint. [13] presents few other protocol variations of this technique.

2.1.3 Logging Mechanism

The checkpoint techniques discussed above are expensive due to various reasons: the process execution gets blocked, flattening the process state is time consuming and storing data on the stable storage is space consuming. The log-based mechanism tries to minimize or liberate these overheads. The log-based rollback recovery makes an explicit assumption based on the piecewise deterministic model [45], based on which all the non-deterministic events can be identified and their respective determinants can be logged during the process execution.

The messages contribute largely as non-deterministic events in the message passing system. During the failure-free execution, the determinants of such identified non-deterministic events should be logged on to a stable storage. In case of failure, the failed process should be able to recover by replaying the logged determinants.

A variant of the above procedure is possible, which can be thought of as an amendment with checkpoint to reduce the amount of replay to quicken the recovery. More flavors of the logging scheme are possible based on the place where the message logging is executed on either the sender-side or the receiver-side. In addition, the way a message is logged, i.e. synchronously or asynchronously, leads to a different variant. [13] presents a detailed discussion on various log-based recovery mechanisms.

2.2 Fault-Tolerance in Parallel Systems

In this section, we discuss few existing parallel-programming environments which are based on MPI (Message Passing Interface) [46] that supports fault-tolerance. MPI is a specification for message passing in the parallel-programming domain. The current MPI specification does not address in depth fault-tolerance like the case where a process fails in the MPI environment. MPI provides two choices for failure handling: (1) In the default option, the application can abort immediately on occurrence of any failure; (2) In the second option, the application is provided with the flexibility to continue execution but with no guarantee that any communication can occur further.

The intended purpose of the second option is to provide flexibility to the application in doing cleanup locally before it terminates. Hence, this might not be sufficient to implement the fault-tolerant techniques which are discussed in [13].

In spite of limitation with the current MPI specification, the different approaches to provide fault-tolerance in MPI programs are discussed in [2]. Each of these approaches has shortcomings due to its limitation to use for a specific program structures. Different fault-tolerance techniques are targeted for different purposes which include process fault-tolerance as in FT-MPI [1], network failure recovery as in LA-MPI [28], message logging technique as in Ediga [29], checkpoint/restart technique as in Starfish [30], CoCheck [31], MPICH-V [5], LAM/MPI [4], Charm++ [3], etc. Few of these implementations which support fault-tolerance are discussed in the following sections.

2.2.1 FT-MPI

FT-MPI [1] is built upon HARNESS (Heterogeneous Adaptable Reconfigurable Networked Systems), a fault-tolerant computing environment. The goal is to provide a communication library in the form of MPI-API while benefiting fault-tolerance from the HARNESS system. The FT-MPI implements a complete MPI-1.2 specification and some parts of the MPI-2. It is aimed at providing a fault-tolerant MPI implementation, which can survive failures. It modifies and extends the semantics of the MPI to provide various intermediate states to help fault recovery in FT-MPI. This way it provides ability to alter the internal state in order to recover from failure in applications.

In FT-MPI, when an error state is identified with a communicator, the new communicator follows one of these semantics: shrink, blank, rebuild, or abort based on its failure mode. More information on the semantics and modes can be found in [1]. The communicator follows a continue/no-operation message mode in the midst of error. From the usage point of view, the fault-tolerance can be achieved by making the error check and corrective action from the implementation effort.

2.2.2 MPICH-V

MPICH-V [5] is a research effort to provide multiple fault-tolerance protocols on the MPICH implementation. It provides automatic fault-tolerance without altering the application. It uses a mix of checkpointing in conjunction with message logging to save the process state and to automatically recover the failed processes. It introduces the use of checkpoint servers, dispatchers and event loggers, which assist in alleviating the fault-

tolerance overhead. The different versions of the MPICH-V implementation support different protocol types. MPICH-V provides flexibility to the end-user in choosing a required protocol during installation.

2.2.3 Charm++

Charm++ [3] is an object-model based approach to the parallel application design and development. It is based on C++. Processor virtualization is one of the core techniques used. It aims at improving the performance of the application, the productivity of the programmer and the scalability. Moreover, an 'Adaptive MPI' version has been implemented conforming to the MPI standards [46]. It provides fault-tolerance support through various schemes: (1) On-disk checkpoint/restart—this approach involves a synchronized checkpoint scheme with a centralized server to store checkpoints on persistent stable storage. It supports only manual restart; (2) Double-memory checkpoint/restart - this approach involves synchronized check-pointing to save states using in-memory stable storage and automatic restart; (3) Double-disk (local) checkpoint/restart - this approach is very similar to the previous approach except that the storage is on persistent local disk; and (4) Message logging schemes - this approach involves message logging on the in-memory scheme with automatic restart without requiring any checkpoint synchronization.

2.3 Fault-Tolerance in Pattern-Based Parallel Systems

In this section, we discuss the pattern-based parallel systems that support fault-tolerance. Design Patterns gained popularity in the field of object oriented design after the

publication of *Design Patterns: Elements of Reusable Object-Oriented Software,* the book authored by Gamma et. al. [32]. In subsequent years, the use of pattern has been explored in almost every domain. In general, design patterns are about providing solutions to commonly recurring problems using the knowledge gained from experience in software design and development.

Patterns gained acknowledgment in parallel-programming through experimentation from different works over the past few years. eSkel [7] is one such early work in the form of algorithmic patterns introduced by Cole at the University of Edinburg from the algorithmic perspective. PAS (Parallel Architectural Skeleton) [8] is a novel approach towards patterns from the architectural/structural perspective. Each skeleton in PAS is an implementation model of patterns in parallel programming. They are provided with pattern-specific primitives, such as communication-synchronization, etc. More information on PAS and its background is discussed in Chapter 4. There exist various other parallel-programming models like Muskel [24], MPIFarm [26], etc. Some of the related works dealing with fault-tolerance are discussed in the following sections.

2.3.1 Muskel

Muskel [24] is a Java structured parallel-programming environment evolved from the Lithium parallel-programming environment targeted for grids. The environment provides run-time support for controlled quality-of-service. An application manager provided with the environment takes care of delivering the quality-of-service to the application. The same has been demonstrated using two structured patterns: the task farm and the pipeline. The application developer is expected to define quality-of-service in terms of contracts.

For example, in the task farm, the performance-contracts used include the parallelism degree and throughput maintenance in terms of the tasks processed per unit time. Similar contracts are defined for faults such as recruiting a new processing resource to substitute a missing one. Thus, the environment delivers quality-of-service to an application dynamically for the user defined contracts.

2.3.2 Persistent Fault-Tolerance for the Divide-and-Conquer Application

The mechanism discussed here focuses on delivering fault-tolerance to an application which operates based on the divide-and-conquer pattern [25]. The fault-tolerance mechanisms are demonstrated with Satin, a Java framework for grid-enabled divide-and-conquer applications. In Satin, the problem decomposition by recursive division leads to entries in the work pool in each processor. The works are distributed across the processors by work stealing: an idle processor steals jobs from the work pool of the other processors. It is obvious that jobs that are stolen from the leaving processor lead to the orphan job problem.

The system provides two fault-tolerance mechanisms to handle such fault. In the first mechanism, in the orphan saving technique, the orphan jobs are handled by saving them in-memory in an orphan table along with the results. Thus, a recovering process does a lookup on the orphan job table to re-use the saved results. But this mechanism does not support the total-loss or the suspend-resume of the application. This results in the proposal of a second mechanism; this strategy is similar to the orphan saving technique but with a minor amendment. It writes the partial results to a checkpoint file on a persistent storage rather than in-memory. Thus, it overcomes the shortcoming of the first

mechanism: tolerating the total-loss and supporting the suspend-resume of the application. More details on these techniques can be found in [25].

2.3.3 MPI Farm Library

MPI Farm Library [26] is a parallel-programming library with higher level interfaces targeted for scientific application development. This library is built on top of the MPI implementation and runtime environment. These APIs are better adapted for problem implementation than that of the MPI [46]. But it supports only those applications which follow the task farm pattern. The task farm, a.k.a. task parallel pattern, is a well-known algorithmic pattern. The farm's inherent nature provides added benefit to support fault-tolerance. Since everything is handled through the master, it becomes a natural place to checkpoint. This library prefers portability instead of transparency; hence, it implements the user-driven application-level checkpointing. On recovery from crash, the master replays the results of jobs which are processed earlier. Thus, it forwards only those pending jobs to the workers. More details on this library can be found in [26].

2.3.4 CoHNOW – FT-DR

CoHNOW FT-DR stands for collection of heterogeneous network of workstations, where FT-DR refers to fault tolerance by means of data replication. Here, the workstations are organized in a hierarchical master/slave scheme. The model includes various logical and execution components for the overall working of the system. The fault-tolerance activities in the model are comprised of three different phases: startup, normal execution and failure recovery. In the startup phase, the activity initializes by replicating the master

information in a protective worker. The worker takes over in case of the master's failure. In the normal execution phase, it involves data replication, fault monitoring and detection. In case of failure, the job related to the faulty worker is reassigned to an available worker. It also involves fault handling of various other internal components. More details on this model can be found in [27].

All the above discussions provide background knowledge on various aspects such as fault-tolerance techniques in general, fault-tolerance in the parallel systems and other related works from the patterns perspective. The general concepts, definitions and techniques that are presented here are referred to and used in later chapters for our discussion.

Chapter 3

Pattern-Specific Fault-Tolerance

As a part of this research, we first classify patterns into sub-patterns based on different fault-tolerant strategies, which are identified based on pattern characteristics. This classification is novel to the best of our knowledge. Unlike existing environments (MPICH-V [5], etc.), the pattern-specific fault-tolerant solution checks the performance overhead, which is incurred when employed with a non-specific solution. We have designed a model and implemented a framework following this classification to demonstrate our ideas. In Section 3.1, we classify patterns into sub-patterns based on the fault-tolerant strategies. In the following section, we discuss two fault-tolerant protocols along with their correctness proof.

3.1 Pattern-Specific Fault-Tolerance Classification

The motivation behind the following discussion is to emphasize our research hypothesis that different fault-tolerance techniques are applicable for different patterns in parallel-programming. In our research, we classify a pattern into sub-patterns based on fault-tolerant strategies. A sub-pattern is a derivative resulted from embedding a suitable fault-tolerant strategy, which is selected based on the problem characteristics. The task farm is a well known pattern, where the fault-tolerant strategies are identified based on the computational intensity of the worker as shown in Figure 2. The fault-tolerant strategies are as follows: (1) Restart recovery and (2) Checkpoint recovery and its variant.

Similarly, in case of the master-slave pattern, fault-tolerant strategies are identified based on communication and synchronization characteristics as shown in Figure 2. They are as follows: (1) Gradient-based checkpoint recovery, (2) Color-based checkpoint recovery, and (3) Application-level checkpoint recovery for iterative problems. Each pattern classification is discussed in detail in the following section.

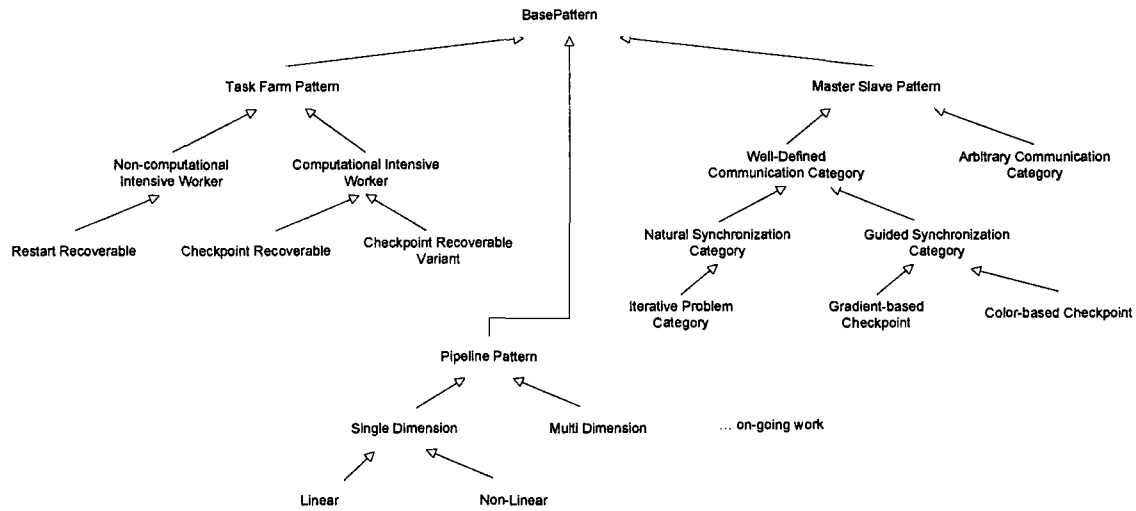


Figure 2: Pattern-specific fault tolerance classification

3.1.1 Task Farm Pattern

The task farm is a well known pattern and is used in many parallel applications. It is also known as dynamic replication pattern. The task farm pattern contains five key components: task pool, result pool, task generator, result collector and workers.

Let X be a problem space that needs to be solved. The task generator decomposes the given problem space X into n independent chunks. Each independent chunk is

represented as x_i, and is computed at a worker. Let $R(x_i)$ be the computed sub-result for an independent chunk x_i. The result collector is responsible for collecting the sub-results.

Based on the characteristics of the range of applications that make use of this pattern, the fault tolerance strategies for the worker can be broadly classified into: (i) restart recoverable and (ii) (independent) checkpoint recoverable. These strategies are based on the workload of the worker relative to the overall problem size. All the components need to be fault-tolerant via checkpointing which include a task pool, a result pool, a task generator and a result collector.

(i) Checkpoint Recoverable Category

Consider a problem space where the decomposed sub-tasks are computational intensive. The time taken to compute such an independent sub-task is significantly large. In such a scenario, re-doing a lost work from the beginning is costlier. Employing an independent checkpoint recovery strategy at each worker can reduce the computation loss significantly. This can be achieved without incurring significant overhead during the failure-free execution for the reasons illustrated below.

Assume that each independent chunk in the task pool is approximately of equal size and all workers are approximately of equal computational capability. Let T be the time to compute an independent sub-task x_i without any overhead. The failure-free execution time for an independent sub-task can be represented as follows, where C_t is the checkpoint time.

$$\text{Failure-Free Execution time } E_t = \begin{cases} T & ; \text{without any overhead} \\ T + C_t & ; \text{with checkpoint overhead} \end{cases}$$

From the above expression, we can infer that the execution time for the failure-free execution can be approximated to T, when C_t is negligibly smaller compared to T (based on the assumption that each sub-task is computational-intensive).

Let N be the total number of checkpoints, I_t be the checkpoint interval, n be the number of checkpoint taken so far and S_t be the computation saved as result of checkpoint operation. They are represented as follows.

$$\text{Checkpoint interval-time } I_t = \frac{T}{N+1}$$

$$\text{Computation saved } S_t = n * I_t \quad ; 1 \leq n \leq N$$

Assume that a failure occurs at execution time t. The computational loss incurred using the checkpoint scheme in a worker can be represented as follows.

$$\text{Computational Loss } L(t) = \begin{cases} t & ; 0 < t \leq I_t \\ t - S_t & ; t > I_t \end{cases}$$

In general, if T is total time taken to compute a sub-task then t represents the computation lost due to failure. As we apply the checkpoint scheme, the loss incurred can be refined as t (when $t \leq I_t$), and $t - S_t$ (when $t > I_t$) based on the time interval during which the failure occurred.

To summarize, the checkpoint overhead does not contribute much to the execution time when compared to the amount of computation it saves when employed with the checkpoint strategy. So, each worker can be independently checkpointed during computation at an arbitrary or pre-defined point based on the characteristics of the application. Hence, during recovery, it should be able to recover from an intermediate recoverable state instead of all over from the beginning.

For example, consider a graphic rendering problem in the field of animation movie production. It uses a render farm, which requires enormous computational power to render thousands of frames. Each rendered frame is time consuming. Thus, it cannot compromise failure in the middle of any single frame rendering. Such applications are categorized as computationally intensive at each worker level. Therefore, each worker needs to save its intermediate state. This can be achieved as either programmer transparent or semi-transparent.

(ii) Restart Recoverable Category

On the contrary, consider a problem space where the decomposed sub-tasks are non-computational intensive. The checkpoint strategy mentioned previously is not suitable due to two reasons: high checkpoint overhead and high recovery overhead.

In such problems, we observe that the computation time of a sub-task is far less compared to the checkpoint time if applied. Thus, the checkpoint overhead contributes significantly to the execution time of the sub-task. In addition, the re-computation cost incurred due to

the checkpoint recovery is comparatively higher than the cost incurred when the sub-tasks are re-computed from the beginning. This can be expressed as follows:

$$T_{ckpt\text{-}recover} > T_{restart\text{-}recover}$$

Where, $T_{ckpt\text{-}recover}$ is the checkpoint recovery overhead, $T_{restart\text{-}recover}$ is the recovery overhead by re-computing the sub-task from the beginning. $T_{ckpt\text{-}recover}$ is expressed as $E_t + R_t + (T - S_t)$, and $T_{restart\text{-}recover}$ is expressed as $E_t + T$. Where, E_t is the environment recovery time, R_t is the time to recover the system state based on the saved checkpoint and $(T - S_t)$ is the remaining time required to finish re-computation on resumption from an intermediate saved state. On substitution, the above expression can be re-written as follows.

$$R_t > S_t$$

For illustration purposes, let us assume that N and n be 1. The expression can be re-written as follows:

$$R_t > T/2$$

From the above deduced expression, we observe that when R_t is greater than T/2 (execution time), overhead incurred from the checkpoint recovery is higher than that incurred from re-starting the sub-task from the beginning.

In general, the problems that fall in this category satisfy the following constraints:

- The workload associated with a given task is so small that the overhead incurred from employing the checkpoint strategy is more than the task execution time.

- The overall problem size is so huge that a worker's failure increases the overall workload in each member. As a result, the time incurred to solve the overall problem also increases.

For example, let us consider the problem of Mandelbrot set [6] based on relation $Z_{n+1} = Z_n^2 + C$, where both Z and C are complex numbers. The overall work that needs to be completed is huge, whereas the individual tasks (each pixel computation) are comparatively small. Thus, failure during processing of an individual task can be redone without any significant computational time loss. However, it requires availability of the workers at all times in order to keep up with the expected completion time. As a result, the workers could be configured to be replaceable with a new worker in case of failure.

(iii) Checkpoint Recoverable Category Variant

As another example of the computational intensive worker that might need a different strategy from the previous one, let us consider a special case where tasks are partitioned and distributed in a single go to the individual workers. The intention is to let a worker compute more than one task at one go, save the partial results locally, and then send the final results back. Therefore, it avoids overhead due to extra communications.

There may or may not be dependencies between the individual tasks assigned to a worker. In either case, there is a need to save the minimal application state as the intermediate state (minimal state retention) between successive processing of the tasks. The application-level checkpointing (saving states of application-specific variables)

rather than system-level checkpointing may be sufficient to recover a failed worker from the preceding task prior to failure (Figure 3).

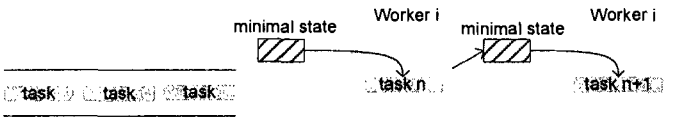


Figure 3: Minimal state retention in a task-farm pattern

The previous discussion shows that the task farm pattern can be further subcategorized based on the fault-tolerance strategies of the workers. There is a need for an additional component: a failure monitor. There can be one or more of these modules based on implementation strategies. All the other modules including the failure monitor are assumed to be failure-free.

3.1.2 Master-Slave Pattern

The master-slave is a commonly used pattern in the parallel programming domain. Here, the slaves are interconnected via a fixed virtual topology (mesh, hypercube, star, etc.). The computational model may or may not be data parallel (e.g., slaves may be performing different tasks), based on the problem at hand. Here, we assume that the number of concurrent computational units required for computation is known at the start.

Each task in this case is usually dependant on a subset of other tasks. These dependencies are resolved via explicit messages within a subgroup of slaves. We call such a subgroup of dependant slaves a communication subgroup, which has localized communication dependencies among the members. There can be more than one communication

subgroup. Fault-tolerance strategies can differ based on the incidence and overlap of these localized subgroups, as discussed in the following section.

We can subcategorize this pattern based on the communication and synchronization characteristics of the slaves. At a higher level they are classified into two, i.e., slaves having (a) *well-defined communication pattern* and (b) *arbitrary communication* (non-pattern). It needs no mentioning that the well-defined communication patterns will create localized communication subgroup(s) among slaves, while arbitrary communication patterns may or may not create proper communication subgroups. Hence, their fault tolerance strategy varies.

(i) Well-Defined Communication Category

There are several applications that involve well-defined communication patterns. In general, the problem that steps into this category meets the following constraints.

- A set of tasks must execute at same time because they require information from other dependent members.
- The communication messages are localized within a subgroup.
- Tasks that are sub-grouped based on message localization or dependencies might be totally independent from tasks of other subgroups.

In this category, each communication subgroup coordinates among its members in order to save checkpoint. In case of failure, all the members of the subgroup are recovered together from the latest consistent cut. Thus, the group checkpoint action and the failure

recovery action are localized within a subgroup, and do not affect other independent subgroups.

The well-defined communication pattern can be further sub-classified based on synchronization characteristics as (a) *natural synchronization* and (b) *guided synchronization.*

(a) Natural Synchronization

Informally, a naturally synchronizing pattern is defined as a pattern that exhibits obvious synchronization points in their behavioral fragments in every member of a subgroup such that inconsistent messages are guaranteed not to exist in the global snapshot saved at that execution point. Thus, such patterns do not require explicit action to synchronize in order to save a consistent global snapshot. Formally, it is defined as follows:

Definition 5 *We refer a pattern as naturally synchronizing if* $B^{(l,k)}$ *is strong, i.e., send event* e^s *in* $B^{(l,k)}$ *implies existence of corresponding receive event* e^r *in* $B^{(l,k)}$ *and vice versa.*

Where $B^{(l,k)} = \bigcup_{i=1}^{n} b_i^{(l,k)}$, $b_i^{(l,k)}$ is a subset of behavioral fragments (discussed previously in Section 2.1.2) of a member i for iteration from l to k.

For example, all iterative based problems such as Jacobi, SOR fall under this category. Providing fault-tolerance for such problems is straightforward. As they exhibit an obvious synchronization point, each member in a subgroup should save its local states at this point to form a consistent global checkpoint.

(b) Guided Synchronization

All the problems that do not exhibit obvious synchronization points are subcategorized under the guided synchronization pattern. Such problems exhibit a well-defined communication pattern with distinct communication subgroups but require explicit action in order to support fault-tolerance, i.e., explicit action to coordinate among the members of the subgroups to save a consistent cut. A suitable non-blocking coordination protocol can be employed using message piggy-backing based on the locality of the messages within the subgroups. Two such protocols related to this category are discussed in the following section: (1) gradient-based checkpoint protocol and (2) color-based checkpoint protocol.

For example, in a specific solution to render the graphics models, two (or more) frames are processed concurrently: the current frame and the speculative processing of the future frames. Each frame is partitioned among a subgroup of slaves using sort-middle or sort-last partitioning strategies [34]. This imposes dependencies among the members of the subgroups. Here, the frames are independent. The subgroups are distinct and each subgroup processes a single frame. The gradient-based checkpoint protocol would fit for such problem category. In certain applications, there exist inter-group dependencies. The intergroup dependencies are resolved via occasional inter-group message exchange. The color-based checkpoint protocol would fit well for such a problem category.

(ii) Arbitrary Communication Category

As an example of the arbitrary communication category, consider the problem of the parallel ray tracing for rendering large scenes. In a data-parallel solution, the problem is geometrically partitioned among the slaves. Based on the movements of the rays (decided

only at run-time), the slaves might require communicating with other slaves in an arbitrary manner. Here, though the communication is arbitrary, the interactions among the slaves are still among the nearest-neighbor, considering the geometric partitioning. Hence, they form the well-defined communication subgroups. These subgroups overlap and merge to create one large communication group, as discussed previously. However, if the partitioning is not geometric, then there is no well-defined communication subgroup. The parallel discrete event simulation is another application domain for arbitrary communications. Traditional coordinated checkpoint protocol [13] would fit well for such problem and uses explicit protocol messages for coordination.

3.2 Protocol Discussion

In this section, we present two modified protocols related to the master-slave pattern. First, we present some required concepts related to the protocols. Next, we discuss the protocols along with their correctness proof.

3.2.1 Checkpoint Gradient

Progressive creation of a new checkpoint leads to the displacement in the most recent state that a process can recover from in case of failure. *Checkpoint gradient* is defined as change in checkpoint number (i.e., checkpoint number of the sender's snapshot) relative to a given reference (i.e., checkpoint number of the receiver's snapshot). Here all the messages are assumed to be tagged with the checkpoint number to indicate the checkpoint snapshot from which it originated.

The gradient can be computed from the tagged checkpoint number of a received message using the receiver's checkpoint number as reference. The *positive gradient* is an indication for a potential orphan message and the *negative gradient* is an indication for an in-transit message. Thus, a received message can be detected as a potential orphan or an in-transit message from the checkpoint gradient computation.

Axiom 1 *A message is detected as an in-transit message when the computed checkpoint gradient value (using receiver's checkpoint as reference) is negative.*

Axiom 2 *Similarly, a message is detected to be a potential orphan message when the computed checkpoint gradient value (using receiver's checkpoint as reference) is positive.*

3.2.2 Checkpoint Dependency Graph

The checkpoint dependency graph is a generic model, i.e., it is common to all the strategies. The basic idea is borrowed from [33] and is modified to accommodate for use in the skeletons. For any given checkpoint and/or logging protocol, its execution results in a protocol-specific dependency graph. Formally, it is represented as follows:

$$CDG = <C, D>$$

This captures dependency relations, where C is a set of checkpoints and D is a set of dependency edges. Symbolically, the k_{th} checkpoint taken at a member P_i is denoted as C_{ik}. The execution behavior of a member P_i is partitioned into fragments associated with each checkpoint. The behavioral fragment corresponding to the checkpoint C_{ik} is denoted

by B_{ik}. B_{ik} contains execution and events that occur after C_{ik} but before (i) the creation of the next checkpoint or (ii) the occurrence of termination or crash.

Here, each checkpoint C_{ik} is represented as three-tuples to capture the information required for recovery. $C_{ik} = <S_{ik}, L_{ik}, N_{ik}>$; where S_{ik} is the local process state of checkpoint C_{ik}, L_{ik} is the log set in B_{ik}, and N_{ik} is the sequence of determinants associated with B_{ik}. Local process state S_{ik} corresponds to the local snapshot that is saved locally in each member. Log set L_{ik} corresponds to the messages logged in a particular behavioral fragment in each member during the fault-tolerance strategy execution.

Definition 6 *A checkpoint dependency graph is said to be proper if it satisfies the following criteria (i) all messages that are exchanged between two behavior fragments are either logged, (i.e., available in a log set) or their dependency recorded (i.e., available in a dependency edge set) and (ii) all the determinants of non-deterministic events and the program order dependencies are recorded [33].*

3.2.3 Gradient-based Checkpoint Protocol

The gradient-based checkpoint protocol is a modified version of a group checkpoint protocol [33] using the checkpoint gradient. This protocol is targeted for the master-slave problem category that exhibits message localization within the subgroup. However, the subgroup as such does not communicate (exchange messages) with the other subgroups. Unlike the original protocol, here the subgroups are assumed to be known from the pre-knowledge of the pattern.

Definition 7 *The gradient-based protocol is correct if (a) the protocol generates a proper checkpoint dependency graph and (b) the protocol generates consistent global snapshots, i.e., all the local checkpoints that form the global checkpoint snapshot are consistent.*

Assumption. The protocol initiator starts the checkpoint operation by taking a self-checkpoint. The checkpoint operation always happens before a message delivery, except in the case of the protocol initiator. The checkpoint operation and the message delivery are assumed to be executed atomically, i.e., entirely or not at all. There exists one kernel per group or one kernel per member. The kernel replays the in-transit message when required; otherwise, it discards it. The subgroups are defined during design time.

Protocol. The protocol executed per communication subgroup is as follows.

Action for member p_i before sending a message m to member p_j:
Piggyback group checkpoint control message (checkpoint number) along with application message m.

Action for an initiator on starting checkpoint protocol at periodic interval:
cur_ckpt = cur_ckpt + 1
Take a new checkpoint

Action for member p_i before delivery of received message m from member p_j:
If message m is piggybacked with a larger checkpoint number (positive gradient) then
    cur_ckpt = piggybacked new checkpoint number
    Take a new checkpoint
    Record checkpoint dependency with member p_j with respect to new checkpoint number
Else if message m is an in-transit message (negative gradient) then
    Log message m

Actions for member p_i during invocation of receive () call (from member p_j):
if member p_i is recovered and there exists replay-message related to member p_j then
    Replay message m from log
Else Execute receive () invocation

Figure 4: Gradient-based checkpoint protocol

Each communication subgroup executes the gradient-based checkpoint protocol. One member in each subgroup is configured as the protocol initiator. The initiator is provided with a logical clock or timer to trigger the protocol initiation. The checkpoint interval is configured during the application development for each subgroup.

On protocol initiation, the initiator does a self checkpoint to save its own state. Then, it piggy-backs all the out-going messages with the control information related to the current checkpoint number.

On message receive, the potential orphan and the in-transit messages are identified. A message originated from the previous checkpoint snapshot (i.e., control information with decreasing checkpoint number) is considered as an in-transit message. Whereas, a message with an increasing checkpoint number, when compared to that of the receiver, is considered as a potential orphan message. An identified in-transit message is logged during the failure-free execution and replayed during the recovery execution. On the other hand, an identified potential orphan message triggers to take a new checkpoint locally.

As a part of protocol, each member should send the checkpoint dependency information to the kernel in order to formulate the checkpoint dependency graph. Checkpoint dependency graph helps in identifying a latest consistent cut for a subgroup belonging to a failed member. This identified consistent cut is used for recovery in case of failure.

When a member fails during execution, the kernel identifies all its dependent members along with the latest consistent cut from the checkpoint dependency graph. Then, the

failed member is recovered and all its dependent members are roll-backed to the consistent cut. During the recovery execution, all the recorded in-transit messages are replayed before receiving any new messages.

Axiom 3 *According to the protocol's logging policy, all messages whose gradient are computed as negative are logged along with the checkpoint. Thus, all the in-transit messages are logged during the failure-free execution.*

Axiom 4 *According to the protocol's checkpoint policy, all messages whose gradient are computed as positive lead to the creation of a new checkpoint snapshot before the actual delivery of the message. Thus, all the potential orphan messages are prevented from becoming orphan.*

Lemma 1 *The gradient-based protocol is proper.*

Proof. This follows immediately from the checkpoint and logging policy of the protocol. The protocol either logs (in case of in-transit messages) or records the message dependency with respect to the associated checkpoints. Thus, the CDG is always proper; hence, the claim holds.

Lemma 2 *The gradient-based protocol is consistent.*

Proof. As the protocol is gradient based, it inherits the capability to detect the in-transit messages and the potential orphan messages implicitly based on Axiom 1 and Axiom 2. According to the protocol, the checkpoint gradient assists in identifying the inconsistent messages. All the positive-gradient messages lead to the creation of a new checkpoint as

per checkpoint policy (Axiom 4) and all the negative-gradient messages lead to logging (Axiom 3). Thus, all the potential inconsistent messages are handled during the failure-free execution. This guarantees the consistency of the local checkpoint created by the protocol. A global checkpoint assembled from all the consistent local-checkpoints is therefore consistent and our claim holds.

Theorem 1 *The gradient-based protocol is correct.*

Proof. Follows directly from Lemma 1 and 2.

3.2.4 Extended Protocol: Color-based Checkpoint Protocol

The color-based checkpoint protocol is similar to the group checkpoint protocol in [33]. This protocol is designed based on the checkpoint gradient discussed earlier along with minor modification to handle the inter-group message as well. It is targeted for the master-slave problem category, which exhibits message localization within the subgroup and exchanges occasional intergroup messages. Each group is assigned a distinct color. A member's color is a two-tuple attribute based on group-color and shade (a.k.a. checkpoint number). All messages are tagged with color-shade tuple, as in the sender. The color helps to identify the message locality, whereas the shade helps to handle the potential inconsistent messages (as in the previous protocol). Unlike the original protocol, here the subgroups are assumed to be known from the pre-knowledge of the pattern.

CID Properties

(a) *Consistency.* The extended protocol is consistent if the protocol generates a consistent global snapshot, i.e., all the local checkpoints that lead to form the global checkpoint snapshot are consistent.

(b) *Independence.* The extended protocol leads to independent recovery of a subgroup if the protocol liberates the inter-group dependency in order to constrain the recovery spread within the subgroup.

(c) *Diligence.* The extended protocol is diligent if uniformly the protocol does not log the messages (i.e., non in-transit messages) exchanged within the same subgroup.

Definition 8 *The extended protocol is correct if (i) the protocol generates a proper checkpoint dependency graph and (ii) the protocol satisfies CID properties.*

Assumption. All assumption from the gradient-based protocol applies here. In addition, we assume that each group will be assigned a distinct color.

Protocol. The protocol executed per communication subgroup is as follows.

Action for an initiator on starting checkpoint protocol at periodic interval:
cur_ckpt = cur_ckpt + 1
Take a new checkpoint.

Action for member p_i before sending a message m to member p_j:
Piggyback group checkpoint control message (containing checkpoint number and color) along with application message m.

Action for member p_i after sending a message m to member p_j:
If p_j is not a member of subgroup then
  Increment send-event count with respect to p_j
End If

Actions for member pi on receive invocation from application to receive message from member pj:
If member p_i is recovered and message exist for replay related to member p_j

```
   Replay recorded message from log related to pj.
Else
   Execute receive action to receive message m from member pj.
End If

Action for member pi before delivering received message m from member pj to application:
If color tag from control information matches with local group color then
   If message m piggybacked with greater checkpoint number (positive gradient) then
     cur_ckpt = piggybacked new checkpoint number
     Take a new checkpoint.
     Record checkpoint dependency with member pj with respect to new checkpoint
   Else if message m is an in-transit message (negative gradient) then
      Log message m.
   End If
Else if message m is inter-group message then
   Log message m.
End If

Actions for member pi on send invocation to send a message to member pj:
If member pi is recovered and pj is not group member then
   Ignore send action.
Else
   Execute send action to send message m to member pj.
End If
```

Figure 5: Color-based checkpoint protocol

Each communication subgroup executes the color-based checkpoint protocol. Each subgroup is colored distinctly. As in the previous case, one member in each subgroup is configured as the protocol initiator. The initiator is provided with a logical clock or timer to trigger the protocol initiation. Also, the checkpoint interval is configured distinctly for each subgroup.

On protocol initiation, the initiator makes a self checkpoint to save its state. Then, it piggy-backs the control information in all the outgoing messages. The control information includes subgroup color and checkpoint number.

On message receive, the inter-group and the in-transit message are identified and logged. The in-transit message is identified by comparing the color first, then comparing the

tagged checkpoint number in the control information with that of the receiver as in the previous protocol (negative checkpoint gradient). The inter-group message is identified by matching the color tag of the control information with the receiver's group color. The color mismatch indicates that the received message is an inter-group message. Similar to the previous protocol, the potential orphan message is identified by computing the checkpoint gradient. The positive gradient leads to creating a new checkpoint in order to prevent the message from becoming orphan.

Similar to the previous case, the kernel manages the checkpoint dependency graph. Each member should send the checkpoint dependency information to the kernel after every checkpoint execution. The formulated checkpoint dependency graph helps in proper recovery in case of failure.

When a member fails during execution, the kernel identifies all its dependent members from the checkpoint dependency graph. Then, the failed member is recovered and all its dependent members are rolled back to the latest consistent cut identified by the kernel from the graph. During the recovery execution, all the recorded in-transit and inter-group messages are replayed. Whereas, all the inter-group message sends are ignored until its recovered state is in sync with the members of the other subgroups.

Axiom 5 *According to the protocol's logging policy, all messages whose color-tags are different from that of the receiver's group color are logged along with the checkpoint. Thus, all the inter-group messages are logged during the failure-free execution.*

Axiom 6 *According to the protocol's logging policy, all messages whose color-tags are the same as that of the receiver's group color and whose gradient is computed as negative are logged along with the checkpoint. Thus, all the in-transit messages are logged during the failure-free execution.*

Axiom 7 *According to the protocol's checkpoint policy, all messages whose color-tags are the same as that of the receiver's group color and whose gradient is computed as positive lead to the creation of a new checkpoint snapshot before the actual delivery of the message. Thus, all the potential orphan messages are prevented from becoming orphan.*

Lemma 3 *The extended protocol is proper.*

Proof. As in Theorem 1, this proof is based on the checkpoint and logging policy of the protocol. As per the checkpoint and logging policy of the extended protocol, all the received messages are either logged (in case of in-transit or inter-group messages) or their dependency recorded with respect to the associated checkpoint. Thus, the checkpoint dependency graph generated by the protocol is always proper and hence, our claim holds.

Lemma 4 *The extended protocol satisfies CID properties.*

Proof. The protocol uses color-tag and checkpoint number as control information for its execution. The checkpoint gradient assists in identifying the inconsistent messages. All the positive-gradient messages lead to the creation of a new checkpoint, as per the checkpoint policy (Axiom 7) and all the negative-gradient messages lead to logging

(Axiom 6). Similarly, the color-tag assists in identifying the inter-group messages and as a result such messages are logged (Axiom 5). Thus, all the inconsistent messages are handled by the protocol. Therefore, all the local checkpoints created are consistent and hence, a global checkpoint formed from the local checkpoints is also consistent. This satisfies condition (a): consistency property. All the messages with different color-tags are considered to be the inter-group messages; as a result, they are logged (Axiom 5). Hence, it frees the dependency of a subgroup with the outside world. This satisfies condition (b): independence property. The color-tag and gradient helps in identifying the intra-group messages (i.e., non in-transit messages). Uniformly, such messages are not logged but with careful attention their dependency with associated checkpoints is recorded to assist in proper recovery. This thereby satisfies condition (c): diligence property.

Theorem 2 *The extended protocol is correct.*

Proof. Follows directly from Lemma 3 and 4.

In the above discussions, we have classified patterns and sub-patterns based on the fault-tolerance strategy. Subsequently, we discussed two protocols related to the master-slave pattern. From the above discussions, we observe that different patterns require different fault-tolerance strategies. The same can be inferred from the evaluation of the different protocols presented in a later chapter. The protocols discussed here are referred in later chapters during the discussions on the framework usages and evaluations.

Chapter 4

Introduction to FT-PAS

In this chapter, we introduce our Fault-Tolerant Parallel Architectural Skeleton (FT-PAS) model. The FT-PAS Model is based on the Parallel Architectural Skeleton (PAS) model [8]. We start in the next section with a discussion on the PAS background. This section presents a brief overview related to the PAS model. Section 4.2 introduces two user groups and their roles in the FT-PAS model. Finally, Section 4.3 introduces the FT-PAS model and discusses its various aspects.

4.1 PAS Overview

In this section, we briefly discuss the Parallel Architectural Skeleton (PAS) model [8]. This system envisions the architectural/structural aspects of the pattern as skeletons. A *skeleton* in PAS is composed of structural/architectural attributes of patterns in parallel computing. Each skeleton in PAS is parameterized based on the pattern-specific structural attributes identified during the skeleton design. As an example, a task-parallel/dynamic replication skeleton, provided by PAS, encapsulates structural aspects of the task-parallel pattern. The task-parallel skeleton includes communication-synchronization primitives. Some of the parameters of the skeleton are: number of workers and the worker itself. These parameters are bound during the application development phase.

A PAS skeleton with unbound parameters is called an *abstract skeleton.* An abstract skeleton becomes a *concrete skeleton* when the parameters of the skeletons are bound to actual values during the application development phase. A concrete skeleton is yet to be filled in with the application-specific code. A concrete skeleton with the filled-in application-specific code results in a *code-complete module* or simply a *module.*

For any given pattern, its corresponding abstract skeleton A_s in the PAS model is composed of the following set of attributes:

(i) The *representative* of an abstract skeleton A_s is empty initially. When concretized by filling with the application-specific code, it represents the module in its action and interaction with other modules.

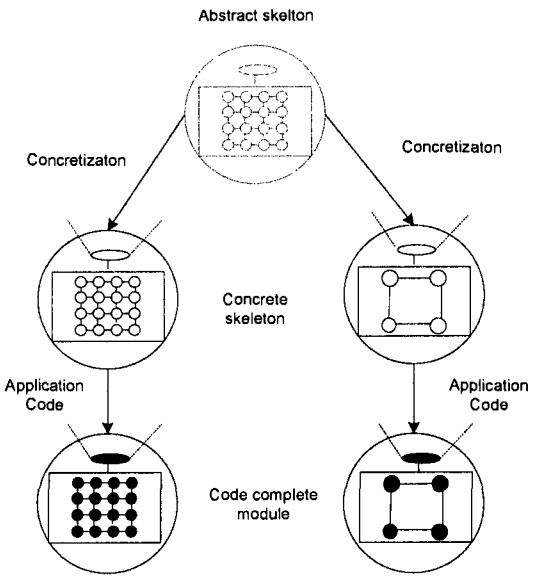


Figure 6: PAS skeleton

(ii) The *back-end* of an abstract skeleton A_s consists of a set of abstract skeletons represented formally as $\{A_{s1}, A_{s2}, \ldots, A_{sn}\}$. Each abstract skeleton in the back-

end of A_s is determined during concretization of A_s. The skeletons contained inside other skeletons result in a (tree-structured) hierarchy. Consequently, each back-end skeleton A_{si} becomes the child of the container skeleton A_s. The children of an abstract skeleton A_s are peers of one another.

(iii) *The topology* provides logical connectivity between the parent-children and among the peers inside the back-end.

(iv) *The internal primitives* are the pattern-specific communication/synchronization primitives. Interactions internal to a skeleton involving the representative and the child modules are performed using these primitives. The internal primitives are the inherent properties of a skeleton. They capture the partial behavior in terms of the communications involved and the topology of the associated pattern.

(v) The *external primitives* of a skeleton are a sub-set of primitives that is used for interactions with its parent and peers.

In addition to the aforementioned skeleton parameters, there exist some pattern-specific parameters. For example, if a chosen pattern is Pipeline, then the number of stages is one parameter and the connectivity of stages is another parameter. An abstract skeleton A_s becomes Concrete skeleton C_s upon configuring these parameters with values. A concrete skeleton C_s leads to a code-complete module when (i) the representative of A_s is filled with the application specific code, and (ii) each child of the back-end is code-complete.

Examples of some of the communication primitive available from the task farm skeleton include SendToMaster(...), ReceiveFromMaster(...), ScatterToWorker(...),

GatherFromWorker(...), etc. Interested readers can find the detailed description of the PAS model with examples in [8].

From the above discussion, it is clear that the PAS follows a hierarchical approach for the application development. An application with code-complete modules inherits all of the above discussed attributes from the abstract skeleton. The code-complete module with no parents represents the root of the hierarchy. The singleton module in parallel application forms the leaf of the hierarchy. All other intermediate modules represent partial parallel applications.

4.2 User Categories and their Roles

FT-PAS categorizes the users of the model into two: a *protocol developer* and an *application developer*. From the PAS overview, we understand that the PAS too has two sets of users whose responsibilities are targeted towards the application related aspects. Whereas, here the user roles of FT-PAS are formulated towards addressing fault-tolerance.

The protocol developer is responsible for designing and implementing new fault-tolerant strategies using the basic building blocks provided from the model. The protocol developer is expected to have a better understanding of the fault-tolerance issues than the counterpart (application developer) discussed subsequently. In addition, the protocol developer is expected to have a good understanding of the FT-PAS model in order to integrate a fault-tolerance solution into an existing skeleton in the model.

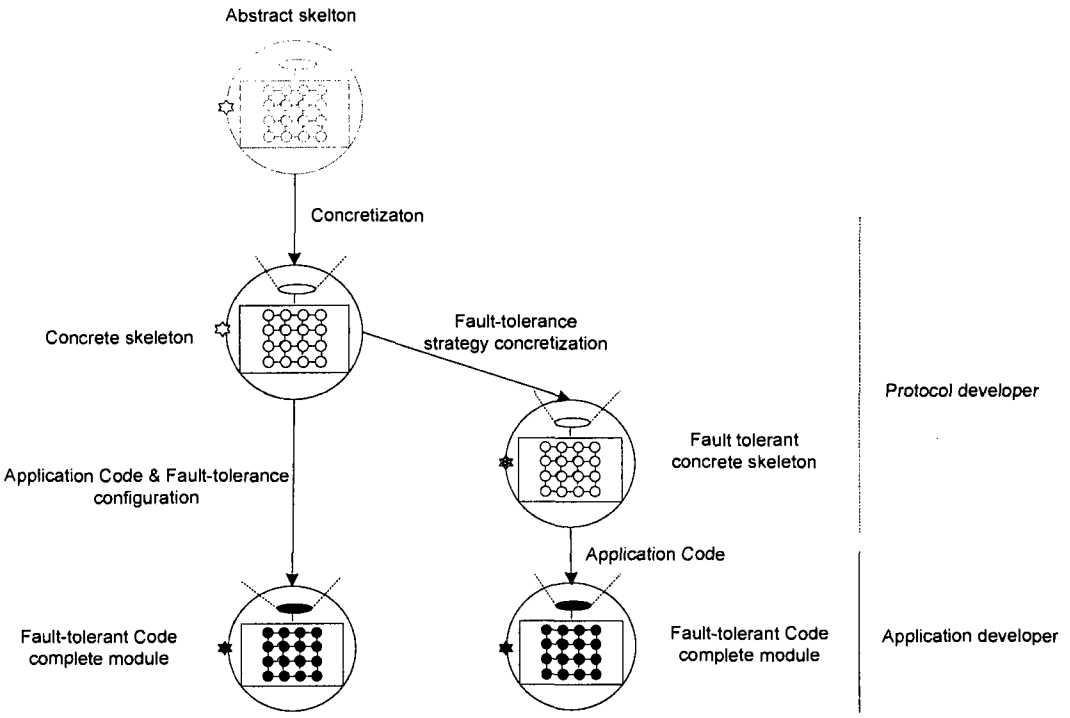


Figure 7: FT-PAS skeleton and its phases

During the application development phase, the application developer chooses a required fault-tolerance strategy for a skeleton based on the given application characteristics. The selected fault-tolerant strategy is configured with the skeleton to support fault-tolerance of the application. The application developer is expected to have some understanding of the fault-tolerance issues but only from the usage perspective rather than from the implementation perspective. Figure 7 illustrates the user roles against the various phases of the skeleton during the application development in the model.

4.3 Introduction to the FT-PAS Model

In this section, we introduce the FT-PAS model by illustrating its concepts in an informal manner. Our idea is generic and can be implemented in any pattern-based parallel

programming model. For the purpose of discussion and to demonstrate our idea, we use the PAS model as a base.

4.3.1 Overview

In practice, patterns in parallel-programming are targeted for application developers in the application-related aspects. In comparison, our research focuses on assisting the application developer in systems-specific aspects, e.g., fault-tolerance. This research emphasizes the following two issues: firstly, different fault-tolerant techniques are well suited for different patterns in parallel programming. Secondly, patterns-specific fault-tolerance strategies can be implemented and pre-packaged in a generic fashion, i.e., independent of a specific application.

We present a new approach to provide fault-tolerance for parallel application using patterns. We aim to achieve three important aspects from the fault-tolerance perspective: (1) Specificity, (2) Separation-of-Concern and (3) Protocol Extension. All of these aspects are discussed in the following sections.

The FT-PAS model is based on the PAS model. It supplements a new layer on top of the PAS to support pattern-specific fault tolerance. Another objective of the FT-PAS model is to provide necessary building blocks to build new fault-tolerance strategies as needed for available skeletons. As a result, it extends the users of the PAS model with new responsibilities related to delivering fault-tolerance.

4.3.2 Specificity

Each pattern is targeted for a different problem type. Similarly, this inherent nature to address varying problem categories can help in addressing the system-side issues too (i.e. fault-tolerance). An abstract skeleton A_s in the PAS is defined as {Rep, BE, Topo, P_{Int}, P_{Ext}}, each of which was elaborated on previously. In the FT-PAS model, we amend this definition by including a new parameter S related to the fault-tolerance. This new parameter related to the fault-tolerance strategy is to be designed, implemented and pre-packaged along with the skeleton. So our new refined definition for the abstract skeleton with the appended fault-tolerance parameter in the FT-PAS model is as follows:

$$A_s = \{Rep, BE, Topo, P_{Int}, P_{Ext}, S\}$$

Here, Rep stands for the representative, BE stands for the back-end, Topo stands for the topology, P_{Int} stands for the internal primitives, P_{Ext} stands for the external primitives, and the new parameter S stands for the fault-tolerance strategy.

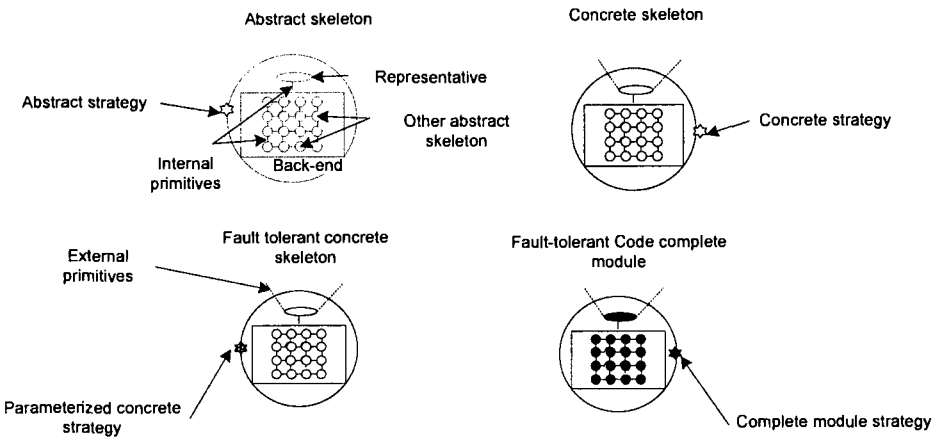


Figure 8: FT-PAS skeleton and its components

The fault-tolerance strategy S is defined as pattern-specific like other primitives of the skeleton. They are built using the basic building blocks and/or implementing the required interfaces available from the model based on the strategy requirement. The various internal components of the skeleton are shown in Figure 8.

4.3.3 Separation of Concern

In practice, the patterns are realized as a skeleton in parallel-programming by providing an abstraction with higher-level programming primitives and by hiding the lower-level issues like communication and synchronization. The PAS is one such model that separates the lower-level parallel programming issues from the application developer to ease the application development.

In addition, in FT-PAS, we address issues concerning fault-tolerance in a pattern-specific manner. We separate the fault-tolerance implementation concern from the application developer. This is by delivering the fault-tolerance implementation pre-packaged in an application independent manner. During the application development phase, an application developer can choose a pattern along with the suitable fault-tolerance protocol which fits a given application.

Thus, the model can aid the developer in choosing an appropriate skeleton (pattern–implementation) using the catalog of skeletons. This approach provides the necessary separation of concern to the application developer as far as fault-tolerance is concerned.

4.3.4 Protocol Extension

Protocol extension is a key requirement for any pattern-based approach. Protocol extension in our context refers to the fault-tolerance protocol extension, provisions for allowing newer fault-tolerance protocols to be integrated to an available pattern, whenever need arises. The FT-PAS model provides various core facilities to the protocol developer in order to design a new strategy. Formally, the core facilities of the model are represented as follows:

$$F_{core} = \{P_b, M, F_r, C_s, L_s, T_s\}$$

Here, F_{core} refers to the core facilities provided by the model. P_b refers to the protocol behavior abstraction, M refers to the marshaller abstraction, F_r refers to the fault reactor abstraction, C_s refers to the checkpoint service, L_s refers to the logging service and T_s refers to the timing service. All these core facilities are explained in detail in the next chapter following the discussion on the framework internals. These core facilities are used in the fault-tolerance protocol extension. In theory, protocol extension is represented as follows:

$$P_x = f_{px} : F_{core} \rightarrow F_{custom}$$

Here, P_x refers to an extended protocol which is to be implemented. F_{core} refers to the core facilities provided by the model; F_{custom} refers to a set of components concretized from the core facilities; and f_{px} refers to the concretization function or action that the protocol developer specializes and/or overrides in order to deliver the extended protocol-specific functionalities.

It is the responsibility of a protocol developer to extend the FT-PAS model with newer fault-tolerance protocols for an existing skeleton. In this context, the model can also serve as a test-bed for evaluating newly designed fault-tolerance protocols. Protocol extension, its primitives and usages, are demonstrated in detail using examples in the next chapter.

4.3.5 Generic Group Definition

From the PAS discussion, we know that a topology is defined as part of a given skeleton. The base level primitives of the core facilities (discussed in the previous section) do not have view of the topology, which is defined at higher level. A 1-D virtual processor array representation is used internally in the FT-PAS model for referring to a node. For a given topology, the individual nodes are mapped on to the 1-D virtual processor array. This is similar to the virtual processor grid mapping in the extensible PAS [41]. However, we use the 1-D virtual processor array for our convenience. In theory, it is represented as follows:

$$\mathcal{M}: \mathcal{T} \rightarrow \mathcal{A}$$

Where, $\mathcal{M}$ is a mapping function that maps nodes from the topology space $\mathcal{T}$ to the 1-D virtual processor array $\mathcal{A}$. There are skeletons which require protocols to be executed in groups like the group checkpoint protocol. The model should provide a generic way to define groups in order for our base primitive to understand them irrespective of the topology definition. This is achieved by using the abstract group mapping function. In theory, a group mapping function is represented as follows:

$$\mathcal{GM}: \mathcal{T} \rightarrow \mathcal{G}$$

Here, $\mathcal{GM}$ is an abstract group mapping function, which is concretized during the concretization phase. This mapping function groups the nodes from the abstract topology space $\mathcal{T}$ to the abstract group space $\mathcal{G}$. The constituents of the group space refer to the members in the 1-D virtual processor array. This enables base primitives to understand the group definition for executing the configured protocol.

```
/* group mapping function*/
void map_group(const Location &loc, SubGroupSet &grpSet)
{
    grpSet[Loc[ROW]].addMember(loc); // members of same row are mapped to a same group
}

/* function to retrieve group id for a given member */
GROUP_ID my_group_id(const Location &loc, SubGroupSet &grpSet)
{
    return grpSet[loc[ROW]].getGroupId();
}
```

Figure 9: Example of generic group mapping

For example, consider a data parallel skeleton in which all members of an identical row should form a group. The group mapping function for such scenario is shown in Figure 9.

In the above discussions, we have introduced the FT-PAS model and various aspects that are addressed in the model. The design and implementation of the FT-PAS model are subsequently presented in the next chapter.

Chapter 5

FT-PAS Design, Usage and Case Study

In this chapter, we discuss the design and implementation of the FT-PAS model. We call the FT-PAS model implementation: the FT-PAS framework. Fault-tolerance protocol extension is an important contribution of this research. The framework provides basic building blocks in order to implement a new fault-tolerance protocol for available skeletons. We also discuss the protocol extension design along with its primitives and usages. In Section 5.1, the architecture of the FT-PAS framework is presented. In Section 5.2, the high level design of the FT-PAS framework is discussed, which includes discussion on the framework internals. In Section 5.3, we discuss protocol extension, its primitives, and framework usage from a two user perspective (skeleton/protocol developer and application developer) along with case studies.

5.1 Framework Architecture

In general, a framework is an abstraction to realize specialized functionalities by using reusable components. The abstract components/interfaces provided with the framework are subsequently concretized. A framework consists of the following:

1. A generic backbone, which aids in design, development and application execution in the framework defined flow of control.

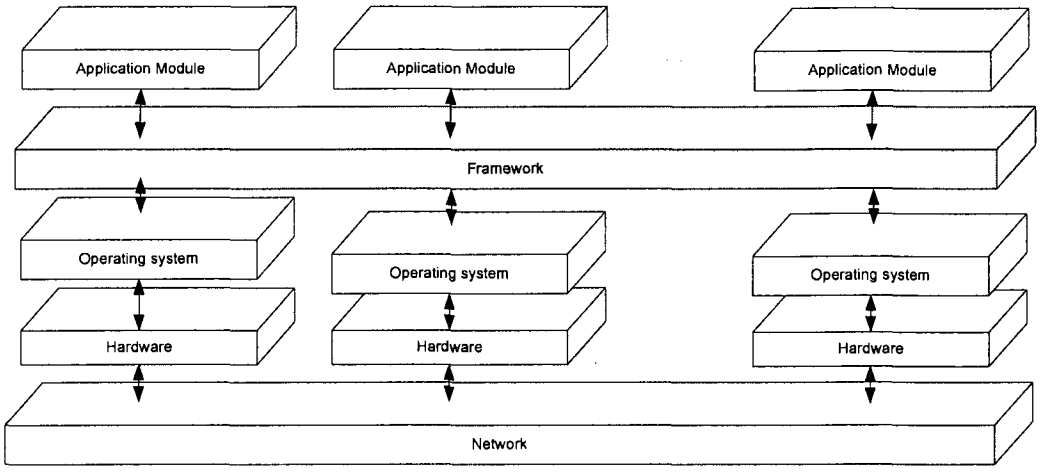


Figure 10: General view of the framework

2. A set of concrete components, which are reusable and commonly used in building an application.
3. A set of abstract components or interfaces, which are to be specialized or overridden to deliver application-specific functionalities.

By implementing the interfaces and embedding the application functionalities, a concrete application can be generated.

5.1.1 The FT-PAS Architecture

Figure 11 shows the architecture with various constituents and support layers of the FT-PAS (Fault-Tolerant PAS) framework. We have demonstrated our idea using the PAS model. As discussed previously, the PAS model generically (i.e., independent of specific patterns and applications) defines the architecture of the patterns. The skeleton-specific communication and synchronization primitives form the part of the PAS model. These

skeleton-specific communication-primitives are defined using the support from the layer underneath, the message passing library.

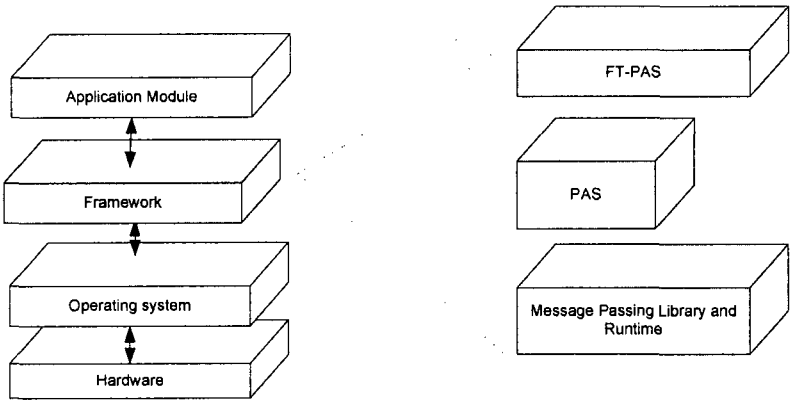


Figure 11: Architecture of the FT-PAS framework

The FT-PAS augments PAS with the patterns-specific fault-tolerance support. The application programmer adds the application-specific behaviors (i.e. code segments specifying control- and data-flow) by choosing the appropriate skeleton(s).

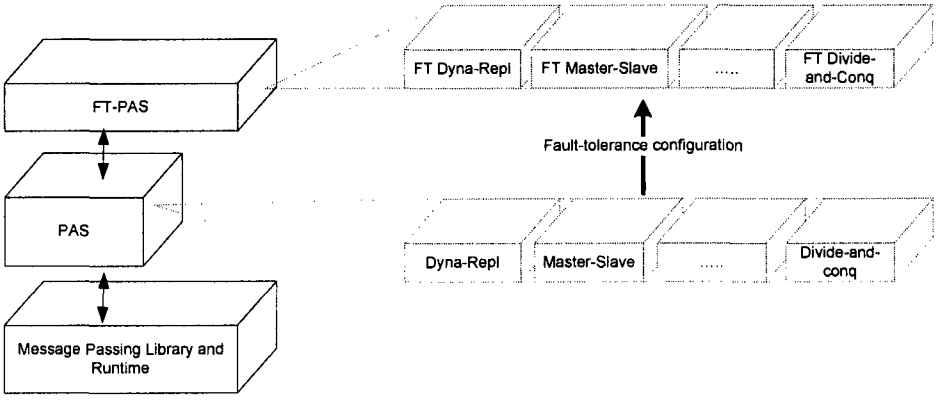


Figure 12: High-level view of the framework

In addition, the application developer who wants to incorporate fault-tolerance support in an application should choose the fault-tolerance strategy and supply the application-

specific fault-tolerance strategy parameter(s). Figure 13 gives the conceptual view of the PAS skeleton embedded with fault-tolerance support.

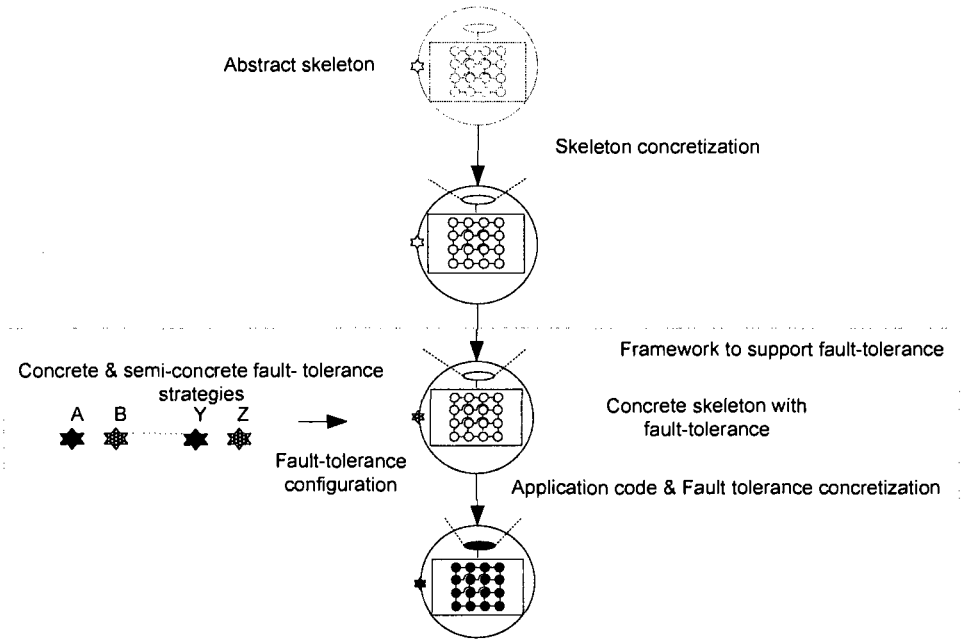


Figure 13: Conceptual view of the fault-tolerant parallel architectural skeleton

Depending on the fault-tolerance strategy at hand, the provisioning of fault-tolerance to an application might be developer transparent or semi-transparent, i.e., requiring some application-specific information.

5.1.2 Framework Assumption

Distributed and parallel systems experience different types of failures, such as crash failure, omission failure and byzantine failure, which are discussed in [22, 40]. Our framework is aimed at handling crash failure (process crash) and assumes fail-stop [22, 40] of software faults, along with the following assumptions: (1) The hardware and operating system can survive, (2) The underlying network service can survive, (3) The

communication link is reliable, and (4) The framework itself is fault-tolerant and hence, can survive.

5.2 Design of the Framework Internals

In this section, we discuss the design of the FT-PAS framework internals. We have classified the constituents the framework into two categories: concrete modules and abstract interfaces. The abstract interfaces are those which need to be specialized in order to implement the fault-tolerance strategy-specific behaviors. The concrete modules are generic functionalities that are commonly used in most of the fault-tolerant strategies. As a result, these modules are provided as part of the framework internals. Figure 14 provides the design view on the modules of the framework internals.

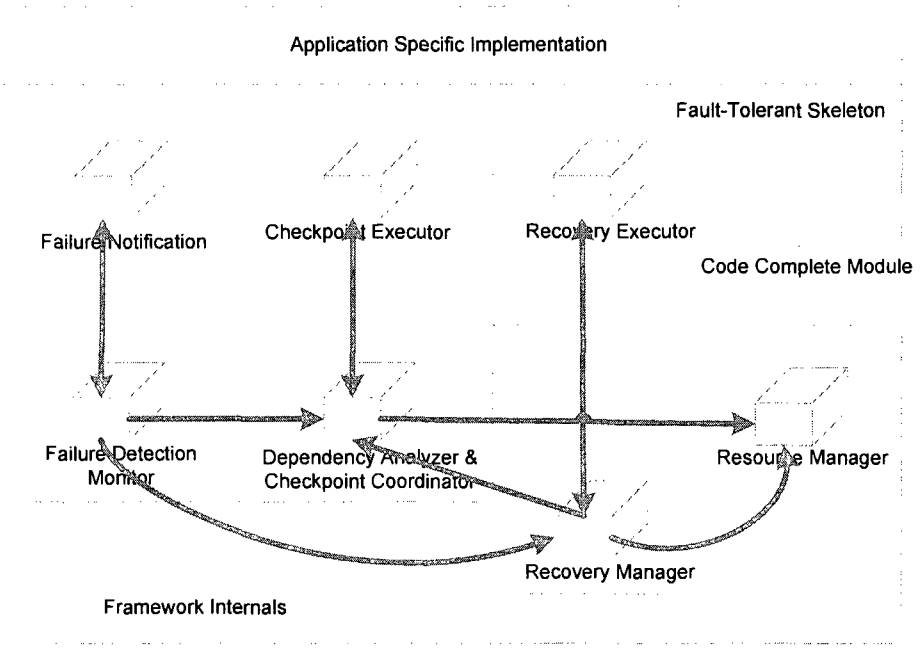


Figure 14: Framework internals

5.2.1 Messaging Mechanism

a) Message Passing Library

The FT-PAS is built on top of sockets due to limitations with MPI for implementing fault-tolerance (discussed previously in Section 2.2). An MPI like communication library is built using sockets. The library provides the minimal necessary functionalities of MPI but with added functionalities related to the fault-tolerance support.

The basic lower-level primitives are used to form the higher-level primitives of the FT-PAS framework. More details on the added functionalities are discussed in the next section.

b) Message Passing Library Extension

From our discussion in Chapter 2, it is clear that provision to support fault-tolerance is required from the underlying system-software. As in FT-MPI [1], the FT-PAS model extends the message passing library with modified semantics in order to provide support for fault-tolerance. The extension to the message passing library is described in the following section.

The communicator in our message passing library is incorporated with modified semantics to handle failure recovery. There are various semantics possible. These semantics guide different recovery strategies at a higher-level of abstraction. Our current implementation supports the following semantics: build, recover, and repair. The build mode is responsible for establishing a communication link among all processes during

normal execution. The repair mode is responsible for reestablishing the communication link to the recovered member in the healthy dependent members. Similarly, the rebuild mode is responsible for reestablishing the communication link to the healthy dependent member in the recovering failed member.

All the above discussed modes and communication primitives are lower-level details. They are used for the internal working of the model and hence, are not visible at a higher layer.

5.2.2 Failure Detection Module

There exist various failure models that are discussed in [22, 40] to address different kinds of faults. In our model, we focus only on the fail-stop fault model, i.e. process failure. The communication link is assumed to be reliable. From the framework internals perspective, the failure detection module consists of two components: Failure Detection Monitor and Failure Notification Service.

a) Failure Detection Monitor

Each process, on successful registration, is monitored by the failure detection monitor. The information (hostname and processed id) required for monitoring is registered to the kernel as part of the framework initialization during the application startup. As mentioned earlier, the failure detection monitor detects only process failure.

The failure detection monitor can detect process failure based on various approaches. In general, they can be classified into two. The first approach is based on the operating

system support [38]. It scans for process availability using simple shell-command (i.e., ps). This, when combined with remote-shell (RSH) or secure-shell (SSH), provides a simple but elegant mechanism to scan for availability of a process on a remote machine. The second approach is based on the pulse or heart-beat technique. This technique is used in most of the parallel and distributed systems for failure detection. The approach is based on two models: the push model (heart-beat) and pull model (are-you-alive) as addressed in [37]. In our framework, we currently support the first approach to scan for faults. The latter approach (heart-beat or pulse technique) can as well be implemented in our model without any impact on the application as it is purely internal to the system and totally transparent to the protocol developer.

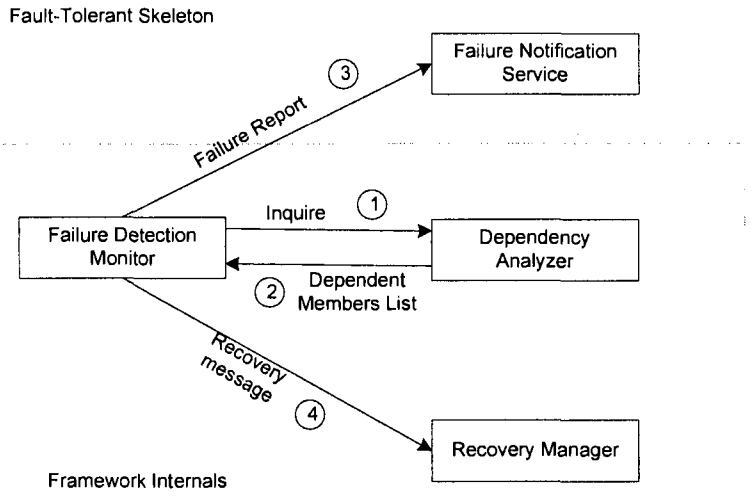


Figure 15: Failure detection monitor - post detection procedure

Upon receiving a failure report, the failure detection monitor checks for the reported failure before proceeding to the post-detection procedure. As shown in Figure 15, upon failure detection, it finds all the dependent members and forwards the failure report to

their failure notification service. Lastly, it passes a message to the recovery manager in order to trigger a recovery operation.

b) Failure Notification Service

The failure notification service receives a failure report from the failure detection monitor. On receiving the failure report, it marks the corresponding dependent as failed in order to avoid the error being cascaded further. In addition, it triggers post failure actions which are protocol-specific behaviors. It also sends a failure report to the failure detection monitor when there is failure during communication with a dependent member.

5.2.3 Checkpoint Module

The checkpoint module consists of three components: dependency analyzer, checkpoint executor and checkpoint coordinator. The dependency analyzer is responsible for maintaining a checkpoint dependency graph (CDG) [33] which is used during recovery. The checkpoint executor is responsible for the following: checkpoint initiation and checkpoint operation. The checkpoint coordinator is responsible for leading the coordination action among the dependent members as a result of checkpoint initiation.

a) Checkpoint Dependency Analyzer

The checkpoint dependency analyzer maintains the checkpoint dependency graph (CDG) (discussed previously in Section 3.2.2). The CDG is maintained per skeleton and it helps to track recovery dependencies among the members of the skeleton. It is common to all the strategies and is used during the recovery of processes. For any given checkpoint

and/or logging protocol, its execution results in a protocol-specific checkpoint dependency graph. This graph captures the dependency relations and is constructed from the dependency information received from each checkpoint executor.

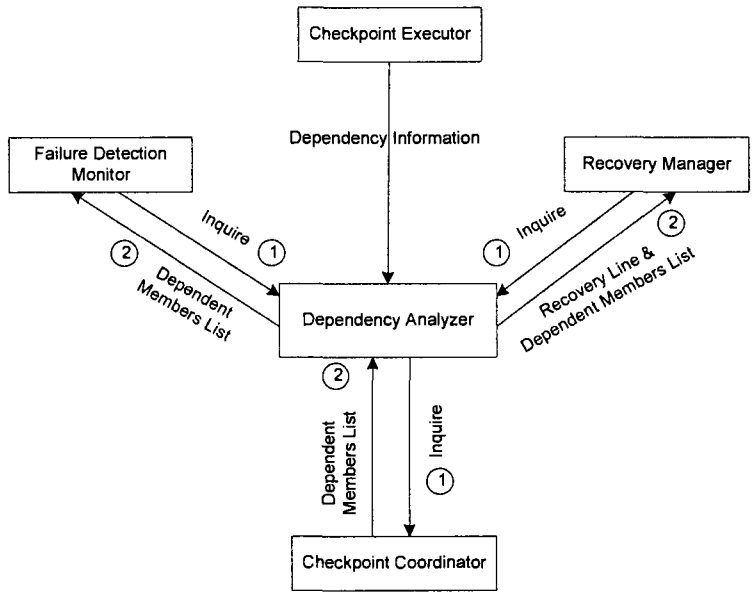


Figure 16: Interaction between dependency analyzer and other components

In addition, the checkpoint dependency analyzer plays the role of serving the dependency information related to the CDG it maintains. It provides dependency information as needed upon request to various components during the protocol execution. The failure detection monitor and checkpoint coordinator are two other components which contact the dependency analyzer to identify the dependent members. The recovery manager is another component which inquires the dependency analyzer to identify the recovery line a.k.a. consistent cut (discussed previously in Section 2.1.2).

b) Checkpoint Executor

As mentioned earlier, the checkpoint executor is responsible for executing the checkpoint operation. As a result of the checkpoint operation, it stores the checkpoint file in a location as directed by the resource manager. It also sends the corresponding dependency information during the checkpoint execution to the dependency analyzer in order to keep the CDG up-to-date.

c) Checkpoint Coordinator

There are fault-tolerance strategies for some patterns that require explicit blocked coordination with their peers. The checkpoint coordinator is used for such cases. It does coordination on behalf of the protocol initiator by executing a three-phase coordination protocol [39]. The three-phase checkpoint protocol is as follows: In the first phase, the coordinator receives a checkpoint initiation request from an initiator. Consequently, it forwards the synchronization request to all peers of the initiator. In the second phase, each member acknowledges back with the ready-message. In the final phase, the coordinator sends the commit message to all peers to checkpoint the system state.

In the second phase of the protocol, a ready-message is sent when a member is ready to participate in the coordination protocol. Otherwise, a busy-message is sent when it is busy waiting to communicate with its dependent. For example, assume that two members are supposed to communicate as per the execution order. The coordinator sends a synchronization request to these two members. The first member happens to receive the coordination signal late because it is currently blocked (waiting to receive a message from its counterpart), while the other member receives the coordination signal on time before the communication-send invocation. Here, the first member is blocked for the

communication to complete, whereas the second member has already sent a ready-message back to participate in the protocol execution. In such a scenario, the kernel releases the second member in order for the blocked member to complete its communication. As a result, the blocked-member along with its counter-part progress forward in order to empty their communication buffer. This mechanism is similar to the bookmark exchange mechanism in [35]. Finally, the remaining members participate voluntarily in the protocol execution (when their communications are complete) by acknowledging back with the ready-messages. Subsequently, the coordinator initiates the third phase of the protocol in order to save a consistent global state.

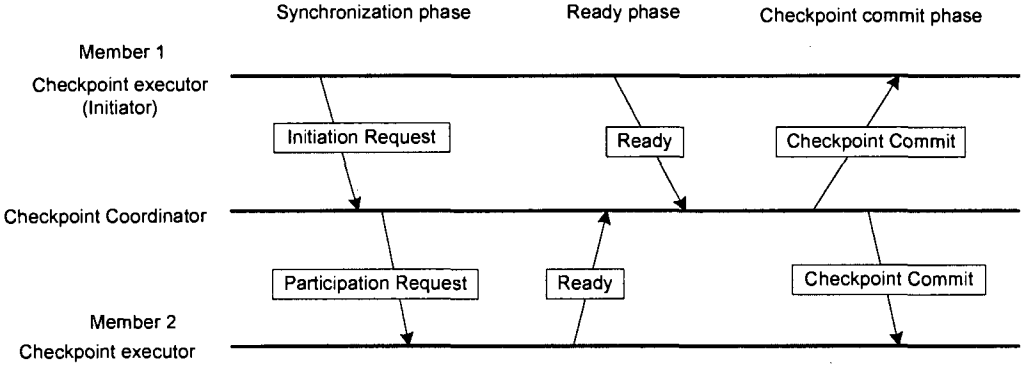


Figure 17: Three phase consistent checkpoint coordination

5.2.4 Resource Management

The kernel manages various resources which include processes and their information, control message queue, and checkpoint/log file. Upon the start of an application, the kernel spawns a required number of processes remotely in the preconfigured list of hosts. Each member registers with the kernel by providing its process information as part of the setup procedure for purposes of failure detection and recovery. Thus, the kernel maintains

the process information for each member of a skeleton. We refer to this registered information as member information. It consists of process id, hostname, state, and identifier. In addition, the kernel registers one listener thread per remotely spawned process in order to handle control messages from the application. All received control messages in each listener are placed in a common synchronized queue for processing.

The kernel organizes the checkpoint and log files related to the application in a directory structure. This is established using a shared Network File System (NFS), making it visible to all nodes. All information regarding where to store and fetch the checkpoint/log file are directed by the kernel to the individual processes. This is done as part of the initialization procedure at the application start.

This basic file management feature can very well be extended to support sophisticated functionalities, e.g., checkpoint/restart fault-tolerance for OpenMPI [36], where remote file management is achieved. In which, the runtime system temporarily saves the snapshot locally. Then, it moves them to a stable storage as post-checkpoint aggregate operations. During the recovery execution, it preloads the checkpoint file from the stable storage to a node in which restoration is targeted.

5.2.5 Recovery Module

Upon receiving intimation from the failure detection monitor, the recovery manager triggers the recovery operation. The recovery operation involves collaboration with various components of the kernel. Recovery is executed in three stages. In the first stage, the recovery manager inquires the dependency analyzer to identify a recovery line from

the CDG. As a result, the dependency analyzer computes the recovery line with respect to the failed member. Both the recovery line and the dependent members list are sent back to the recovery manager.

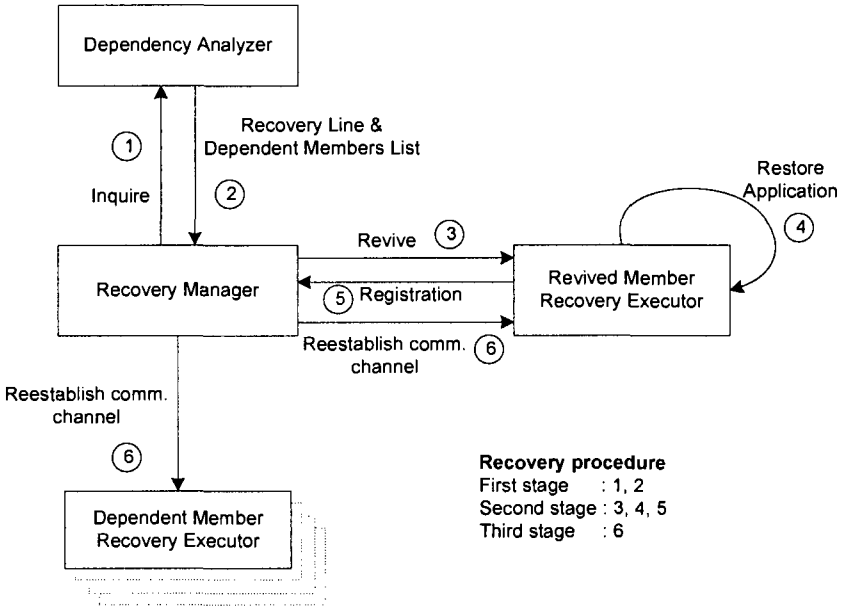


Figure 18: Recovery module and its action

In the second stage, the recovery manager restores the failed member and its recovery executor retrieves all resources required for the application restoration. It recovers the application state to an earlier saved system state. Note that the recovery executor is a protocol specific module which is implemented by the protocol developer to incorporate specific fault-tolerance behaviors. Finally, when the application is recovered, the recovery executor registers with the kernel before resuming the application execution.

In the third stage, the recovery manager broadcasts the re-establishment-message to all the members engaged in the recovery procedure. Thus, the communication links between revived and healthy members are restored. Figure 18 illustrates these actions.

The recovery procedure can be as simple as creating a replacement member, ready to take over a failed process to compute a new job (e.g., restart-recovery strategy for the dynamic-replication skeleton). Whereas, a sophisticated mechanism might involve recovering a failed member to resume its execution from an intermediate checkpointed location (e.g. gradient-based checkpoint protocol).

5.3 Protocol Extension: Primitives, Usages and Case Studies

In this section, we discuss various core facilities that are used in building new fault-tolerance protocols. The core facilities include abstract interfaces and semi-concrete components which facilitate the design of the protocol-specific behaviors. Subsequently, we discuss the primitives that are available from these core facilities. In addition, we show the framework usages from the perspective of a protocol developer and an application developer. We demonstrate how these primitives are used for the protocol extension to design new strategies via difficult case studies.

5.3.1 Overview of the Protocol Extension

Protocol extension refers to provisions for integrating new fault-tolerance protocols to an available pattern, whenever need arises. In this section, we discuss the core facilities and its interfaces which are used in the protocol development as part of the protocol extension.

The framework provides the following key facilities to a protocol developer in order to integrate new strategies. Most of these core facilities implement part of the kernel functionalities and define interfaces required for the protocol-specific behavior

implementation. This enables capabilities to integrate the protocol-specific behaviors with the framework internals. The core facilities are as follows:

(i) The *failure reactor* implements failure notification service as part of kernel functionalities and defines *abstract post failure action* which is to be implemented specifically for a protocol. As part of the failure notification service, it automatically triggers post failure actions which are protocol-specific.

(ii) *Checkpoint and logging services* implement the checkpoint executor as part of the kernel functionalities. They provide built-in checkpoint and logging facilities which create checkpoint, generate logs, save checkpoint and log data in a stable storage (either locally or remotely on NFS) as directed by the resource manager. The default action(s) can be overwritten by the protocol developer, e.g., what information to save, for instance, in an application-level check-pointing or where to save the logged information for the purpose of performance tuning.

(iii) The *recovery handler* implements the recovery executor as part of the kernel functionalities. It provides a default abstract recovery implementation. The protocol developer can define the protocol-specific recovery initialization and post-recovery procedures. It collaborates with the kernel in recovering both the application and the communication links among the members of the skeleton.

(iv) The *marshaller* provides an interface for implementation of the protocol-specific data marshalling and un-marshalling capabilities. The framework provides a default marshaller that facilitates the marshalling and un-marshalling of the application's

contiguous data (without pointers). It can be extended to support complex non-contiguous data as well. The default marshaller can also be extended by the protocol developer for marshalling/un-marshalling of the fault-tolerance protocol-specific control information.

(v) The *fault-tolerance protocol behavior module* provides an interface to the protocol developer in order to incorporate fault-tolerance protocol specific behavior which is executed as part of the protocol. For example, in case of the gradient-based checkpoint protocol, its behavior includes starting the checkpoint action by the protocol initiator. Consequently, the checkpoint action is executed in other group members after receiving a control message with checkpoint flag set, etc. Similarly, the handling of the in-transit messages (i.e., by logging/replaying during the failure-free/recovery execution) is all part of the protocol specific behaviors.

5.3.2 Primitives for the Protocol Extension

In this section, we discuss the primitives available from the various core facilities introduced in the previous section.

a) Checkpoint service

The checkpoint service implements the checkpoint facility with two flavors: system-level check-pointing and application-level check-pointing. The type to use for a given protocol is defined as part of the strategy definition by the protocol developer.

In case of system-level checkpoint, the checkpoint service uses a customized checkpoint library to save the checkpoint state. This customized version is derived from a well

known checkpoint library [10] from the PSNC research group. The derived version has been tailored specifically to the needs of our framework. In case of application level checkpoint, it uses a simple mechanism to save the application developer's identified state on to the stable storage. Both implementations are wrapped around a common interface, though the underlying technique/mechanism involved in these two variants is different. Below is the list of primitives available from the checkpoint service to a protocol developer:

(i) Primitive for Setup Action: This primitive is used to setup the checkpoint service in a skeleton. It is invoked as part of strategy initialization in the skeleton.

(ii) Primitive for Checkpoint Action: This primitive is used to initiate the checkpoint action. It is invoked during the failure-free execution to save the system state that is used during the recovery.

(iii) Primitive for Recovery Action: This primitive is used to initiate the recovery action. It is invoked as part of the recovery execution and recovers the system state to an earlier saved-execution state stored during the failure-free execution.

(iv) *Primitive for Call Back Registration Action:* Each strategy is unique and requires a way to define strategy-specific action as part of a protocol, i.e., post checkpoint action and post recovery action. The checkpoint service provides a *callback mechanism* to achieve this. This primitive is used by the protocol developer to register a callback method. It is registered either as post checkpoint action type or post recovery action type. The checkpoint service invokes the callback method automatically as post-failure actions based on callback type (defined during the service registration).

b) Logging service

The logging service provides necessary basic interfaces required for the protocol development related to logging. The logging service is used for different purposes based on the fault-tolerance strategy. It is used for logging messages and for recording send events. In case of gradient-based checkpoint protocol, it is used to log the in-transit messages. Whereas, in the color-based checkpoint protocol, it is used to log messages of two types - in-transit messages and inter-group messages. In the traditional logging protocol, it is used to log all the messages exchanged among the members. The purpose of the logging service varies based on the fault-tolerance strategy. Below is the list of primitives available from the logging service to a protocol developer:

(i) Primitive for Setup Action: This primitive is used to setup the logging service in a skeleton. It is executed as part of the fault-tolerance strategy initialization in the skeleton. It is invoked during both the normal execution and the recovery execution.

(ii) Primitive for Cleanup Action: This primitive is used to clean up the log generated during the fault-tolerance strategy execution.

(iii) Primitive for Message Record Action: This primitive is used for logging a message. It is invoked as part of the fault-tolerance strategy execution.

(iv) Primitive for Message Replay Action: This primitive is used to replay an earlier recorded message during the application recovery. It is a part of the recovery execution strategy.

(v) Primitive for Replay-Message Existence Check: This primitive is used to check for existence of a replay message. This decision is used by the protocol in order to decide for a message replay action.

(vi) Primitive for Send Redundancy Check: This primitive is used to check whether a send-action is to be ignored or not. It returns true when the send-action is a redundant action. This decision is used by the protocol to ignore the send-action replay related to a non-dependent member which is not part of the recovery group.

c) Marshaller service

The marshaller service facilitates the protocol developers to incorporate packing/unpacking actions to embed the protocol-specific control information. This service provides a default packing and unpacking implementation for the application messages. It can be extended by a protocol developer to incorporate strategy-specific actions such as piggy-backing the control information with the application messages. Below are the data packing primitives available from the marshaller service:

(i) Primitive for Data Packing: This primitive is used to marshal the input data into a specified target buffer.

(ii) Primitive for Data Unpacking: This primitive is used to un-marshal the encoded data into a specified target buffer.

Below are the marshaller interfaces which need to be concretized in order to specify protocol-specific actions:

(i) Interface for Marshal Action: This interface should be concretized to implement the strategy-specific marshalling actions. It is invoked on a send-action as part of the pre-processing operation.

(ii) Interface for UnMarshal Action: This interface should be concretized to implement the strategy-specific un-marshalling actions. It is invoked on a receive-action as part of the post-processing operation. A default marshaller implementation to pack/unpack the application message is shown below.

```
class DefaultMarshaller : public Marshaller
{    void marshalAction(...) /* Default marshalling procedure */
     {   DataPacking::pack(...); /*Pack application data */  }
     void unmarshalAction(...) /* Default un-marshalling action */
     {    int offset = DataPacking::unpack(...); /*Unpack application data */  }
}
```

d) Failure reactor service

The implementation of the framework provides the default failure reactor as shown below.

```
class DefaultFailureReactor: public FailureNotificationService{
void action()
{
    /* Re-establishes communication link with recovered member */
   ftCommService.ReconnectComm(....);
   ...
}}
```

Note that there can be differences in these actions, depending on the protocol. For example, in a color-based checkpoint protocol where the groups can exchange occasional messages among them, the re-establishment of the communication links can be delayed as some sends are discarded (redundant sends). Similarly, if receives need to replay logged messages then re-establishment of the communication links can be delayed. Hence, the framework facilitates by providing interfaces to both define and register

strategy-specific failure reactors for different communication methods. The *Failure Reactor Registration Primitive* is used to register a failure reactor module to the framework. The registered failure reactors are invoked by the failure notification service. They are executed as part of the pre-processing operations during recovery.

e) Fault-tolerance behavior

This service provides interface methods which are to be implemented to define protocol specific behaviors. The list of interface methods available to a protocol developer are as follows: *(i) Pre-Send Fault-Tolerance Behavior, (ii) Post-Send Fault-Tolerance Behavior, (iii) Pre-Delivery Fault-Tolerance Behavior, (iv) Post-Delivery Fault-Tolerance Behavior, (v) Send Fault-Tolerance Behavior* and *(vi) Receive Fault-Tolerance Behavior.*

All the above interface methods should be concretized by the protocol developer to define pre- and post-actions related to a fault-tolerance strategy. Their uses in few of the strategies are shown as part of case studies in the following sections.

f) Integration

The protocol developer has to integrate all the protocol services used in the fault-tolerance strategy design. A service registration method (*registerFTServices*) is defined in the protocol class to facilitate integration. The protocol developer has to implement this method. In this method, the protocol developer defines necessary service instances and invokes their corresponding service registration method to configure services. The framework provides one primitive for registration action for each service. These

primitives are listed as follows: *(i) Primitive for Marshaller Registration (ii) Primitive for Timing Service Registration, (iii) Primitive for Checkpoint Service Registration, (iv) Primitive for Logging Service Registration, (v) Primitive for Failure Reactor Registration* and *(vi) Primitive for Fault-Tolerance Behavior Registration.*

The *registerFTServices* method is invoked as part of the fault-tolerance strategy initialization. In addition, the protocol developer needs to implement protocol-specific cleanup action if required. Those fault-tolerance service-specific cleanups are invoked in the *cleanupFTServices* method of the protocol class. In turn, this will be invoked as part of the protocol cleanup action.

5.3.3 Framework Usages and Case Studies

In this section, we illustrate the framework usage by implementing two pattern-specific fault-tolerance protocols using the above discussed primitives.

5.3.3.1 Case Study 1: Gradient-based Checkpoint Protocol

In this section, we illustrate how to implement a variation of a group checkpoint protocol for the master-slave skeleton using the core facilities provided by the framework. The protocol implemented here is the 'gradient-based checkpoint protocol' that is discussed in Section 3.2. The protocol assumes that the subgroups are independent, i.e., there are no (occasional) interactions across subgroups. The coordination among subgroup members are achieved via piggybacking of application messages with control information. Thus, there are no explicit protocol messages.

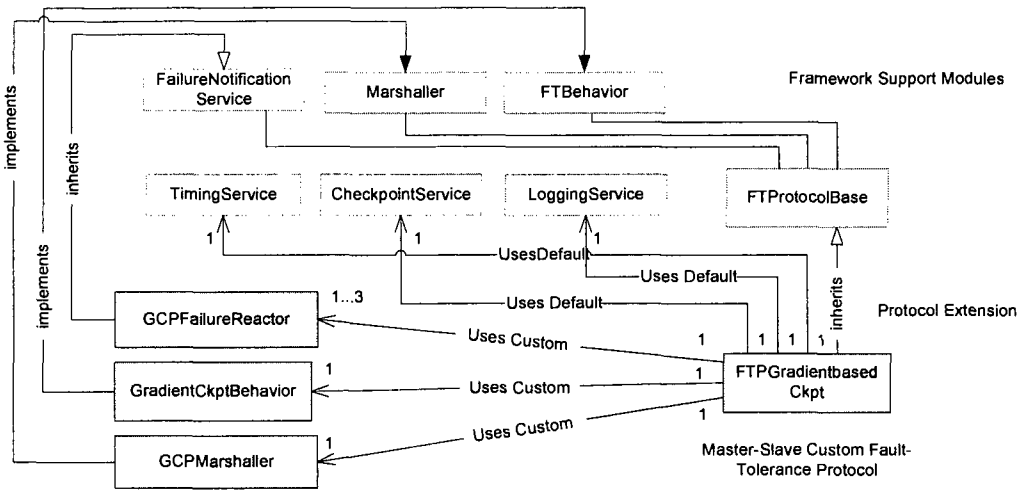


Figure 19: High-level class diagram of gradient-based checkpoint protocol extension

Figure 19 provides a high-level class hierarchy diagram illustrating the framework modules involved and their extensions by a protocol developer, which are elaborated in the following:

a) Usages of the framework from a protocol developer's perspective

The following discussion illustrates a protocol developer's involvement in implementing the gradient-based checkpoint protocol using the core functionalities of the framework. Each fault-tolerance protocol has a protocol-specific behavior class which is extended by the protocol developer from the default behavior class, *FTBehavior*. In this specific example, we name this extended class as *GradientCkptBehavior*. This extended behavior class implements all the protocol-specific actions as shown in the pseudo code of the gradient-based checkpoint protocol in Figure 20.

```
Protocol specific fault tolerant behavior for pre-send action:
  If I am the Initiator and the checkpoint flag is enabled
    Increment the checkpoint number.
    Take a new checkpoint.
  End If

Protocol specific fault tolerant behavior for pre message-delivery action:
  If the computed checkpoint gradient is positive (i.e., piggybacked checkpoint number is large)
    Update receiver's checkpoint number with the piggybacked checkpoint number.
    Take a new checkpoint.
    Record checkpoint dependency.
  Else if the computed checkpoint gradient is negative (i.e., in-transit message)
    Record the received message.
  End If

Protocol specific fault tolerant behavior for post receive-invocation action:
  If the member is currently recovered and there exist messages available for replay then
    Replay-message from log.
    Returns a flag to indicate the existence of the replay-message.
  Else
    Returns a flag to indicate the non-existence of the replay-message.
  End If
```

Figure 20: Gradient-based checkpoint protocol - protocol behavior

In this protocol, each member process logs the in-transit messages. During recovery, the revived member needs to replay these messages and hence, re-establishment of the communication links could be delayed. The module inherits the failure notifications service and implements post-recovery actions as shown in Figure 21. Such actions are also defined for other communication methods, e.g. send, probe, etc.

```
Protocol specific failure reactor action for receive communication method:
  If replay-messages do not exist then
    Re-establish the communication link.
  End If
```

Figure 21: Gradient-based checkpoint protocol - failure reactor

Similarly, as mentioned earlier, the framework provides default data marshalling and un-marshalling facilities. It can be extended by a protocol developer to incorporate protocol-

specific marshalling action by implementing the *Marshaller* interface. In this particular example of the gradient checkpoint protocol, it is required to piggyback protocol-specific control information (i.e. checkpoint number). The pseudo-code of the protocol-specific *GCPMarshaller* action implementations are shown in Figure 22.

```
Protocol specific marshal action:
(i) Pack the application message to the target buffer using the data packing utility.
(ii) Pack the control message to the target buffer using the data packing utility.

Protocol specific unmarshal action:
(i) Unpack the application message from the input buffer using the data packing utility.
(ii) Unpack the control message from the input buffer using the data packing utility.
```

Figure 22: Gradient-based checkpoint protocol - marshaller

Now we implement the protocol class, *FTPGradientbasedCkpt*. This protocol class inherits from the *FTProtocolBase* class of the framework. The protocol developer overrides two initialization methods and provides protocol-specific initialization actions as shown in Figure 23.

In the overridden *startupInitialize* method, the default base protocol's initialization method is first invoked. This establishes the communication link from the member to the framework kernel. Next, the setup method for the checkpoint service is invoked to initialize the service. This is followed by setting up a checkpoint interval using the in-built per-process timer service through invocation of the *setTimer* method. Lastly, the logging service setup routine is invoked to initialize the logger in order to record and replay the in-transit messages.

```
class FTPGradientbasedCkpt : public FTProtocolBase
{
   private:
   struct ControlInfo
   {  int ckpt_number;
      .... /* other support method definitions */
   };
   public:
   /* Extension with protocol specific startup initialization */
   void startupInitialize()
   { FTProtocolBase::startupInitialize();/* Framework provided initialization */
     getCheckpointService().setup(....);/* Checkpoint service setup */
     getTimingService().setTimer(....);/* Timer initialization */
     getLoggingService().setup(....);/* Logger initialization */
   }
   /* Extension with protocol specific recovery initialization */
   void recoveryInitialize()
   { FTProtocolBase::recoveryInitialize(); /* Framework provided initialization*/
     getCheckpointService().setupAction(....);/* Checkpoint service setup */
     getCheckpointService().recoveryAction();/* Checkpoint recovery step */
   }

   /* Extension with protocol specific post checkpoint recovery */
   void postRecoveryCallBack()
   { FTProtocolBase::registerInfoToKernel(); /* Register info. with kernel*/
     FTCommService ftComm;
     ftComm.reset(); /* Reset comm. channel state */
     getLoggingService().setup1(....); /* Setup logging service to replay messages*/
   }
   ....
}
```

Figure 23: Gradient-based checkpoint protocol class

Similar initializations are done during the recovery execution. They are amended in the *recoveryInitialize* method. This protocol uses the checkpoint service as part of protocol behavior. Hence, the protocol should implement post-recovery callback methods (Figure 23) in order to execute the necessary protocol-specific post-recovery actions. Finally, all the protocol-specific service implementations are integrated using the *registerFTServices* method as shown in Figure 24.

```
void FTPGradientbasedCkpt::registerFTServices()
{
   /* Failure reactor registration */
   registerFailureReactor(RECV_FAILURE_REACTOR, new GCPRecvFailureReactor());
   .... /* Similar registration of other reactor */

   registerMarshaller(new GCPMarshaller()); /*Marshaller registration */
   registerTimingService(new Clock()); /* Timing service registration */

   registerLoggingService(new LoggingService()); /* Logging service registration */
   CheckpointService ckptService = new CheckpointService(SYSTEM_LEVEL);
   registerCheckpointService(ckptService); /* Checkpoint service registration */

   /* Post checkpoint action and post recovery action registration */
   ckptService->registerCallBack(POST_RECOVERY_ACTION,
         CheckpointService::CallBack(this, &FTProtocolInterface::postRecoveryCallBack));
   .... /* Similar registration of postCheckpointCallBack method */

   registerFTBehavior(new GradientCkptBehavior()); /* FT Behavior registration*/
}
```

Figure 24: Gradient-based checkpoint protocol - service registration

In the previous discussion, a protocol developer is assumed to be knowledgeable about the framework's services and its interfaces. Also, the protocol developer is required to be knowledgeable about the system's specific issues, e.g., fault-tolerance protocol design. On the contrary, an application developer is expected to be minimally knowledgeable about the system's specific issues. The following section illustrates an application developer's involvement in embedding the previous fault-tolerance protocol into an application code that uses the master-slave skeleton.

b) Usages from an application developer's perspective

In a blocking checkpoint protocol, where all processes are part of one group, the application developer has virtually no involvement other than choosing a checkpoint interval (if not using the default). In case of the previous protocol, the application developer has to specify the subgroups and the protocol initiator for each subgroup.

```
class Slave : public SingletonSkeleton<CommProt_MS, FTPGradientbasedCkpt>
{ /* The Slave module extends the SingletonSkeleton in PAS. Its communication protocol is
CommProt_MS and FT protocol is FTPGradientCkpt, which is the extended FT protocol class defined in
the previous section */
   void run() { /* application specific code */}
}

class Application : public MasterSlaveSkeleton <Slave, CommProt_MS, VOID, FTPGradientbasedCkpt>
{ /* An application that uses the Master-Slave skeleton */
   void run()  { /* application specific code */}
   void FTConfigure() /* FT protocol specific configuration */
   {  SubGroups subgrpSet; /* Set of communication subgroups */
      subgrpSet.setSize(NUM_OF_SUBGRPS); /* Specify the size */
      /* create a subgroup: groupid, number of members, member enumeration */
      SubGroup grp1(GRP1_ID, GRP1_SIZE, GRP1_MEMBERS);
      grp1.ProtocolInitiator (GRP1_INITIATOR_ID); /* set protocol initiator id */
      grp1.CkptInterval = GRP1_CKPT_INTERVAL; /* set checkpoint interval */
      subgrpSet.addSubGroup (grp1); /* add the above defined subgroup to the set */
      /* ...Other subgroup declaration are omitted... */
    }
}
```

Figure 25: Gradient-based checkpoint protocol - application developer's perspective

Each instantiated FT-PAS module has a run method and an *FTConfigure* method for fault tolerance configuration. This is illustrated in Figure 25. The implementation of the *FTConfigure* method is the only involvement of the application developer from the fault-tolerance perspective (Figure 25). For certain protocols, the configure methods can use the default in-built parameters and functionalities. Hence, the fault tolerance support becomes completely application-developer transparent in such cases.

5.3.3.2 Case Study 2: Application-level checkpointing for Iterative Problems

In this section, we demonstrate how to implement the fault-tolerance protocol for the master-slave skeleton where the slaves are naturally synchronizing, i.e., iterative in nature. This protocol uses the application-level checkpoint to save states.

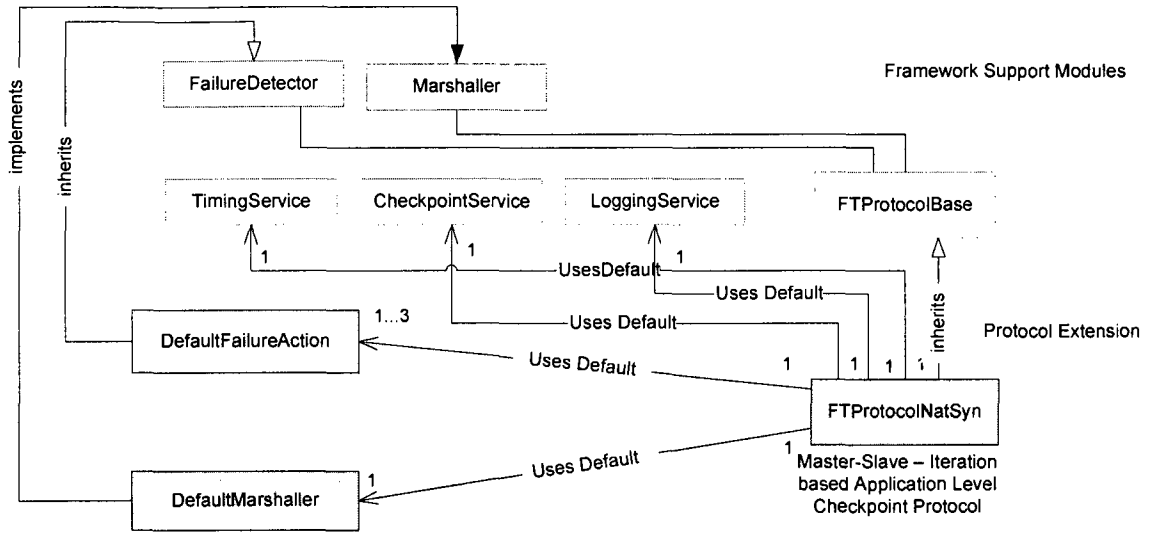


Figure 26: High-level class diagram - fault-tolerance protocol for iterative problem

It is assumed that the application developer identifies all the application state variables. Only these user-identified application states are saved and restored during the checkpoint and recovery execution. Figure 26 illustrates a high-level class diagram of the various classes involved in the protocol implementation.

a) Usage of the framework from a protocol developer's perspective

The following section illustrates a protocol developer's involvement in implementing the fault-tolerance protocol for the naturally synchronizing slaves. In order to support the application-level checkpointing, the framework provides a state class as shown in Figure 27. In addition, the framework provides an abstract iterator as shown in Figure 27. This protocol implements this abstract iterator to plug-in the fault-tolerance (application-level checkpoint) support for the naturally synchronizing slaves.

```
class State
{   ...
    int iterationCount;
    int ckptInterval;
}

template <class TState, class FT>
class AbstractIterator : public FT
{   virtual void Init(TState &myState)=0;
    virtual bool Check(TState &myState, bool *ret)=0;
    virtual void Finalize(TState &myState)=0;
    virtual void Iteration(TState &myState)=0;
    virtual void PostIteration(TState &myState)=0;
    virtual void Start(TState &myState)=0;
}
```

Figure 27: AbstractIterator and State interfaces

The state class is amended with a set of minimal states which need to be saved for the internal working of the iterator. In the next subsection, we show how this specialized iterator is used by the application developer for concretization of the slave (application-specific).

As mentioned earlier, this protocol is targeted for problems which exhibit natural synchrony in their behavioral pattern (i.e., slaves which are iterative in nature). Thus, the protocol requires no explicit coordination action or protocol behavior and does not exchange any protocol-specific control information. Hence, it uses the default marshaller for marshalling the application messages. It uses the default failure reactor for re-establishing the communication link. Timing service is used to trigger the checkpoint at regular iteration interval.

```
void FTProtocolNatuSync::registerFTServices()
{
   /* Failure reactor registration */
   registerFailureReactor(RECV_FAILURE_REACTOR, new DefaultFailureReactor());
   .... /* Similar registration of other reactor */

   registerMarshaller(new DefaultMarshaller()); /*Marshaller registration */
   CheckpointService ckptService = new CheckpointService(APPLICATION_LEVEL);
   registerCheckpointService(ckptService); /* Checkpoint service registration */
}
```

Figure 28: Iteration-based application level checkpoint protocol class

Similar to the protocol discussed earlier, the *FTProtocolNatuSync* protocol class provides protocol initialization actions for both the startup and recovery execution. Subsequently, the protocol class provides implementation to the *registerFTServices* method in order to integrate various services used in the protocol implementation (Figure 28). The checkpoint service instance used here is configured to support the application-level checkpointing.

The *FTIteratorAppLvlCkpt* class implements the abstract iterator by using the fault-tolerance protocol as the FTProtocolNatuSync protocol class. In the iterator implementation, the protocol developer provides implementation only for the *Start* and *PostIteration* methods. This is illustrated in Figure 29.

```
Protocol specific start action:
   If the recovery flag set then
      Recover the application-states.
   Else
      Initialize the application-specific states.
   End If
   Loop until the exit condition defined in the check method is satisfied
      Execute the iterator method.
   End Loop
   Execute the finalize method for the cleanup.
```

```
Protocol specific Post-Iteration action:
   If the iteration count matches the checkpoint iteration interval the
      Takes a new checkpoint.
   End If
```

Figure 29: Iterator with fault-tolerance actions

All other method definitions (i.e., *Init, Check, Iterator, Finalize*) are delegated to the application developer in order to define the application-specific behaviors.

b) Usages from an application developer's perspective

The application developer's involvement in using the above designed protocol is shown in Figure 30.

```
class AppState : public State
{  Work work;
   Result partialResult;
}

class Slave : public SingletonSkeleton<CommProt_MS, FTIteratorAppLvlCkpt<AppState>>
{ /* The Slave module extends the SingletonSkeleton using CommProt_MS as communication protocol
and FTIteratorAppLvlCkpt as FT protocol, which is an iterator implementation embedded with fault-
tolerance protocol */
   AppState myState; /* Application state instance */
   void Init(AppState *state){/* Application specific code */}
   void Iterator(AppState *state){/* Application specific code */}
   void Check(AppState *state){/* Application specific code */}
   void Finalize(AppState *state){/* Application specific code */}
}

class MSApplication : public MasterSlaveSkeleton <Slave, CommProt_MS, VOID,
FTIteratorAppLvlCkpt>
{  /* An application that uses the Master-Slave skeleton */
   void run()  { /* application specific code */}

   void FTConfigure()
   { /* FT protocol specific configuration */
      ...
      SubGroups subgrpSet; /* Set of communication subgroups */
      subgrpSet.setSize(NUM_OF_SUBGRPS); /* Specify the size */
```

```
        /* create a subgroup: groupid, number of members, member enumeration */
        SubGroup grp1(GRP1_ID, GRP1_SIZE, GRP1_MEMBERS);
        grp1.ProtocolInitiator (GRP1_INITIATOR_ID); /* set protocol initiator id */
        grp1.CkptInterval = CKPT_ITERA_INTERVAL; /* set checkpoint iteration interval */
        subgrpSet.addSubGroup (grp1); /* add the above defined subgroup to the set */
        ....
    }
}
```

Figure 30: Fault-tolerant iterative application – application developer's perspective

As with other protocols discussed earlier, the application developer should concretize *FTConfigure* as part of the fault-tolerance configuration. The application developer should define a class inheriting the *state* class like *AppState* (Figure 30). This class should be defined with all the application-specific state variables which need to be saved as part of the application-level checkpointing. In addition, the application developer should concretize the iterator methods inherited in the slave class with the application-specific code. Moreover, the slave class should declare an instance of the *AppState* class and use this instance to hold any application state during processing.

In the above discussions, we have demonstrated the framework design, its primitives and usages through two case studies. The evaluation of the above built pattern-specific fault-tolerance protocols along with others are presented in the next chapter.

Chapter 6

Evaluation

In this chapter, we discuss the evaluation of the framework in terms of its usages and performance. In Section 6.1, we discuss the environment and implementation issues related to the FT-PAS framework. Subsequently, we summarize our experience on the usages of the framework and its related issues. Finally, in Section 6.3, we present the experimental results and discuss the performance overhead of the FT-PAS framework.

6.1 Environment

The current implementation of the FT-PAS framework is in C++. The test environment consists of Sun-Fire-280R workstations. Each workstation has 2 CPUs (UltraSPARC III Cu processors); it operates at 1015 MHz and has 4 GB RAM. All the workstations are running the Solaris 9 operating system (SunOS) and are connected by LAN.

The framework uses a customized version of the PSNC Checkpoint library. The original version of the library was written in C. We ported it to C++ and customized it to the needs of the framework.

The framework is currently implemented on the Solaris platform. The experiments are conducted in a LAN of homogeneous workstations. The development system uses standard tools like GNC C++ library, etc., for compilation and execution. The underlying communication layer uses sockets.

6.2 Experiences on the Framework Usages

To evaluate the usages of the FT-PAS, we implemented a set of protocols for the master-slave skeleton. We designed the gradient-based checkpoint protocol and the color-based checkpoint protocol (discussed previously in Section 3.2). We implemented both the protocols and tested their performance with the above discussed test environment. The experimental results are presented in the next section.

Below are few observations made from the experiments conducted during our evaluation.

- Using the FT-PAS, it is expected that the effort required to develop fault-tolerant parallel applications are minimized.
    - It provides built-in fault-tolerant skeletons, readily usable for application-development with minimal effort.
    - It provides capability to choose fault-tolerance strategies based on the application-characteristics from a list of supported protocols for a given skeleton.
- Moreover, the FT-PAS is expected to reduce the protocol development time for implementing new fault-tolerant strategies from the perspective of a protocol developer.
    - It provides concrete reusable services such as checkpoint service, logging service, fault monitor, etc., in order to reduce strategy implementation time.

- It provides semi-concrete or abstract interfaces to incorporate strategy-specific behavior.

- The FT-PAS objective is to achieve a separation-of-concern, by separating the fault-tolerance implementations from the application-specific details. Thus, it reduces the application developer's burden.
- Using the FT-PAS, the protocol developer can extend the fault-tolerance protocol base supported for a given skeleton by implementing new fault-tolerance strategies.
- Unlike many existing systems, the FT-PAS is aimed at addressing concern related to delivering fault-tolerance support in a pattern-specific manner instead of having one common fault-tolerance strategy for all application types.

6.3 Experimentation and Results

We conducted experiments to measure the performance of the framework using the test environment described in Section 6.1. The results observed from various experiments are discussed in the following section. In general, the objectives of these experiments are to measure the framework overhead incurred due to fault-tolerance. This section is divided into two subsections. Each of these subsections discusses a different set of experiments for different objectives and interprets their results.

In the first subsection, the objective is to measure the framework overhead with and without fault-tolerance. In the second subsection, the objective is to measure and compare the overhead incurred due to different fault-tolerance strategies. In all these experiments,

the FT-PAS framework components are configured to run in a single workstation, while the application modules run on the other workstations.

6.3.1 Framework Overhead

In the first set of experiments, the objective is to measure the overhead due to logging. Table 1 presents the test results observed from the experiment by varying a single parameter, i.e., number of communication events.

Total no. of comm. events	Execution time without fault-tolerance (in sec).	Execution time with logging protocol (in sec).	Overhead on use of logging protocol (in sec).	Overhead on use of logging protocol (in percent).
120	1.9090	2.1441	0.2351	12.32%
240	3.5893	4.0737	0.4844	13.50%
360	5.2327	5.9893	0.7566	14.46%
480	6.8943	8.2407	1.3464	19.53%

Table 1: Overhead incurred with and without the simple logging protocol

In theory, the logging overhead is expected to increase linearly with the increase in the communication events (illustrated using dotted line in Figure 31). However, from the above experiment, we observe an exponential increase (Figure 31). This deviation might be due to the framework overhead (i.e. overhead incurred because of using a central data store for message logging).

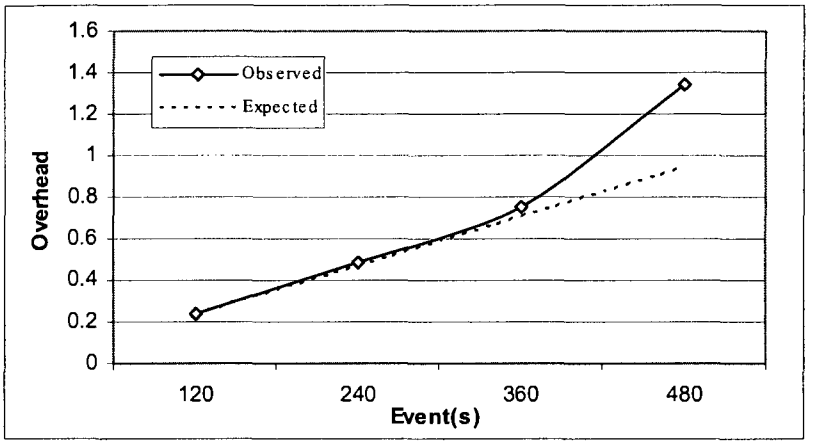


Figure 31: Overhead due to logging

In the second set of experiments, the objective is to measure the overhead due to the checkpointing (system-level). Table 2 illustrates the test results observed by varying the number of checkpoints.

Num. of checkpoint	Execution time without fault-tolerance (in sec).	Execution time with checkpointing (in sec).	Overhead on use of checkpointing (in sec).	Overhead on use of checkpointing (in percent)
1	2.2677	2.3836	0.1159	5.11%
2	7.0509	7.4402	0.3893	5.52%
3	13.7527	14.6365	0.8838	6.43%
4	23.3919	25.1313	1.7394	7.44%

Table 2: Overhead incurred with and without checkpointing

In theory, the checkpoint overhead is expected to increase linearly with increase in the number of checkpoints (illustrated using dotted line in Figure 32). However, from the above experiment, we observe an exponential increase. This deviation might be due to

the framework overhead (i.e. overhead incurred because of using a central data store for checkpoint state saving).

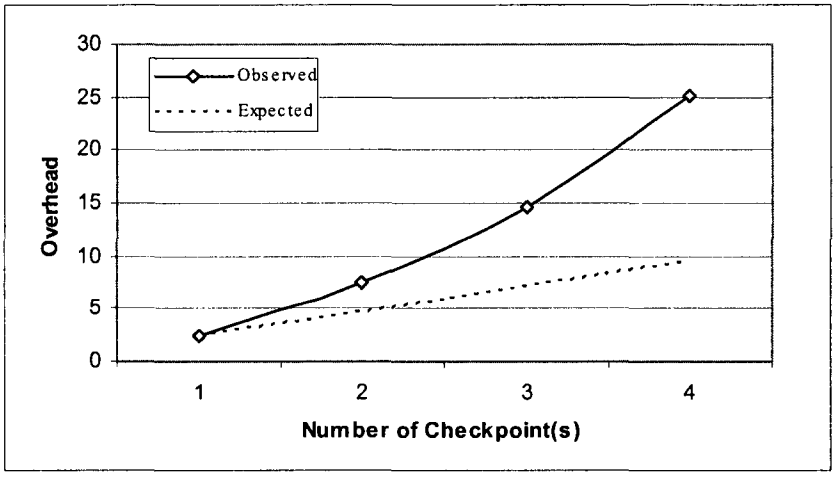


Figure 32: Overhead due to checkpointing

6.3.1 Comparison of the Different Fault-Tolerance Protocols

In this subsection, we discuss two sets of experiments in order to compare two different fault-tolerance protocols. In the first set of experiments, the objective is to compare the overhead incurred using the color-based checkpoint protocol (discussed in Section 3.2) and the blocking checkpoint protocol. First, we illustrate how the overhead changes in varying the message localization density. *Message localization density* is defined as the average message group density divided by the total inter-group messages; whereas, the *message group density* is defined as the number of intra-group messages divided by the group size. We fixed all parameters, such as the number of slaves per group, number of groups and total number of intra-group messages per group but varied the message localization density. The message localization density is varied by varying the number of

inter-group messages exchanged between two groups. Table 3 shows the overhead observed from executing four test cases, each with varying message localization density.

Avg. message localization density	Execution time without FT (in sec).	Execution time with color-based ckpt protocol (in sec).	Overhead on use of color-based ckpt protocol (in sec).	Overhead on use of color-based ckpt protocol (in percent)
0.83	4.1888	5.2375	1.05	25.07%
1.11	3.9789	4.6824	0.70	17.59%
1.67	3.9472	4.4284	0.48	12.16%
3.33	3.8919	4.1638	0.27	6.94%

Table 3: Fault tolerance overhead - varying message localization density

The overhead decreased as the message localization density increased. This is from the fact that the more messages are localized within a group, the less become the inter-group messages. Thus, the overall overhead decreases as the logging overhead incurred from inter-group message decreases, which is observed from the graph (Figure 33).

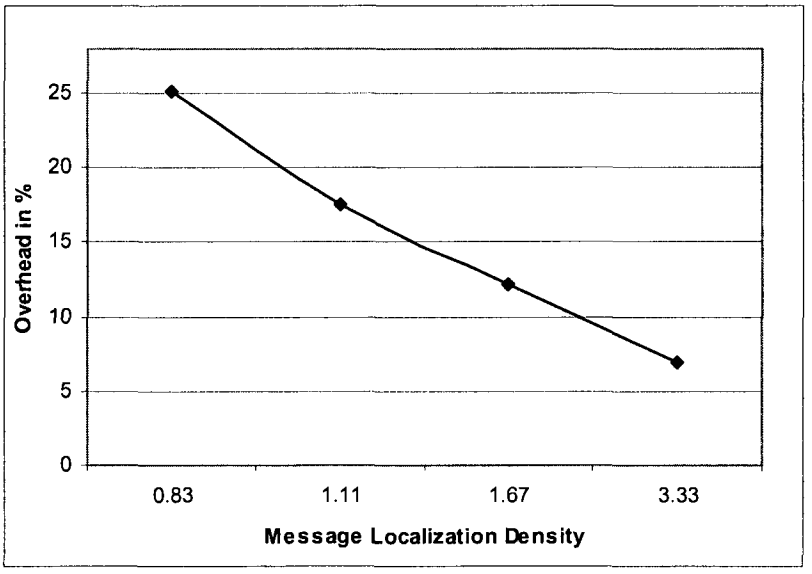


Figure 33: Fault tolerance overhead percent - varying message localization density

Also, we measured the overhead due to the blocked checkpoint protocol; its average overhead is computed as 20.2 sec.

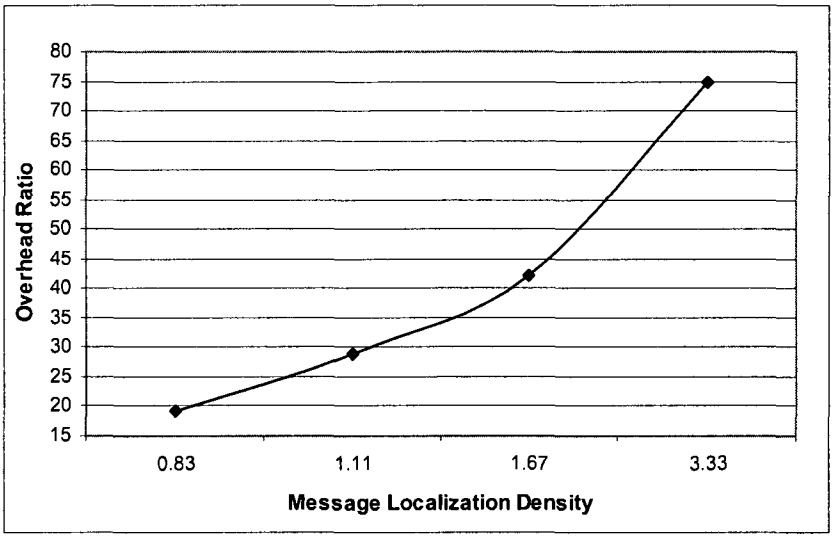


Figure 34: Overhead ratio - varying message localization density

Figure 34 shows the overhead ratio of the blocked checkpoint protocol over the color-based checkpoint protocol by varying the message localization density. The overhead ratio increases as the density increases. Thus we can interpret from the graph that the color-based checkpoints do comparatively better than the blocked checkpoint protocol for applications that have higher message localization density.

Num. of checkpoint	Execution time without fault-tolerance (in sec).	Execution time with appl-level ckpt (in sec).	Overhead on use of appl-level ckpt (in percent).	Execution time with sys-level ckpt (in sec).	Overhead on use of syst-level ckpt (in percent).
1	2.2677	2.3192	2.27%	2.3836	5.11%
2	7.0509	7.3256	3.90%	7.4402	5.52%
3	13.7527	14.4813	5.30%	14.6365	6.43%
4	23.3919	24.6791	5.50%	25.1313	7.44%

Table 4: Overhead comparison – application-level and system-level checkpoint

In the final set of experiments, the objective is to compare the overhead incurred due to the system-level checkpoint protocol with the application-level checkpoint protocol for a problem of iterative type such as Jacobi. The overhead incurred using these two protocols is observed by varying the number of checkpoints (Table 4).

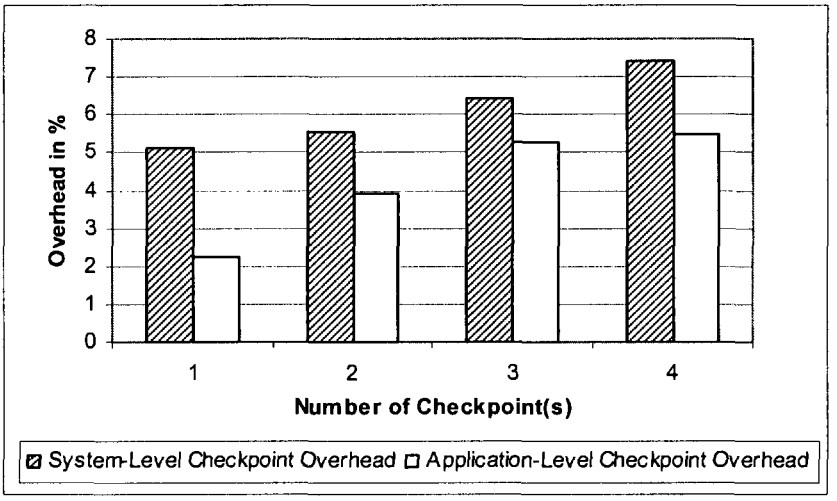


Figure 35: Overhead comparison – application-level and system-level checkpoint

The size of the state information that is saved at each checkpoint for the system-level checkpoint is significantly higher than that of the application-level checkpoint. Figure 35 illustrates the increasing trend of the checkpoint overhead comparing the two checkpoint protocols. We can interpret that as the number of checkpoints increases, the percentage-overhead increase due to the system-level checkpoint is higher compared to that of the application-level checkpoint. Thus, we can infer that the application-level checkpoint protocol do comparatively better than the system-level checkpoint protocol for the long running parallel applications.

Chapter 7

Conclusion and Future Research

In this thesis, we have classified patterns into sub-patterns based on the fault-tolerance strategies, which are identified based on pattern characteristics. We have presented a model to achieve application-specific fault-tolerance in parallel programming.

The FT-PAS model is based on the PAS model. The FT-PAS addresses issues from a two user group perspective: the application developer and the protocol developer. The FT-PAS provides patterns implementation along with their supported fault-tolerance strategies. This pre-packaged and pre-implemented solution delivers maximum possible separation-of-concern, i.e., to alleviate the application developer's burden due to the fault-tolerance implementation-specific issues.

The protocol developer is responsible for extending existing skeletons with newer fault-tolerance protocols based on need. Hence, the protocol developer is expected to be well experienced with systems-specific issues. The FT-PAS model contributes a set of core facilities to support the protocol extension. Thus, the protocol developer can use these core facilities to build new fault-tolerance strategies. From that perspective, the framework can also be regarded as a test-bed for evaluating newer fault-tolerance protocols.

Future studies can possibly focus on some areas of enhancement and limitation of the current FT-PAS model, of which few are briefed here. Currently, we assume that the

internal-component of the FT-PAS model is failure-free. It is possible to overcome this limitation by making the internal components fault-tolerant. The centralized checkpoint dependency graph used in the FT-PAS model is another limitation which leads to total loss of data when the central resource fails. This limitation can be resolved by managing the checkpoint dependency graph in a distributed manner.

Further extensions can be amended to the model in order to contribute more flexibility in terms of providing fault-tolerance. Further investigation is required in order to address other issues such as compose-ability and adaptability with respect to fault-tolerance.

Compose-ability refers to addressing concerns in order to support fault-tolerance in skeleton composition [41]; whereas, adaptability refers to investigating the need for variable fault-tolerances in an application based on the run-time characteristics. Currently, the fault-tolerance provided for an application in other existing systems, including ours, is based on a single strategy (configured statically before compilation). Whereas, an application might require choosing and adapting its strategy, in such case it needs to be configured with more than one strategy. Thus, the strategy to use gets selected at runtime based on the application's runtime characteristics.

In addition, the FT-PAS model can be extended to support Extensible PAS [41]. A graphical user interface can be amended to the FT-PAS model to ease the users' involvement related to application development and protocol development. All these are potentially candidates that lead us in an interesting direction for future research.

Bibliography

[1] G. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. *In Proc. of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI,* LNCS, vol. 1908, Springer-Verlag, 2000, pp. 346-353.

[2] William Gropp and Ewing Lusk. Fault Tolerance in MPI Programs. *In Proc. of the Cluster Computing and Grid Systems Conference*, Dec. 2002.

[3] Gengbin Zheng, Lixia Shi and Laxmikant V. Kale. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. *In Proc. of IEEE International Conference on Cluster Computing*, Sep. 2004, pp. 93-103.

[4] Andrew Lumsdaine, Jeffrey M. Squyres and Brian Barrett. Reliability in LAM/ MPI Requirements Specification. Technical Report TR563, Department of Computer Science, Indiana University, Jun. 2002.

[5] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. *In Proc. of the IEEE/ACM SC2002 Conference*, Nov. 2002, pp. 29.

[6] Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill. Patterns for Parallel Programming. Software Patterns Series, Addison-Wesley Professional, 2004.

[7] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30 (3) (2004) 389-406.

[8] D. Goswami, Ajit Singh and Bruno R. Preiss. From Design patterns to Parallel Architectural Skeletons. *Journal of Parallel and Distributed Computing*, 62 (4) (2002) 669-695.

[9] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li. Libckpt: Transparent Checkpointing under Unix. *In Proc. of the Usenix Winter 1995 Technical Conference*, Jan. 1995, pp. 213-223.

[10] Poznan Supercomputing and Networking Center Checkpoint library, Poznan Supercomputing and Networking Center. Available from: <http://checkpointing.psnc.pl/>.

[11] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. *In Proc. of the Usenix Winter 1992 Technical Conference*, Jan. 1992, pp. 283-290.

[12] J. Duell, P. Hargrove and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. *Berkeley Lab Technical Report (publication LBNL-54941)*, Dec. 2002.

[13] E.N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34 (3) (Sep. 2002) 375 – 408.

[14] M. Treaster. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. *ACM Computing Research Repository (CoRR)*, (cs.DC/0501002), Jan. 2005.

[15] D. Goldberg, M. Li, W. Tao and Y. Tamir. The design and implementation of a fault-tolerant cluster manager. *Technical Report Computer Science Department (Technical Report CSD-010040)*, Oct. 2001.

[16] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. *In Proc. of the International conference on Parallel Processing*, 1984, pp. 32-41.

[17] M. Chandy and L. Lamport. Distributed snapshot: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 31 (1) (1985), pp. 63-75.

[18] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25 (1987) 153-158.

[19] E.N. Elnozahy, D.B. Johnson and W. Zwaenepoel. The performance of consistent checkpointing. *In Proc. of the 11th Symposium on Reliable Distributed Systems*, Oct. 1992, 39-47.

[20] B. Bhargava and S.R. Lian. Independent checkpointing and concurrent rollback for recovery – An optimistic approach. *In Proc. of the 7th Symposium on Reliable Distributed Systems*, Oct. 1988, pp. 3-12.

[21] W.M. Wang. Checkpoint Space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6 (5) (1995) 546-554.

[22] Dmitry Mogilevsky and Sean Keller. SafeMPI - Extending MPI for Byzantine Error Detection on Parallel Clusters. *Technical Report (CoRR abs/cs/0506001)*, 2005.

[23] A. Cherif, M. Suzuki and T. Katayama. A Novel Replication Technique for Implementing Fault-Tolerant Parallel Software. *Kluwer Academic Publishers, Fault Tolerant Parallel and Distributed Systems*, Jan. 1998, pp. 373-384.

[24] Marco Danelutto. QoS in Parallel Programming through Application Managers. *In Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb. 2005, pp. 282-289.

[25] Gosia Wrzesinska, Ana-Maria Oprescu, Thilo Kielmann and Henri Bal. Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid. *In Euro-Par 2007: Proc. of the European Conference on Parallel Processing,* LNCS, vol. 4641, Springer-Berlin, 2007, pp. 425-436.

[26] Nuno Fonseca and João Gabriel Silva. MPI Farm Programs on Non-dedicated Clusters. *In Proc. of European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI 2003*, LNCS, vol. 2840, Springer-Berlin, 2003, pp. 473-481.

[27] J. Rodrigues de Souza, E. Argollo, A. Duarte, D. Rexachs and E. Luque. Fault Tolerant Master-Worker over a Multi-Cluster Architecture. *In ParCo 2005: Proc. of Parallel Computing*, 33 (2005) 465-472.

[28] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. Risinger and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31 (4) (2003) 285–303.

[29] S. Rao, L. Alvisi and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. *In FTCS '99: Proc. of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999, pp. 48.

[30] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *In HPDC '99: Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999, pp. 167-176.

[31] G. Stellner. CoCheck: Checkpointing and process migration for MPI. *In IPPS '96: Proc. of the 10th International Parallel Processing Symposium*, Apr. 1996, pp. 526-531.

[32] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Computing Series, Addison-Wesley Professional, 1994.

[33] Zunce Wei, Hon F. Li and Dhrubajyoti Goswami. A locality-driven atomic group checkpoint protocol. *In Proc. of the 7th International Conference on Parallel and Distributed Computing, Application and Technologies*, Dec. 2006, pp. 558-564.

[34] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Algorithms*, Jul. 1994, pp. 23-32.

[35] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay and Andrew Lumsdaine. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, 19 (4) (2005) 479-493.

[36] J. Hursey, J.M. Squyres, T.I. Mattox and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. *In IPDPS 2007: Proc. of the IEEE International Parallel and Distributed Processing Symposium,* Mar 2007, pp. 1-8.

[37] Pascal Felber, Xavier Défago and Rachid Guerraoui. Failure detectors as first class objects. *In DOA'99: Proc. of the 1st IEEE Intl. Symposium on Distributed Objects and Applications*, 1999, pp. 132-141.

[38] P. Stelling, I. Foster, C. Kesselman, C. Lee and G. Von Laszewski. A fault detection service for wide area distributed computations. *In Proc. of the 7th International Symposium on High Performance Distributed Computing*, Jul 1998, pp. 268-278.

[39] Georg Stellner. Consistent Checkpoints of PVM Applications. *In Proc. of 1st European PVM User Group Meeting*, 1994.

[40] George Kola, Tevfik Kosar and Miron Livny. Faults in Large Distributed Systems and What We Can Do About Them. *In Euro-Par 2005: Proc. of 11th European Conference on Parallel Processing*, LNCS, vol. 3648, Springer-Berlin, Aug 2005, pp. 442-453.

[41] Mohammad Mursalin Akon, Dhrubjyoti Goswami and Hon Fung Li. SuperPAS: A Parallel Architectural Skeleton Model Supporting Extensibility and Skeleton Composition. *In ISPA 2004: Proc. of International Symposium on Parallel and Distributed Processing and Applications*, LNCS, vol. 3358, Springer-Berlin, pp. 985-996.

[42] Jie Wu. Distributed System Design. CRC Press LLC, 1998.

[43] Ajay D. Kshemkalyani and Mukesh Singhal. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2008.

[44] G. Cao and M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9 (12) (1998) 1213-1225.

[45] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. Computer Systems*, 3 (3) (1985) 204–226.

[46] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI: The Complete Reference. Scientific and Engineering Computation Series, MIT Press, 1998.