LOCALITY-DRIVEN CHECKPOINT AND RECOVERY

ZUNCE WEI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (PH.D.) AT

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2010

© ZUNCE WEI, 2010

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-67379-9
Our file Notre référence
ISBN: 978-0-494-67379-9

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Locality-Driven Checkpoint and Recovery

Zunce Wei, Ph.D.

Concordia University, 2010

Checkpoint and recovery are important fault-tolerance techniques for distributed systems. The two categories of existing strategies incur unacceptable performance cost either at run time or upon failure recovery, when applied to large-scale distributed systems. In particular, the large number of messages and processes in these systems causes either considerable checkpoint as well as logging overhead, or catastrophic global-wise recovery effect. This thesis proposes a locality-driven strategy for efficiently checkpointing and recovering such systems with both affordable runtime cost and controllable failure recoverability.

Messages establish dependencies between distributed processes, which can be either preserved by coordinated checkpoints or removed via logging. Existing strategies enforce a uniform handling policy for all message dependencies, and hence gains advantage at one end but bears disadvantage at the other. In this thesis, a generic theory of Quasi-Atomic Recovery has been formulated to accommodate message handling requirements of both kinds, and to allow using different message handling methods together. Quasi-atomicity of recovery blocks implies proper confinement of recoveries, and thus enables localization of checkpointing and recovery around such a block and consequently a hybrid strategy with combined advantages from both ends.

A strategy of group checkpointing with selective logging has been proposed, based

on the observation of message localization around 'locality regions' in distributed systems. In essence, a group-wise coordinated checkpoint is created around such a region and only the few inter-region messages are logged subsequently. Runtime overhead is optimized due to largely reduced logging efforts, and recovery spread is as localized as region-wise. Various protocols have been developed to provide trade-offs between flexibility and performance. Also proposed is the idea of process clone that can be used to effectively remove program-order recovery dependencies among successive group checkpoints and thus to stop inter-group recovery spread.

Distributed executions exhibit locality of message interactions. Such locality originates from resolving distributed dependency localization via message passing, and appears as a hierarchical 'region-transition' pattern. A bottom-up approach has been proposed to identify those regions, by detecting popular recurrence patterns from individual processes as 'locality intervals', and then composing them into 'locality regions' based on their tight message coupling relations between each other. Experiments conducted on real-life applications have shown the existence of hierarchical locality regions and have justified the feasibility of this approach.

Performance optimization of group checkpoint strategies has to do with their uses of locality. An abstract performance measure has been proposed to properly integrate both runtime overhead and failure recoverability in a region-wise manner. Taking this measure as the optimization objective, a greedy heuristic has been introduced to decompose a given distributed execution into optimized regions. Analysis implies that an execution pattern with good locality leads to good optimized performance, and the locality pattern itself can serve as a good candidate for the optimal decomposition. Consequently, checkpoint protocols have been developed to efficiently identify optimized regions in such an execution, with assistance of either design-time or runtime knowledge.

Acknowledgements

First and foremost, I would like to express my sincere gratitude and appreciation to my supervisors Dr. Hon F. Li and Dr. Dhrubajyoti Goswami, for their continuous support to my Ph.D. study, for their patience, motivation, enthusiasm, and immense knowledge, and for their valuable guidance in all the time of this research as well as my thesis preparation.

Besides my supervisors, I am grateful to the rest of my thesis committee: Dr. Azzedine Boukerche, Dr. Ferhat Khendek, Dr. Sudhir P. Mudur, and Dr. Juergen Rilling, for their insightful comments and suggestions.

Also, I am indebted to my parents for their constant understanding as well as encouragements throughout my life.

My special thanks also go to my fellow labmates in Distributed Systems Laboratory of Concordia University, for all the stimulating discussions and the memorable time that we had together.

Last but not the least, I would like to thank all my friends in Concordia and Montreal who has directly or indirectly helped me, by any means, to make this thesis possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed systems have gained wide popularity in many application areas, due to the increasing demand for many of their new features like high reliability and scalability. Typical examples are distributed multi-agent systems [SL09], such as those designed for e-commerce, network management, supply-chain management and distributed constraint optimization [WGP03, MLY06, Mei08]. In such systems, agents are autonomous objects that interact with each other via message passing [SDP+96, AL98, Liu01]. An agent can perform many of its tasks, independently or through message coordination with other agents. A multi-agent system is hence a special distributed system with mission-driven agent behaviors. This thesis takes such systems as the default objects of study.

Role-based models [Ken98, Ken00, SE05] are useful to understand the abstraction of agent interaction and coordination. As a specification tool for agent design, it provides a means to decompose and then distribute tasks among agents. An after-decomposition task can be accomplished by involved agents via coordination. The roles played by an agent correspond to its potential behaviors and meanwhile indicate its partners, especially in terms of interactions. In general, an agent can take one or several roles simultaneously, which complicates the resulted runtime behavior. On the other hand, an agent can take different roles at different time. Relationships between specific roles played by agents can be modeled as protocol diagrams [OPB00, BMO01, OMG03].

Multi-agent systems are usually more complicated than other distributed systems, due to the following properties: i) a multi-agent system usually consists of many agents that are distributed in different hosts, involving a large number of interaction messages among each other; and ii) an agent may autonomously make different decisions on performing its roles in different situations, making its behavior more unpredictable. Such increased complexity as well as unpredictability has put more challenges on system design and maintenance for distributed agents.

Fault-tolerance is an important design objective of distributed systems. At run time, an application agent is a distributed process that might crash due to many reasons. The ability to tolerate such failures will be an attractive feature of multi-agent systems. Various studies have been made on fault-handling as well as fault-tolerance for such systems, and corresponding strategies have been proposed in the literature from different perspectives. For example, an agent system could have special sentinel agents [Hag96] that guard some predefined functions or system states by monitoring behaviors of other agents. Faults at those suspected points can then be easily detected and corrected. Alternatively, an open agent society could have public exception handling services [KRD03] for agents. This creates a layer of specialized fault-handling capability that is efficient but requires agent designers to make use of them. In some cases [KCL00], important agents such as broker agents can form a team so that team-mates can substitute and restart one another in case of member crash. These approaches work well for specialized agent systems but will become expensive when applied to large-scale multi-agent systems. In addition, they are not general enough to provide application-transparent fault-tolerance. Besides, there are also fault-tolerant frameworks [SSS+99, PPG00, MBS03, HSP05] that provide development support and runtime environment for agent applications. However, they only focus on fault-tolerance features of specific aspects

such as mobility, and do not provide support for fault-tolerance of general application agents. This thesis aims to develop an application-transparent fault-tolerance methodology that can be used generally for all distributed systems. To achieve application-level transparency, a failure-free distributed kernel is assumed to sit aside each agent, taking care of all fault-tolerance actions for that agent.

As an add-on to the underlying distributed system, a fault-tolerant strategy is expected to enhance system reliability without significantly impacting its performance. Process replication [GS99, GFB05] is a useful technique to achieve fault-tolerance, especially for those storage-intensive or service-oriented applications. However, this technique is not adequate for many of other agent-based systems, due to their severe requirements on computational resources, memory occupation, etc. Checkpoint and rollback-recovery [KT87, EAW+02] use another strategy to provide fault-tolerance: a checkpoint is taken as a process snapshot from which the process can reset its local state and resume its execution upon failure. This is a more generic and flexible strategy, as it turns a failure into the temporary unavailability of some process(es) with certain cost of stable storage. So far two categories of checkpoint and recovery strategies have been proposed. In log-based recovery [SY85, AER+99], events of message interactions are all logged and upon failure related messages will be replayed to deterministically reincarnate the failed process. Checkpoints are taken only to trim the storage cost of message log. In recovery, only a few (and usually just one) processes have to be rolled back. The disadvantage is that, logging every message is expensive in terms of both runtime overhead and storage cost, especially for those communication-intensive applications. Also, to ensure that a faulty process's behavior with respect to the rest of the system will remain the same, non-deterministic events must also be recorded at run time and be enforced during process reincarnation. In contrast, checkpoint-based recovery [TS84,

KT87, LB88] relies on well-designed algorithms to take coordinated checkpoints. Upon process failure a consistent recovery line [Ran75] is formed based on the most recent coordinated checkpoint. Subsequently all relevant processes will be rolled back and the system will be restored to a globally consistent state. Runtime performance in this case will become much better as there is very little need for message logging as well as replaying. However, global checkpoint coordination still incurs runtime overhead in both space and time. In addition, it is possible that the failure of a single process leads to a global recovery of all processes, which is disastrous for large-scale distributed systems.

Since the introduction of the above strategies, various improvements have been proposed, aiming at runtime optimization of (i) the size of message logs and checkpoints, and (ii) synchronization delays of message logging or checkpointing. For example, strong checkpoints [HNR99] form recovery lines that do not contain messages in transit, which eliminates the necessity of message playback at recovery time. Optimistic logging [SY85] records messages in a non-blocking manner by taking the risk of involving other processes into recovery. Non-blocking checkpoint protocols [CL85, CS98] reduces the coordination latency in the underlying application with the cost of extra coordination messages. Complicated algorithms are employed in communication-induced checkpoints [MS96, AER+99] and causal logging [Alv96], for the purpose of relaxing the checkpoint coordination or message synchronization requirement. From another perspective, object-based checkpointing [THT98] distinguishes "influential messages" that change object states from other messages, making it possible to use an inconsistent global state as a recovery line.

1.1 The problem

A multi-agent system usually consists of a large number of agents as well as interaction

messages that are dispersed throughout the network. Existing strategies for checkpoint and recovery are not favorable in such a large-scale system. For example, traditional log-based recovery is expensive due to the large amount of agent application messages that need to be logged. Although optimistic logging [SY85] or causal logging [Alv96] can be used to improve the runtime performance, it is inevitable but meanwhile undesirable to record all message events and non-determinisms. Moreover, it is still possible to have unexpected recovery situations due to the unpredictability of agent behaviors. On the other hand, coordinated checkpointing incurs global efforts in both checkpoint coordination and failure recovery. The large number of agents hence involved can lead to significant blocking of runtime execution and unnecessary rollback of application progress. Certain checkpoint protocols [KY87, CS98, AER+99] are designed to take partial rather than global-wise consistent checkpoints, in order to minimize the coordination efforts. However, since checkpoints can be initiated arbitrarily, a recovery might still involve many processes if some checkpoints are taken at inappropriate points of execution. The resulted situation is very similar to the one where independent checkpointing causes domino effect [Ran75].

It is clear that existing checkpoint and recovery strategies are designed for two different application situations. Checkpoint-based strategies have advantages in runtime performance but suffer from uncontrollable global recovery effect, therefore are better used in computation-intensive systems with rare failures. Log-based strategies preserve good recoverability but greatly harm runtime performance, and hence are more applicable to communication-based collaborative systems. When applied to a multi-agent system, these strategies will introduce either too much runtime overhead to the failure-free execution, or too much recovery effort upon failure. In fact, the large-scale-ness of such a system decides that it is not affordable to go to any of these two extremes by sacrificing

the other aspect of system performance. Instead, an appropriate checkpoint and recovery strategy should be able to flexibly trade off between, and if possible to efficiently integrate both of, the runtime performance and the failure recoverability.

This thesis aims at providing a general and efficient checkpoint and recovery solution for large-scale distributed agent systems, with both affordable runtime overhead and controllable recovery effect. Such a fault-tolerant solution will distinguish itself from existing ones through the following features: i) it will serve as a part of the system kernel and will be able to tolerate crash failures of any application agent without any intervention from developers; ii) it will work efficiently with distributed systems in different scales, especially those large-scale multi-agent systems; iii) it will improve the system efficiency at both run time and recovery time; iv) it will provide fine-grained performance tuning in terms of runtime overhead and recovery cost. General-purpose checkpoint and recovery solutions proposed so far do not have good scalability and hence are not able to achieve performance improvement at both ends. Furthermore, none of those solutions provides a quantitative measuring or adjusting approach that can be used for optimizing system performance.

Proposing the above solution raises challenging issues to be addressed in this thesis. In distributed systems, message interactions create dependencies that often spread across processes. As a result, tracking and restoring such dependencies require global-wise coordination and recovery. Also, their removal requires proper handling (e.g., logging and replaying) of corresponding messages with considerable overhead. The necessity of so doing originates from the correctness requirement of rollback recovery. In general, a message dependency has to be taken care of in either way, incurring different kinds of overhead. The target of reducing overhead at both ends naturally gives rise to a hybrid strategy with less intensive message logging and localized coordination as well as

recovery. This in turn requires selectively logging certain parts of message dependencies and meanwhile tracking as well as restoring the rest. Existing strategies are based on unnecessarily tight requirements such as globally-consistent recovery line [Ran75] and individually-consistent process state [SY85]. New strategies with promising performance will rather encourage a carefully-revised theory of checkpoint and recovery, in particular, a relaxed requirement that accommodates selective processing of performance-related dependencies. Also, since there are so many possibilities to decide the set of messages to be logged, a critical issue is how to find the one with the optimized performance. Further study requires proper modeling of overall performance during both checkpointing and recovery, formulation of the optimization problem, and development of algorithms towards the optimum solution. Finally, there is also the need of designing various protocols that can manage checkpointing and recovery actions with respect to the optimized performance.

1.2 Ideas and Contributions

The way of handling message dependencies is directly related to the system performance. Runtime overhead and recovery cost vary largely with the pattern of message interactions. However, existing checkpoint and recovery strategies have significantly ignored its importance. For example, coordinated checkpointing [KT87, EAW+02] was initially proposed to avoid the domino effect [Ran75] introduced by independent checkpointing. It forces a simple and uniform policy that globally tracks and restores all message dependencies. The resulted system has a fixed overhead of global coordination and recovery effect, which is independent of the actual pattern of message interactions. Meanwhile, this eliminates the possibility of having improved performance by taking checkpoints at proper locations. Log-based protocols force another uniform policy that

removes all message dependencies. There are also checkpoint protocols [KY87, CS98, AER+99] that are developed to minimize coordination efforts by allowing partial coordinated checkpoints. Since these partial checkpoints are not coordinated with each other and their locations could be arbitrary, it is possible that message dependencies will spread among these independent partial checkpoints and cause similar domino effects upon failure. All the above indicates the possibility as well as the necessity of improving performance from the perspective of studying message dependency patterns.

In a distributed system, correct rollback recovery requires proper maintenance of all failure-affected dependency relations. The purpose of checkpointing as well as logging is to pre-record sufficient dependency information for future use. From this perspective, both runtime overhead and recovery cost can be measured by the quantity of dependency relations being involved. Consequently, the actual dependency pattern of a distributed execution can create big differences in these two aspects of performance, due to the variations and complexity introduced by the large number of processes and messages. Since all processes execute sequentially and the corresponding program-order dependencies are just regular total orders, the dependency pattern of a distributed execution is mostly based on the pattern of message interactions.

Observations on real-life multi-agent applications have shown considerable signs of message interaction patterns, where messages and processes tend to localize in different ways. In particular, a process over a time period only interacts with a subset of processes rather than with all processes of the system. For example, in an agent-mediated e-commerce auction system [GMM98, GW03], a bidder agent might frequently interact with a rather exclusive subset of agents including a buyer, an auctioneer and other mediators until an auction concludes. During the auction, the various participating agents have lifelines that are coupled together and form a message-intensive sub-region in space

and time. Moreover, during an execution of the auction system, messages are mostly localized within different sub-regions and there are relatively fewer messages in between. In fact, many distributed applications exhibit similar regions of messages, in different forms.

This locality phenomenon can be tracked back to agent design as well as development. Agent applications are role-oriented. Each role played by an agent defines its position, personality and interaction capability, which actually demonstrate its "working set" [Den76] of interactions for a certain period of time. Hence agents are highly likely to exhibit locality in terms of interactions, in both space and time. This knowledge of working (locality) set, which actually demonstrates the agent interaction pattern, can be exploited to guide the design of efficient checkpoint and recovery strategies, in order to improve the runtime performance and recoverability of fault-tolerant multi-agent systems. In particular, a number of processes, which are closely-coupled for a period of time via frequent message interactions, often form a space-time sub-region that contains most of messages exchanged among them. In fact, an execution of distributed applications usually observes such distinct 'locality regions', covering majority of their message interactions. Since each region involve only a few processes of the whole system, these processes can be grouped together to take a coordinated checkpoint around the region. Such a group-wise checkpoint in turn has only localized coordination as well as recovery effect. On the other hand, locality regions are almost disjoint from each other as there are relatively few messages in between. Therefore it is affordable to selectively log the inter-region messages so as to remove the corresponding dependency relations among groups. Locality of message interactions hence makes it possible to achieve both localized recovery effect and reduced logging overhead in large-scale distributed agent systems.

Based on the above idea, this thesis has developed results in the forms of both theories and techniques, detailed as follows.

(a) *Group checkpoint and recovery.* A new theory of Quasi-Atomic Recovery is formulated and it accommodates both global and individual consistencies, providing the correctness foundation for the new locality-driven checkpoint and recovery strategies. A Checkpoint Dependency Graph (CDG) model captures the effective dependency relations among checkpoints, and hence simplifies the design as well as the understanding of new checkpoint protocols. The strategy of group checkpointing and selective logging is properly developed, and a new technique based on 'process cloning' is also introduced to properly handle program-order dependency relations that might exist between group checkpoints during recovery. Various group checkpoint protocols featuring strong-ness, atomicity, non-atomicity, and k-bounded-ness are designed, providing efficient trade-offs between flexibility and performance.

(b) *Locality of message interactions.* The locality phenomenon of message interactions is first studied from the perspective of distributed program design. Its origination is analyzed in terms of dependency localization via message passing, and its exhibition is characterized as a hierarchical 'region-transition' pattern in distributed systems. A bottom-up research approach is then proposed to identify such regions in a given distributed execution. In particular, a generic notion of 'locality interval' is developed to capture popular recurrence patterns within individual process lifelines, and frequent message interactions between such intervals are modeled as 'important coupling', by which they form a 'locality region' of localized messages. Experiments conducted on real-life distributed applications justify the feasibility of the proposed approach as well as existence of the modeled locality.

(c) *Locality-driven checkpoint and recovery.* Performance optimization of group

checkpointing is studied and an abstract performance measure is proposed to capture both runtime overhead and failure recoverability in a region-wise manner. Using this measure as the optimization objective, a greedy heuristic is developed to decompose a distributed execution into its locality regions via region merging. A merging criterion is reasoned, which implies that performance optimality is related to locality of message interactions. In particular, a pattern with good locality leads to good performance, and the pattern itself can serve as a good candidate for the optimal solution. Strategies as well as checkpoint protocols are developed to detect locality regions in a given pattern, with assistance of design time or runtime knowledge. In addition, various design choices for checkpoint protocols are analyzed with respect to their performance costs, revealing the reasons behind flexibility and performance.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives an introduction to the research works proposed in the area of distributed checkpoint and recovery, including the correctness modeling and consistency theories, and detailed protocols from the two existing approaches. Chapter 3 presents the new results on checkpoint and recovery theories as well as techniques, such as Quasi-Atomic Recovery, CDG, selected group checkpoint protocols, and the idea of cloning-based recovery. Chapter 4 models the locality phenomenon of message interactions in terms of locality intervals and locality regions, and verifies the existence of such locality properties in real-life distributed agent applications. Chapter 5 discusses performance measuring as well as optimization, and presents strategies that work with the new checkpoint protocols using design-time or runtime assistance. Chapter 6 summarizes the thesis with a discussion on the future research directions.

Chapter 2

Distributed Checkpoint and Recovery

Checkpoint and rollback-recovery are popular techniques to achieve fault-tolerance in distributed systems. A recovery is a dynamic manipulation of the original execution triggered by the unexpected occurrence of a failure. The correctness of recovery is traditionally based on the consistency of the after-recovery global state.

Refinements of the above consistency lead to two different categories of recovery strategies. Since any global state of the failure-free execution is naturally consistent by itself, a simple choice is to make the recovered global state identical as before failure, so that the resumed execution can continue as if the failure never occurs. Log-based recovery takes this strategy by making the following piecewise deterministic (PWD) assumption [SY85]: all non-deterministic events can be identified and logged. By deterministically replaying the messages logged at run time, a failed process can be recovered to the same state right before failure. In contrast, checkpoint-based recovery records recent consistent global states as coordinated checkpoints, which upon failure will serve as recovery lines [Ran75] and the whole system will be resumed to the most recent one. With such a strategy, there is no need to log and replay every message, and no guarantee of deterministic recreation of the original execution as well. The resumed execution might not proceed exactly as before, but it is still a legal one based on the program statements.

Rollback-recovery requires runtime recording of past information about the failure-free execution, such as checkpoints and message events. In general, availability of such information automatically decides the recoverability of a failure, i.e., the minimum recovery effect, and consequently the set of useless checkpoints to be removed via garbage collection. Apparently, strategies resulting in better runtime performance are usually equipped with a complicated recovery protocol, and those suffering from performance degradation often have a simple recovery protocol.

2.1 Model and Consistency

A distributed (multi-agent) system consists of a set of processes (or agents) that interact only via messages passing. Message delivery is through reliable channels and takes unpredictable but finite delay. Each process has a set of totally-ordered local events including message sending and receiving. For a message m, $send(m)$ and $recv(m)$ denote respectively its send and receive event. Messages also introduces partial ordering among events of different processes and the resulted dependencies are captured by Lamport's happened-before relation (usually denoted as $\rightarrow$) [Lam78]. Processes (or agents) are subject to crash failures, in which case it loses its volatile state and stops execution according to the fail-stop model [SS83]. They have access to a stable storage that can survive all tolerated failures. Checkpoint and recovery strategies use this device to save recovery information (i.e., checkpoints, message log, etc.) periodically during failure-free execution. A generic correctness condition for rollback-recovery has been proposed informally as follows: "A system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure" [SY85].

2.1.1 Consistent Global Checkpoint

A *global state* of a message-passing distributed system is a collection of individual states

of all processes and message channels within the system. A global state is said *consistent* if when the state of a process reflects a message receipt, the state of the corresponding sender also reflects sending of that message [CL85]. Figure 2.1 shows an example distributed system consisting of three processes, p_1, p_2, and p_3, with two messages m_1 and m_2 among them. The global state in Figure 2.1 (a) is consistent and it has a non-empty channel state with an *in-transit message* m_1. The global state in Figure 2.1 (b) is inconsistent because the local sate of p_3 reflects receipt of m_2 but the local state of p_2 reflects m_2 is not sent yet. An inconsistent state violates causality and is only possible in case of a process failure and a subsequent inappropriate recovery. A correct rollback-recovery protocol is required to eventually recover the system to a consistent global state.

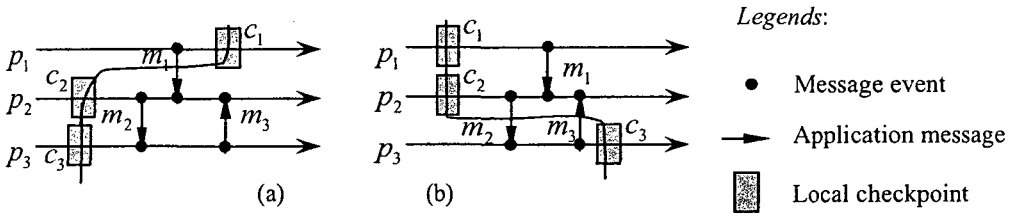


Figure 2.1 Consistent and inconsistent states

A *local checkpoint* is a record of a process state taken at run time. It contains all local information that is necessary for a process to resume its execution. Similarly, a *global checkpoint* can be considered as a recorded global state of the system. It usually consists of a set of local checkpoints, one from each process, but can also involve logs of in-transit messages as corresponding channel states. A consistent global checkpoint is one whose corresponding global state is consistent. To guarantee consistency, checkpoint-based recovery strategies manage processes to take global checkpoints in a coordinated manner. Upon failure, the recovery protocol will construct the recovery line based on the most recent consistent global checkpoint and the system state will be restored correspondingly. Due to the non-determinism of distributed executions, the system might not proceed as same as before failure. In the above example, if the global state in Figure

2.1 (a) is used as a recovery line, p_2 could receive m_3 before receiving m_1 in a resumed execution.

For a set of local checkpoints, one per process, to form a consistent global checkpoint, there should be no "orphan message" in the corresponding system state. A message becomes an *orphan* if its receive event is recorded but its send event is not, e.g., m_2 in Figure 2.1 (b). An orphan message introduces an explicit ordering pattern between two local checkpoints, e.g., $c_2 \rightarrow send(m_2) \rightarrow recv(m_2) \rightarrow c_3$ in Figure 2.1 (b). Existence of orphan message(s) across multiple processes is captured by the generalized notion of zigzag path [NX95] or z-dependence [CS98]. Intuitively, a zigzag path is a sequence of messages that establishes an ordering pattern with the same recovery dependency as a single orphan message between two local checkpoints. As an example, in Figure 2.1 (b), m_1 and m_2 form a zigzag path between c_1 and c_3. Consequently, there is a recovery dependency established between c_1 and c_3. This has the same effect as having a single orphan message sent after c_1 and received before c_3. It can be proved that orphan message(s) is observable in any set of local checkpoints that involves a zigzag path. A set of local checkpoints can form a consistent global checkpoint if and only if there is no zigzag path between any two local checkpoints.

In-transit messages might exist and sometimes are inevitable when taking a consistent global checkpoint. Such messages should be logged and replayed for the purpose of reconstructing the channel state. An extreme example is log-based recovery, in which case only one process is involved so that its local checkpoint is consistent with itself and all messages have to be replayed. On the other hand, there is also a notion of transitless global checkpoint [HNR99], which deserves the property that message logging can be avoided during checkpointing. A global checkpoint is *transitless* if there is no in-transit message in the corresponding global state. Note that orphan messages are allowed

in this case, hence a transitless global checkpoint is not necessarily consistent. Figure 2.2 (a) shows such a global checkpoint. If a transitless global checkpoint is consistent, it will be called a *strong* consistent checkpoint [HNR99], or simply strong checkpoint. An example of strong checkpoint is given in Figure 2.2 (b). Strong consistency implies no necessity of message logging or replaying and consequently efficient checkpoint and recovery.

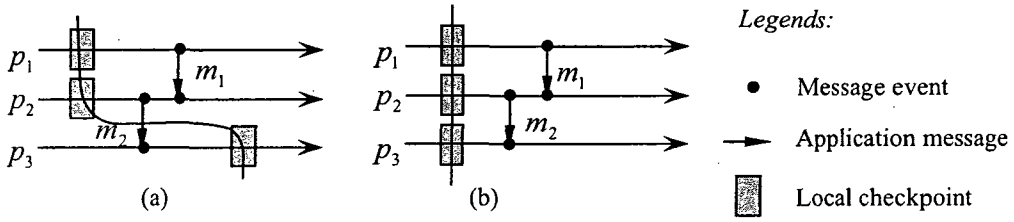


Figure 2.2 Transitless and strong checkpoints

Taking consistent global checkpoints usually requires all processes to coordinate their actions at run time, and consequent global delay causes performance degradation. This consistency requirement is actually driven by the strategy of global recovery. It is therefore sufficient but not always necessary for the eventual purpose of recovering the system into a consistent global state. In other words, inconsistency could be another choice of checkpointing if there is a way to properly recover the system state from an inconsistent global checkpoint. For example, in-transit messages need to be logged at run time so that they can be replayed for recovery. Similarly, orphan messages within inconsistent global checkpoints also needs proper handling at both run time and recovery time. In particular, there should be techniques to identify and to discard those orphan messages during recovery, which might require special logging of certain information of such messages. In fact, message discard as a technique has been used in specific applications [BMP+03] and it is possible to adapt it to checkpoint and recovery protocols.

On the other hand, a consistent global checkpoint is not necessarily taken always global-wise. A process failure might only affect a few other processes, if message

dependencies have not yet spread very far since the last global checkpoint. It hence will be efficient to manage a partial instead of global checkpoint among a subset of processes. In fact, several checkpoint protocols [KT87, CS98] have been proposed to take such partial checkpoints for the purpose of minimizing coordination efforts. However, message dependencies are still possible between these partial checkpoints since they are not coordinated with each other while being taken. Upon failure, the recovery protocol has to track all dependencies in order to construct the recovery line efficiently.

2.1.2 Always-No-Orphans Consistency

In contrast to checkpoint-based recovery that focuses on global consistency, log-based recovery aims at maintaining individual consistency of each process. In particular, a process execution is modeled as a sequence of deterministic state intervals, each starting from an event that has non-deterministic timing or computational result [SY85]. Examples of such events include the receipt of a message and an internal event like random number generation. Note that the sending of a message is determined by the results of its proceeding events and hence is not a non-deterministic event by itself. Log-based recovery relies on the piece-wise deterministic (PWD) assumption [SY85], which claims that any non-deterministic event executed by each process can be identified and logged as a *determinant* [Alv96] that is necessary to replay the event deterministically. For example, the determinant for (the receive event of) a message m includes the message itself (i.e., both the header and the body) and its receipt (which specifies the original order of m's delivery). Log-based strategies need to record not only all messages but also all determinants. In comparison, checkpoint-based strategies only log the in-transit messages. Upon failure, only the failed process needs to be recovered from its most recent checkpoint. All subsequent messages will be replayed deterministically afterwards, and its local state will be restored exactly as before failure.

Logging of messages and determinants has specific synchronization requirements. If a process fails before a message log is committed into a stable storage, the deterministic recreation of that message will not be guaranteed. As a result, another process might be in an inconsistent state. For the example shown in Figure 2.1 (a), suppose all messages are logged except m_2. If p_1 rolls back from its checkpoint, p_2 will be inconsistent with respect to other processes. p_2 is hence called an *orphan process* and is forced to rollback with p_1. To avoid such a situation, a general "always-no-orphans" condition [AM98] requires that for any non-deterministic event e, if the determinant of e is subject to loss (i.e., not committed into stable storage yet), the processes whose states are depending on e should keep a copy of its determinant, either independently or collectively. If not, it is possibly to have an orphan process in case e cannot be regenerated. Refinement of this condition leads to different logging strategies, which are introduced in Chapter 2.3.

The PWD assumption makes it possible to precisely recreate the system state, and hence is specifically useful for situations involving un-recoverable units. For example, an open distributed system can have message interactions with the outside world, which is usually modeled as an Outside World Process (OWP) [SY85]. Such a process is independent of the system and is not supposed to be affected by any of its recovery actions. Messages related to that process should be logged properly and be replayed deterministically in case of failure. The PWD assumption hence is necessary to separate recovery spread among processes with the assistance of certain logging strategies.

2.2 Checkpoint-Based Recovery

Checkpoint-based recovery does not log every message or any determinant, and hence avoids the corresponding runtime latency. Instead, it employs efficient checkpoint and recovery protocols to restore the system into a past consistent global state. Checkpoints

from different processes can be taken independently or coordinately in various manners. Consequently, a process failure will lead to recoveries in different scales.

2.2.1 Independent Checkpointing

In general, a (local) checkpoint is a snapshot of a process state by which the process can reset its local state and resume its execution. The decision of taking a local checkpoint can be made independently by each process. Independent checkpointing [BL88] is simple to implement, as each process has the maximum autonomy to decide where to checkpoint. Also there is no additional effort or latency introduced by either message logging or coordination with other processes. However, there are several undesirable disadvantages: i) it is sometimes inevitable to have domino effect during recovery, which could force the whole system to restart from the beginning; ii) it can create useless checkpoints that will never be a part of consistent state and hence will never be used in any recovery; iii) each process has to keep multiple checkpoints and check their usefulness from time to time, causing increased storage and effort for garbage collection.

The autonomous nature of independent checkpointing requires special treatment of recovery dependencies in order to find the recovery line. At runtime, message interactions create dependency relations among checkpoints from different processes. Such dependency information can be piggybacked with the corresponding message and is recorded by the receiver for recovery use. Upon failure, the recovering process will calculate the most recent consistent recovery line by requesting dependency information from all other processes. In particular, rollback-dependency graph [BL88] and checkpoint graph [Wan93] have been proposed to help tracing the recovery dependencies among checkpoints via reachability analysis. These two graphs have similar semantics and equivalent output of the recovery line. On the other hand, checkpoints preceding the most recent recovery line are no longer useful, and therefore should be removed from time to

time. This garbage collection procedure can also be applied to find useless checkpoints.

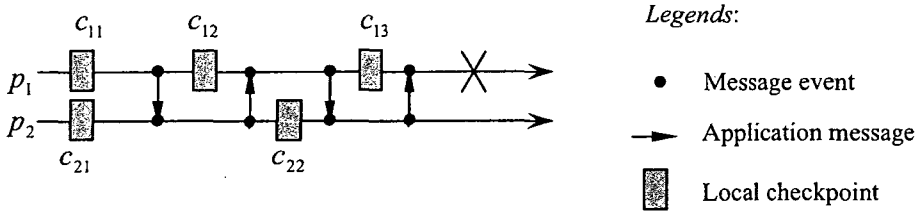


Figure 2.3 An example of domino effect

Depending on the pattern of independent checkpoints and message interactions, a recovery might involve different numbers of processes. The domino effect [Ran75] is the most unfavorable recovery situation, as it can result in huge waste of computational effort. As shown by the example in Figure 2.3, upon failure, p_1 rolls back to its most recent checkpoint c_{13}. Since no message is ever logged, in order to find a consistent recovery line, p_2 rolls back to c_{22}, which in turn triggers p_1 to rollback from c_{12}. This propagation never stops until both processes rollback to the beginning. All work that has been done so far is lost and all checkpoints that are taken previously hence become useless. The autonomy of individual decision-making eliminates the necessity of checkpoint coordination, but on the other hand ignores the impact of message dependency pattern. Unlike log-based recovery, no action is taken at runtime to control the dependency spread. As a result, independent checkpointing has uncontrollable recovery effect.

2.2.2 Coordinated Checkpointing

To avoid domino effects as well as useless checkpoints, strategies are proposed to synchronize the checkpointing actions of processes in order to take a consistent global checkpoint. Such a checkpoint is expected be an on-the-fly record of a valid global system state, which is naturally consistent by itself. This in turn requires every process to take a local checkpoint and each in-transit message to be logged as a part of the corresponding channel state. As introduced in Chapter 2.1.1, to guarantee the consistency of a global checkpoint, orphan messages have to be avoided and there should be no

zigzag path between any two local checkpoints. In particular, since an orphan message leads to an ordering pattern, the checkpointing actions of any two processes should be un-ordered with respect to all in-transit messages between them. Usually an application-transparent checkpoint and recovery protocol is superimposed on top of the underlying computation and the consistency requirement is only regarding application messages. As a result, the synchronization of un-ordered checkpointing actions can be achieved with by using extra system-level messages.

A simple approach to take coordinated checkpoints is to suspend the normal execution until the checkpointing is done [TS84]. An initiator process can stop its execution and send a request to every other process, which in turn will stop its execution, flush all communication channels, take a tentative checkpoint, and send a feedback to the initiator. After collecting feedbacks from all processes, the initiator will broadcast a commitment for them to change their tentative checkpoints into permanent and resume their execution. Since message channels are not assumed as FIFO, this protocol uses CRC vector to make sure the channel-flushing procedure properly records all in-transit messages. This two-phase coordination is based on explicit system-level messages. Consistency is guaranteed by instant blocking of execution upon receipt of a checkpoint request. At runtime, each process keeps only one permanent checkpoint in stable storage. There is no useless checkpoint and the storage overhead is minimal. Garbage collection also becomes very simple as the old global checkpoint is removable once a new one is taken. The main disadvantage is, the failure-free execution is totally blocked during checkpoint coordination and the consequent large latency will harm the performance.

A global checkpoint can be also taken as a distributed snapshot [CL85] in a non-blocking manner. This approach assumes that all channels are in FIFO order, for both application- and system-level messages. In the beginning, the checkpoint initiator takes a

local checkpoint, and sends to each of its neighbors a special marker message, which will be also forwarded by the receiver to its other neighbors. A local checkpoint will be taken upon the first time of receiving a marker message, and application messages received afterwards will be recorded as corresponding channel states. Such recording will not stop for a particular channel until a marker message is received from that channel. In this protocol, FIFO-ness of channels guarantees completeness of channel state as well as consistency between local checkpoints, and also eliminates the necessity of blocking the failure-free execution. Global-wise synchronization is avoided as overhead is only created while a local checkpoint is being committed to stable storage. Further performance improvement towards reducing the stable storage contention is proposed as staggered checkpointing [Vai99, MJY+05], which employs a refined version of distributed snapshot to take a consistent "logical checkpoint".

Instead of allowing just one initiator at a time, any process can autonomously perform a global checkpoint initiation, by taking a tentative checkpoint and then logging messages received thereafter [JLM08]. Knowledge can be spread asynchronously via message piggybacking to inform processes the current status about each other. A tentative checkpoint will be finalized into an effective checkpoint if a process collects enough knowledge to realize that every other process already establishes a tentative checkpoint. Therefore in this protocol, a finalized checkpoint actually consists of a tentative checkpoint, which is the local process snapshot, plus a sequence of messages logged afterwards. Consistency of global checkpoint cut is guaranteed by forcing a receiver to finalize its own checkpoint before receiving a message sent after the sender's finalization. This protocol is asynchronous but its convergence of involving every process depends on the actual message pattern. To force the convergence of global checkpoint finalization, additional control mechanism can be implemented via explicit kernel messages.

Coordinated global checkpoint is usually simple and synchronous. Global-wise synchronization involves all processes at run time as well as recovery time, and is hence not appreciated in many circumstances. However, due to the simplicity of implementation, it is still applicable under certain situations such as for clusters [BP07], mobile computing [Kum08], and grid computing [JKN06].

2.2.3 Minimal Checkpoint Coordination

In general, a global state of a distributed system involves every process as well as every non-empty message channel. As a result, coordinated checkpoint strategies usually force all processes to synchronize their checkpointing actions. However, this is a sufficient but not always necessary condition from the perspective of maintaining consistency. For example, if since last global checkpoint a subset of processes never interact with the rest of processes, then a consistent snapshot of this subset is always consistent with any consistent snapshot of the rest world. Consequently, this subset of processes can take a partial coordinated checkpoint involving only themselves, instead of also enforcing all other processes to checkpoint together. The coordination effort hence can be reduced from global-wise to group-wise. This approach is essentially captured by the notion of "min-process" checkpointing [CS98, CS03], which requires all processes that develop z-dependency relations to coordinate their checkpointing actions. A process p z-depends on q if there is a zigzag path from p to q since their last checkpoints. Obviously the above requirement avoids inconsistency among such z-dependents. It can be proved that "min-process" checkpointing only involves a minimum number of processes in taking each coordinated checkpoint. Corresponding protocols hence have the advantage of minimal coordination effort. On the other hand, it also makes sense that all collaborating processes should save their periodic effort by taking a coordinated checkpoint together.

It has been proved that non-blocking min-process protocol doesn't exist [CS98]. A

blocking min-process protocol can employ a two-phase commit procedure for both checkpointing and rollback-recovery [KT87]. In the first phase, the initiator takes a tentative checkpoint and sends checkpoint requests to other processes that z-depends on it. Upon receiving a request, a process will take a tentative checkpoint and forward the request to its own z-dependents. After the initiator learns all such processes have taken their tentative checkpoints, in the second phase it will propagate its decision of turning all checkpoints into permanent. Upon failure, the failed process will employ the similar procedure to locate all its z-dependents and request them to recover together. All related processes have to be blocked until the second phase is completed. To avoid execution blocking, a checkpoint sequence number *csn* can be used to facilitate the two-phase commit procedure. It is piggybacked by a sender process with each of its application messages, playing a role similar to the marker message in distributed snapshot [CL85]: each process receiving a message with a *csn* greater than its own will take a "forced" checkpoint [CS98, CS03]. Such a forced checkpoint is being taken for the purpose of avoiding potential inconsistency, and however might not be always necessary. Therefore a process needs to wait for a checkpoint request propagated from the initiator, before it can decide to turn a forced checkpoint into tentative or to discard otherwise. Since z-dependency is transitive but not always causal, a process p z-depending on q might not be able to know the existence of such a dependency only by comparing the *csn*'s that is causally piggybacked with application messages. Consequently, non-blocking protocols have to involve a number of extra processes into the checkpointing procedure.

Min-process checkpointing protocols take partial rather than global coordinated checkpoints, and hence have advantage in terms of minimum or reduced coordination effort as well as improved runtime performance. However, they have disadvantages at recovery time: i) z-dependency relations useful for recovery have to be completely rebuilt,

as those exploited during the checkpointing procedure cannot be reused; ii) recovery might still involve a large number of processes and there is no control of the recovery effect at all. This is because in these protocols, checkpointing and rollback-recovery are using two different kinds of z-dependency relations: the former is based on the past message interactions, while the latter is based on the future communication pattern. At runtime, there is no coordination between the groups of partial checkpoints, and hence no management of recovery dependencies. As a result, recovery spreads with the growth of message interactions. On the other hand, since checkpoints can be initiated arbitrarily, it is possible to take checkpoints at inappropriate points of execution and to cause too many processes involved in recovery unnecessarily.

2.2.4 Communication-Induced Checkpointing

Recent efforts [Tsa03, TM05, SGP+05, SG06] of the distributed checkpoint community have been mainly spent on exploiting and evaluating the idea of Communication-Induced Checkpointing (CIC). Compared with the completely autonomous independent checkpointing and totally coordinated global checkpointing, CIC can be considered as a quasi-synchronous strategy. It allows autonomous checkpoints and also forces coordinated checkpoints, in order to avoid domino effect as well as useless checkpoints. This is based on the observation that a local checkpoint becomes useless once it is involved into a z-cycle, which is a zigzag path that begins and ends with the same checkpoint [HMN+97]. Avoiding useless checkpoints requires preventing or breaking the formation of z-cycles. CIC protocols are hence developed to take forced checkpoints in addition to autonomous checkpoints before z-cycles can be actually formed.

There are two approaches in general to avoid z-cycles. One approach is to associate a timestamp function with each checkpoint. For example [BGS84], each process can manage a logical clock with its local checkpoints such that: i) successive checkpoints

have increasing clocks; ii) clocks are piggyback with messages; and iii) upon receiving a message, a process takes a forced checkpoint if the piggybacked clock is greater than its own, and also updates its own clock accordingly. Further information can be piggybacked to support more complicated protocols [HMR97a], e.g., for the purpose of reducing the number of forced checkpoints. It is clear that the set of checkpoints with the same logical clock forms a consistent state. Instead of using timestamps, the other approach is to detect all suspected communication and checkpoint patterns and to prevent them by taking forced checkpoints beforehand [BQC98]. It has been proved that these two approaches are equivalent in terms of creating the same set of forced checkpoints [HMR97b].

CIC protocols are non-blocking as the coordination information is piggybacked with application messages and hence no latency is introduced. Since CIC protocols allow certain degree of autonomy, the application can choose to do checkpointing whenever it incurs small overhead, in order to get better runtime performance. On the other hand, if autonomous checkpoints can be taken as initiation of regular coordinated checkpoints, then CIC is very similar to min-process checkpointing, as they are both induced by communications and their checkpoints do not cause global-wise coordination unnecessarily. Having the above advantages, CIC protocols pays the prices of increased application message size due to information piggybacking and checkpoint committing latency before message processing. In addition, not all z-cycles are on-the-fly trackable [BHR01], and it is impossible to add a minimum number of checkpoints for removal of all z-cycles [ABL+07]. Inevitably, each process has to keep several checkpoints on stable storage, and extra forced checkpoints are possible even sophisticated mechanisms are employed. Although CIC protocols seldom cause creation of global checkpoints, study has shown that they do not scale well for large-scale distributed systems [AER+99], as forced checkpoints are intrinsically inevitable due to spread of message interactions.

2.3 Log-Based Recovery

In log-based recovery, non-deterministic events such as message delivery are recorded as determinants [SY85] so that they can be replayed to deterministically resume the execution of a process upon failure. Checkpoints are created periodically just to trim the length of message logs and then reduce the recovery time. All log-based protocols require that upon recovery, there is no orphan process, whose state is inconsistent with the recovered process. Orphan processes are possible due to loss of message determinants. Log-based protocols vary in the way how determinants are committed into stable storage. In particular, they differ in their refinements of the always-no-orphans requirement: if the determinant of a non-deterministic event is not yet logged, its dependent processes should keep a copy of its determinant (refer to Chapter 2.1.2).

Pessimistic message logging tries to restrict the scope of dependent processes. A straightforward strategy is to synchronize the logging of each determinant, and consequently each process will block its execution upon receipt of every message [BLL89]. This atomicity condition can be relaxed by making the set of dependent processes of a message to involve its receiver only. All determinants then should be logged before the effects of their corresponding events can be seen by other processes. Corresponding protocols hence can avoid the creation of orphan processes by blocking the sending of a message m until all messages delivered before m are logged [JZ87]. Upon failure, recovery is simply rolling back the failed process to the most recent checkpoint and replaying all the messages from logs. Since logging a message may take time, pessimistic protocols can slow down the throughput of interaction-intensive applications with the overhead of logging every message and the latency of blocking the execution. Such protocols are pessimistic in the sense that they assume failure occurs after any non-deterministic event, which is in fact very rare. Alternatively, messages

could be logged at sender side and be retrieved from them for playback purpose in case of receiver failure. A sender might choose to keep messages in its volatile memory or flush them into its stable storage, with assistance of specific protocols [AAJ06].

In contrast, optimistic protocols [SY85] allow asynchronous logging, with the optimistic assumption that logging can be completed before failure occurs. Determinants are spooled into stable storage in a non-blocking manner, and as a result, orphan processes become possible in case of failure. Such orphan processes will then be detected and rolled back together to make their process states consistent. The always-no-orphan condition is not maintained all the time but is finally reached when recovery is complete. Optimistic protocols take the risk of creating temporary orphan processes to obtain better performance for the failure-free executions. Consequently, a failure recovery might involve several processes rather than just the failed one. Optimistic protocols usually employ complicated mechanism to track dependencies among processes at runtime, and to decide orphan processes at recovery time. In addition, the recovered execution might become different from the previous one, due to the rollback of orphan processes and the fact that message channels are allowed to be unreliable.

There are causal message-logging protocols [Alv96] that neither create orphan processes upon failure nor block the failure-free execution. They hence have the runtime performance of optimistic logging and meanwhile the recovery effect of pessimistic logging. These protocols properly implement the always-no-orphan condition so that, each determinant is available from either the stable storage or the volatile log of a process whose state causally succeeds the determinant's event. The logging atomicity then can be relaxed, because a determinant is always obtainable from other processes even a failure occurs before its actual commit. In fact, the delivery of a message m will not introduce an orphan process until the delivery of another message m' that causally depends on m.

Therefore, in a causal protocol, determinant of m is piggybacked with m' and is kept by m's receiver before m' is delivered. In recovery, those parts of logged information will be retrieved in order to replay m. As long as m is not logged in stable storage, it should be piggybacked and then propagated with its dependent messages, in order to tolerate process failures. The moment of output commit hence can be delayed with the cost of increased message size. Another price to pay is that causal protocols usually require complicated recovery mechanism to retrieve all logged messages.

The above three logging approaches have different focuses and applications. Pessimistic logging cares more about safety and recovery effect rather than runtime performance, and is better for applications with strict recovery requirement and frequent failure events. In contrast, optimistic logging takes the risk of having large recovery effects for the return of better performance, and hence is better for performance-oriented applications. Causal logging combines advantages from pessimistic and optimistic strategies but avoids their disadvantages, and hence is applicable in many situations. However, in a distributed system involving a large number of messages, the use of determinant piggybacking in causal logging might heavily burden the communication network. Also, since causal logging propagates determinants along with spread of message interactions, upon failure a recovery might involve a large scope of other processes to retrieve the causal information of certain messages. For message-intensive systems, it is unaffordable to use causal logging as a fault-tolerant solution.

2.4 Summary

In distributed systems, message interactions introduce recovery dependencies among processes and hence the necessity of tracking or recording them, via checkpoint coordination or message logging respectively. Since a dependency relation can only be

handled one way or the other, the above two approaches have almost contradictive effects in runtime performance and recovery effect. Checkpoint-based recovery avoids logging overhead but results in coordination latency as well as poor recoverability. Log-based recovery has better recovery control but generates excessive performance cost. These properties are directly from their ways of dependency handling and hence are intrinsic to their corresponding protocols. Since these two ways contradict each other, checkpoint- and log-based approaches form two extreme ends of rollback-recovery. As a result, a distributed system can only take advantages from one end by sacrificing those of the other. Although quite a few improvements have been proposed at both ends, it is difficult to find a protocol that features all of their advantages but none of their disadvantages.

Recently, with the dramatic increase of processor speed and network bandwidth, compared with the relatively less-improved storage access speed, checkpoint and logging overhead has become the major source of runtime performance cost. A checkpoint-based recovery strategy is hence more favorable than a log-based strategy. On the other hand, the ability to interact with the unrecoverable units (e.g. the outside world) makes logging still valuable in certain situations. However, none of these two strategies are applicable in a large scale distributed system such as a multi-agent system. Due to the large number of processes as well as messages being involved, such a system will suffer from either uncontrollable coordination and recovery effect, or unaffordable runtime overhead. Instead, a proper fault-tolerant approach for a large-scale distributed system should have performance advantages at both run time and recovery time. It is desirable to develop a hybrid scheme that incorporates both checkpoint coordination as well as message logging. Obviously this requires a meaningful revision of the two existing approaches, in particular, a major change to the way of handling different recovery dependencies. Next chapter gives detailed discussions on this issue and the corresponding technical results.

Chapter 3

Group Checkpoint and Recovery

Current large-scale distributed systems appreciate a checkpoint and recovery strategy that features both affordable runtime performance and controllable failure recoverability. This is a combination of advantages from the two approaches that have been proposed so far, namely, checkpoint- and log-based recovery. Analysis in the previous chapter shows that these approaches are based on different correctness requirements of consistency, and are using contradictive ways in handling message dependencies. As a result, it is impossible to simply use these two approaches together to get advantages from both ends.

Thoughts from another perspective give rise to a possibility of harmoniously combining these two approaches. It is clear that both approaches try to apply a unique and uniform policy of handling all message dependencies in a system. Checkpoint-based recovery manages to track all dependencies, and hence has to grow its coordination as well as recovery effort along with the spread of message interactions. Log-based recovery tries to record all message determinants, and therefore suffers from huge commit latency at runtime. More importantly, the uniformity of such policies makes it impossible to have these two approaches work together. Then an alternative is to selectively change the handling policy for certain dependencies, i.e., some message dependencies can be removed via logging, while others are tracked under checkpoint coordination.

Further analysis sheds light on a promising approach of checkpoint and recovery. In

general, to reduce the runtime overhead, the portion of messages selected for logging should be as less as possible. On the other hand, since a coordinated checkpoint can avoid logging the message interactions that occur after checkpointing, the rest portion of messages can be covered by the corresponding pre-created coordinated checkpoints. Objectively, the selective logging policy when applied should effectively separate the spread of checkpoint coordination as well as recovery. It is hence straightforward that the expected approach should be able to manage all message interactions into disjoint subsets, taking a partial coordinated checkpoint for each subset, and subsequently logging the messages between the subsets. Since each subset involves a group of processes that checkpoint and recover together, and also apply a policy of only logging messages from other groups, this approach can be named group checkpoint and recovery.

Following the above idea, the rest of this chapter presents the results from further development. Chapter 3.1 and 3.2 build the theoretical foundation of the new approach, including a generic Quasi-Atomic Recovery theory that accommodates all situations of consistency, and a Checkpoint Dependency Graph model that captures all kinds of recovery dependencies and also harmonizes different dependency handling techniques. Chapter 3.3 develops the group checkpoint strategy with details, and also demonstrates several representative checkpoint protocols. Also introduced is a new recovery technique based on process cloning, with its use in localizing group recovery. Chapter 3.4 summarizes this chapter and brings out the significance of this research. Remaining issues include how to meaningfully decide the groups and how to effectively identify them, which leads to the follow-up study on locality of message interactions in Chapter 4.

3.1 Quasi-Atomic Recovery

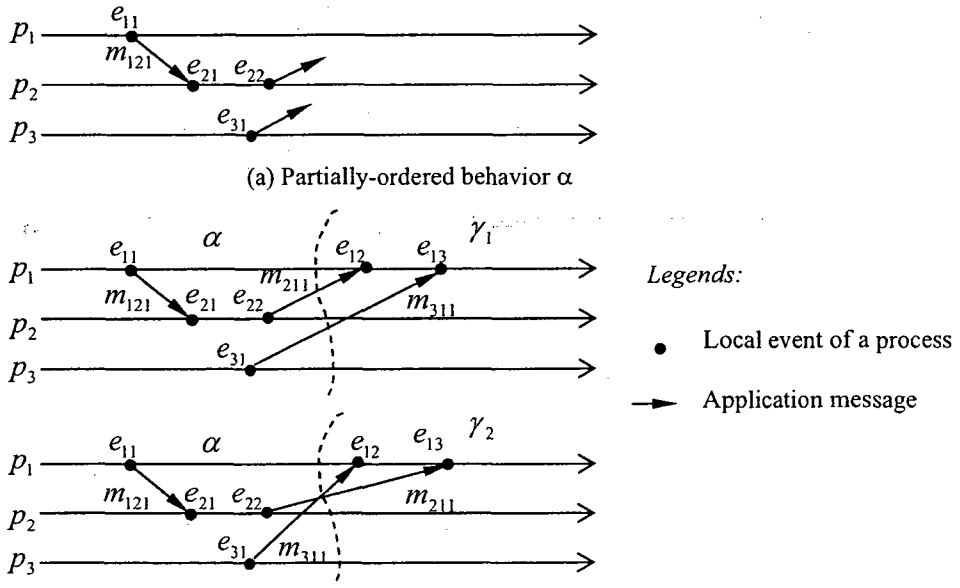
This chapter introduces a generic model that captures different correctness requirements

of recovery, including those applied to existing techniques such as deterministic as well as non-deterministic, and single as well as simultaneous recoveries. In particular, the notions of *atomic* and *quasi-atomic* recovery blocks are proposed to capture the subset of events nullified in a single recovery. It is proved that correctness of multiple recoveries is guaranteed if a recovery technique ensures well-ordering of corresponding quasi-atomic recovery blocks [LWG06]. This general correctness requirement not only accommodates both global and individual state consistencies used in checkpoint- and log-based approaches, but also makes it possible to develop new hybrid recovery strategies that combine the features of these two. The following discussion uses the traditional asynchronous model of distributed systems, and the results are applicable to any distributed system in general, including large-scale ones such as multi-agent systems.

3.1.1 System Model

A distributed system consists of a set of processes $P = \{p_i \mid 0 \leq i < n\}$ that interact with one another only through messages. Message delivery is through reliable FIFO channels and takes arbitrary but finite delay. Given a message m, *send(m)* and *recv(m)* denote m's send and receive event, from its sender and receiver process, respectively. Events of the same process form a total order, and traditionally e_{ij} denotes the j^{th} event of process p_i. All events are assumed to be instantaneous. The set of events from all processes form an irreflexive partial order under the following *happened-before* relation [Lam78]: event e happened before event e' (written as $e \rightarrow e'$) if i) e and e' belong to the same process and e occurs before e', or ii) e is the send event and e' is the receive event of the same message, or iii) there exists an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$. Given an event e_{ij} of process p_i, the set of values of its local variables immediately after the occurrence of e_{ij} defines its *local state* (denoted as s_{ij}) at that time.

A *behavior* α is a partial order $\langle E, \rightarrow \rangle$ where E is the set of events that occur during an execution of a distributed system, and $\rightarrow$ is the happened-before relation. For simplicity, in this chapter a behavior symbol α is used to denote both the corresponding partial order and its event set E. A *distributed computation* D is represented by a set of *complete behaviors* $\{B_i \mid i = 1,2,\ldots\}$, each of which corresponds to a complete and correct execution instance of D. Multiple complete behaviors in D arise when the system exhibits non-determinism, such as critical race among asynchronous receive events, or the use of random number generator. Each complete behavior in D is hence distinct. A behavior α is a *prefix* of another behavior α' (written as $\alpha \leq \alpha'$) if α is a subset of α' and α satisfies the following closure property: if $e \in \alpha$ and $e' \rightarrow e \in \alpha'$ then $e' \in \alpha$. A prefix α of a complete behavior $B \in D$ is a partial behavior of D. The *prefix closure* D^* of complete behaviors in D (i.e., $D^* = \{\alpha \mid \exists\, B \in D: \alpha \leq B\}$) contains the set of all allowable incomplete behaviors generated from D. Given $\alpha \leq \alpha' \in D^*$, $\gamma = \alpha' - \alpha$ is a *midfix* of α' (written as $\alpha' = \alpha \cdot \gamma$). Here, midfix γ is an extension for the behavior α in forming α'.



(a) Partially-ordered behavior α

(b) Two possible extensions of behavior α due to message race

Figure 3.1 An example behavior α of a distributed computation

Figure 3.1 (a) shows a behavior $\alpha = (e_{11}, e_{21}, e_{22}, e_{31})$ involving three processes.

Upon completing its execution associated with α, process p_1 can receive messages from p_2 and p_3 in either order due to the presence of message race. As is shown in Figure 3.1 (b), in one case prefix α is extended with a midfix $\gamma_1 = (e_{12}, e_{13})$ to become $\alpha_1 = \alpha \cdot \gamma_1 = (e_{11}, e_{21}, e_{22}, e_{31}, e_{12}, e_{13})$, and in the other case α is extended with a midfix $\gamma_2 = (e_{12}, e_{13})$ to become $\alpha_2 = \alpha \cdot \gamma_2 = (e_{11}, e_{21}, e_{22}, e_{31}, e_{12}, e_{13})$. Here α_1 and α_2 have the same event set but different partial ordering of these events. If the processes terminate after these events, then each of α_1 and α_2 is a complete behavior of the distributed computation D.

The (final) *global state* associated with a prefix α, denoted by $S(\alpha)$, is the set of process states and channel states upon the occurrence of all events in α. Given $\alpha' = \alpha \cdot \gamma$, the *initial global state* ${}^0S(\gamma)$ of midfix γ is also the global state associated with α (i.e., ${}^0S(\gamma) = S(\alpha)$). Upon completing the execution associated with prefix α, the processes continue with one of the possible midfixes starting from global state $S(\alpha)$, e.g., γ_1 and γ_2 in Figure 3.1 (b), hence ${}^0S(\gamma_1) = S(\alpha) = {}^0S(\gamma_2)$. A behavior γ is a midfix of D^* if there exists some behaviors $\alpha, \alpha' \in D^*$ such that γ is a midfix extension of α in forming α', i.e., $\alpha' = \alpha \cdot \gamma$. In other words, γ is an intermediate behavior that can be used to correctly extend some behavior of D^* to form a new behavior of D^*. Therefore, for a failure-free behavior α in D^*, midfix γ is a natural extension which is also failure-free. On the other hand, a behavior γ is considered as a midfix of D^* as long as it can extend α in a failure-free manner. The following *Global State Axiom* shows an essential property of a midfix in forming different behaviors with respect to D^*.

Axiom 3.1:

Given a midfix γ of D^* and a behavior $\alpha \in D^*$, if $S(\alpha) = {}^0S(\gamma)$, then $\alpha' = \alpha \cdot \gamma \in D^*$.

For a distributed computation D, the global state $S(\alpha)$ of an incomplete execution α completely defines what its future behavior (extension) can be, independent of how that

global state has been reached from the past. In particular, if γ is a midfix extension of some behavior $\lambda \neq \alpha$ then, as long as $S(\alpha) = {}^{0}S(\gamma)$, γ can also be used as an extension of α. In this case, both $\lambda \cdot \gamma$ and $\alpha \cdot \gamma$ are behaviors of D^*.

3.1.2 Proper Recovery Behavior

The term recovery behavior is used to refer to the observable behaviors of a distributed computation D in the presence of failure and recovery events. A failure-free behavior only contains events associated with the application. Assuming application transparent fault-tolerance, all recovery actions will be taken care of by a distributed recovery kernel, one per process. A *recovery behavior* of a computation D is a (failure-free) behavior of D extended with kernel events. There are three types of kernel actions used in existing recovery protocols: i) local state reset (to an earlier checkpoint state); ii) message playback [AM98, BMP03, SY85], and iii) message discard [BMP03].

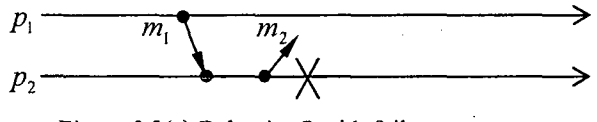


Figure 3.2(a) Behavior B with failure

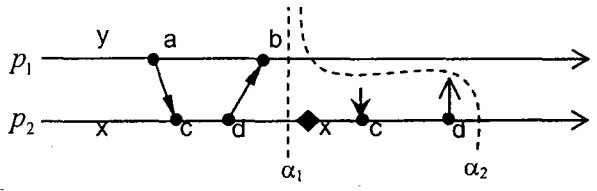


Figure 3.2(b) Behavior B_1 with recovery

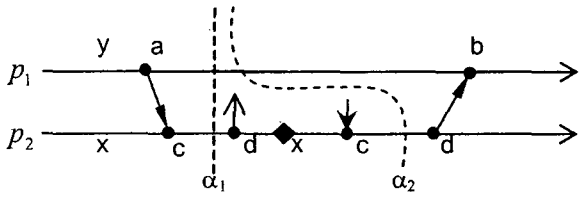


Figure 3.2(c) Behavior B_2 with recovery

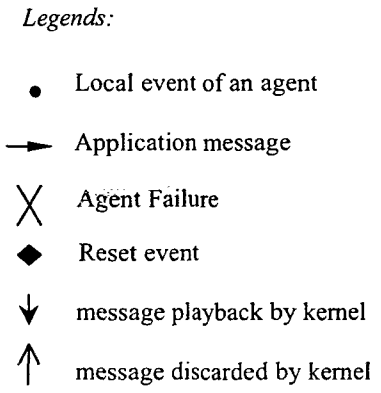


Figure 3.2 An example behavior with failure and recovery

Figure 3.2 (a) shows an example behavior B involving two processes p_1 and p_2 and a single failure with p_2. Message m_1 is received but m_2 is not. For simplicity, the local states of the processes are labeled (as a, b, c, x, etc) right after the corresponding local events.

Two possible recovery behaviors (B_1 and B_2) may occur, as shown in Figure 3.2 (b) and Figure 3.2 (c) respectively. In Figure 3.2 (b), process p_2 recovers independently from a previous local state x using log-based strategies [SY85, AM98]. Message m_1 is replayed so that p_2 can reach its local state d just prior to reset, and m_2 is considered as an orphan message and is discarded by p_1. Figure 3.2 (c) shows another possibility of recovery where message m_2 is discarded by (the kernel of) p_2. This is possible when the failure of p_2 is detected before m_2 is actually sent out.

Inclusion of kernel-level recovery events in a recovery behavior needs to satisfy the following requirements towards a correct recovery:

Definition 3.1 (Proper Recovery Behavior):

A recovery behavior $\hat{\alpha}$ is *proper* if $\hat{\alpha} = \alpha_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \ldots \cdot \gamma_{2i} \cdot \gamma_{2i+1} \cdot \ldots \cdot \gamma_{2k} \cdot \gamma_{2k+1}$ for some $k \geq 1$ such that:

a. $S(\alpha_1) = S(\alpha_2)$, $S(\alpha_3) = S(\alpha_4)$, ..., $S(\alpha_{2i-1}) = S(\alpha_{2i})$, ..., $S(\alpha_{2k-1}) = S(\alpha_{2k})$, where $\alpha_i = \alpha_1 \cdot \gamma_2 \cdot \ldots \cdot \gamma_i$, for $i = 2, 3, \ldots, 2k$;

b. $\alpha_1 \in D^*$, and

c. $\gamma_3, \gamma_5, \ldots, \gamma_{2i+1}, \ldots, \gamma_{2k+1}$ are midfixes of D^* and hence kernel events are not contained in these midfixes.

In other words, in a proper recovery behavior $\hat{\alpha}$, kernel events are contained only in isolated midfixes ($\gamma_2, \gamma_4, \ldots, \gamma_{2i}, \ldots, \gamma_{2k}$). A process behavior extended with such a midfix is finally brought to its initial state just prior to the midfix. Hence such a midfix serves to correctly recover a failure. In the previous example, both recovery behaviors in Figure 3.2 (b) and Figure 3.2 (c) are proper, as it is easy to check that $S(\alpha_1) = S(\alpha_2)$ in both the behaviors and all recovery events are contained in $\gamma_2 = \alpha_2 - \alpha_1$ only. The following *Midfix Deletion Lemma* captures the implications of a proper recovery behavior.

Lemma 3.2: Given a proper recovery behavior $\hat{\alpha} = \alpha_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \ldots \cdot \gamma_{2i} \cdot \gamma_{2i+1} \cdot \ldots \cdot \gamma_{2k} \cdot \gamma_{2k+1}$, $\alpha = \alpha_1 \cdot \gamma_3 \cdot \gamma_5 \cdot \ldots \cdot \gamma_{2i+1} \cdot \ldots \cdot \gamma_{2k+1}$ is a behavior of D^*.

Proof: The proof is by construction, first to show that $\alpha_1 \cdot \gamma_3$ is a behavior of D^*.

(i) $\alpha_1 \in D^*$. This is directly from condition (b) of proper recovery behavior;

(ii) $\alpha_1 \cdot \gamma_3 \in D^*$. Since $S(\alpha_1) = S(\alpha_2) = {}^0S(\gamma_3)$ and γ_3 is a midfix of D^*, by following the global state axiom discussed previously, γ_3 is a failure-free extension of α_1 in D^*,i.e., $\alpha_1 \cdot \gamma_3$ is a behavior of D^*.

The above construction can be repeated for $\gamma_3 \cdot \gamma_5 \cdot \ldots \cdot \gamma_{2i+1} \cdot \ldots \cdot \gamma_{2k+1}$ and hence the claim holds. □

Lemma 3.2 establishes an important relationship between a proper recovery behavior and a failure-free behavior of D^*. In particular, by 'ignoring' midfixes that contain recovery events as if they are stuttering events, the behavior that remains is a behavior of D^* as if the failures have never occurred. Hence, a proper recovery behavior can be mapped into a correct behavior in D^*. This correctness requirement follows from the set of allowable behaviors in a distributed computation D, and shows that a proper (correct) recovery behavior should consist of special midfixes whose removal leaves behind a correct behavior of D^*. It is a general model that captures both deterministic (e.g., log-based) and non-deterministic (e.g., coordinated checkpoint based) recovery strategies by allowing recovery actions to include playback and discard of messages. Also, it allows multiple crash recoveries that may or may not overlap throughout the system's lifetime.

3.1.3 Quasi-Atomic Recovery Block

A normal execution usually contains midfixes that are exclusively localized to a subset of processes, e.g. process conversations [Ran75] and coordinated atomic actions [XBR+95]. These midfixes are 'independent' in the sense that events inside such a midfix have no effect on the advancement of those processes not involved in and vice versa. Similarly, in

a recovery behavior, special midfixes containing recovery events can have the same independence property and consequently isolate the effects of failure recovery, as follows:

Definition 3.2 (Quasi-Atomic Recovery Block):

A midfix β of a recovery behavior $\hat{\alpha}$ forms a *quasi-atomic recovery block* iff

(i) $send(m) \in \beta \Rightarrow$ either $recv(m) \in \beta$ or m is discarded by the kernel, and

(ii) $recv(m) \in \beta \Rightarrow$ either $send(m) \in \beta$ or m is replayed by the kernel, and

(iii) β contains at least one reset event from each process involved in β, and

(ii) the initial state of each process before β is identical to its state after β.

In addition, if for any message m, $send(m) \in \beta \Leftrightarrow recv(m) \in \beta$, i.e., there is no message playback or discard, β is called an *atomic recovery block.*

Quasi-atomicity ensures independent advancements of the events in β in presence of failure and recovery events. Its difference from atomicity lies in the existence of special message handling actions that are required in recovery. Hence an atomic recovery block satisfies quasi-atomicity but not vice versa. As immediate examples, the midfixes shown in Figure 3.2 (b) and Figure 3.2 (c) corresponding to $\alpha_2 - \alpha_1$ are quasi-atomic. The notion of quasi-atomic recovery block is to capture a midfix that can be deleted from a proper recovery behavior as if the events in this midfix have never occurred (refer to Lemma 3.2). Since the processes are brought back to their initial states right before the block, all events inside such a block can be considered as if they never happened.

A midfix β is *ordered before* another non-overlapping midfix β' in the same behavior (written $\beta \rightarrow \beta'$) iff

(i) an event in β happened before some event in β', or

(ii) there exists another midfix β'' such that $\beta \rightarrow \beta''$ and $\beta'' \rightarrow \beta'$.

A set $\bar{E}$ of midfixes is *well-ordered* if $\forall\ \gamma_i, \gamma_j \in \bar{E} : (\gamma_i \rightarrow \gamma_j) \Rightarrow \neg(\gamma_j \rightarrow \gamma_i)$. In other words, the ordering is anti-symmetric. Two midfixes γ_i, γ_j are *un-ordered* if $\neg(\gamma_i \rightarrow \gamma_j) \wedge$

$\neg(\gamma_j \rightarrow \gamma_i)$. The next theorem shows that well-ordered quasi-atomic recovery blocks are related to proper recovery.

Theorem 3.3: A recovery behavior $\hat{\alpha}$ is proper if its kernel events are contained in well-ordered quasi-atomic recovery blocks.

Proof: Given k well-ordered quasi-atomic recovery blocks in $\hat{\alpha}$, from well ordering requirements, these blocks can be labeled as $\gamma_2, \gamma_4, \ldots, \gamma_{2i}, \ldots, \gamma_{2k}$ such that $i > j \Rightarrow \neg(\gamma_{2i} \rightarrow \gamma_{2j})$. We prove that $\hat{\alpha}$ can be written as $\hat{\alpha} = \alpha_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \ldots \cdot \gamma_{2i} \cdot \ldots \cdot \gamma_{2k} \cdot \gamma_{2k+1}$. Consider α_1 to be the set of events that are not in γ_2 but that happen before some event in γ_2, i.e., $\alpha_1 = \{e \mid \exists\, e' \in \gamma_2 : e \rightarrow e' \wedge e \notin \gamma_2\}$. Consequently, both α_1 and $\alpha_2 = \alpha_1 \cdot \gamma_2$ are prefixes of $\hat{\alpha}$. Now consider $\alpha_3 = \{e \mid \exists\, e' \in \gamma_4 : e \rightarrow e' \wedge e \notin \gamma_4\}$, and $\gamma_3 = \alpha_3 - \alpha_2$. Then $\alpha_3 = \alpha_2 \cdot \gamma_3$, and $\alpha_4 = \alpha_3 \cdot \gamma_4$ are prefixes of $\hat{\alpha}$. By repeating the above construction, we arrive at $\alpha_{2k} = \alpha_{2k-1} \cdot \gamma_{2k}$ as a prefix of $\hat{\alpha}$. Finally, $\hat{\alpha}$ can be constructed as $\hat{\alpha} = \alpha_{2k+1} = \alpha_{2k} \cdot \gamma_{2k+1} = \alpha_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \ldots \cdot \gamma_{2i} \cdot \ldots \cdot \gamma_{2k} \cdot \gamma_{2k+1}$.

By quasi-atomicity, all kernel events are contained in $\gamma_2, \gamma_4, \ldots, \gamma_{2i}, \ldots, \gamma_{2k}$. Therefore α_1 is a failure-free behavior of D^*. Moreover, each of the recovery blocks has identical initial and final global state, i.e., $S(\alpha_1) = S(\alpha 2), S(\alpha 3) = S(\alpha 4)$, etc. Hence the claim holds. □

The implication of the above theorem is very general and is applicable to any distributed system. Suppose a checkpoint-recovery protocol ensures that the resulting recovery behavior satisfies the condition in Theorem 3.3. From Lemma 3.2, the recovery behavior 'embeds' a valid behavior in D^*, and consequently will deliver the same results as the latter. For this to hold, application results should be delivered upon termination of

the execution (i.e., as if in a closed system). If the application involves interaction with the outside world process (OWP) [SY85], it is required that these interactions should not occur in the midfixes γ_{2i} that are deleted in forming a behavior in D^*. However, for simplicity, such an extension involving OWP has been omitted.

The quasi-atomic recovery model is generic and it accommodates both deterministic and non-deterministic recoveries [SY85, JZ88]. In the former case, there is an implicit quasi-atomic recovery block formed by the portion of events between the maximum recoverable system state and the after-recovery system state, which serve as its identical initial and final states. On the other hand, piecewise deterministic (PWD) is not enforced in this model, and hence an after-recovery execution could be different from the one before recovery. As a result, this model is able to capture non-deterministic recovery behaviors resulted from coordinated checkpointing strategies.

In addition, the notion of quasi-atomic recovery block models the portion of process behaviors being nullified during a single recovery. It can be easily applied to capture the consistency requirements of existing strategies. For example, global checkpoints [TS84, CL85] form a consistent recovery line, which corresponds to having a single quasi-atomic recovery block that ends with a reset event in every process. For min-process and CIC protocols [KT87, CS98, AER+99], the resulted quasi-atomic recovery block involves a subset of processes that interact via messages after their last checkpoints. In log-based recovery [SY85], a quasi-atomic recovery block contains a reset event in each recovered process, followed by a sequence of message events that are replayed deterministically. The property of quasi-atomicity is very general and it covers all kinds of available checkpoint and recovery techniques, including partial or complete recovery lines, consistent or inconsistent recovery lines, message playback and discard, etc. This makes it possible to develop new strategies that feature different combinations of these

techniques with more desirable properties. For example, the group checkpoint and recovery strategy introduced in Chapter 3.3 can recover well from a partial inconsistent recovery line assisted by message playback and discard, which is impossible for traditional checkpoint- or log-based recoveries.

3.2 Checkpoint Dependency and Recovery

As checkpoints are taken in a distributed execution, recovery dependencies between checkpoints also arise. Such dependencies can be caused by program order between two checkpoints or message interactions between the program fragments that follow two checkpoints. Dependency models have been proposed in literatures from different perspectives. Rollback-dependency graph [BL88] and checkpoint graph [Wan93] are built to trace dependencies among uncoordinated checkpoints in order to determine a recovery line. Z-path and Z-cycle [NX95] are formulated to capture special checkpoint-message patterns, and are also useful in avoiding useless checkpoints in communication-induced checkpointing [AER+99]. These models capture particular characteristics associated with the corresponding checkpoint strategies. This chapter presents a generic model named Checkpoint Dependency Graph (CDG) [WLG06] that aims at capturing the essential recovery requirements applicable to all existing strategies, as well as the new group checkpoint strategy to be introduced in the next chapter.

3.2.1 Checkpoint Dependency Graph

Common to all checkpoint protocols, local states of processes are saved as local checkpoints. A local state keeps the minimal information by which a process can recover upon failure via resetting to that state. Hence, a local checkpoint containing only the local state is a *minimal checkpoint*. However, a process checkpoint often contains more information than just the local state, such as a channel state in the case of coordinated

checkpointing and a message log in the case of log-based recovery. From this perspective, a minimal checkpoint contains the minimal information retained in a checkpoint, independent of the details of the checkpoint protocol. Symbolically, the k^{th} checkpoint taken by process p_i will be denoted as c_{ik}. In addition, the behavior associated with process p_i is partitioned into fragments due to the creation of minimal checkpoints. There is a distinct *behavioral fragment* B_{ik} associated with each checkpoint c_{ik}. Contained in B_{ik} are the events that occur after c_{ik} but before (i) the creation of the next checkpoint c_{ik+1}, or (ii) occurrence of a process crash of p_i, or (iii) proper termination of p_i.

Recovery dependency can be formed among minimal checkpoints at runtime, due to either local program order or message interactions. Sequential execution of each process p_i causes a *program-order dependency* between any two successive checkpoints c_{ik} and c_{ik+1} (written as $c_{ik} \rightarrow c_{ik+1}$). As a result, when p_i is reset to recover B_{ik}, B_{ik+1} will also be recovered. On the other hand, a message m sent in B_{ik} and received in $B_{jk'}$ creates a symmetrical *message dependency* between c_{ik} and $c_{jk'}$ (written as $c_{ik} \leftrightarrow c_{jk'}$). Recovery of B_{ik} requires recovery of $B_{jk'}$ so that message m can be properly received again by p_j as a recovery partner. Similarly, recovery of $B_{jk'}$ requires recovery of B_{ik} so that the message m can be reproduced to be received by the recovered p_j.

A *baseline Checkpoint Dependency Graph* (baseline CDG) can be constructed using both kinds of dependency relations among minimal checkpoints. As an example, Figure 3.3 (a) and Figure 3.3 (b) show an execution involving two processes and the corresponding baseline CDG respectively. Since recovery dependency is transitive, when a process crashes and recovers at its most recent checkpoint, say c_{ik}, it could induce other processes to recover with it. In a baseline CDG, reachability from a checkpoint c_{ik} identifies the set of other checkpoints (and hence the corresponding behavioral fragments) that have to recover with c_{ik}. This set is called the *reachable component* of c_{ik}. A baseline

CDG captures all recovery dependency relations established without any message logging, e.g. in case of uncoordinated checkpointing. Similar to existing dependency graph models [BL88, Wan93], it can be used to analyze domino effect [Ran75] and determine the recovery line among uncoordinated checkpoints.

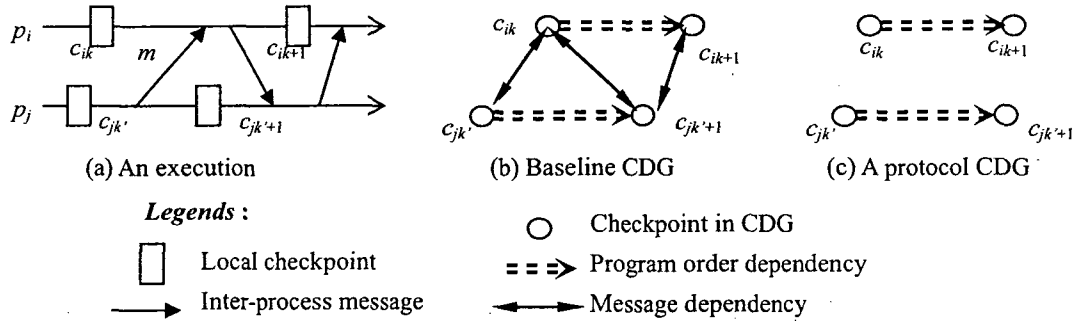


Figure 3.3 Checkpoint Dependency Graph for a distributed execution

In order to reduce dependency edges and to control recovery spread, checkpoint protocols usually record more information about behavioral fragments. For example, in log-based recovery [SY85], all messages and event determinants are logged for deterministic recovery. As shown in Figure 3.3 (c), in such a case the resulting CDG has no message dependency edges, i.e. they are effectively removed due to logging of corresponding messages. Consequently for a particular checkpoint and logging protocol, there is a resulted specific *protocol CDG*, $G = <C, D>$, that captures the remaining dependency relations after the removal of some of the dependency edges. Here, C is the set of checkpoints and D is the set of dependency edges. In a protocol CDG, a (local) *checkpoint* c_{ik} is represented as a three-tuple, in order to capture relevant information saved at runtime for the recovery of B_{ik}. Formally, $c_{ik} = <S_{ik}, L_{ik}, N_{ik}>$ where

S_{ik} = local process state recorded in checkpoint c_{ik},

L_{ik} = a log containing a subset of messages received in B_{ik}, and

N_{ik} = a sequence of determinants associated with B_{ik}.

The set of dependency edges D in a protocol CDG is a subset of those in the

corresponding baseline CDG resulted from selectively removing some of the message dependency edges (i.e., via message logging). For example, coordinated checkpoint protocols only remove the edges corresponding to in-transit messages across a global checkpoint, while in log-based recovery all message dependency edges should be effectively removed. For simplicity, the rest of this thesis will use the term CDG to refer to protocol CDG.

A protocol CDG $G = <C, D>$ is *proper* if it satisfies the following requirements:

(a) D includes all program-order dependency edges among checkpoints in C.

(b) If message interaction exists between B_{ik} *and* $B_{jk'}$, then either $c_{ik} \leftrightarrow c_{jk'}$ is included in D, or otherwise every such message is logged in L_{ik} or $L_{jk'}$.

(c) All event determinants are recorded.

The log set can be implemented at either the sender or the receiver side. For ease of presentation, this thesis assumes receiver-side logging. In addition, it is assumed that all event determinants are identifiable and can be logged.

Properness of a CDG captures the consistency requirement between removing message dependencies and recording runtime information. In other words, necessary information (i.e. messages and event determinants) must be recorded in order that reachability in a proper CDG can be used to identify the set of checkpoints required for a correct recovery. In addition, both information recording and dependency removing can be done at run time by the checkpoint protocol. From this perspective, once a proper CDG is generated by a checkpoint protocol, it can be used for correct recovery by identifying the reachable component from a failed process's behavior fragment. This allows separating the design of specific checkpoint protocol from a generic recovery protocol that can be used independently. In fact, it will be proved next that as long as the CDG is proper, the generic recovery protocol can guarantee correctness of process crash

recovery, without knowing any details of the checkpoint protocol.

Logging all messages is expensive and may incur unaffordable delay to the underlying execution. Properness of a CDG provides the possibility of optimizing logging in checkpoint protocols. In particular, by selectively removing some message dependency edges, it is possible to trade off the runtime overhead caused by logging and the latency caused by recovery spread.

3.2.2 Recovery with CDG

Usually a process stops execution as soon as it crashes [SS83]. Recovery protocols are designed to recover such process failures from saved runtime information such as checkpoints and message logs. In general, a recovery is considered as successful if the system can be restored back to normal as if there is no failure occurred. This relates a correct recovery to a quasi-atomic recovery block (see Chapter 3.1.3).

Definition 3.3 (Correct Recovery of Process Crash):
A crash recovery is *correct* if the recovery leads to a quasi-atomic recovery block in the corresponding execution behavior.

A quasi-atomic recovery block isolates its recovery behaviors from outside processes. Correctness is guaranteed by ensuring the pre- and post-recovery local states of involved processes to be identical. As an example, in Figure 3.4, the crash of p_i as indicated by X triggers a recovery of p_i and p_j, in particular, their corresponding behavioral fragments B_{ik}, $B_{jk'}$ and $B_{jk'+1}$. However, neither p_m nor p_n is involved in the recovery, as the message dependencies created by m_3 and m_4 are decoupled through message discard and replay, respectively. p_m and p_n advance their behaviors forward asynchronously without being affected by the events in the shaded quasi-atomic recovery block. Upon completing the events in this block, the recovered states of p_i and p_j can be part of a global state (cut) that

depends on the asynchronous advancements of p_m and p_n, somewhere bounded between the global cut 1 and global cut 2 (which limits the extent that p_m and p_n can advance without p_i and p_j).

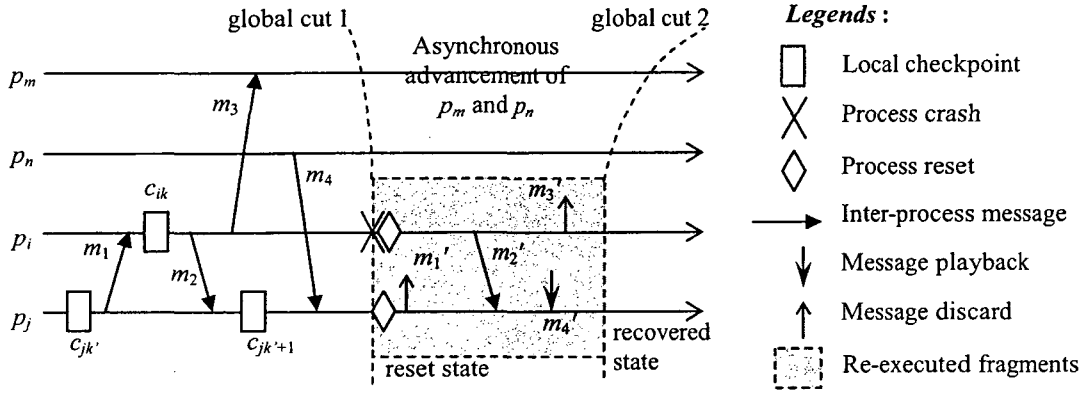


Figure 3.4 Deterministic recovery of behavior fragments

In the above example, both p_i and p_j re-execute a set of behavioral fragments during recovery. If this re-execution is deterministic, p_i and p_j will reach a recovered state that is the same as the reset state reached before the recovery action. This is an instance of deterministic recovery, which can be defined as follows.

Definition 3.4 (Deterministic Recovery of Behavioral Fragments):
A set of behavioral fragments S is deterministically recovered if for each fragment $B \in S$, the following properties are guaranteed in its re-executed fragment B' after its process being reset:

(a) The local state at the beginning of B' is the same as in the original fragment B.

(b) The same sequence of events occurs in B' as in the original fragment B.

(c) The local state of B' after each event is the same as in the original fragment B.

This definition captures the minimal abstract requirements for re-executing a behavior fragment during recovery. Unlike non-deterministic recovery that requires the system to roll back to a consistent global checkpoint, deterministic recovery usually

works with message logging but involves fewer processes. In addition, it is always necessary when there is an un-recoverable outside world process (OWP) [SY85]. To have a localized recovery effect, in which case the set of all other processes not being recovered is acting as an OWP, deterministic recovery should be employed.

Theorem 3.4: A process crash can be recovered correctly by the deterministic recovery of a set of behavioral fragments S provided:

(i) S includes the most recent behavioral fragment of the crashed process;

(ii) If a fragment B_{ik} is in S, then all subsequent fragments $B_{ik'}$ $(k' > k)$ are in S, and

(iii) Every message re-sent in a re-executed fragment in S is discarded if the original message was received in a fragment not in S.

Proof: From (i) and Definition 3.4, the crashed process upon recovery will reach its state right before the crash. For each process involved in recovery, from (ii) and the property of deterministic recovery, that process will also reach its state right before recovery. Hence in the recovery block formed by the re-executed fragments, all involved processes have the identical pre- and post-recovery states. From (b) of Definition 3.4, all events will be re-executed in this recovery block, which implies that for any message m involved, i) if m is contained in S, then both *send*(m) and *recv*(m) will be re-executed; ii) if only *recv*(m) is included in S, it will be re-executed via replaying of message m; iii) if only *send*(m) is included in S, (iii) ensures that m will be re-sent and then discarded. The recovery block is hence quasi-atomic. From Definition 3.3, the recovery is correct. □

Deterministic recovery can be implemented through a generic recovery protocol,

using the information recorded at run time. Given a CDG $G = <C, D>$ maintained by some checkpoint (and logging) protocol, and a set of behavioral fragments S with the corresponding checkpoints in C, the generic recovery protocol performs recovery of the behavioral fragments in S as follows:

i) Re-execute each process involved in S from its earliest fragment included in S.

ii) The re-execution is performed by applying the following rules:

(a) Before re-execution, a recovering process p_i will check each behavioral fragment B_{ik} included in S, and will remove from the corresponding L_{ik} all the message logs that are sent from a behavioral fragment $B_{jk'}$ that is also included in S.

(b) When p_i re-executes a message receive in a fragment B_{ik} in S, it will essentially remove the corresponding message involving that channel from L_{ik}, provided that such a message exists in the log, and will replay the message.

(c) When a message m re-sent from a re-executed fragment B_{ik} is received, the receiver, say p_j, will discard this message if it appears in some $L_{jk'}$.

(d) In order to ensure deterministic recovery of every event in B_{ik}, determinants recorded in N_{ik} of the corresponding c_{ik} will be applied accordingly.

For the example in Figure 3.4, suppose that all messages are recorded in the corresponding log sets, and S is the set containing c_{ik}, $c_{jk'}$ and $c_{jk'+1}$. Before re-execution, the generic recovery protocol will first remove m_2 from $L_{jk'}$, while m_1 and m_4 remain in L_{ik-1} and $L_{jk'+1}$ respectively, and L_{ik} and $L_{jk'}$ are empty. During the re-execution, m_1 is re-sent as m_1' and is then discarded; m_2 is re-sent as m_2' and finally is received by p_j. Since p_m is not involved in recovery, m_3 is in a log set of p_m. Therefore, when m_3 is re-sent as m_3' to p_m, m_3' will be discarded. When p_j re-executes a receive event regarding m_4, it will remove m_4 from $L_{jk'+1}$ and replay m_4 as m_4'.

Corollary 3.5: Given a proper CDG and the most recent checkpoint c_{ik} of a crashed

process p_i, the generic recovery protocol performs correct recovery using the set S of behavioral fragments associated with the reachable component of c_{ik}.

Proof: Conditions (i) and (ii) of Theorem 3.4 are satisfied by the use of the reachable component of c_{ik} in generating S. Rules (a), (b) and (d) in the generic recovery protocol ensure deterministic recovery of all the behavioral fragments in S, while rule (c) ensures condition (iii) of Theorem 3.4. □

Corollary 3.6: The generic recovery protocol can properly recover simultaneous crashes.

Proof: Suppose there are multiple processes p_i, p_j, ..., p_n that crashes simultaneously. Given the reachable components of their most recent checkpoint c_{ik}, c_{jk}, ..., c_{nk}, from Corollary 3.5, the generic recovery protocol can individually perform a correct recovery for each of them if they are disjoint from each other, leading to well-ordered quasi-atomic recovery blocks. If a behavioral fragment, say B_{ik}, is involved in multiple reachable components, from rule (a) of the generic recovery protocol, the corresponding process p_i will remove from L_{ik} the messages that are sent and received between those overlapping reachable components, and as a result, their recoveries will be effectively merged into one. From Theorem 3.3, the simultaneous crashes can be recovered properly. □

The generic recovery protocol is checkpoint protocol independent. Correctness of crash recovery is guaranteed as long as the checkpoint protocol creates a proper CDG. Hence correctness of a checkpoint protocol can be separately addressed by checking if the resulting CDG is proper.

Definition 3.5 (Correctness of a Checkpoint Protocol):

A checkpoint protocol is *correct* iff it always generates a proper CDG.

CDG also plays an important role in garbage collection of unusable checkpoints. In general, as checkpoints accumulate in stable storage, some become no longer usable and hence need to be discarded from time to time. A local checkpoint is safe to be discarded only if it will no longer be used for the recovery of any process failure. Obviously the most recent checkpoint is un-discard-able, so are the checkpoints in their reachable components. In principle, a checkpoint that is not reachable from any most recent checkpoint can be discarded by garbage collection. Consequently, given a CDG $G = <C, D>$ maintained by some checkpoint and logging protocol, a generic garbage collection protocol can be designed to locate the set of most recent checkpoints with their reachable components, and then to discard unusable checkpoints from time to time. Similarly, it is independent of any checkpoint as well as recovery protocol. Unlike the generic recovery protocol that is triggered by a process crash, garbage collection is triggered by creation of new checkpoints, usually upon completion of a (coordinated) checkpoint.

3.3 Group Checkpointing

Both checkpoint- and log-based recovery strategies suffer from inevitable disadvantages, which contradict each other's advantages and also restrict their applicability. A natural alternative is to develop a hybrid strategy that takes advantage of both the approaches in a harmonious manner. Essentially, only a subset of processes is expected to be involved during checkpoint as well as recovery, and only a small portion of messages is expected to be logged at run time. This is particularly important in large-scale and possibly geographically distributed systems, in order to localize a crash recovery to a small sub-

system of the entire system without sacrificing too much runtime performance.

In the literature, there are various protocols proposed with similar checkpoint or recovery effect, but none of their recoveries is controllable. For example, min-process checkpoint protocols [KT87, CS98] manage a partial coordinated checkpoint only among processes that have message interactions after their last checkpoints. The checkpoint coordination is targeted as 'minimal' but the recovery depends on the unknown future communications. CIC protocols [AER+99] employ a similar mechanism and have the same recovery effect. Optimistic logging [SY85] can involve an unpredictable number of 'orphan' processes into recovery, which is completely a by-chance result. All these strategies fail because they are not designed for such a purpose, and are not resulted from recovery dependency analysis as well.

Based on the theoretical results presented previously, a group checkpointing approach is proposed and developed in this chapter, with the target of addressing the above issues. In principle, this approach employs a selective policy in dependency handling, making it possible to have both coordinated checkpoint and message logging at the same time. A group or subset of processes can take a coordinated checkpoint, and then only log messages from other groups. The resulted partial checkpoint is called a *group checkpoint*, which differs from a global checkpoint and a local (individual) checkpoint. To facilitate logging, each group can be assigned a unique color and messages sent by a group can be colored accordingly. Coordinated group checkpoints avoid logging inside each group, while logging removes inter-group recovery dependencies. Both logging and recovery effect hence can be effectively controlled.

This chapter starts with an example group checkpoint protocol that is based on specific agent communication structures, followed by a discussion about general group checkpointing and the related message coloring mechanism. Group formation can be

atomic or non-atomic, with different focuses on performance and flexibility. Similar to logging that removes message dependencies, process cloning can be used to effectively remove program-order dependencies. Based on this technique, results for recovery with bounded size are developed and protocols are also presented.

3.3.1 An Example Group Checkpoint Protocol

One important objective of the target hybrid strategy in this chapter is to improve the runtime performance, in particular, to avoid logging as much as possible. The ultimate choice is hence to not log any message at all, which in turn also avoids message playback during recovery. The resulted checkpoints are called 'strongly consistent' [HNR99], i.e., there is no orphan or in-transit message across the corresponding recovery line. Also, checkpoints are better taken in a non-blocking manner, as the underlying failure-free execution hence will not be slowed down. The other objective is to localize the effect of checkpoint coordination as well as recovery, so that it only involves a small subset of processes rather than the entire universe. A candidate solution is to carefully manage a partial coordinated checkpoint around a group of processes, which are coupled with each other via message interactions and will recover together upon the failure of each other.

Among all the above requirements, the property of strong consistency is critical: the positions of checkpointing should be carefully selected; or otherwise the failure-free execution has to be blocked. In fact, real-life distributed systems such as multi-agent applications often consist of special communication structures that can be made use of. Agents are role-based [Ken98, SE05] and they perform their tasks via agent protocol sessions [BMO01, OMG03]. At design time, roles specify task decomposition and collaboration among agents, and are usually implemented as a set of agent protocols. An agent can play multiple roles at different times. As a result, message interactions among agents form agent protocol sessions. For a period of time, an agent usually interacts only

with a subset of others, which have lifelines that are coupled together, forming a localized sub-region in space and time. In general, such localization of agent interactions are often around the following units: i) a single agent protocol session, ii) a subset of concurrent protocol sessions that merge due to multiple roles played by a same agent, and iii) a subset of protocol sessions exercised causally after a particular protocol session. Agents within such a unit interact exclusively with each other and there is no message between units. Consequently a strong group checkpoint can be managed around these units. For example, before sending or receiving any message in such a unit, the local state of each participant agent can be saved for future use. Due to the exclusiveness of agent interactions, the corresponding group checkpoint is strong and there is no necessity to log any message.

Another requirement is the asynchrony of checkpointing, i.e., checkpoint creation should not block the failure-free execution. Fortunately agent protocols are well-structured by design, which helps to achieve the non-blocking feature of agent group checkpointing. More specifically, in an agent protocol there is usually a special agent acting as the session initiator, and a chain of agent messages causally passing from the initiator to each member agent and eventually going back to the initiator. Each member agent starts its participation into the session when it is involved into such a 'message chain'. A protocol session is started when the initiator sends its first message in the session, and is ended when the initiator has completed all its events in the session. As message chain is widely used in agent application design as well as agent protocol programming, the above property is very popular in many FIPA protocols [FIP99]. This in turn enables embedding of checkpoint coordination within normal application messages, and thus eliminates the necessity of explicit synchronization between agents. It is easy to prove that by piggybacking the membership information onto application

messages, eventually the initiator and each participating member will be aware of each other. Coordination of group checkpointing hence can be managed asynchronously.

This chapter presents a simple non-blocking protocol that creates strong group checkpoint for each agent protocol session. For each agent, the checkpoint creation time is right before its participation into a session. Overlapped protocol sessions have message interactions between each other, and hence need to checkpoint together. In other words, agents participating in these sessions form a single group, and only one group checkpoint is created for such a group. This in turn implies that each agent only needs to take a checkpoint when it has completed its events of all previous sessions and is ready to start a new one. At runtime, message piggybacking is used to identify overlapping of protocol sessions and to manage recording of related information for recovery. Given below is an informal description of this *Agent Strong Group Checkpoint* (ASGC) protocol.

Group checkpoint initiation: A group checkpoint can be initiated by an initiator agent a_i of a protocol session P_i, provided that it is not currently involved in another session P_j that overlaps P_i. To start the initiation, a_i simply takes a local checkpoint before participating in P_i.

Group construction: Once associated with a group, each member agent will propagate group membership information to other members. Such group information is also saved with the corresponding local checkpoint for later use during recovery.

Checkpoint coordination: For each member agent receiving an application message of a protocol session for the first time, if it is not currently involved in any other protocol session, it will first take a local checkpoint corresponding to the group before the actual delivery of the message. Otherwise, it will simply associate the current local checkpoint with the new group.

A more detailed description is shown as follows.

Data structure for an agent a_i :

inc	= current incarnation number, initialized to 0, to be used during recovery;
current	= current local checkpoint number, initialized to 0;
group_id	= ⟨initiator id, *current* checkpoint number of initiator⟩, the identifier for a protocol session;
member (*group_id*)	= set of participating agents in *group_id*. Initialized to empty set;

receivers (*group_id*) = set of agents in *group_id* to whom a_i has sent an application message;

label (*current*) = set of protocol sessions with which the current local checkpoint is associated;

Actions for an initiator a_i on starting a protocol session P_i:

if a_i has completed all previous sessions **or** P_i is the first session of a_i **then**

begin

Save local data structure with the current checkpoint;

current = *current* + 1; Create a new local checkpoint;

group_id = ⟨*i*, *current*⟩; member (*group_id*) = {*i*}; label(*current*) = {};

end

group_id = ⟨*i*, *current*⟩;

label(*current*) = label(*current*) ∪ {*group_id*};

Actions for an agent a_i on sending an application message m pertaining to protocol session P_k to agent a_j:

Piggyback message *m* with ⟨*group_id*, member(*group_id*), *inc*⟩;

receivers (*group_id*) = receivers (*group_id*) ∪ {*j*};

Actions for an agent a_i on receiving a message m piggybacked with ⟨*group_id'*, *member'*, *inc'*⟩:

if this is the first time receiving a message with *group_id'* and a_i has completed all previous sessions **then**

begin

Save local data structure with the current checkpoint; *current* = *current* + 1;

Create a new local checkpoint before the actual delivery of message *m*;

label (*current*) = {*group_id'*};

end

else label (*current*) = label (*current*) ∪ {*group_id'*};

member (*group_id'*) = member (*group_id'*) ∪ {*i*} ∪ *member'*.

Figure 3.5 shows an example of the ASGC protocol. There are altogether four agent protocol sessions, identified as P_1 to P_4. P_1 overlaps with P_2 and P_3. From the protocol, they are expected to form two strong group checkpoints: one before the overlapped sessions P_1, P_2 and P_3; and the other before P_4. The initiator of each protocol session is the agent that sends the first application message in the session. As the initiator of P_1, agent a_3 creates checkpoint c_{31} and labels it with {⟨3, 1⟩} to reflect that it is associated with a group checkpoint identified as ⟨3, 1⟩. As the protocol session progresses,

checkpoints c_{21} and c_{41} are created in agents a_2 and a_4 respectively. However, agent a_2 starts another protocol session P_2 before it terminates its participation in P_1. According to the checkpoint protocol, it will skip the creation of a new checkpoint and proceed to merge P_1 and P_2 as a single protocol session. As the initiator of P_2, a_2 uses $\langle 2, 1 \rangle$ as the identifier for P_2. This, in turn, causes the creation of c_{11} in a_1, with label $(c_{11}) = \{\langle 2, 1 \rangle\}$ as its identity. Progressing further, before ending P_1, agent a_3 participates in yet another protocol session P_3. Rather than creating a new checkpoint for P_3, agent a_3 simply modifies the label of c_{31} to become $\{\langle 3, 1 \rangle, \langle 5, 1 \rangle\}$, indicating that c_{31} is used as the checkpoint of agent a_3 for both sessions, one started by a_3 and identified as $\langle 3, 1 \rangle$ and the other started by a_5 and identified as $\langle 5, 1 \rangle$. In Figure 3.5, local checkpoints c_{11} - c_{21} - c_{31} - c_{41} - c_{51} form a strong group checkpoint due to the overlapping of corresponding protocol sessions. Afterwards, when a_1 terminates its participation in P_2 and starts a new protocol session P_4, c_{12} and c_{22} will be created and labeled as $\{\langle 1, 2 \rangle\}$, forming another strong group checkpoint.

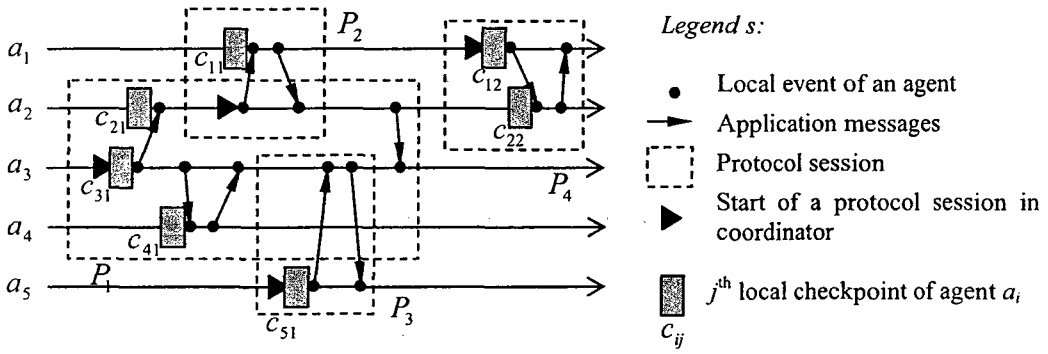


Figure 3.5 Group checkpoints for overlapped protocol sessions

Lemma 3.7: The ASGC protocol creates strong group checkpoints.

Proof: According to the protocol, each agent takes a local checkpoint if and only if it is going to participate in a new protocol session and meanwhile it is not currently involved in a previous protocol session. To show that the group checkpoint thus formed is strong, suppose that there is an in-transit

message m across the cut for a protocol session P_i (i.e., message m is sent before the cut but received after the cut), and then m belongs to a previous protocol session P_j that overlaps P_i. This contradicts the checkpoint protocol since an agent creates no local checkpoint if its previous protocol session is not finished. Consequently, the checkpoint protocol creates a strong group checkpoint, where a group is formed by either a single non-overlapped protocol session or a merger of all overlapped protocol sessions. □

Corollary 3.8: The ASGC protocol is correct.

Proof: From Lemma 3.7, the protocol creates strong group checkpoints and there is no message logging at all. For the corresponding protocol CDG to be proper, all program-order and message dependencies should be recorded. From the protocol, group information is piggybacked and propagated along with all application messages, and is saved upon checkpoint creation. As a result, both the sender and the receiver of a message record the corresponding message dependency. On the other hand, program-order dependency is maintained via the sequentialization of local checkpoints by the same agent. The protocol hence creates a proper CDG and is correct. □

The ASGC protocol is simple and efficient. Agent execution is not blocked, as checkpoint coordination is done via message piggybacking. A localized group checkpoint is created for each (non-overlapping or merged) agent protocol session, which usually involves a small number of agents due to localization of agent message interactions. Strong consistency of group checkpoints avoids message logging and hence reduces runtime overhead.

Besides checkpoints, additional group information is also created and saved with individual checkpoints, in order to speedup recovery upon agent crash. Message chains convey to each member of a protocol session the identity of the initiator, and also enable the initiator know all the members. In addition, each member learns of the protocol session(s) associated with each checkpoint. All of this information is useful for identifying recovery spread when a crash failure occurs (with either the initiator or a member agent). Essentially the recovery protocol needs to locate all checkpoints that depend on the most recent checkpoint of the crashed agent. From Corollary 3.8, all program-order and message dependencies should be retrieved, which is very similar to the dependency tracing among independent checkpoints [BL88, Wan93]. Besides, agent group checkpoints also record group information that can facilitate the reachability analysis. The following presents a recovery protocol that uses a three-phase procedure to find all affected agents. As a widely-held assumption, there is a recovery kernel associated with each process or agent. All special kernel messages used in the recovery protocol, e.g. freeze and acknowledgement messages in the following, are explicit and are different from normal application messages.

Recovery initiation: Recovery starts from the crashed agent, say a_i. (The kernel of) a_i will serve as the recovery coordinator. The most recent checkpoint of a_i will be used as the seed of recovery.

Locating relevant agents (Phase 1): A freeze message is propagated from the recovery coordinator to all relevant agents. This is accomplished as follows: i) the freeze message is first sent to the initiator of a_i's current protocol session; ii) upon receipt of such a message, an agent will freeze its execution and forward the message to every other member agent that it has talked to during this protocol session; iv) moreover, upon receiving such a message regarding a protocol session P_j, an agent will also forward the received message to each P_k if P_j overlaps or happened before P_k in this common agent.

Reporting to coordinator (Phase 2): Once frozen, every agent acknowledges the coordinator about the relevant protocol session(s) by which it is frozen, and the other agents it has tried to freeze.

Reset of agents (Phase 3): When the coordinator has received all acknowledgements, it will broadcast reset messages to all member agents hence identified. Each of these agents, in turn, will reset its state to its earliest relevant checkpoint and resume execution.

Special handling of messages: In-transit application messages corresponding to incomplete protocol

sessions are discarded (i.e., not delivered) by the kernel through the use of incarnation numbers.

In general, the above protocol recovers all agent protocol sessions that are affected by the crashed agent. It should be noted that at run time a protocol session might be incomplete when some member agent crashes before all its events in the session have occurred. Such a protocol session leads to the following possible complications: i) the initiator may not know of every member of the session, and ii) some protocol message may be in transit while the receiver has crashed or is involved in recovery. Fortunately, the agent group checkpointing is unaffected by these scenarios, as the occurrence of crash and recovery will directly cancel the corresponding group checkpoint. On the other hand, the above recovery protocol takes care of this incompleteness of group information by employing a diffusion-base mechanism for dependency tracing. A more detailed pseudo code version of this recovery protocol can be found in a recent publication [LWG06].

3.3.2 Group Checkpointing and Coloring

The features of strong consistency and asynchrony in ASGC benefit from the message chain assumption about agent protocol sessions. However, such an assumption might not hold all the time. For example, in a contract net protocol, an agent upon receiving a task message is allowed to ignore the message if it is not interested or not capable to perform that task. As a result, there is no message chain going back to the task initiator. In such a case, the task message becomes a "non-influential" message [THT98] which if ignored has no effect on either the correctness or the progress of the agent application. However, from the perspective of a recovery kernel, such a message needs to be taken care of so that it will not cause dependency or consistency error in case of failure. A simple choice is to log this message, so that the corresponding protocol sessions can be decoupled in recovery. It introduces a bit additional overhead at both run time as well as recovery time, and on the other hand adds to the strategy more flexibility in terms of group management.

The ASGC protocol has a localized but uncontrollable recovery effect. In case of agent crash, all agent protocol sessions that overlap with or happen after the one involving the crashed agent need be recovered as well. Due to localization of agent interactions, the recovery spread among protocol sessions is also localized. However, to maintain the strong consistency of group checkpoints, this spread is subject to the actual message pattern which is not controllable at all. As a result, the current strategy of checkpointing every protocol session takes many checkpoints that are in fact useless. Also, frequent creation of these checkpoints does harm the runtime performance. Stopping the recovery spread requires effective removal of all dependencies among checkpoints. As in many checkpoint and recovery protocols, logging is a useful tool for removal of message dependencies. Meanwhile, logging is also considered as expensive and should be employed only where justified. In practice, it is worthwhile to pay the price of logging a few messages for the return of more localized and controllable recovery. The real challenge is hence to carefully decide the portion of messages to be logged.

To improve the efficiency of checkpointing, group checkpoints should be managed at proper positions with respect to the actual message pattern, which might or might not be in the form of agent protocol sessions. This in turn requires a thorough study of the localization of agent interactions. For ease of presentation, this chapter uses a general notion of 'locality' to refer to localized message interactions. In the design of parallel and distributed applications, dependencies between processes are usually decomposed into a set of well-structured sub-tasks. As a result, corresponding message interactions tend to be localized within these sub-tasks and form small sub-regions in space and time. This modular principle is well supported by popular design techniques such as agent role model [Ken00] and parallel architectural skeletons [GSP02]. In general, distributed applications are likely to exhibit locality in terms of message interactions, and a

corresponding execution often consists of 'locality regions' with the following properties: i) they are almost disjoint from each other in space and time; ii) each locality region involves a relatively small percentage of processes; iii) majority messages are clustered inside these regions and there are a few inter-region messages. Examples of locality regions include but are not limited to (individual or merged) agent protocol sessions, etc. Consequently for a particular process, there are almost distinct locality intervals where the process interacts more frequently with only the small subset of partners of its corresponding locality region. In other words, within such a locality interval, whoever a process interacts with recently will likely be involved in future interactions. This makes it possible for a process to have quite accurate perceptions about its near future, based on the knowledge about its past interactions. These locality properties all together motivate the group checkpoint strategy to be discussed below. Detailed study of the locality phenomenon is given separately in Chapter 4.

The strategy of group checkpointing targets at both optimizing message logging overhead and limiting the recovery spread. ASGC performs well on one hand but has no control on the other. Also, it loses commonality as well as flexibility when being put into application. Therefore, an improved strategy should be based on the general rather than specific features of distributed applications. From the results developed in Chapter 3.2, a recovery can be limited within a subset S of behavioral fragments, if and only if all dependencies between S and other fragments outside of S are effectively removed. In principle, processes involved in S need to apply a logging policy by which all messages from outside of S are logged. This policy is specific to those behavioral fragments in S, and is also useful to distinguish all checkpoints associated with S from others. The term "group checkpointing" hence no longer refers to just checkpoint coordination, but a more general management of all processes involved in S, including group formation,

checkpoint creation, logging policy and consequent group recovery.

Assuming that processes form locality regions of message interactions during their lifetimes, a group checkpoint can be managed around such a region so that only a subset of processes will be involved. Similar to global checkpointing that avoids message logging by coordinating all processes, processes of each locality region can avoid logging the large number of intra-region messages by taking a coordinated group checkpoint before they start the region. Meanwhile, a selective logging policy can be applied to log only the small number of inter-region messages, which will effectively remove the corresponding recovery dependency established between group checkpoints. From the locality assumption, majority of messages are clustered inside locality regions, and hence the runtime logging overhead can be greatly reduced.

The above group checkpoint strategy requires member processes of a same group to act in coordination for both checkpointing and logging. This can be managed via a group coloring mechanism, by which a unique group identity is propagated among group members during checkpoint coordination, and afterwards each group member colors its outgoing messages with its current group identity. Upon receiving a message, a process can decide if logging should be performed, by simply checking if the message carries a same color. From a global view, there are groups of checkpoints, each associated with a specific logging policy that is similar to a distinct color assigned to it. In the above procedure, coloring of a group is the result of checkpoint coordination, and then is used to implement the group logging policy.

The CDG model hence can be extended with a coloring feature, in order to capture the group-specific logging policy regarding particular checkpoints. Formally, a checkpoint $c_{ik} = < S_{ik}, L_{ik}, N_{ik}>$ is extended to a *colored checkpoint* $<S_{ik}, L_{ik}, N_{ik}, T_{ik}>$, where T_{ik} = a tag of effective color for checkpoint c_{ik} (and the associated behavioral

fragment B_{ik}) such that message interactions between behavior fragments of identical color need not be logged. The resulting CDG with colored checkpoints is called a *colored CDG*. As a result of the logging policy, message dependency may exist in the colored CDG only between any two nodes with identical color. A colored CDG becomes *proper* if all inter-group messages are logged so that message dependency does not exist between nodes with different colors.

As an example, coordinated checkpoint protocols [TS84, CL85] form a proper CDG such that nodes associated with a same global checkpoint are of the same color. Message dependency edges exist only among these checkpoints. Other messages (such as in-transit messages) are recorded into the logs associated with corresponding checkpoints for replay upon recovery. On the contrary, log-based protocols [BLL89, SY85] intend to create a proper CDG such that each node has a distinct color, and there is no message dependency edge at all.

The coloring feature captures the intrinsic nature of group checkpointing. In a colored CDG, checkpoints form groups according to their colors. Members of the same group can avoid logging messages between each other. Actions of assigning a same color to group members should be coordinated in order to avoid unnecessary logging of intra-group messages. Introduced next is the notion of atomic group formation towards minimizing such logging efforts.

3.3.3 Atomic and Non-Atomic Group Checkpoint

Coloring of a group checkpoint can be done atomically or non-atomically. Atomicity means that the coloring procedure is as if it is instantaneous across all group members. Note that a group is defined as 'formed' once all member processes join the group by creating a local checkpoint. Upon formation a group starts its occupation of a space-time sub-region in a distributed execution. This is also the time when the group is supposed to

be atomically colored, which equivalently requires that the unique group identity is conveyed to every member process as if instantaneously. Afterwards, any message interaction between members of the same group is not logged. Hence atomic coloring minimizes logging latency, as a process will not be delayed by logging messages from other processes with the same color. Atomic group coloring also ensures consistency among the local checkpoints of the same color (i.e., forming a consistent cut). In formal terms, we have the following definition about atomic group checkpoint coloring.

Definition 3.6 (Atomicity of Group Checkpoint):
A group (of checkpoints) is atomically colored if it satisfies the following conditions: (a) *consistency*: the local states associated with these checkpoints are consistent, i.e., can be used to form a consistent global state, and (b) *uniformity*: uniformly logging is not applied to message exchanges between behavioral fragments in the same group.

It is certainly impossible to instantaneously color an atomic group with designated member processes. However, it can be managed if the group membership is decided on the fly. Described below is a simple asynchronous *Atomic Group Checkpoint* (AGC) protocol that exploits locality of process coupling using an invitation-based mechanism.

Starting a group: An initiator process can start a new group of tightly-coupled behavioral fragments, by first taking a colored checkpoint and then append its color to its outgoing messages. An invitation list consisting of processes invited to join a group is constructed based on the initiator's perception of the future locality.

Propagation of invitation: The invitation list is appended to the first message sent from a member of the group to a process in the invitation list.

Acceptance/decline of invitation: An invited member may decline by taking no action if its perception of its future locality differs from the invitation. However, if it accepts an invitation, it will create a new checkpoint with the new color.

Logging policy: Messages are always colored. Any Message tagged with the same color as that of the receiver will not be logged.

From the above description, it is apparent that the formation of a group involves a master and a set of consent slaves. In actual implementation, both design time and runtime knowledge of locality may be incorporated to decide on the missing details of the protocol, e.g., the construction of the invitation list and the criterion to decline an invitation. For example, in task-oriented applications like multi-agent systems, design time knowledge may be used in deciding the future interaction locality of an initiator agent for the purpose of initiating a group formation. On the other hand, the initiator's perception about locality may not be accurate. In such a case, runtime or design time knowledge of an invitee agent may be used to decline an invitation. A detailed description of the checkpoint protocol is given below. For simplicity, logging of determinants is assumed implicitly.

Data structure for a process p_i:

cur_chpt = The current local checkpoint number, initialized to 0;

cur_color = A two-tuple indicating current color of p_i, initialized to $(i, 0)$;

cur_group = A set of processes as members of p_i's current group, initialized to $\{p_i\}$;

Actions for an initiator p_i on starting a new group:

cur_chpt = *cur_chpt* + 1;

Take a new checkpoint;

cur_color = (*i*, *cur_chpt*);

cur_group = {invited processes, p_i};

Actions for process p_i on sending a message m to p_j:

Tag m with *cur_color*;

if this is the first message to p_j after current checkpoint **and** $p_j \in$ *cur_group*

then tag m with *cur_group*;

Actions for process p_i on receiving a message m colored in C from process p_j:

if it is the first time to receive a message tagged with *group* **and** p_i decides to join the group **then begin**

cur_chpt = *cur_chpt* + 1;

Take a new checkpoint before m is delivered;

cur_color = C;

cur_group = *group*;

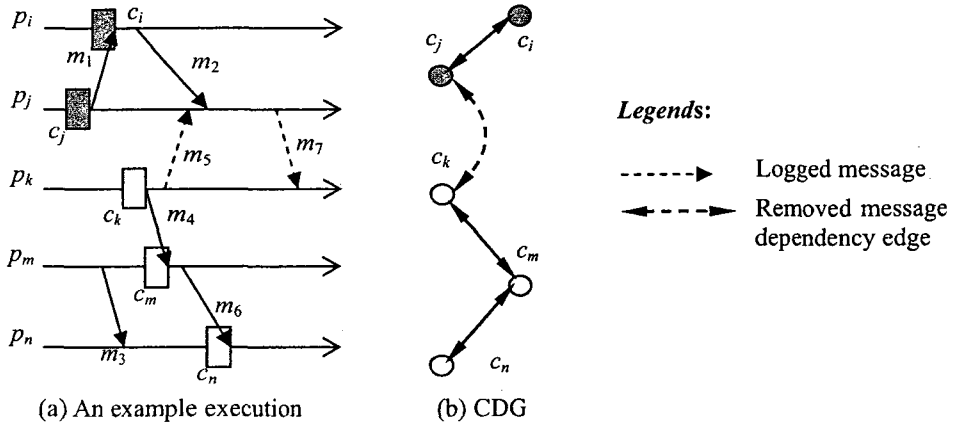
Record message dependency with process p_j with respect to color C;

end

else if $c \neq cur\_color$

then log message m;

else record message dependency with process p_j with respect to color C.



(a) An example execution (b) CDG

Figure 3.6 Checkpointing with the AGC protocol

As an example, in Figure 3.6(a), p_j initiates a 'red' group involving p_i and p_j by tagging an invitation on message m_1. p_i accepts the invitation by creating checkpoint c_i. c_j and c_i form the red group. Afterwards, p_k tries to start a 'white' group involving also p_j, p_m, *and* p_n by tagging the invitation on application messages m_4 and m_5. p_m decides to join by taking c_m and later propagates the invitation to p_n by tagging it on m_6. As a result, p_n also joins the group by taking c_n. On the other hand, upon receipt of m_5, p_j decides not to join by logging m_5. As a result, m_7 is still colored in 'red', and hence is logged by p_k. Figure 3.6 (b) shows the corresponding CDG, where the message dependency between c_j and c_k is removed.

Theorem 3.9: The AGC protocol is correct.

Proof: This follows immediately from the logging policy: messages between fragments of different colors are always logged. Moreover, a message dependency edge is not included in the protocol CDG, whenever all messages between the corresponding fragments are logged. Hence the

CDG is always proper and the AGC protocol is correct. □

Theorem 3.10: Identically colored checkpoints created by AGC form an atomic group.

Proof: According to the protocol, at the first instance a process receives an invitation tagged with an application message from a member of a group, (the kernel of) the process decides whether to join the group. Upon acceptance, it first creates a checkpoint of the same color as that of the initiator, before actual receipt of the message (i.e., delivery of the message to the process by its kernel). As a result, any two checkpoints of the same color are causally consistent. Hence condition (a) on consistency holds. In addition, once a checkpoint is created, the process will exercise the no-logging policy on all received messages tagged with the same color. Together with the fact that any message it sends out is tagged with the new color, no behavioral fragment of the same color will log this message as well. Thus condition (b) on uniformity is also satisfied. □

The AGC protocol is a simple and asynchronous protocol. Simple color tagging and invitation propagation suffice to achieve the objective of atomicity. In joining or starting a group, a process does not have to block for handshake acknowledgement from other processes. Flexibility is introduced by allowing an invited process to decline from joining a group. Thus, the actual group formed is a subset of the original invitation list, and the color identifies the actual group.

There are other possible variants to the AGC protocol without affecting correctness and atomicity. For example, instead of fixing the invitation list at the initiator, the invitation list can be modified by any member of the group before it is tagged with the first message to be sent to another process on the invitation list. This modification does

not alter correctness: logging is still maintained for messages exchanged between fragments with different colors. Due to the restriction that acceptance to join a group can occur only when receiving invitation for the first time, atomicity remains unaffected: checkpoints of an identical color remain consistent following the same arguments as before, and same is the case for uniformity.

Groups can also be colored in a non-atomic manner. In fact, neither consistency nor uniformity is necessary for creating a proper CDG. Inconsistent checkpoints or cuts can be used in recovery if the backward messages across the cut can be identified and discarded [BMP+03]. The generic recovery protocol introduced in Chapter 3.2.2 guarantees this since messages recorded in earlier log sets are not re-delivered. Non-atomic group coloring provides more flexibility for members joining a group and taking a checkpoint, at the expense of some redundant logging. The following *Non-atomic Group Checkpoint* (NGC) protocol is such a candidate.

Starting a group: An initiator process can start a new group of tightly-coupled behavioral fragments, by first taking a colored checkpoint and then append its color to its outgoing messages. An invitation list consisting of processes invited to join this group is constructed based on the initiator's perception of locality.

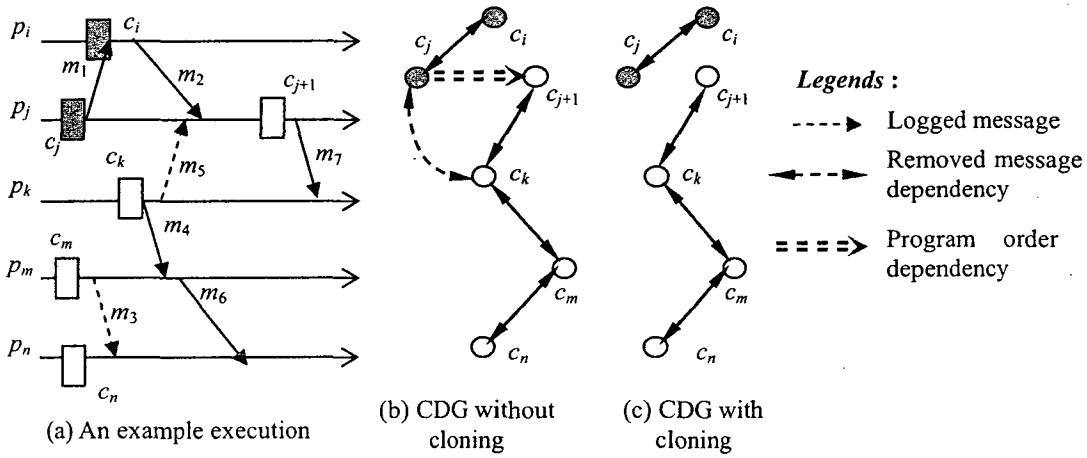
Propagation of invitation: The invitation list is appended to the first application message sent from a member of the group to a process in the invitation list.

Acceptance/decline of invitation: An invited member may decline to join a group by taking no action if its perception of its future locality differs from the initiator. However, if it accepts an invitation (promptly or later), either it will create a new checkpoint with the new color, or it will convert the existing colorless checkpoint into one with the chosen color.

Leaving the current group: A process can create a minimal and colorless checkpoint at any time when it perceives no further tight coupling with the current group. A colorless checkpoint reflects that it is not part of any group yet.

Logging policy: Messages are tagged with the color of the sender. Any message tagged with the same color as that of the receiver will not be logged, and the corresponding message dependency edge will be recorded. A message from/to a colorless checkpoint is always logged.

In the above description, both design time and runtime knowledge of locality can be used for decision making. A process has the option to decline an invitation, based on its own perception of locality. Detailed algorithmic description is omitted and can be found in [WLG05].



(a) An example execution (b) CDG without cloning (c) CDG with cloning

Figure 3.7 Checkpointing with the NGC protocol

Figure 3.7 shows an example of non-atomic group coloring. Process p_m and p_n have initially left their previous localities and created minimal (and colorless) checkpoints c_m and c_n. p_j and p_i take checkpoints c_j and c_i respectively to form a 'red' group. When p_k tries to initiate a 'white' group involving p_j, p_m and p_n, it tags the invitation on messages m_4 and m_5. Upon receipt of m_5, p_j decides to decline (ignore for the time being) this invitation in order to complete its interactions in its current locality (the red group). Hence it will log m_5. Some time later, p_j finishes its task in the 'red' group. It decides to join the 'white' group and take a new checkpoint c_{j+1}. As a result, m_7 is colored in 'white' and is not logged by p_k. In the other side of the world, when p_m receives m_4, it decides to accept the invitation by simply setting the color of c_m to white. Similarly, later when p_n receives m_6 it also decides to join the white group by setting the color of c_n to white. As a result, messages m_3 and m_5 are the only messages logged. Figure 3.7 (b) shows the corresponding CDG, where the message dependency between c_j and c_k is removed.

Theorem 3.11: The NGC protocol is correct.

Proof: This immediately follows from the logging policy. In particular, a message is always logged unless the sender and receiver colors match. Otherwise, the corresponding message dependency is recorded. Hence the properness requirement (b) (see Chapter 3.2.1) is always satisfied. By assumption, the checkpoint protocol records all determinants for deterministic re-execution of each behavioral fragment and maintains program order in its CDG. Hence the protocol CDG is proper and the NGC protocol is correct.□

The NGC protocol relaxes the atomicity requirement and provides more flexibility in group formation around locality. In particular, when a process receives an invitation, it may defer joining the new group in order to finish its current task in the present locality region. This avoids coupling of two groups together as well as logging of many messages later. Furthermore, a process can exit from its current locality independently by creating a new checkpoint. This separates the process from the old group so that its future failure will not affect its past partners. Such a separation need not be immediately accompanied with the formation or participation in a new group. The latter can be decided at a later, more opportune time. There are also variants of NGC that can be developed, in particular with regard to the modification of an invitation list.

3.3.4 Cloning-Based Recovery

Checkpoint coloring reduces recovery spread by removing message dependencies between different colors via message logging. Since message dependency is the major source of recovery spread, it is conceivable that group checkpoint protocols can form appropriate groups to localize the recovery spread, especially with applications that exhibit good locality. However, since a process can belong to multiple groups in its lifetime, program-order dependencies can still cause recovery spread between these

groups. If groups are not appropriately formed, there could be cyclic dependencies among them and even domino-like effect in recovery. In such a case, recovery of a newly created group will finally cause recovery of some groups created much earlier. This could happen when asynchronous participation is allowed in group formation. Similar to the domino effect [Ran75] in uncoordinated checkpointing, avalanche could arise because groups are formed independent of each other.

Avalanche rollback can be avoided if program-order dependency among different colored groups can be removed. In general, the program-order dependency between two adjacent checkpoints c_{ik} and c_{ik+1} arises because, in recovery, process p_i needs to continue its execution after finishing the recovery of behavioral fragment B_{ik} in order to complete its work. However, this is only necessary for the failed process, say p_j, but not for other processes that are forced to recover with p_j as a result of message dependency.

In recovery, a failure-free process p_i only needs to re-execute those behavioral fragments that are relevant to the failed p_j's recovery, in order that the dependency requirements for p_j's computation can be re-generated. Therefore, instead of rolling back a failure-free process p_i to help p_j, the recovery protocol can spawn a replica of p_i to do exactly the recovery job for those behavioral fragments on behalf of p_i, while the original p_i continues with its execution. This replicated copy is called a (process) *clone* of p_i, which performs a deterministic re-execution of a subset of p_i's behavioral fragments. In general, such a clone has the following properties:

i) **Resumable execution**: A clone is started from a particular local checkpoint of the corresponding process.

ii) **Limited lifetime**: The execution of a clone is terminated at the point where another local checkpoint is to be created, i.e., it re-executes only a subset of the behavioral fragments of the original process;

iii) **Deterministic external behaviors**: A clone deterministically re-executes the behavioral fragments, faithfully reproducing message interactions with respect to other processes.

Process cloning can be implemented using regular checkpoint and recovery techniques. For example, a clone of p_i can be initiated from a local checkpoint of p_i, similar to resuming p_i from that checkpoint. Deterministic re-execution of behavioral fragments can be guaranteed by the generic recovery protocol. When finishing its job, a clone can be terminated by the kernel, in the same way it terminates a normal process for recovery. The point of termination can be decided by counting down the number of message events to be executed by a clone in each behavioral fragment, and this number can be saved at runtime with a checkpoint when the next checkpoint is taken. In addition, since a process and its clone can co-exist, additional mechanisms should be applied to distinguish messages for the original process and the clone. Effective use of clones is certainly platform dependent.

Assuming that clones are used, each fragment can be activated as a clone during recovery. The corresponding checkpoint is said to be *cloneable*. Consequently, in a CDG, the program-order dependency edge $c_{ik} \rightarrow c_{ik+1}$ can be removed if c_{ik} is cloneable. The properness of CDG, $G = <C, D>$, hence can be extended (refer to section 3.2.1), by changing property (a) as follows:

(a) D includes all program-order dependency edges $c_{ik} \rightarrow c_{ik+1}$ among checkpoints in C, unless c_{ik} is cloneable.

The generic recovery protocol can also be extended accordingly, in order to accommodate the use of clones during recovery. This includes termination of clones, and messages deliveries between clones and other processes. When a given reachable component during recovery does not involve the most recent checkpoint of a process p_i,

the generic recovery protocol will start a clone of p_i instead. Correctness is guaranteed by the satisfaction of the above extended property (a).

Process cloning reduces recovery spread by removing program-order dependencies. When applied with group checkpoint protocols, cloning can isolate groups from each other during recovery. Recovery is therefore localized to a single group, while the growth of a group is totally controllable by selective logging. For the example of using NGC protocol discussed previously, Figure 3.7 (c) shows the resultant CDG with cloning.

Theorem 3.12: When cloning is applied to all checkpoints, each colored group created by the NGC protocol forms a reachable component.

Proof: In the NGC protocol, message dependencies only exist between checkpoints of the same color. Since all checkpoints are cloneable, there is no program-order dependency edge in the corresponding CDG. A reachable component therefore can only involve those checkpoints with a same color. Hence the claim. □

Compared to log-based recovery, clone-based recovery will likely help group checkpoint strategy to produce better runtime performance, when only the few messages between groups are logged. Moreover, the localized group recovery effect makes it more attractive as protocol designers can focus on effectively controlling the size of each group in group formation. Discussed next are issues about limiting the growth of a group.

3.3.5 Checkpointing for Bounded Recovery

Cloning decouples program-order dependencies among groups and isolates groups from each other during recovery. A colored group hence serves as an independent recovery unit, i.e., any failure inside of a group triggers a recovery involving only member behavioral fragments of that group. In particular, in addition to the failed process, only clones

corresponding to checkpoints of the same colored group are needed. From the recovery perspective, obviously a large group involving too many processes or too much work is not favorable. In large-scale systems, localized recovery will prefer groups with bounded size.

Limiting the size of a colored group also places a bound on the recovery latency. The size limit of a group is referred as its *group size*. A group checkpoint protocol is *k-bounded* if it satisfies the following:

(i) Group size is at most k.

(ii) A process willing to join a group should not be declined unless the limit k has been reached.

Condition (i) establishes the maximum size limit of each group, and condition (ii) ensures that the growth of a group is always enabled until the limit is reached.

Depending on whether group membership is decided a priori (upon initiation) or is distributively decided by interested processes subsequently, there are two types of k-bounded group checkpoint protocols, namely, *static* and *dynamic k-bounded* protocols respectively. A static k-bounded protocol is not flexible and cannot adjust to runtime conditions unknown at application design time. A dynamic k-bounded protocol allows both compile time and runtime optimizations for the formation of groups. As in the NGC protocol presented in Chapter 3.3.3, a process may decide to join a group or reject an invitation dynamically, depending on the runtime knowledge accumulated at that time.

Limiting the growth of a group necessitates additional performance cost, since inter-group dependencies need to be removed and corresponding messages have to be logged. Hence the bound k must be carefully chosen in practice.

Checkpoint protocols, such as the NGC, involve synchronization information that must be propagated from process to process. Information delivery can involve either

explicit protocol messages or embedded application messages. The former introduces additional message overhead, unlike the latter which piggybacks synchronization information into application messages.

Definition 3.7 (Embedded Protocol)
Protocols that embed all synchronization information in application messages are termed as *embedded protocols*. Usually embedded protocols are preferred.

Lemma 3.13: Embedded dynamic k-bounded atomic group checkpoint (AGC) protocol does not exist.

Proof: Consider a counter-example in which a group starts with an initiator process that sends an application message to every other process. The synchronization information embedded in each application message cannot carry the exact membership information due to the dynamic k-bounded-ness assumption. Upon receipt of this message, a process must decide to join or not to join the group before moving forward. This decision cannot be made without violating either the size limit or the progress constraint, as no further protocol-specific message exchanges can occur in the system. □

Clearly, formation of bounded groups requires explicit protocol messages that cannot be embedded in application messages. This originates from the fact that consistent global information cannot be known instantaneously by all processes. Explicit protocol messages are actual overheads that should be minimized. The following lemma establishes a lower bound on the number of explicit messages for a dynamic k-bounded AGC protocol.

Lemma 3.14: A dynamic k-bounded AGC protocol requires at least $m - 1$ protocol

messages in a system consisting of m processes.

Proof: Each of the $m - 1$ processes other then the initiator must be invited or otherwise an interested process may never be involved to make up the group size k, thereby violating condition (ii) discussed previously. □

The previous lemma assumes nothing about the application message pattern. However, for many applications that exhibit good locality, it is observable that a locality region usually involves a large enough number of tightly-coupled processes and each process is reachable via a message chain from an initiator process. Each process in such a locality region can be considered as a willing process in joining the corresponding group. Under these assumptions, a simple dynamic k-bounded AGC protocol can be developed as a hybrid of message piggybacking and explicit protocol messages.

Under the locality assumption explained earlier, a group is formed based on the clustered message interactions that occur in space and time. However, in practice, it is entirely possible for a process to be active in multiple groups at the same time. For example, a process may interleave between two roles in its interactions with other processes. As a result, a group may actually contain more than one checkpoint fragment from the same process.

The use of cloning provides the flexibility to make use of such a colored group. Since cloning decouples program-order dependencies among groups, such a group can still recover independently. The following token-based *Bounded Atomic Group Checkpoint* (BAGC) protocol is such an example.

Starting a group: An initiator process can start a new group by first taking a colored checkpoint and then append its color to its outgoing messages. The initiator will start with exactly k tokens in order to limit the number of fragments in the group.

Distribution of tokens: A process holding some tokens can invite its partner processes in its perceivable locality to join its current group, by offering them some of the tokens. These tokens will be embedded in its first message to an intended group member.

Handling an invitation: Upon receiving a message of a specific color for the first time and the message is embedded with tokens as an invitation, then the receiver must decide if it will join. If it decides to join, it must first return all remaining tokens of the old group that it is holding to the initiator of that group. Then it will create a new checkpoint with the color of the new group.

To account for the tokens received, it performs the following: (i) consume a token (for its current fragment), (ii) keep some tokens for distribution to its perceived partners, and (iii) return the rest of the tokens to the initiator. If it decides not to join, it will return all the tokens to the initiator. Tokens coming from any subsequent messages of the current checkpoint color will also be returned to the initiator.

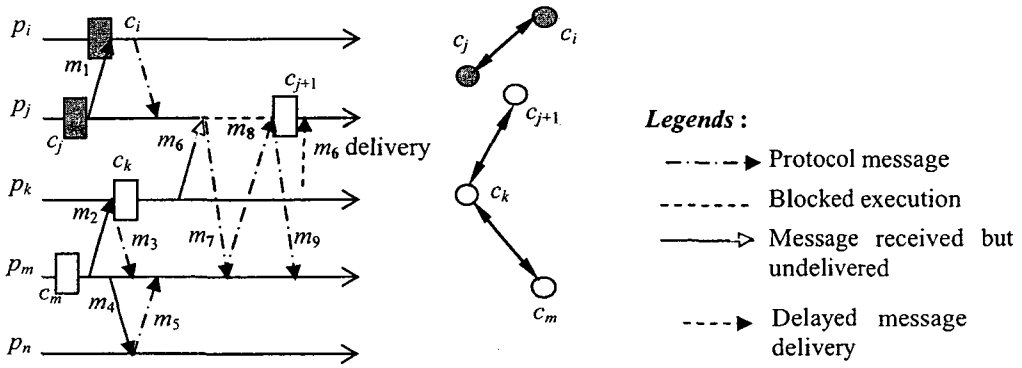
Reporting of token consumption: Each token consumer will report its consumption to the initiator. The initiator will construct its knowledge base about token consumption in the system, according to the reports it receives.

Requesting of tokens: Upon receiving a message of a specific color for the first time, if the message is not embedded with any token and if the receiver wants to join the group, it will send a token request to the initiator and wait for either a token grant or a reject message before proceeding further.

Collection/re-distribution of tokens: The initiator will collect all returned tokens. It will eventually respond a token requestor process with either a token grant message when some tokens become available, or a rejection message if all tokens are used up (based on its knowledge of token consumption).

Rejoining a previous group: Upon receiving a colored message of a previous group for the first time since leaving the group, the receiver can decide to rejoin by accepting an invitation or requesting a token as discussed previously (depending on whether or not some token is embedded with that message).

Logging Policy: Messages are always colored. Any message tagged with the same color as that of the receiver will not be logged, and the corresponding message dependency edge will be recorded.



(a) An example execution (b) CDG with cloning

Figure 3.8 Checkpointing with the BAGC protocol

As an example, in Figure 3.8(a), p_j initiates a 'red' group involving its partner p_i by tagging a token on message m_1. p_i accepts the token by creating checkpoint c_i. c_j and c_i form the red group. Afterwards, p_m starts a 'white' group by tagging a token on each

application message m_2 and m_4. p_k decides to join the group by taking checkpoint c_k, while p_n decides not to join and returns the token to p_m via a token return message m_5. Later p_k sends a 'white' message m_6 to p_j without a token. Upon receiving m_6, p_j intends to join and it sends a token request message m_7 to p_m, and then blocks execution before actual delivery of m_6 (by kernel). As a result, p_m sends a token via token grant message m_8 to p_j. Upon receiving m_8, p_j resumes execution and joins the 'white' group by taking a checkpoint. Message m_6 is subsequently delivered (by kernel) after the creation of c_{j+1}. m_3 and m_9 are the messages reporting token consumption, from p_k and p_j respectively. Figure 3.8 (b) shows the corresponding CDG in presence of cloning.

Theorem 3.15: The BAGC protocol is correct and it generates k-bounded atomic groups.

Proof: The correctness follows immediately from the logging policy: messages between fragments with different colors are always logged. Moreover, a message dependency edge is always recorded whenever there is an un-logged message between two fragments of the same color. Since cloning is assumed with all checkpoints, there is no program-order dependency edge in the corresponding CDG. Hence the CDG satisfies the extended properness property (refer to section 3.3.4) and consequently the BAGC protocol is correct.

Atomicity is ensured that the set S of earliest checkpoints of each process in the group are consistent: there can never be a message sent by a member process p_i of the group after its checkpoint c_{ik} in S but is received by another member process p_j of the group before its checkpoint $c_{jk'}$ in S without violating FIFO-ness of message delivery, since no message with this group color is ever received before $c_{jk'}$. In addition, within each checkpoint fragment of the group, the no-logging policy of messages

between group members is uniformly enforced on all messages as a process joins a group before receiving any message from the group.

For k-bounded-ness, we prove that for each group under formation, (i) at every step, the number of tokens (taken up or in transit) is equal to k, and (ii) at any global state, if there is a process interested in joining the group, then within a finite number of steps there is a future global state such that the number of free tokens will be reduced accordingly, until the number of free tokens becomes zero. Property (i) is straightforward and follows immediately from the fact that each process action preserves the number of tokens. Regarding property (ii), if a process interested in joining is one that has just received a message tagged with tokens, it simply joins and thus property (ii) holds. Otherwise, the process is one that has received a colored message but the message does not carry any token. It then sends a request to the initiator. If the system state contains at least one free token, this token will be forwarded in finite number of steps to the initiator and eventually to the requesting process. Thus property (ii) must hold. □

The BAGC protocol is simple. Token distribution via message tagging allows a receiver process to join or re-join a group immediately upon receipt of a message. Flexibility is introduced by allowing tokens to be redistributed. Explicit reporting of token consumption enables the initiator to unblock a waiting requestor and is necessary for progress requirement.

The BAGC protocol can be extended into other protocols. For example, by relaxing the atomicity into non-atomicity, a token requestor can progress by pessimistically logging messages from the group it intends to join, without waiting for a token from the initiator. The resulting protocol satisfies a *weak k-bound* property: a process that is

persistently willing to join a group is not declined unless the limit k has been reached. In addition, more flexibility can be introduced in terms of token management. For example, instead of fixing the token adoption at the time of joining, a group member can keep some tokens that it has received, rather than directly returning them to the initiator. Furthermore, knowledge propagation through messages can be used for decision-making and token distribution.

3.4 Remarks

One of the main contributions of this chapter is the generic model of quasi-atomic recovery block that accommodates both deterministic and non-deterministic recoveries. The notion of a 'quasi-atomic' recovery block captures the portion of process behaviors being nullified during a single recovery. This property of quasi-atomicity is very general and it differs from traditional 'recovery blocks' in the following: quasi-atomicity covers all kinds of existing checkpoint and recovery techniques, including partial or complete recovery lines, consistent or inconsistent recovery lines, message playback and discard, etc. Then an important theorem is proved regarding the correctness of multiple recoveries within a recovery behavior: every recovery is confined within a quasi-atomic recovery block, and all blocks should be well-ordered. Well ordering in addition guarantees quasi-atomicity while capturing potential concurrency among multiple recoveries. This integrated view provides more general results as compared to other correctness requirements adopted by the various checkpoint and recovery strategies in the past. The model hence is not only useful for understanding existing protocols, but also helpful in developing new checkpoint and recovery strategies.

Another major contribution of this chapter lies in the group-based checkpoint and recovery strategy. Aiming at a hybrid objective of both reducing runtime overhead and

localizing recovery effect, this strategy explores some special characteristics of large-scale distributed systems, namely locality of message interactions among processes. In particular, exhibition of 'locality regions' facilitates performing group checkpointing and selective logging efficiently, which are otherwise expensive to apply or difficult to decide in a distributed system involving a large number of processes and messages. In group-based checkpoint protocols hence developed such as AGC and NGC, simplicity and efficiency are well achieved due to the following reason: group checkpoint coordination is managed asynchronously via piggybacking invitations over application messages, thus eliminates needs for synchronization through explicit kernel messages. The resulted checkpoint protocols are hence non-blocking. Flexibility is also introduced during group formation (in terms of maintaining group atomicity or not), on the cost of logging a few extra messages. On the other hand, process cloning is proposed as an effective way of handling program-order dependency between group checkpoints. As a result, a token-based BAGC protocol can limit the growth of a group by generating k-bounded atomic groups. Compared with global checkpointing, group-based checkpoints are localized without involving the entire universe of processes, and recovery spread is restricted to relevant group members only. In contrast to log-based approaches, there is no much message logging at run time as well as message playback during recovery. In other words, runtime performance is greatly improved as the cost of both global-wise checkpoint coordination and necessity of logging all messages is avoided. Locality of message interactions plays an important role in both aspects.

Recent reports from the parallel computing community seem to have justified the feasibility of the group checkpoint strategy. Similar strategies are adopted for MPI jobs [GHK+07, HWL08] and grid computing tasks [MSM08], not very long after the introduction of the work presented in this chapter [WLG05, LWG06, WLG06]. In

particular, each MPI job or grid task consists of multiple processes, which are divided into several groups and checkpointed separately in a group-wise manner. Inter-group messages are logged [HWL08, MSM08] or deferred (until all group checkpoints form an actual global checkpoint for the job) [GHK+07]. Group formation is based on the characteristics of job units (i.e., sub-jobs or sub-tasks). Besides the above similarities, these strategies are much simpler, compared with the AGC or NGC protocol. For example, in these strategies, consistency of group checkpoint is mandatory. In addition, the grouping of processes is only for the purpose of reducing global checkpoint latency [GHK+07] or preventing catastrophic global recovery of the whole grid [MSM08]. In [HWL08], a communication tracer is used to monitor and identify intensively communicating processes, and groups are formed based on the trace analyzer. However, unlike distributed systems, MPI programs are much more sensitive to network delay instead of the communication pattern itself. This strategy hence might be only subject to physical locality, e.g., it is better applied to the processes within the same cluster.

Chapter 4

Locality of Message Interactions

Message passing distributed systems are widely used in resolving large-scale problems like e-commerce and supply-chain management [WGP03, MLY06]. Such a system usually consists of a large number of messages and processes that tend to exhibit specific interaction patterns. In particular, messages are not distributed uniformly or arbitrarily among processes; rather they are quite localized around groups of processes over a period of time. For example, in an agent-based manufacturing process system [PBC01, Dee03], a sales agent might interact only with a few other agents including a customer, a sales manager and possibly some stock managers until a product order is approved or declined. During this negotiation process, all participating agents form a rather exclusive group, where they frequently interact with each other and their corresponding lifelines form a message-intensive sub-region in space and time. In many distributed applications, such regions are observable over the whole execution, containing the majority of all messages and leaving only a few messages in between. Each region hence forms a distinct locality of message interactions, which might display as different patterns in various applications.

Such locality phenomenon is a new observation and is clearly different from the one captured by the traditional locality notion [Bel66]. In the literature, relevant studies have been reported mostly on the caching problem of memory references [Den68] and web references [JB00a, JB00b, FAC+03], where locality is modeled as a recurrence pattern

over time, and is measured by the overall caching performance. Further study also shows existence of fine-grained recurrence structures and their hierarchies within both memory and web references [MB76, CC00]. Locality of message interactions, however, is about recurrence structures spreading over both space and time. Unlike in the totally-ordered time dimension, there is no such a concept of 'spatial distance' among processes, making the existing research methods no longer applicable. A new approach is hence required for understanding the properties about locality of message interactions, in particular, the origination and formation of message localization, the identification and measurement of locality patterns, and their availability as well as characteristics in distributed applications.

In general, locality is intrinsic to every computation due to localization of internal data dependencies. Traditional locality of memory references arises from the sequentialization of such dependencies resolved by local memory variables. Locality of message interactions comes from decomposing a computation into sub-tasks and resolving the inter-task dependencies via message passing. The hierarchy of localities naturally is often formed based on hierarchical organization of computations. In most cases, a distributed execution only comes with plain information about a set of partially-ordered message events. Due to lack of design time knowledge of the system, it is impossible to pin-point the localities corresponding to each sub-task. Even though, some ideal or simple locality patterns can still be easily identified, because certain locality properties they preserve can illustrate the actual design purpose. For example, a group of processes with no external communications can locate each other as group members by following the path of any message interaction. Similarly, a process group with only a few external messages can be identified via the path of frequent message interactions. Intuitively, such simpler locality patterns have more localized effect of messages, and are also more identifiable. A study on locality properties of such patterns would give rise to a

reverse engineering approach, by which detection of these properties will lead to effective identification of the corresponding localities. Since most distributed applications tend to employ simple synchronization protocols as well as communication patterns, this approach is expected to work in most cases.

An ideal or 'good enough' locality of message interactions usually observes the following properties: i) it contains most or even all associated messages inside and hence have a small percentage of external messages; ii) each pair of participating processes are tightly coupled via frequent message interactions with each other, directly or indirectly; iii) each participating process has a distinct interval of localized interactions almost exclusively with other participants. Consequently, identification of such a locality can start from investigating those localized interaction patterns within individual process lifelines. Such identified interaction intervals can be grouped together via the connectivity of frequent message interactions among them, and the resulted regions of localities are expected to have a small percentage of external messages.

This chapter presents detailed results and discoveries on locality of message interactions developed based on the above discussion. Chapter 4.1 reviews existing research works on locality of memory references and web references, etc. Chapter 4.2 discusses the origination as well as formation of locality of message interactions due to computation parallelization in message passing distributed systems. Chapter 4.3 introduces a generic notion of 'locality interval' for capturing popular recurrence patterns within individual process lifelines. Also discussed are issues regarding detection of locality intervals and formation of 'locality regions' via frequent message coupling among these intervals, based on plain knowledge about a given distributed execution. Chapter 4.4 presents experimental results on 'static' execution traces of two example distributed applications, including the traditional locality views of individual process

lifelines, identification of locality intervals from different processes, and characteristics of locality regions hence formed. Chapter 4.5 brings out the contributions of this chapter and compares the results with related work.

4.1 Traditional Locality of Reference

Locality of reference, also known as the principle of locality, is a phenomenon of continuous recurrences around the same subset of objects (among others). It was first observed as a process's references to entities in memory caching systems [Bel66], and was characterized by the well-known "working set" model [Den68]. An important property of such a "reference sequence" or "reference string" [Den70] is established based on the property observable at a time instance as "bursts of references are made in the near future to objects referenced in the recent past" [Van05]. Studies have been reported mostly on the caching problem of memory references [Bel66, Den68, DS72] and web references [JB00a, JB00b, MEW00], where locality is related to the overall caching performance, e.g. in terms of hit ratio [MGS+70], lifetime function [Bel69] and miss rate [Van05]. Then further study shows that a reference string of both memory and web references contains fine-grained structures called "recurrence patterns" [Van05]. This is reflected by the property of a time period as "for relatively extended periods of time, a program references only some subset of its name space or virtual address space" [MB76]. Related studies focus on qualitatively characterizing the existence as well as the hierarchy of ideal recurrence patterns named "Bounded Locality Intervals" [MB76], and stochastically modeling a reference string as a "phase-transition behavior" [DK75] for synthetic trace generation. In general, locality of reference is about localized recurrences along the timeline of a sequential behavior, with fine-grained structures in terms of recurrence patterns.

4.1.1 Localization of Recurrences

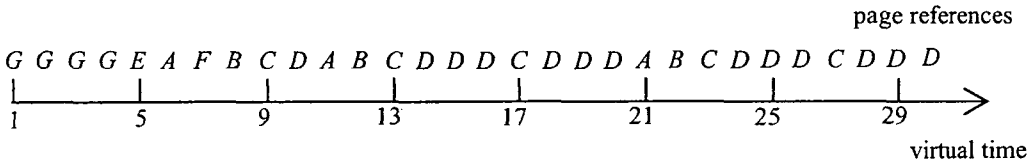


Figure 4.1 An example reference string from a program behavior

Locality of reference was discovered from improving the performance of virtual memory [KEL+62] in computer systems. With the innovation of storage hierarchy and paging mechanism, pages loaded in main memory need to be replaced by pages from the disk upon page faults. Since a page fault introduces a significant delay due to the much slower disk access, the replacement algorithm should efficiently minimize page faults, and hence improve the hit ratio [Mat71]. This consequently raised the question of which pages are essential to a program and which are replaceable, and eventually led to the introduction of the "working set" notion. Given a sequential program, a *working set* $W(t, \tau)$ is defined as the set of pages it has referenced in the virtual time window of length τ preceding time t [Den68]. Here virtual time is measured by the number of memory references, and the sequence of pages being referenced along virtual time is termed as a *reference string* [Den70]. Figure 4.1 shows an example reference string, where the working set at virtual time $t = 9$ with a window length $\tau = 4$ is $W(9, 4) = \{A, F, B, C\}$. A working set $W(t, \tau)$ constitutes a good prediction of an immediate future working set $W(t + \alpha, \tau)$ for a small time α. Intuitively, pages referenced in the past are highly likely to be reused in the near future. Such an assumption was justified in many application programs, where long phases of memory references were observed around relatively small page sets. An example behavior is illustrated in Figure 4.2, where dots depict references to the corresponding page space at the given virtual time.

The term locality was then used for the observation that a program behavior clusters its references around small subsets of pages for extended intervals. Two types of such

clustering were distinguished, namely temporal and spatial locality. The former is about the tendency of repeatedly referring to a same page, which could be due to looping, subroutine, and modular execution with private data. The latter is about the likelihood of adjacent pages being frequently referenced, which might be caused by related data being grouped and stored together or closely, e.g. arrays and sequential codes. Temporal locality encourages caching and spatial locality encourages prefetching of related objects. Both reflect the use of a generic problem-solving strategy: divide and conquer. In other words, a large problem is divided into sub-tasks which are then processed sequentially in a particular order. Locality of memory references is the localization of data accesses related to individual sub-tasks.

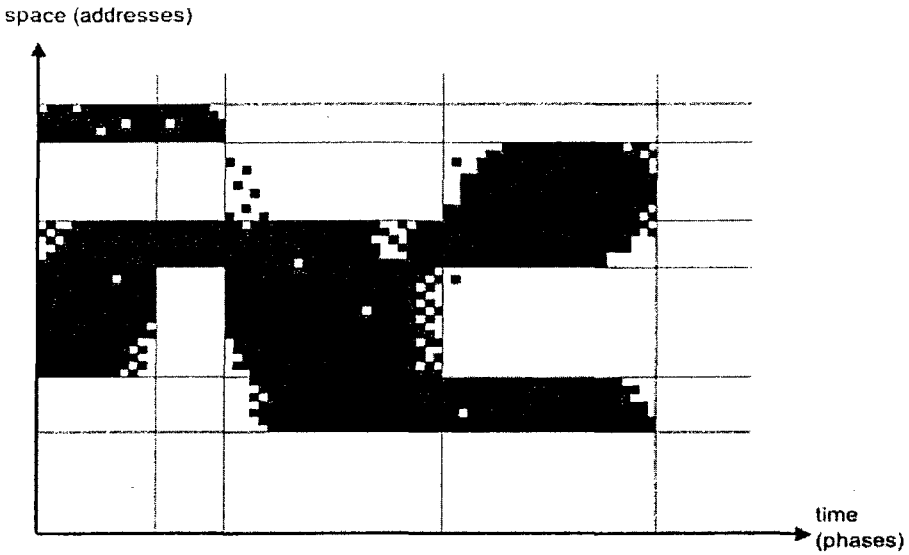


Figure 4.2 Locality behavior observed during program execution [Den05]

The locality principle together with the working set model brought new insights to understanding program behaviors. It benefits directly the prediction of a program's future memory needs, as employed in various page replacement algorithms. Intuitively, upon page fault, it is better to replace the page that is most unlikely to be reused in the future, i.e., whose removal will introduce maximum references until the next page fault. Locality of reference indicates that during any interval of execution, a program favors a subset of

its pages, whose membership changes slowly. In particular, a page being referenced more recently has a larger probability of being referenced again in the immediate future. The well-known LRU (least-recently-used) algorithm follows this principle by employing a stack that keeps the m most recently used pages (where m is the stack size in terms of pages) and was considered as the best performer among others [Bel66].

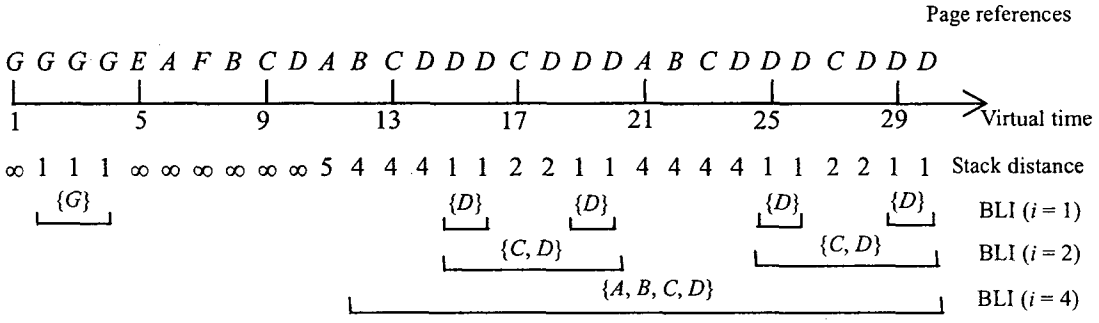


Figure 4.3 Stack distance and bounded locality intervals over a reference sting

Locality of reference is an experimentally observed phenomenon, and might not hold all the time. This certainly has impact on the overall performance of the replacement algorithms, which in turn can be used to evaluate the corresponding locality. For example, a program behavior with good locality, when applied with the LRU algorithm, will show good performance, in terms of large hit ratio, small miss rate of pages, etc. However, such a performance is subject to the size limit of LRU stack used in the algorithm, rather than the specific program behavior. A notion of "stack distance" or "recurrence distance" [MGS+70] was introduced based on LRU stack and can be used as the locality measure of a given behavior. Upon virtual time t, an LRU stack S_t has its topmost n entries $\{s_t(1), s_t(2), ..., s_t(n)\}$ keeping the n most recently referenced pages. Suppose the page x_t at t has been previously referenced, it hence can be found in the stack, say at position i. The *stack distance* Δ_t at time t is defined as $\Delta_t = i$ if $x_t = s_t(i)$, and $\Delta_t = \infty$ if x_t has never been referenced. Here Δ_t is the stack capacity required for avoiding a page fault at time t. Figure 4.3 shows the stack distances for the example reference string in previous discussions. More importantly, the average stack distance Δ over the timeline of a

program behavior measures the expected stack capacity for its proper execution. In addition, the normalized stack distance Δ/P (where P is the total number of all referenced pages) shows how localized a behavior is around small subsets of pages.

As an empirical result, reference locality is observable in many areas of computer science. It has been applied in designing operating system, database, and hardware architectures. The mechanism of caching or prefetching frequently-used objects is considered as a fundamental principle for performance acceleration. Besides, the locality principle also holds in networked environments, such as system-level network packet traffic and application-level web page references [CC00], web searching requests [MSA+08], and P2P data streams [CW06]. Packets are the basic units of network traffic between pairs of nodes, and are usually resulted from partitioning a big file being transferred between the same pair of source and destination nodes. Consequently, packets form streams of network traffic and exhibit temporal locality [JR86]. Also, page requests arriving at a web server exhibit reference locality, which is reflected by a small stack distance [ABC+96]. In fact, "10% of the file accessed on the server typically account for 90% of the server requests and 90% of the bytes transferred" [AW96]. This implies that certain web documents are really popular than others and hence better remain in the server caching system. On the other hand, client-side web browsers already take advantage of reference locality by caching the most frequently browsed pages. In general, locality of reference follows human practice of problem solving, and hence is critical to performance improvement.

4.1.2 Recurrence Patterns

The principle of locality asserts that during any period of time, a program behavior favors a subset of its reference pages, which tends to change slowly. The working set of immediate past tends to be quite similar to that of immediate future, making it possible

for mechanisms like caching and prefetching to work efficiently in memory management. However, correlation between disjoint reference patterns tends to be low as distance between them tends to large. Programs rather make transitions from time to time among different localities, each formed by a distinct subset of pages. This is described as "phase-transition behavior" [DS72], by which a program behavior is regarded as a sequence of phases, each covering a sequence of references over an associated "locality set". In general, a phase corresponds to a program's residency within a sub-task, while a transition is due to termination of the current sub-task and move-forward to the next one that establishes a new locality.

The phases can be modeled by a notion of "*bounded locality interval*" (BLI) [MB76], a maximal interval in which LRU stack distance does not exceed a given i and every one of the topmost i stack objects is referenced at least once. The set of topmost i stack objects remains active within the whole interval, and is hence called the *activity set* for the corresponding BLI. Figure 4.3 shows BLI's as well as their activity sets for i = 1, 2, and 4 within the previous example reference string. Since a stack distance $\Delta_t \leq i$ implies that one of the most recent i objects is re-referenced, a sequence of such time points captures the localization of re-references to these i objects over time. A BLI starts from the time when its activity set first occupy the topmost i positions of the LRU stack, and ends upon a reference to some new object no longer within its activity set. It is a rather 'distinctive' phase where a program has a stable and predictable working set. Between such phases are transitions where a program changes its working set, usually quite significantly. Formation of an activity set is a part of the transition preceding the corresponding BLI.

Given a particular i, BLI's can be detected based on calculation of stack distances. Denote them the level i BLI's. Clearly the BLI's at the same level are disjoint from each

other. As shown in the above figure, a level i BLI is properly contained within a level $i + n$ ($n > 0$) BLI that covers the same time point. The properness of this subset relationship is easy to prove using stack distance. The hierarchy of BLI's captures the hierarchical nature of task decomposition in sequential program design and development. In particular, the low-level BLI's contained within a same high-level BLI correspond to sub-tasks resulted from decomposing the same task. Associated experiments [MB76] on actual program behaviors confirm the existence of "phase-transition behavior". Moreover, the outermost level of BLI's captures the major phases of program execution, which covers the majority of program lifetime, with transitions between nearly disjoint activity sets. The inner levels, on the other hand, are shorter phases with relatively more overlapping activity sets. These results reflect the following intuition of program design: after the high-level task decomposition, there are fewer inter-dependencies between the resulted coarse-grained computations; in comparison, low-level sub-tasks are much more inter-related in terms of data references.

Following the above results, a program behavior can be modeled from two levels [DK75]: a macro model that captures the phase-transition behavior, and a micro model that specifies the reference patterns within phases. Experiments confirm that such a model is capable to mimic program behaviors with observed properties, e.g., in terms of re-producing the same type of LRU lifetime function [Bel69] curves. The modeling of phase-transition behavior is critical: without the macro model, it would be incapable of synthesizing program traces featured with the known properties.

The existence of phase-transition behavior has specific effect on the page replacement algorithms. The efficiency of such algorithms is mostly based on the locality principle, i.e., stable working sets and their associated long-life phases. A reference string with a high rate of phase transition is a phenomenon of 'anti-locality': the program is not

localizing its references around a subset of objects; rather it is trying to reference more objects within a small time interval. With the increase of phase transition rate, it would not be surprising that the responsiveness of replacement algorithms will largely degrade [Mas77]. Both caching and prefetching do not work efficiently as the working set of immediate future is no longer predictable based on that of the immediate past. Fortunately, as discussed above, disruptive transitions across disjoint locality sets are only between high-level major phases. Programs drift between low-level phases that have rather overlapping locality sets. This guarantees that replacement algorithms would work efficiently for most of the time during program execution.

Recurrence patterns like phase-transition behaviors also exist in web reference streams [ABC+96] and data streams [LCK+06], as termed "correlation structures" and modeled using stack distance self-similarity. In general, the locality of web references can be attributed to two reasons [Jin00a]: long-term popularity of frequently-requested web documents, and short-term correlation of concentrated requests. The former reflects the absolute closeness of references to objects that are popular over the whole reference string. The latter refers to the relative closeness of references that are localized only for a particular period of time. Obviously the latter is related to existence of recurrence patterns and is different from the former. Like phase-transition behavior in modeling program behavior, short-term correlation is critical to web reference characterization and is related synthetic trace generation. It can be measured by scaled stack distance [CC00], a stack distance normalized by the expected stack distance of the same object, assuming uniform distribution of references to different objects. The scaled stack distance is insensitive to popularity, and is hence used to recreate stack distance patterns that preserve short-term correlation. A comprehensive comparison of related work on modeling web references could be found in [Van05].

Traditional locality of reference, either to memory pages or to web documents, is about a sequence of occurrences that are ordered along an individual timeline. In distributed systems, processes and messages create partial ordering among relevant events, making traditional locality apparently no longer observable. However, the next chapter shows that in fact the locality principle still holds in terms of message interactions.

4.2 Origination of Interaction Locality

In general, a computation can be viewed as a set of operations ordered by data dependencies between each other [HP03]. When taken into implementation, it is usually first decomposed into a number of sub-tasks. As an application of the widely-used divide-and-conquer strategy, the decomposition is not performed arbitrarily. A major principle or guide line is to minimize the inter-dependencies among resulted sub-tasks, as well as to maintain an adequate number of sub-tasks. Intuitively, this helps proper management of the task structure, and has been considered as effective in many engineering fields including software engineering [ATG08]. Program design techniques, such as modularity, encapsulation, and data abstraction in object oriented programming [Weg90], provide full support to this principle. Consequently, the majority of data dependencies are hidden inside those sub-tasks until further decomposition is applied to them. The hierarchical nature of divide-and-conquer implies that there are always more fine-grained dependencies localized in the lower-level sub-tasks.

Control dependencies come into the picture when the sub-tasks are programmed in a sequential or parallel manner. In a sequential program, data dependencies are resolved via references to memory variables, and control dependencies are established to maintain the sequential ordering of memory references. A sub-task corresponds to a sequence of

memory references localized around a small data set. In addition, sub-tasks are sequentialized along time, and in most cases such sequentialization is performed hierarchically. In other words, a children sub-task is usually ordered before any successor of its parent task. As a result, a sequential program drifts its residency between children sub-tasks until their parent task ends. The corresponding working set hence changes slowly except at major task boundaries, and the phase-transition behavior therefore is only explicit at the outermost level of program execution. Since data dependencies are most localized at the lowest level of task granularity, traditional locality of reference is actually a phenomenon about localization of the finest-grained dependencies.

A distributed application, on the other hand, is a parallel solution in which sub-tasks are first parallelized and then distributed among groups of processes. Sub-tasks decomposed from the same task are usually assigned to the same process group. Each process is a sequential program, and hence has its own local data and control dependencies. In addition, there are two kinds of global dependencies established across different processes: the former is global data dependency resolved by variable reference messages, while the latter is global control dependency resolved by process synchronization messages. Such messages span sub-regions in both process space and time. In particular, each sub-task corresponds to a region that involves a number of processes as well as messages. Due to the hierarchical nature of task organization, sub-tasks belonging to the same parent task are managed adjacent to each other in space and time. Consequently, since fine-grained dependencies are more localized in low-level sub-tasks, regions corresponding to high-level tasks are relatively distinctive from each other as there are fewer messages in between. Moreover, each region involves a specific subset of processes that collaborate towards resolving the same task, and the membership of this subset remains quite stable until the end of region. A distributed execution hence exhibits

a '*region-transition*' pattern, which is quite similar to the phase-transition behavior observed in sequential programs. In other words, though each process still exhibits traditional locality of memory references from its local perspective, the locality principle also holds globally in terms of message interactions. Obviously, these two types of localities differ from each other, as one refers to recurrences of dependency pairs (i.e., messages) while the other refers to recurrences of individual dependencies. This thesis studies the former and uses the term *locality of (message) interactions*, or *interaction locality* to distinguish it from other locality notions.

In a distributed execution, global locality of interaction co-exists with local locality of reference. These two types of localities are properly nested in two different levels of the task hierarchy. In particular, reference locality is about memory references that resolve local dependencies, which are at the finest granularity and are only available within the lowest-level sub-tasks of individual processes (e.g., behavioral fragments as defined in Chapter 3.2.1). Interaction locality, on the other hand, is about messages that resolve inter-process dependencies, which are at a coarser granularity for maintaining the partial ordering between low-level sub-tasks from different processes. Usually in a distributed application, inter-process dependencies form only a small portion of all dependencies. Message interactions are hence much less frequent than memory accesses in individual processes. Unlike traditional locality that is observable almost in every program, interaction locality is only explicit in large scale distributed systems.

Locality of message interactions benefits largely from distributed system design and development techniques. Hierarchical task decomposition as well as modular component design is supported by popular distributed or parallel programming paradigms. For example, parallel architectural skeletons [GSP99] and synchronization skeletons [WLG04] support hierarchical composability. Parallel design patterns such as master-

slave, pipeline, divide-and-conquer can be used as building blocks to form larger patterns, from either architectural or behavioral aspects. This makes it much easier to parallelize a computation and manage the resulted sub-tasks in different levels of granularity. Also, agent role model [Ken00] and coordination protocols [DWK01] provide support for managing agent interactions in terms of communication patterns. Such patterns, namely blackboard, meeting, market, etc., can be constructed and implemented using small pre-specified communication routines termed agent interaction protocols [FIP03]. Each interaction protocol might involve several rounds of synchronization messages between two or more agents, for achieving a simple goal such as requesting, querying, contracting, brokering, etc. In addition, there are agent platforms and frameworks [BPR99, SSS+99, PPG00, MBS03] that provide development and runtime support to the above techniques. As a result, interaction locality becomes a popular property of distributed applications.

4.3 Identifying Interaction Locality

Interaction locality could appear in different ways within a distributed application. Intuitively, a locality of messages corresponds to a 'distinctive' space-time sub-region, where processes exchange messages more intensively with each other compared with outside. In an ideal case, a locality might have no external messages at all and hence is very easy to locate. Alternatively, there could be more or less inter-locality messages so that the locality boundaries become ambiguous and localities are difficult to distinguish from each other. It is obvious that a locality of messages indicates the coverage of a corresponding sub-task over space and time. In general, the above ideal case is rare and might only be observable at a high level of task granularity, when several independent sub-tasks are executed in parallel. The non-ideal case, on the other hand, is much popular especially for low-level sub-tasks, which are likely to be inter-coupled together towards

resolving a major task. Since more messages are hidden inside lower-level sub-tasks, the intensity of messages usually implies the locality granularity. Consequently, identifying lower-level localities will be more difficult and the result will be less accurate as well.

As discussed in Chapter 4.2, interaction locality can be pictured as a hierarchical region-transition behavior. For a given distributed execution, identifying localities is actually about finding the corresponding regions at different levels. Each region, in turn, is indexed by its hierarchy level, say i, and can be obtained from decomposing its parent region at level $i - 1$. Intuitively, this process can be applied recursively to a given distributed execution and all levels of localities will be hence identified. However, a distributed execution contains only finest-grained runtime information about processes and messages. Due to lack of design time knowledge about the hierarchy organization, it is not feasible to directly apply such a global-wise decomposition strategy. In particular, the decomposition at topmost levels is likely to incur great complexity, as it has to deal with large scale regions containing intensive messages and events. On the other hand, finding localities is different from just decomposing a distributed execution into a set of disjoint and complementary regions. For example, there could be a few asynchronous or random messages, which are not a part of the system design and hence do not belong to any locality at all. In other words, a set of properly identified localities need not cover all messages or events. Rather, they are natural localizations of messages corresponding to a set of by-design sub-tasks. From this perspective, the identification results are verifiable against the design information of the given distributed application.

Identifying localities of a given distributed execution is to abstract its hierarchical locality structure based on the plain message pattern it provides. It is more natural to follow a bottom-up approach, i.e., by starting from small locality entities and composing them into larger ones. For example, a distributed execution can be considered as initiated

by a set of special processes, say task initiators, whose first event is a send event. Each of these processes proceeds by spanning its own set of interaction partners and hence forms a locality of corresponding messages. In certain cases, two or more localities might merge into one when many enough messages are exchanged between them. A locality terminates after its task is accomplished, and the rest of distributed execution continues with new initiator processes towards forming new localities. The above procedure captures the formation as well as transition of localities from the perspective of individual processes. Within each locality, the involvement of a participating process forms a sub-sequence of localized interactions which are almost exclusively with other participants of the same locality. Due to the region-transition behavior between adjacent localities, such a sequence is usually distinctive by itself and hence can be identified locally. On the other hand, sequences from different participants of the same locality are often tightly coupled by message interactions, directly or indirectly. This makes it possible to find a locality via the coupling relations between individual sequences. In other words, these properties are likely to be preserved by distributed executions with good interaction locality, and can be used to guide the identification of localities.

This chapter evaluates the above idea and proposes models as well as techniques for constructing localities based on detection of individual locality entities. In particular, a notion of 'locality interval' is introduced to model a popular locality pattern of individual process interactions: a process interacts almost exclusively with a subset of recurring partners and its corresponding lifetime forms a distinct interval of localized interactions. Between locality intervals from different processes, an 'important coupling' relationship can be established based on the frequency of their message interactions. The transitive closure of important coupling then forms 'locality regions', each of which is distinct space-time sub-region that covers majority of its associated messages inside. On the other

hand, locality intervals are hierarchical in nature, and so are the locality regions. The bottom-up approach allows locality regions to be identified via controlling the creation of locality intervals at a certain level. Consequently, there will be hierarchical locality regions that are very likely to correspond to sub-tasks at different levels of the task decomposition. The rest of this chapter will use the term 'level' to refer to that of the by-design sub-tasks, as well as that of the corresponding locality regions or locality intervals to be identified.

4.3.1 Locality Interval

A distributed system consists of a set of processes that interact with one another only through messages. Each message m is a set of two events, a send event *send*(m) and a receive event *recv*(m), associated with a sender process and a receiver process respectively. Message events of a same process are totally ordered in virtual time [Den68], and each of them is associated with a specific *partner*. Message events from different process are partially ordered under Lamport's happened-before relation [Lam78]. A distributed execution is modeled as the partial order of all message events that occur during an execution of a distributed system. Within a distributed execution, the sequence of all message events associated with a process forms its *lifeline*. Similar to the notion of "reference string" in memory references, any sub-sequence of message events within a process lifeline is termed an *interaction string*. For simplicity, a distributed execution or an interaction string is represented by the corresponding set of involved events. Given an interaction string S_i of a process p_i in a distributed execution E, $S_i \subset E$ and $S_i(t)$ denotes its t^{th} message event (for any virtual time $t > 0$). Consequently, a sub-sequence of S_i that starts at $S_i(t_1)$ and ends at $S_i(t_n)$ is denoted as $S_i[t_1, t_n]$ ($t_n \geq t_1 > 0$).

A process lifeline often observes intervals of localized interactions with the following properties: i) recurrence: such an interval consists of many message events around a

small subset of partners and hence most events are recurrences; ii) migration: the set of (frequent) partners of an interval tends to change as the process lifeline proceeds, and consequently the process migrates from one interval to another; iii) hierarchy: a same time instance might be covered by different intervals observable at different granularities, each associated with a unique subset of partners. Such an interval can be demonstrated by the similarity between the recent past and the immediate future of a time instance.

Within an interaction string S_i, given $k = 1, 2, 3, \ldots$, a past window $W_-(t, k)$ at time instance t is defined as the maximum sub-sequence of S_i that ends at t and involves k partners. Similarly, a future window $W_+(t, k)$ at t is the maximum sub-sequence of S_i that starts from t and involves k partners. Denote the corresponding partner sets of $W_-(t, k)$ and $W_+(t, k)$ as $s_-(t, k)$ and $s_+(t, k)$, respectively. Note that by definition $W_-(t, k) \cap W_+(t, k) = S_i[t, t]$. Process p_i is said to have a *rank-k locality* at time t iff there exists $k' \geq k$ such that $s_-(t, k') \supseteq s_+(t, k)$. In other words, the immediate future starting at t will be all recurrences of a subset of k partners out of its k' most recent partners. The ranking k estimates approximately the number of frequent partners in this locality. Notice that $s_-(t, k'') \supseteq s_+(t, k)$ holds for any $k'' \geq k'$. However, there is a minimum $k^* \geq k$ such that $s_-(t, k^*) \supseteq s_+(t, k)$. The set of k^* past partners is unique as it defines the exact space of objects localized at t. It hence is called the *locality set* of this rank-k locality and is denoted as $l(t, k)$. Also, it has a continuous lifetime that forms a distinct *locality interval* within S_i.

Definition 4.1 (Locality Interval):

Given an interaction string S_i of process p_i, a locality interval $L(t, k)$ is the sub-sequence of S_i that is covered by the past window $W_-(t, k^*)$ and the future window $W_+(t, k)$ at time instance t, where p_i has a rank-k locality with a locality set of k^* partners.

Figure 4.4 shows an example interaction string S_1 of a process p_1, where each

message event is labeled as the corresponding partner. It is observable that p_1 has a rank-2 locality at $t = 5$, since $k = 2$ partners $\{B, D\}$ among its $k' = 3$ past partners $\{B, C, D\}$ have recurrences at the immediate future, i.e. in its future window $W_+(5, 2) = S_1[5, 7]$. Here $k^* = 3$ and $l(5, 2) = \{B, C, D\}$. The corresponding locality interval $L(5, 2) = S_1[3, 7]$.

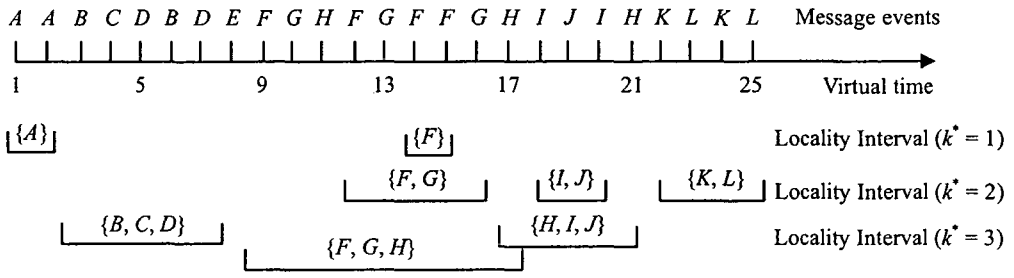


Figure 4.4 Hierarchy of all locality intervals in an interaction string S_1

By definition, there is a rank-1 locality at each time instance t of an interaction string S_i, whose locality set involves exactly the partner at t. The corresponding locality interval usually cover only a single event $S_i(t)$ and hence is called a *singleton* interval. However, it might not always be a singleton as the process could continuously interact with a same partner. The locality interval $L(1, 1) = S_1[1, 2]$ in Figure 4.4 is such an example. If process p_i has no rank-k locality at a certain time instance t, then $l(t, k) = \varnothing$. In Figure 4.4, there is no rank-3 or higher rank locality at $t = 5$, therefore $l(5, k) = \varnothing$ for any $k \geq 3$.

A locality interval usually consists of three stages: a prefix, a midfix, and a suffix. Given a locality interval $L(t, k)$ with a locality set of k^* partners, *prefix*$(L(t, k))$ is defined as the minimum sub-sequence that starts from the beginning of $L(t, k)$ and involves k^* partners. Similarly, *suffix*$(L(t, k))$ is the minimum sub-sequence that terminates at the end of $L(t, k)$ and involves k^* partners. The sub-sequence between the prefix and the suffix, if exists, is *midfix*$(L(t, k))$. As an example, the locality interval $L(11, 3)$ in Figure 4.4 consists of a prefix $S_1[9, 11]$, a midfix $S_1[12, 14]$ and a suffix $S_1[15, 17]$. Each locality interval has a *seed*, which is defined as the last time instance of its prefix. Since *prefix*$(L(t, k))$ involves all k^* partners of $l(t, k)$, any message event occurred after the seed is a

recurrence event, and the corresponding partner is predictable by $l(t, k)$. In general, the prefix captures the rising stage of a locality interval where partners begin to appear and form its locality set; the suffix captures the falling stage where partners begin to leave the locality set; the midfix captures the stable stage where all partners have recurrences at each time instance. It is possible that in some locality interval $L(t, k)$, its prefix might overlap with its suffix, e.g. $L(5, 2)$ in Figure 4.4. Such an interval is 'non-ideal' as some partners within $l(t, k)$ might have no recurrence at all. Hence in such a case $k^* > k$. The notion of "Bounded Locality Interval" [MB76] actually captures an 'ideal' locality interval where every partner must have recurrence after the seed. Therefore within such an interval $k^* = k$ always holds. Note that a bounded locality interval might still have no midfix, e.g. $L(23, 2)$ in Figure 4.4. Good locality of an interval is reflected by a small k^* that is close to k.

A locality interval $L(t_2, k_2)$ is said to be *contained* within another one $L(t_1, k_1)$ (written as $L(t_1, k_1) \supset L(t_2, k_2)$) iff $\forall\, t \in L(t_2, k_2)$: $t \in L(t_1, k_1)$. For example, given $k_1 > k_2 > 0$, a process could have localities in both rank-k_1 and rank-k_2 at a same time instance t. Consequently $l(t, k_1) \supset l(t, k_2)$, and $L(t, k_2) \supset L(t, k_1)$. In such a case, $L(t, k_1)$ is called a *child* interval of $L(t, k_2)$ and $L(t, k_2)$ is the *parent* interval of $L(t, k_1)$. In Figure 4.4, $L(18, 3) \supset L(18, 2)$ and these two intervals have the same seed t. It is also possible that a child interval might not cover the seed of its parent, e.g., in Figure 4.4 $L(11, 3) \supset l(13, 2)$ but $L(13, 2)$ doesn't cover the seed $S_1(11)$ of $L(11, 3)$. Given a process lifeline, each locality interval $L(t, k)$ with $l(t, k) \neq \varnothing$ can be identified. The containing relationship hence builds a hierarchy of all intervals in different granularities. It is worthwhile to note that some time instance t within an interaction string might only be covered by a singleton locality interval $L(t, 1)$. Figure 4.4 shows the hierarchy of locality intervals that exist in the interaction string S_1. For simplicity, all singleton locality intervals are omitted. It is clear

that the ranking of locality intervals is related to their hierarchy as well as their corresponding sub-tasks at different levels. For two locality intervals that one contains the other, a higher rank implies a higher level. However, this does not hold for all locality intervals. In fact, determining the level for a specific locality interval requires statistical information of its process lifeline and will be further discussed in Chapter4.3.

Two locality intervals *overlap* if they are not contained in each other but cover a same sub-sequence. For example, in Figure 4.4 $L(11, 3) \cap L(18, 3) = S_1[17, 17]$. In such a case, $L(11, 3)$ is called a *predecessor* and $L(18, 3)$ is a *successor*. It is easy to prove that the intersection of two overlapping locality intervals is a sub-sequence of the predecessor's suffix and a sub-sequence of the successor's prefix as well (otherwise it will involve some part of a midfix, and consequently one interval will contain all members of the other, which contradicts the definition of overlapping). An intersection is the transition stage between two consecutive locality intervals, where a process migrates its locality set from the predecessor to the successor.

4.3.2 Important Coupling and Locality Region

In a distributed execution, message interactions establish a *coupling* relation between interaction strings from different processes: an interaction string S_i of a process p_i is coupled with interaction string S_j from another process $p_j \neq p_i$ (written as $S_i \leftrightarrow S_j$.) if

(i) there exists a message m such that $send(m) \in S_i$ and $recv(m) \in S_j$ or vice versa, or

(ii) there exists an interaction string S_k such that $S_i \leftrightarrow S_k$ and $S_k \leftrightarrow S_j$.

Message coupling could also spread among multiple processes. As a result, the corresponding interaction strings and messages occupy a sub-region in space and time, which is called an *interaction region*.

Definition 4.2 (Interaction Region):

An interaction region of a distributed execution E is the set of message events within a

subset S of interaction strings in E where $\forall S_i, S_j \in S$: $S_i \leftrightarrow S_j$.

In other words, transitive closure of coupling relation leads to formation of interaction regions. Any two processes, say p_i and p_j, of an interaction region R should have corresponding interaction strings $S_i \subset R$ and $S_j \subset R$ such that $S_i \leftrightarrow S_j$. Call S_i, S_j as *components* of R. Consequently, for any message m coupling S_i and S_j, $m \subset R$ (m is hence called an *internal* message of R). In addition, it is possible that there exists some *external* message(s) coupling a component string with a non-component string, in which case the interaction region does not contain all associated messages. An interaction region R is called *atomic* iff $\forall$ m: $send(m) \in R \Leftrightarrow recv(m) \in R$, i.e., all message associated with an atomic interaction region are internal messages. Within a distributed execution, atomic interaction regions are all disjoint from each other in space and time, and there is no message coupling in between.

Locality intervals, which are interaction strings preserving specific locality properties, usually have coupling relations between each other. Transitive closure of the coupling between locality intervals forms atomic interaction regions. However, such regions hide too much information about interaction locality, and hence could not demonstrate the natural localization of messages. For example, in many distributed applications, there are system maintenance processes sending a few asynchronous messages to others from time to time. As a result, many processes have their locality intervals coupled together and form a large but somewhat loosely-coupled atomic interaction region. This however does not reflect the fact that there are small but more tightly-coupled groups of processes inside such a region. Also, there are processes which only participate in a task for a few rounds of negotiation and decide to turn to another task afterwards. Similarly, this leads to an atomic interaction region where a few messages couple two tightly-coupled groups that are performing two different tasks. All these facts show that within atomic interaction

regions there are fine-grained locality structures caused by the differences in coupling strength.

The strength of coupling between two locality sequences is directly related to occurrences of corresponding partners. Within a locality interval, partners have different importance in terms of their occurrence frequencies. In fact, observation shows that the distribution of their frequencies is usually bi-modal: there is a group of *important* partners each with many occurrences, among the rest of *occasional* partners each with much less occurrences. Non-recurring partners in 'non-ideal' locality intervals are examples of such occasional partners. Usually important partners are the major co-workers of a process that are persistently performing a sub-task, and they often appear as common partners of overlapped locality intervals. Occasional partners on the other hand could be either temporary co-workers that interact with a process for only a few rounds of handshaking, or some system maintenance processes that contact a process only through asynchronous messages. Such distinction between important and occasional partners reflects the hierarchical nature of interaction locality. In particular, important partners perform fine-grained intra-task interactions for accomplishing a sub-task, while occasional partners incur coarse-grained inter-task messages for sub-task coordination and management. In practice, identification of important partners can be based on detection of bi-modal distribution of interaction frequencies of partners, and is further discussed in the next chapter.

An *important coupling* relation can be established between two inter-coupled locality intervals: a locality interval S_i from process p_i has important coupling with locality interval S_j from another process $p_j \neq p_i$ (written as $S_i \rightarrow S_j$.) if $S_i \leftrightarrow S_j$ and p_j is an important partner of S_i. The directional relation of important coupling is usually symmetric and transitive. In other words, if $S_i \rightarrow S_j$, it is usually observable that $S_j \rightarrow S_i$;

if there is another $S_j \rightarrow S_k$, then it is likely that $S_i \rightarrow S_k$ also holds. Chapter 4.4.3 presents experimental results regarding such properties exhibited in real-life applications. These properties of important coupling come from the following fact: processes in a distributed application are usually designed to work together closely via massive interactions between each other for a designated sub-task. In turn, transitive closure of important coupling forms localized sub-regions of messages in space and time, which are called *locality regions*. The nature of important coupling implies that locality regions are *quasi-atomic* interaction regions, meaning there could be message interactions between such regions. However, since quasi-atomic locality regions are formed via important coupling, majority of messages associated with such a region is inside the region and there is only a small portion of external messages (i.e. those between locality regions). Such quasi-atomicity is an approximation of the 'ideal' atomicity, and its degree of closeness to atomicity reflects how messages are localized within such a region: the smaller percentage of external messages it has, the better interaction locality it exhibits.

4.3.3 Identification of Locality Regions

Given a distributed execution, its locality regions can be identified intuitively via a three-step procedure: i) detecting appropriate locality intervals within individual process lifelines; ii) identifying important partners of each locality interval; iii) composing locality intervals inter-connected via important coupling to form locality regions. However, in order to properly extract the hierarchical structure of locality regions that correspond to the by-design sub-tasks at different levels of decomposition, there are still quite a few detailed and yet subtle issues to be addressed: i) certain locality intervals contain very few recurrences and are hence not "good" enough to be usable; ii) there are cases in which important partners of a locality intervals are not easily distinguishable; iii) important coupling relations might be established between locality intervals that are

actually not in the same sub-task. This subsection discusses practical solutions for resolving these issues.

The notion of locality interval is generic as it captures a wide range of recurrence patterns. In particular, it is able to model the 'non-ideal' but also much popular case where non-recurring partners accidentally appear among other recurring ones. Obviously such a case is better considered as one locality entity, rather than several separated ones if modeled by the 'ideal' notion of BLI. On the other hand, there could also be non-ideal locality intervals with very few recurrences, which are certainly not locality patterns and should be filtered out. This leads to the qualification of locality intervals. Intuitively, any interval with better locality should have a large percentage of recurrence events, say at least 50%. The recurrence percentage hence can be considered as its 'quality', and is better as large as possible. A process lifeline preserving good locality should observe locality intervals with good (average) 'quality'. In addition, within the same process lifeline, high-rank locality intervals are supposed to have better 'quality' than low-rank ones. A locality interval with a 'quality' exceeding a given threshold is called a *qualified locality interval* (QLI). The threshold value might vary depending on the use of QLI's, and could be decided based on the quality distribution of a process's locality intervals.

Assuming a reasonable quality threshold is applied, each QLI should observe that the majority or even all of its interactions are recurrences of important partners. Each of these partners has a large number of occurrences, distinguishing itself from other occasional partners that have only a few occurrences. In other words, the occurrence of all partners within a QLI is likely to have a bimodal distribution, i.e., a mixture of two distributions with probabilities α and $1 - \alpha$ ($0 < \alpha < 1$), regarding important and occasional partners, respectively. Identification of important partners hence can be based on detection of such bimodality. Note that in certain cases, the mixture coefficient α might be too large to

produce a bimodal distribution, e.g., all partners have almost equal occurrences and hence are all important partners. In general, the notion of average stack distance measures the number of localized objects for a given reference string. Consequently, for a given locality interval, the number of identified important partners is expected to be close to its average stack distance.

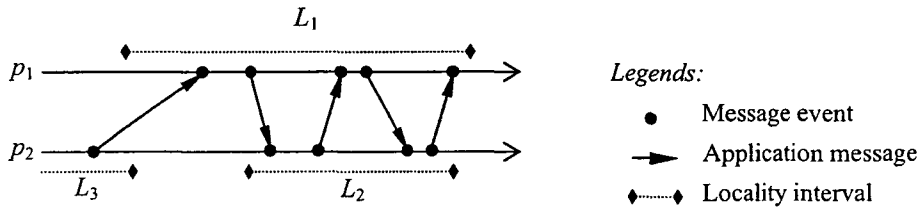


Figure 4.5 An example of mismatch between locality intervals

Important coupling relations can be established between inter-coupled QLI's after identification of their important partners. Subsequently, locality regions can be formed via the transitive closure of important coupling. Meanwhile, symmetry and transitivity of important coupling can also be checked, and such properties are expected to hold in most cases. However, since detection of locality intervals and identification of important partners are all local decisions, it is possible that an important coupling relation is established between two locality intervals that have 'mismatch' in terms of message correspondence. Figure 4.5 shows such an example where a QLI L_1 of process p_1 is coupled with two QLI's L_2 and L_3 of its important partner p_2. Obviously in such a case, the coupling relation $L_1 \rightarrow L_3$ should be omitted as there is only one message between them and most messages between p_1 and p_2 corresponds to $L_1 \rightarrow L_2$. In general, there are two reasons that might cause such a mismatch situation: i) the QLI's are from different levels of task granularity and hence the messages between them correspond to different percentages of occurrences within each other; ii) the QLI's are from same or similar levels of task granularity but are coupled with infrequent messages. The former case requires choosing locality intervals at proper levels, while the latter case requires filtering

inappropriate coupling relations.

The 'level' of locality intervals can be determined by the number of important partners. In general, sub-tasks at different levels have hierarchical space-time coverage, and their corresponding locality regions involve different sets of participating processes. As a result, hierarchical locality intervals of a same process differ in their numbers of important partners. Such a number hence can be taken as a parameter, and varying its value will help selecting locality intervals at different levels. Note that it is different from the ranking of locality interval discussed in Chapter 4.3.1, as the former is more meaningful and needs to be decided statistically. Among all values of this parameter, the one corresponding to the top-level locality intervals is important, as it decides the actual lower bound for all locality intervals of a process. In practice, this value can be approximated by the average stack distance of the given process lifeline.

Important coupling relations between QLI's can be filtered based on their relative coupling strength. Intuitively, a 'strong' enough important coupling relation is expected to involve the majority of all message interactions that are contributed by an important partner. Given QLI's S_i from p_i, S_j from p_j, and their important coupling relation $S_i \rightarrow S_j$, define its coupling strength as μ_{ij}/ρ_j, where μ_{ij} is the number of messages involved in $S_i \rightarrow S_j$, and ρ_j is the number of p_j's occurrences in S_i. Similar to the qualification of locality intervals, there should be a strength threshold for qualifying important coupling relations. More specifically, if the coupling strength of $S_i \rightarrow S_j$ exceeds a given threshold, say 50%, it is called a *primary important coupling* (PIC) of S_i. This notion captures the essence of interaction locality and is critical for forming locality regions. However, some QLI's, especially those at lower levels, might not have primary important coupling at all. This could be caused by two reasons: i) the probability of 'mismatch' between locality intervals usually increases at low levels; ii) the deviation of coupling strength could be

enlarged due to small sampling of occurrences in low-level QLI's. To avoid the above ambiguity issues, it is better to start calculating and determining the primary important coupling of QLI's at higher levels. Information about primary important coupling at higher levels will be used to guide the determination process at lower levels. Such a top-down strategy will eliminate the fine-grained errors caused by low-level randomness. As mentioned above, for a given process lifeline, the topmost level to start with can be measured by its average stack distance.

In summary, locality regions of a distributed execution can be identified recursively in a top-down manner. To start up, the topmost level locality intervals can be addressed using the average stack distance of each process lifeline. Qualified locality intervals (QLI's) can identify their important partners by detecting the bimodal distribution of partner occurrences. Subsequently, the primary important coupling of QLI's can be determined. Locality regions can be formed by QLI's inter-connected via primary important coupling. Each resulted region then will be treated as a distributed execution and be applied with the above procedure to identify low-level locality regions.

4.4 Interaction Locality in Distributed Applications

As discussed previously, interaction locality is actually the locality of dependencies resolved by message interactions, and is likely to be exhibited in many distributed applications. Related locality entities, including locality intervals and locality regions, can be identified either locally or via frequent message coupling. These analysis, notions and strategies are better justified with real-life applications. Important issues to be checked include the general availability of interaction locality, the effectiveness of their identification approach, and the statistical characteristics of identified locality entities.

This chapter presents experimental studies on various distributed applications, for

evaluating and verifying the above-mentioned notions as well as strategies. In particular, justifications are prepared on four aspects of interaction locality, i.e., the traditional locality views of individual process lifelines (in terms of stack distance), identification of locality intervals as well as their important partners, formation of locality regions formed via primary important coupling of QLI's, and characteristics of their coverage at different hierarchical levels. Test cases are designed accordingly and experiments are conducted on two example distributed applications. Analysis on experimental results confirms the feasibility of identification approach as well as the existence of interaction locality.

4.4.1 Hypotheses and Test Cases

Analysis on the origination of interaction locality implies that interaction locality is the intrinsic property of distributed applications. Consequently, it leads to a major hypothesis about the hierarchical region-transition behaviors of distributed applications, which can be further interpreted from the following perspectives.

Hypothesis 4.1 (Global view):

A distributed execution has majority of its messages contained within a set of top-level interaction regions that are disjoint and distinct from each other in space and time. Each region itself also has a similar internal structure involving a set of sub-regions at a lower level, and such a hierarchical structure could be observed at several levels.

Hypothesis 4.2 (Region-wise view):

Such an interaction region consists of one interaction string from each involved process. Easy pair of interaction strings is inter-coupled directly or indirectly via frequent messages. The interaction region covers a major portion of all its associated messages.

Hypothesis 4.3 (Individual process view):

Each process lifeline has majority of its message events contained within a set of top-

level distinct interaction strings. Each interaction string itself also has a similar internal structure involving a set of interaction strings at a lower level, and such hierarchical structure could be observed at several levels.

Corollary 4.4 (Traditional locality view):
Each process lifeline exhibits traditional reference locality in terms of message interactions.

Following the notions and strategies discussed in Chapter 4.3, a step-by-step process can be used to justify each of the above hypotheses. In particular, good traditional reference locality can be measured by a small stack distance for a given process lifeline. The hierarchical distinct interaction strings within a process lifeline can be identified as qualified locality intervals (QLI's). Locality regions formed via primary important coupling of QLI's are good candidates for the disjoint and distinct interaction regions of a given distributed execution. Hierarchy of locality regions can be identified and characterized recursively using the top-down approach discussed in Chapter 4.3.3. The following test cases are designed as an implementation of this justification process, whose success will also prove the effectiveness of the identification approach.

Test case 1 (Reference locality):
The objective of this test is to verify the existence of reference locality within individual process lifelines of distributed applications. Given a process lifeline, the stack distance of each time point is calculated. Statistics on these stack distance values should observe a small average stack distance, and a large percentage of time points that have a small stack distance value.

Test case 2 (Locality intervals):
This test is prepared to detect hierarchical locality intervals of individual process lifelines.

Given a process lifeline, its hierarchy of locality intervals is constructed in a top-down manner. In other words, the topmost level QLI's are first detected and then are considered as input interaction strings for detecting the next level QLI's. The average stack distance of the given process lifeline is used as index value of the topmost level QLI's, i.e., the lower bound on the number of their important partners.

At a certain level, locality intervals are detected one after another, by the following steps: i) the detection process scans the given process lifeline from its beginning for the first interaction string that involves many enough partners (with respect to the given lower bound); ii) if this interaction string is a qualified locality interval, its important partners will be identified and checked against the lower bound; iii) if the number of important partners exceeds the lower bound, the detection process will move forward to the next locality interval that possibly overlaps with the current one, i.e., the start of its suffix; iv) otherwise, it will try to include more time points and check if there is a larger qualified locality interval; v) the above steps will repeat till the end of process lifeline.

Statistics on the resulted locality intervals should have the following observations, especially at the top levels: i) there is usually a small percentage of un-qualified locality intervals; ii) majority of time points within the given process lifeline is covered by locality intervals; iii) qualified locality intervals have a large percentage of time points associated with important partners.

Test case 3 (Locality regions)

This test aims at composing qualified locality intervals via their primary important coupling relations to form the top-level locality regions. The qualified locality intervals detected in test case 2 are directly used and the primary important coupling relations are identified. Results should show that a large percentage of messages are captured by primary important coupling, and quite a few of such relations are symmetric or transitive.

Regarding locality regions formed following these relations, there should be a large ratio of internal messages for each region, and a majority of all messages covered by regions.

Test case 4 (Hierarchical locality regions)

This test studies the hierarchical structures of locality regions identified at different levels of granularity. A recursive identification approach is used from top-down, by which the topmost level locality regions are identified first and then are used as input interaction regions for the next round of identification. Results should show that locality regions at a higher level have larger ratios of internal messages as well as larger coverage of all messages.

4.4.2 Application Traces

Performing the previously-proposed test cases requires trace data to be collected from real-life distributed applications. Trace data here refers to the static information about a complete distributed execution, including all process lifelines and all message interactions. Since all locality entities and strategies are based on virtual time of message events, collecting trace data is actually recording the messages events of a distributed execution and tracking their partial order relations. This chapter evaluates two example distributed applications and collects trace data from their executions in different scales. These two applications belong to different application domains: one is an e-commerce application and the other is an agent-based manufacturing process. Both applications are based on agent role model [Ken00] and have implementations on a popular agent development platform JADE [BPR99]. As an intrinsic property of distributed applications, interaction locality is expected to be application domain independent and should be observable in both applications. The characteristics of these two applications are described briefly below.

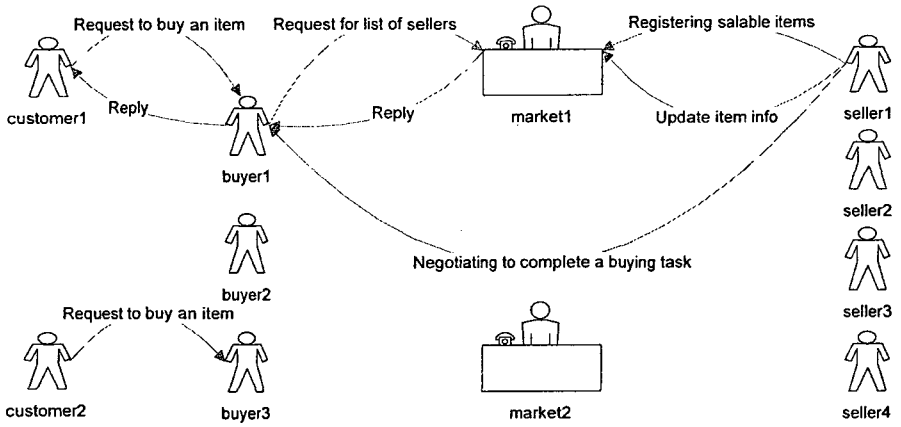


Figure 4.6 Work flow of agent-based e-commerce application

The e-commerce application is designed for buying and selling products electronically with the help of software agents. There are four different types of agents: customer, buyer, market and seller. Figure 4.6 shows the typical interactions between these agents via an example work flow. In general, a buying task is initiated by a customer agent when it sends a request to a buyer for buying a specific item. To fulfill the request, the buyer communicates with one or more market agents and requests for a list of potential sellers of the specific item. Each market maintains updated seller information about different items and also replies query requests from buyers. Upon receipt of the seller list, the buyer negotiates with one or more sellers on the list to complete the buying task and then return the result to the customer. A buyer may also initiate an English or Dutch auction with a group of sellers. In this application, the number of agents for each type is variable and can be fed as parameters during the initiation of the application. The interaction between agents could be carried out via individual messages or using fundamental FIPA interaction protocols [FIP03]. In addition, a seller agent can participate in multiple protocol sessions concurrently.

The other application is an automation of the entire manufacturing process of a production house. The house business starts with acceptance of product orders from customers. The order is then furnished in the manufacturing plant and then products are

delivered to the customer. Operational entities in the product line are modeled as software agents who perform one or more specific tasks. Two major processes of the system are order processing and manufacturing. A typical order processing scenario involves the following types of agents: customer, sales and sales manager. Initially, a customer agent makes an order for a list of products to one of the sales agents. The sales agent forwards it to the sales manger for approval and replies to the customer accordingly after getting a reply. A manufacturing process usually involves maker agent, bin agent, packer agent and product line supervisor agent. A maker agent takes raw material as input to make products and puts the furnished products into a dedicated bin agent. The packer agent picks products from the bin, packs them and updates the finished product stock. The line supervisor agent monitors the empty status of the bin and schedules it between makers and packers. In this application, the number of assembly lines varies depending on the market demand. The association of a maker, packer or bin to a specific assembly line is dynamic. The number of agents is also variable and can be fed as an input parameter during the application initiation. Figure 4.7 shows the work flow of these two processes.

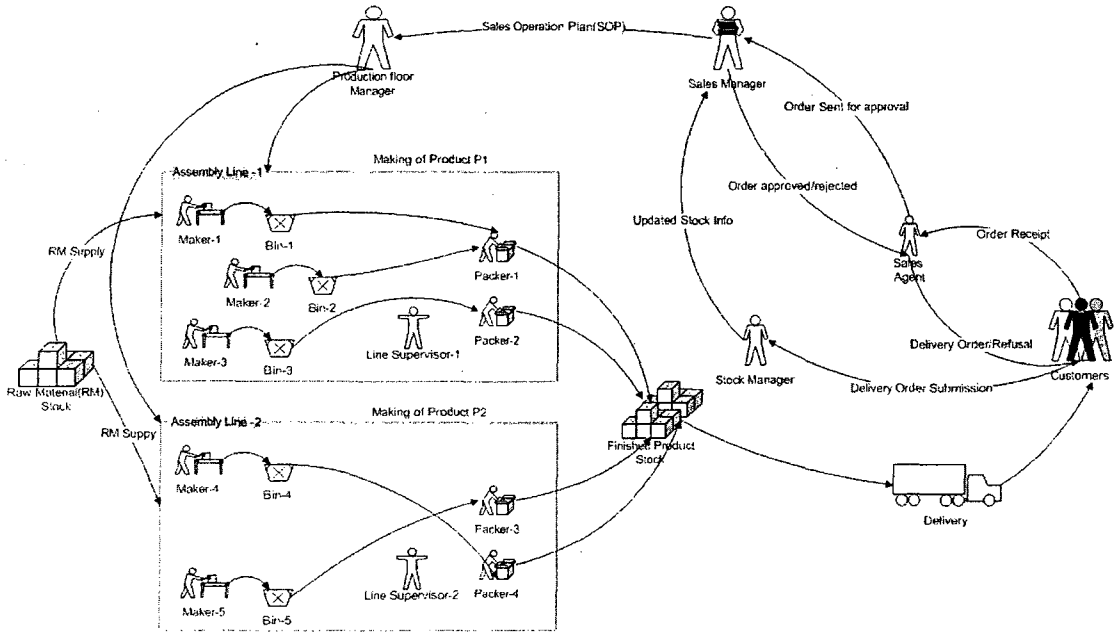


Figure 4.7 Work flow of agent-based manufacturing process application

The above work flows show clear sign of interaction locality in these two applications. A top-level locality is usually organized around a major task, e.g., a buying task in the former, and an order processing or manufacturing process of a product in the latter. Agents form groups and group members cooperate with each other to accomplish their specific tasks. Such cooperation incurs intensive message interactions that are almost exclusively localized between group members. In addition, within a major task there could be sub-tasks carried on in parallel, e.g., different product lines of a manufacturing process for a same product, forming localities in a lower level. Such knowledge can be used to verify the identification of locality regions for the corresponding application.

4.4.3 Experimental Results and Analysis

Based on the design of test cases in Chapter 4.4.1, experiments have been conducted accordingly on the two example distributed applications. Trace data has been collected on their executions in different scales, which are obtained by varying their parameters for controlling the numbers of agents. Table 4.1 shows the details on the number of agent as well as messages involved in the two applications. Unless stated otherwise, experimental results in the following sub-chapters are based on the medium-scale executions.

Application	e-commerce			manufacturing process		
Scale	small	Medium	large	small	medium	large
No. of all agents	17	22	30	13	23	61
No. of all messages	636	2231	1287	533	1887	3248

Table 4.1 Information about two example applications in different scales

4.4.3.1 Test Case 1: Exhibition of Traditional Locality

This test gives a traditional locality view on individual process lifelines within distributed applications. Stack distance is used as a measured of traditional locality and is calculated over lifelines of all processes of each execution.

a) An example process lifeline: buyer1 from e-commerce application

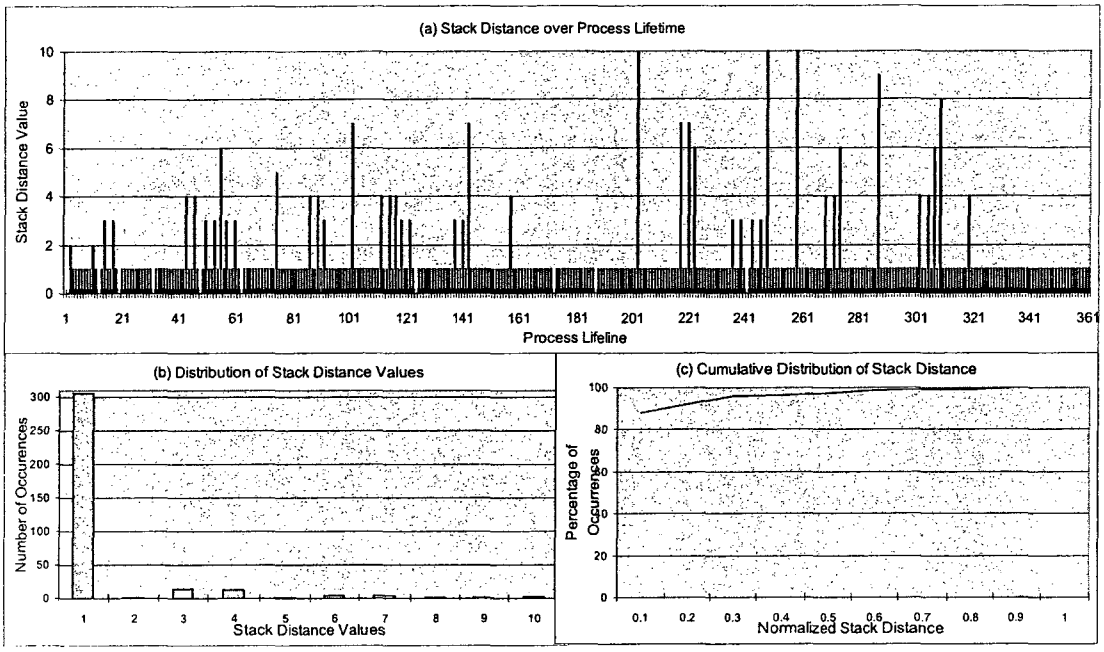


Figure 4.8 Stack distance distribution for buyer1 of e-commerce application

A buyer agent is a typical process in the e-commerce application. Figure 4.8 (a) shows the stack distance values along the lifeline of such an agent, buyer1, which interacts with a total of 11 other agents. For most of the time, its stack distance remains as 1, meaning it almost interacts with only one partner. Statistics on its stack distance values in Figure 4.8 (b) also leads to the same conclusion. In addition, Figure 4.8 (c) plots the cumulative distribution of its stack distance, where the X-axis represents the normalized stack distance values and Y-axis represents the percentage of occurrences of corresponding values. It concludes that for most of the lifetime, buyer1 keeps interacting with one of its recent partners, which is consistent with its average stack distance value of 1.46.

b) Traditional locality of all agents in e-commerce application

Figure 4.9 (a) shows the values of average stack distance as well as the overall partner set size for all agents in the e-commerce application. Most agents have good traditional locality represented by a small average stack distance (around 2), even though they might

have quite different partner set sizes. On the other hand, the curves for their cumulative distribution of stack distance show noticeable differences in Figure 4.9 (b). This figure uses a simplified naming scheme for agents: b1 for buyer1, c2 for customer2, m1 for market1, s3 for seller3, etc. It is clear that many agents have their majority of interactions, say 80%, localized with a small subset of all partners (represented by a small value of normalized stack distance). A few other agents, however, distribute their interactions with a wide range of partners. For example, c1, c2, and m1 require nearly 40% of partners in order to cover 80% of interactions. This might be due to different types of reasons. In the e-commerce application, customer agents only talk to buyer agents using a very small number of interactions. In such a case, even their interactions are quite localized with just a few buyers, they do not have good curves for cumulative distribution. Market agents, on the other hand, interact with almost every agent except customers. Consequently, the curves for their cumulative distribution of stack distance do reflect the lack of traditional locality in their process lifelines.

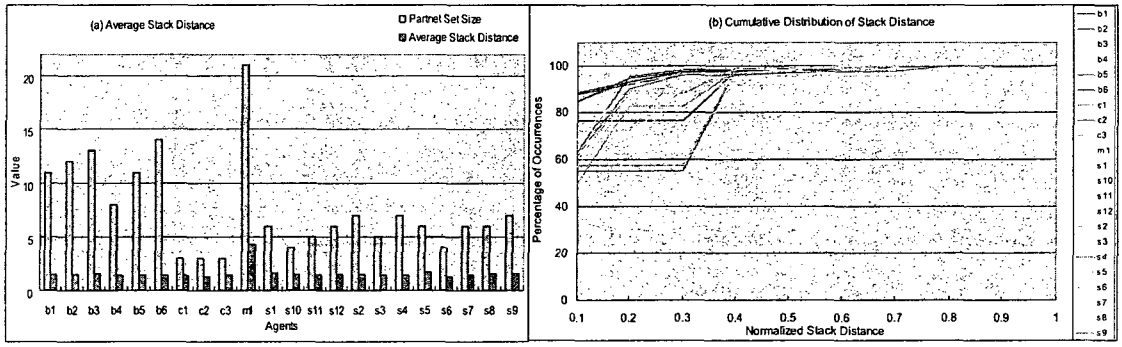


Figure 4.9 Stack distance distribution for all agents of e-commerce application

c) Traditional locality of all agents in manufacturing process application

Figure 4.10 shows that the manufacturing process application has similar traditional locality properties as the e-commerce application does. Most agents have a very small average stack distance as well as a very good curve of cumulative distribution. A few others, including ls1, sa1 and sa2, have a relatively large average stack distance and also

require a large percentage of partners to cover majority of their interactions. In this application, these names stand for line supervisor 1, sales agent 1, and sales agent 2, respectively. Due to the roles they are playing, such agents need to interact with many agents and hence do not have very localized interactions.

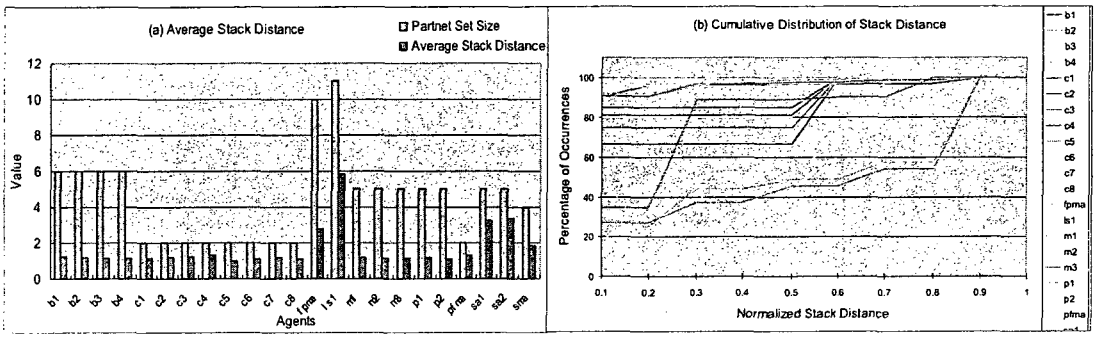


Figure 4.10 Stack distance distribution for all agents of manufacturing process application

4.4.3.2 Test Case 2: Detection of Locality intervals

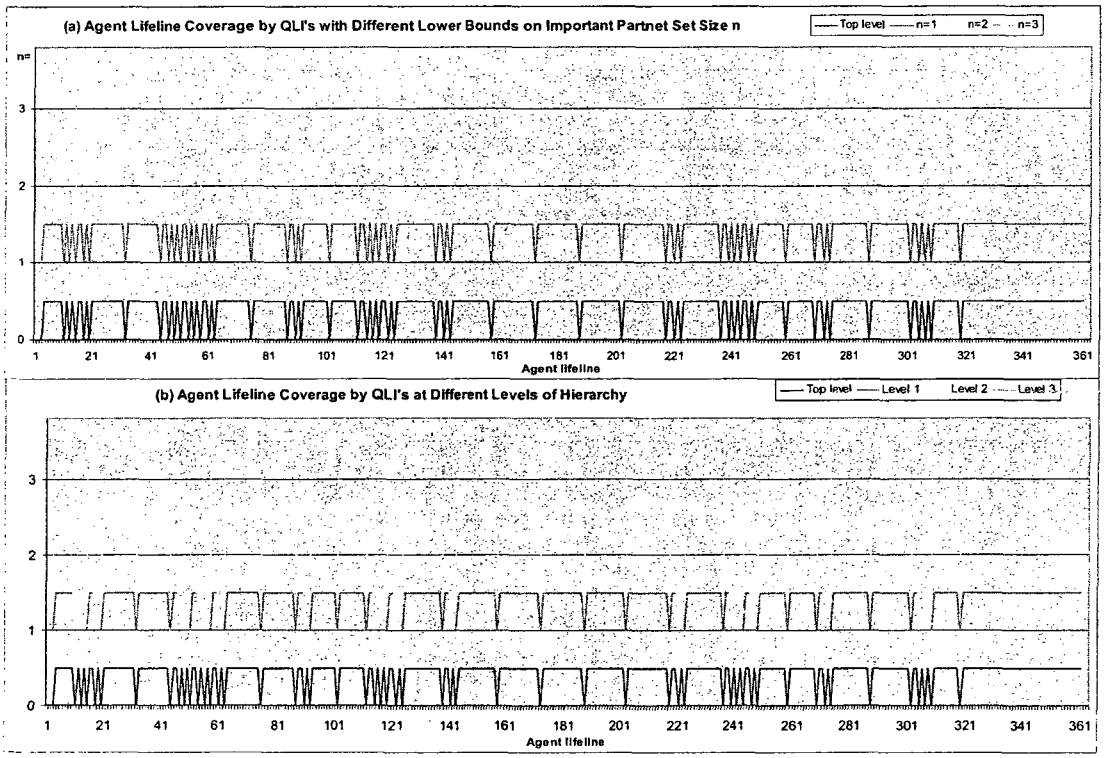


Figure 4.11 Properties of QLI's detected in buyer1's lifeline of e-commerce application

It is clear that in both applications there are two types of agents in terms of their locality

of interactions: i) a large number of 'simple' agents that play one role only and have simple lifelines with quite localized interactions; ii) a few 'popular' agents that play multiple roles simultaneously and have discreet interactions with many other agents. As a result, agents of the former type usually have a small average stack distance and their lifelines often involve only one level of locality intervals. Agents of the latter type have a large value of average stack distance, and their lifelines contain several levels of locality intervals that form a hierarchy.

a) Locality intervals at different levels

Given an agent lifeline, its top-level locality intervals can be detected by varying the lower bound of important partner set size n. Figure 4.11 (a) shows the coverage of qualified locality intervals (QLI's) detected in a 'simple' buyer1 agent lifeline of the e-commerce application, by setting n = 1, 2, 3 and a criteria of 50% for qualification of locality intervals. Given the fact that buyer1 has an average stack distance of 1.46, n = 1 is the optimum choice towards forming top-level QLI's. The resulted locality intervals correspond to its simple lifetime periods and they also form a perfect coverage of its whole lifeline. Figure 4.11 (b) shows the coverage of QLI's detected in different levels of hierarchy, i.e., by recursively applying the detection procedure to each level-n QLI in order to obtain the level-n+1 QLI's. In this figure, all sub-levels of QLI's fall into the same and they are no much difference from the top-level ones, which are already a perfect picture of buyer1's simple activities. As a comparison, Figure 4.12(a) shows the coverage of QLI's detected in a 'popular' market1 agent lifeline of the same application (with an average stack distance of 4.26). The lower bound n varies from 1 to 8, and again a choice of n = 4 based on its average stack distance makes the optimum coverage of lifeline. On the other hand, Figure 4.12(b) shows two levels of QLI's that are at finer granularities. Statistics on QLI's detected in the manufacturing process application show

similar conclusions about the distinction of 'simple' and 'popular' agents.

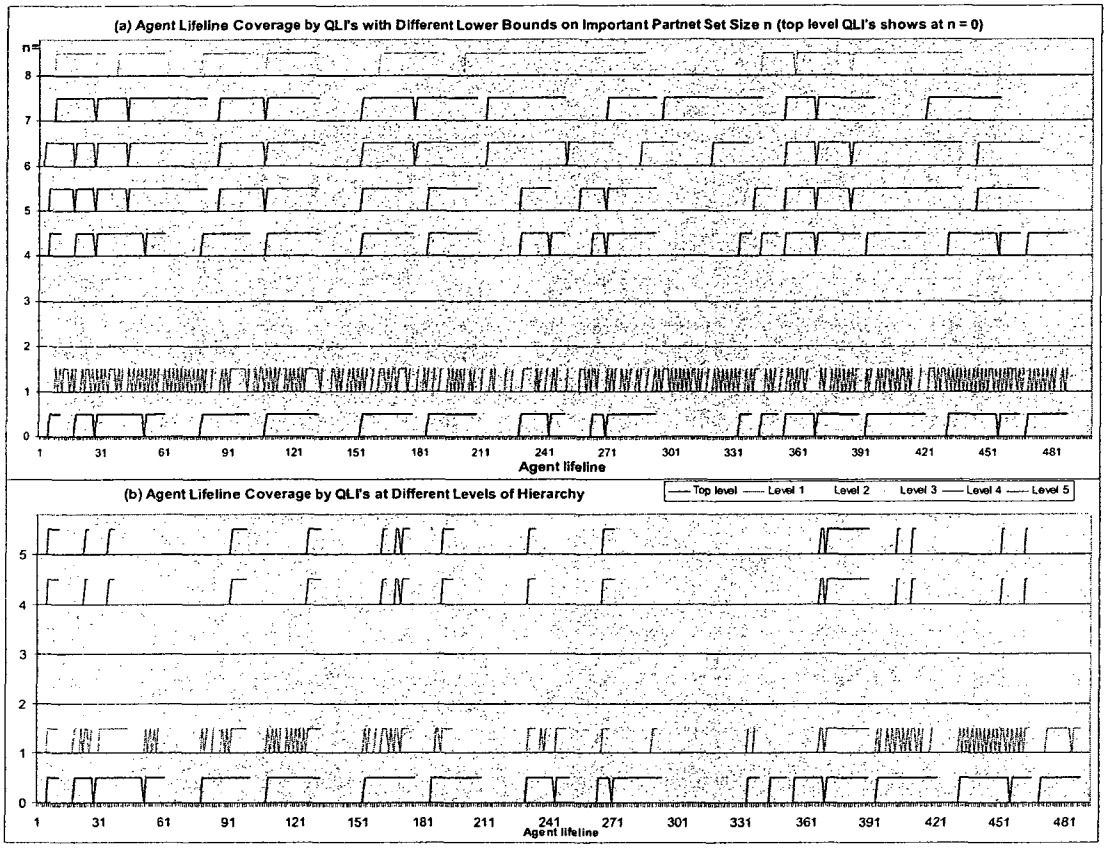


Figure 4.12 Properties of QLI's detected in market1's lifeline of e-commerce application

b) Top-level locality intervals

Both of the above agents, buyer1 and market1, have quite large coverage of their lifelines by top-level locality intervals hence detected. Figure 4.13(a) shows a statistic of overall QLI coverage for each agent lifeline of e-commerce application. It is not surprising that many other agents also have majority of their lifelines covered by top-level locality intervals. In addition, most events hence covered are associated with important partners of corresponding locality intervals. Note that a 'popular' agent m1 has the worst coverage percentage (70%) compared to others (above 80%). Figure 4.13(b) plots a curve for every agent, which shows the percentage of agent lifelines covered by the top-level QLI's that exceed a given length. It can be observed that customer agents usually have a small

length limit for their locality intervals (5 or 6), while seller agents have a moderate length limit (around 15), and buyer agent usually have very large locality intervals (with a length of 40 or above). All these agents follow a similar pattern: their curves drop quickly as the locality interval length reaches a certain boundary value, meaning majority of their lifelines is covered by locality intervals that are at least as large as what its boundary value specifies. Such values vary with different types of agents, e.g. 3 for customer agents, 11 for buyer agents and 4 for seller agents. Some buyer agents have a long tail in their curves, indicating the existence of a single large locality interval. In contrast, the 'popular' agent m1 (which is the only market agent) has a different curve that changes slowly as the locality interval length increases, indicating that the corresponding locality intervals have a wide range of length distribution.

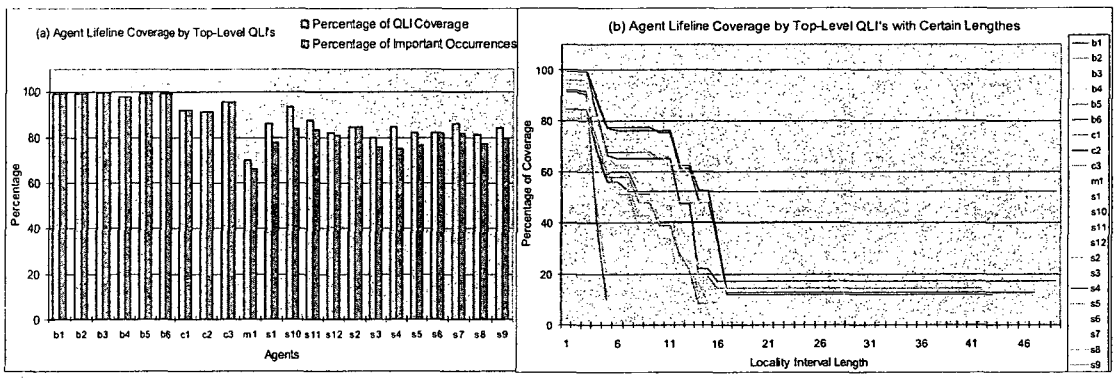


Figure 4.13 Properties of top-level QLI's from agent lifelines of e-commerce application

Observation on locality intervals of the manufacturing process application leads to a similar conclusion. As shown in Figure 4.14(a), most agents have large lifeline coverage ratios by their top-level locality intervals, except a 'popular' line supervisor agent ls1 (which has less than 60%). Besides, there is no lifeline coverage for two sales agents, sa1 and sa2. This is due to the fact that sales agent has a very short lifeline in which they interact with a wide range of other agents, and as a result, no top-level locality interval is detected. From Figure 4.14 (b), it is clear that agents in this application fall into three

categories: i) customer agents that have simple short lifeline with small locality intervals; ii) bin, maker and packer agents that have a simple long lifeline with large locality intervals; iii) finished product manager, line supervisor, and sales manager agent that have a long complex lifeline with large locality intervals. The first two categories are formed by 'simple' agents with much localized interactions, while the last category consists of 'popular' agents that interact with many other agents.

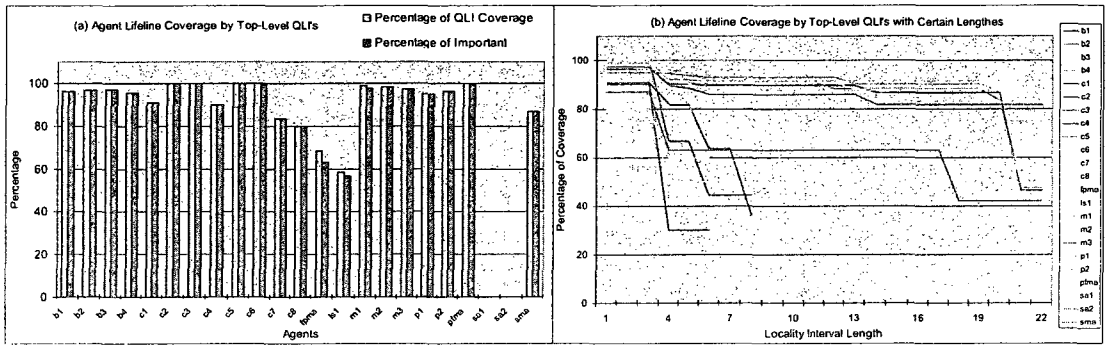


Figure 4.14 Properties of top-level QLI's from agent lifelines of manufacturing process application

4.4.3.3 Test Case 3: Identification of Locality Regions

This test forms locality regions by composing QLI's via primary important coupling relations. The top-level QLI's obtained previously are used, and as a first step, primary important coupling is detected between them.

a) Primary important coupling between QLI's

Primary important coupling plays a critical role in connecting locality intervals from different processes. Given a threshold of 50%, such relations are established between each pair of QLI's. For each QLI and each agent lifeline, a large percentage of their message events are expected to be associated with primary important coupling relations. Figure 4.15(a) shows the percentage of such association as per agent lifeline for the e-commerce application. Most 'simple' agents appear to have a majority of their lifelines associated with primary important coupling, i.e., 70% - 80% for sellers and customers,

and 60% for buyers as shown in Figure 4.15(a). In comparison, the 'popular' agent market1 only has a coverage percentage of 14%. Figure 4.15(b) shows the distribution of the association percentage values as per locality interval, where the X-axis represents the values and the Y-axis represents their distribution. From this figure, more than 70% of QLI's have a full coverage by such coupling relations, while around 10% of QLI's have no primary coupling association at all. As a matter of fact, the former type of QLI's is mostly from 'simple' agents and the latter is from market1. The manufacturing process application has similar statistics, as shown in Figure 4.16. Besides, 'simple' customer agents (i.e., c1 – c8) in this application have small coverage (around 30%) that is comparable to 'popular' agents such as line supervisor 1 (ls1) and the stock manager agent (sma).

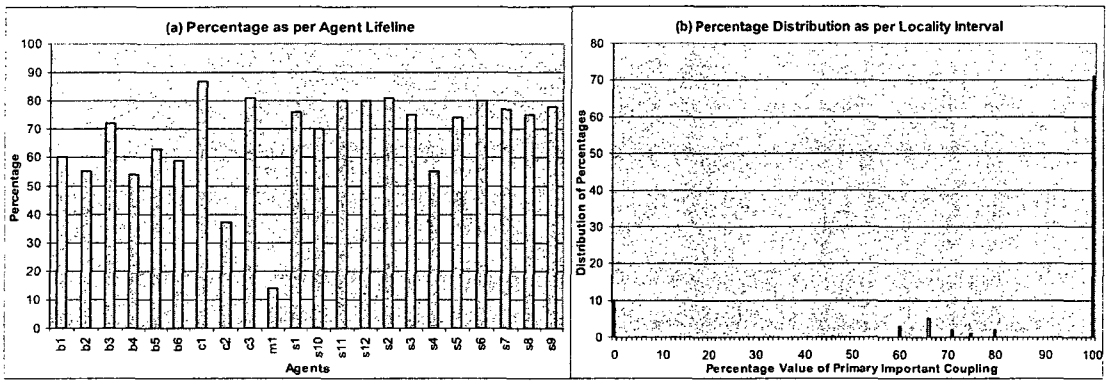


Figure 4.15 Event association with primary important coupling for e-commerce application

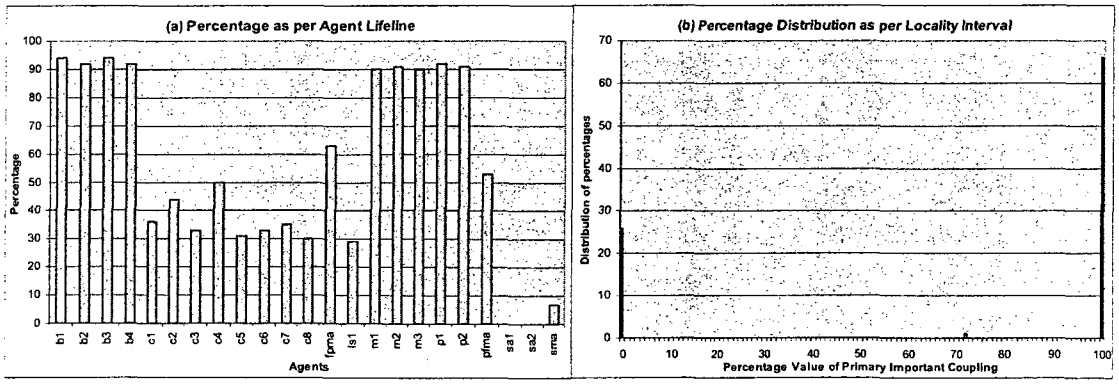


Figure 4.16 Event association with primary important coupling for manufacturing process application

In overall, 70% of all QLI events from all agents of the e-commerce application are associated with primary important coupling. For the locality regions formed following such coupling, 36% and 49% of all internal messages are associated with symmetric and transitive primary important coupling, respectively. In comparison, the manufacturing process application has relatively higher ratios. Table 4.2 summarizes the corresponding results for the two applications in different scales.

Application	e-commerce			manufacturing process		
Scale	small	medium	large	small	medium	large
Ratio of QLI events associated with primary important coupling	69%	70%	69%	93%	89%	87%
Ratio of internal messages associated with symmetric primary important coupling	28%	36%	32%	84%	77%	73%
Ratio of internal messages associated with transitive primary important coupling	43%	49%	47%	84%	79%	74%

Table 4.2 Statistics on primary important coupling in different scales

b) Properties of top-level locality regions

Application	e-commerce			manufacturing process		
Scale	small	medium	large	small	medium	large
No. of locality regions	18	120	42	26	81	135
No. of all messages associated	406	1859	1107	482	1612	2891
Average No. of messages associated per locality region	22	15	26	18	19	21
Overall ratio of internal messages	55%	72%	79%	86%	81%	80%

Table 4.3 Statistics on locality regions in different scales

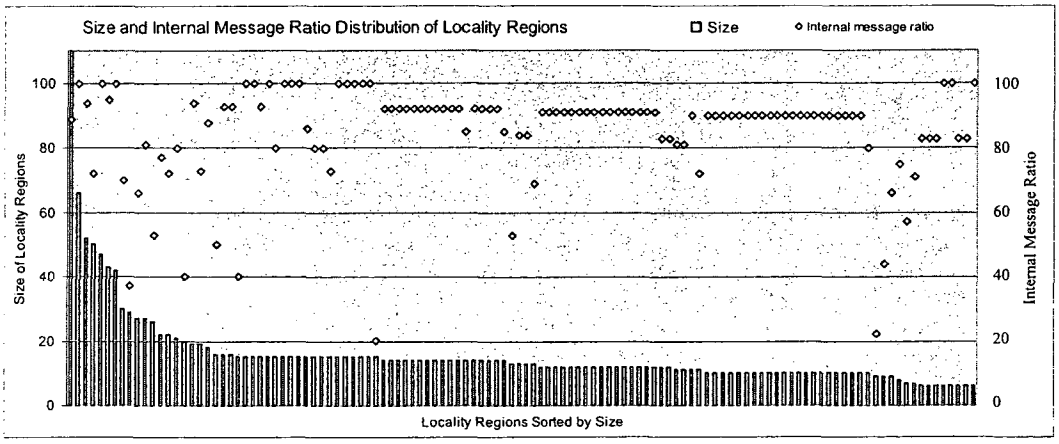


Figure 4.17 Size and internal message ratio of locality regions detected in e-commerce application

Quasi-atomic locality regions are formed as transitive closure of primary important coupling relations. In a medium-scale execution of the e-commerce application, there are 120 locality regions detected, covering 72% of all 2231 messages as their internal messages. In the manufacturing process application, 81 regions cover 81% of all 1887 messages. Table 4.3 gives the detailed statistics of such results. In most cases, especially for large-scale executions, majority of messages are covered by top-level locality regions identified via QLI's and their primary important coupling.

The resulted locality regions vary in different sizes (in terms of number of messages associated) and also have different ratios of internal messages. As shown in Figure 4.17, locality regions in the e-commerce application have a large range of distribution in both aspects. Majority of these regions have a high internal message ratio (above 80%). On the other hand, there are a large number of small regions (with a size less than 20), together with a few large ones (with a size above 20). Each region corresponds to a buying task, which by design could be either large or small, depending on the parameters specified. As shown in Figure 4.18, the distributions for the manufacturing process application are much simpler. Most regions are of the similar size (around 20) and have the perfect internal message ratio of 100%. This is due to the fact that most of such manufacturing tasks are in regular shape.

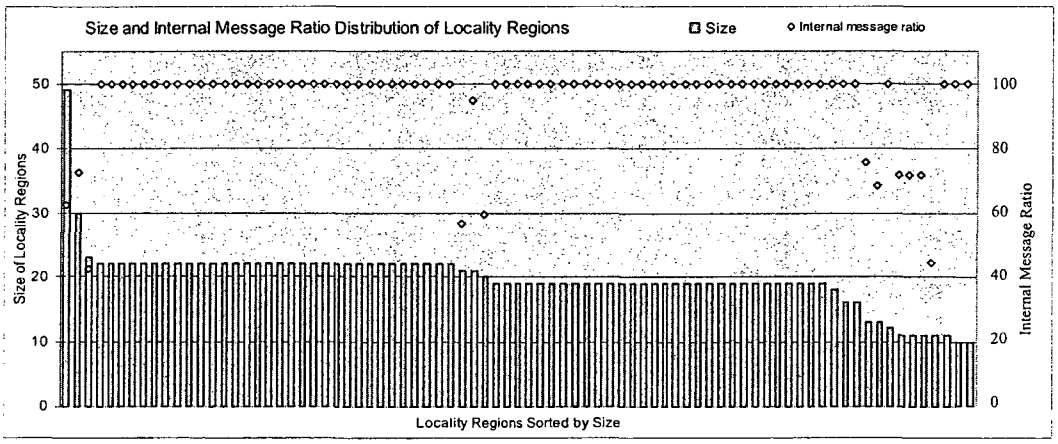


Figure 4.18 Size and internal message ratio of locality regions in manufacturing process application

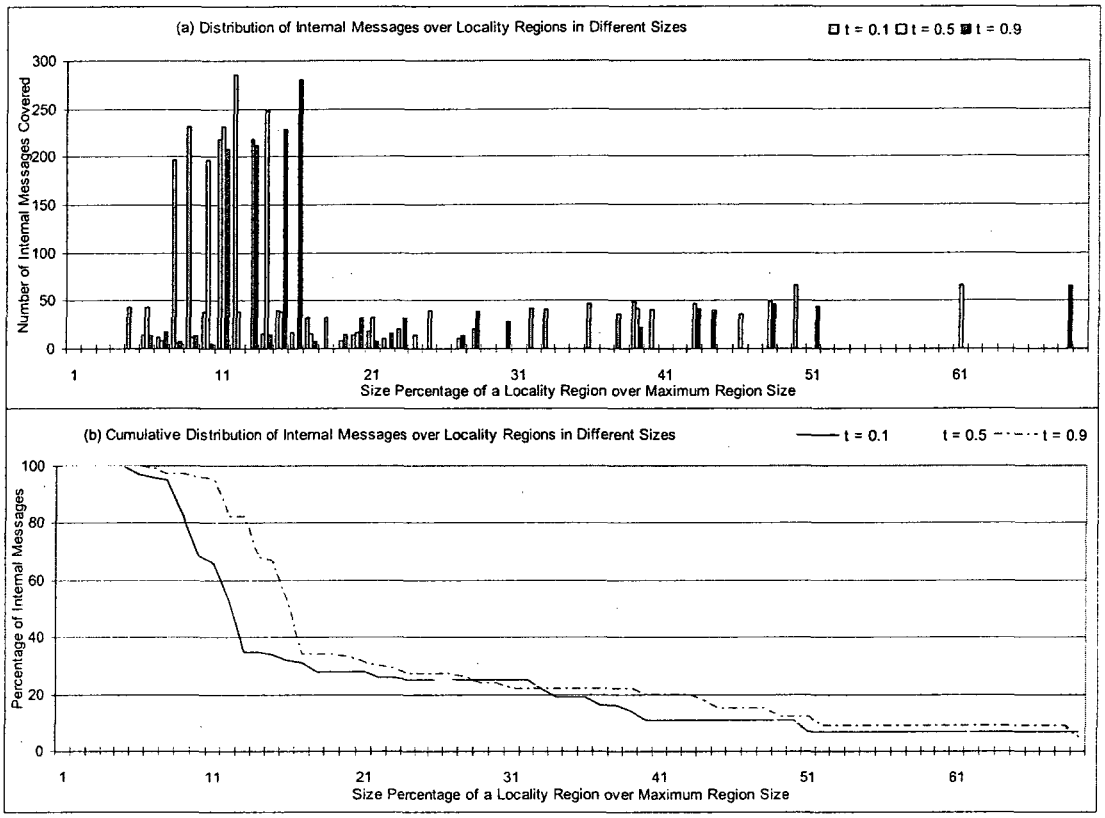


Figure 4.19 Distribution of internal messages over locality regions in different sizes for e-commerce application

The choice on threshold of primary important coupling has certain effect on the size distribution of locality regions hence formed. Figure 4.19(a) shows for the e-commerce application the overall distribution of internal messages over locality regions in different sizes, when varying the threshold t for primary important coupling. The X-axis represents the percentage of a region's size over the maximum region size that could be detected given a particular value of t. The Y-axis represents the total number of internal messages that all regions at a certain size could cover. Parameter t are tested with five values, 0.1, 0.3, 0.5, 0.7 and 0.9, and the same set of results is obtained for $t = 0.3$ and $t = 0.5$ (with a maximum locality region size of 110), as well as for $t = 0.7$ and $t = 0.9$ (with a maximum region size of 49). Clearly a small value of t leads to forming regions of smaller sizes. Enlarging the threshold t for primary important coupling encourages forming large

regions, but does not change the overall picture of distribution as well as the above conclusion. This is also implied by the corresponding cumulative distribution curves plotted in Figure 4.19(b), where the Y-axis represents the percentage of total internal messages covered by all regions no larger than a certain size. Figure 4.20 shows similar but on the other hand quite simpler results for the manufacturing process application. Again, this results from the simplicity of parameterizing at the design stage.

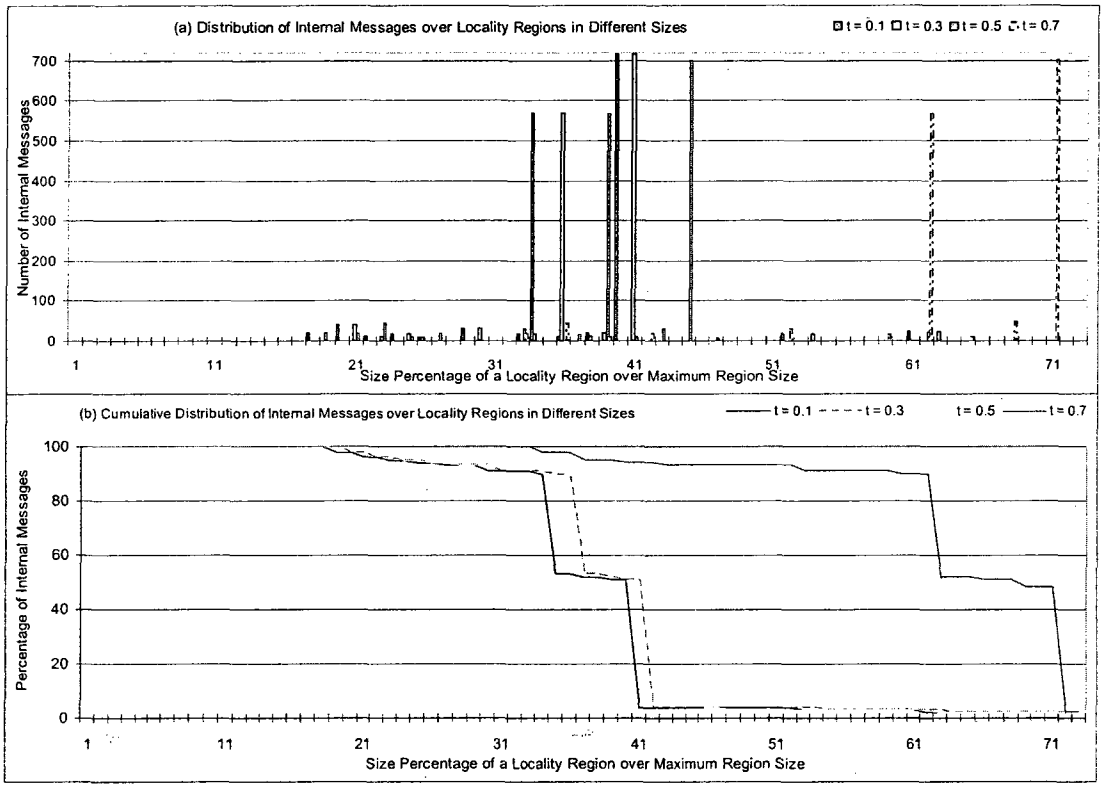


Figure 4.20 Distribution of internal messages over locality regions in different sizes for manufacturing process application

4.4.3.4 Test Case 4: Properties of hierarchical locality regions

Locality regions at different levels of hierarchy can be formed as transitive closure of QLI's at the corresponding level. As shown in Figure 4.11(b) and Figure 4.12(b), there are only two levels of hierarchy for QLI's in the e-commerce application. Therefore, it is not surprising that the locality regions hence formed also show only two levels of details.

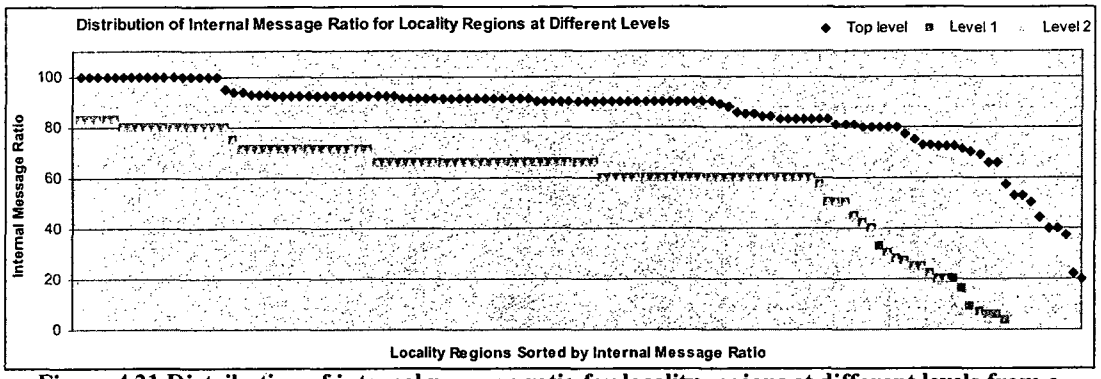


Figure 4.21 Distribution of internal message ratio for locality regions at different levels from e-commerce application

In particular, the top-level regions cover 72% of all messages, and the next-level regions cover 50% of messages found at the top-level, or 36% of all messages within the whole execution. Figure 4.21 shows the distribution of internal message ratios per locality region, where the regions are sorted by their values of internal message ratio in a descending manner along the X-axis. Clearly, locality regions at the top-level have larger internal message ratio than those at level-1. Note that the set of results for level-2 is almost the same as that of level-1, meaning in fact there is no more hierarchical structure onwards. This is also confirmed by the statistics in Table 4.4, with respect to executions in different scales. From the perspective of application design, such a two-tier structure is resulted from the existence of high-level buying tasks and the use of various FIPA protocols as building blocks for those tasks.

Application		e-commerce			manufacturing process		
Scale		small	medium	large	small	medium	large
top-level	No. of locality regions	18	120	42	26	81	135
	No. of all messages associated	406	1859	1107	482	1612	2891
	Overall ratio of internal messages	55%	72%	79%	86%	81%	80%
level-2	No. of locality regions	17	111	36	22	78	141
	No. of all messages associated	252	1479	864	430	1493	2670
	Overall ratio of internal messages	38%	36%	43%	81%	74%	71%
level-3	No. of locality regions	16	109	35	22	75	135
	No. of all messages associated	244	1461	838	430	1469	2559
	Overall ratio of internal messages	38%	36%	43%	81%	74%	71%

Table 4.4 Statistics on hierarchical locality regions

Results in Figure 4.22 leads to a different conclusion for the manufacturing progress application. For most instances, the set of results for top-level locality regions coincides

with those for level-1 and level-2. In other words, there is actually no hierarchical locality structure for this application. This is resulted from the fact that all negotiating and manufacturing tasks in this application are implemented by a few simple messages instead of complicated communication protocols.

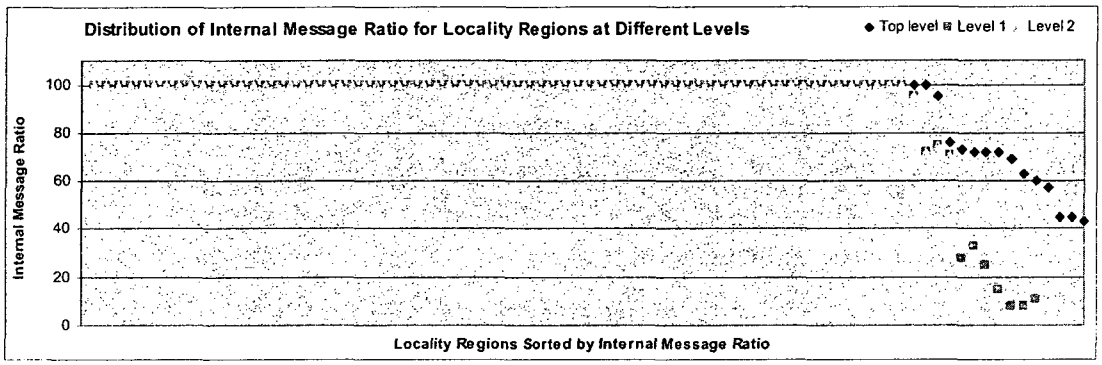


Figure 4.22 Distribution of internal message ratio for locality regions at different levels from manufacturing process application

The above two example applications are of small scale in real life, in terms of both numbers of processes as well as messages involved, and design complexity with respect to use of various patterns, protocols, and implementations. Regular large-scale distributed applications have choices to employ techniques at different levels, including automatic task scheduling, specific problem-resolving patterns, synchronization protocols, and standard communication primitives. As a result, there will be a rich multi-level structure at the design stage, and corresponding hierarchical locality regions observable at run time.

4.5 Remarks

This chapter studies the locality phenomenon of message interactions in distributed systems and makes its contributions mainly in three aspects. Firstly, analysis from the perspective of distributed program design reveals that interaction locality originates from resolving dependency localization via message passing, and is exhibited as a hierarchical 'region-transition' pattern in many distributed applications. Secondly, a bottom-up

approach is proposed to identify those by-design regions in a given distributed execution. In particular, a generic notion of 'locality interval' is developed to capture popular recurrence patterns within individual process lifelines, and frequent message interactions between such intervals are modeled as 'important coupling', by which they form a 'locality region' of localized messages. Concepts of qualified locality interval (QLI) and primary important coupling are also introduced to resolve certain practical issues. Finally, experiments are conducted on traces from real-life distributed applications, and justifications are made on four perspectives of interaction locality, namely traditional locality view, individual process view, region-wise view, and global view. Corresponding results and analysis justify the feasibility of the proposed approach as well as existence of the modeled locality.

In the literature, there are two notions of locality that are understood about a reference string, namely long-term popularity and short-term correlation. The former captures the recurrence property of references to the same object, measured at a particular time instance. It is often related to a particular context such as caching problem with an LRU stack, which has a 'long-term' measure for a whole reference string such as hit ratio [Den68] and stack distance [MGS+70]. The latter captures the recurrence property of references to a subset of objects, measured over a time period. It is more related to the fine-grained structuring of a reference string, which can be modeled as 'short-term' reference sequences with possibly hierarchical properties, e.g. bounded locality interval (BLI) [MB76]. These two notions refer to different aspects of traditional locality and corresponding properties co-exist within a reference string. This chapter shows that such properties are also observed during the study of interaction locality. For example, individual process lifelines of a distributed execution are interaction strings that exhibit traditional reference locality. This is reflected by their small 'long-term' average

recurrence distances and the existence of 'short-term' locality intervals. In fact, the notion of locality intervals captures all BLI's and also other popular recurrence patterns. Also, interaction locality appears in a form of 'region-transition' pattern, which is quite similar to the 'phase-transition' behavior [DK75] characterized in reference locality. In particular, both higher-level regions and higher-level phases are more distinctive from each other, and they both have more coverage of message or memory references compared with lower-level ones. However, interaction locality is much more complicated than traditional reference locality. In a distributed application, message references span in both space and time and form locality regions. As a result, there are two extra perspectives of interaction locality, in addition to traditional views from the above two notions. The region-wise view reveals the existence of important coupling relations between individual locality intervals of a same region, while the global view shows that there are hierarchical structures between locality regions at different levels of granularity. More importantly, such perspectives give rise to the bottom-up approach proposed in this chapter, making it possible to identify the actual locality regions for a given distributed execution. On the other hand, study on traditional reference locality is usually driven by a specific optimization problem, while this chapter only intends to show the by-design locality regions that are naturally observable. Detecting locality regions towards optimizing the overall performance of distributed checkpoint and recovery is further discussed in the next chapter.

Chapter 5

Locality-Driven Checkpoint and Recovery

Locality is traditionally an important factor related to performance, e.g. in memory and web caching systems. As an empirical result demonstrated in Chapter 4, distributed applications exhibit a new type of locality with various similar properties. Consequently, localization of such (computation or synchronization) efforts can be used as an effective strategy to facilitate performance-oriented strategies in the corresponding systems, in particular, large-scale distributed systems that involve a large number of messages and processes. An important application is distributed checkpoint and recovery.

In distributed systems, messages as well as process advancement create dependencies that often spread far and wide among each other. Removal of such dependencies requires proper handling of corresponding messages or processes, which necessarily incurs considerable overhead (refer to Chapter 3.2). As a result, distributed strategies usually could not achieve both good scalability and good performance at the same time. Instead, they might only perform well when deployed around a small scope of entities. In particular, distributed checkpoint and recovery strategies usually suffer from either global recovery effect due to checkpoint coordination [TS84, CL85] or degraded runtime performance caused by message logging [AM98, SY85]. However, existence of interaction locality provides a possibility to have advantages from both aspects. As an example, a hybrid group checkpoint strategy (discussed in Chapter 3.3) can have

minimized logging overhead with limited recovery spread, by forcing subsets of processes with localized message interactions to take coordinated group checkpoints and only logging the inter-group messages afterwards. This is due to the fact that interaction locality implies naturally-formed small scopes of entities with greatly reduced dependencies in between. Based on the same principle, locality of interactions can also be used to help resolving quite a few other problems in large-scale distributed systems. In general, solution strategies of such problems are applied to small locality entities of message interactions that satisfy certain localization criteria, which are often driven by both aspects of optimization requirements, i.e., scalability and performance. For distributed checkpoint and recovery, this refers to an optimized cost involving both recovery effect and runtime overhead.

An optimal solution for distributed checkpoint and recovery is a result of decomposing a distributed execution into a proper set of quasi-atomic locality regions. On-the-fly identification of the corresponding regions in turn requires knowledge about concurrent and future progress of all relevant processes. Obviously complete knowledge of concurrent as well as future interactions is a physical impossibility. However, existence of locality properties makes it quite possible to dynamically detect and predict the formation of certain locality regions at run time, and hence to produce a reasonably good enough solution for practical use. In particular, a process can detect the formation and also to predict the future growth of a locality interval by locally monitoring its recent message interactions. Moreover, it only needs to know the concurrent progress of its important partners (instead of watching over every other process), which is much easier to obtain via their tight coupling of frequent message interactions. Besides, there are also certain levels of detailed knowledge that can be made available at design time, providing further flexibility as well as choices for runtime strategies to identify locality regions

more accurately.

This chapter continues with the above idea and develops results on optimizing distributed checkpoint and recovery by locality-driven strategies. The rest of this chapter is organized as follows. Chapter 5.1 proposes an abstract performance measure for checkpoint and recovery protocols, and demonstrates its relationship to interaction locality as well as the optimal result of decomposition. Chapter 5.2 introduces various strategies of managing knowledge in order to assist runtime identification of locality regions. Chapter 5.3 discusses design choices available for fine-tuning the performance of group checkpoint protocols. Chapter 5.4 reviews the contribution of this chapter.

5.1 Performance of Checkpoint and Recovery

The term "performance" could be used to refer to various aspects of checkpoint and recovery. The two existing categories of strategies have different performance advantages, and are subject to corresponding performance-related factors. For example, log-based strategies are sensitive to the commit latency regarding stable storage. In checkpoint-based strategies, the important performance factors are checkpoint creation overhead, protocol coordination overhead, storage consumption, etc. Due to the presence of failures, one might also need to consider the costs occurred at recovery time, such as recovery protocol overhead, computation waste, and the number of processes being rolled back.

In the literature, performance of checkpoint and recovery has been measured differently. A simple way to understand the performance of a protocol is to check the extra time it adds on a failure-free execution. Contrasts can be made on the actual time of specific costs that a protocol incurs, including checkpoint overhead, logging overhead, and rollback delay [BLL89, NF96, NF98, RAV99, MM05, ZHK06]. Similarly, such raw experimental data can also be obtained for special factors such as CPU overhead,

communication cost [BLL89], disk consumption and storage latency [Pla96]. A more meaningful way is to compare the extra add-on time with the failure-free execution itself. Notions of "percentage overhead" [Vai94] and "fractural time overhead" [RUI97] have been introduced to represent the percentage increase of overall completion time of a given execution. A small percentage implies little cost and hence good performance. Results based on such measures have been reported with respect to specific costs [EJZ92, EZ94, SS97, Sen97, RAV99]. Based on the same principle, a comprehensive measure called 'overall gain' [PGB01] has also been proposed as the average per process fraction of total completion time of a given execution with versus without the application of a particular checkpoint and recovery protocol. In other words, the comparison is made between the execution that involves all runtime checkpoint as well as recovery costs, and the execution that only have failures and subsequent recovery costs. In addition, the calculation is from the perspective of individual processes in average instead of considering all processes together. Obviously, an overall gain less than 1.0 implies the significance of applying a checkpoint and recovery strategy, and the lower value it has, the better is the corresponding performance.

Existing performance measures are not good candidates for evaluating locality-driven checkpoint and recovery strategies. The raw data of extra time values is straightforward and detailed, and is only useable for understanding specific protocols or performance factors. The percentage-based measures are intuitive but have the same problem of only dealing with specific factors. The notion of 'overall gain' considers the costs of both checkpoint and recovery, and can present an overview about how a protocol could perform in general. However, it measures performance in a per-processes manner, and records nothing about inter-process message interactions. It therefore provides no benefit for understanding the locality pattern of a given execution, or its relation to the

performance of group checkpoint protocols. A new measure is hence necessary, and should involve the factors that are important and relevant to interaction locality.

5.1.1 An Abstract Performance Measure

Exhibition of interaction locality makes it possible for checkpoint and recovery strategies to have good performance at run time and recovery time as well. Locality patterns of distributed executions could be very different, which will lead to different performance. Intuitively, good interaction locality should give rise to good performance in both aspects, if proper checkpoint and recovery strategies could be applied. Given a distributed execution, the optimal performance is only subject to its locality pattern, and is independent of the availability of checkpoint and recovery protocols. This is similar to the effect of traditional reference locality in improving the overall caching performance. A measure of the optimal performance hence should be generic, comprehensive and locality-sensitive. This chapter proposes the checkpoint-recovery cost-efficiency as such a measure, which is defined as the overall checkpoint-recovery cost normalized by the original failure-free execution time of a given distributed execution.

From the Quasi-Atomic Recovery theory, a distributed execution can be treated as a set of well-ordered quasi-atomic recovery blocks (refer to Theorem 3.3 in Chapter 3.1.3). At run time, checkpoint protocols perform proper handling of message- and program-order dependencies, causing a decomposition of the given execution into a set of disjoint quasi-atomic interaction regions. For ease of discussion, they are termed *recovery regions*, as each of such regions could confine failures inside and hence corresponds to a quasi-atomic recovery block. For example, such regions are formed between two consecutive global checkpoints, and between two consecutive individual checkpoints in log-based recovery as well. Note that by definition there is no recovery spread between recovery regions, assuming all dependency relations have been properly taken care of. Creation of

checkpoints and message logs will superpose extra costs over the failure-free execution at run time, namely checkpoint and logging overhead. Also, possible failure and consequent recovery will lead to loss or waste of work, which forms another extra cost (term it as recovery waste). These three types of costs are the common as well as major considerations of most checkpoint and recovery strategies, and meanwhile can be measured separately in a region-wise manner. There could also be protocol-specific costs, such as the latency caused by coordination and blocking, overhead of logging extra messages (refer to the NGC protocol in Chapter 3.3.3), etc. These are subject to the proper design of checkpoint and recovery protocols, and related issues will be discussed in Chapter 5.3.

In a distributed system, an execution is usually modeled as a set of logical message events, without considering the actual time elapsed in between. Equivalently, there is a uniform distribution of message events and each behavioral fragment is associated with the same execution time. Such an assumption is adequate due to the fact that most distributed systems are message-intensive and transaction-oriented, and their progress is hence subject to the logical time (or virtual time) measured by the number of message events. The above-mentioned checkpoint and recovery costs as well as the failure-free execution time can also be modeled quantitatively in the same way. Given a distributed execution that consists of a set of disjoint recovery regions, its original failure-free execution time is proportional to the number of its message events A, which is a summation of the number of message events A_i for each component region R_i. The checkpoint overhead c_i for R_i is measured as $c_i = k_i \times c_1$, where k_i is the number of processes in R_i, and c_1 is a constant measuring the overhead per checkpoint. Similarly, its logging overhead is measured as $l_i = m_i \times c_2/2$, where m_i is the number of external messages for R_i, and c_2 is a constant measuring the overhead per message. Here an

external message is counted for a 'half' time in each region, and two half's will make one message when all regions are counted.

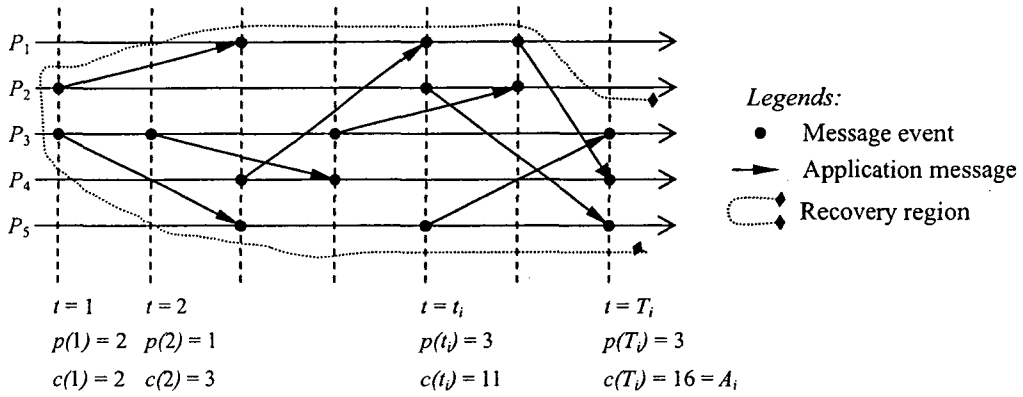


Figure 5.1 An example recovery region with logical clock labeling

Assuming equal probability of failure at every message event (denote it as a constant c_0), the expected recovery cost of a given distributed execution can be calculated in a region-wise manner. For each recovery region R_i, a positive number (e.g., logical clock [Lam78]) can be assigned to each message event such that the ordering of these numbers does not violate causality. Figure 5.1 shows an example of such numbering via logical time (with a range from $t = 1$ to $t = T_i$). Within a recovery region R_i, there are two functions associated for each time point t_i: $p(t_i)$ denotes the number of processes that have an event at t_i, and $c(t_i)$ denotes the total number of events from $t = 1$ to $t = t_i$. Obviously $c(t_i) = \sum_1^{ti} p(t)$. Suppose a failure occurs at a particular event on t_i, $c(t_i)$ captures the corresponding recovery waste and $p(t_i)$ captures the total number of events that have the same recovery waste. Hence the expected recovery waste of R_i can be calculated as: $r_i = c_0 \times \sum_1^{Ti} (c(t) \times p(t))$, where T_i is the largest logical clock value of t_i in R_i. Notice that $c(T_i) = \sum_1^{Ti} p(t) = A_i$, therefore $r_i = c_0 \times \sum_0^{Ti} (c(t) \times p(t)) = c_0 \times A_i^2/2$. This result is better understood via the following equations in calculus form.

For functions $c(t)$ and $p(t)$, $0 \le t \le T_i$, and $c(t) = \int_{x=0}^{t} p(x)dx$.

Since $p(0) = 0$, $c(0) = 0$, therefore $p(x)dx = d(c(x))$.

Hence, $r_i = c_0 \int_{t=0}^{Ti} c(t)p(t)dt = c_0 \int_{t=0}^{Ti} c(t)d(c(t)) = c_0 \times \frac{c^2(T_i)}{2} = \frac{c_0 A_i^2}{2}$.

Consequently the overall checkpoint and recovery cost C for the whole distributed execution is: $C = \sum_i (r_i + c_i + l_i) = \sum_i (c_0 A_i^2/2 + c_1 k_i + c_2 m_i/2)$. The performance of checkpoint and recovery can be measured from the perspective of cost efficiency, i.e., the overall cost normalized by the original failure-free execution time, as follows:

$E = C/A = 1/A \times \sum_i (c_0 A_i^2/2 + c_1 k_i + c_2 m_i/2)$

The above formula is only subject to the decomposition result of a given execution. Due to the existence of message interaction patterns, the measure E could vary largely due to different ways of decomposition. In this formula, the three components play different roles during such variations. Towards minimizing E, the first component $c_0 A_i^2/2$ represents the need of forming as small as possible recovery regions, and the latter two components $c_1 k_i + c_2 m_i/2$ represent the balancing forces to grow recovery regions from both space and time. Since $A = \sum_i A_i$, minimizing $\sum_i c_0 A_i^2/2$ means that all regions better have the same size and they should be as small as possible: the extreme will be that each region only contains one message event. On the other hand, minimizing $c_1 k_i$ means to avoid taking checkpoints and hence to keep as many as possible message events within each behavioral fragment, while minimizing $c_2 m_i/2$ means to avoid logging and hence to involve as many as possible processes within each region. The latter two components thus explain how log-based recovery and coordinated checkpointing could work, respectively. Taking these two components together into consideration, the extreme on this side is to have only one recovery region for the whole execution. Obviously the optimal decomposition with minimized E should be a balanced result between these two extremes, i.e., a set of regions at a certain size, which have a few external messages as well as common processes. In other words, it implies a locality pattern at a certain level in the given execution. This coincides with the intuition of group checkpointing as well

as its driving force of interaction locality. The formula hence reflects a convexity property of how decompositions could vary, and also demonstrates the feasibility of the locality-driven group checkpoint strategy.

The above cost-efficiency measure is abstract and generic. It starts from the fundamental perspective of recovery regions, and is able to capture the three essential costs that are common and critical to all types of strategies. Such costs are inevitable and hence form the minimum price to be paid when applying checkpointing and recovery. Protocol-specific costs are neglected and will be addressed separately in Chapter 5.3. On the other hand, these three factors are locally decidable and are all related to patterns of interaction locality. The measure hence provides a new view for studying the important factors and their effects in optimizing the overall performance.

5.1.2 Towards Performance Optimization

Given a distributed execution, it is critical to find the optimal decomposition that minimizes the cost-efficiency measure: $E = 1/A \times \sum_i (c_0 A_i^2/2 + c_1 k_i + c_2 m_i/2)$. However, even if full information is made available in terms of the whole execution trace, decomposing a distributed execution still incur undesirable complexity and finding the optimal solution hence becomes unrealistic. Alternatively, heuristic strategies might be desirable and can be developed from various perspectives. For example, based on the convexity property of E with respect to the size change of recovery regions, a heuristic could make its first move from either ends, i.e., either starting from the whole execution and trying to split it into smaller regions, or starting from each individual message event and trying to merge them into larger regions. The latter is more practical and meaningful, regarding the availability of knowledge within a distributed circumstance. In fact, the bottom-up approach proposed in Chapter 4 already demonstrates its feasibility in a similar way.

In general, to make the latter approach work, one needs to know where to begin with, how to grow a region, and when to stop growing. A simple prefix-pruning scheme can be suggested as following: starting from the initial cut of the given execution, a quasi-atomic interaction region can be grown gradually via merging small regions, which could also be single message events or individual behavioral fragments, through the paths of both message and program-order dependencies until it becomes large enough; then it will be chopped off in the form of a quasi-prefix and new regions will start over from the suffix that is left behind. During this region-growing procedure, one expects to see a continuous decrease of the overall cost for the subject region, until some point where the cost starts increasing and then the region stops growing and is pruned.

For each region R_i, its contribution to the overall cost measure E is $C_i = c_0A_i^2/2A + (c_1k_i + c_2m_i/2)/A$. Let p_i denote A_i/A, the percentage of R_i within the execution, and o_i denote $(c_1k_i + c_2m_i/2)/A_i$, the average per event runtime overhead for R_i, then $C_i = (c_0A_i/2 + o_i)p_i$. Consequently $E = \sum_i (c_0A_i/2 + o_i)\, p_i$ with $\sum_i p_i = 1$. $(c_0A_i/2 + o_i)$ represents the total cost of R_i and p_i represents its relative weight among all regions. E is then the weighted cost of all regions. This formula also means the cost of each region can be calculated separately and can be used as an indicator during the above merging procedure.

The criteria for merging two (disjoint) regions R_1 and R_2 is that, the resulted region R_{1+2} should have less cost contribution compared with the cost of two regions before merging. In other words, $p_1(c_0A_1/2 + o_1) + p_2(c_0A_2/2 + o_2) > (p_1 + p_2)(c_0A_1/2 + c_0A_2/2 + o_{1+2})$. Suppose there are k_{12} common processes and m_{12} inter-region messages between R_1 and R_2, then $o_{1+2} = [c_1(k_1 + k_2 - k_{12}) + c_2(m_1 + m_2 - 2m_{12})/2]/(A_1 + A_2)$. Finally, the merging criteria can be derived (after a few steps) into the following form:

$c_1k_{12} + c_2m_{12} > c_0A_1A_2$

The above inequality has several implications. First of all, merging is more likely

between small or asymmetric regions (in terms of size) within a large execution. This also verifies the feasibility of the above region-growing approach, which starts from single message events and individual behavioral fragments. Also, for two regions that have no common process or inter-region message, merging will never be appreciated: they better stay separately as their merge will never gain any decrease of the overall cost. Instead, merging is much favored between regions that have more common processes and/or inter-region messages. This means that the grown of a region is more preferably through the paths of intensive program-order and/or message dependencies. In addition, since usually $c_1 >> c_2$, a region will rather choose to proceed ahead with its current members than recruiting new ones, if it is having many enough message interactions with the outside. Moreover, all parameters in this inequality are locally obtainable and hence keeping or stopping the growth of a region is completely a local decision. In other words, decisions for concurrent regions can be made independently and their results will not affect the optimality of each other at all.

In general, for the merging of more than two regions, the criteria will become:

$c_1\sum_i\sum_{j\neq i} k_{ij} + c_2\sum_i\sum_{j\neq i} m_{ij} > c_0\sum_i\sum_{j\neq i} A_iA_j$

In the region-growing procedure, a recovery region will stop merging when the criteria no longer holds. In particular, given an under-growing region R_i, the condition by which it stops growing will be: $c_1k + c_2m < c_0A_iA'$ for any region R', where k and m denotes the number of common processes and inter-regions messages respectively. Consequently for any region R', $A_i > (c_1k + c_2m)/c_0A'$. Note that in this inequality, A_i is for an after-merge region which will never exist. In other words, A_i will never become larger than $(c_1k + c_2m)/c_0A'$, as once it reaches that value it will then stop. Since $k \leq A'$ and $m \leq A'$, the function $(c_1k + c_2m)/c_0A'$ has a maximum value when $k = m = A'$ (denote it as A^*):

$A^* = (c_1 + c_2)/c_0$

In case of merging involving more than two regions, the resulted maximum region size will be less than A^*. It is easy to prove that in an optimal decomposition, no region has a size larger than A^*, since if otherwise, such a region can be decomposed into two smaller regions and their cost contribution will decrease, which contradicts with the optimality.

Existence of such a maximum size for recovery regions has certain implications. It justifies the use of bounded region size in group checkpointing (refer to k-bounded protocols in Chapter 3.3.5). Maintaining over-sized regions will harm recoverability and hence the overall performance. On the other hand, it only serves as a theoretical upper bound and might not be used to replace the merging criteria itself. Even in an optimal decomposition, not every region has to have a size of A^*. In fact, there could be no such region at all.

A^* is only based on the three constant cost factors, and is easy to calculate. Larger factors of runtime overheads (i.e., c_1 and c_2) together with smaller factor of failure probability (i.e., c_0) encourage merging and hence formation of larger recovery regions. From the merging criteria, larger c_1 and c_2 with smaller c_0 also lead to clearer distinction between resulted regions, which could be either or both of less external messages and less common processes. Notice that these reflect the hierarchical properties of interaction locality discussed in Chapter 4.2, i.e., higher-level locality regions have larger sizes as well as fewer inter-region messages. Intuitively, this implies that the optimal decomposition is very likely to coincide with the interaction locality at a certain level, which is directly related to A^*. As a result, the problem of finding the optimal decomposition turns into detecting locality regions at the corresponding level, with the assistance of the merging criteria and the region size upper bound A^*.

5.1.3 Interaction Locality and Performance Optimality

For a given execution with fixed locality pattern, there is an obvious connection between performance optimality and interaction locality. However, locality patterns could vary largely in different distributed applications. It is also interesting to know how interaction locality could affect the performance optimality. In other words, what kind of locality pattern leads to the best (optimal) performance?

Following the discussion about the performance measure $E = 1/A \times \sum_i (c_0 A_i^2/2 + c_1 k_i + c_2 m_i/2)$ in Chapter 5.1.1, optimizing E requires minimizing all three components for each region, $c_0 A_i^2/2$, $c_1 k_i$ and $c_2 m_i/2$. Since messages could be either within or between regions, a simple situation would be having all messages contained inside regions, and each m_i thus becomes zero. In addition, each k_i could be as small as 2, and $c_0 A_i^2/2$ has minimized value if each region has the same size. Assume there are totally n such regions, i.e., $n = A/A_i$, it can be derived that in such a case $E = c_0 A/2n + c_1 2n/A$, which is a convex function of variable n, and has a minimum value when $n = (c_0/c_1)^{1/2} A/2$. This corresponds to an ideal pattern of n equally-sized atomic regions, each involving two processes.

Obviously the ideal pattern does not exist in real-life distributed applications. However, the above result indicates the effective properties of locality patterns that will lead to better performance: 1) equally- or similarly- sized regions at a proper level; 2) a minimum number of member processes for each region; and 3) a minimum number of inter-region messages. In other words, a set of performance-oriented locality regions should be regularized in size, mutually distinct in message inter-coupling, and each involving only a small number of processes. In fact, such regions are typical in distributed executions that exhibit good interaction locality.

In tradition, 'good' locality refers to the fact that a reference string shows a clear sign of recurrence patterns, which can be modeled in different ways, e.g., with notions like Working Set, etc. In general, the degree of 'goodness' can be qualitatively measured in

terms of average stack distance. Such a measure, on the other hand, captures nothing about the existence of fine-grained structures such as Bounded Locality Intervals. Similarly, a distributed execution is said to exhibit good interaction locality if region-transition patterns can be observed clearly and significantly. However, due to the hierarchical nature of interaction locality, there is no unique measure of how good locality a distributed execution could exhibit. Instead, good interaction locality is reflected separately from perspectives of both individual processes and locality regions. In general, locality is a phenomenon of continuous recurrences around the same subset of objects. Given a behavioral fragment of a process, good locality simply means: i) a small (important) partner set compared with a large fragment size, and ii) a large ratio of important partner messages. Notice that such locality properties are just limited and partial projections of those for its corresponding locality region. For each locality region, good interaction locality intuitively implies: i) a small membership set compared with a large region size; ii) a small ratio of external messages. Both properties actually lead to good performance due to the above performance-oriented requirements 2) and 3).

In a distributed execution, having equally- or similarly- sized locality regions does not seem to be a 'pure' locality property. However, it is not counter-intuitive either. At the design stage of distributed applications, proper task decomposition intends to make sub-tasks at a same level have similar scales, which in turn form corresponding locality regions with similar sizes. This is also demonstrated by the experimental results on the two example applications in Chapter 4 (refer to Figure 4.17 and Figure 4.18 in Chapter 4.4.3.3).

The above discussion concludes that good interaction locality lead to good performance. Notice that such a pattern is already a decomposition result for the corresponding distributed execution, and is in fact a good candidate for the optimal if

compared with other possible results. For each region R_i of the optimal decomposition, the merging criteria requires that R_i could not merge with anything of the rest world. In other words, it is a quasi-atomic region that is distinct and significant enough from its surroundings. Consequently the following inequality holds for each R_i: $c_1 k_i/A_i + c_2 m_i/A_i \leq c_0(A - A_i)$, assuming the same tradition of symbol uses. $A - A_i$ on the right hand side represents the rest world of the given execution except R_i. The left hand side contains two ratios that are directly related to locality of R_i: k_i/A_i compares its membership set with its region size, and m_i/A_i is a ratio of its external messages. The smaller these ratios are, the better locality a region exhibits. Obviously a pattern with better locality is likely to have more of its regions satisfy this requirement. On the other hand, this inequality also captures a balancing mechanism of the region-merging procedure discussed previously. With the growth of a region, a larger region size A_i tends to force smaller values for these two ratios, hence better locality for this region. In short, larger regions are likely to have better locality. This property is derived from the optimality of decomposition, but reflects the fact of interaction locality that higher-level locality regions are both larger and more distinct (refer to Chapter 4.2). In other words, the optimality intends to address the set of locality regions that are naturally formed and exhibited as good locality patterns.

It will not be surprising that the optimal decomposition result also exhibits good interaction locality. In a distributed execution with a pattern of good locality, locality regions are distinct from each other, and majority of their associated messages are internal in the form of coupling relations between member processes. Intuitively, such coupling should be tight enough to prevent members from splitting apart, and the tightness is rather a relative measure than an absolute one (e.g., in terms of the number of coupling messages). This property has been implied via the merging criteria. The optimal decomposition requires that any two mutually-complement parts of a region are better

merged together for improving its performance measure. Suppose a region is formed by two such parts x and y that have with no common process, i.e., they each involves at most one behavioral fragment from a member process. The following inequality holds: $m/(A_xA_y) > c_0/c_2$, where A_x and A_y denote respectively the size for x and y, and m denotes the number of their messages in between. The tightness of coupling hence can be measured by $m/(A_xA_y)$ and is required to be at least proportional to the size of both parts (since c_0/c_2 is a constant). It applies to any two mutually-complement parts within a region. Let k_x and k_y denote the number of processes involved in x and y respectively. In case that $k_x = 1$, part x is actually a behavioral fragment of a process, say p_x. Good locality of a behavioral fragment is usually reflected by a large ratio of important partner messages, and a small number of important partners compared with a large fragment size. Both properties are derivable from the above inequality. Each behavioral fragment satisfies the following: $m/A_x > A_yc_0/c_2$ and $m/k_y > A_x \times (A_y/k_y) \times (c_0/c_2)$. Here m/A_x represents the ratio of p_x's messages interacted with the other k_y processes of the locality region, which are usually the important partners for p_x in this behavioral fragment. This ratio is required be at least proportional to A_y, the total number of message events from other processes. In addition, m/k_y represents the average recurrence of important partners and should be at least proportional to both A_x and A_y/k_y, i.e., the length of the behavioral fragment of p_x, and the average length of those from its important partners.

To sum up, good interaction locality leads to good performance, and can serve as a good candidate for the optimal decomposition. Patterns with good interaction locality consist of distinct locality regions which are feasible to identify. As shown in Chapter 4, a bottom-up approach can be adopted to decompose the 'static' trace of a given distributed execution. However, such information will become incomplete for checkpoint and recovery protocols at run time. Next chapter addresses related issues on managing

information of locality regions for optimized overall performance of group checkpointing.

5.2 Identifying Locality Regions for Group Checkpointing

Given a distributed execution, group checkpoints are better managed around locality regions for improved overall performance. Since group checkpoints are created along with the execution progress, each locality region needs to be identified accordingly on the fly. In fact, each group checkpoint protocol can be considered as an abstract framework for locality region identification, using a dynamic mechanism that is similar to the prefix-pruning scheme discussed previously in Chapter 5.1.2. For example, in the AGC protocol (refer to Chapter 3.3.3), a group initiator spreads the group formation invitations to its potential members, which upon receipt of such a message will decide to join or not to. The decision of both invitation list and group joining is based on individual perception of future locality from respective processes. A region is hence formed by composing behavioral fragments of actual member processes, and will be considered as equivalently pruned (i.e., losing its effective value of use) once successive group checkpoints are created afterwards. In this protocol, the missing part is the actual decision-making mechanism supported by appropriate locality knowledge about the given execution. On the other hand, interaction locality is hierarchical in nature, and the optimal performance leads to decomposed locality regions at a certain level. The desired locality knowledge is therefore only decidable if the performance constants c_0, c_1 and c_2 are specified. This chapter assumes that these constants are given a priori in the following discussions. Also assumed is the proper handling of program-order dependencies among group checkpoint, say via process cloning, or its alternatives that are discussed in Chapter 5.3.

A challenge for on-the-fly identification of locality regions arises from the unavailability of concurrent as well as future knowledge in distributed systems. The

identification procedure requires such knowledge because group checkpoints and their corresponding logging policies are determined right before the formation of locality regions. Regarding each locality region, a group checkpoint protocol needs to decide the following a priori: i) which process should initiate a group checkpoint and when to do so; ii) which processes should be involved; iii) when each of its members should leave the group. All these decisions are to be made based on awareness of the concurrent and future locality patterns from each involved process. Inaccuracy or unavailability of such information might cause undesirable and even greatly degraded performance.

Fortunately, properties of interaction locality make it possible to obtain sufficient locality knowledge from both design time and run time. In particular, information about task decomposition and the hence-resulted locality management is the major outcome at each design level. It could be easily made available for runtime uses in different ways, e.g., in terms of agent roles, agent protocol sessions, parallel architectural skeletons, etc. In fact, such information is widely-used in standard communication protocols, programming libraries, application design platforms, etc. On the other hand, due to the intrinsic nature of locality, future interaction patterns of a process is largely predictable at run time, based on observation of its recent activities. In properly-designed checkpoint protocols, concurrent knowledge can be obtained at run time, through implicit or explicit message exchanges. And thanks to the tight-coupling of group members, concurrent knowledge could be delivered effectively and in time. In addition, performance will not be greatly affected since this could be done asynchronous via message piggybacking, or synchronously via a few extra system-level messages. Discussed in the following chapters are detailed strategies with example protocols that make use of such design time or runtime knowledge for group checkpointing.

5.2.1 With Design Time Assistance

Distributed applications, especially those in large scales, are complicated systems. Design of such a system usually has to follow a top-down approach and to apply the divide-and-conquer strategy hierarchically. The results produced at each level could be very useful in terms of locality identification. The following demonstrates such a design procedure.

In general, a distributed application is about resolving a specific task in a distributed manner. At the first design stage, a given task is decomposed into a set of subtasks together with an abstract task pattern, e.g., a parallel structural or behavioral skeleton [WLG04]. Depending on the solution strategy used for the given task, this task pattern could be of different types, e.g., singleton, master-slave, pipeline, divide-and-conquer, etc. Each task pattern captures the pattern-specific communications between subtasks and their corresponding interfaces, which could be customized by specifying certain parameters upon actual use. In addition, a task pattern is usually composable and its sub-tasks can also be further resolved in a similar way. The result of this design stage is a hierarchical composition of subtasks, each formed by an abstract singleton that is to be concretized with its own subtask logic and control mechanism.

A subtask is accomplished through collaboration of processes that play different roles. The next stage of design addresses this issue and decomposes each subtask into a set of coordination patterns. A coordination pattern consists of a set of roles that follow a specific communication procedure to achieve a goal. For example, a blackboard pattern provides a medium that allows all participants to monitor the progress of each other and to share public information. Popular coordination patterns include blackboard, meeting, market-maker, master-slave, negotiate, etc. [DWK01]. Nesting of patterns is also possible, in which case a sub-coordination works as a small and temporary subtask. For example, an interim meeting could be scheduled between several brokers within a large market-maker pattern. A coordination pattern defines the life-cycle for each member role, and

hence introduces temporal and spatial coupling between them. As a result, its subtask has a corresponding space-time sub-region that involves all of its communication messages.

A role is a logical behavioral unit and needs to be mapped into an actual process when a design is final. Simultaneous roles are possible for one process if these roles are within the same subtask. In addition, popular interaction patterns between processes, such as request, query, contract-net, auction, proposal, have been standardized and widely used, e.g, in the form of FIPA interaction protocols [FIP03]. Such patterns usually involve only two or a little more processes and several rounds of message interactions, therefore serve as a low-level communication layer in addition to the coordination patterns at the middle level. Due to the complication rising from role mapping, a corresponding sub-region at this level might not have interaction locality as good as the mid-level ones.

The above design procedure shows clearly the hierarchy as well as the proper containment relations between patterns at different design stages, namely, task pattern, coordination pattern and interaction pattern. Due to the complexity introduced by large-scale-ness, design of many distributed applications might not follow exactly these steps, but are very likely to result in the similar hierarchical structures. Good locality originates from well localized application logic, and leads to proper decomposition at each stage as well as distinct locality regions at corresponding levels. On the other hand, considerable efforts have been made to develop design methodologies [Ken00, DWK01, GSP02] as well as programming frameworks [HSP05, SEA+07, HRV+08], in order to provide supports for all types of patterns. These results and products not only benefit application designers and programmers, but also disclose information about the locality patterns at different levels. For example, in a distributed application that uses FIPA interaction protocols, a message is usually tagged with: i) the type of the interaction protocol in which it is generated, and ii) the identifier for that protocol instance (i.e., 'conversation-

id' in ACL specification [FIP03]). Roles of sender and receiver could also be tagged if necessary. Similarly, information about task patterns and coordination patterns can be made available at run time, in both individual processes and their messages. Such information might include the type as well as instance identifier of each pattern, and the types of important roles played by each process, in particular, subtask managers and coordination initiators.

As discussed previously, a group checkpoint protocol needs to know the following about a locality region: i) the checkpoint initiator and its emerging moment; ii) the list of group members; and iii) the dismiss time for each member. A specific protocol might not need all the above information. For example, the AGC protocol requires i) and ii) only because group members automatically dismiss when new checkpoints are taken successively. In general, requirement i) is easy to assure at design time and to become accurate enough at run time. A simple way is to make use of the message tagging and checking mechanism such that a pattern initiator could be identified once it starts sending its pattern-specific messages. Requirements ii) and iii) can be satisfied via the diffusion of messages that are tagged with pattern identifiers. In other words, the pattern identifier when tagged over messages will serve as a unique 'color' for its corresponding group, and hence will be able to distinguish the group from others (refer to the group coloring mechanism introduced in Chapter 3.3.2). What is essentially required is a numbering scheme for all pattern instances at each level and the corresponding message tagging. This can be easily implemented by modifying the supporting libraries of the abstract patterns and their messaging mechanism.

Another critical issue to be resolved is regarding the hierarchical nature of interaction locality. Given a set of performance constants c_0, c_1, and c_2, analysis from Chapter 5.1 concludes that the optimal decomposition coincides with locality at a certain level instead

of an arbitrary level. A protocol needs to know which levels of locality patterns should be used for group checkpointing. In other words, should groups be formed around subtasks, coordination patterns, or interaction protocol sessions? The size upper bound A^* for locality regions (refer to Chapter 5.1.2) implies that the number of message events associated with each group should be limited (up to A^*). In fact, the scale of a certain type of pattern is usually well parametrized at design stage and is specified during implementation. Such examples include the number of processes involved in a master-slave skeleton, the number of roles instantiated in a blackboard coordination pattern serving as participants, and the rounds of bids to be performed in an auction protocol session. Related information can be made available for runtime use, e.g., by inserting into each pattern a counting layer that works during the pattern initialization.

Assuming feasibility of obtaining the above-discussed knowledge, a simple group checkpoint protocol can be developed, as a variant of the AGC protocol. In particular, it relies its efficiency on the following assumptions about a given distributed execution:

(a) By-design locality patterns are hierarchical and are properly contained. Pattern embedding or overlapping is considered as forming next-level patterns.

(b) Locality patterns at the same level are numbered, each assigned with an instance identifier. Any locality pattern L_i at a lowest level k is presented by a unique tuple $L_i = <i_0, i_1, i_2, \ldots, i_x, .., i_k>$, $1 \leq x \leq k$, where each i_x is the instance identifier for its containing pattern at level x. A tuple L_i is said *compatible* with tuple L_j (written as $L_i \subseteq L_j$) if its locality pattern is the same as or is contained within that of L_j.

(c) Each role inherits the same tuple from its host pattern. Simultaneous mapping of multiple roles $L_0, L_1, \ldots, L_i$ to a same process p_i is only for roles with tuples that are compatible with one another, i.e., $L_0 \supseteq L_1 \supseteq \ldots \supseteq L_i$, and p_i inherits the most compatible tuple L_i.

(d) Each application message is tagged with a pair of tuples from both the sender and the receiver.

(e) Each pattern has a designated initiator. Knowledge about the size of each pattern is made available to the initiator upon its participation into the pattern.

Among the above assumptions, condition (a) guarantees that all patterns are properly contained in one another and hence each of them could be identified with respect to its containers at each level. Condition (c) is prepared to avoid messages across low-level patterns that belong to different high-level containers. Note that the purpose of these two conditions is just to simplify the following protocol design. They could be relaxed with use of complicated control mechanism. Described below is an optimized AGC (oAGC) protocol that makes use of the above conditions for achieving better performance.

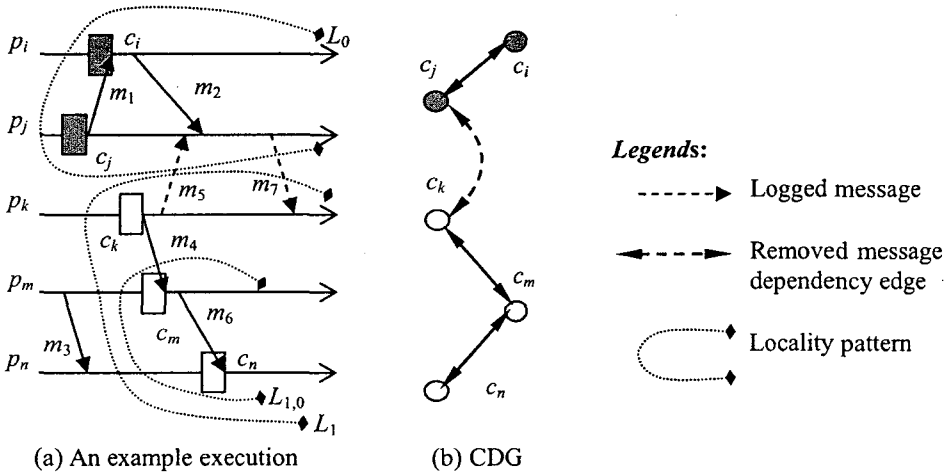
Starting a group: An initiator process can start a new group around a locality pattern at a certain level, based on its knowledge about the pattern size. It will use the pattern tuple as the group color and take a colored checkpoint before sending the first message of that pattern.

Joining a group: Upon receiving a message in a new color, if the tagged tuples are compatible with one another and are also compatible with the color tuple, a process will join the corresponding group by taking a new checkpoint with the new color.

Logging policy: Messages are all colored at sender side. A message will not be logged only if its tagged tuples are compatible with one another and are also compatible with its color tuple, and the corresponding message dependency edge will be recorded.

Figure 5.2 shows the same example used for the AGC protocol in Chapter 3.3.3, except that there are three properly numbered locality patterns L_0 = <0>, L_1 = <1> and $L_{1,0}$ = <1,0> formed by design and such information is available to their initiators p_j, p_k, and p_m respectively. Before sending message m_1, p_j starts a 'red' group around L_0 and colors m_1 with L_0. Since m_1 is colored as same as its sender and receiver, p_i joins the 'red' group by taking c_i upon receiving it. Similarly, m_2 will not be logged. Meanwhile, p_k initiates another 'white' group around L_1, which subsequently involves p_m and p_n via

message m_4 and m_6. Note that within L_1, there is a sub-level locality pattern $L_{1,0}$ that involves p_m and p_n only. Upon receiving m_4, p_m joins the 'white' group, since its own tuple <1,0> is compatible with m_4's sender tuple <1> that is as same as m_4's color tuple. p_n also joins by observing that m6 has the same tuple <1,0> for both sender and receiver, which is compatible with its color tuple <1>. In addition, messages m_5 and m_7 have different tuples for their sender and receiver and they will be hence logged.



(a) An example execution (b) CDG

Figure 5.2 Checkpointing with the oAGC protocol

Theorem 5.1: The oAGC protocol is correct and creates atomic group checkpoints.

Proof: In the logging policy, the compatibility checking guarantees that messages sent from outside of the locality pattern being checkpointed are always logged. Otherwise, a message dependency edge is included in the protocol CDG. Hence the CDG is always proper and the protocol is correct.

From the protocol, a checkpoint is always created before a group member sends or receives its first message of the locality pattern being checkpointed. As a result, any two checkpoints of the same group are causally consistent. Hence the condition (a) on consistency holds (refer to Definition 3.6 in Chapter 3.3.3). In addition, once a checkpoint is created, the process will apply the logging policy to all received messages based on

their coloring. Since any message carries the new color once a checkpoint is taken, no group member will log a message with the same color. Thus condition (b) on uniformity also holds. □

The oAGC protocol is simple and efficient. The pair of sender/receiver tuples tagged with each message automatically informs the receiver about its host pattern and compatible (container) patterns at all levels, as well as those for the message itself. The group color then indicates the locality pattern to be checkpointed, and its spread along with messages actually covers the target locality pattern. The compatibility checking mechanism works like the masking scheme used in network addresses: messages are logged if they are not between processes within the desired pattern. In addition, the proper containment of hierarchical patterns guarantees that the tuple representation will work effectively and efficiently. The simultaneous role mapping assumption eliminates complicated pattern overlapping and simplifies both group checkpoint and logging.

The oAGC protocol can effectively manage group checkpoints around desired locality patterns. On the other hand, the accuracy of so doing towards performance optimization is completely based on the fixed knowledge of initiators. In fact, such information is only easy to collect statically and in the spatial form, e.g., the number of participants during initialization of locality patterns. However, there are situations where roles and coordination patterns are dynamically added due to real time needs, and this could affect the accuracy and hence the performance. Alternatively, these dynamic changes could be monitored and estimated on the fly, and could be then made use of to assist group checkpointing. Discussed next are issues and strategies in this regard.

5.2.2 Via On-The-Fly Detection

Identifying a locality pattern is complicated in case of no enough design time knowledge. For example, a pattern initiator is only recognizable as the process which sends the first

pattern message. However, to distinguish initiators for patterns at different levels, it is still necessary for messages to tag their pattern types. In addition, any process can only decide its participating time for a locality pattern once it sends or receives its first pattern message.

Interaction locality provides another choice from a different perspective. Since a locality pattern forms a locality region at the corresponding level, each participant process often has a behavioral fragment with quite good locality as well. As discussed previously in Chapter 5.1.3, this implies a small (important) partner set with a large fragment size, and a large ratio of important partner messages. Consequently, the important partner set can be detected after just a few rounds of interactions, e.g., by observing a stabilization of stack distance values. In addition, a major change of such values and the top-most stack objects (i.e., the set of most recent partners) is very likely to indicate a transition between adjacent locality patterns. On the other hand, similar to the bottom-up approach used in Chapter 4, individual process can compose their behavioral fragments together following the tight-coupling relations between each other. More importantly, due to the hierarchical nature of interaction locality, information about the most recent locality pattern can serve as a good predication for the next future locality pattern at the same level, e.g., the important partner set and the expected participating duration of each process. Good interaction locality benefits all the above schemes: a small number of occasional partners leads to fast detection of the important partner set for each process; a small portion of inter-region messages helps identifying individual locality patterns; moreover, good predictability provides desired information and hence improves the detection procedure.

Given a set of performance constants c_0, c_1, and c_2, the optimal decomposition coincides with locality patterns at a certain level. From Chapter 5.1.2, the resulted

locality regions have an upper bound A^* for their sizes. A simple solution hence is to use A^* as the given bound k in the BAGC protocol (refer to Chapter 3.3.5) and to form group checkpoints accordingly. However, A^* only serves as a theoretical limit, and its value could be larger than what is actually required to be. In other words, many regions might have a better performance measure if their sizes could become a bit smaller than the exact value of A^*. In fact, using the merging criteria, the prefix-pruning scheme discussed in Chapter 5.1.2 can be adapted to a greedy algorithm, which dynamically composes tightly-coupled processes to form adequate locality regions. In particular, based on the most recent interactions, a group initiator can detect its important partner set and initiate invitations to such partners for joining its group. The decision on invitation list as well as group joining will be based on the merging criteria. The group will keep growing until the group measure stops being improved. During this procedure, information about the past groups will be used to decide the future groups. The following dynamic NGC (dNGC) protocol is such a candidate. It is based on an assumption that special processes are designated at design time as locality pattern initiators, which is valid in most cases especially for low-level locality patterns, e.g., FIPA interaction protocols.

Starting a group: An initiator process can start a new group by first taking a colored checkpoint and then append its color to its outgoing messages. An invitation list consisting of important partners invited to join this group is constructed based on the initiator's perception of future locality.

Perceiving the future interactions: A process perceives a future locality of interactions once it observes a stabilization of both its stack distance values and its most recent partner set. It then constructs its important partner set accordingly.

Propagation of invitation: The invitation list is appended to the first application message sent from a member of the group to a process in the invitation list.

Propagation of causal group knowledge: A process collects knowledge about its group partners that is causally available to it via the messages received from its partners. Such knowledge includes the group partner set, and the behavioral fragment size of each partner. It continues propagating the knowledge by appending its current knowledge base to every outgoing message.

This causal knowledge will be used for estimating the merging criteria when deciding to join or leave a group, or to merge two groups. A decision of growing a group will be made only if the resulted group

performance measure can be improved.

Leaving the current group: A process can create a minimal and colorless checkpoint at any time when it perceives no further tight coupling with the current group (hence no more improvement of the group measure). A colorless checkpoint reflects that it is not part of any group yet.

Acceptance/decline of invitation individually: An invitee process may make an individual decision upon receiving a token invitation. It can decline to join a group by taking no action, if its perceived future locality differs from the inviter's. It might accept an invitation (promptly or later) as an individual, if its perceived future locality is similar to the inviter's but different from its current group. In such as case, it will either create a new checkpoint with the new color, or convert the existing colorless checkpoint into one with the chosen color.

Merging of two groups: An invitee process might also suggest a merging of its old group with the inviting group (based on its local knowledge and received knowledge from the inviter), by submitting a suggestion message to each initiator. The invitation message will be logged based on its color.

The suggestion message will be appended with the invitee's knowledge about each group, including the color, the initiator, and the merging factors. The two initiators then will establish a two-phase handshaking to reach an agreement about whether to merge or not, say started by the initiator with a smaller process id. Decision will be made at both sides based on their estimation of the merging criteria. In case of merging, a new initiator will be elected, say the one with a smaller process id, and information about the merged group will be broadcasted to every process of the new group, which upon receipt of such information will update its color, initiator, and local knowledge base.

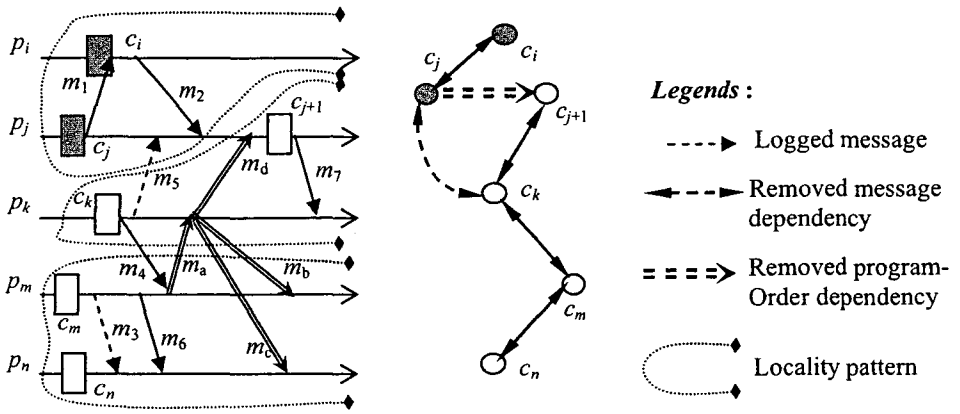
During this procedure, any merging suggestion received will be re-considered after an agreement is made, and any handshaking message regarding another merging agreement will be replied with an answer of no merging. No action will be taken if a merging decision is not made.

Group dismiss: A group initiator will enforce a group dismiss if it observes an increase of the group performance measure. It will broadcast a dismiss message to every member, which in turn will create a new colorless checkpoint.

Logging policy: Messages are tagged with the color of the sender. Any message tagged with the same color as that of the receiver will not be logged, and the corresponding message dependency edge will be recorded. A message from/to a colorless checkpoint is always logged.

Figure 5.3 shows an example of dNGC checkpointing around three locality patterns with initiators p_j, p_k, and p_m respectively. Processes p_m and p_n have initially left their previous localities and created minimal (and colorless) checkpoints c_m and c_n. Message m_3 is hence logged. p_j sends an invitation message m_1 to p_i based on its perception of future locality, and p_i takes c_i to join the corresponding 'red' group. When p_k perceives a

future locality involving p_j and p_m, it sends its invitations via messages m_4 and m_5 in order to form a 'white' group. However p_j is still in its current locality with p_i upon receiving m_5, it logs m_5 for the time being. Later on when p_j perceives a change of its locality, it makes an individual decision of joining the 'white' group (by taking a new checkpoint c_{j+1}). Consequently, m_7 is sent with a 'white' color and is not logged. On the other hand, when p_m enters a locality that involves p_n, it sends out its invitation via m_6 and p_n in turn joins its 'blank' group. Later when p_m receives m_4, it decides to merge its current group with the inviting 'white' group by sending a suggestion message m_a to p_k. Since p_m is currently a group initiator, an agreement is made once p_k accepts the group merging. p_k then notifies everyone in the new group including pm via messages m_b, m_c, and m_d. Subsequently p_m and p_n change their colors into 'white'. Figure 5.3(b) shows the corresponding CDG with use of cloning, where the message dependency between c_j and c_k and the program-order dependency between c_j and c_{j+1} are removed.



(a) An example execution (b) CDG with cloning

Figure 5.3 Checkpointing with the dNGC protocol

Theorem 5.2: The dNGC protocol is correct.

Proof: Since neither the group merging nor the group dismiss affects message coloring or message dependency recording, the logging policy maintains

the properness of the protocol CDG in the same way it does for the NGC protocol. Hence the claim. □

The dNGC protocol provides the option of group merging, which allows dynamically growing a group checkpoint towards optimizing the overall performance. The relaxed requirement of non-atomicity is critical here, as otherwise it is impossible to maintain consistency of the after-merging group checkpoint or the uniformity of logging policy. On the other hand, the protocol requires each group to dismiss once the corresponding group measure starts to increase. This prevents group from getting over-sized and thus further improves the decomposition result.

Good locality leads to good overall performance of this protocol. It should be noticed that the above advantages as well as flexibilities are on the cost of logging extra messages, especially those between groups being merged. Good individual locality results in fast detection of important partners and hence avoids further logging of such messages. In addition, the decomposition result will become close to the optimal if accurate enough information has been taken for estimating the merging criteria. Good locality of locality regions leads to frequent message passing between partner processes within each region, which benefits the propagation of causal knowledge and hence improves the knowledge accuracy. Moreover, the dNGC protocol employs explicit kernel messages for merging suggesting, agreement making, and decision broadcasting, which also adds extra costs to the runtime performance. However, in locality regions with good locality, partners usually have uniform inter-coupling between each other. As a result, chances of group merging will be greatly reduced and so as the corresponding kernel messages.

5.3 Design Choices for Checkpoint Protocols

The performance of checkpoint and recovery is affected mainly by the three factors captured in the cost-efficient measure E regarding each recovery group, i.e., the group size, the number of participant processes, and the number of external messages. The various group checkpoint protocols demonstrated previously in this chapter as well as those in Chapter 3 are proposed to improve the overall performance from these aspects. Besides the above major and inevitable concerns, there are also several types of minor performance costs that can be flexibly tuned upon the protocol designer's choice.

In general, a checkpoint protocol is designed to properly handle all dependency relations that exist between its target recovery regions in order that failures will be well confined within each individual region. As discussed in Chapter 3.2, message logging and process cloning are effective ways for handling respectively message and program-order dependencies. In particular, the following needs to be guaranteed for each recovery region by the checkpoint protocol: i) in-transit messages across the group checkpoint cut and external incoming messages across the group boarder should be fully logged; ii) orphan messages across the group checkpoint cut and external outgoing messages across the group boarder should have at least their sequence numbers logged; and iii) a clone is created for any process that could possibly leave the group while others are still in inside.

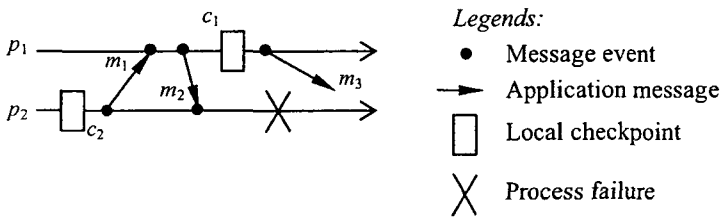


Figure 5.4 An example execution with failure

A message could be logged (fully or otherwise) by either or both of the sender and the receiver. Figure 5.4 shows an example execution involving two processes p_1 and p_2.

Suppose p_2 has a failure before receiving m_3 and has to roll back from checkpoint c_2. Afterwards p_2 requires to: i) discard m_1; ii) replay m_2; and iii) receive m_3. Discard of m_1 can be done by either p_1 or p_2, which should have a record of m_1's sequence number. Similarly, m_2 can be replayed either locally by p_2, or remotely by p_1. Obviously sender-side discard and receiver-side playback are more efficient and hence more preferable. In addition, receipt of m_3 should be supported by the communication layer such that any still-in-transit message like m_3 will be eventually delivered to the receiver once the receiver is recovered to its pre-failure state. In other words, the receiver queue integrity must be preserved. For efficiency sake, it is better to log (full) messages at the receiver side, which is also the assumption used in many checkpoint protocols. However, as discussed in the following, the choice of (redundant) sender-side logging provides more flexibility for checkpoint protocol design.

5.3.1 Atomic Group Dismiss

Process cloning is a recovery technique for proper removal of program-order dependency relations between checkpoints. In group checkpoint protocols, it can be used to prevent recovery spread among successive groups. For example, in Figure 5.5, p_2 forms a 'white' group with p_1 upon the invitation message m_1, and later on chooses to join a 'red' group with p_3 upon an invitation via m_3. However, p_1 still remains as a member of the 'white' group after p_2's leave. Some time later, a failure of p_1 (as shown in the figure) will trigger recovery of both groups due to existence of the common process p_2. In particular, message m_2 needs to be replayed by p_2 that is rolled back to c_2, unless a process clone of p_2 is in pace and does this job for it. In particular, such a clone is created for the purpose of replaying messages for other members of the old 'white' group, as these messages are not logged at either the sender side or the receiver side. It hence has a limited lifetime and will be killed after the playback mission is accomplished, say via a countdown of the

number of its outgoing messages. Obviously the trade-off here is between the cost of creating a clone at recovery time and the cost of logging all the messages to be replayed at run time. However, such a decision is never straightforward to make unless upon joining p_2 is known to be leaving the 'white' group early. Another choice is to have a clone standing by when p_2 leaves the 'white' group. This stand-by clone will be in action just like p_2, only within the old group and only when a recovery of that group is triggered. It will exit when that group is finally dismissed. Again the trade-off between the two types of clones is cost on recovery need versus cost of standing by.

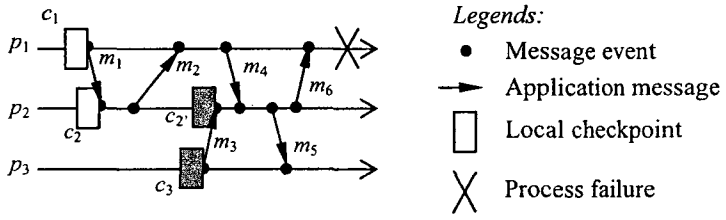


Figure 5.5 An example of atomic group dismiss

A clone is not always necessary, for example, if a group could dismiss atomically. In other words, all members leave the group as if instantaneously, and a subsequent failure of any member will not trigger a recovery of this group. Atomicity might be made possible via proper protocol design.

In general, atomic protocols are usually preferred in distributed systems. A group action is said to be atomic if the corresponding result is as if it has been performed instantaneously. As an example, the AGC protocol is atomic because it creates every group checkpoint as if instantaneously. In this protocol, atomicity also implies uniformity of message logging with respect to the group formation. The as-if instantaneity is managed via on-the-fly decision of the group membership: a diffusion-based procedure spreads the group color that is carried over every group message and the receiver can easily decide its logging (based on the assumption of receiver-side logging). This protocol is appreciated due to its minimized logging efforts that avoid logging waste, and

its non-blocking property that eliminates progressing waste. Alternatively, blocking protocols are easier to design, and non-atomic protocols are more flexible to use.

A naïve atomic group dismiss is to block the execution of any group leaver until the group finally ends. However, this requirement can be relaxed to only blocking certain actions of the group leavers. In general, if a process leaves its old group early, it does not wish to spread a recovery from its own group to its new group. To do that, it has to be blocked from sending any message to its new group, until the old group dismisses. At the same time, it also needs to log the messages received from its new group. In other words, it behaves as if it is not a member of the new group yet, and thus it has enough information logged to be able to recover with the old group alone (instead of involving other members of its new group). The checkpoint it takes upon leaving the old group is hence tentatively colorless until that group ends. In this strategy, instead of the complete execution blocking, a group leaver pays a price of extra logging, plus the potential blocking at the point of sending a message to the new group.

The above strategy can be further improved to become non-blocking. For a group leaver to be able to send messages to its new group safely, it also needs to perform sender-side logging of (the sequence numbers of) those messages. This will guarantee that if its old group recovers, it could discard such messages by itself. For the above example in Figure 5.5, upon the invitation message m_3 from the new 'red' group, p_2 leaves its old 'white' group by taking a colorless checkpoint, and then proceed tentatively with sender- and receiver-side logging of messages to and from the new group, respectively, e.g., m_5 and m_3. Note that it also perform receiver-side logging of message from its old 'white' group, e.g., m_4, due to the color difference between its current checkpoint and its old group. In case that a failure occurs in the old group, say in p_1, all previous group members including p_2 will roll back to their 'white' checkpoint. Since p_2

keeps a log for both incoming and outgoing messages of its new group (m_3 and m_5 respectively in this example), it is able to perform receiver-side playback and sender-side discard accordingly. Consequently, there is no recovery spread to its new group. On the other hand, if a failure occurs in its new group, say in p_3, recovery will roll back all members, including p_2, to their corresponding checkpoints. Since p_2 also has a log of incoming messages from its old group, e.g., m_4 in the figure, it could perform receiver-side playback accordingly and its old group will not be involved in the recovery as well. Note that m_6 will be replayed but will then be discarded by p_1 based on its receiver-side logging (m_6 is logged initially due to the color difference). The above redundant logging with respect to the new group will stop when the old group actually ends, i.e., all members have created a new checkpoint.

The non-blocking atomic group dismiss is feasible with the cost of further extra logging. A trade-off can be made based on estimation or perception of future interactions. For example, a future locality of frequent interactions with the new group members will encourage blocking instead of proceeding and logging.

In summary, to prevent recovery spread among group checkpoints, either process cloning or atomic group dismiss is required. The former provides choices between cost on recovery need and cost of standing by. The latter pays the price of execution blocking and/or extra logging of certain messages with respect to the successive new group.

5.4 Remarks

This chapter addresses performance-related issues in locality-driven checkpoint and recovery, and makes its contributions in three aspects. An abstract performance measure is first proposed from the perspective of decomposed recovery regions, and captures the major influential factors of distributed checkpoint and recovery. It is generic enough to

measure the performance of existing as well as the new group-based checkpoint strategies. It can also serve as an optimization objective to guide the development of group checkpoint protocols. Consequently, a distributed greedy scheme is proposed for dynamic detection of locality regions, in which locality regions are locally decidable through region merging. A merging criterion is deduced with discussion of its implications. In particular, the optimal decomposition is demonstrated to be related to interaction locality at a certain level. Good interaction locality leads to good performance optimality, and can serve as a good candidate for the optimal decomposition. Based on these results, this chapter develops runtime strategies for detecting locality regions in a given distributed execution. Availability of design time knowledge is discussed and a simple group checkpoint protocol is developed accordingly. On-the-fly detection of locality regions benefits from good interaction locality, and is demonstrated through another group checkpoint protocol. Finally, design choices for checkpoint protocols are discussed with respect to their performance costs. In particular, atomic group dismiss strategies are presented as alternatives to process cloning.

This chapter develops many of its results based on the abstract performance measure. The measure is simple and yet powerful enough to capture the essence of locality-driven checkpoint and recovery. On the other hand, it is based on two simple assumptions about uniform distribution of events and equal failure probability over events, which certainly have their limitations of applicability. The measure as well as the corresponding model can be amended by more complicated assumptions, e.g., with annotating actual execution time to each behavioral fragments, and using different failure probability distribution functions. However, this will only complicate the analysis process, and will not change the principle of locality or the applicability of the results developed in this chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Checkpoint and recovery provide application-transparent fault-tolerance to distributed systems. Existing strategies, checkpoint- and log-based recovery, suffer from either severe runtime performance downgrade or catastrophic global-wise recovery waste in large-scale distributed systems, due to the large numbers of processes and messages that could be involved. In this thesis, a locality-driven checkpoint and recovery strategy has been proposed to offer such systems benefits in both aspects, making it possible and efficient to achieve affordable runtime performance and controllable failure recoverability in a harmonious manner.

The two categories of existing checkpoint and recovery strategies have different consistency requirements due to their ways of handling message dependency relations between processes. By forcing a specific and uniform policy to all message dependencies, such a strategy gains advantage at one end but has to bear disadvantage at the other end. The theory of Quasi-Atomic Recovery hence has been formulated as a more generic foundation to accommodate the specific consistency requirements of both kinds. The quasi-atomicity property of recovery blocks covers all kinds of existing checkpoint and recovery techniques, and allows the use of different message handling methods together

at the same time. As a result, it is possible to selectively handle certain messages in one way and the rest in the other way. The theorem on confinement of recoveries within well-ordered quasi-atomic recovery blocks, on the other hand, provides an integrated view for investigating localizable checkpoints and recoveries, and consequently having a hybrid strategy with combined advantages from both ends. The Checkpoint Dependency Graph (CDG) model has also been presented to capture effective dependency relations among checkpoints, namely message dependency and program-order dependency. Correctness requirement of checkpoint protocols is therefore simplified as demanding properness of the corresponding CDG. In addition, a generic recovery protocol has been proposed with proved capability of performing correct recoveries.

Based on the observation of localized message interactions in distributed systems, the idea of group checkpointing and selective logging has been proposed and illustrated. The existence of 'locality regions' makes it feasible to manage group-wise coordinated checkpoints around subsets of processes that are tightly coupled via frequent message interactions, and to log only the few inter-group messages afterwards. Runtime overhead is optimized due to the reduced efforts of message logging, and recovery spread is localized via the proper handling of inter-group dependencies. Various group checkpoint protocols that make use of the locality phenomenon have been developed with proofs of correctness using the CDG model. These protocols employ simple and asynchronous designs such as message piggybacking, invitation-based group formation, and use of color tagging for group identification. They have respective features including strong-ness, atomicity, non-atomicity, and k-bounded-ness, providing efficient trade-offs between flexibility and performance. The technique of cloning-based recovery has also been introduced to effectively remove program-order dependency, hence to provide further design choices for limiting the growth of group checkpoints. Group checkpoint

protocols with cloneable checkpoints have recoveries easily confined within each group, which otherwise requires complicated protocol design and might not be always achievable.

The locality phenomenon of message interactions in distributed systems has been examined and exploited. Analysis has been performed to provide an overall view from the perspective of distributed program design and implementation. Like traditional locality of reference, interaction locality is resulted from practices of applying the generic divide-and-conquer strategy in problem solving. Global dependencies in distributed applications are parallelized and localized among processes, and are finally resolved via message passing. Distributed executions are hence likely to exhibit a hierarchical 'region-transition' pattern. Each region consists of a sequence of localized interactions from each participant process. Based on this observation, a bottom-up approach has been proposed to start identifying those regions from the perspective of individual processes. A generic notion of 'locality interval' is introduced to capture popular recurrence patterns within individual process lifelines. Frequent message interactions between such intervals are modeled as 'important coupling', by which they form a 'locality region' that contains localized messages. Concepts of qualified locality intervals and primary important coupling are also brought up to filter out trivialities and deviations that arise in practice. Test cases have been designed and experiments have been conducted accordingly on traces from two real-life distributed applications. Results and analysis have justified the existence of hierarchical locality regions and the feasibility of proposed approach as well.

Finally, performance optimization of group checkpoint strategies has been studied through demonstrating its relation to interaction locality. An abstract performance measure has been proposed to properly integrate the major aspects of both runtime overhead and failure recoverability in a region-wise manner. It hence can serve as an

optimization objective of the overall performance of distributed checkpoint and recovery. A greedy optimization heuristic is proposed to decompose a given distribution into a set of disjoint recovery regions, in which such regions are locally decidable through region merging as long as it improves the objective. A merging criterion is reasoned and it implies that performance optimality is related to interaction locality. A pattern with good locality leads to good performance, and the locality pattern itself can serve as a good candidate for the optimal decomposition. Subsequently, detailed strategies have been developed to detect locality patterns in a given distributed execution, and specific protocols are presented with assistance of either design time knowledge or runtime locality prediction. In addition, performance-related design choices for checkpoint protocols have been discussed with respect to their specific requirements and costs. Aiming at preventing recovery spread across group checkpoints, strategies for atomic group dismiss strategies are introduced in particular, with contrast to process cloning.

6.2 Future Work

This thesis has laid down the major milestones for achieving well-performing checkpoint and recovery strategies in a locality-driven manner. Besides, there are a number of directions for this work to continue with.

A straightforward way to examine the locality-driven checkpoint and recovery strategy is to conduct experiments and to compare the actual performance with other strategies. This is actually a piece of work that is already launched and is currently undergoing. In particular, a discreet event simulator is used as the experimentation platform, and a synthetic trace generator of distributed systems is built to prepare the input data. The simulator has been adapted to become a specific test-bed for checkpoint and recovery algorithms, with several traditional algorithms implemented. Proper test

cases are to be designed with careful examination of different runtime parameters. Execution traces with good locality should observe obvious out-performance of group checkpoint protocols over traditional ones. Large scale executions with hierarchical and complicated locality patterns might require proper fine-tuning of the group checkpoint protocols before the results could spell out themselves. In particular, the protocols based on runtime knowledge of locality properties would have to go through quite a few rounds of tests on their detection and prediction strategies, which could possibly raise challenging questions and lead to a next major research.

Further and thorough study on complicated distributed applications is a next-step necessity. The two example applications used in this thesis only show limited hierarchical structures due to the simplicity of their nature. Large scale and complicated applications are likely to be designed in a more organized hence more localized way. As a result, their executions are likely to exhibit more interesting locality properties that are worthwhile to study. For example, domain-specific applications should have their own domain-related locality patterns, which once being identified can also be very useable in many aspects.

The discussion in Chapter 5 leads to the idea of having a comprehensive development framework that can provide design time knowledge for locality-driven strategies. Such a project is complicated, as related information needs to be carefully examined at each level of details, with consideration of both feasibility and efficiency. However, interaction locality is a popular phenomenon and is important to performance of many real-life distributed problems. Availability of such a framework together with proper locality-driven strategies will greatly improve the performance and hence the attractiveness of related distributed applications.

Bibliography

[AAJ06] M. Aminian, M.K. Akbari, and B. Javadi. Coordinated checkpoint from message payload in pessimistic sender-based message logging. In *IPDPS'06: Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[ABC+96] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *PDIS'96: Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, Miami, FL, pages 92–103, 1996.

[ABL+07] Luca Allulli, Roberto Baldoni, Luigi Laura, and Sara Tucci Piergiovanni. On the complexity of removing Z-Cycles from a checkpoints and communication pattern. *IEEE Trans. Computers (TC)* 56(6):853-858, 2007.

[AER+99] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed A. Husain, and Asanka de Mel. An analysis of communication-induced checkpointing. In *FTCS'99: Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 242-249, Madison, Wisconsin, USA, 1999. IEEE Computer Society.

[AK98] Lorenzo Alvisi and Keith Marzullo. Message logging: pessimistic, optimistic, causal and optimal. *IEEE Trans. Software Eng.*, 24(2):149-159, 1998.

[AL98] Yariv Aridor and Danny B. Lange. Agent design patterns: elements of agent application design. In *Agents'98: Proceedings of the 2nd International Conference on Autonomous Agents*, pages 108-115, Minneapolis, Minnesota, 1998.

[Alv96] Lorenzo Alvisi. Understanding the message logging paradigm for masking

process crashes, Ph.D. Thesis, Cornell University, 1996.

[ATG08] Bharat B. Agarwal, Sumit P. Tayal, and M. Gupta. *Software Engineering & Testing: An Introduction*. Jones and Bartlett Publishers, Sudburry, MA, 2008.

[AW96] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. *SIGMETRICS Perform. Eval. Rev.*, 24(1):126-137, 1996.

[BCS84] D. Briatico, Augusto Ciuffoletti, and Luca Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, pages 207-215, 1984.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[BHR01] Roberto Baldoni, Jean-Michel Helary and Michel Raynal. Rollback-dependency trackability: a minimal characterization and its protocol. *Information and Computation*, 165:144-173, 2001.

[BK69] Laszlo A. Belady and C.J. Kuehner. Dynamic space sharing in computer systems. *Comm. ACM*, 12(5):282-288, 1969.

[BL88] Bharat K. Bhargava and Shy-Renn Lian. Independent checkpointing and concurrent rollback for recovery - an optimistic approach. In *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems*, pages 3-12, Columbus, OH, 1988.

[BLL89] Bharat K. Bhargava, Shy-Renn Lian, and Pei-Jyun Leu. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proceedings of the 6th International Conference on Data Engineering*, pages182-189, 1990.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: a formalism for specifying multiagent interaction. In *Proceedings of Agent-Oriented Software Engineering*, pages 91-103, 2001.

[BMP+03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill.

Automated application-level checkpointing of MPI programs. In *PPOPP'03: Proceedings of the 9th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 84-94, San Diego, California, USA, 2003.

[BP07] Ujwala Baruah and Himadri Sekhar Paul. A consistent global checkpoint and recovery protocol in cluster-based distributed systems. In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 64-68, 2007.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - FIPA compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, pages 97-108, 1999.

[BQC98] Roberto Baldoni, Francesco Quaglia, and Bruno Ciciani. A VP-accordant checkpointing protocol preventing useless checkpoints. In *Proceedings of The 7th Symposium on Reliable Distributed Systems*, pages 20-22, West Lafayette, Indiana, USA, 1998. IEEE Computer Society.

[CC00] Ludmila Cherkasova and Gianfranco Ciardo. Characterizing temporal locality and its impact on web server performance. In *Proceedings of 9th International Conference on Computer Communication and Networks*, pages 434-441, 2000.

[CL85] K. Mani Chandy, Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Computing Systems*, 3(1): 63-75, 1985.

[CS98] Guohong Cao, Mukesh Singhal. On coordinated checkpointing in distributed systems. *IEEE Trans. Parallel and Distributed Systems*, 9(12): 1213-1225, 1998.

[CS03] Guohong Cao, Mukesh Singhal. Checkpointing with mutable checkpoints. *Theoretical Computer Science*, 290(2): 1127-1148, 2003.

[CW06] Hailong Cai and Jun Wang: Exploiting geographical and temporal locality to boost search efficiency in peer-to-peer systems. *IEEE Trans. Parallel Distrib. Syst.*

(TPDS),17(10):1189-1203, 2006.

[Dee03] Sayyed M. Deen (Ed.). *Agent Based Manufacturing: Advances in Holonic Approach*, Springer, New York, 2003.

[Den68] Peter J. Denning. The working set model for program behavior. *Comm. ACM*, 11(5):323–333, 1968.

[Den70] Peter J. Denning. Virtual memory. *ACM Compting Surveys*, 2(3): 153-189, 1970.

[Den76] P.J. Denning. The working set model for program behavior. *Comm. ACM*, 19(5): 285- 294, 1976.

[Den05] Peter J. Denning. The locality principle. *Comm. ACM*, 48(7): 19-24, 2005.

[DK75] Peter J. Denning and Kevin C. Kahn. A study of program locality and lifetime functions. In *Proceedings of the 5th ACM Symp. on Operating System Principles*, pages 207–216, 1975.

[DS72] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Comm. ACM*, 15(3):191–198, 1972.

[DWK01] Dwight Deugo, Michael Weiss, and Elizabeth A. Kendall. Reusable patterns for agent coordination. In *Coordination of Internet Agents: Models, Technologies, and Applications 2001*, pages 347-368, Springer, 2001.

[EAW+02] 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): 375-408, 2002.

[EJZ92] E. N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symp. on Reliable Distributed Systems*, pages 39-47, 1992.

[EZ94] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *FTCS-24: Proceedings of the 24th International*

Symposium on Fault-Tolerant Computing, pages 298-307, Austin, Texas, 1994.

[FAC+03] Rodrigo C. Fonseca, Virgilio Almeida, Mark Crovella, and Bruno D. Abrahao. On the intrinsic locality properties of Web reference streams. In *INFOCOM 2003: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications*, pages 448-458, San Francisco, CA, 2003.

[FIP99] FIPA. FIPA'99 Specification Part 2: Agent Communication Language. http://www.fipa.org, 1999.

[FIP03] FIPA. FIPA Interaction protocol specifications. http://www.fipa.org, 2003.

[GFB05] Zahia Guessoum, Nora Faci, and Jean-Pierre Briot. Adaptive replication of large-scale multi-agent systems: towards a fault-tolerant multi-agent platform. *ACM SIGSOFT Software Engineering Notes (SIGSOFT)*, 30(4):1-6, 2005.

[GHK+07] Qi Gao, Wei Huang, Matthew J. Koop, and Dhabaleswar K. Panda. Group-based coordinated checkpointing for MPI: a case study on InfiniBand. In *ICPP'07: Proceedings of the International Conference on Parallel Processing*, pages 47, 2007.

[GMM98] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated electronic commerce: a survey. *Knowledge Engineering Review*, 13(2):147-159, 1998.

[GS99] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe'96, LNCS 1088*, pages 38-57, Springer, 1996.

[GSP99] Dhrubajyoti Goswami, Ajit Singh and Bruno R. Preiss. Architectural skeletons: the re-usable building-blocks for parallel applications. In *PDPTA'99: Proceedings of the 1999 International Conference on Parallel and Distributed Processing, Techniques and Applications*, pages 1250-1256, Las Vegas, NV, 1999.

[GSP02] Dhrubajyoti Goswami, Ajit Singh and Bruno R. Preiss. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing (JPDC)*,

62(4): 669-695, 2002.

[GW03] Dawn G. Gregg and Steven Walczak. E-commerce auction agents and online-auction dynamics. *Electronic Markets (ELECTRONICMARKETS)* 13(3), 2003.

[Hag96] Staffan Hagg. A sentinel approach to fault handling in multi-agent systems. In *Proceedings of the 2nd Australian Workshop on Distributed AI*, pages 181-195, Cairns, Australia, 1996.

[HMN+97] Jean-Michel Hélary, Achour Mostéfaoui, Robert H. B. Netzer, and Michel Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the 16th IEEE Intl. Symp. on Reliable Distributed Systems*, pages 183-190, Durham, NC, USA, 1997.

[HMR97a] Jean-Michel Hélary, Achour Mostéfaoui and Michel Raynal. Cycle Prevention in Distributed Checkpointing. In *PODS'97: Proceedings of the 1997 International Conference on Principles Of Distributed Systems*, pages 309-318, Chantilly, France, 1997.

[HMR97b]Jean-Michel Hélary, Achour Mostéfaoui and Michel Raynal. Virtual precedence in asynchronous systems: concepts and applications. In *WDAG'97: Proceedings of the 11th Workshop on Distributed Algorithms*, pages 170-184, 1997.

[HMR99] Jean-Michel Hélary, Achour Mostéfaoui and Michel Raynal. Consistency issues in distributed checkpoints. *IEEE Trans. Software Eng.*, 25(2): 274-281, 1999.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2003

[HSP05] Mohammad Tanvir Huda, Heinz W. Schmidt, and Ian D. Peake. An agent oriented proactive fault-tolerant framework for grid computing. In *e-Science 2005: Proceedings of the 1st International Conference on e-Science and Grid Technologies*, pages 304-311, Melbourne, Australia, 2005.

[HRV+08] Brahim Hamid, Ansgar Radermacher, Patrick Vanuxeem, Agnes Lanusse, and Sebastien Gerard. A fault-tolerance framework for distributed component systems. In *SEAA 2008: Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, pages 84-91, Parma, Italy, 2008.

[HWL08] Justin C. Y. Ho, Cho-Li Wang and Francis C.M. Lau. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *IPDPS'08: Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[JB00a] Shudong Jin and Azer Bestavros. Sources and characteristics of Web temporal locality. In *MASCOTS 2000: The IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 28-35, San Francisco, CA, 2000.

[JB00b] Shudong Jin and Azer Bestavros. Temporal locality in Web request streams: Sources, characteristics, and caching implications (extended abstract). In *Proceedings of the 2000 ACM SIGMETRICS Conference*, pages 110-111, Santa Clara, CA, 2000.

[JKN06] Taichi Jinno, Tokimasa Kamiya, and Motoyasu Nagata. Coordinated checkpointing using vector timestamp in grid computing. In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 710-716, 2006.

[JLM08] Qiangfeng Jiang, Yi Luo and D. Manivannan. An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems. *Journal of Parallel and Distributed Computing*, 68(12): 1575-1589, 2008.

[JR86]Raj Jain and Shawn Routhier. Packet trains - measurements and a new model for computer network traffic. *IEEE Journal of Selected Areas in Communications*, SAC-4(6): 986-995, 1986. Reprinted in Amit Bhargava (Ed.). *Integrated Broadband Networks.*

Artech House, Norwood, MA, 1990.

[JZ87] David B. Johnson and Willy Zwaenepoel, Sender-based message logging. In *Digest of Paper: 17th Symp. on Fault-Tolerant Computing*, pages 14-19, IEEE Computer Society, 1987.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171-181, Toronto, ON, Canada, 1988.

[KCL00] Sanjeev Kumar, Philip R. Cohen, and Hector Levesque. The adaptive agent architecture: achieving fault-tolerance using persistent broker teams. In *ICMAS 2000: Proceedings of the 4th Int'l Conf. on Multiagent Systems*, pages 159-166, Boston, 2000.

[KEL+62] Tom Kilburn, David B.G. Edwards, Michael J. Lanigan, and Frank H. Sumner. One level storage system. *IRE Trans. Electronic Computers*, EC-11(2): 223- 235, 1962.

[Ken98] Elizabeth A. Kendall. Agent roles and aspects. In *Proceedings of ECOOP Workshops*, pages 440, LNCS 1543, Springer-Verlag, 1998.

[Ken00] Elizabeth A. Kendall. Agent software engineering with role modeling. In *AOSE-2000: Proceedings of the 1st International Workshop*, pages 163-170, Springer-Verlag, Berlin, Germany, 2000.

[KRD03] Mark Klein, Juan A. Rodríguez-Aguilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: the case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179-189, 2003.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Trans. Software Eng.*, 13(1): 23-31, 1987.

[Kum08] Parveen Kumar: A low-cost hybrid coordinated checkpointing protocol for

mobile distributed systems. *Mobile Information Systems (MIS)*, 4(1):13-32, 2008.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21(7): 558-565, 1978.

[LB88] Pei-Jyun Leu and Bharat K. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *ICDE'88: Proceedings of the 4th Int'l Conf. on Data Engineering, Los Angeles*, pages 154-163, Los Angeles, CA, 1988.

[LCK+06] Feifei LI, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE'06: Proceedings of the 22nd International Conference on Data Engineering*, pages 81-92, 2006.

[Liu01] Jiming Liu. *Autonomous Agents and Multi-Agent Systems: An Introduction*, World Scientific, Singapore, 2001.

[LWD06] Hon F. Li, Zunce Wei, and Dhrubajyoti Goswami. Quasi-atomic recovery for distributed agents. *Parallel Computing*, 32(10): 733-758, 2006.

[Mas77] Takashi Masuda. Effect of program localities on memory management strategies. *ACM SIGOPS Operating Systems Review*, 11(5): 117-124, 1977.

[Mat71] Richard L. Mattson. Evaluation of multilevel memories. *IEEE Trans. on Magnetics*, 7(4): 814-819, 1971.

[MB76] A. Wayne Madison and Alan P. Bates. Characteristics of program localities. *Comm ACM*, 19(5):285--294, 1976.

[MBS03] Olivier Marin, Marin Bertier, and Pierre Sens. DARX - a framework for the fault-tolerant support of agent software. In *ISSRE 2003: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 406-418, 2003.

[Mei08] Amnon Meisels. *Distributed search by constrained agents: algorithms, performance, communication*. Springer-Verlag, London, UK, 2008.

[MEW00] Anirban Mahanti, Derek Eager, and Carey Williamson. Temporal locality and

its impact on Web proxy cache performance. *Performance Evaluation*, Special Issue on Performance Modelling, 42(2-3):187–203, 2000.

[MGS+70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[MJY+05] D. Manivannan, Qiangfeng Jiang, Jianchang Yang, Karl E. Persson, Mukesh Singhal: An Asynchronous Recovery Algorithm Based on a Staggered Quasi-Synchronous Checkpointing Algorithm. In *IWDC 2005: Proceedings of the 7th International Workshop on Distributed Computing*, pages 117-128, LNCS 3742, 2005.

[MLY06] Romeo Mark A. Mateo, Jaewan Lee, and Hyunho Yang. Optimization of location management in the distributed location-based services using collaborative agents. In *ICCSA 2006: Proceedings of International Conference on Computational Science and Its Applications*, pages 178-187, Glasgow, UK, 2006.

[MM05] Partha S. Mandal and Krishnendu Mukhopadhyaya. Performance analysis of different checkpointing and recovery schemes using stochasticmodel. *Journal of Parallel and Distributed Computing*, 66(1): 99 - 107, 2006.

[MS96] D. Manivannan and Mukesh Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *ICDCS'96: Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 100-107, Hong Kong, 1996.

[MSA+08] Edleno Silva de Moura, Célia Francisca dos Santos, Bruno Dos Santos de Araujo, Altigran Soares da Silva, Pável Calado, and Mario A. Nascimento. Locality-based pruning methods for web search. *ACM Trans. Inf. Syst. (TOIS)*, 26(2), 2008.

[MSM08] John Mehnert-Spahn, Michael Schöttner, and Christine Morin. Checkpointing process groups in a grid environment. In *PDCAT'08:Proceedings of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 243-251, 2008.

[NF96] Nuno Neves and W. Kent Fuchs. Using time to improve the performance of coordinated checkpointing. In *IPDS'96: Proceedings of the 2nd International Computer Performance and Dependability Symposium*, pages 282, 1996.

[NF98] Nuno Neves and W. Kent Fuchs. RENEW: a tool for fast and efficient implementation of checkpoint protocols. In *Proceedings of the 28th IEEE Fault-Tolerant Computing Symposium*, pages 58-67, 1998.

[NX95] Robert H.B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel and Distributed Systems*, 6(2): 165-169, 1995.

[OMG03] OMG. UML 2.0 Superstructure Specification. *OMG document* ptc/03-08-02, OMG, Framingham, MA, 2003.

[OPB00] James Odell, H. Van Dyke Paranak, and Bernhard Bauer. Extending UML for agents. In *AOIS Workshop at AAAI 2000: Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3-17, Austin, TX, USA, 2000.

[PBC01] H. Van Dyke Parunak, Albert D. Baker, and Steven J. Clark. AARIA agent architecture: from manufacturing requirements to agent-based system design. *Integr. Computer-Aid. Eng.*, 8(1): 45–58, 2001.

[PBK+95] James S. Plank, Micah Beck, Gerry Kingsley, Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of Usenix Winter 1995 Technical Conference*, pages 213-223, New Orleans, LA, 1995.

[PGB01] Hamadri S. Paul, Arobinda Gupta, and R. Badrinath. Evaluation of Different Classes of Checkpoint and Recovery Protocols with dPSIM. In *Proceedings of the International Conference on Information Technology*, pages 315 - 320, 2001.

[Pla93] James S. Plank. Efficient checkpointing on MIMD Architectures, Ph.D. Thesis,

Princeton University, 1993.

[Pla96] James S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pages 76 – 85, 1996.

[Pla97] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report of University of Tennessee, UT-CS-97-372, 1997.

[PPG00] Holger Pals, Stefan Petri, and Claus Grewe. FANTOMAS: fault tolerance for mobile agents in clusters. In *Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1236-1247, Cancun, Mexico, LNCS 1800, Springer-Verlag, 2000.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Trans. Software Eng.*, 1(2): 220-232, 1975.

[RAV99] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: an extensible toolkit for low-overhead fault-tolerance. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 48, 1999.

[RUI97] B. Ramamurthy, S.J. Upadhyaya, and R.K. Iyer. An ooject-oriented testbed for the evaluation of checkpointing and recovery systems. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 194, 1997.

[SDP+96] Katia Sycara, Keith Decker, Anandeep Pannu, Mike Williamson, and Dajun Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6): 36-46, 1996.

[SE05] Yunus Emre Selçuk and Nadia Erdogan. A role model for description of agent behavior and coordination. In *ESAW 2005: Proceedings of the 6th International Workshop on Engineering Societies in the Agents World VI*, pages 29-48, Kusadasi, Turkey, LNCS 3963, Springer, 2005.

[SEA+07] Iman Saleh, Mohamed Eltoweissy, Adnan Agbaria, and Hesham El-Sayed. A fault tolerance management framework for wireless sensor networks. *Journal of Communications*, 2(4):38-48, 2007.

[Sen97] Pierre Sens. Performance evaluation of fault tolerance for parallel applications in networked environments. In *ICPP'97: Proceedings of the 1997 International Conference on Parallel Processing*, pages 334, 1997.

[SG06] Tiemi C. Sakata and Islene C. Garcia. Non-blocking synchronous checkpointing based on rollback-dependency trackability. In *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 411-420, 2006.

[SGP+05] Rodrigo Schmidt, Islene C. Garcia, Fernando Pedone, and Luiz Eduardo Buzato. Optimal asynchronous garbage collection for RDT checkpointing protocols. In *ICDCS'05: Proceedings of the 25th International Conference on Distributed Computing Systems*, pages 167-176, 2005.

[SL09] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: algorithmic, game-theoretic, and logical foundations*. Cambridge Univeristy Press, Cambridge, UK, 2009.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Computer Systems*, 1(3): 222—238, 1983.

[SS99] Luis M. Silva and Joao G. Silva. The performance of coordinated and independent checkpointing. In *IPPS/SPDP'99: Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 280 – 284, 1999.

[SSS+99] Louis M. Silva, Paulo Simões, Guilherme Soares, Paulo Martins, Victor Batista, Carlos Renato, Leonor Almeida, and Norbert Stohr. JAMES: a platform of mobile agents for the management of telecommunication networks. In *IATA'99: Proceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications*, pages

76-95, LNCS 1699, Springer-Verlag, 1999.

[SY85] Robert E. Strom and Shaula A. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Computer Systems*, 3(3): 204-226, 1985.

[THT98] Katsuya Tanaka, Hiroaki Higaki, and Makoto Takizawa. Object-based checkpoints in distributed systems. *J. of Computer Systems Science and Engineering*, 13(3): 125-131, 1998.

[TM05] Tongchit Tantikul and D. Manivannan: A communication-induced checkpointing and asynchronous recovery protocol for mobile computing systems. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 70-74, 2005.

[TS84] Yuval Tamir and Carlo H. Sequin. Error recovery in multicomputers using global checkpoints. In *ICPP'84: Proceedings of the 13th Int'l Conf. on Parallel Processing*, pages 32-41, Bellaire, MI, 1984.

[Tsa03] Jichiang Tsai: On properties of RDT communication-induced checkpointing protocols. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 14(8):755-764, 2003.

[Vai99] Nitin H. Vaidya. Staggered consistent checkpointing, IEEE Trans. Parallel and *Distributed Systems*, 10(7): 694-702, 1999.

[Van05] Sarut Vanichpun. *Comparing Strength of Locality of Reference: Popularity, Temporal Correlations, and Some Folk Theorems for the Miss Rates and Outputs of Caches*. Ph.D. Thesis, University of Maryland, 2005.

[Wan93] Yi-Min Wang. *Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems*, Ph.D. Thesis, University of Illinois, 1993.

[Weg90]Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1)7–87, 1990.

[WGP03] Thomas Wagner, Valerie Guralnik, and John Phelps. TÆMS agents: enabling

dynamic distributed supply chain management. *Electronic Commerce Research and Applications (ECRA)*, 2(2):114-132, 2003.

[WLD+04] Lin Wang, Hon F. Li, Dhrubajyoti Goswami, and Zunce Wei. A fault-tolerant multi-agent development framework. In *ISPA'04: Proceedings of the 2nd Int'l Symp. on Parallel and Distributed Processing and Applications*, pages 126-135, Hong Kong, LNCS 3358, Springer-Verlag, 2004.

[WLG04] Zunce Wei, Hon F. Li, and Dhrubajyoti Goswami. Composable skeletons for parallel programming. In *PDPTA'04: Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1256-1261, Las Vegas, NV, 2004.

[WLG05] Zunce Wei, Hon F. Li, and Dhrubajyoti Goswami. Cloning-based checkpoint for localized recovery. In *I-SPAN 2005: Proceedings of the 8th Int'l Symp. on Parallel Architectures, Algorithms and Networks*, 174-181, Las Vegas, NV, 2005.

[WLG06] Zunce Wei, Hon F. Li, and Dhrubajyoti Goswami. A Locality-Driven Atomic Group Checkpoint Protocol. In *PDCAT 2006: Proceedings of the 7th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 558-564, Taipei, Taiwan, 2006.

[XRR+95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS'95: Proceedings of the 25th Symposium on Fault-Tolerant Computing*, pages 499–508, Pasadena, CA, 1995.

[ZHK06] Gengbin Zheng, Chao Huang, Laxmikant V. Kalé: Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++. *Operating Systems Review (SIGOPS)* 40(2):90-99, 2006.