Fault-tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions

Karpjoo Jeong
January, 1996

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved: _____

Professor Dennis Shasha
Research Advisor

ABSTRACT
Fault-tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions

Karpjoo Jeong

Research Advisor: Professor Dennis Shasha

With the advent of high performance workstations and fast LANs, networks of workstations have recently emerged as a promising computing platform for long-running coarse grain parallel applications. Their advantages are wide availability and cost-effectiveness, as compared to massively parallel computers. Long-running computation in the workstation environment, however, requires both fault tolerance and the effective utilization of idle workstations.

In this dissertation, we present a variant of Linda, called Persistent Linda (PLinda), that treats these two issues uniformly: specifically, PLinda treats non-idleness as failure.

PLinda provides a combination of checkpointing and transaction support on both data and program state (an encoding of continuations). The traditional transaction model is optimized and extended to support robust parallel computation. Treatable failures include processor and main memory hard and slowdown failures, and network omission and corruption failures.

The programmer can customize fault tolerance when constructing an application, trading failure-free performance against recovery time. When creating a PLinda program, the programmer can decide on the frequency of transactions and the encoding of continuations to be saved upon transaction commit. At runtime, the programmer can decide to suppress certain continuations for better failure-free performance.

PLinda has been applied to corporate bond index statistics computation and biological pattern recognition. Typical speedups over conventional sequential programs range over 15 times for 30 processors.

Acknowledgments

I have been very fortunate to have had Prof. Dennis Shasha as my advisor during my Ph.D. study. For the last six years, he has not only given me many precious opportunities for interesting research problems but also taught me how to tackle those problems. I am extremely grateful to him for helping me out whenever I needed his help.

I would like to give special thanks to friends at Courant for their friendship: Nick Afshartous, Arash Baratloo, Shih-Hua Chao, Churngwei Chu, Yaw-Tai Lee, Bin Li, Jyh-Jong Liu, Peter Piatko, Suren Talla, Chih-Chien Tu, Tsong-Li Wang, Roman Yangarber, Chi Yao, and Peter Wyckoff (in the alphabetical order). I feel grateful to Suren Talla and Peter Wyckoff for reading my thesis draft and giving me a lot of good comments. Suren has also made a great deal of contribution to the development of the current PLinda prototype system. Tsong-Li Wang and I have been working on various projects together and he has always been a great inspiration to me. Yaw-Tai Lee, Jyh-Jong Liu, Chih-Chien Tu and Chi Yao have always treated me like one of their Chinese friends.

I would like to give many thanks to friends at the Korean Graduate Students Association, NYU, especially, Jungsup Kim and Jae Woo Kim. I would also like to thank my two ex-roommates, June-Yub Lee and Joonsoo Choi. They put up with my habit to work late (often until 3:00am or 4:00am).

Finally, I would like to thank my parents, especially my mother, for their unlimited and unconditional love and trust. Without them, I could not have made it. There are two special friends, Jung Woo and Haekyung Kim that I must express my sincere gratitude. They have given me more than I could expect from any friend.

January, 1996
Karpjoo Jeong

Contents

1 Introduction **1**

1.1 Motivation 1
1.2 Networked Workstations as a Parallel Computing Platform 2
1.3 Developing Fault-tolerant Software 3
1.3.1 Failure Model 3
1.3.2 Program Structuring Paradigms 3
1.3.3 Fault-tolerant Programming Languages 4
1.4 Linda 5
1.4.1 Brief Review of Linda 5
1.4.2 Linda and Fault Tolerance 6
1.4.3 Linda and Transactions 6
1.5 Persistent Linda 2.0 6
1.6 Outline of the Dissertation 8

2 Persistent Linda **9**

2.1 Introduction 9
2.2 Transactions and Robust Parallel Computation in the Linda Model 10
2.3 Transactions in PLinda 12
2.4 Fault-tolerant Multiple Tuple Spaces 18
2.5 Continuation Committing 22
2.6 Tuple Groups: a namespace management issue 23
2.7 Process Management 26
2.8 Related work 27
2.8.1 Fault Tolerance Work on Linda 27
2.8.2 Programming Languages and Systems Supporting Transactions . . 29
2.9 Summary 30

3 Tunable Execution **32**

3.1 Introduction 32
3.2 Motivation 33
3.3 Commit-consistent Execution 34
3.4 Message Logging/Replay 36
3.5 Coordinated Checkpointing 38
3.6 Correctness Proof 40

3.6.1 Proof . . . 42
3.7 Related Work . . . 45
3.8 Summary . . . 46

4 Using Idle Workstations for Parallel Computation 48
4.1 Introduction . . . 48
4.2 Finding Idle Workstations and Scheduling Processes . . . 49
4.2.1 Issues . . . 49
4.2.2 The Current PLinda Design . . . 50
4.3 Process Resiliency as Process Migration . . . 51
4.4 PLinda Infrastructure . . . 52
4.4.1 Major Runtime System Components . . . 52
4.4.2 Parallel Virtual Machine in PLinda . . . 53
4.5 Related Work . . . 54
4.6 Summary . . . 55

5 Implementation and Experiments 57
5.1 PLinda Server . . . 57
5.1.1 Architecture . . . 58
5.1.2 Tuple Space Management . . . 59
5.1.3 Transactions . . . 62
5.1.4 Process Management . . . 63
5.1.5 Tunable Execution of Continuation Committing . . . 66
5.1.6 Tuple Space Checkpointing . . . 68
5.2 Daemon Processes . . . 68
5.3 Administration Process . . . 69
5.4 Experiments . . . 71
5.4.1 Performance of PLinda primitive operations . . . 71
5.4.2 Performance of the Three Execution Methods . . . 72
5.4.3 Biological Pattern Discovery . . . 76
5.4.4 Corporate Bond Index Statistics . . . 80
5.5 Summary . . . 83

6 Conclusions and Future Work 84
6.1 Future Work . . . 86

Bibliography 87

List of Figures

2.1 Execution of A Process as Multiple Transactions 13
2.2 Master/Worker Programming Model 15
2.3 A Fault Tolerant Worker Process 16
2.4 A Master Process 17
2.5 Checkpoint-protected Tuple Space 20
2.6 Stable Tuple Space 21
2.7 A Fault Tolerant Master Process 24
2.8 Tuple Space Creation and Destruction in PLinda 25
2.9 Producer in PLinda 27
2.10 Consumer in PLinda 28

3.1 Commit-consistent Execution 35
3.2 Message Logging and Replay 37
3.3 Coordinated Checkpointing 39

4.1 Parallel Virtual Machine in PLinda 54

5.1 Server Architecture 59
5.2 Structure of a Tuple Group 61
5.3 Special Tuple Groups for Process Management 66
5.4 Algorithm for the PLinda daemon process 69
5.5 Performance of failure recovery 75
5.6 Performance results of the PLinda biological pattern discovery program with seven Sparc5's 78
5.7 Performance results of the PLinda biological pattern discovery program on 30 Sparc5's at AT&T Bell Labs in Whippany 79
5.8 Performance of the PLinda Bond Index Statistics Computation Program with Seven Sparc5's 81
5.9 Performance of the PLinda Bond Index Statistics Computation Program with 45 Machines 82

Chapter 1

Introduction

1.1 Motivation

Many scientific and engineering problems from Biology, Physics and other areas require computing a large number of mostly independent tasks[30, 32, 50, 63, 64]. A typical example is the analysis of a large (high energy) physics data set[30, 32]. The data set consists of $10^5 - 10^8$ event records which can be analyzed independently. The analysis of each event record is usually compute-intensive. Regarding these problems, we observe:

- Parallel processing is indispensable due to the enormous amount of computation.

  Fortunately, they are *coarse grain parallel* or *embarrassingly parallelizable*. That is, it is straightforward to parallelize them into large seldomly interacting tasks. Unfortunately, the necessary process interaction is tedious and error-prone if done manually.

- Fault tolerance is crucial because execution of these applications takes a lot of time even with parallel computers. The probability of failure grows as execution time or the number of processors increases. Although a single processor failure is in fact rare, the possibility of failure cannot be ignored if execution time is on the order of months or the execution involves a few hundred processors. Without fault tolerance, a single component failure can cause an entire computation to be lost.

  Fortunately, there is a great potential to easily customize lightweight fault tolerance for these problems. For example, atomic execution of each task is natural for these problems because each task can be executed independently of other tasks. Given fault tolerance abstractions such as transactions and reliable storage, we can maintain the description of all the tasks and all the results collected from the completed tasks in reliable storage and design processes to execute each task atomically. In fact, such a scheme has already been demonstrated in fault-tolerant parallel computing systems such as FT-Linda[4] and PLinda[38].

- Using workstations connected by LANs or even WANs, large scale high performance parallel processing is possible for these problems because computation basically consists of a large number of mostly independent tasks.

However, it is difficult for the end-user to find workstations which are idle for a long time. Workstations are only intermittently idle as a rule. A system which can utilize intermittently idle workstations can make computing on networked workstations very cost effective. In fact, for sequential or semi-parallel (i.e., multiple tasks with no inter-dependency) jobs, there are systems[2, 14, 18, 11, 25, 28, 44, 48, 52, 53] to utilize idle or under-utilize workstations effectively. For parallel computation, there are also systems[2, 11, 25, 44].

In the last several years, there has been a proliferation of commercial and research prototype parallel software systems on networks of workstations. Popular systems include Linda[13], PVM[60, 24], MPI[29] and Express. Unfortunately, few support fault tolerance or utilization of idle workstations.

Also, there has been a considerable amount of work [37, 51] on fault tolerance in distributed systems, but most of the work has not addressed the problem of utilizing idle workstations for parallel computation.

There have been research efforts[2, 14, 18, 11, 25, 48, 53] to develop software systems to utilize idle workstations and some of them[2, 14, 11, 25] are designed to address fault tolerance. However, to our knowledge, there is no system to aim at supporting both parallel processing and fault tolerance together with the utilization of idle workstations.

Addressing the three challenges —parallel processing on non-shared memory machines, fault tolerance, and effective use of intermittently idle machines — is what this dissertation describes for the first time. We present a Linda-variant parallel computing system, called Persistent Linda 2.0 (hereafter, just PLinda), which both supports fault tolerance and utilizes idle workstations.

1.2 Networked Workstations as a Parallel Computing Platform

Recently, networks of workstations have emerged as a promising parallel computing platform. Their advantages over massively parallel computers are wide availability and cost-effectiveness. First, unlike supercomputers installed in a few institutions, these machines are widely available; many institutions have hundreds of high performance workstations which are unused most of the time[25, 52]. Second, they are already paid for and are connected via communication networks; no additional cost is required for parallel processing. Finally, they can rival supercomputers with their aggregate computing power and main memory.

However, most machines are private (they are usually sitting on people's desks and are supposed to be used by them) and the owners of these workstations are afraid to allow compute-intensive jobs to be run on their machine for fear of performance degradation when they do want to use their machine. Therefore, it is crucial to guarantee that workstations will be used only while they are idle.

To address this issue, various *work-stealing* systems have been developed[2, 14, 18, 11, 25, 28, 44, 48, 52, 53]. During execution, these systems monitor the idleness status of

workstations and migrate processes from busy or overloaded machines to idle or underutilized ones. We will discuss work-stealing systems in detail in Chapter 4.

1.3 Developing Fault-tolerant Software

Developing fault-tolerant software on a distributed network of workstations is a challenging task for two main reasons. First, any part of the system can fail at any time; in other words, the fault tolerant software developer must anticipate numerous possible failure cases. Second, it is often very costly to find a consistent global state in a distributed system, yet fault tolerance requires such a notion.

To address these problems, various techniques such as failure models and program structuring paradigms have been developed. In this section, we will describe these techniques and give a review of programming languages designed for constructing fault tolerant software (which we call *fault-tolerant programming languages*).

1.3.1 Failure Model

Failure models are designed to specify the behavior of a component on failure and therefore to help the fault-tolerant software developer to reason about the effects of failure on applications.

Failure models about processors[20] which are commonly used are (from least permissive to most permissive):

- *Fail-stop*. The processor fails by stopping without making any inconsistent state transitions[55].
- *Omission* and *timing*. The processor fails by not responding to an input or by giving an untimely response, respectively.
- *Byzantine*. The processor fails in an arbitrary manner.

In general, fault-tolerant software is designed to assure correct behavior in the face of failures characterized by a failure model. Databases and most fault-tolerant parallel applications are designed to achieve partial correctness (any completed computation will have the same effect as a failure-free computation) in the face of fail-stop failures.

1.3.2 Program Structuring Paradigms

Program structuring paradigms provide the programmer with standard ways of writing programs. There are three common program structuring paradigms for fault-tolerant software[51]: the object/action model, the restartable action paradigm, and the replicated state machine paradigm.

In the *object/action* model, an application program consists of objects and actions. Objects encapsulate critical data in local state and export certain operations to modify data. Typically, the data is assumed to be long-lived and stored on stable storage.

Actions are threads that execute on objects, and their execution is transactional. That is, these actions are serializable and recoverable in spite of failure.

In the *restartable action paradigm*, the runtime system periodically saves the local states of processes to stable storage. On processor failure, it restarts failed processes on another processor by recovering their states from stable storage. The checkpointing and rollback scheme[22] is the most commonly used technique to implement this paradigm.

In the *replicated state machine* paradigm[57], an application is structured as a set of services, and each service is implemented as multiple deterministic processes which are identical. Each request for a service is broadcast to every process implementing the service. Each process operates like a state machine which modifies its state variables in response to commands (i.e., requests) that are received from other state machines or the environment. In this way, every process has the same state in a failure-free execution. Upon disagreement, the minority is ignored.

1.3.3 Fault-tolerant Programming Languages

Various fault-tolerant programming languages have been developed to ease the task of constructing fault-tolerant programs. Examples are Argus[47], Avalon[26], Fault-tolerant Concurrent C[19], FT-Linda[4], Orca[41] and FT-SR[56],

In general, these fault-tolerant programming languages are distinguished by what program structuring paradigms they support since they all assume the fail-stop processor failure model. Argus and Avalon support the object/action model. Reliability and concurrency control are supported by saving local state to disk and running every object invocation as a transaction or a nested transaction.

Fault-tolerant Concurrent C[19] and FT-Linda[4] support the replicated state machine paradigm. The primary extension of fault-tolerant Concurrent C which extends Concurrent C[33] is a set of primitives for replicating processes. The runtime system guarantees that all the replicas of a process behave as if they were a single process. FT-Linda is a fault-tolerant variant of Linda. Rather than using the state machine paradigm to replicate processes, FT-Linda uses it to replicate the Linda shared memory.

Programming languages to support the restartable action paradigm usually use mechanisms for checkpointing processes to disk and recovering them from the last checkpoints on disk. The language runtime system can handle checkpointing and rollback programmer-transparently. Orca is a language that automatically checkpoints parallel applications. It uses reliable broadcast to ensure that a globally consistent checkpoint is taken.

FT-SR is a language designed to support multiple program structuring paradigms. It provides an ordered multicast mechanism that can be used to implement the replicated state machine paradigm. Also, the programmer can define variables to be stable; these variables are stored on disk and survive failure. Using these mechanisms, the programmer can implement a custom restartable action paradigm.

1.4 Linda

PLinda is a set of extensions to Linda designed to support robust parallel computation. The primary extension is transactions. In this section, we will give a brief review of the Linda model and discuss the characteristics of the model which make it suitable for fault tolerance and transactions.

1.4.1 Brief Review of Linda

Linda is a parallel programming model which is based on virtual shared memory called *tuple space*. Processes communicate and synchronize by creating data objects called tuples in the tuple space (this is analogous to sending a message) and retrieving them from the tuple space *associatively* (this is analogous to receiving a message) using a relational database-style pattern matching capability. A tuple contains a sequence of typed data elements where the data types are basic types such as integers, floats, characters, and arrays of these.

Linda provides four operations: `out` for tuple creation, `eval` for process creation, `in` for destructive retrieval and `rd` for non-destructive retrieval. In the destructive case, data objects are removed on retrieval. Destructive retrieval is often used for a "point-to-point" style of communication; in contrast, non-destructive retrieval for a "broadcast" style of communication.

Here is a summary of the operations:

1. **out**. Takes a sequence of typed expressions as arguments. It evaluates them, constructs a tuple from them, and inserts the tuple into tuple space.

2. **eval**. Like `out`, `eval` creates a tuple from its arguments, but a new process is created to evaluate the arguments. In Linda, this is the only way to create a new process.

3. **in** and **rd**. Take a typed pattern for a tuple as their argument and retrieve a tuple to match the pattern in an associative manner. A pattern is a series of typed fields; some are values and others are typed place-holders. A place-holder is prefixed with a question mark. For example,

   ```
   ("foo", ?f, ?i, y).
   ```

   The first and last fields are values (a constant and a program variable respectively); the middle two fields are place-holders. On retrieval, place-holders are set to values in the corresponding fields of the matching tuple, respectively.

   This pattern will match any tuple having four fields whose first field is the string `"foo"` and the last field is the value of `y`. If there are multiple matching tuples, then one of them is randomly selected and retrieved. If no matching tuple is found, then `in` and `rd` block until a matching tuple is inserted. The difference is that `in` is destructive (i.e. removes the tuple) while `rd` is not.

A more detailed description of the Linda model is found in [9, 12, 13, 44, 46].

Linda has several characteristics which make it popular. First, the model is simple. Accessing tuple space is intuitive and requires only four operations. Second, the model is flexible. Various styles of process interaction such as synchronous and asynchronous communication can be easily programmed in this model. Finally, the model is designed to be combined into existing sequential programming languages. The programmer is not forced to learn a new parallel programming language.

1.4.2 Linda and Fault Tolerance

Linda's tuple space model facilitates fault tolerance. Communication and synchronization via the tuple space are anonymous. That is, processes do not have to identify each other for interaction. This property simplifies the replacement of processes by new processes, since process ids need not be recovered nor are site ids of any significance. Failed processes can be recovered on any host. In fact a stable tuple space and knowledge of the critical portions of the state of the failing process are all that is needed.

1.4.3 Linda and Transactions

Transactions are in particular effective for Linda's tuple space abstraction because tuple space is a shared data resource much like a database[34, 6]. Tuple space can be optimized for transactions independently because it is a separate data storage from process address spaces. Finally, in tuple space, there is a logical unit of data (which is a tuple) for manipulation which is independent of a language type system. There are only four operations to access tuples in tuple space. Therefore, only these operations are required to be extended for transactions.

1.5 Persistent Linda 2.0

In this section, we give an overview of Persistent Linda 2.0 (PLinda), a Linda-variant parallel programming system that supports fault tolerance and uses idle workstations for parallel computation.

PLinda is a set of extensions to Linda. The three major extensions are: *lightweight transactions*, *continuation committing*, and *checkpoint-protected tuple space*. These extensions are fault tolerance abstractions through which the programmer can design an application to be fault tolerant.

The mechanisms allow a programmer to take advantage of the characteristics of a given application when making it failure-resilient. However, the drawback is additional programming work. For coarse grain parallel applications which have simple control structures, the additional programming work is usually negligible [4, 38] and therefore the customization approach is more appropriate.

In PLinda, the transaction commit mechanism stores data in the volatile tuple space

and is therefore extremely lightweight[1]. Such a commit mechanism does not guarantee the durability of committed transactions, since the tuple space might fail. Tuple space is made fault-tolerant by checkpointing it to disk periodically. The frequency of checkpointing is a runtime tuning parameter.[2]

Based on lightweight transactions, PLinda supports the following program structuring paradigm. In the paradigm, each process is executed as a sequence of transactions. The transaction and checkpointing mechanisms guarantee that transactions execute atomically even in the presence of failures. Therefore a process logically fails only between transactions. Thus, the programmer only has to consider failures between transactions when making an application failure-resilient.

At each commit, each process saves critical information (usually the contents of a few variables) about its local state to tuple space. By critical information, we mean a set of process variables from which we can reconstruct its local state. We call critical information *encoded continuation* (continuation, for short)[58] and the saving operation *continuation committing*. On failure recovery, the failed process restores a continuation from tuple space. That is, transactions and continuation committing enable processes to make progress (i.e., restart from the last commit point) in spite of failure of a client process assuming that the tuple space doesn't fail. If the tuple space fails, then continuation committing and checkpointing ensures that the execution resumes in a transaction consistent state (i.e. a state of committed transactions that might be reached in a failure-free execution).

An important advantage of saving encoded continuations instead of process images is that encoded continuations support the use of heterogeneous machines because continuations which consist of the contents of variables can be made architecture-independent. By contrast, saving a process image is architecture-dependent.

Besides process failure-resiliency, the PLinda system is designed to utilize idle workstations for parallel computation. It monitors the idleness status of workstations and creates processes on idle ones. When a machine where processes are running becomes busy, the runtime system considers the machine to have failed and migrates the processes to an idle machine either immediately or some time later. PLinda migrates a process by killing it on one machine and recovering it on another using the fault tolerance mechanism.

The contributions of this dissertation are:

- To show how the traditional transaction model can be optimized and extended to support robust parallel computation efficiently.

- To propose a fault-tolerant programming model by which the programmer can

[1] In the current PLinda implementation, the performance of transaction commits is usually comparable to that of the other tuple space access operations such as `out` and `in`.

[2] Database aficionados will note that a volatile tuple space may allow committed data to be lost. That would be bad for a database, but is acceptable in our case, since long-running parallel applications require only that the final result be correct, since the programmer never acts on intermediate data. We ensure the correctness of the final result using checkpointing of the tuple space and critical program variables as we explain below.

easily make long-running coarse grain parallel applications fault tolerant without incurring high runtime fault tolerance overhead.

- To develop a process resiliency method which can support heterogeneous computing environments.
- To show that process resiliency mechanisms can also be used for process migration for dynamic load balancing.
- To demonstrate the implementation of a parallel programming system that supports both fault tolerance and utilization of idle workstations using a single efficient mechanism.

1.6 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the design of PLinda in detail. We explain the fault tolerance mechanisms such as transactions, continuation committing, and checkpoint-protected tuple space, and show how to construct fault-tolerant applications using these mechanisms.

In Chapter 3, we explain the PLinda tunable execution mechanism. This tuning feature is aimed at applications where processes have large continuations.

In Chapter 4, we describe how PLinda uses idle workstations for parallel computation. The issues involved in utilization of idle workstations are idleness detection, process scheduling and process migration. We discuss them and explain how the PLinda process resiliency mechanisms can be used for process migration.

In Chapter 5, we explain the implementation of PLinda and present experimental results. These results show that the current PLinda prototype can execute coarse grain parallel applications efficiently while supporting fault tolerance and utilizing idle workstations.

In Chapter 6, we present concluding remarks and future research directions for PLinda.

Chapter 2

Persistent Linda

2.1 Introduction

PLinda is a set of extensions to Linda to support robust parallel computation as well as computing using idle machines (where busy-ness = failure). The three major extensions are:

- *Lightweight transactions.* Used to maintain a consistent global state efficiently regardless of failure.
- *Continuation committing.* Used to make processes resilient to failure without relying on disk.
- *Checkpoint-protected tuple space.* A fault-tolerant tuple space based on checkpointing.

This design requires explicit operations for fault tolerance. We think that we will be able to address the programming issue via development of high-level programming toolkits or a Linda-PLinda translator later, but this will be our project's future work.

The PLinda system model is a collection of loosely coupled processors (no physically shared memory) communicating over a network (for example, networks of workstations). PLinda fundamentally assumes the "fail-stop" processor model (i.e., processors fail by stopping), but can also handle "slowdown" and network failures which do not affect tuple space. In PLinda (in fact, Linda), processes can not communicate directly with each other, but only via tuple space. Therefore, timeout can be used to turn slow processors to failed ones safely by preventing them from accessing tuple space afterwards (i.e., stopping them from communicating with the other processes). Likewise, network failures can be handled.

The rest of this chapter is organized as follows. We begin by discussing the issues concerning the application of transactions to robust parallel computation. Then, we describe the design of transactions in PLinda. Section 2.4 explains how to make tuple space failure-resilient in PLinda. Section 2.5 discusses how to extend the transaction mechanism to make processes resilient to failure. In Sections 2.6 and 2.7, we explain

multiple tuple spaces and process management in PLinda. Section 2.8 compares PLinda with other fault-tolerant Linda-variant systems and programming languages/systems that support transactions. Section 2.9 concludes this chapter.

2.2 Transactions and Robust Parallel Computation in the Linda Model

There are several fault tolerance abstractions for building failure-resilient applications.Among the popular ones are transactions[6, 34] and ordered atomic broadcast[7, 8, 40, 61]. Transactions have been mostly used for database applications such as bank or airline reservation applications where reliable management of persistent data is crucial. PLinda is a research effort to apply the transaction processing technology to a different class of applications, robust parallel computation. In the design of PLinda we chose transactions as the primary fault tolerance mechanism for the following reason. Transactions are a simple but effective abstraction for controlling concurrent access to shared data and maintaining a consistent state of shared data in the presence of failure[34]. Transactions are especially effective in the Linda model where the tuple space (shared memory) is the only mechanism for communication between processes and storage of shared data. For the same reason, most other fault tolerant work on Linda also supports some similar mechanism, though with less functionality[3, 4].

In this section, we discuss the issues that are raised when transactions are used for robust parallel computation. First, we explain transactions in the context of databases[1]. Transactions are an abstraction with the following **ACID** properties:

- *Atomicity.* Regardless of failure, a transaction executes a series of database accessing operations in an "all or nothing" manner. That is, the database reflects all of the updates (in the *commit* case) or none of them (in the *abort* case) — there is no other possibility. This property implies that partial results from aborted transactions will not affect other transactions.

- *Consistency.* A transaction produces a consistent result, provided that the transaction program is correct and the initial state of the database is consistent.

- *Isolation* (or *serializability*). The execution of transactions appears as if they were executed in a serial order. Transactions require concurrency control for this property. The most common implementation technique for concurrency control is two phase locking[6, 34]. For a transaction, two phase locking holds locks on accessed data items until the transaction commits, and therefore prevents transactions from accessing uncommitted updates.

- *Durability.* Committed updates in databases survive failure. Disk is generally used for stable storage and therefore, the updates made by a transaction are saved to disk

[1] Transactions are a general abstraction which is not restricted to database applications. However, since those applications are typical, our explanation is based on that viewpoint.

at commit. Thus, this property requires disk writes to be implicit in transaction commits. Since disk access incurs high access latency, transaction commits are expensive as compared to in-memory operations.

We examine these properties from the point of view of parallel processing. For robust parallel computation, our goal is to minimize the amount of lost work on failure but with only low runtime overhead. Runtime overhead comes mostly from concurrency control and transaction commit overhead. A second concern is programming overhead when the programmer explicitly uses transactions to build a fault-tolerant application.

Sometimes, there is a tradeoff between the two. For example, for database applications, an entire thread execution often encloses a single transaction. On failure, all the intermediate results are aborted and the entire thread is restarted from scratch; that is, the whole computation is lost, but data consistency is maintained at little cost in programming complexity. Fine grain transactions reduce the amount of lost work on failure. However, if execution of an application consists of a number of transactions, then transactions can not assure a clean runtime behavior on failure because restarting the application thread can neither be restarted from scratch nor declared completed. Instead, the thread must leave a persistent indication of where it is in the set of transactions it has executed. In databases, such persistent indications allowing a multi-transaction thread to resume from failure is known as mini-batching[34]. Unfortunately, any such technique to make continuations persistent requires additional programming effort or new higher level abstractions. (PLinda currently offers only low level abstractions as we will explain later on.)

The serializability property reduces parallelism. For example, if a process is executed as a single transaction, then it can not communicate to other processes during execution because its computation results become accessible to other processes only after it terminates.

Most parallel applications do not need the serializability property. We use concurrency control (a weakened form of two phase locking) in order to facilitate a consistent global state after client failure. In our version of two phase locking, write locks are held until commit, but read locks are held only while the read occurs (known as degree two serializability[6, 34] in the database world). The write locks prevent transactions from accessing updates made by a transaction until the transaction commits. So, a transaction abort affects no others — it's as if the transaction had never started. Since read locks do not influence states of other processes, they do not need to be held until the commit point. Database systems use that only to ensure serializability which we don't need.

The durability property requires that updates be written to disk at commit. Therefore, fine grain transactions incur relatively higher runtime overhead. In order to support robust parallel computation, tuple space must be fault-tolerant. However, robust parallel computation does not necessarily require intermediate results to survive failure, but intends to minimize the amount of lost work on failure. That is, it is not necessary that updates be saved to disk every time a transaction commits.

In summary, when the programmer uses transactions for parallel applications, he or she can have control over simplicity, the amount of lost work on failure, concurrency,

and commit overhead. In PLinda, we hope to give the programmer the ability to tune this tradeoff in many different ways.

2.3 Transactions in PLinda

In this section, we first explain the design of PLinda transactions, then discuss how they can be used, and discuss what extensions are needed to make processes failure-resilient. Throughout this section, we assume that tuple space is fault-tolerant, because PLinda treats transactions and fault tolerance in tuple space in an orthogonal way. The next section will discuss fault tolerance in tuple space.

We have designed the transaction mechanism for PLinda in the following way:

- A process is executed as a series of transactions. When a process experiences failure, the transaction mechanism aborts the currently active transaction of the process; that is, the failed process does not lose all the computation work. The resulting global state is as if the process failed immediately after the end of the last committed transaction.

- A process makes all the intermediate results accessible (i.e., releases all the write locks) to the other processes at each commit. Thus, processes can communicate with each other before they terminate.

- Transaction commits result in writes to tuple space, but not to disk. That is, other processes use the results under the assumption that tuple space and in turn the results are reliable. Thus, transaction commits require only in-memory operations. (If tuple space fails, then the commmits of many transactions will have to be undone; this is fine because the computations we are interested in have no user interaction until they complete.)

- The degree two variant of strict two-phase locking is used to prevent processes from accessing intermediate results from uncommitted transactions during runtime. This is PLinda's default. When the user wants serializability, he or she can tune the system to use the full strict two phase locking.

  The locking protocols allow the runtime system to abort any transaction of a process without affecting the execution of other processes — *process-private recovery of a consistent state*. This scheme is well suited to process migration, because a process can suspend execution (i.e., fail) on one machine and resume on another independently.

Let's look at an example in Figure 2.1. In the example, process A is executed as a sequence of three transactions and process B as a sequence of four transactions. Process A committed the first transaction whose result has already been used by the second transaction of process B. Then, process A failed during the second transaction. Even though process A produced intermediate results in tuple space during the second transaction, the strict two phase locking or degree two serializability protocol guarantees

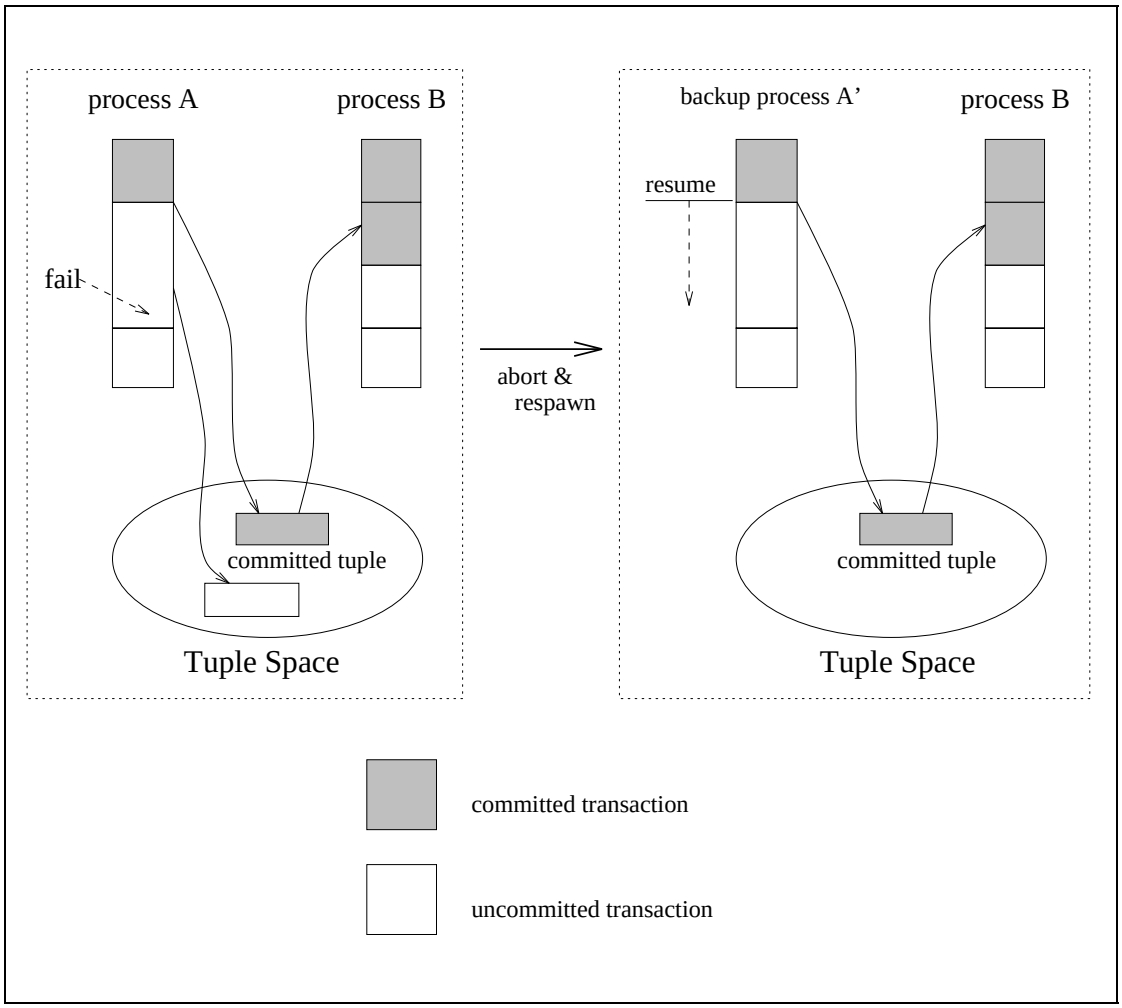

FIGURE 2.1: Execution of A Process as Multiple Transactions

that the results have not been accessed by process B. Therefore, aborting the second transaction of process A does not affect process B.

PLinda provides two language constructs to make a piece of code execute as a transaction: `xstart` and `xcommit`. `Xstart` begins a new transaction, and `xcommit` commits it as shown below:

```
xstart();
  arbitraryBlockOfCode
xcommit(tupleSpecification);
```

Here, *arbitraryBlockOfCode* is a sequence of arbitrary operations except `xstart` or `xcommit` (currently, PLinda does not support nested transactions). The entire *arbitraryBlockOfCode* is guaranteed to execute atomically (i.e., in an "all-or-nothing" fashion). In addition, the `xcommit` operation can also create a tuple as specified in *tupleSpecification*. The tuple is used for continuation committing which will be explained in Section 2.5. If the expression *tupleSpecification* is omitted, then no tuple is created. In addition to tuple space operations, process creation operations are also treated as updates to tuple space and therefore transactions govern their effect as well. Since a process can start and commit a transaction explicitly, it may run multiple transactions.

Let's look at some examples to explain how transactions are used for robust parallel computation. The examples are based on the master/worker programming paradigm that is most often used in Linda. This paradigm consists of a master process and a pool of identical worker processes that the master spawns. (The workers are identical in that they execute the same code; they may do so at different speeds.) During execution, the workers repeatedly grab tasks from a bag of tasks and carry them out. The master collects results generated by the workers. Figure 2.2 illustrates the programming model.

Figure 2.3 gives skeleton code of a worker process using a transaction in PLinda. In the example code, the transaction guarantees that each task is executed atomically in spite of failures. For example, suppose that a worker process fails while performing a task in a transaction. Then, the transaction will be automatically aborted; all the updates made for the task including the removal of the task tuple will be undone. Thus, the transaction mechanism restores a state in which tuple space appears as if the task had never been started. Another worker that is still functioning will be able to take and re-execute the task later. In the master/worker programming model, worker processes normally keep no state. Therefore, the code given in Figure 2.3 is resilient to failure in the sense that the entire execution can continue and complete correctly in spite of worker process failure as long as one worker is alive.

Figure 2.4 shows skeleton code for a master process. The master process creates task tuples in first transaction and then collects results in the next transaction. If the master process fails during execution of the first transaction, there will be no task tuples in tuple space after abort. If the master fails during the second transaction, the failure will not affect the tasks tuples already created in the first transaction, but only the second transaction will be aborted; all the deleted (in fact, marked as deleted) result tuples

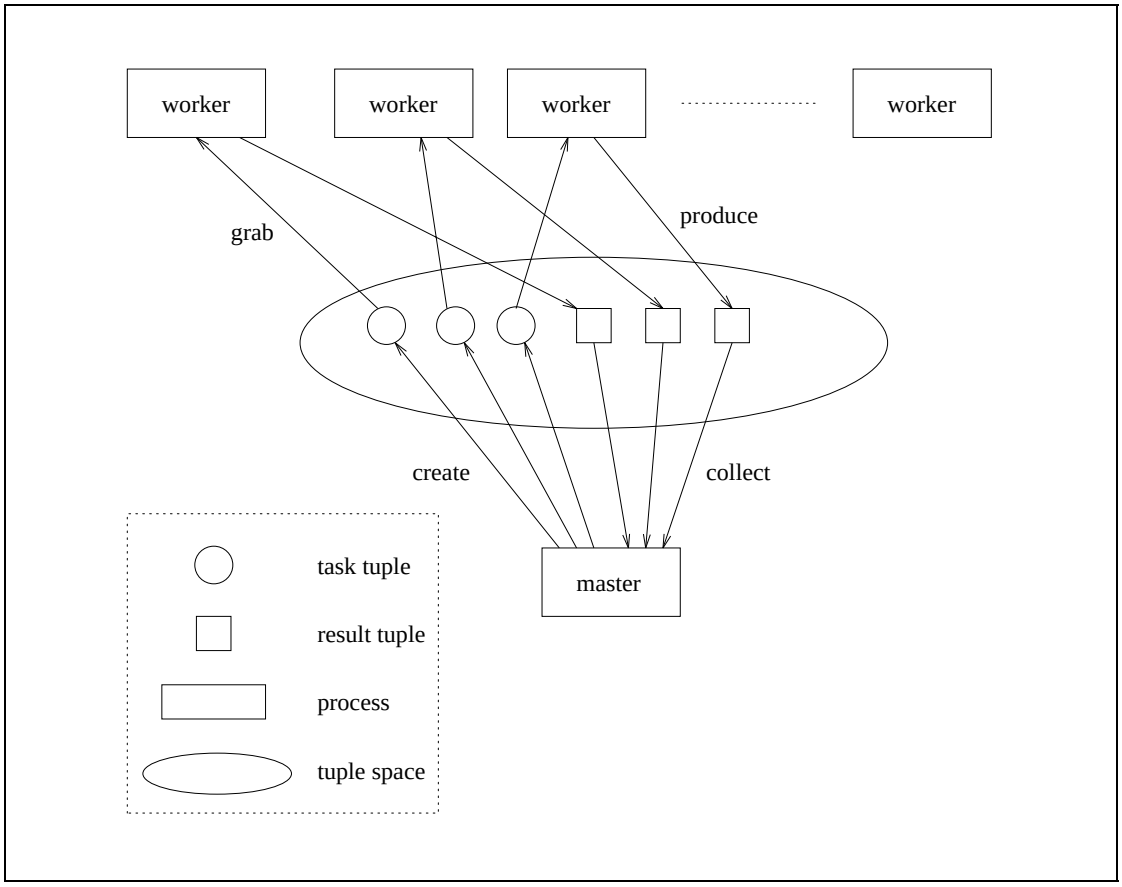

FIGURE 2.2: Master/Worker Programming Model

```
/* worker process */
int worker() {
  struct TaskType task;
  struct ResultType result;
  while(1) {
    xstart();

      /* grab a task */
      in("task", ?task);

      executeTask(&task, &result);

      /* produce a result */
      out("result", result);

    xcommit();
  } /* while */
}
```

FIGURE 2.3: A Fault Tolerant Worker Process

```
/* master process */
int master() {
int idx,numTasks;
struct TaskType tasks[MAX_NUM_TASKS];
struct ResultType results[MAX_NUM_TASKS];

  xstart();
    readSetting(&numTasks);
    readTasksFromFile(numTaks, tasks);

    for(idx=0; idx<numTasks; ++idx) {
      out("task", tasks[idx]);
    }
  xcommit();

  xstart();
    for(idx=0; idx<numTasks; ++idx) {
      in("result", ?results[idx]);
    }
    writeResultToFile(numTasks, results);
  xcommit;
}
```

FIGURE 2.4: A Master Process

become accessible again. That is, transactions ensure that the tuple space is restored to a state which does not contain any intermediate updates made by aborted transactions. However, transactions do not restore the local state of the failed process, for example, the values of local variables such as `numTasks` or information about where the master process has failed. In order to recover from failure, the master process needs to preserve such information regardless of failure.

PLinda processes use transactions in a few different ways:

- *Single-transaction processes*: Execute an entire process as a single transaction or simply repeat execution of the same code where each execution is a separate transaction but there is no inter-transaction dependency. The worker processes explained above are an example. Transactions make single-transaction processes resilient to failure.

- *Multi-transaction processes*: Run multiple transactions within a process. An example is the master process shown in Figure 2.4. The transaction mechanism guarantees that processes appear as if they only fail between two consecutive transactions. On failure of a multi-transaction process, the transaction mechanism restores a consistent state of tuple space, but does not recover the state of the process at failure point. Continuation committing, to be discussed, does this.

In summary, the transaction mechanism of PLinda is designed to reduce the amount of lost work on failure and to restore a consistent state of tuple space after failure, without significant degradation of parallelism or high runtime overhead.

However, the design also raises two issues: reliability in the tuple space and failure-recovery of a process.

- Unlike database transactions which support reliability for databases, the PLinda transaction mechanism simply assumes reliability of the tuple space and do not flush updates to disk at commit.

- If the entire execution of an application is treated as a single transaction, then the runtime system can automatically re-execute the application from scratch. A failed multi-transaction process can not be restarted from scratch, because that may cause committed transactions to be executed again. The process must resume from the point where the last transaction committed, and therefore will not repeat execution of the committed transactions.

In the following two sections, we will discuss these two issues.

2.4 Fault-tolerant Multiple Tuple Spaces

As mentioned in the previous section, PLinda treats transactions and reliability of tuple space in an orthogonal way. In this section, we discuss how to make tuple space failure-resilient.

Failure resiliency requires some form of redundancy, either replication on different processors connected by a reliable network or replication on different storage media (e.g., non-volatile RAM or disk). Due to orthogonality, PLinda can support both approaches as a tunable feature. Currently, PLinda supports only the latter approach because disks are better suited to our separate goal of persistence. However, we plan to add the former approach to the implementation in the future.

PLinda is designed to support two kinds of tuple space:

- *Checkpoint-protected tuple space.* Unlike most transaction processing systems[6, 34], PLinda replicates the "transaction-consistent" state of tuple space on disk only periodically called *tuple space checkpoint.* Transaction commit operations do not require updates to be written to disk before they are finished. The transaction-consistent state is one that reflects the updates made by all and only the committed transactions. Figure 2.5 shows a snapshot of execution using checkpoint-protected tuple space. If the tuple space fails and recovers, then it will be restored to the last checkpointed state on disk. However, when a process suffers failure, tuple space can recover a consistent state by aborting the transaction being executed by the process, without restoring the last checkpointed state.

  The advantages of this design are to allow tuple space to execute transaction commits efficiently and the user to decide the frequency of checkpointing at runtime. This is our default runtime option and works provided the user cares only about the final result of the computation, not about any intermediate result.

- *Stable tuple space*[2]. All the updates made by a transaction are replicated on disk, before the transaction commits as in conventional database systems. Therefore, the latest transaction-consistent state of tuple space is always maintained on disk. Figure 2.5 illustrates stable tuple space. On recovery after failure, tuple space will always be restored to the last transaction-consistent state. In this case, updates made by committed transactions will survive failures of tuple space — *durability.*

In the case of tuple space failure, checkpoint-protected tuple space may lose consistency with processes. More specifically, a recovered state of tuple space loses the updates made by the transactions committed between the last checkpoint and the failure point. In Figure 2.5, states of processes and the checkpointed state of tuple space on disk are inconsistent, because the checkpointed state does not reflect the updates made by the third transaction of process A and the second transaction of process B. The transaction mechanism cannot undo the effect of the committed transactions on running processes and states of the processes may already reflect the lost transactions.

In this case, the PLinda runtime system forces all the running processes to perform failure-recovery operations. In PLinda, each process saves local state to tuple space at each commit and recovers the state saved at the last commit from tuple space on recovery after failure. The process restores consistency with tuple space on failure recovery. Failure-recovery of a process will be explained in the next section.

[2]Stable tuple space has not fully been implemented in the current prototype system

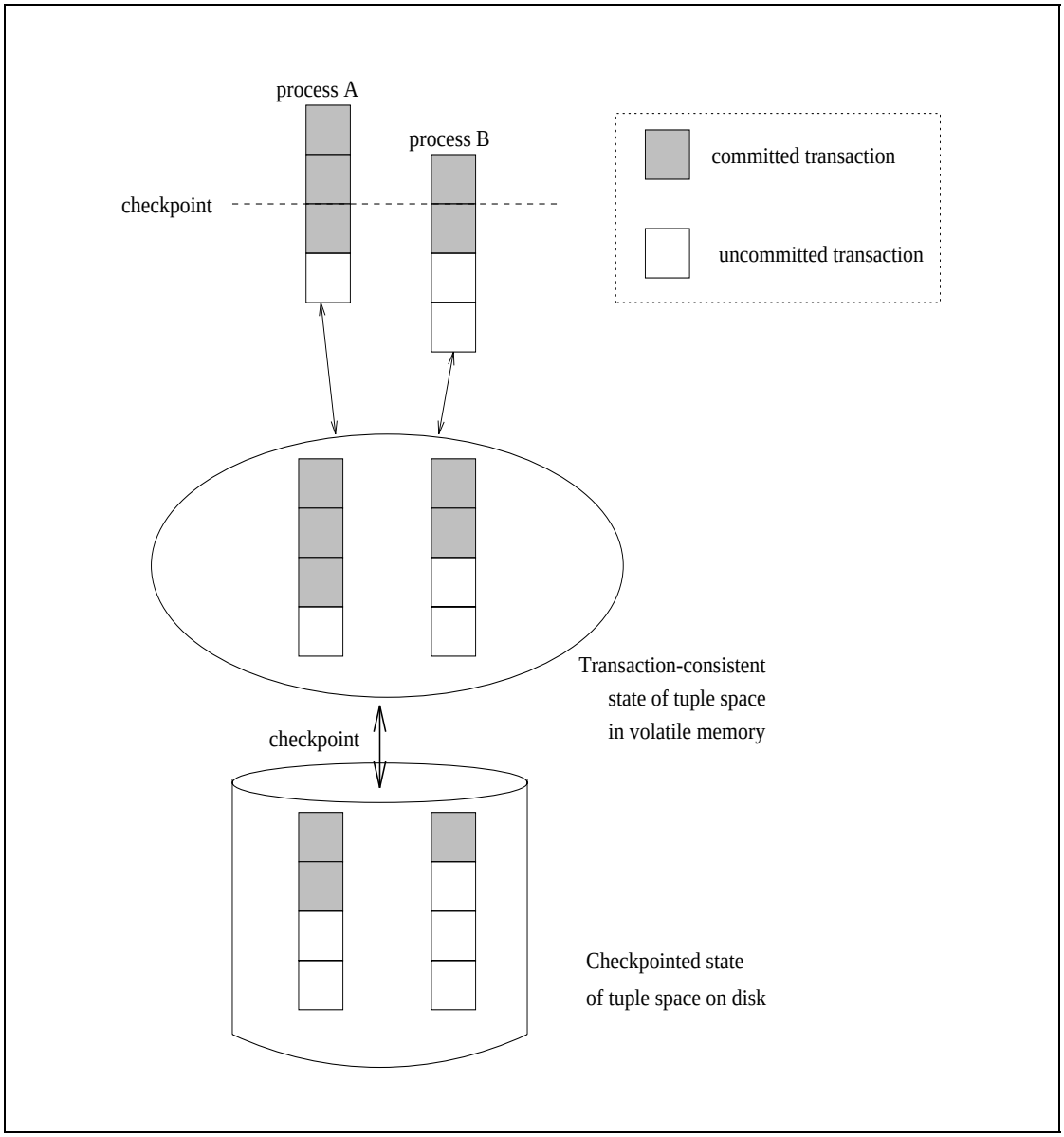

FIGURE 2.5: Checkpoint-protected Tuple Space

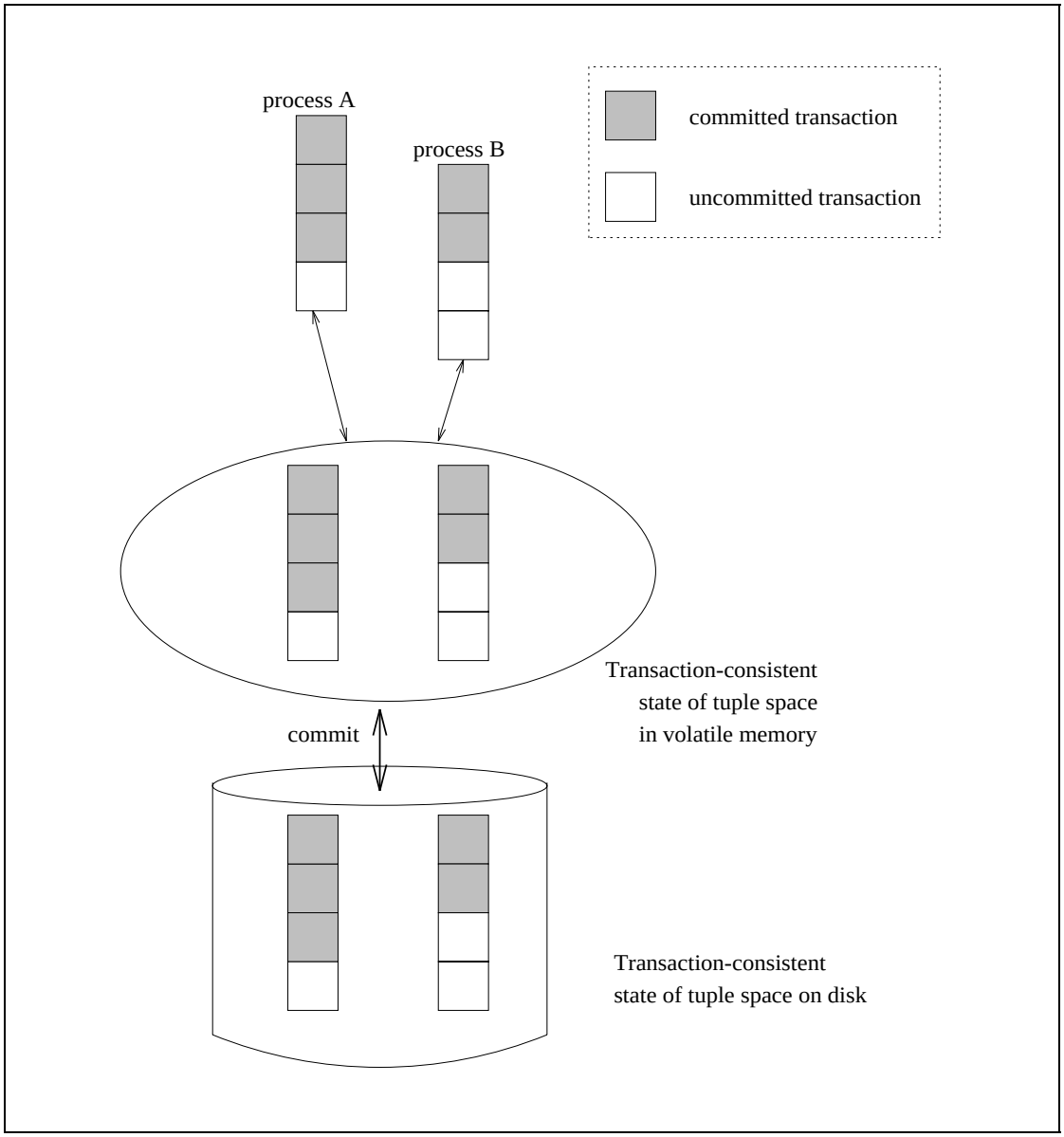

FIGURE 2.6: Stable Tuple Space

Stable tuple space does not lose updates made by committed transactions as shown in Figure 2.5, so is less expensive than checkpoint-protected tuple space on recovery because committed transactions are not required to be re-executed. In contrast, checkpoint-protected tuple space is efficient during normal execution but incurs overhead on recovery from failure. Since failure is in fact rare, checkpoint-protected tuple space is better suited to robust parallel computation in which only the final answer is important.

If by contrast, intermediate transaction commits might be important, as in a transaction processing monitor application[34], stable tuple space would be more appropriate.

2.5 Continuation Committing

In this section, we discuss how to make processes resilient to failure. There are two approaches to resilient processes: process replication with atomic broadcast and process checkpointing. PLinda takes the latter approach for resilient processes because the former approach requires redundant processors for each process as well as synchronization which can be too costly for parallel computation.

In order to be failure-resilient, a process needs to replicate sufficient information about its state to continue executing after failure. One simple approach is to save the entire process image (i.e., a control stack, state of address space and contents of machine registers). This approach makes failure-recovery simple (in fact, user-transparent recovery is possible) because all information is available at that point, but is generally expensive because the size of a process image is large. Another more complex but less expensive approach is to save only critical information about the state during normal execution and to re-construct state from that information on recovery from failure. This approach requires explicit operations for recovery.

PLinda has adopted the second approach (which we call *continuation committing*). This scheme allows the programmer to customize the replication operation in the way that each process saves only critical data required to resume execution on recovery after failure. Typically, this includes an indication of which transactions have completed, and any key data variables.

The continuation committing mechanism is based on the following two operations:

- `Xcommit`: Allows a process to save local state to tuple space at each commit. `Xcommit` can create a tuple in tuple space that reflects local state at that point. If there is already a tuple created by the previous `xcommit`, then it is overwritten. The tuple is only accessible when the `xcommit` operation is finished; that is, its transaction is committed. Only the process or its backup processes can access the tuple using the `xrecover` operation explained below.

- `Xrecover`: Retrieves the tuple created by the last committed transaction. It can be used in the same way as `rdp`. If there is no tuple created by the `xcommit` operation, then it simply returns false. A backup process which takes over the remaining task of a failed process uses the `xrecover` operation to restore the state of the failed process at the last commit point.

Using these operations, the programmer designs each process to save information about local state (in fact, only contents of local variables to be required for failure-recovery) at each commit and to restore the state saved at the last commit on recovery from failure.

The example in Figure 2.7 shows how to use these operations. It presents a fault-tolerant version of the code given in Figure 2.4. In the code, only the underlined operations are those which are added for continuation committing. How it works is almost self-explanatory, but we give a brief explanation. For this process, the values of only two variables `transId` and `numTasks` are sufficient to restore local state on recovery and therefore saved to tuple space at each commit. In order to record where failure happens, the code assigns a logical identifier of integer type to each transaction and always sets local variable `transId` to point to the current transaction. On recovery, this variable is used to find out where the process fails. Also, variable `numTasks` is needed because the first and the second transactions share the variable and failure may happen between them.

On the failure of a process, the runtime system automatically detects it and respawns the process again. We call the new process the "backup process." In PLinda, the primary process and backup processes run the same executable but are designed to behave differently. At the beginning of the execution, the code checks if the current process is a backup process, by calling the `xrecover` operation. If the process is a backup one, then the `xrecover` operation retrieves contents of the local variables saved by the failed process and returns true; otherwise, the operation simply returns false.

Writing a recoverable process entails inserting operations to skip committed transactions. This is usually simple as illustrated in Figure 2.7. The technique is to set a local variable to point to the next transaction upon commit. The recovery code simply locates that transaction.

The advantage of continuation committing is that the programmer can take advantage of characteristics of the application when designing processes to be resilient to failure. For example, the master process in Figure 2.4 can construct local state from only two variables `transId` and `numTasks` on recovery. In Figure 2.7, the code exploits the characteristic. Therefore, efficiency and flexibility are advantages of continuation committing, but additional programming overhead is a drawback.

The scheme is motivated by the fact that many coarse grain parallel applications which have simple control structures can be easily made fault-tolerant if transactions are available.

2.6 Tuple Groups: a namespace management issue

In addition to failure-resiliency, PLinda extends the Linda flat tuple space model to support multiple tuple spaces which are called *tuple group*s. In Linda, all tuples are always accessible to every process. So, tuple space access operations from a process may conflict with those from others by accident if they happen to use the same pattern of tuples for different purposes. Such cases are likely in practice, especially, in large scale

```
/* master process */
int master() {

int transId = 0;
int idx, numTasks;
struct TaskType tasks[MAX_NUM_TASKS];
struct ResultType results[MAX_NUM_TASKS];

// retrieve state tuple
xrecover(?transId, ?numTasks);

if(transId == 0) {
  xstart();
    readSetting(&numTasks);
    readTasksFromFile(numTasks, tasks);

    for(idx=0; idx<numTasks; ++idx) {
      out("task", tasks[idx]);
    }
  xcommit(++transId, numTasks);
}

if(transId == 1) {
  xstart();
    for(idx=0; idx<numTasks; ++idx) {
      in("result", ?results[idx]);
    }
    writeResultToFile(numTasks, results);
  xcommit(++transId, numTasks);
}

}
```

FIGURE 2.7: A Fault Tolerant Master Process

```
gid ts_handle;
int ivar;
float fvar;
ts_handle = create_group("my tuple space");
out[ts_handle]("sample tuple", 1, 2.5);
in[ts_handle]("sample tuple", ?ivar, ?fvar);
destroy_group(ts_handle);
```

FIGURE 2.8: Tuple Space Creation and Destruction in PLinda

applications which include a large number of components or are developed over a long period of time.

In PLinda, applications can have multiple tuple spaces. They explicitly create and destroy tuple spaces using the `create_group` and `destroy_group` operations which have the following form:

create_group(*tupleSpaceName*)
destroy_group(*tupleSpaceHandle*)

Here, *tupleSpaceName* is the name of a new tuple space to be created and is supposed to be either a string constant or a variable of string type. The `create_group` operation returns a handle to the new tuple space whose data type is called `gid`. The `destroy_group` takes either a `gid` constant or a variable of `gid` type.

Tuple spaces can be accessed by only those processes with handles to them; that is, the tuple space access operations such as `out`, `in`, `rd`, `inp` and `rdp` take a tuple space handle as follows:

operator[*tupleSpaceHandle*](*tupleSpecification*)

If the expression [*tupleSpaceHandle*] is omitted as in Figures 2.3 and 2.4, then the access operations assume a default tuple space. An example of multiple tuple spaces is shown in Figure 2.8.

In PLinda, tuple group handles are first-class objects which processes can pass through tuple space. Thus, a group of processes may communicate privately by making one of the group members create a tuple group and pass the handle only to the group members.

2.7 Process Management

Chapter 1 explained process management in Linda which is not resilient to failure. PLinda redesigns it to provide fault tolerance and to facilitate portability. Portability is especially important on networks of workstations because they generally consist of heterogeneous machines and operating systems. Regarding process management, PLinda differs from Linda in the following ways:

- Different unit of parallelism: an executable file at the OS-level.
- Transactional process creation.
- Automatic failure detection and backup process restart.

First, PLinda uses an OS-level executable file (like `a.out` generated by the `cc` compiler) as the unit of parallelism; that is, a process is invoked to execute an executable file. This design allows the runtime system to exploit the process management facility available in the underlying OS such as UNIX, instead of providing customized mechanisms. For example, the current implementation invokes a PLinda process using the `execl` library function in the UNIX or UNIX-variant OS's. Therefore, the design facilitates portability and heterogeneous processing.

Because of the different unit of parallelism PLinda provides two new operations for process invocation: `proc_eval` and `arg_rdp`. They have the following forms:

proc_eval(*tupleSpecification*)
arg_rdp(*accessPatternSpecification*)

The design is intended to be reminiscent of the Linda `eval` operation. Like `eval` in Linda, the `proc_eval` operation takes a series of expressions to be converted to a tuple (which is called *argument tuple*). However, the first field of an argument tuple must be the name of an executable file. The `proc_eval` operation constructs the argument tuple first, and then creates a process to run the executable file specified in the field.

The new process can retrieve the argument tuple using the `arg_rdp` operation. The `arg_rdp` operation is used exactly in the same way as `rdp`. However, the argument tuple is private to the process or its backup processes in case of failure; in other words, the other processes can not access it. Thus, the spawning and spawned processes can communicate via the argument safely.

Figures 2.9 and 2.10 give an example showing how to create processes in PLinda. The example is a "single-producer/multiple-consumer" program consisting of two files: `producer.c` and `consumer.c`. As in Linda, PLinda assumes a predefined name for the main function which is `real_main`. A producer runs the code in `producer.c` and consumers executes the code in `consumer.c`. In `producer.c`, the name of the executable for consumers is assumed to be `consumer`.

As explained in Section 2.3, the transaction mechanism also governs the effect of process creation. For example, consider the two `proc_eval` operations in Figure 2.9.

```
int real_main(int argc, char** argv, char** env) {

char data[DATA_LEN];
gid ts_handle;

xstart();
  ts_handle = create_group("channel");
  proc_eval("consumer", ts_handle);
  proc_eval("consumer", ts_handle);
xcommit(); // after commit, the two consumers can start

while(1) {
  xstart();
    if(readData(data)<0) break;
    out[ts_handle]("data",data);
  xcommit();
}

}
```

FIGURE 2.9: Producer in PLinda

The processes which they spawn start to run only after the transaction commits (i.e., the `xcommit` operation completes). If the transaction aborts due to failure, then the newly created processes are aborted automatically.

Finally, the runtime system detects failure of a process, once it starts, and automatically spawns another backup process on failure. The backup process takes over the task of the failed process. Unlike transparent approaches where the backup process continues execution transparently, PLinda requires backup processes to take over tasks of their failed processes explicitly.

2.8 Related work

2.8.1 Fault Tolerance Work on Linda

In [3, 4], Bakken and Schlichting present FT-Linda, a variant of Linda that addresses fault tolerance. For tuple space reliability, FT-Linda assumes a set of replicated tuple spaces connected together by an ordered atomic broadcast network. Thus, FT-Linda is better than PLinda for fault tolerant applications where availability is important. However, the PLinda checkpoint-protected tuple space can be more efficient during normal execution (unless there is special hardware support for communication) because it does not require

```
int real_main(int argc, char** argv, char** env) {

char name[LEN];
gid ts_handle;

// retrieve the argument tuple
arg_rdp(?name, ?ts_handle);

char data[DATA_LEN];
while(1) {
  xstart();
    in[ts_handle]("data",?data);
    processData(data);
  xcommit();
}

}
```

FIGURE 2.10: Consumer in PLinda

runtime overhead due to ordered atomic broadcast.

Like PLinda, FT-Linda does not provide transparent process resiliency but allows the programmer to make processes resilient to failure using a restricted form of transaction mechanism called atomic guarded statements. Atomic guarded statements can execute multiple tuple space operations atomically, but do not allow computation between the operations.

With respect to programming effort, FT-Linda requires programmers to clean up intermediate results of failed processes in tuple space and to respawn backup processes explicitly. In contrast, PLinda cleans up intermediate results of failed processes and respawns backup processes automatically.

FT-Linda saves local state of a process using Linda tuple space operations, but PLinda supports separate continuation committing operations for that purpose. In PLinda, the runtime system optimizes such checkpointing operation.

Power failures are the most frequent kind of hardware failures. They usually cause all processors to fail. FT-Linda stores everything in volatile storage and loses it in the event of total failure. However, processes and tuple space can still survive total failure in PLinda because they rely on persistent storage such as disk. In institutes where all the machines are rebooted periodically, fault-tolerant long-running parallel applications also need to rely on disks.

The Piranha system is a Linda variant designed to utilize idle workstations effectively for parallel computation [44]. In spite of its different objective and no direct concern for

fault tolerance, Piranha deals with fault tolerance-related issues, and shows another use of fault tolerance.

In the Piranha system, worker processes, called Piranha, execute tasks on idle workstations. When an idle workstation becomes busy again, the Piranha processes on the workstation "retreat." The retreat operation requires the Piranha processes to abort their current tasks and terminate immediately; thus, retreat has the same effect as failure. Then, the aborted tasks are re-executed by Piranhas on other idle workstations. The Piranha model assumes that Piranha processes execute each task atomically in spite of retreat.

From the point of view of fault tolerance, the current Piranha system has one drawback: it requires programmers to clean up intermediate results at retreat. That makes programming difficult.

Piranha could be enhanced by mechanisms such as PLinda transactions to make the clean-up of intermediate results automatic. Moreover, such mechanisms would make Piranha processes resilient to processor failure. In addition, designing stateful Piranha processes would also be simpler with mechanisms like continuation committing in PLinda.

Other fault tolerance work has also produced useful ideas. Xu and Liskov proposed a protocol to replicate tuples and to maintain consistency of replicas, despite processor failures[66]. [43] discussed the performance and availability issues concerning replication techniques for tuple space. [17] also presented a protocol to relax the consistency of tuple space replicas to improve performance. [42] proposed a scheme based on checkpointing the processes and logging all the tuple space accesses.

2.8.2 Programming Languages and Systems Supporting Transactions

There have been research efforts to develop programming languages and systems to use transactions as the foundation for constructing distributed applications. They are Argus[47], Avalon[26], Camelot[26], Clouds[21] and TABS[27].

Argus is a programming language and system to support the implementation and execution of distributed applications such as mail systems and inventory control systems[47]. The principal mechanism of Argus is guardians which are a special kind of abstract objects. Guardians encapsulate information within local state and permit it to be accessed by means of special procedures, called handlers, that can be called from other guardians. Reliability and concurrency control are supported by saving local state to disk and running every handler call as a transaction or a nested transaction.

Avalon[26] is a set of linguistic constructs which can be implemented as extensions to familiar programming languages such as C++, Common Lisp and Ada. The computation model of Avalon resembles that of Argus. However, Avalon gives the programmer explicit control over commit and abort operations but Argus does not.

One may use one of these systems for coarse grain parallel applications because they support parallelism. However, there are several drawbacks against such use. First, they are explicitly designed for the client-server programming style which is not common for parallel applications. Parallel application programmers must be forced to use the style. Second, nested transactions do not require disk writes at commit but increase book-

keeping overhead in the runtime system. Top-level transactions are expensive because updates need to be saved to disk at commit. Finally, even though nested transactions allow more concurrency, they are still based on a hierarchical calling structure and prevent interaction between sibling transactions. In Avalon, locking protocols can be customized, but such custom locking protocols complicate failure recovery.

In contrast, PLinda allows the computation of any process to be simply divided into a sequence of separate transactions. Thus, processes can be easily designed to communicate with each other. Implementation of the runtime system is also relatively simpler because of the flat transaction model. Checkpoint-protected tuple space makes transaction commits efficient.

Camelot[26], Clouds[21] and TABS[27] provide distributed transaction facilities as a set of user libraries or operating system features. Like Avalon and Argus, they are aimed at distributed applications and have similar drawbacks as Avalon and Argus, when they are used for parallel computation.

2.9 Summary

In this chapter, we described the design of PLinda. Throughout the design of PLinda, our research effort has been focused on: (1) studying issues relating transactions to parallel computation and (2) optimizing and extending transaction mechanisms to address the issues.

Transactions are designed to support the ACID properties, but it is not necessary to use all the properties. For example, isolation and durability are not needed for robust parallel computation. Also, the length of transactions has significant impact on simplicity, the amount of lost work on failure, concurrency and transaction commit overhead. There is a tradeoff between short and long transactions. Short transactions lose less work on failure and give greater concurrency, but increase the programming complexity and commit overhead.

In order to reduce commit overhead, PLinda treats transactions and reliability of tuple space orthogonally. Transactions do not write to disk on commit, but simply to the tuple space. This allows efficient transaction commits and minimizes the losses in case a client processor fails or becomes busy. PLinda makes the tuple space resilient to failure by using checkpointing. This is called checkpoint-protected tuple space. This works, but entails relatively high recovery overhead if the tuple space server crashes.

In PLinda, the programmer designs processes to be failure-resilient explicitly, using the transaction and continuation committing mechanisms. Each process saves enough information about its state to tuple space at each commit to form its continuation. This scheme has advantages of efficiency and flexibility as explained in Section 2.5, but requires explicit operations for fault tolerance. In practice, we usually use a few transactions for coarse grain parallel applications and therefore the amount of additional programming overhead is often negligible.

In addition to fault tolerance, PLinda supports multiple tuple spaces to facilitate tuple management in large-scale parallel applications. Also, PLinda designs the process

management to enhance portability. Portability is important for networks of workstations.

Chapter 3

Tunable Execution

3.1 Introduction

In the last chapter, we described how a fault-tolerant application is designed in PLinda. In this chapter, we discuss how the execution of such an application can be tuned at runtime. A *resilient* process is one that can make progress and terminate correctly regardless of failure during execution. To make a process resilient, it is essential to replicate state to other processors or disks. Replication constitutes the major overhead of resilience. The amount of overhead depends on the size of data for replication and the frequency of replication operations. Also, there is a tradeoff between the frequency of replication and the amount of work lost on failure.

Recall that PLinda allows the programmer to control the size of data for replication (i.e., contents of continuation and the shared data) and the frequency of replication operations (i.e., transaction commit points) when designing fault-tolerant processes. The only constraint on the programmer is that commits are required for communication: a process must commit before any of the tuples it has produced can be read by other processes.

The current design of PLinda allows the user to tune execution of the continuation committing operations in three ways:

- *Commit-consistent execution.* A process saves continuation to tuple space at every commit as specified by the programmer. On failure, the process recovers its state from the continuation of the last commit, independently of other processes.

  This method works well when continuations are small.

- *Message logging/replay.* A process saves continuation to tuple space only when requested by the runtime system (the frequency is a tuning parameter). However, its tuple space access operations are recorded during normal execution and replayed to reconstruct the state at the last commit on failure recovery. On failure, the process may need to re-execute committed transactions but still can recover independently of others.

This method works well when continuations are large, but tuple space modifications are small.

- *Coordinated checkpointing.* Processes do not save continuations to tuple space independently. Periodically, the runtime system forces all the processes to save continuations to tuple space and then checkpoints tuple space to disk. On any failure, tuple space is first restored to the last checkpointed state on disk and processes recover their states from the restored tuple space. That is, a single failure leads to massive rollback.

  This method is fastest in the failure-free case.

In this chapter, we consider only client side failures for the two reasons. First, processes and tuple space are treated orthogonally in PLinda with respect to fault tolerance. Second, all the three methods handle server failures in the exactly same way: tuple space is first restored to the checkpoint on disk and then all the processes recover their state from the restored tuple space.

The rest of this chapter is organized as follows. We start with the motivation for this tunable approach. Then, Sections 3.3, 3.4 and 3.5 describe the commit-consistent execution, message logging/replay and coordinated checkpointing methods, respectively. In Section 3.6, we present correctness proofs for these execution methods. In Section 3.7, we compare the tunable execution mechanism of PLinda with transparent checkpointing and rollback-recovery techniques. Finally, we summarize this chapter in Section 3.8.

3.2 Motivation

In this section, we discuss example applications to motivate our tunable approach to the execution of fault-tolerant processes. We first look at applications which can be executed efficiently without tuning. Then, we consider those which would require runtime tuning so as not to suffer severe performance overhead due to continuation committing.

For the first class of applications, we discuss those based on the "master/worker" programming model. In this model, task descriptions, data and intermediate results are placed in shared memory or a globally accessible location. Processes (called workers) read task descriptions and data from shared memory, carry them out, and put results back into shared memory. Since a task can be executed by any process (processes are decoupled from tasks and data), a slow process does not become a bottleneck. In other words, processes on fast machines do more work and those on slow machines do less thus achieving load balancing. This model is widely used for applications targeted at networks of workstations where load balancing is crucial for performance.

The PLinda programming PLinda model is well-suited to applications based on the master/worker model for the following reason. Since processes (i.e., workers) do not maintain intermediate results in local state, they do not have much critical information in local state; that is, they have small continuations. Therefore, continuation committing operations are cheap so the programmer can design processes to execute continuation committing frequently to reduce the amount of work lost on failure.

However, the master/worker programming model is not suitable for applications where tasks need a lot of input data and produce a lot of result data. A typical example is the class of scientific applications which solve partial differential equations. They usually update large matrices continuously during computation. For these applications, a programming technique called *data partitioning* (also called *domain decomposition*) is often used. This programming technique reduces data movement (i.e., communication) among processes by designing an application to distribute data into the local state of processes at the beginning and to perform computation and to maintain intermediate results as locally as possible.

Since the intermediate results in local state must survive failure, they must be included in continuations which are thus required to be large. Therefore, continuation committing is very expensive for these processes. Since continuations are required only on failure and failure is in fact rare, it makes sense to perform continuation committing less frequently for better runtime performance although this increases failure recovery overhead. In PLinda, the runtime tuning capability allows such tradeoffs. Also, such approach is effective in situations where idle machines are utilized for parallel computation and non-idleness (which is treated as failure) is rare, for example, during off-duty hours.

In summary, PLinda allows the programmer to reduce the size of data for continuation committing by including only critical information in continuations when designing a fault-tolerant application. If processes have small continuations, this scheme allows efficient execution of continuation committing. However, continuation committing becomes expensive for applications where processes have large continuations. For these applications, PLinda provides the tunable execution mechanism which allows the end-user to trade frequency of continuation committing against failure recovery overhead.

3.3 Commit-consistent Execution

In the commit-consistent execution method, each process's continuation is saved to tuple space at each commit as specified in the code. That is, the method is consistent with the programmer's viewpoint.

This execution method is most effective for applications where processes have small continuations. Applications based on the master/worker programming model are typical examples.

Figure 3.1 illustrates this execution method. The figure shows how a multi-transaction process recovers from failure. The process is executed as a series of transactions; T_i and C_i denote the i-th transaction and continuation at the commit of the i-th transaction, respectively. C_i is saved to tuple space at the commit of T_i. In the figure, the process experiences failure during transaction T_4 — *physical failure*. Then, the transaction mechanism aborts all the intermediate work of T_4. After the abort, the state of execution appears as if the process failed after T_3 and before T_4 — *logical failure*. On recovery from failure, the process recovers continuation C_3 from tuple space and restarts from the beginning of transaction T_4.

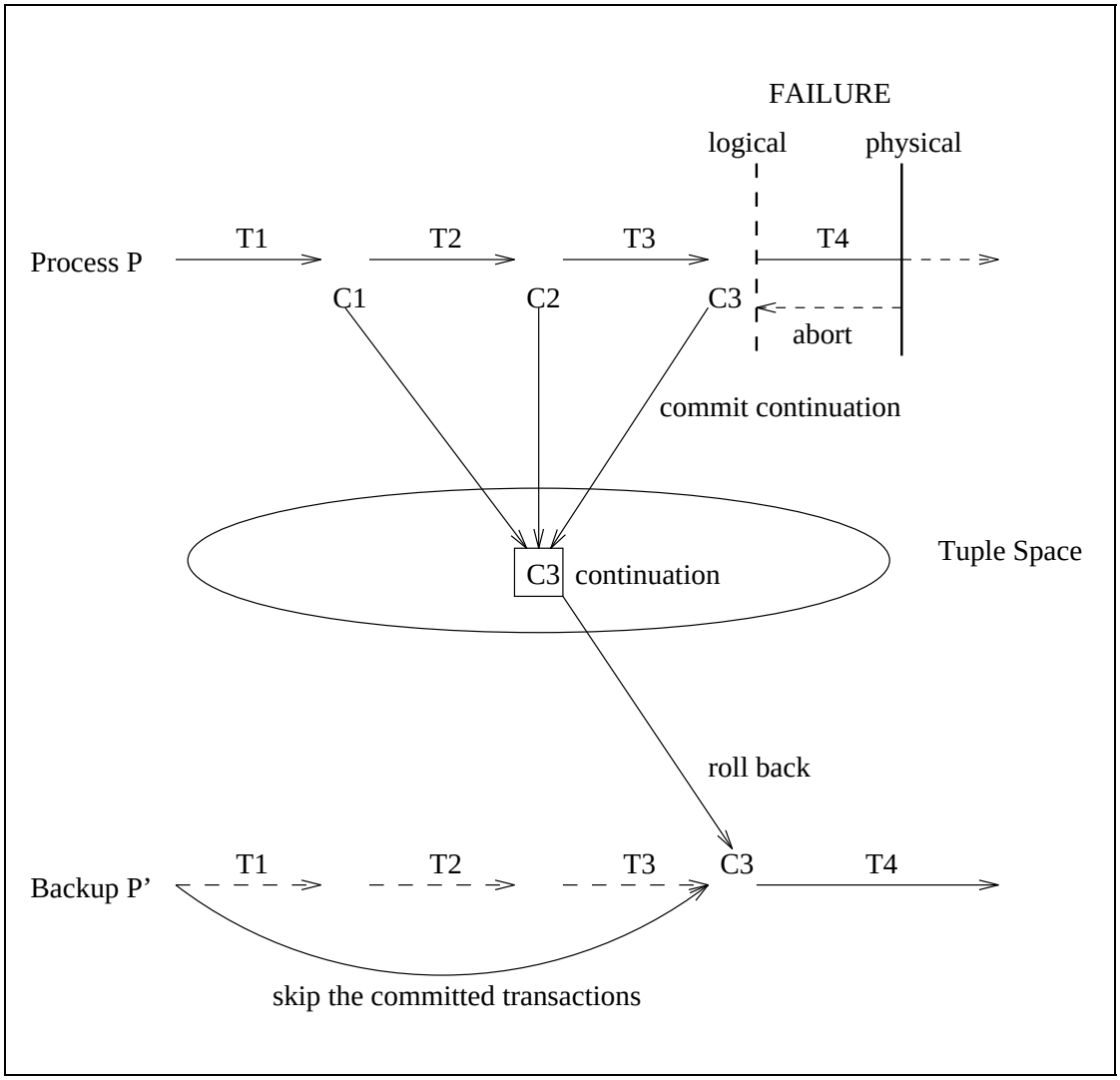

FIGURE 3.1: Commit-consistent Execution

3.4 Message Logging/Replay

In PLinda, fault-tolerant processes are designed to perform continuation committing at each commit. However, the message logging/replay method allows these processes to avoid executing a continuation committing operation at each commit. This implies that processes may have to re-execute transactions committed since the latest continuation committing operation on failure recovery. In other words, failure recovery overhead is traded for better runtime performance.

In this method, processes can still recover from failure independently of other processes as in commit-consistent execution, but message logging (i.e., recording execution of tuple retrieval operations such as `in` and `rd`) is required during normal execution. Therefore, this method is well-suited to applications where processes have large continuation, but access tuple space rarely.

In this method, a process's continuation is saved to tuple space only periodically but the continuation at the last commit is reconstructed, instead of being retrieved from tuple space, on failure recovery. Reconstruction of continuation is based on message logging and replay techniques which are similar to work by Strom and Yemini[59]. Specifically, the method is as follows:

- *Periodic continuation committing and continuous message logging.*

    1. A process performs a continuation committing operation on tuple space only when the runtime system explicitly requests it. (How frequently that occurs is a tuning parameter.) That is, although the semantics of the operations is that they occur at each commit point, the system will not implement them that way.
    2. For a running process, the tuple space server (the process which manages tuple space at runtime) maintains the history of the committed input tuple space operations (i.e., the `in` and `rd` operations) executed by the process since the last continuation commit. Thus, for each input tuple retrieval operation (`in`, `rd`), the history contains information about which tuple is accessed and its value. As we said above, these input operations carry much less data than the continuation committing operations in the applications we know about.
    3. Once a process saves its private state to tuple space (i.e., performs the continuation committing operation), the history discards all the process's previous input tuple space operations. It begins to collect these again starting with the transaction following the continuation committing operation.

- *Rollback and Replay*

    1. If a process fails and recovers, then it first restores the latest saved private state and resumes execution from that state. However, the continuation at the last commit is reconstructed replaying the history as explained below.
    2. In the history, `out` and `eval` operations are ignored, because the tuple space has already reflected their execution.

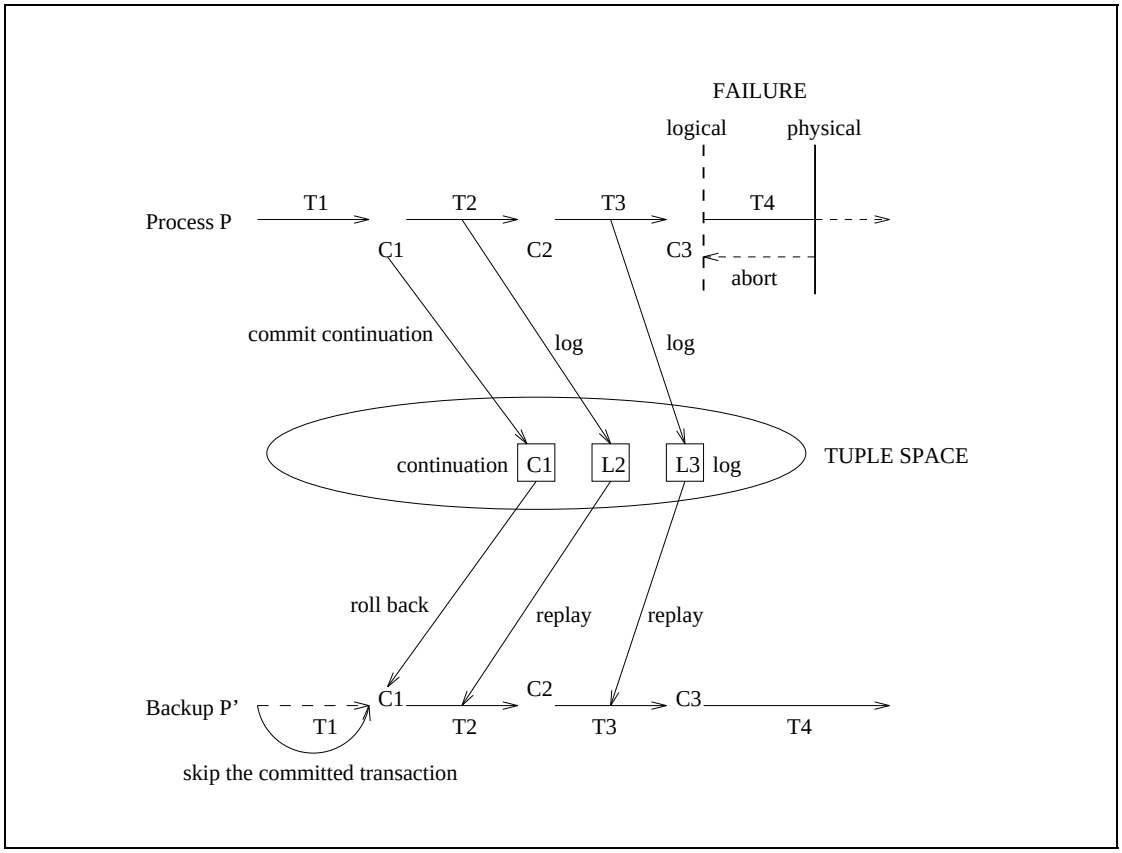

FIGURE 3.2: Message Logging and Replay

3. For `rd` and `in` operations, the history is replayed.

Figure 3.2 illustrates the message logging and replay method. The figure shows how the message logging/reply method executes and recovers the same multi-transaction process shown in Figure 3.1. In the figure, the process saves continuation C_1 to tuple space when it commits transaction T_1. After that, the process does not perform the continuation committing operations for transactions T_2 and T_3. Instead, the runtime system records the tuple space operations for T_2 and T_3. The process experiences failure during transaction T_4. On recovery from failure, the process recovers state C_1 from tuple space and reconstructs state C_3 by replaying the recorded operations for T_2 and T_3. The process resumes normal execution from C_3.

The PLinda message logging/replay method is similar to work by Strom and Yemini[59]. However, their and our techniques differ in several ways. First, the PLinda message logging/replay method is based on the transaction mechanism. Therefore, transaction processing overhead (mainly lock management in PLinda) is implicit in PLinda. Second, disk access is not required in PLinda. Finally, the PLinda message logging/replay method allows processes to save local state and recover from failure independently of other processes; by contrast, Strom and Yemini's techniques need to perform dependency tracking on recovery from failure and to roll back other processes (i.e., cascaded

rollbacks).

3.5 Coordinated Checkpointing

The message logging/replay method is efficient for applications in which processes have large continuations (e.g., processes based on data partitioning) but communicate (i.e., access tuple space) rarely. However, if processes communicate frequently, continuous message logging overhead may offset the performance gain due to the avoidance of saving continuation at each commit.

For such applications, PLinda supports the coordinated checkpointing method[1]. In this execution method, a process does not save its continuation to tuple space at each commit, and message logging is not performed during normal execution. Instead, the runtime system takes a global snapshot and saves it to disk periodically. This method is reminiscent of work by Koo and Tueg[45]. Specifically, this execution method is as follows:

1. As in message logging/replay, a process performs continuation committing only when the runtime system explicitly requests it. Instead, the process makes a local copy of continuation at each commit and keeps it until the next commit. Maintenance of a local copy is required because a continuation can be requested at any point.

2. Periodically, the tuple space server broadcasts a continuation committing request to every process — *activation* of coordinated checkpointing. Unlike transparent coordinated checkpointing which makes the entire system quiescent, the tuple space server still allows tuple space access and process creation operations but blocks transaction commit requests until the coordinated checkpointing operation is finished. Transaction commit requests need to be blocked in order to take a snapshot of a transaction-consistent global state.

3. Once the tuple space server receives continuations from all the processes, it saves a transaction-consistent state of the tuple space including those continuations to disk — *termination* of coordinated checkpointing.

4. Each transaction-consistent snapshot forms a *recovery line*. If a process or tuple space fails, then all the processes and the tuple space roll backs to the latest transaction-consistent snapshot from disk.

Figure 3.3 illustrates the coordinated checkpointing method. In the figure, processes only make local copies of continuation at each commit. Coordinated checkpointing is activated when processes 1, process 2, and process 3 are executing transactions T_4^1, T_3^2

[1] We do not aim at fine grain parallel applications but we believe PLinda can perform adequately on those applications which have a modest amount of communication. Regardless of fault tolerance, fine grain parallel applications do not perform well on networks of workstations because of slow communication.

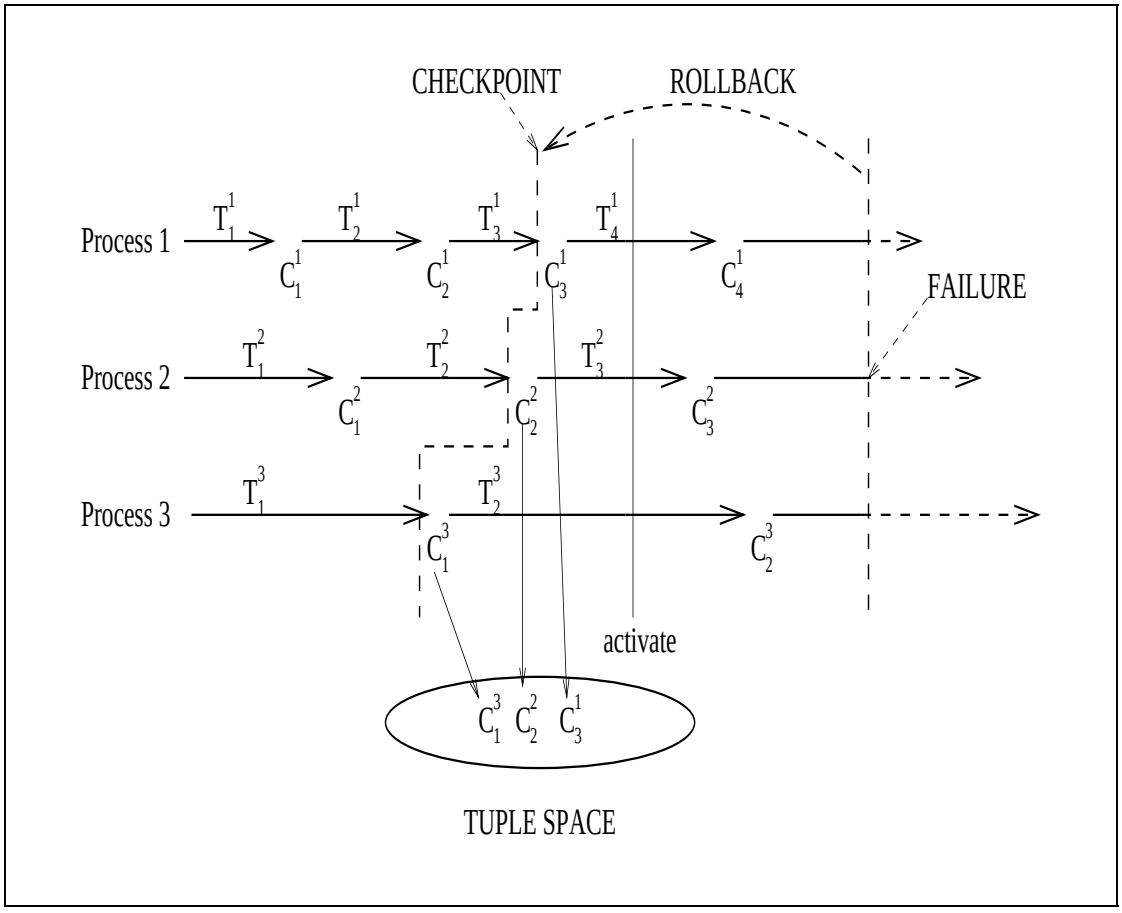

FIGURE 3.3: Coordinated Checkpointing

and T_2^3, respectively. Then, they save local copies of continuation made at the last commits to tuple space and the state of tuple space is checkpointed to disk.

Process 2 fails later and the failure forces all the processes and tuple space to restore their states from the last checkpointed state on disk. That is, failure recovery is expensive in this method.

3.6 Correctness Proof

In Chapter 2 and this chapter, we presented the PLinda fault tolerance scheme which consists of the fault tolerance mechanisms, programming techniques for constructing a fault-tolerant application and the three execution methods. In this section, we prove the correctness of the fault tolerance scheme.

Our proof strategy is as follows. We first formalize the properties of a fault tolerant PLinda program that is built according to the fault tolerance scheme. Based on the properties, we then prove inductively that for a completed execution of a fault tolerant PLinda program P, there is some equivalent possible execution of the Linda program L where L is obtained from P by removing statements aimed at fault tolerance. We call L the *Linda Kernel* of P.

We assume that an execution of a PLinda program may experience two types of failures: *tuple space* failure and *process* failure. Both of these are clean fail-stop failures. When the tuple space fails, the tuple space in virtual memory is corrupted but any checkpoints to secondary storage (which can be mirrored) are not disturbed. A process failure is the death of a process (though slowdowns may be treated as process death by the runtime system.)

We present a series of definitions leading to the correctness proof.

Definition 1 *For a process, a* local state *is defined to be the entire process image (i.e., the control stack and the data and code segments) at a certain point.*

Definition 2 *For a local state of a process, an* encoded continuation *(for short, a continuation) is defined to be the contents of a set of local variables from which the local state can be re-constructed.*

In Chapter 2, we showed a way to define an encoded continuation. Since a local state can be re-constructed from the corresponding continuation, we use them interchangeably in this proof.

Definition 3 *A state of tuple space consists of tuples and all the processes' latest continuations. Tuples may be locked. For a process, the continuation is not necessarily the one at the last commit.*

Definition 4 *A PLinda transaction is* abortable *if it accesses only local variables, unlocked tuples, tuple space variables locked by that transaction, or read-only files.*

The PLinda transaction mechanism can guarantee that no other transaction can access the updates or processes created by an aborted transaction.

Definition 5 *A* multi-transaction *process is defined as follows:*

1. *The execution is composed of a sequence of abortable transactions,* $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$*, for some* n*.*

2. *The execution makes a series of local state transitions,* $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ *where* s_0 *is the initial local state and* s_i *is the local state at the commit of transaction* T_i *for* $i = 1, \cdots, n$*. The commit of transaction Ti consists of unlocking any tuples that Ti modified and may consist of storing the continuation encoding the process state in tuple space at the end of Ti.*

Definition 6 *A PLinda process is* time-independent *if its output values and assignments to its local variables depend only on the values it reads from the tuple space and read-only files and on the initial setting of its program variables.*

For a time-independent process, delays do not influence behavior.

Definition 7 *A PLinda process is defined to be* Failure Tolerance-structured *(for short,* FT-structured*) if it is designed as follows:*

1. *The process is time-independent.*

2. *The process is a multi-transaction process.*

3. *The process saves a continuation* C_n *to tuple space at the commit of each transaction* T_n *for some* n*.*

4. *If the process fails while executing a transaction* T_{k+1}*, it restarts from the beginning of* T_{k+1} *by restoring a continuation* C_k *from tuple space.*

Note that this definition specifies how to design an FT-structured process, but, as we will see, the execution of the process may differ if the message logging/replay or coordinated checkpointing methods are used at runtime.

Definition 8 *A computation is* output-only *if:*

1. *The computation reads input data from read-only files or tuple space.*

2. *Commits are atomic. That is, no two commits overlap in time.*[2]

3. *Upon termination of the computation, all the processes terminate and all the transactions are committed.*

4. *The computation stores the final results in tuple space.*

5. *The end-user observes the only final results in tuple space as opposed to intermediate results or local variables.*

[2] In general, we need only that the execution is equivalent to this condition.

An output-only process may redo some of its transactions provided this repeated work does not affect its final outcome.

Definition 9 *An entire PLinda program is defined to be FT-structured if it is output-only and all the processes are FT-structured.*

In Section 2.5, we explained how to design a process to be FT-structured using transaction and continuation committing statements.

Definition 10 *For an execution of an FT-structured PLinda program, a* system state *consists of all the committed tuples in the tuple space and all the processes' continuations (or, local states) at the last commit points. The final state contains only committed tuples because all the processes terminate upon termination of the computation.*

Note that by this definition, executing a transaction doesn't change the system state (since local variable changes are ignored). Since the commit operation atomically changes both the tuple space and the last continuation of a process, all normal state changes occur at commit points.

Definition 11 *A* failure-free *system state is one that is reachable in a failure-free execution.*

Definition 12 *Suppose two output-only PLinda (or Linda) executions start from the same failure-free system state S. They are defined to be* equivalent *if they terminate with the same final system state S'.*

3.6.1 Proof

We consider operations or events in the execution of an FT-structured PLinda program. The execution consists of tuple space access or process creation operations (e.g., `in`, `rd`, `out`, `proc_eval`), transaction-start, transaction-commit, checkpoint-begin, checkpointing operations, checkpoint-end, process failure-occur, process recovery operations, process recovery-complete, and tuple space failure/checkpoint recovery.

Since transaction-start, tuple space access or process creation operations of some transation T are not observed by other processes until transaction T commits, we ignore them and consider only transaction commit.

We create a total order of these events based on their order in time. (If two are concurrent, then order them in either way.) We call this the *event schedule*.

For the sake of clarity, we describe rules of the checkpointing operation and the execution methods. We first consider the checkpointing operation. Recall that the checkpointing operation writes the tuple space to secondary storage media, e.g., disks.

1. The checkpointing operation is activated only when previous failures are fully recovered.

2. All the processes save their continuations at the last commits to tuple space before the checkpointing operation starts.

3. No process is allowed to commit transactions during checkpointing.

4. The initial checkpoint is empty. If the execution recovers a system state from the initial checkpoint, then it restarts from scratch.

5. The checkpointed state on disk is not corrupted even if a failure occurs during checkpointing. Dual checkpoints (the latest and the next latest checkpoints) are maintained; checksums and last update times are used to find the latest valid checkpoint on failure recovery.

We now consider the execution methods.

1. Commit-consistent execution method. This method executes the continuation committing operation at each commit. That is, tuple space always contains the continuation at the last commit for each process.

   If a process fails while executing the *(k+1)-th* transaction T_{k+1}, the method restarts the process from the beginning of T_{k+1} by restoring the continuation C_k at the commit of the *k-th* transaction from tuple space.

2. Message logging/replay. This method suppresses normal continuation committing operations. However, the tuple space server logs tuple space operations for each process since the last checkpoint; each process maintains a local copy of the continuation at the last commit. Further, all the processes saves local copies of continuations to tuple space at each checkpoint.

   If a process fails while executing the *(k+1)-th* transaction T_{k+1}, the process may restores a continuation C_i from tuple space for $i < k$. By replaying the execution history of the process (but omitting writes the process made to tuple space), the method guarantees that the process recovers C_k without affecting the execution of other processes.

3. Coordinated checkpointing method. As in the message logging/replay, this method suppresses the continuation committing operations, and every process maintains a local copy of the continuation at the last commit and saves the latest copy to tuple space at each checkpoint. However, the tuple space server does not log tuple space operations.

   If a process fails, the tuple space server is restored to the checkpointed state on disk and all the processes recover their states from the restored tuple space.

We now construct a correctness proof using the definitions given above.

Lemma 1 *For a terminating execution E of an FT-Structured PLinda program P that has suffered no failures and begins in a failure-free initial state, there is some equivalent execution of its Linda kernel L.*

Proof. During E, the operations aimed at fault tolerance (i.e., transactions, continuation committing, and tuple space checkpointing) only delay other Linda or Linda-variant

operations by locking. They also generate data (continuations or checkpoints) which are never read as long as there is no failure.

However, the Linda model does not specify how long a tuple access operation takes or what tuple it retrieves. Thus, delays due to locking could happen in a failure-free execution of the Linda kernel. Since E has not experienced any failure, continuation committing and checkpointing operations do not affect other Linda operations on which the final system state (or, output) depends. Hence, there is some equivalent execution of L for E.

Theorem 2 *For a terminating execution E of an FT-Structured PLinda program P beginning in a failure-free state, there is some equivalent execution of its Linda kernel.*

Proof. We first show that the final system state of E is failure-free (i.e., reachable by a failure-free execution). That is, E is a possible result of a failure-free execution E'. We prove this by induction on the event schedule.

1. Suppose that the event schedule of E has n events, the system state reached after the *i-th* event is S_i for $1 \leq i \leq n$, and S_0 is the initial state. We proceed by induction.

2. Base case. By definition, the initial system state S_0 is failure-free.

3. Inductive case: By event. Suppose that S_i is failure-free for $1 \leq i \leq k$. Then, we show that S_{k+1} is failure-free. Let the *(k+1)-th* event be:

   (a) Transaction-commit, This commit operation transforms the system state S_k into another system state S_{k+1}. Since S_k is failure-free and E does not experience failure between the *k-th* and *(k+1)-th* events, S_{k+1} could arise in a failure-free execution.

   (b) Checkpoint-begin or checkpointing operations. During checkpointing, the system state is not changed and the checkpointing operations do not affect the system state or corrupt the latest checkpointed state on disk even if a failure occurs. Therefore, $S_{k+1} = S_k$ and so S_{k+1} is failure-free.

   (c) Checkpoint-end. The current system state is saved to disk, but not changed. Therefore, $S_{k+1} = S_k$ and so S_{k+1} is failure-free.

   (d) Tuple space failure/checkpoint recovery. On tuple space failure, the entire execution stops and E is rolled back to the system state S_i at the last checkpoint-end event where $i \leq k$. By inductive assumption, S_i is failure-free. So is S_{k+1}.

   (e) Process failure-occur, process recovery operations or process recovery-complete under the commit-consistent execution and message logging/replay methods. Suppose that process p in E fails while executing the *(i+1)-th* transaction.

      i. Commit-consistent execution method. The *(i+1)-th* transaction is aborted and p restarts from the beginning of the *(i+1)-th* transaction. Since p is FT-structured, no other process can read the values written by the

aborted transaction. The net effect is as if p is delayed between the *i-th* and *(i+1)-th* transactions. Therefore, the failure recovery does not affect the execution of other processes or the system state. Therefore, $S_{k+1} = S_k$ and so S_{k+1} is failure-free.

ii. Message logging/replay method. Failure recovery is handled in a similar way as in the commit-consistent execution method. In the message logging/replay method, the operations in the history log are also replayed. This replaying operation does not affect the execution of other processes or the system state. Therefore, $S_{k+1} = S_k$ and so S_{k+1} is failure-free.

(f) Process failure-occur, process recovery operations or process recovery-complete under the coordinated checkpointing method. E is rolled back to the state S_i at the last checkpoint-end event where $i \leq k$. Thus, $S_{k+1} = S_i$. By inductive assumption, S_i is failure-free. So is S_{k+1}.

Hence, S_i is failure-free (i.e., reachable by a failure-free execution) for $i = 1, \cdots, n$. That is, the final system state (i.e., output) of E is always reachable by a failure-free execution E'. E and E' are equivalent.

By Lemma 1, there is some execution E_L of the Linda kernel which is equivalent to E'. Therefore, E_L is equivalent to E.

3.7 Related Work

There has been a considerable amount of work done on checkpointing and rollback-recover techniques for distributed systems whose execution mechanisms are similar to the PLinda mechanism. In this section, we will only review the most closely related work. See [22] for a comprehensive survey.

Among various approaches to making parallel/distributed systems failure-resilient, backward error recovery is the most general and commonly used[37]. Backward error recovery is the only known mechanism that can tolerate faults which were unexpected at system design time[54, 37]. For example, the approach can make arbitrary distributed programs fault-tolerant in a programmer-transparent manner. The technique used in backward error recovery is checkpointing and rollback.

Checkpointing techniques have been widely used for database systems to make recovery fast[6, 34] Since database applications do not require process resiliency, those techniques don't recover process state. However, they ensure that committed transactions are serialized and never lost. They must do so because the user can see results of committed transactions immediately and expect them to survive failure. In contrast, PLinda supports process resiliency, but allows committed transactions to be lost for better runtime performance.

There are two common and general approaches to checkpointing and rollback recovery in distributed systems: *coordinated checkpointing* and *message logging/replay*. In coordinated checkpointing, (also called synchronous or distributed checkpointing), all processes stop at checkpoint time, synchronize to agree on a consistent global state and

write their states to disks[45, 41] together. If a process fails, then all the processes usually roll back to the last checkpoint.

In message logging/replay[59, 39], also called independent checkpointing, processes save local state to disk independently of one another. They also record all the inter-process messages in message logs and periodically save the log to disk. On process failure, the rollback-recovery scheme restarts the failed process from the previous checkpoint (which may not be consistent with the states of the other processes) and replays the messages in the same order to restore the process back to a consistent state. Therefore, the processes must be *deterministic*; that is, given the same input, they should produce the same output. In spite of message logging, this technique may still lose messages during rollback-recovery because the message log is only periodically saved to disk. In this case, the rollback-recovery scheme forces all the other processes depending on the those messages to roll back to previous checkpoints.

Coordinated checkpointing aims at low runtime overhead during normal execution. It makes the entire system quiescent during checkpoint, and a single failure requires the entire system to roll back to the last checkpoint[3]. In contrast, message logging/replay incurs low rollback-recovery overhead and aims at reducing the amount of lost work on failure. However, the algorithms are relatively more complicated and entail more runtime overhead due to message logging.

PLinda offers a flexible recovery mechanism that can be partly tuned at runtime. This allows PLinda to be usd for a variety of parallel applications.

Drawbacks are that it requires programming work for fault tolerance and that it incurs transaction processing overhead during normal execution. As explained in Chapter 2, PLinda supports lightweight transactions which do not cause high runtime overhead to coarse grain parallel applications. For coarse grain parallel applications the additional programming work is often negligible.

3.8 Summary

In this chapter, we described the tunable execution mechanism of PLinda for resilient processes. We gave the motivation for our tunable approach to execution: an execution model needs to be flexible in order to to deal with different application characteristics.

The message logging/replay and coordinated checkpointing methods are similar to work by work by Koo and Tueg[45] and Strom and Yemini[59], respectively. Their techniques are aimed either at supporting efficient normal execution or at avoiding massive rollback on failure, not both. In contrast, PLinda allows the end-user to choose one of three execution methods for an application at runtime, depending on application characteristics. Table 3.1 shows a comparison of the three execution methods..

[3]There are dependency tracking techniques that may reduce the number of processes to rollback.

Commit-consistent execution	
1	Only uncommitted transactions
2	Continuation committing at each commit
3	Yes
4	With small continuations
Message logging/replay	
1	Both uncommitted and committed transactions
2	Periodic continuation committing but continuous message logging
3	Yes
4	With large continuations but rare access to tuple space
Coordinated checkpointing	
1	The entire system rolls back to the last checkpoint
2	Only periodic continuation committing
3	No
4	With large continuations and a modest amount of access to tuple space

1: Work lost on failure

2: Overhead during normal execution

3: Independent failure recovery

4: Suitable Applications

TABLE 3.1: Comparison of the Three Execution Methods

Chapter 4

Using Idle Workstations for Parallel Computation

4.1 Introduction

The increasing availability of powerful networked workstations has resulted in high performance distributed computing systems. Typically such networks of workstations are principally used to support individual users connected by email, but can also be used as distributed parallel systems attempting to solve large scientific or engineering problems. Thus, there is a large potential for these underutilized machines to run parallel applications. Any system which attempts to use underutilized machines on a network for sequential/parallel computation must address three issues:

- Idleness detection.
- Scheduling.
- Process migration from machines when they become busy.
- Process migration to machines when they become idle.

There has been a considerable amount of research done on how to utilize idle or under-utilized workstations connected by a network[2, 14, 18, 11, 25, 28, 44, 48, 52, 53]. In that work various systems were developed to address the above issues. We call them "work-stealing" systems.

Among them, there are systems[2, 11, 25, 44] which support parallel computation, but few support both fault tolerance and idle workstation utilization. However, it is crucial to provide all three facilities for long-running parallel applications on networks of workstations: parallelism for computation instensive tasks, fault tolerance for both hard and performance failures, and idle workstation utilization for cost effectiveness.

For idleness detection and scheduling we do not intend to re-invent techniques for PLinda. Instead, we take advantage of various techniques already developed by other work-stealing systems[2, 14, 18, 11, 25, 28, 44, 48, 52, 53]. The process migration issue is

addressed by treating "busy" machines (i.e., those which are being used by their owners) as failed; that is, owner activity is handled as failure. More specifically, we migrate processes from busy machines to idle ones by killing the processes on busy machines and recovering them on idle machines by the existing PLinda fault tolerance mechanisms.

This chapter is organized as follows. In Section 4.2, we describe the issues involved in workstation idleness detection and process scheduling, and explain how the current PLinda design addresses those issues. Section 4.3 presents the rationales and advantages of using the PLinda fault tolerance mechanisms for process migration. In Section 4.4, we explain how the current PLinda infrastructure is designed to utilize idle machines. Related work is reviewed in Section 4.5. In Section 4.6, we conclude this chapter with a summary.

4.2 Finding Idle Workstations and Scheduling Processes

We first discuss the issues involved in detecting whether a machine is idle or busy and how to do scheduling. Then, we describe how the current design of PLinda addresses them.

4.2.1 Issues

Intuitively, a machine is defined to be busy when the owner is using it; otherwise it is idle. Unfortunately, it is not simple to determine whether an owner is using his or her machine at any given instant. Constant polling might be used but incurs significant overhead. For this reason, work-stealing systems have devised idleness criteria which the runtime system can use to detect the idleness of a machine without polling. Common idleness criteria are:

- *Keyboard, mouse and console idle times.*
- *Load average.*
- *The number of users logged on.*
- *Remote-login-session idle time.*

Besides these, there are other criteria such as time of day which are used relatively less often. See [44] for a comprehensive description about idleness criteria.

There have been several studies to find effective settings for idleness criteria[44, 53]. Kaminsky reported that the Piranha project team had run large production applications on a volunteered pool of over 60 workstations and only one machine was withdrawn from the pool. The idleness criteria used in those experiments were that the keyboard, mouse and remote logins had to be idle for five minutes and the one, five and ten minute load averages had to be below 0.4, 0.3 and 0.1, respectively. Other experimental results have also shown the effectiveness of similar idleness criteria[53].

Also, there have been various studies to measure how often workstations are idle[44, 52]. They found that machines are idle most of the time. For example, Mutka and Livny

found that 70%-80% of machines are idle weekends and that 50% were idle even at peak times[52]. Kaminsky reported that 86%, 94%, and 75% were idle on average, at night, and during the daytime, respectively. Also, we have conducted a one week experiment with 19 machines at the NYU computer science department. Our results showed that 86%, 92%, and 60% were idle on average, at night, and during the daytime, respectively.

We will now discuss process scheduling issues. A scheduler may be either centralized or decentralized. The advantage of a centralized scheduler is that it is simple and easy to implement because there is only one scheduler process which maintains any necessary information. A centralized scheduler works efficiently for small and medium scale systems. For these reasons, most systems designed to utilize idle workstations employ centralized schedulers. However, a centralized scheduler generally has a scalability problem; that is, the scheduler becomes a bottleneck as the numbers of processes and processors increase.

In contrast, a distributed scheduler, which by definition consists of multiple processes, does not cause a single process to be a bottleneck. But the scheduler processes must achieve agreement on resource allocation. Such agreement is both difficult to implement and expensive at runtime due to required communication and synchronization costs.

There are also hybrid approaches to scheduling. Market systems[18, 44, 49, 62] are an example. These systems apply the idea of economic bidding to the scheduling problem. Each machine bids autonomously and a single broker process collects the bids. The broker assigns tasks to machines based on their bids.

Regarding scheduling parallel applications over networks of workstations, another important issue is supporting multiple applications. Since many users share networks of workstations, there are likely to be multiple parallel applications running concurrently. Therefore, scheduling multiple parallel applications is crucial to effectively utilizing idle workstations. Also, both throughput and turnaround time can benefit because most parallel applications do not exibit linear speedup; that is, they are more efficient on a smaller number of processors[44]. A commonly used scheme is to partition processors and to assign partitions to applications. For example, processors can be evenly divided or each processor can select an application randomly.

Scheduling processes over distributed computing systems is still an important research issue and idleness criteria need more study to determine how to use idle times of machines more effectively and to satisfy cautious owners. We expect that there will be more work on these issues. For PLinda, we do not intend to re-invent techniques for detecting idle machines and scheduling processes over those machines. Instead, we intend to use the techniques which are already developed and have proven to be effective.

4.2.2 The Current PLinda Design

In the current implementation of PLinda, we are using keyboard, mouse and console idle times for idleness criteria. The implementation is based on a centralized scheduler and supports multiple applications. How the scheduler works will be explained in detail in Section 4.4.

4.3 Process Resiliency as Process Migration

In this section, we discuss process migration in PLinda. We first explain two seemingly orthogonal but similar ideas: process migration for utilization of idle machines and process failure-resiliency. Then, we show how process failure-resiliency can be used efficiently for process migration in PLinda. We also explain why process resiliency has not been previously used for process migration. We conclude this section by discussing the need for fault-tolerance in an environment that utilizes idle machines and the merits of our method for process migration over prevalent methods.

Process migration and process failure-resiliency are designed for different purposes. Process resiliency for parallel computation aims at enabling processes to continue and finish computation correctly even though some of them experience failure during execution. In contrast, process migration is intended to move processes from overloaded or busy machines or to under-utilized or idle machines.

In spite of different motivations, the two techniques require the same functionality — *moving processes from one machine to another during execution.* Therefore, process resiliency can theoretically be used for process migration. That is, we can treat process migration as failure recovery by killing a process on the machine that has become overloaded and allowing the fault-tolerance mechanisms to restart the process on another machine. In fact, process resiliency performs process migration under a more difficult condition — unexpected failure.

Heretofore, process resiliency has not been used for process migration for a number of reasons. First, a process usually has to lose a lot of work on failure recovery (or, migration). Most techniques for process resiliency entail periodic checkpoints of process images to disk. On failure recovery, processes roll back to the last checkpoint on disk[22]. In general, checkpoints are taken infrequently for runtime efficiency and therefore, failure recovery causes a lot of work to be lost. Process migration for load balancing reasons may occur frequently and therefore a lot of work should not be lost on migration.

In addition to losing a lot of work on failure, failure recovery usually requires not only failed processes but also live processes to roll back to previous checkpoints. For example, techniques for coordinated checkpointing[45] require massive rollback on failure, and those for message logging/replay may also require cascading rollbacks on failure.

In contrast to other fault tolerance methods the following features of the PLinda commit-consistent fault tolerance execution method make it suitable for process migration on networks of heterogeneous workstations.

- *Independent continuation committing and rollback-recovery.* Each process can perform continuation committing and rollback-recovery operations independently of other processes. This allows a process to migrate without affecting other processes.

- *Lightweight mechanisms.* Continuation committing and rollback recovery save only a few process variables to tuple space at commit and retrieve them on recovery. These operations are efficient because they do not require disk access or re-execution of committed transactions.

- *Heterogeneous processing*[1]. Continuation committing and rollback recovery are based only on process variables which are architecture-independent. This allows processes to migrate among heterogeneous machines.

In addition to supporting both process migration and failure-resiliency, the fault tolerance mechanism based technique has another merit. As explained earlier, a typical way to utilize workstations is to use them for parallel computation when they are idle. In this case, a work-stealing system must immediately retreat processes from a machine when the owner resumes working on the machine. On retreat, process migration facilities first prepare for migration (e.g., save process images) and then move processes to another machine. Thus, retreat is slow. In contrast, PLinda treats owner activity as failure and immediately destroys processes on the machine. Thus, retreat is fast. Therefore, parallel computation with the PLinda process migration facility will not disturb the owners of private workstations.

In summary, PLinda is one of very few systems to support fault tolerance and utilization of workstations for long-running coarse grain parallel applications. The user can run long-running parallel applications without fear of processor failure during the long period of execution or disturbing owners of private workstations. Also, using a single mechanism for fault tolerance and process migration allows simplicity in system implementation and quick retreat on owner activity.

4.4 PLinda Infrastructure

The PLinda infrastructure is designed to allow the user to use networks of workstations as a parallel virtual machine which both supports fault tolerance and utilizes only idle workstations. The parallel virtual machine reconfigures itself automatically as the status of each processor changes (e.g., a process fails or becomes busy or idle).

4.4.1 Major Runtime System Components

The prototype PLinda system is based on the client-server architecture model. In the PLinda system, application processes are clients and the runtime kernel is the server. The current runtime system consists of three major components:

- Daemon processes. The runtime system runs a daemon on each workstation. Using idleness criteria explained in Section 4.2, the daemon monitors the idleness status of the local host machine and informs the server of status changes to the machine. Also, it manages processes on the machine; that is, it invokes application processes upon request from the server and kills them as soon as the machine becomes busy. While the owner of a machine is working, the daemon intermittently checks the status of the machine and sleeps the rest of the time so as not to disturb him or her.

[1] The current system runs on only Sun Sparc workstations, but other ports are underway and should be easy.

- The server[2]. The server manages tuple space and schedules client processes. The scheduler in the server is designed to meet the following criteria:
  - Allow multiple applications. As explained in Section 2.4, applications do not affect one another due to multiple tuple spaces.
  - When processes outnumber physical processors, multiple processes should be spawned on each machine.
  - Rely on the OS kernel to schedule processes running on each machine. As explained in Section 2.7, PLinda processes are, in fact, processes at the OS level. Therefore, once multiple processes are spawned on a workstation, the OS kernel on the machine automatically schedules them.
  - The number of PLinda processes on each machine should be used as the measure of the workload for that machine.
- Administration process. The administration process allows the PLinda user to observe and control runtime behaviors during execution. Also, the user can add or delete workstations to the parallel virtual machine at any time.

4.4.2 Parallel Virtual Machine in PLinda

The current PLinda prototype allows the user to use networks of workstations as one fault-tolerant parallel virtual machine whose processors are only idle workstations. However, the current implementation assumes that the user runs the server and administration processes on machines dedicated to the parallel application. That is, the server and administration processes are not migrated during execution.

In the prototype system, such a virtual machine is implemented as follows:

1. *Processor pool.* The parallel virtual machine consists of a pool of processors which are running or failed. Idle workstations are treated as running processors and busy or failed workstations are treated as failed processors. Only running processors participate in computation.

2. *Registering a processor.* Using the administration process, the user can register a workstation in the PLinda runtime system at any point. This causes the administration process to create a daemon on the workstation.

3. *Process creation.* If the server needs to create a client process, it selects one of the running processors which executes the smallest number of processes (i.e., a least loaded one), and spawns the client process there.

4. *Process migration.* When a running processor becomes failed (because of either owner activity or a real processor failure), its daemon or the failure immediately destroys all the running client processes on the machine. Then, the server is either informed of the simulated failure or it detects the real failure.

[2]The current implementation is based on a single server.

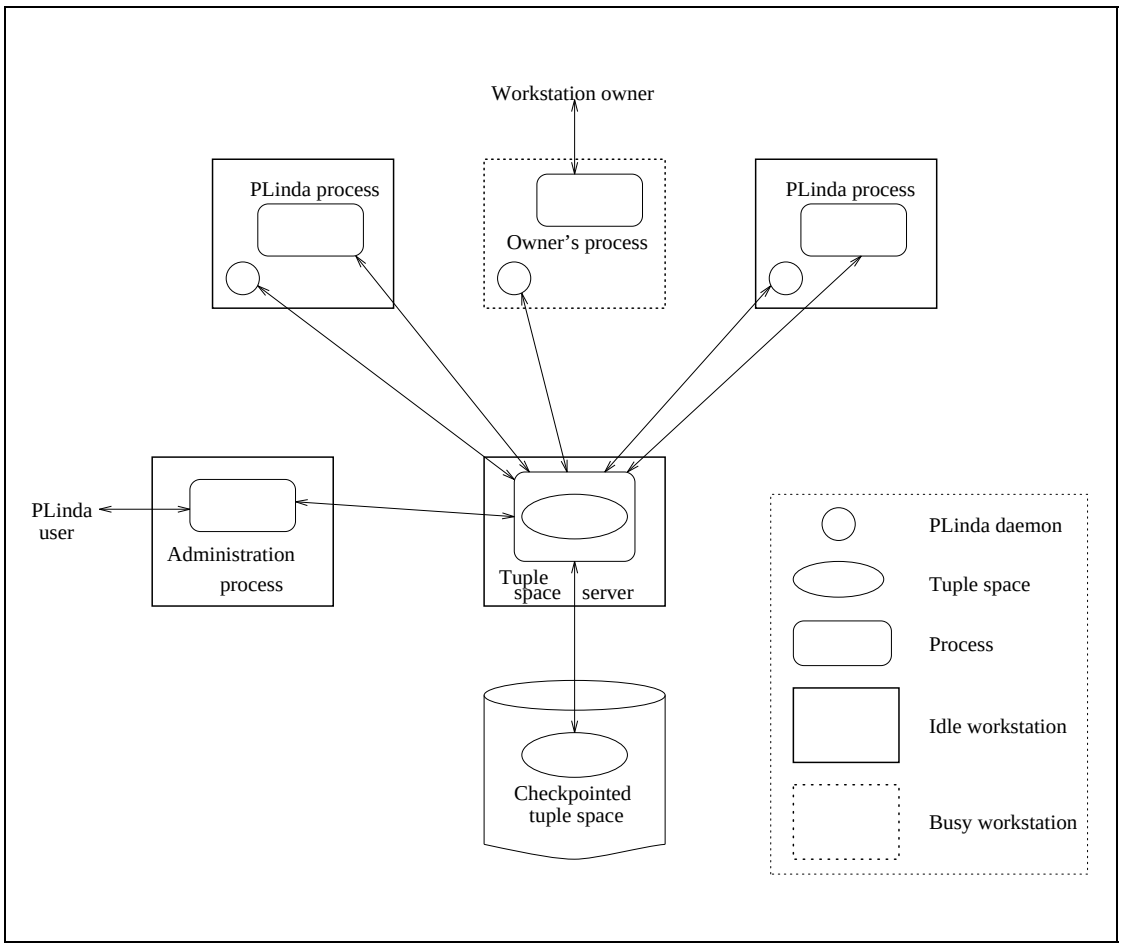

FIGURE 4.1: Parallel Virtual Machine in PLinda

5. *Load balancing.* Periodically, the server performs load balancing. It migrates processes from overloaded processors to less loaded ones. Since process migration is relatively expensive, load balancing is performed only periodically. The frequency is a tuning parameter.

Figure 4.1 illustrates the virtual machine. In the figure, there are three workstations A, B, and C in the processor pool. There is one daemon running on each machine. A and C are idle and being used for parallel computation by the runtime system. B is being used by its owner. However, the daemon on B is still monitoring if B is idle, and when the owner is away from B, it will inform the server.

4.5 Related Work

In this section we will give a brief overview of the vast amount of recent research on process migration which is aimed at harnessing the power of underutilized workstations. These systems can be classified by their target applications into two categories:

sequential/semi-parallel and parallel.

The sequential systems[48, 18] offload sequential jobs (i.e., jobs that do not have internal parallelism) from overloaded machines to under-utilized or idle ones. Since these systems are only intended for sequential jobs, they are not applicable to parallel application users. There are also systems that are aimed at those semi-parallel applications which consist of only independent tasks (no inter-task communication)[16, 62, 65, 67]. The sequential/semi-parallel systems usually support fault tolerance using transparent checkpointing and rollback recovery techniques, but this scheme does not readily support the use of heterogeneous workstations.

In contrast to the sequential/semi-parallel systems, PLinda supports fault tolerance and the utilization of idle processors for arbitrary parallel applications on networks of heterogeneous workstations. Also, because the programmer can customize fault tolerance based on application characteristics, PLinda is an efficient and convenient way to execute semi-parallel applications.

The parallel systems which intend to utilize idle workstations for parallel applications[2, 11, 21, 25, 44] either depend on support from the underlying operating systems or assume restricted programming models or characteristics about the applications they are intended for. Few of them can tolerate failure during parallel computation.

Among those systems, Piranha[44] is a Linda-variant system aimed at utilizing idle workstations. The current design of the PLinda idleness detection mechanisms is inspired by ideas proposed by Piranha.

4.6 Summary

In this chapter, we discussed how to utilize idle workstations for parallel computation in PLinda. Throughout our discussion, we considered three issues: idleness detection, process scheduling and process migration.

In the design of PLinda, we do not attempt to study techniques for idleness detection and process scheduling, but intend to take advantage of those already developed by other work-stealing systems. In Section 4.2, we discussed process scheduling and idleness detection and how the current PLinda design addresses those issues.

A novel feature of PLinda is to treat process migration and process failure-resiliency uniformly. More specifically, we use the fault tolerance mechanisms to migrate processes from busy machines to idle ones by treating owner activity as processor failure. The fault tolerance mechanisms can be used for process migration because they are lightweight and allow a process to save its state and recover from failure independently of other processes. That is, a process can be migrated efficiently and independently, using the fault tolerance mechanisms. Also, since the PLinda fault tolerance mechanisms can be designed to support heterogeneous processing, processes can be migrated over heterogeneous machines.

There are two principal advantages to using the same mechanism for fault tolerance as for process migration:

1. it is simpler

2. it allows for an entirely general design of migratable processes, since migration can occur as an instantaneous failure.

Chapter 5

Implementation and Experiments

In this chapter, we explain the implementation of the PLinda prototype and present performance results. The PLinda prototype system is based on the client-server architecture model. In the current implementation, application processes are clients and the runtime kernel is the server. The server is not yet distributed.

The prototype is built on UNIX and TCP/IP using C++. The implementation consists of four major components: the server program, the daemon program, the administration program, and the client library.

There is a vast amount of literature about the implementation of tuple space and transaction processing systems[9, 6, 12, 34, 46]. In the design of the PLinda runtime system, we have focused on how to incorporate existing implementation techniques for transactions into those for tuple space. Throughout this chapter, we therefore concentrate on presenting implementation strategies rather than describing implementation details.

This chapter is organized as follows. In Section 5.1, we explain the implementation of the PLinda server by describing the architecture, tuple space management, transactions, process management, tunable execution of continuation committing, and tuple space checkpointing.

In the following sections, implementations of the PLinda daemon and the administration tools are discussed, respectively. Then, we present performance results from experiments in Section 5.4. Finally, we summarize this chapter in Section 5.5.

5.1 PLinda Server

The current PLinda client-server prototype uses a single server and many application processes being clients.

The server performs the following functions:

- Manage tuple space. The server provides transactional tuple space access and process creation.
- Manage processes. The server schedules processes and also detects failure and

recovers failed processes. As explained in Chapter 4, the server treats non-idleness as failure.

- Support fault tolerance mechanisms. The server provides mechanisms for transactions, continuation committing and tuple space checkpointing.
- Manage workstations. The server runs one daemon process on each workstation to monitor idleness.

5.1.1 Architecture

The architecture of the PLinda server is designed in an object-oriented style. In the program for the server, runtime resources (e.g., processors, client processes and tuple groups) and fault tolerance abstractions (e.g., transactions and checkpointing) are represented as C++ classes. That is, these classes define the runtime behaviors of these resources and abstractions. The main C++ classes are:

- `Scheduler`. In the server, one `Scheduler` object is created and it schedules all the tasks. The tasks include servicing client requests (e.g., tuple space access, process creation and transaction operations), checkpointing tuple space and communicating with clients.
- `CheckpointManager`. One `CheckpointManager` object is created at runtime. This saves tuple space to disk periodically and on failure recovery, restores the checkpointed state from disk.
- `ClientProcess`. The server maintains one `ClientProcess` object for each executing client process. Each `ClientProcess` object services requests (e.g., tuple space access) from its corresponding real process. A workstation may have several real processes.
- `DaemonProcess`. Like `ClientProcess` objects, the server creates one `DaemonProcess` object for each real PLinda daemon process. Each PLinda daemon process is responsible for one workstation.
- `Transaction`. The server maintains one `Transaction` object for each active transaction. A `Transaction` object keeps track of tuple space access operations by maintaining lock information about accessed tuples. It also performs the commit or abort operation for the transaction.
- `TupleGroup`. The server has one `TupleGroup` object for each tuple group. A `TupleGroup` object manages tuples for its tuple group. It also saves the state to disk when requested by the `CheckpointManager` object.
- `ServerCommLink` and `ClientCommLink`. Every client process and daemon process are required to set up a connection at the beginning of their execution. Each connection consists of two objects: a `ServerCommLink` object on the server side

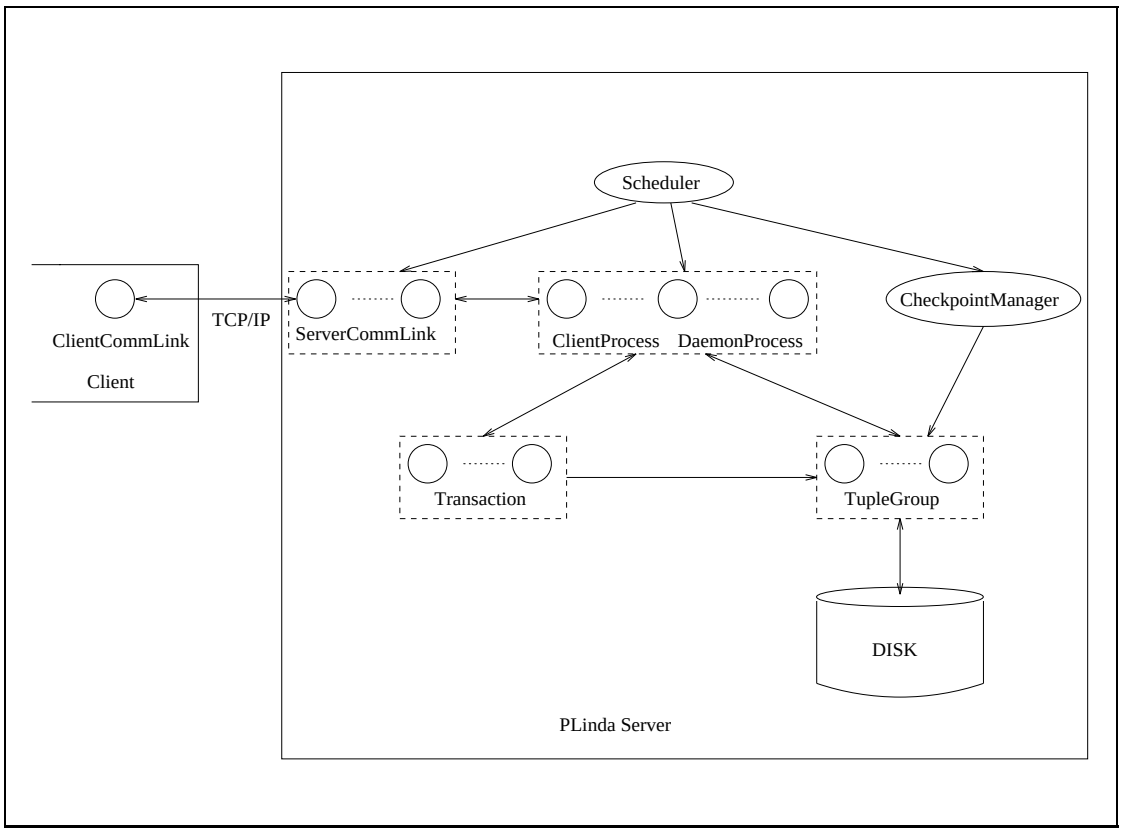

FIGURE 5.1: Server Architecture

and a `ClientCommLink` object on the client or daemon process side. These objects hide all details about communication from the other objects.

Figure 5.1 shows the architecture of the server.

5.1.2 Tuple Space Management

In PLinda, the tuple space storage mechanism is required both to manage tuples and to cooperate with transaction mechanisms. Unlike common storage management systems for transactions which are usually aimed at efficient and reliable manipulation of data on disk, the current PLinda storage management for tuple space is focused on in-memory data for the following reasons. First, in the current prototype, it is assumed that tuple space is used for communication and coordination among processes, but not for management of a large amount of data. Thus, the entire tuple space is designed to reside in the address space of the server. Second, the server does not access disk except for periodic tuple space checkpointing and rare recovery from tuple space failure.

The current implementation of tuple space management is designed as follows. The server loads the entire tuple space at the beginning of an execution or on recovery from tuple space failure and manipulates it inside virtual memory throughout the execution. Therefore, there is no buffer (cache) management intended for efficient access to data on

disk. The server uses the memory management facility (i.e., `new` and `delete`)supported by the C++ language library to allocate memory space for tuples. This design allows for efficient tuple manipulation of the PLinda operations except during tuple space checkpointing and recovery. For tuple space checkpointing and recovery, this design requires the flattening operation (i.e., converting tuples from an in-memory format to a byte stream without pointers) at each checkpoint and the inflating operation (i.e., the reverse operation of flattening) on recovery.

In PLinda, tuple space is partitioned into tuple groups. The server maintains a `TupleGroup` object for each tuple group. A `TupleGroup` object (which we will call a tuple group unless clarification is needed) manages tuples as follows:

1. *Tuple Group Partitioning.* The tuple group partitions the tuples according to their patterns and maintains a linked list whose nodes point to each tuple pattern. (We call the list the tuple pattern list.) A pattern is a sequence of types $t1, t2, \cdots, tn$. A tuple has a pattern P if the number of fields in the tuple equals the number of types in the pattern and every field i in the tuple is of type t where t is the i-th type in the pattern.

2. *Data Structures.* The tuple group also maintains both a linked list (called a tuple list) and a hash table (called a tuple hash table) to manage the tuples in the same tuple pattern. It currently uses the first field of the tuple for hashing.

3. *Tuple Creation.* For a tuple creation request, the tuple group first searches for a node in the tuple pattern list which represents the pattern of the tuple to be created. If the tuple group fails to find such a node, then it creates a new node in the tuple pattern list which represents the pattern of the new tuple. The tuple group inserts the tuple (more precisely, the pointer to the tuple) into the tuple list and tuple hash table.

4. *Tuple Retrieval.*

   (a) *Searching.* For a tuple retrieval request, the tuple group first searches for the corresponding node in the linked list of tuple patterns. If the request has a constant value in the first field, the tuple group uses hashing to find a matching tuple; otherwise, it uses sequential scanning.

   (b) *Lock compatibility checking.* When the tuple group matches a tuple retrieval request by finding an existing tuple, it checks lock compatibility. If there is a conflicting lock on the tuple, then the tuple group considers the matching operation to have failed. In the next subsection, lock management will be discussed in more detail.

   (c) *Blocking.* The tuple group blocks tuple retrieval requests which fail to find matching tuples. It maintains a linked list for blocked requests. It checks blocked requests in this list whenever it creates a tuple.

Figure 5.2 shows the storage structure of a `TupleGroup` object.

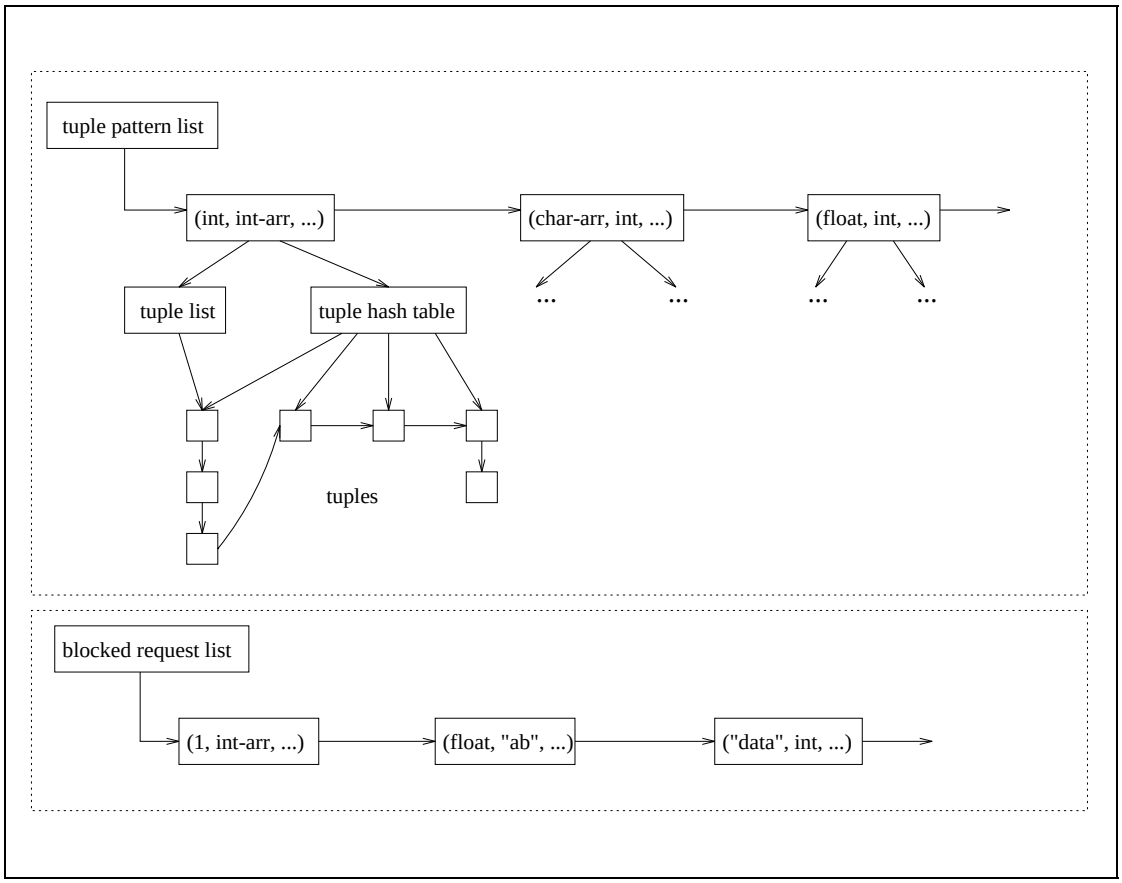

FIGURE 5.2: Structure of a Tuple Group

In the current implementation, we use the GNU C++ class library for data structures such as linked lists and hash tables. To maintain portability, all the interfaces with the class library are encapsulated inside a few wrapper classes so that the GNU class library-dependent code can be easily replaced.

5.1.3 Transactions

The traditional transaction model is designed to support *concurrency control* and *reliability*. As explained in Section 2.3, the PLinda transaction mechanism provides only concurrency control for maintaining a consistent state of the tuple space in spite of process failure. For reliability of the tuple space, checkpointing is used.

For each transaction, the server maintains a `Transaction` object. This object manages all the locking information for concurrency control and carries out the commit and abort operations.

In the current prototype, we use a variant of two phase locking (2PL)[6, 34] for concurrency control. In 2PL, transactions first obtain locks on data items and then access them. Also, transactions release locks only when they commit. That is, for each transaction, there are two phases: the first phase in which locks are obtained, and the second phase where all the locks are released (technically, this is known as strict two phase locking, but is the form normally used). When a transaction attempts to access a data item already locked by another transaction, the transaction is blocked until the other transaction commits and releases the lock on the data item.

The standard 2PL scheme is not well-suited to the Linda model for two reasons. First, in Linda, a tuple retrieval operation (`rd` or `in`) is blocked when there is no matching tuple; that is, this blocking happens regardless of transactions. Thus, if the standard 2PL is added to the Linda model, then there are two cases for blocking: in the presence of a locked matching tuple and in the absence of a matching tuple.

Second, 2PL may decrease concurrency more than necessary. For example, suppose that when a process attempts to retrieve a tuple, all the matching tuples are locked. In this case, both 2PL and the Linda model block the process. However, 2PL blocks the process on the lock of one of the matching tuples and therefore releases the process only when the transaction that holds the lock commits. That is, 2PL does not release the process when a new matching tuple is inserted later. In contrast, the Linda model releases the blocked process immediately when a new matching tuple is added. In Linda applications, such a case is common.

In the current PLinda prototype, 2PL is modified as follows.

1. There are three lock types: `rd`, `in` and `out`. The `rd` lock corresponds to the read (or, shared) lock in 2PL and the `in` and `out` to the write (or, exclusive) lock. Thus, `rd` locks are compatible with one another, but `in` and `out` locks conflict with any other locks.

2. As in standard 2PL, tuple retrieval operations in a transaction search for tuples which are not locked or whose locks are compatible. However, while searching,

these operations do not block on locks; that is, they skip data items which have conflicting locks.

3. As explained in Subsection 5.1.2, retrieval operations block only when there is no matching tuple available. However, they are released immediately when a matching tuple which has no lock or a compatible lock is added.

4. Transactions by default release `rd` locks immediately after access — degree 2 serializability. However, the server can be tuned to support full serializability (the standard 2PL) where read locks are released after the transaction commits.

Each `Transaction` object maintains three linked lists for locks: one for `rd` operations, another for `in` operations, and the third for `out` operations. We call them the *rd list*, the *in list* and the *out list*, respectively.

In the current server, transaction commits and aborts are executed as follows:

1. *Commit.* The corresponding `Transaction` object unlocks the tuples pointed to by the rd and out lists and deletes the tuples pointed to by the in list.

2. *Abort.* The corresponding `Transaction` object unlocks the tuples pointed to by the rd and in lists and deletes the tuples pointed to by the out list.

In both cases, the `Transaction` object is removed after the commit or abort operation is finished.

5.1.4 Process Management

One of the features that distinguishes PLinda from Linda and other parallel/distributed programming models is transactional process management. In PLinda, transactions govern process management almost in the same way as tuple space access. For example, processes created inside a transaction start to execute only after the transaction commits and if the transaction is aborted because of failure, these process are automatically destroyed. In the case of tuple space failure, the server loses all the updates made by the transactions committed after the last checkpoint. Thus processes created by these lost transactions must also be destroyed.

Special Tuple Groups for Process Management

The PLinda server maintains three special tuple groups for process management. They store tuples which are generated by execution of `proc_eval` operations. Execution of a `proc_eval` operation creates two tuples: the *argument tuple* and the *identity tuple*. The argument tuple is constructed by evaluating a series of expressions passed to the operation. The first field of an argument tuple must be the name of an executable file. The identity tuple contains identity information about the process to be created by the `proc_eval` operation. The information contains: (1) a system-assigned process identifier, (2) the file name of the executable to be run, (3) the user-interactive mode,

and (4) the number of times that the process has been re-spawned/spawned (which we call the creation attempt counter).

In the current server, a process (e.g., a master process) can be user-interactive. For such a process, an `xterm` window is created and the user can communicate with the process via the window. In addition, the server maintains information about how many times the process has been re-spawned so far.

The three special groups for process management are:

- `Arg_group`. This group stores argument tuples. The `arg_rdp` operation accesses this group.
- `Eval_group`. This group stores identity tuples for those processes which will be created or re-spawned due to failure.
- `Proc_group`. This group stores identity tuples for those processes which are running.

At tuple space checkpoint, these tuple groups are saved to disk and therefore, information about processes survive failure. How this information is used will be explained in the rest of this subsection.

Process Creation

In the server, process creation consists of the following steps:

1. *Execution of the* `proc_eval` *operation.* A process executes a `proc_eval` operation inside a transaction. The execution of the operation inserts an argument tuple in the `arg_group` tuple group and an identity tuple in the `eval_group` tuple group. These tuple creation operations are executed in the context of the current transaction. Thus, other transactions cannot access them until the current transaction commits. If the transaction is aborted due to failure, then these tuples will be automatically destroyed.

2. *Dispatching.* The `Scheduler` object always waits for an identity tuple in the `eval_group` tuple group by using the `in` operation. Once an identity tuple is retrieved, the object increments the creation attempt counter in the tuple and inserts the tuple into the `proc_group` tuple group. The object executes this pair of operations as a single transaction. That is, the object starts a transaction, executes an `in` operation on `eval_group`, executes an `out` operation on `proc_eval` and commits the transaction. The transaction mechanism guarantees that the object always retrieves a committed identity tuple from `eval_group`. Once an identity tuple is retrieved, the `Scheduler` object schedules the process.

3. *Scheduling* The `Scheduler` object chooses a workstation having the smallest number of processes at that point (in fact, selects the corresponding `DaemonProcess` object). Then the `Scheduler` object asks the daemon process on the workstation to spawn a process as specified in the original `proc_eval` operation.

For a new process, the `Scheduler` object also creates a `ClientProcess` object. The `Scheduler` object finds this information by examining the creation attempt counter.

4. *Process Spawning.* On a process creation request, the daemon process uses the `execl` UNIX library function to spawn the process. At this point, the daemon process gives the process information about how to connect to the server using command line arguments.

Failure Detection and Recovery

The PLinda runtime system is designed to handle both processor failure (more specifically, fail-stop failure) due to hardware faults and fail-stop process failure due to software faults. The runtime system uses both failure notification from the underlying TCP/IP communication system and a custom timeout scheme to detect failure. For a failed process (processor failure is treated as failure of all the processes on the failed processor), the runtime system moves the identity tuple of the process from `proc_group` to `eval_group` so that its backup process (which will resume from the last continuation) can be re-spawned.

Failures due to software bugs are either *permanent* or *transient*. For example, a server process may crash because its request queue overflows due to an abrupt burst of requests from client processes. This kind of a software bug is considered as a transient fault because the fault (i.e., queue overflow) that caused the failure disappears when the server is restarted. In contrast, re-execution of processes that failed due to arithmetic exceptions results in permanent failure. Transient software faults are, by nature, difficult to detect and production quality software may suffer such faults. Therefore, it is desirable to recover processes that fail due to transient software faults and PLinda does so.

One implication is that the runtime system may, however, attempt to recover a process indefinitely often, if the process fails because of a permanent software fault. For this reason, the current prototype maintains the creation attempt counter in the identity tuple for each process. If a process fails more than a predefined number of times, then the runtime system concludes that the process has a permanent software fault, and aborts the entire execution.

To detect both process and processor failures, the server uses two schemes. First, the server uses failure notification from the underlying TCP/IP communication system to detect client process failure. In the current implementation, the server and clients communicate via TCP/IP. TCP/IP is a connection-oriented (or, point-to-point) communication protocol. If one end among a pair of processes which communicate via TCP/IP fails, the TCP/IP communication system detects it immediately and notifies the process at the other end.

In addition, the server uses a custom timeout scheme which is independent of TCP/IP. In this scheme, the server periodically checks if all the client processes are alive (which is called *failure checkup*). The server keeps track of interaction with each process and sends a ping message to those which have not communicated with the server since the

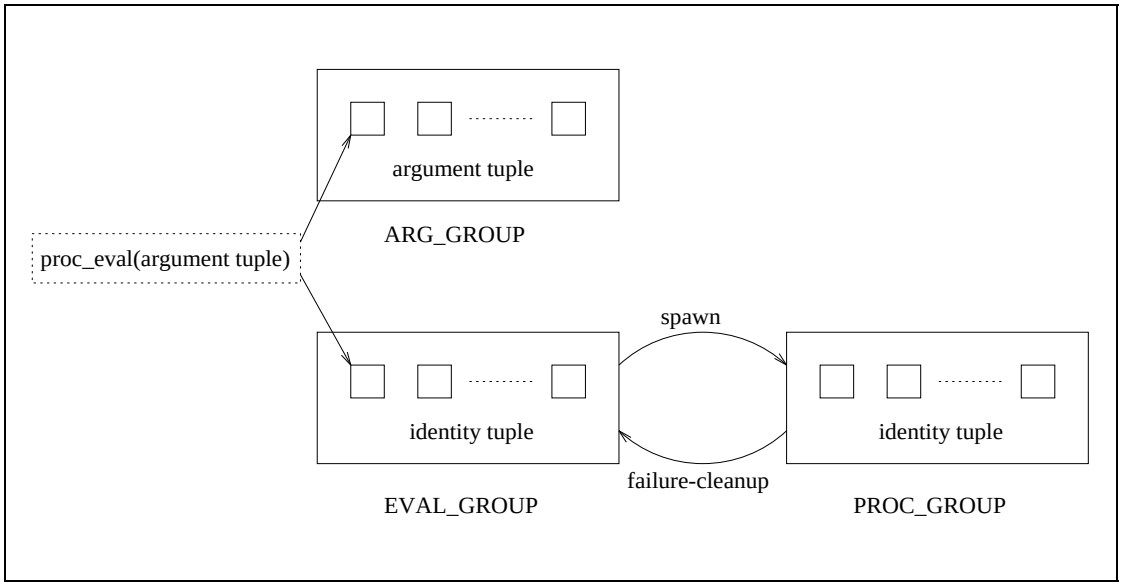

FIGURE 5.3: Special Tuple Groups for Process Management

last failure checkup point. Those processes which are sent a ping message but do not reply within a timeout are treated as failed.

On failure, a process is recovered as follows:

1. The corresponding `ClientProcess` object aborts the currently active transaction, if any.
2. The identity tuple for the failed process is moved from the `proc_group` to the `eval_group` tuple group. At this time, the counter of spawning operations is incremented. All these tuple moving operations are executed as a transaction.
3. The `ClientProcess` object records this failure.
4. The `Scheduler` object moves the identity tuple from the `eval_group` to `proc_group` tuple group and respawns the process.

Figure 5.3 shows how the three tuple groups are used for process management.

5.1.5 Tunable Execution of Continuation Committing

In PLinda, failure-resilient processes are designed to save continuations to tuple space at each commit — *continuation committing*. On failure, they recover the continuations saved at the last commit. Execution of continuation committing can be tuned to reduce runtime overhead for processes which have large continuations, but such tuning incurs more failure recovery overhead. The current runtime system supports three execution methods: commit-consistent, message logging/replay and coordinated checkpointing. The message logging/replay and coordinated checkpointing methods are aimed at applications in which processes have large continuations. In Chapter 3, we explained the design of these three execution methods in detail.

In the current design, the continuation committing operation is internally implemented as a tuple creation operation. However, the continuation committing operation differs from the regular tuple creation operation (i.e., `out`) as follows. First, the continuation committing operation generates a special tuple (which is called a *continuation tuple*) that is private to the process and its backup processes and accessed by only the `xrecover` operation. Second, there is only one continuation tuple for each process. Each time a continuation committing operation is executed, the previous continuation tuple is removed and a new one is inserted. Third, execution of continuation committing can be suppressed to achieve better runtime performance, at the cost of increased failure recovery overhead.

In the current server, the tunable execution mechanism for continuation committing is implemented as follows:

1. The server informs each process of the current execution method when the process first communicates with the server.

2. In the commit-consistent execution method, the `xcommit` operation always sends a continuation tuple to the server, together with a commit request.

   In the other two execution methods, the `xcommit` operation stores the tuple in the local address space and sends only the commit request. The server explicitly requests client processes to send a local copy of the last continuations when required (e.g., at checkpoint).

3. In the server, each `ClientProcess` object maintains a continuation tuple for the corresponding client process. The object saves a continuation tuple either in the virtual memory address space or on disk, depending on the size of the tuple. That is, execution of continuation committing requires disk access for a large continuation.

4. In the coordinated checkpointing method, the server asks all the processes to flush the last continuations when checkpointing tuple space.

5. In the message logging/replay method, each `ClientProcess` object maintains a log, called an *event history*, which contains records about tuple space access operations for the corresponding client process. Whenever the `ClientProcess` object receives a continuation, it removes all the records in the event history.

   To prevent an event history from growing too big, its size is limited. When the event history for a client process grows bigger than the limit, the server requests the client process to send its last continuation.

   In the message logging/replay method, the server asks all the client processes to flush the last continuations when checkpointing tuple space. Alternatively, the server could save event histories to disk at checkpoint, but currently, we use the former method because it is already available for the coordinated checkpointing method.

5.1.6 Tuple Space Checkpointing

In PLinda, checkpointing is used to make tuple space fault-tolerant. Tuple space is saved to disk periodically and on failure, restored to the last checkpointed state. The server is designed to ensure:

- A transaction-consistent snapshot is taken. A transaction-consistent snapshot reflects only and all of the updates made by the transactions committed at a specific point in time.
- The checkpointing operation itself is fault-tolerant.

Since tuple space is accessed transactionally, the tuples accessed by uncommitted transactions are locked. As mentioned above, there are three types of locks: `rd`, `in` and `out`. In PLinda, only committed tuples can be accessed by `in` or `rd`. Therefore, tuples which are rd-locked or in-locked are those which are generated by committed transactions but accessed by uncommitted transactions. By contrast, those which are out-locked are created by uncommitted transactions. Thus, a transaction-consistent snapshot consists of the tuples which are unlocked, rd-locked, or in-locked. The snapshot also includes the continuation tuple for each process. In both the message logging/replay and coordinated checkpointing methods, the server ensures that all the processes flush their continuations saved at the last commit before a snapshot is taken. Since continuation committing is a part of the commit operation, continuation tuples are always unlocked.

The server can fail while checkpointing tuple space. If a single copy of a snapshot is maintained and a new snapshot is overwritten on an old one, then server failure may corrupt the single copy. Therefore, the current server maintains two snapshots: the latest and the penultimate. Each snapshot has a header which is written to disk at the end of checkpointing; that is, the successful write of a header guarantees that the snapshot is correctly written to disk. The header contains information about system-assigned identifiers such as the last assigned process identifier. It also includes a checksum information both at the beginning and the end to be used for validity check on failure recovery.

On recovery from failure, the server first examines the checksum information in the headers of both snapshots to check whether they are corrupted, and compares the last update times to find out which one is the latest. Then, the server recovers the state from the latest valid snapshot. Once tuple space is recovered, the server (more specifically, the `Scheduler` object) moves all the identity tuples from the `proc_group` to `eval_group` tuple group. The server then resumes normal execution. All the processes whose identity tuples are now in the `eval_group` tuple group are automatically re-spawned.

5.2 Daemon Processes

The PLinda runtime system runs one daemon process on each workstation. Each daemon process monitors the idleness of its local host workstation and manages processes on the host. Process management includes starting processes and killing them when the owner resumes using the local host.

```
int interval = INTERVAL_IN_NONIDLENESS;

while(1) {
  /* check if the local host is idle */
  if(check_idleness()) {
  /* the local host is idle */
    if(interval == INTERVAL_IN_NONIDLENESS) {
      /* the host was busy */
      interval = INTERVAL_IN_IDLENESS;
      inform_server(IDLE);
    }

  } else {
  /* the local host is busy */
    if(interval == INTERVAL_IN_IDLENESS) {
      /* the host was idle */
      interval = INTERVAL_IN_NONIDLENESS;
      kill_clients();
      inform_server(BUSY);
    }
  }

  /* wait for a request from the server or becomes inactive */
  wait_for_server_request(interval);
}
```

FIGURE 5.4: Algorithm for the PLinda daemon process

Each daemon executes an infinite loop which is shown in Figure 5.4. First, it checks if the local host is idle. Currently, the idle times of the keyboard, the mouse, and the console are checked. The daemon informs the server of only changes to the idleness of the local host. When an idle machine becomes busy, the daemon process immediately kills all the client processes and informs the server of that.

A daemon waits for requests from the server when its host is idle, or becomes inactive when the host is busy. In both cases, the daemon uses the UNIX `select` system function which can time out.

Figure 5.4 shows the algorithm for the PLinda daemon process.

5.3 Administration Process

The administration process provides the user with tools for:

- Adding or deleting workstations.
- Monitoring the idleness of workstations.
- Monitoring execution behaviors of processes.
- Displaying execution traces of processes after the entire execution is finished.
- Controlling the runtime behaviors of the server.

The administration process is implemented in C++ and tcl/tk[1].

The user gives the administration process the list of all the host names when he or she starts the process. Currently, the administration process is designed to look for host names in a file named `plinda.hosts` from the working directory. However, the administration process does not invoke daemons on these machines until the user explicitly requests it. Once the administration process loads the file, the user can add machines listed in the file to parallel computation or delete them at any time. When the user adds a machine, the administration process first registers it in the server and then spawns a daemon process on the machine. However, the machine joins parallel computation only when the machine is idle. The daemon monitors the idleness of the machine and informs the server. If the user asks the administration process to delete a machine, then the administration process forwards the request to the server and the server sends a termination request to the daemon process on the machine. The daemon process kills all the processes running on the machine and terminates.

For each process, the administration process currently displays the execution status (e.g., running or blocked) and the last communication point (more specifically, the source code file name and the line number) where the process interacted with the server. To collect information about file names and line numbers, C macro variables `__FILE__` and `__LINE__` are used in the PLinda client library. At compile time, C or C++ compilers automatically translate these variables into the corresponding source file name and the line number of the location where these variables appear. The user can monitor the execution progress of processes using this information.

Using the administration process, the user can make client processes generate execution trace files at the end and display them. For each process, the execution trace file contains information about when and which PLinda operations the process executed, and how long each operation was blocked. Displaying execution trace files for processes graphically, the user can easily locate a performance bottleneck. In the current implementation of this facility, we assume that client processes and the administration process can access the same disk.

The user can control the runtime behaviors of the server using the administration process. For example, he or she can tune the frequency of tuple space checkpointing or the minimum idle time for idleness criteria. Also, he or she can choose an execution method for continuation committing.

In the current design, the administration process interacts with only the server. It forwards user commands to the server, obtains information from the server and display

[1] The administration process is developed with the help of Suren.

information graphically. To reduce communication with the server, the administration process could be designed to communicate with client and daemon processes directly. However, since the administration process obtains information from the server only intermittently (the frequency is a tuning parameter) and the server already maintains all the necessary information, we have chosen the current design decision.

5.4 Experiments

In this section, we present some performance results using the current PLinda prototype system. The objectives of our experiments are: (1) to measure the performance of tuple space operations and the execution and failure recovery mechanisms. (2) to test the effectiveness of these operations and mechanisms in real applications.

What we have done so far is preliminary as the PLinda group is continually trying to improve performance.

5.4.1 Performance of PLinda primitive operations

We examine the performance of four operations: `in`, `rd`, `out`, and `xcommit`. These operations were chosen because they represent four major operations in the PLinda system: destructive and non-destructive tuple retrieval, tuple creation and transaction commit. In the current implementation, all the other operations (e.g., `proc_eval` as tuple creation) are internally implemented as one or as a combination of these operations.

To measure the performance of `in`, `rd` and `out`, we used a simple process that creates or retrieves tuples in tuple space. The process works as follows:

1. Begin a transaction and start a timer.
2. Create (i.e., call `out`) or retrieve (i.e., call `in` or `rd`) 1000 tuples. The size of each tuple is 1000 bytes. The tuples contains only one character array field and the overhead due to matching is minimal.
3. Stop the timer and commit the transaction.

We ran the process and the server on different machines. For each operation, we have run this code several times. Their average execution times are shown in Table 5.1. There is a significant performance difference between `rd` and `in`. We are currently investigating what caused the difference in the implementation.

To measure the performance of `xcommit`, we used another simple process designed as follows:

1. Start a transaction.
2. Retrieve ten tuples of 100,000 bytes using `in`.
3. Create ten tuples of 100,000 bytes using `out`.
4. Start a timer, commit the transaction, and stop the timer.

Operation	Performance
`out`	7.26
`in`	9.56
`rd`	4.86
`xcommit`	3.10

TABLE 5.1: Performance of PLinda primitive operations. Time is given in milliseconds. The test process and the server were run on Sparc2 and Sparc5, respectively.

In order to collect reliable data, we have run this code many times. In the code, the transaction is designed to retrieve only ten tuples and add ten tuples since PLinda applications usually executes a small number of tuple space access operations in a transaction. However, those tuples have a rather large size (i.e., 100,000 bytes). The performance result is shown in Table 5.1. As the table shows, `xcommit` outperforms the three tuple space access operations.

5.4.2 Performance of the Three Execution Methods

In this subsection, we present the performance of the three execution methods: commit-consistent execution, message logging/replay and coordinated checkpointing. We first show their performance during normal execution. We then compare the failure recovery performance of the commit-consistent execution and message logging/replay methods.

In the current implementation, the server recovery from a checkpoint requires human intervention (explicit re-invocation of the server). That is, tuple space failure and process failure in the coordinated checkpointing method require human assistance. Therefore, we do not discuss the performance of server recovery.

In the experiments whose results are presented in this subsection, we examined the effects of continuation committing, message logging and the amount of work lost on failure with respect to runtime and failure recovery performance

We have designed a simple test program where we can easily change the size of a continuation (i.e., continuation committing overhead) and the number of tuple space access operations inside a transaction (i.e., message logging overhead). The program is based on the master/worker programming model and designed as follows:

- At the beginning of each execution, the user specifies four parameters:

  1. Number of tasks.
  2. Size of a continuation.
  3. Number of tuple space access operations per task.
  4. Size of data tuples.

  It is assumed that all the tasks are identical, all the workers have the same size of continuations, and all the data tuples have the same size.

- Master process.

  1. Starts the first transaction.
  2. Spawns eight workers and creates task tuples in tuple space.

3. Commits the transaction. (After this commit, these workers start to run.)
4. Starts a timer and starts the second transaction.
5. Collects results from workers.
6. Commits the transaction and stops the timer.

- Each worker repeatedly executes the following code:

1. Starts a transaction.
2. Grabs a task tuple and performs one million floating point multiplication operations.
3. Retrieve (`rd`) a certain number (input parameter) of data tuples.
4. Inserts a result tuple.
5. Commits the transaction with a continuation whose size is an input parameter.

We have experimented with six different settings shown in Table 5.2. The first three settings 1 through 3 are intended to compare the runtime performance of the three methods with respect to the size of a continuation. The last three settings 4 through 6 are designed to examine the runtime performance with respect to message logging overhead. The performance results are presented in Table **??**. In all executions, tuple space is checkpointed every 100 seconds.

In Setting 1, where processes have small continuations and the overhead due to message logging is minimal, the three methods showed similar performance results.

In Setting 2, where processes have medium size continuations but message logging overhead overhead is still minimal, the commit-consistent execution method became slower because of continuation committing overhead, but the message logging/replay and coordinated checkpointing methods have similar performance to that of Setting 1.

In Setting 3, where processes have large continuations, the commit-consistent execution method shows a lot worse performance than the other two methods. Since the coordinated checkpointing method does not incur any runtime overhead, it showed the best performance in this setting.

In Settings 4 and 5, where processes have large continuations but message logging overhead is small or modest, the message logging/replay and coordinated checkpointing methods have similar performance. However, in Setting 6 where processes have a large continuation and also suffer a lot of message logging overhead, the commit-consistent execution method outperformed the message logging execution method.

In all the settings, the coordinated checkpointing method had the best performance because it does not incur any overhead due to continuation committing (except checkpointing) or message logging.

We now compare the performance of the failure recovery of the commit-consistent execution and message logging/replay methods. Recall a failure here can mean that a client workstation becomes non-idle, causing PLinda to retreat so failure performance is a significant factor. To measure the failure recovery overhead as accurately as possible, We experimented with one process. This process performs one hundred transactions.

setting no.	# of tasks	continuation size	# of `rd`'s per task	tuple size
1	1,000	1,000	0	0
2	1,000	100,000	0	0
3	100	1,000,000	0	0
4	100	1,000,000	100	10
5	100	1,000,000	100	1,000
6	100	1,000,000	1,000	1,000

setting	commit-consistent	message logging	coordinated
1	197.83	200.87	196.50
2	250.40	204.58	199.26
3	201.76	33.81	36.59
4	207.80	48.79	47.44
5	216.43	58.66	45.46
6	361.78	389.53	213.75

TABLE 5.2: Performance results with the three execution methods. Time is given in seconds. For each execution, one master and eight worker processes were used. These workers ran on eight Sparc stations (two Sparc Classic's, four Sparc2's and two Sparc5's) and the master and the PLinda server on the same Sparc5 machine. Tuple space is checkpointed every 100 seconds.

Each transaction consists of one `in` operation, one million floating point multiplication operations, and one `out` operation. The size of the worker's continuation is set to 10,000 bytes. We ran the experiment many times and killed the worker up to three times for each execution. Tuple space is checkpointed every 100 seconds (60 - 80 transactions were executed in 100 seconds). When the 20th, 40th and 60th transactions were finished, we decided whether to kill the process. On failure, the runtime system immediately recovered the process.

The experimental result is presented in Figure 5.5. When there was no failure, the commit-consistent execution and message logging/replay methods showed almost the same performance. This is an expected result because the process has a small continuation and there is no overhead due to message logging.

However, the two methods showed different performance results when the process experienced failure. In the case of the commit-consistent execution method, the overhead due to process failure is negligible. For example, execution times when failure occurred once were about a couple of seconds slower than failure-free executions. For more failures, execution times increased only by a few seconds. This result is possible because on failure, the runtime system is immediately informed of the failure by the TCP/IP communication system and re-spawns another backup process. This implies that the PLinda failure recovery mechanism is viable for process migration for idle workstation utilization unless the idleness status of workstations changes too frequently. In our experience, workstations usually stay idle for a sufficiently long time (e.g, at least, a few minutes) once they become idle[44].

For the message logging/replay method, we observed a significant increase in execution time when we increased the number of worker process failures. The reason for such increase is that the worker process lost the previous committed work on failure and repeated it after recovery.

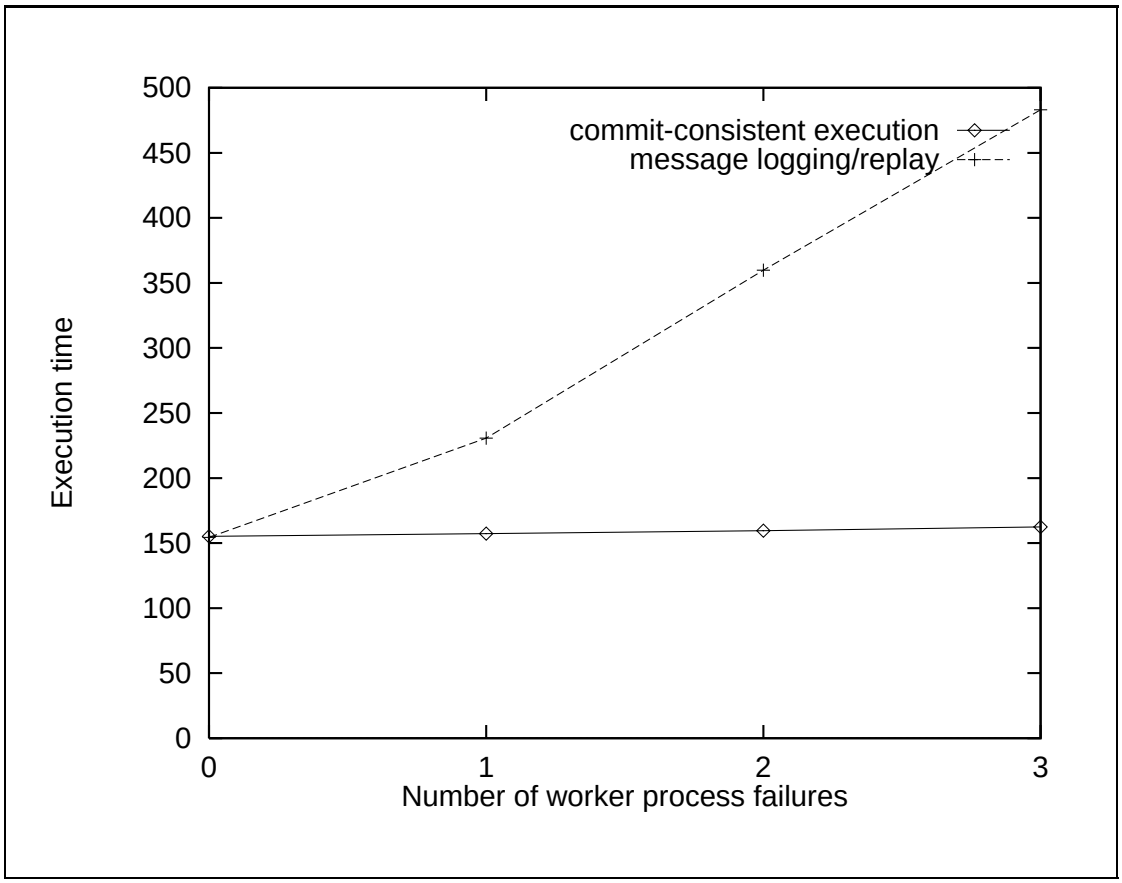

FIGURE 5.5: Performance of failure recovery. One process was used. The process ran on one Sparc5 and the server on another Sparc5. Tuple space is checkpointed every 100 seconds.

5.4.3 Biological Pattern Discovery

In this subsection, we present the results of our experiment to apply PLinda to a data mining application. We parallelized a sequential *biological pattern discovery* program[63] using PLinda. These types of data mining applications[1, 23, 35] are interesting because they are compute-intensive and they are usually coarse grain parallel problems.;

First, we explain the problem and the sequential approach. We then describe our parallel approach and show the performance results.

Biological pattern discovery is the problem of finding various patterns in protein databases where proteins are represented as sequences of letters (hereafter, just sequences)[31]. Interesting patterns are usually those which appear frequently in sequences in the database. Among the various algorithms in the field, we have parallelized the work presented in [63] where the problem is defined as follows:

- Database D. A set of sequences.
- Patterns. The pattern has the form *X1*X2*$\cdots$ where X1 and X2 denote subsequences (called *segments* in [63]) and * represents a variable length "dont-care" (VLDC). A VLDC can match zero or more letters. The length of a pattern is defined to be the number of the non-VLDC letters.
- Distance metric. The distance between two sequences is defined to be the minimum editing cost (with free substitution for VLDCs) needed to making them identical.
- Query. The user looks for a pattern of interest by giving the following form of a query:
  1. The minimum length of the pattern.
  2. The allowed distance.
  3. The minimum number of sequences in D which match the pattern within the allowed distance.
- Answer. The patterns which satisfy the requirements specified in the query.

The sequential algorithm consists of two phases: generating candidate patterns and evaluating these patterns with respect to the database.

Candidate patterns are formed by finding all promising segments (i.e., those which have a chance to appear in patterns of interest) and combining them. To facilitate finding segments, this algorithm uses a generalized suffix tree[36] (GST) constructed from a small sample of sequences (the size of the sample depends on the requirement for result accuracy). In GST, the edges are labeled with character strings and the concatenation of the edge labels on the path from the root to a leaf with an index i is a suffix of the $i-th$ string. Thus, each suffix of a string is represented by a leaf. Candidate patterns are all possible combinations of promising segments.

Once candidate patterns are obtained, the rest of the computation is to evaluate them with respect to the database. In order to reduce the amount of computation (the

complexity of the comparing operation of a pattern P and a sequence S is $O(|P| \times |S|)$), candidate patterns are first ranked from highest to lowest according to the occurrence numbers on the sample with respect to the allowed distance. Only the most likely candidate patterns are evaluated on the database.

In this algorithm, the evaluation of patterns with respect the database is most compute-intensive and easy to parallelize because each pattern can be evaluated independently. Our parallel approach[2] is as follows:

1. Use the master/worker programming model.
2. The master.
   (a) Manages the GST.
   (b) First transaction: creates worker processes and generates and distributes candidate patterns.
   (c) Second transaction: collects results from workers.
3. Worker processes.
   (a) Have a local copy of the database.
   (b) Evaluate candidate patterns on the database. The evaluation may cause the candidate to be discarded or may introduce new candidates Each evaluation is performed as a separate transaction.

We have experimented with this PLinda biological pattern discovery program on seven Sparc5's at the NYU Computer Science Department. The master and the server ran on the same Sparc5 station and each worker on a separate machine. We experimented with up to six workers. The database consists of 18 protein sequences and 4169 patterns were evaluated.

Some experimental results are shown in Figure 5.6. The performance of the PLinda program with one and two workers was slower than that of the sequential program. However, the PLinda program with four and six workers outperformed the sequential program. In the PLinda executions with only one worker, the message logging/replay and coordinated checkpointing methods outperformed the commit-consistent execution method because the overhead due to 4169 continuation committing operations in the tuple space server also affects the performance of the single worker directly. However, with two or more workers, all of the execution methods showed similar performance.

In Table 5.3, we present the performance of the PLinda biological pattern discovery program in the presence of failure. We did not remove workstations from computation, but killed a worker which the runtime system immediately recovered. The table shows that the failure recovery operation takes only a couple of seconds.

There are ongoing experiments with 100 Sparc5's and 50 Sparc2's at AT&T Bell Labs in Whippany. Some preliminary results are shown in Figure 5.7. The database consists of 47 protein sequences and 16544 patterns were evaluated.

[2]The PLinda program is implemented by Bin Li. He can be contacted by email (binli@cs.nyu.edu).

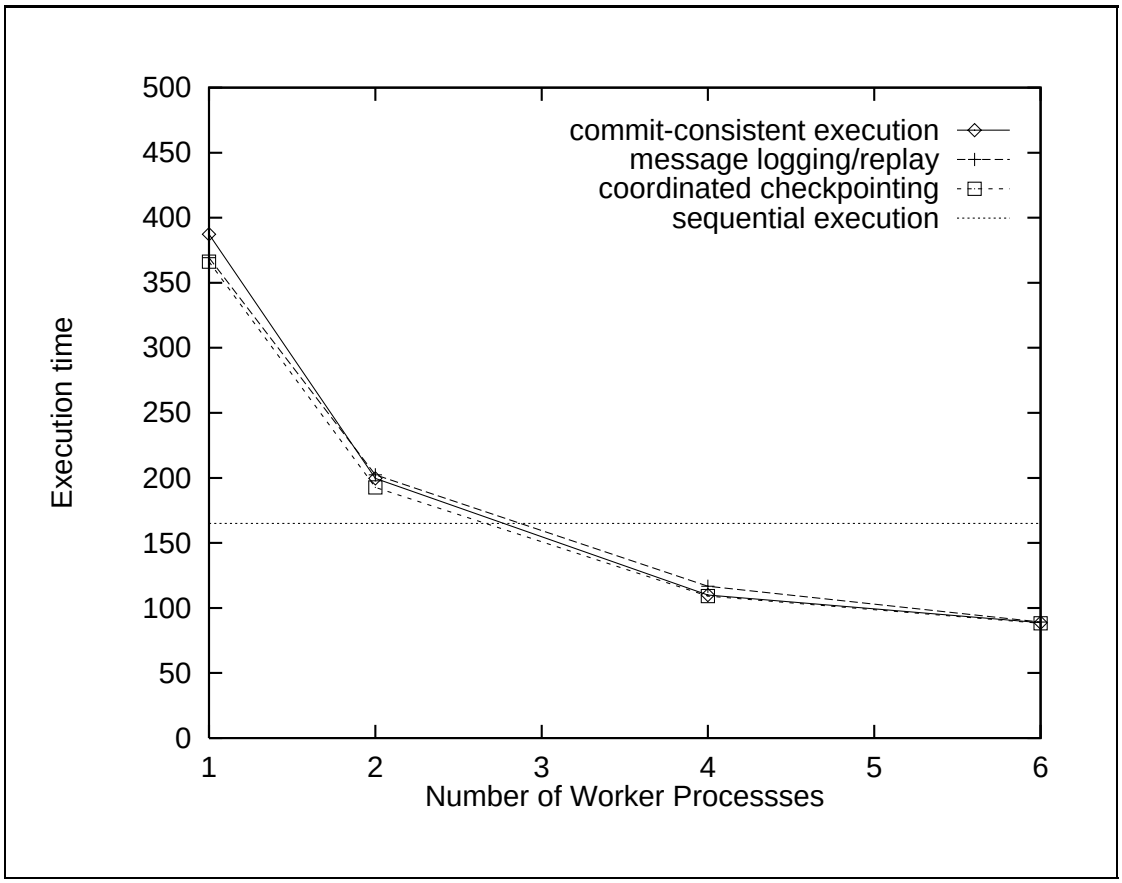

FIGURE 5.6: Performance results of the PLinda biological pattern discovery program with seven Sparc5's. Time is given in seconds. Tuple space is checkpointed every 100 seconds. The sequential code took 165 seconds for the same query and database.

# of failures	commit-consistent execution
0	89.313
1	89.716
2	90.819
3	92.118

TABLE 5.3: Failure recovery performance of the PLinda biological pattern discovery program with seven Sparc5's. Only the commit-consistent execution method was tested. Time is given in seconds. Six workers were used. The master and the server ran on the same workstation and each worker ran on a different workstation.

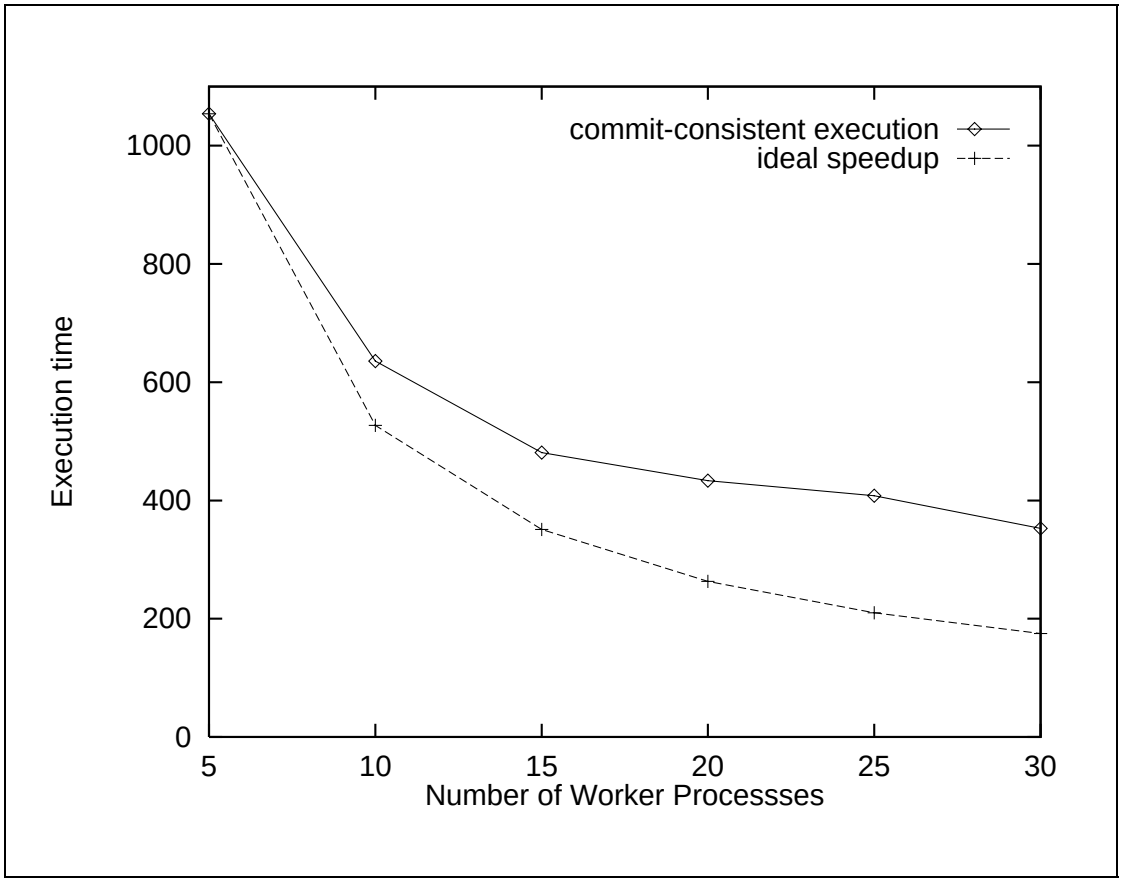

FIGURE 5.7: Performance results of the PLinda biological pattern discovery program on 30 Sparc5's at AT&T Bell Labs in Whippany. Tuple space was checkpointed every 100 seconds. Only the commit-consistent execution method was used. The sequential code took 2355 seconds for the same query and database.

5.4.4 Corporate Bond Index Statistics

In this subsection, we show the performance of the PLinda system on a compute-intensive financial application. The problem is to compute characteristics of the market-weighted averages of the collection of securities grouped by criteria (i.e., indices). Specifically, the Option-Adjusted-Spread (OAS) and the embedded option value of a bond are calculated and the market-weighted average of the resulting OAS and option values are computed[10].

We parallelized a sequential program implemented by Dmitri Krakovsky[3] at New York University. The program uses a binomial tree option pricing model[10] for the OAS and the option values of bonds. The program consists of three phases:

1. Input the observed and analyzed market data.
2. Calculate the value of the OAS for each bond in the index.
3. Compute the market-weighted average of the resulting OAS and option values.

See [10, 5] for details.

We parallelized the second phase because the computation in the second phase is most compute-intensive and each bond can be analyzed independently of other bonds.

Since this problem is easily parallelizable, parallelizing the sequential code using PLinda was straightforward and took only a few hours. The PLinda program is designed as follows:

1. Uses the master/worker programming model.
2. The master is responsible for the first and last phases.
3. The computation on each bond is treated as a task.

Figure 5.8 shows performance results obtained from experiments with seven Sparc5's. In this experiment, 50 security bonds were analyzed. The three execution methods showed similar performance results and there is no significant performance difference even when the fault tolerance mechanisms are disabled. The current implementation is designed to disable all the fault tolerance mechanisms. In this case, the client library completely ignores transaction operations and the tuple space server treats each tuple space operation as a separate transaction. In this way, their updates are immediately accessible.

Figure 5.9 shows the performance results obtained from experiments with 45 Sparc stations (three Sparc10's, three Sparc5's, 18 Sparc2's and 21 Sparc1's). In this experiment, 400 security bonds were analyzed. There were only negligible performance differences (below 10 seconds) between executions with the fault tolerance mechanisms enabled and those with the mechanisms disabled.

[3]He can be contacted by email (m-dk0027@cs.nyu.edu).

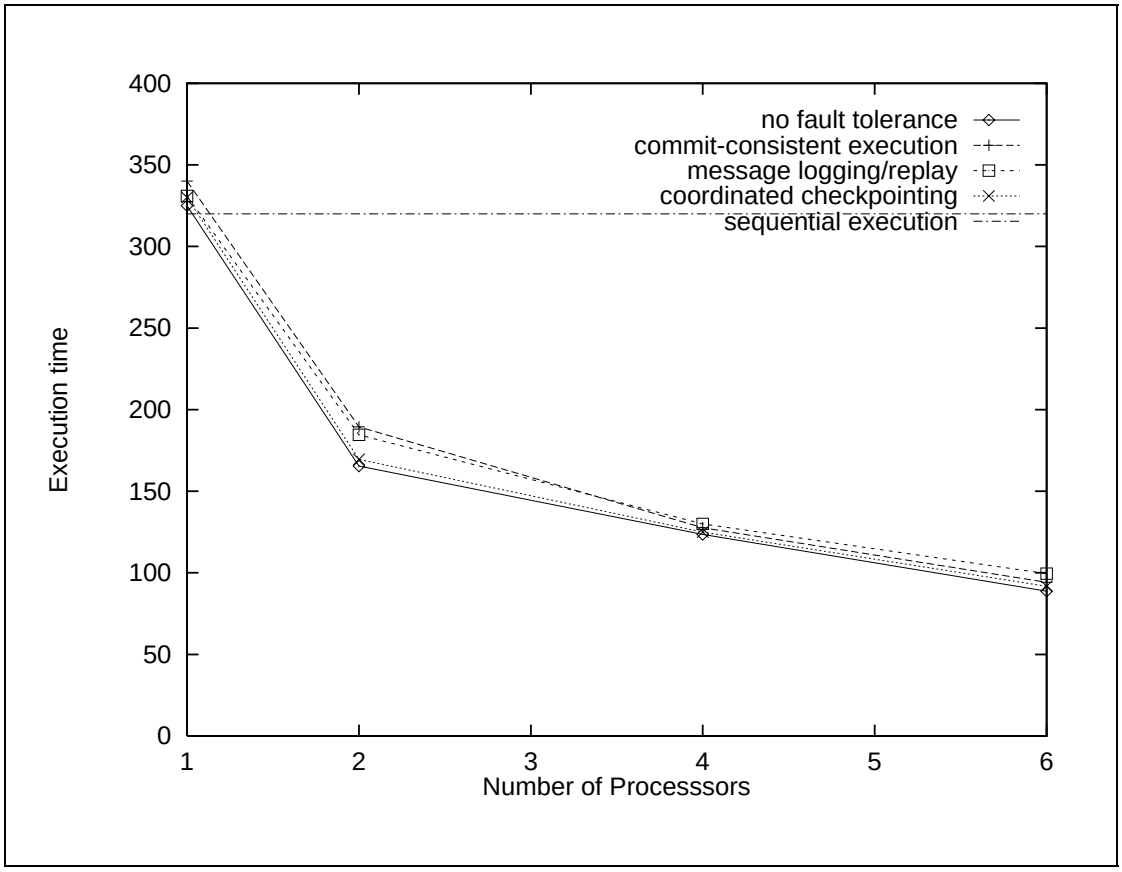

FIGURE 5.8: Performance of the PLinda Bond Index Statistics Computation Program with Seven Sparc5's. The master and the server ran on the same machine and each worker ran on a different machine. Tuple space is checkpointed every 100 seconds.

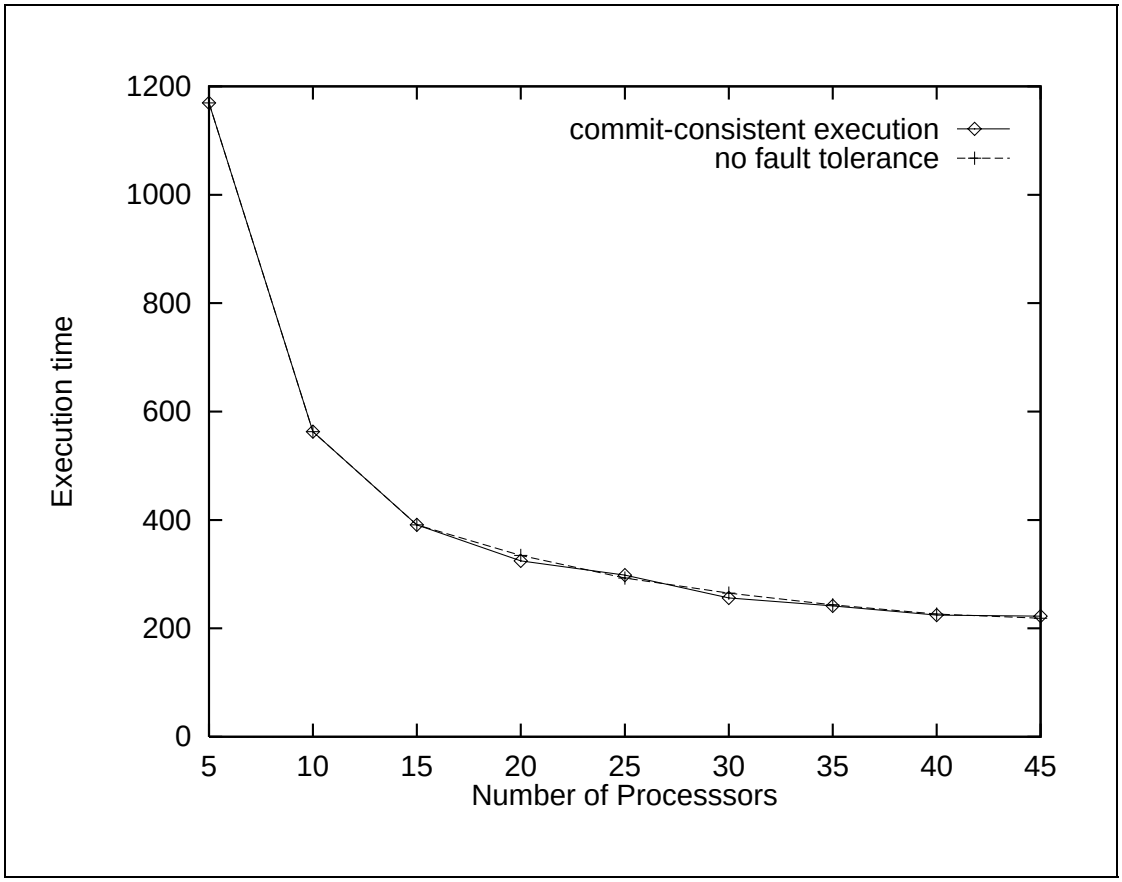

FIGURE 5.9: Performance of the PLinda Bond Index Statistics Computation Program with 45 Machines

5.5 Summary

In this chapter, we examined the implementation and performance of the current PLinda prototype system in an attempt to elucidate the performance tradeoffs offered by our three commitment mechanisms.

The implementation consists of four major components: the server program, the daemon program, the administration program, and the client library. In Section 5.1, we described the implementation of the PLinda server: the architecture, tuple space management, transactions, process management, and the execution methods. The tuple space management implementation ensures the efficiency of in-memory operations rather than that of the manipulation of persistent data on disk. In PLinda, transaction commits are designed to be lightweight in that they do not require disk access. However, committed updates may be lost upon tuple space failure. Checkpointing is used to make tuple space reliable. The transaction and checkpointing mechanisms are coordinated to maintain a consistent state in spite of tuple space failure.

In Section 5.2 and 5.3, we explained the implementation of the daemon and administration processes. The daemon process monitors the idleness of workstations and manages processes on local workstations. The administration process allows the user to observe and control the runtime behaviors of the PLinda system and applications.

In Section 5.4, we presented the performance of the PLinda prototype system. We first showed the performance of PLinda primitive operations and the three execution methods, and some results from experiments to apply PLinda to two real coarse grain parallel problems: biological pattern discovery and corporate bond index statistics computation. The comparative results suggest the possibility of a tuning tool that can switch from one commit method to another.

Experimental data show that the `xcommit` operation outperformed the tuple space access operations such as `in`, `rd` and `out`. In the commit-consistent execution method, the recovery of a failed process (more specifically, only the process failed but its processor is still operating, therefore the runtime system is immediately informed of the failure) took only a couple of seconds.

In Subsections 5.4.3 and 5.4.4, we described the performance of the PLinda system on two real applications. We presented performance results which we obtained from experiments with up to 45 machines.

Chapter 6

Conclusions and Future Work

Scientists and engineers increasingly use computers to solve problems that require an enormous amount of computation and therefore make parallel processing desirable or indispensable. Among those problems, many problems can be considered as coarse grain parallel or embarrassingly parallelizable; that is, they are easy to parallelize in large mutually independent chunks of computation. As a result of the recent proliferation of parallel software systems[15], we should therefore see those scientists benefit from parallel processing in solving coarse grain parallel problems. Unfortunately, the reality is not the case.

The goal of the PLinda project is to allow "poor" scientists (who have no easy access to parallel computers and can not afford buying them) to use widely available networked workstations as a parallel computing platform for solving their coarse grain parallel problems. In this dissertation, we have argued that two problems must be solved for this to be the case: (1) utilization of intermittently idle workstations for parallel computation (2) support for fault tolerance.

We have presented a Linda-variant parallel computing system, called Persistent Linda 2.0 (PLinda). PLinda is designed to offer: parallel processing on non-shared memory machines, fault tolerance, and the effective use of intermittently idle machines.

PLinda consists of a few extensions to Linda. Three major fault tolerance extensions are lightweight transactions, continuation committing, and checkpoint-protected tuple space. The programmer designs an application to be fault-tolerant using these mechanisms. The PLinda runtime system is designed to run processes on only idle machines and to treat non-idleness as failure. That is, when an owner comes back to his or her machine, the runtime system immediately "kills" processes on the machine and recovers them on idle machines by using the fault tolerance mechanisms.

In the PLinda project, our major research focus has been on how to incorporate transactions into the Linda model without entailing high runtime overhead or complicating the model. Unlike other fault tolerance work[3, 4] on Linda which attempts to use only some limited form of transactions and therefore complicates the Linda programming paradigm, we have sought to integrate the full functionality of transactions into Linda.

When transactions are used for parallel computation, an immediate concern is transaction commit overhead (i.e., disk writes implicit in transaction commits). In order to

reduce commit overhead, PLinda treats transactions and the reliability of tuple space orthogonally. Transactions do not write to disk on commit, but simply to the tuple space in volatile memory. This allows efficient transaction commits. In Section 5.4, we showed experimental results in which the performance of the PLinda commit operation, `xcommit`, is comparable to that of tuple space access operations such as `out`, `in` and `rd` in the current PLinda prototype system. (Actually, `xcommit` is less expensive.)

This lightweight transaction may lose committed data when tuple space fails (either the supporting computer or its volatile memory). PLinda recovers from tuple space failures by using checkpointing (whose frequency is a tuning parameter). This yields what we call checkpoint-protected tuple space. The transaction and checkpointing mechanisms are coordinated to guarantee the correctness of the final result. This is sufficient, since scientists who program long-running parallel computations don't care about intermediate results, only about the final results. Lost transactions, therefore, are acceptable if the final result is correct.

We have proposed a simple programming model for constructing a fault-tolerant application and extended transactions as a programming construct for the model. In the model, the programmer designs each process to be executed as a series of transactions. The transaction mechanism guarantees the atomic execution of each transaction; that is, the programmer can assume that processes logically fail only between consecutive transactions. In order to enable a process to recover its local state after failure, the programmer must specify a set of variables to be saved at each commit point that are sufficient to encode the process's continuation (we call these variables the encoded continuation). Each commit then saves all modified tuples as well as the encoded continuation. On recovery from failure, the process can resume from the last commit point by restoring the information.

Thus, this programming model allows the programmer to design lightweight custom fault tolerance for an application by taking advantage of characteristics of the application — copying only critical information about local state to tuple space in volatile memory at each commit. (In contrast, systems[22] that support transparent fault tolerance save entire process images to disk.) Experimental results from real coarse grain parallel applications show that the performance differences between executions with fault tolerance mechanisms activated and those with the mechanisms disabled are not significant (the difference is in fact negligible for the corporate bond index statistics computation application).

In addition to low runtime overhead, another important advantage of this scheme over transparent approaches[22] to fault tolerance is to support robust parallel computation on heterogeneous machines. Continuation committing and failure recovery are based only on process variables which are architecture-independent.

In this dissertation, we have further extended this PLinda fault tolerance scheme to support process migration for the utilization of idle workstations. The PLinda system is designed to treat non-idleness as failure; that is, process migration is implemented as forced failure on busy machines and recovery on idle machines.

This approach works well for two reasons. First, each process can perform continuation committing and rollback-recovery operations independently of other processes.

This allows a process to migrate without affecting other processes. Second, the PLinda fault tolerance mechanisms are lightweight. Continuation committing and failure recovery save only a few process variables to tuple space at commit and retrieve them on recovery. These operations are efficient because they do not require disk access nor re-execution of committed transactions. Experimental results show that recovering a failed process usually takes only about a couple of seconds.

In summary, PLinda addresses three challenging issues — parallel processing, fault tolerance, and utilization of idle workstations — by a few extensions to Linda which are surprisingly simple when the complexity of these issues is considered. The current PLinda prototype system has shown promising performance for various experiments including two real coarse grain parallel applications.

6.1 Future Work

Like most other ongoing research projects, PLinda has both various potentials and issues for further improvements and future work.

First, we will plan to assist the programmer with high-level programming toolkits or a Linda-PLinda compiler. In the current design, the correctness of a fault-tolerant application completely relies on the programmer. Although designing a fault-tolerant application in PLinda is usually straightforward for coarse grain parallel applications (the major target application class of PLinda), we think that such programming is potentially error-prone.

Our future work includes an automatic tuning method for deciding which commit method to use depending on the probability of workstation business and the size of checkpoints.

Second, a drawback of the current PLinda design is that recovery from server failure is costly; the tuple space server is restored to the checkpointed state on disk and all the client processes have to roll back. Expensive failure recovery for such failures is currently acceptable because the servers run on dedicated machines so server failures are real failures (as opposed to non-idleness) and therefore rare.

If we want to use our idle machine strategy for servers, one possible method is to replicate the server over idle machines using the replicated state machine paradigm[57]. In this method, client processes communicate with the server replicas via atomic ordered broadcast. When a machine where a server replica is running becomes busy or fails, the replica is destroyed. Then, a new server replica is spawned on an idle machine and recovers state from another replica.

The advantages of this approach are that the server can be migrated and also, client processes do not have to roll back due to server failure unless all the replicas fail at the same time. The disadvantage is that additional communication overhead due to atomic ordered broadcast must be paid for every tuple space access.

Third, an important research issue which still remains open in the PLinda project is *scalability*. That is, how can we extend PLinda to handle thousands of workstations spread over Internet for long-running coarse grain parallel applications? Certainly, long-running coarse grain parallel applications provide a great deal of potential for such

grand scale parallel computing because computation can be divided into numerous but independent tasks.

In the current PLinda system, the single server becomes a performance bottleneck even before one hundred of workstations are used. One intuitive approach to scalability is to distribute the server. This will increase the performance of the tuple space access operations such as `out`, `in` and `rd`. However, a distributed server will increase overhead due to transaction processing (e.g., two phase commit). Moreover, it will complicate checkpointing tuple space and replicating server processes.

If we aim at building a large scale parallel computing platform for long-running coarse grain parallel applications, another interesting approach to scalability is to use a server manager to control multiple independent PLinda servers of the current design. By "independent" we mean that there are no conflicting accesses to tuples at the different servers (e.g. the servers share read-only data and produce write-once tuples). To support long-running coarse grain parallel applications, such a server manager would spawn servers at various widely distributed sites, assigns tasks to the servers, and collect results when all the servers finishes their tasks. This would introduce a second level of server. This scheme is viable for long-running coarse grain parallel applications where computation can first be divided into independent and still-large subcomputations and then these subcomputations can further be divided into a large number of independent tasks.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.

[2] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, pages 47–56, September 1989.

[3] D. E. Bakken and R. D. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, pages 248–255, June 1991.

[4] D. E. Bakken and R. D. Schlichting. Supporting fault tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[5] Arash Baratloo, Partha Dasgupta, Zvi M. Kedem, and Dmitri Krakovsky. Calypso goes to wall street: A case study. In *Proceedings of the Third International Conference on Artificial Intelligence Applications on Wall Street*, June 1995.

[6] P.A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[7] K. P. Birman. The process group approach to reliable distributed computing. Technical report, Cornell University, July 1991.

[8] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. *ACM Operating Systems Review*, 21(5):123–138, November 1987.

[9] R. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, 1992.

[10] F. Black, E. Derman, and W. Toy. One-factor model of interest rates and its application to treasury bond options. *Financial Analysts Journal*, pages 33–39, January-February 90.

[11] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[12] N. Carriero. *Implementing Tuple Space Machines*. PhD thesis, Yale University, Department of Computer Science, 1987.

[13] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[14] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[15] D. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, 1993.

[16] David Cheriton. The V distributed system. *Communication of the ACM*, pages 314–333, March 1988.

[17] S. Chiba, K. Kato, and T. Masuda. Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 416–423, June 1992.

[18] Henry Clark and Bruce McMillin. DAWGS—a distributed computer server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14:175–186, 1992.

[19] R. F. Cmelik, N. H. Gehani, and W. D. Roome. Fault tolerant concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the Ninteenth International Symposium on Fault-Tolerant Computing*, pages 55–61, June 1988.

[20] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[21] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, pages 34–44, November 1991.

[22] Geert Deconinck, Johan Vounckx, Rudi Cuyvers, and Rudy Lauwereins. Survey of checkpointing and rollback techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, June 1993.

[23] V. Dhar and A. Tuzhilin. Abstract-driven pattern discovery in databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):926–938, December 1993.

[24] J. Dongarra, G. A. Geist, R. Mancheck, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, 1993.

[25] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[26] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[27] A. Z. Spector et al. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, 1985.

[28] R. Felderman, E. Schooler, and L. Klienrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2), February 1989.

[29] The MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[30] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works !* Morgan Kaufmann, 1994.

[31] K.A. Frenkel. The human genome project and informatics. *Communications of the ACM*, 34(11):41–51, November 1991.

[32] I. Gaines and T. Nash. Use of new computer technologies in elementary particle physics. *Ann. Rev. of Nucl. Part. Sci.*, 37, 1987.

[33] N. H. Gehani and W. D. Roome. Concurrent C. *Software–Practice and Experience*, 16(9):821–844, 86.

[34] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[35] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 547–559, August 1992.

[36] L. C. K. Hui. *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, volume 644, chapter Color Set Size Problem with Applications to String Matching, pages 230–243. Springer-Verlag, 1992.

[37] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[38] K. Jeong and D. Shasha. PLinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[39] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, 1989.

[40] T. A. Joseph and K. P. Birman. Reliable Broadcast Protocols. In Sape Mullender, editor, *Distributed Systems*, chapter 14., pages 293–318. ACM Press, 1989.

[41] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerant in parallel Orca programs. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1992.

[42] S. Kambhatla. Recovery with limited replay: Fault-tolerant processes in Linda. Technical Report CS/E 90-019, Oregon Graduate Institute, February 1990.

[43] S. Kambhatla. Replication issues for a distributed and highly available Linda tuple space. Master's thesis, Department of Computer Science, Oregon Graduate Institute, 1991.

[44] D. Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, Department of Computer Science, 1994.

[45] R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, 1987.

[46] J. Leichter. *Shared Tuple Memories: Shared Memories, Buses and LAN's–Linda Implementation Across the Spectrum of Connectivity*. PhD thesis, Yale University, Department of Computer Science, 1989.

[47] B. Liskov. Distributed programming in Argus. *Communication of the ACM*, 31(3):300–312, March 1988.

[48] M. Litzkow, M. Livny, and M.W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[49] T.W. Malone, R.E. Fikes, K.R. Grant, and M.T. Howard. *Enterprise: A Market-like Task Scheduler for Distributed Computing Environments*. Elsevier Science Publishers, 1988.

[50] Timothy G. Mattson, editor. *Parallel Computing in Computational Chemistry*. ACS Symposium Series 592. American Chemical Society, 1995.

[51] Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, University of Arizona, Department of Computer Science, 1992.

[52] M.W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Performance '87*, pages 529–544. Elsevier Science Publishers B.V., 1988.

[53] D.A. Nichols. Using idle workstations in a shared computing environment. *ACM Operating Systems Review*, 21(5), 1987.

[54] James Steven Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, June 1993.

[55] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, August 1983.

[56] Richard D. Schlichting and Vicraj T. Thomas. FT-SR: A programming language for constructing fault-tolerant distributed systems. Technical Report TR 92-31, University of Arizona, Department of Computer Science, 1992.

[57] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[58] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison Wesley, 1995.

[59] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[60] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[61] Robbert van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. Cornell University, 1992.

[62] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[63] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 115–125, May 1994.

[64] J. T. L. Wang, T. G. Marr, D. Shasha, B. A. Shapiro, and G.-W. Chirn. Discovering active motifs in sets of related protein sequences and using them for classification. *Nucleic Acids Research*, 22(14):2769–2775, August 1994.

[65] Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, University of Toronto, April 1993.

[66] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implemenation of Linda. In *Proceedings of the Ninteenth International Symposium on Fault-Tolerant Computing*, pages 199–206, June 1989.

[67] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-257, University of Toronto, April 1992.