PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda*

Karpjoo Jeong
(jeong@cs.nyu.edu)

Dennis Shasha
(shasha@cs.nyu.edu)

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York, NY 10012

Abstract

Robust parallel computation in Linda requires both tuple space and processes to be resilient to failure. In this paper, we present PLinda 2.0, set of extensions to Linda to support robust parallel computation on loosely coupled processors communicating over a network. The principal extensions of PLinda 2.0 to Linda are transaction mechanisms for reliable tuple space and process-private logging mechanisms for resilient processes. The transaction mechanisms support two kinds of tuple space: stable tuple space always guaranteed to reflect state as of last committed transaction, and unstable tuple space protected by a transaction-consistent checkpoint. The process-private logging mechanisms are provided as tools for a process checkpointing scheme. These mechanisms allow the customization of checkpointing and recovery operations in each process to achieve low runtime overhead.

1 Introduction

One of the issues that distributed programming systems must address is fault tolerance[4]. On loosely coupled workstations communicating over a network, the autonomous nature of each workstation provides a great potential for fault tolerance. Even though some of the workstations fail, others can still function correctly. Hence, tasks running on a failed processor can be moved to live processors to continue execution.

*This work was supported, in part, by the National Science Foundation under Grants IRI-9224601 and IRI-9224602 and by the Office of Naval Research under Grants and N00014-92-J-1719.

The need for fault tolerance also arises when networked workstations are used as a computing platform for long-running parallel applications. The probability of a failure, naturally, grows high, as execution time or the number of workstations increases. The crash of a single workstation may cause the entire execution to fail.

Currently, there are many parallel/distributed programming systems [8] available. Among the most popular is Linda [5,6,7]. Unfortunately, most such systems including Linda have not yet adequately addressed fault tolerance.

In this paper, we present PLinda 2.0 (hereafter, just PLinda), an approach to fault tolerant Linda. PLinda assumes the "fail-stop" processor model (i.e., processors fail by stopping). The two principal extensions to Linda in PLinda are transaction mechanisms and process-private logging mechanisms.

In order to support arbitrary Linda programs, fault tolerant Linda requires tuple space, the shared memory of Linda, and processes to be resilient to failure. PLinda uses transactions, because they have proven to be a simple but effective abstraction for controlling concurrent access to shared data in a reliable manner despite failure [10]. For the same reason, most other fault tolerant work on Linda also supports some similar mechanism, but only with limited functionality, compared to transactions in PLinda.

While transactions are effective mechanisms for reliable tuple space, they are not designed to make processes resilient to failure. In this paper, we will show how transactions can be extended to support process resiliency without high runtime overhead. Our basic approach is to meld a process checkpointing scheme into the transaction mechanism. Based on this melded mechanism, we will explain how to make arbitrary

Linda programs fault tolerant in PLinda.

PLinda uses transactions for parallel computation. An immediate concern is the overhead due to the disk writes implicit in transaction commits. PLinda reduces this runtime overhead by using checkpoint-protected unstable tuple space which does not require each transaction to write to disk, and non-blocking transaction commits which allow processes to go on to other transactions while their commits occur asynchronously.

In this paper, we also want to persuade you that transactions (i) are about as cheap as other mechanisms for fault tolerance on Linda, (ii) are simple for programmers to use, and (iii) are the best approach for certain fault tolerant applications. That is, we will argue that transactions as embodied in PLinda are never worse than other fault tolerant mechanisms and are sometimes much better. We expect to be able to demonstrate our results at the symposium. Readers can obtain them from us by email.

The rest of this paper is organized as follows: Section 2 reviews Linda. In Sections 3 and 4, we discuss transaction mechanisms for reliable tuple space. In Section 5, we talk about how to build resilient processes in PLinda. In Section 6, we present a correctness proof for our scheme. Section 7 presents an optimization strategy for process-private logging. Section 8 compares PLinda with related work. Finally, we conclude this paper in Section 9.

2 Brief review of Linda

Linda is a *shared memory*-based parallel computation model. In Linda, processes in an application cooperate by communicating through the shared memory called *tuple space*.

Each tuple in tuple space contains a sequence of typed data elements where the data types are basic types such as integers, floats, characters, and arrays of these. In Linda, processes access tuple space by using the following six 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.

```
/* worker process */
int worker() {
  while(1) {
    xstart;
      /* grab a task */
      in("task", ?task_description);

      execute_task(task_description,
                   &result);

      /* produce a result */
      out("result", result);
    xcommit;
  } /* while */
}
```

Figure 1: A Fault Tolerant Worker 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,

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

   The first and last fields are values; the middle two fields are place-holders. 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.

4. **inp** and **rdp**. They are *non-blocking* versions of `in` and `rd`, respectively. That is, they return immediately if no tuple matches their argument.

For a more detailed description of Linda, refer to papers about Linda[5,6,7].

3 Transactions

In this section, we will first discuss transactions that recover tuple space to a consistent state after failure, and then the types of processes that transactions can make resilient to failure. In the next section, we will talk about how to make tuple space survive failure.

Transactions guarantee atomic execution of a sequence of operations despite failures. That is, all of the operations take effect in the shared storage (in the "commit" case) or none of them do (in the "abort" case). In addition, transactions ensure isolation: all committed transactions appear to execute serially with respect to one another. These two properties ensure that failures logically appear only between transactions.

When designing a parallel/distributed application to be fault tolerant, it is especially critical to control the effects of failures upon interactions between processes and shared data. In Linda, tuple space is the only vehicle for communication between processes, and storage of shared data. That's why most fault tolerant work in Linda provides some sort of transaction mechanism to execute a series of tuple space operations atomically despite failures. PLinda supports the most powerful and general such mechanism of any effort we know.

In PLinda, transactions have the following properties:

- **Atomicity** Shared memory (i.e., tuple space) reflects all of the updates of a transaction if the transaction commits; but reflects none of these updates if the transaction aborts. A transaction ends by either committing or aborting — there are no other possibilities.
- **Isolation** A transaction's updates are accessible only after the transaction has committed.
- **Automatic Clean-up** Uncommitted transactions in failed processes are automatically aborted.

PLinda requires every tuple space operation to be performed inside a transaction. 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.

In addition to tuple space operations, process creation is also treated as an update to tuple space. All the process creation operations (i.e., `eval` calls) in a transaction will take effect only after the transaction has committed; none of them will if the transaction aborts due to failure.

Now, let's look at examples to discuss what types of processes can be made fault tolerant using transactions and what types can not. 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 at the beginning. During execution, the workers repeatedly grab tasks from a bag of tasks and carry them out. The master collects results from the workers.

```
/* master process */
int master() {
  xstart;
    for(i=0; i<NUM_WORKERS; ++i) {
      eval("worker", worker());
    }
    for(i=0; i<NUM_TASKS; ++i) {
      out("task", task[i]);
    }
  xcommit;

  xstart;
    for(i=0; i<NUM_WORKERS; ++i) {
      in("result", ?res);
    }
  xcommit;
}
```

Figure 2: A Master Processes

Figure 1 gives a skeleton code of a worker process using a transaction in PLinda. In the example code, the transaction guarantees that execution of a task be atomic in spite of failures. For example, suppose that a worker process of the code 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. Therefore, another worker that is still functioning will be able to take and re-execute the task later. In the set-up of this example, a worker process can not have private state; otherwise, if the process fails, then it is impossible to recover the state later.

Figure 2 shows a skeleton code for a master process. The master process creates worker processes and task tuples in a transaction. Then, it collects results in another transaction. If the master process fails during execution of the first transaction, then there will be no worker process and task tuples; that is, it will appear that the transaction has not happened. If it fails in the second transaction, the failure will not affect the worker processes and tasks tuples, but only the second transaction will be aborted . Upon recovery from failure, it is impossible to find out where the master process has failed; that is, it is impossible to recover the master process. Transactions recovers a consistent state of tuple space after failure, but do not recover local states of failed processes. This means that, only with transactions, it is not possible to make stateful processes (i.e., those which executes multiple inter-dependent transactions) resilient to failure.

4 Reliable tuple space

In Linda, processes communicate only via tuple space and maintain shared data in it. Therefore, "failure-survival" of tuple space is the most important functional requirement for fault tolerance in Linda. Such reliability 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). PLinda takes the latter approach, because disks seem better suited to our separate goal of persistence.

The PLinda transaction mechanisms support two kinds of tuple space:

- **Stable Tuple Space**. Before a transaction commits, all the updates made by the transaction are replicated on disk.

  If tuple space fails and then recovers, it will be restored to the "transaction-consistent" state. The "transaction-consistent" state is one that reflects the updates made by all and only the transactions that committed before the failure. Thus, updates made by committed transactions will survive failures of tuple space.

- **Checkpoint-protected Tuple Space**. The entire tuple space is periodically written to disk (which is called a checkpoint). For checkpointing, the runtime system first prevents new transactions from starting and waits for all the active transactions to commit. Then, it saves the entire tuple space on disk.

  If tuple space fails and recovers, then it will be restored to the last checkpointed state. 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.

In PLinda, the difference between these two methods is transparent to programmers. When a user runs a PLinda application, he or she can choose one of them, depending on the application.

The advantage of checkpointing over transactional persistence is that there is no need to write to disk for each transaction. Its shortcoming is quiescing the runtime system, though there are implementation techniques to get around this. (Basically, one keeps track of data as of the time of the checkpoint and writes out that version even if a later version has committed.)

5 Resilient processes

A resilient process is one that can continue and complete execution, even though it suffers failure during execution. There are two approaches to resilient processes: process replication with atomic broadcast and process checkpointing. PLinda takes the latter approach for resilient processes because it is more consistent with other mechanisms already supported in PLinda.

In addition to process resiliency, the checkpointing scheme in PLinda also supports non-blocking transaction commits; processes can start other transactions before their commits are completed.

5.1 Process-private logging mechanisms

PLinda's approach differs from transparent schemes [14,15] in two ways. First, PLinda does not provide a transparent service but process-private logging mechanisms as tools. The mechanisms are designed to customize checkpointing operations for each process; that is, processes are allowed to save only minimal information required for recovery at checkpoint. That makes the mechanisms cheaper than transparent approaches which, by contrast, generally save the entire process image in stable storage. However, they require explicit checkpointing operations.

Second, process-private logging mechanisms are integrated into transaction mechanisms. Transaction mechanisms enable processes to restore a globally-consistent state at rollback recovery, without any special consistency protocols.

Here, we will first explain process-private logging mechanisms.

When a transaction commits, the runtime system assigns a system-defined identifier to each process to be created as a result of the commit operation, and saves information on stable storage which is required to create and manage the process. This will allow the process to be recovered even when the entire system fails (this can happen due to power failure).

Once a process is created by a committed transaction, it will be automatically recreated and restarted by the runtime system when the process fails. Logging operations of the recreated process will access the log that the failed one wrote to.

Each process can check if it is restarted due to failure or transaction abort, by calling the function `restart` supported by the runtime system. `Restart` returns true if it is called by the process that is restarted due to failure or transaction abort; otherwise, it returns false. If `restart` returns true, then the process needs to recover local state.

If tuple space fails, all the running processes are logically considered to fail and restart execution from the beginning. At this time, the function `restart` will return true.

PLinda provides two special operators, `log_inout` and `log_rdp`, to allow a process to save local state (i.e., contents of local variables) in stable storage during execution (which is called *local-checkpointing* or *l-checkpointing* for short) and restore it later when the process restarts after failure (which is called *rollback recovery*). They are designed to be similar to `out` and `rdp` in their syntax and use; that is, they logically generate and read tuples (we call them log tuples), respectively. However, they differ from `out` and `rdp` in several ways.

First, when a process performs logging operations, it will access log tuples associated with itself even if it restarts after failure. Thus, `log_inout` and `log_rdp` depend on the identity of the calling process.

Second, `log_inout` is designed to do an update in place. Every log tuple is required to have the first field as a unique key. `Log_inout` removes the existing tuple with the same key, if there is one, and replaces it with its argument. Otherwise the operation does an insertion. This simplifies logging contents of variables.

Third, like tuple space operations, all the `log_inout` operations in a transaction will take effect if and only if the transaction commits. If the transaction aborts due to failure, then they will take no effect. Thus, process-proviate logging mechanisms also support atomicity as transaction mechanisms.

Finally, tuples referenced by `log_inout` and `log_rdp` will not go into tuple space but rather into a separate log space which is treated as a sequential medium. However, the log space is always guaranteed to be consistent with tuple space with respect to transaction commits. For example, if tuple space is restored to the last checkpointed state and as a result, loses the updates made by some committed transactions, then the log space must also discard information checkpointed at those transactions.

Now, let's discuss how process-private logging mechanisms support non-blocking transaction commits. Apparently, non-blocking commits do not cause any inconsistency between processes as long as tuple space does not fail. If tuple space fails and then recovers, that makes the states of tuple space and processes inconsistent because uncommitted transactions are aborted or the updates made by committed transactions are lost. In that case, process-private logging mechanisms, however, force all the running processes to recover local state from the log space which is always guaranteed to be consistent with tuple space. Therefore, processes recover a globally consistent state.

5.2 Designing resilient processes

As explained before, process-private logging mechanisms are designed to incur low runtime overhead, but require explicit checkpointing operations.

Here we present a simple programming method of designing resilient processes using process-private logging mechanisms in PLinda. At this point, the method requires some programming effort but we think that we will be able to automate most of it. That will be our future work.

This programming method utilizes the following two characteristics of the PLinda model. First, processes execute a sequence of transactions with respect to tuple space (i.e., shared data or process interaction), because PLinda requires every tuple space operation to be performed inside a transaction. Second, processes conceptually fail only between transactions.

The main idea of the method is that a process l-checkpoints its *continuation* at the end of each transaction, and resumes execution from the continuation l-checkpointed at the last committed transaction if it fails. Continuation consists of private state at that point (i.e., contents of process variables) and the identity of the next transaction to be executed. Thus, this method assumes that a process can assign a logical identity to each transaction, and restart execution at a particular transaction when it fails. This assumption might look like a serious limitation, but it turns out to be easy in most cases as examples in this section will show. This is reminiscent of the "mini-batch" technique described in [10].

In the remainder of this section, we will present examples to illustrate how the method can be used in applications. Since it is straightforward to save and restore contents of local variables, we will focus on how to assign a logical identity to each transaction and how to resume execution using such an identity.

First, let's revisit the code for a master process shown in Figure 2. Figure 3 shows a fault tolerant version of the master process. In the example, since transactions execute different code, the master process can explicitly assign a logical identity to each transaction — `TRANS_ONE` and `TRANS_TWO`. Before beginning the whole computation, the process calls `restart` to check if it is restarted due to failure or transaction abort. If so, then it restores the information about which transaction to execute. Otherwise it starts with the first transaction. This way, the master process will not repeat committed transactions when it restarts. The master process l-checkpoints the identity of the

```
int master() {
  int next_trans;
  next_trans = TRANS_ONE;
  if(restart()) {
    /* if this tuple does not exist,
       then next_trans is not changed. */
    log_rdp("transaction", ?next_trans);
  }

  if(next_trans == TRANS_ONE) {
    xstart; /* TRANS_ONE. */
      for(i=0; i<NUM_WORKERS; ++i)
          eval("worker", worker());

      for(i=0; i<NUM_TASKS; ++i)
          out("task", task[i]);

      /* l-checkpoint */
      next_trans = TRANS_TWO;
      log_inout("transaction", next_trans);
    xcommit;
  }

  if(next_trans == TRANS_TWO) {
    xstart; /* TRANS_TWO. */
      for(i=0; i<NUM_WORKERS; ++i)
          in("worker", ?res);

      /* l-checkpoint */
      next_trans = NO_TRANS;
      log_inout("transaction", next_trans);
    xcommit;
  }
}
```

Figure 3: A Fault Tolerant Master Process

next transaction at the end of each transaction which is the continuation of the master process. If the process fails, then it will restore its continuation at the beginning and resume by executing the transaction to be recorded in the continuation. In the example, the process uses `if` statements to restart at a particular transaction when it fails.

When a process runs transactions in an iteration loop that execute the same code, we can not assign a logical identity to transactions explicitly as shown in Figure 3. Instead, we use the index(es) of the loop as the identity of each transaction. Figure 4 shows

```
real_main() {
  int index;
  struct stateinfo local_state;

  index = LOWER_BOUND;
  if(restart()) {
    log_rdp("index" ?index, ?local_state);
  }
  for(; index <= UPPER_BOUND;) {
    xstart;
      /* update tuple space,
         and the private state. */
      compute(index, local_state);

      /* l-checkpoint */
      log_inout("index", ++index, local_state);
    xcommit;
  }
}
```

Figure 4: Loop Index as a Transaction Identity

an example. In the example, the continuation consists of the index of the next iteration and the contents of local variable `local_state`.

6 Correctness of our scheme

In this paper, we have presented a programming method for designing a fault tolerant application in PLinda. The method is based on transactions and process-private logging operations. In this section, we will prove the correctness of the method.

Our proof strategy is as follows. First of all, we assume that processes have deterministic behaviors depending only on input data; that is, they neither interact with the user nor access updatable external data nor interact with a clock. (These are reasonable assumptions for most Linda applications.) Then, we formalize the properties of a fault tolerant PLinda program that is built using the method. Based on the properties, we then prove that for a completed execution of a fault tolerant PLinda program P_L, there is some possible execution of the Linda program L, where L is obtained from P_L only by removing transaction and logging statements.[1]

Definition 1 *A PLinda process is defined to be* Failure Tolerance-structured *(for short,* FT-structured*) if*

[1] In this proof, we assume a single threaded transaction, that is, one that is executed by only a single process. However, we think that the proof will be similar, when multi-threaded transactions are considered.

it satisfies the following properties:

1. *It executes a sequence of transactions, $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$. Without loss of generality, we assume all the process variables are uninitialized and the process uses T_1 only to initialize the process variables.*
2. *Its initial state is s_0 where all the process variables are uninitialized. The execution of transaction T_i changes the state from s_{i-1} to s_i.*
3. *At the end of transaction T_i, it logs all process variables that are still in scope plus enough information to indicate that it has completed transaction T_i. That is, the process logs its continuation at the point s_i.*
4. *If it fails while executing T_{i+1}, it returns to s_i and resumes execution. This will cause it to restart T_{i+1}.*

In Section 5.1 and 5.2, we explained how to design a process to be FT-structured using transactions and logging statements.

Definition 2 *An entire PLinda program is defined to be FT-structured if all the processes are FT-structured.*

Definition 3 *Two executions are said to be equivalent if the same operations execute in the same order and each read operation returns the same value and each write operation writes the same value, where reads and writes include tuple space operations.*

Lemma 1 *Suppose P_L is an FT-structured PLinda program. Then, for an execution of P_L, there is some possible execution of L that is equivalent, provided that there is no failure and the program completes.*

Proof. If there is no failure or deadlocks, transactions only delay the issuing of `out` and `eval` operations until the commit point. Logging operations only save process variables on the log, but have no further influence on the computation. Therefore, the execution of P_L is a possible execution of L.

Lemma 2 *For an FT-structured PLinda process P, an execution of P that experiences failures is equivalent to a failure-free execution of P.*

Proof. We prove this by induction.

Case 1. Suppose that P fails while executing the first transaction T_1. Then, the transactional mechanisms will guarantee that all the updates to tuple space made by T_1 be undone and the other processes should not yet have accessed them. (This makes use of the fact that reads require no locks but read only unlocked, and therefore committed, tuples.) Thus, it will appear to the other processes that the transaction has never happened. The failure will cause the local state of the process to be lost, so when P recovers, it will do so in its initial state s_0. Thus a failure of the first transaction will be equivalent to a delay in the execution of that transaction.

Case 2. Suppose that P fails while executing T_{k+1}, where k+1 $\leq$ n. As for a failure of T_1, the transactional mechanisms will guarantee that it appear to the other processes that T_{k+1} has never happened. When P recovers, it will restore state s_k, and execute T_{k+1} from the beginning. This is the same situation as if T_{k+1} had simply been delayed.

Based on case 1 and 2, any failures of P will not affect the execution of itself or those of the other processes. Therefore, an execution of P that experiences failures is equivalent to a failure free execution of P.

Theorem 3 *Suppose P_L is an FT-structured PLinda program. Then, for an execution of P_L, there is some possible execution of L that is equivalent, provided that the program completes.*

Proof. By definition, all the processes to be created by P_L are FT-structured. In Lemma 2, we proved that execution of each process is equivalent to a failure-free execution, even if the process suffers failures.

Since all the processes show the behavior of a failure-free execution despite failures, the execution of P_L will also show the behavior of a failure-free execution despite failures.

Based on these and according to Lemma 1, for the execution of P_L, there is some possible execution of L that is equivalent, despite failures.

7 An Optimization strategy for process-private logging

One of the advantages of explicit logging operators instead of using the tuple space as log is to provide good targets for internal runtime optimization.

Since recovering from an l-checkpointed state occurs only rarely, it may be worthwhile to incur more recovery time in order to gain runtime efficiency. In particular, the strategy we propose is effective in applications where processes have a lot of local data but exchange a small amount of data during execution (a common situation in scientific applications).

Here, we present a runtime optimization strategy based on *replay* for the l-checkpointing operations. This strategy is similar to the work by Strom and Yemini[15]. However, we incorporate their techniques into our transactional model. This strategy is designed for the transactional persistence technique where updates are written to disk at commit.

The strategy is as follows:

1. A process performs an l-checkpointing operation on its private state only when the runtime system explicitly requests it. (How frequently that occurs is a tuning parameter.) That is, even though the semantics of the logging 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 maintains the history of the committed input tuple space operations (i.e., the in and rd operations) executed by the process since the last l-checkpoint. Thus, for each input tuple retrieval operation (**in**, **rd inp**, or **rdp**) 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 logging operations in the applications we know about.

3. The tuple space server guarantees that the history of the committed input tuple space operations survive failure; that is, when a transaction commits, the history of its input tuple space operations is also stored on stable storage.

4. Once a process l-checkpoints its private state, the history discards all the process's previous input tuple space operations. It begins to collect these again starting with the transaction following the l-checkpointing operation.

5. If a process fails and recovers, then it just restores the latest l-checkpointed private state and resumes execution from the state using the history.

6. Replay is performed as follows:

   - In the history, **out** and **eval** operations are ignored, because the tuple space has already reflected their execution.

   - For **rd**, **rdp**, **in** and **inp** operations, the history contains the information about which tuples were retrieved and their values. This is reminiscent of [15].

We give here an informal but, we hope, convincing argument for the following statement:

> *For a process, this strategy is guaranteed to reconstruct the private state l-checkpointed by the last committeed transaction, when the process fails.*

Suppose that a process P has committed a sequence of transactions, $T_1 \rightarrow \cdots \rightarrow T_k \rightarrow T_{k+1} \rightarrow \cdots \rightarrow T_n$, it l-checkpointed its private state at T_k, and it failed while executing T_{n+1}. When P restarts, it will restore the state l-checkpointed at T_k and resume execution from T_{k+1}. The strategy guarantees that the history for P contains all the input tuple space operations from T_{k+1} through T_n because P has committed them. Therefore, the tuple space server will replay the history for P until the commit of T_n, and so P will reconstruct the state when it committed T_n.

8 Related work

As it happens, PLinda is one of the older attempts to put fault tolerance in Linda[1]. However, others have beat us to implementation. So, this section compares PLinda with other work as if the other work were earlier.

In [2,3], 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. It supports a restricted form of transaction mechanism, called atomic guarded statements, that can execute multiple tuple space operations atomically. FT-Linda does not provide explicit facilities for resilient processes but allows a process to obtain information about its identity or to declare its identity as a specific process.

Due to replication of tuple space over processors, FT-Linda is better than PLinda for fault tolerant applications where real time responses are important. With respect to programming effort, FT-Linda requires programmers to clean up intermediate results of failed processes in tuple space explicitly. This results from the fact that no computation is allowed between tuple space operations to be executed atomically. In addition, programmers also need to design stateful processes to be resilient to failure, depending only on information about process identity. In contrast, PLinda automatically cleans up intermediate results of failed processes and supports process-private logging operations for resilient processes.

A PLinda worker process incurs transaction overhead to commit, because it must do a write to stable storage, but does not require the overhead of ordered atomic broadcast. The write takes longer than the broadcast done by FT-Linda, but PLinda supports group commit for reducing this overhead. A checkpoint-protected tuple space does not even require each transaction to write to disk. In addition, PLinda allows processes to go on to other work while their transaction commits occur asynchronously. In contrast to a normal transaction processing system, the worker need not wait until the commit completes.

Power failures are the most frequent kind of hardware failures. They usually cause all processors to fail. FT-Linda keeps 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 depend on disks. FT-Linda advocates might argue that non-volatile RAMs are just around the bend. PLinda could use those to make our log writes faster.

In soon-to-be-published work, Silva, Veer and Silva present an idea of making the master/worker paradigm fault tolerant in Linda. They implemented the idea in a commercial product called the Tuple Space Library or TsLib for short.

In their design, tuple space is on the same host as the master. Periodically, as a computation proceeds, the system checkpoints both the master process state and the tuple space state after quiescing the system. The workers execute transactions, writing to tuple space only at commit time. TsLib worker processes are assumed to be stateless.

TsLib is a close cousin of PLinda with respect to the fundamental design approach, but its features are still primitive compared to those in PLinda. First, TsLib is aimed only at a single programming model, the master/worker programming model. Second, its tuple space does not yet support pattern matching (tuples are just byte streams tagged with integer values and accessed with tags). Finally, its transaction mechanisms are still primitive. For example, transactions must start with an `in` or `inp` operation and commit with an `out` operation.

The Piranha system is a Linda variant designed to utilize idle workstations effectively for parallel computation [13]. 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 different from that described in this paper.

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 are supposed to "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 will probably make 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 process-private logging in PLinda.

Some other fault tolerance work has proposed useful ideas. Xu and Liskov proposed a protocol to replicate tuples and to maintain consistency of replicas, despite processor failures[16]. [12] discussed the performance and availability issues concerning replication techniques for tuple space. [9] also presented a protocol to relax the consistency of tuple space replicas to improve performance. [11] proposed a scheme based on checkpointing the processes and logging all the tuple space accesses.

9 Conclusion

In this paper, we presented PLinda 2.0, an approach to fault tolerant Linda. We showed how transactions augmented with process-private logging can make arbitrary Linda programs fault tolerant. In addition, we tried to demonstrate that these mechanisms are simpler to program in than those supported by the other fault tolerance work on Linda. One just adds `xstart` and `xcommit` operators and process-private logging operations.

We also explained how such features as process-private logging mechanisms, a checkpoint-protected tuple space and non-blocking transaction commits can reduce runtime overhead. Based on these design techniques (most of which, in fact, are well-known in database contexts), we argued that fault tolerant mechanisms in PLinda are about as cheap as other mechanisms such as replicated tuple space with ordered atomic broadcast, and sometimes better.

Our prototype PLinda 2.0 system currently runs on a network of SUN Workstations, but implementation is close to completion. The current design is based on a client-server architecture where a single server process handles requests from multiple client processes.

Once implementation is completed, we will focus on experimenting with applications and collecting performance results. Based on them, we will then decide how to extend our current implementation.

As we said at the beginning, we make these brave assertions without including the data in this report. Send us mail and we will send you our results.

References

[1] B. Anderson and D. Shasha. *Persistent Linda: Linda + Transactions + Query processing*, volume 574, pages 93–109. Springer-Verlag, June 1992.

[2] 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.

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

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

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

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

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

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

[9] 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.

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

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

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

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

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

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

[16] 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.