Checkpointing Transaction-based Distributed Shared Memory

by

Qiusheng Chen

M.S., Brown University, 1999

Thesis

Submitted in partial fulfillment of the requirements for the Degree of Master of Computer Science in the Computer Science Department at Brown University

May 1999

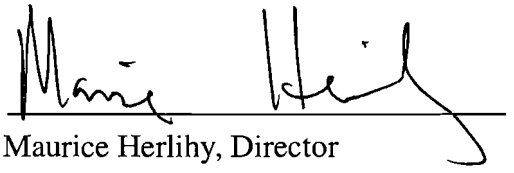
This thesis by Qiusheng Chen

is accepted in its present form by the Computer Science

Department as satisfying the

thesis requirement for the degree of Master of Computer Science

Date 19 MAY



Maurice Herlihy, Director

Approved by the Graduate Council

Date _____

_____

Peder J. Estrup
Dean of the Graduate School and Research

Acknowledgments

I am grateful to Professor Maurice Herlihy, my advisor, for his direction, criticism, constant help and support throughout this work. He gives me many suggestion to help me to finish this project.

Checkpointing Transaction-based Distributed Shared Memory

Qiusheng Chen
Computer Science Department
Brown University
Providence, RI 02912
cq@cs.brown.edu

Abstract

The Aleph Toolkit is a collection of Java packages that implements a simple, platform-independent distributed shared memory[MPH]. But currently it lacks of failure recovery mechanism and is not suitable for long-running distributed applications. The objective of this project is to make Aleph to be a fault-tolerant distributed shared memory computing platform.

Based on the combination of the concepts of transaction and checkpointing, this project implements a user-level, consistent and independent checkpointing scheme. Based on the checkpoints made, our approach can guarantee recovery from at least single node failure, and it can detect whether the whole computation can be restored to a consistent state in the event of multiple failures.

Keywords

DSM, transaction, distributed transaction, checkpointing, primary coordinator, backup coordinator, failure(detection), failure recovery

1 Introduction

Distributed shared memory(DSM) implements a shared memory programming interface on systems without any hardware support for shared memory(e.g., distributed memory multicomputers, networks of workstations). Nowadays networks of workstations exist in many organizations, and their number is growing rapidly. Thus, it is very attractive and cost-effective to utilize this frequently wasted resource.

Although Distributed shared memory provides a useful paradigm for developing distributed applications. As the number of processors(nodes) in the system and running time of the distributed applications increase, the likelihood of the processor(nodes) failure also increases. The whole application need to be restarted even in the event of single node failure. If we just simply restart the whole application from its initial state, a lots of resource will be wasted(e.g., computation, network bandwidth). In order to improve the performance of a DSM system, the system must be fault-tolerant, that is to say, it can recover from a crash to a consistent state and resumes execution.

Checkpointing is the act of saving the state of a running program so that it may be reconstructed later in time[JSK]. There have been a variety of checkpointing schemes that have been used to implement fault-tolerant DSM systems. The most difficult part of a recoverable DSM is to recover the whole system into a consistent state after a crash. Generally, there are two types of checkpointing schemes: *consistent checkpointing* and *uncoordinated checkpointing*. The global consistent checkpointing approach coordinates all the corresponding distributed processes to do a checkpoint so that all checkpoints made are consistent. Uncoordinated checkpointing allows processors to checkpoint independently; when crash happens, all processes are coordinated to recover to a global consistent state and then resume execution. In some cases, some processes need to roll-back to resume to a global consistent state.

2. Related Work

Many checkpointing-based recoverable DSM schemes have been presented in the literature. Most of them are based on independent checkpointing. In this kind of approach, individual processes checkpoint independently without any coordination with the others. When crash happens, all the surviving processes must coordinated together to compute a consistent collection of checkpoints. Sometimes it will lead to domino effect while determining the recovery line. Message logging has been proposed to avoid such a problem: when a process fails, it locally recovers by restarting from its last checkpoint and by replaying the messages from the log. This approach assumes that the running of the crashed processes can be replayed. [WF] requires that each process save a checkpoint at each communication with other process. Such solution requires an unnecessarily high checkpointing frequency and checkpointing traffic and thus leads to a high overhead during fail-free operations. [NCG] presents a checkpoint protocol for a multi-threades distributed shared memory. The protocol keeps a distributed log of all shared data accesses in the volatile memory of the processes, taking advantage of the independent failure property of workstation clusters. Each process checkpoints its own state independently whenever its log reaches a highwater mark.

This protocol guarantees recovery from single node and can determine if the system can be brought to a consistent state in the event of multiple machine crashes. The drawback is that the log can be very big because it records all information about the shared object accessed since last checkpoint.

In the consistent checkpointing approach, the checkpointing action of individual processes is synchronized so that the set of checkpoints represents a consistent state of the whole system. [KCGMP] proposes a checkpointing mechanism relying on a recoverable distributed shared memory. The DSM's consistency protocol is extended so as to ensure that the recovery copies of each shared page always exist. One benifit of this scheme is that it uses main memory instead of disk to store recovery data and thus leads to a performance gain when using a high speed communication network. Since this solution ensures that at least two recovery copies of the page are stored in different nodes, it can recovers from single node failure. [CMP] introduces a consistent checkpointing protocol that relies on the integration of checkpoints within synchronization barries already existing in applications, which avoids the need to introduce an additional synchronization mechanism. The advantage of this scheme is that performance degradation arises only when a checkpoint is being taken; hence, the programmers have the flexibility to adjust the trade-off between the cost of checkpointing and the cost of longer rollbacks by adjusting the time between two successive checkpoints.

Generally, we see the currently implemented Revoverable DSMs(RDMS) have the following drawbacks. Firstly, all the RDSM need to save almost everything related to the process being checkpointed and this results the saving of a huge amount of data, and it is well known that the low speed of hard disk access will result in a high checkpointing overhead. Secondly, saving the whole running environment will make this kind of RDSM only suitable for homogenous platforms. Thirdly, as for the global consistent checkpointing schemes[CMP], the recovery is costly since all surviving nodes need to roll back to last checkpoint have been made. As for the uncoordinated checkpointing, [WF] results in a very high checkpointing overhead, and the failure-related surviving processes in [NCG] must wait till the failed process have been recovered. And finally, all of these schemes are so complicated that make it difficult to implement.

[HDJKK] introduced a model for structuring and coordinating multi-transaction activities. This model includes mechanisms for communication between transactions, for compensating transactions after an activity has failed, for dynamic creation and binding of activities, and for checkpointing the progress of an activity. In this model, module(transaction) is the basic units of an activity in that it's the unit of program composition, execution, atomicity, compensation and checkpointing. Checkpointing a module gurrantees that a completed step will never have to be rerun due to a system failure.

3 Motivation

A *transaction* is unit of program execution that accesses and possibly updates various data items[SSK]. Transactions make it easy to meet data integrity requirements in the face of concurrence and failures since transactions have the following so-called ACID-properties:

- *Atomicity* Shared memory reflects all of the updates of a transaction if the transaction commits; but reflects none of these updates if the transaction aborts(e.g. due to a crash). A transaction ends by either committing or aborting -- there are no other possibilities.
- *Consistency* Execution of a transaction in isolation preserves the consistency of shared data.
- *Isolation* A transaction's updates are accessible only after the transaction has committed.
- *Durability* After a transaction completes successfully, the changes it has made persist, even if there are system failures.

Transaction-based Distributed Shared Memory is an interesting topic to investigate in that it is the natural research area based on the combination of two interesting topics: transactions and data sharing. Combining data sharing and transactions offers the benefits of both, and the combined system provides opportunities for optimizing performance. Moreover, another merit is that implementing such a prototype system can be feasibly easy than other DSM system. In addition, the concept of checkpointing is tightly related to the implementation of transaction's *Durability* property.

In this project, we present a recoverable Distributed Shared Memory scheme that integrates transaction and checkpointing. We implement a user-level, consistent but independent checkpointing scheme, which has the potential to gain high performance. Based on the checkpoints made, our approach can guarantee recovery from at least single node failure, and it can detect whether the whole computation can be restored to a consistent state in the event of multiple failures.

Java is a platform-independent language that runs on virtually every commonly used, modern operating system. It provides a rich set of packages for developing both sequential and distributed applications.

Perhaps the most powerful Java feature for fault tolerant programming is object serialization, which can read and write entire objects to and from streams. During object serialization, a snapshot of the object, along with every directly and indirectly referenced object, is made and converted into a stream of bytes. This snapshot records the value of every non-transient-non-static variable in the involved objects. The stream of bytes can later be used to recreate an equivalent object. Such a powerful facility makes it feasible for application programs to make their own checkpoints. However, a serious restriction is that the java class *Thread* is not *serializble*. This means that the position of execution within a java program can not be captured directly with object serialization. One approach to this problem is to modify the Java Virtual machine to dump the entire state of a Java application on demand. But this can only be implemented by sacrificing Java's attractive *portability* feature because a number of JVM would have to be modified. Another reason is that one checkpointed thread might not be resumed on a surviving machine with different JVMs.

Although Java does not provide sufficient facility to support completely application-transparent fault tolerance. But we believe that with the use of appropriate fault tolerant programming design patterns, Java provides a viable platform for developing high quality, portable, fault-tolerant applications. Obviously this kind of application need to be designed carefully and it will increase the burden of programmers.

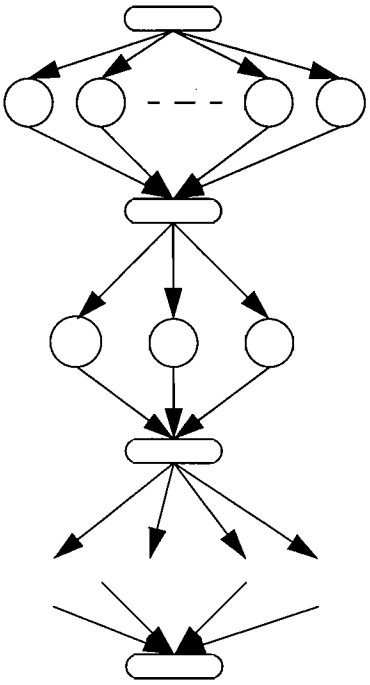
We consider a pattern for applications with non-trival, in-core state so that *object serialization* can be used to capture application state, and upon a crash/stop and restart, the application can proceed consistently. The basic approach of this pattern is to use object serialization to periodically capture the state of the computation. When the application is restarted due to a failure, its state can be loaded from an object byte stream that has been checkpointed.

In order to make both the main thread and distributed threads to be checkpointed, we let them implements the java.io.Serializable interface.

4 Computation and Programming Model

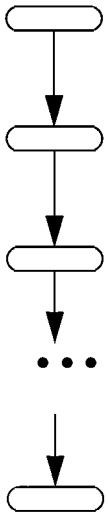
In our model, a computation(we call the thread that starts the computation main thread/process) is composed of a sequence of steps(this assumption is consistent with a set of distributed computation applications), while each step consists of a number of parallel running threads distributed over different nodes, we call them distributed threads, and each of which consists of a sequence of transactions. All these threads(including main thread and distributed threads) communicate via global(shared) objects.

The system we consider is composed of a set of workstations(or PCs) connected with an interconnection network. Two machines are selected an coordinator to run the main process, one of the coordinators is called primary coordinator, the other backup coordinator. We called other machines taking participate in the computation as client nodes. The two coordinators are responsible for the whole computation and node failure recovery. The primary coordinator machine dispatches distributed processes (or threads) to the client machines and run there, while the backup coordinator always mimics the activity of the primary coordinator. One client machine can accommodate more than one thread.



: Distributed thread

Figure 1 Main thread



: Transaction

Figure 2 Distributed thread

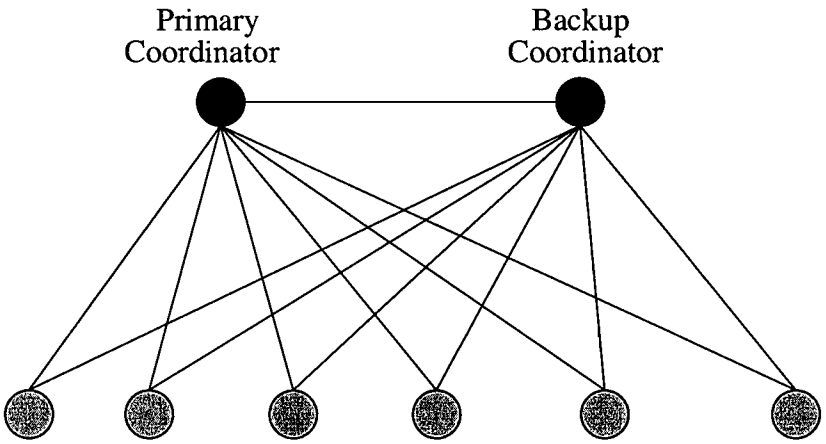


Figure 3 System Model

The following pseudo code are examples of the structure of main thread and distributed thread.

MainThread() { // constructor of the MainThread

```
                next_step = 0;
                join = new Join();
               }
void run()   {
             if (next_step == 1) {
                    initialization of this step;
                    spawn some distributed threads P1, P2, P3, ... Pn;
                    send P1, P2, P3, ... Pn to Client Nodes and run there;
                    next_step = 2;
                   }
             if (next_step == 2)
                   { join.waitFor(this);  next_step = 3; }
                  ......
           }

DistributedThread {
    DistributedThread()  { CurrentTransactionSeq = 0; }
    void run () {
          while (CurrentTransactionSeq > MaxTransactionSeq)   {
             switch (CurrentTransactionSeq)
              { case 0: Initialization;
                        break;
                case 1: Transaction_1.begin(); ...
                        Transaction_1.commit();
                        break;

                case i:  Transaction_i.begin(); ...
                        Transaction_i.commit();
                        break;
                } // end of switch
             } // end of while
        } // end of run
}// end of class DistributedThread
```

Backup Coordinator can be viewed as a proxy of the Primary Coordinator. The Backup Coordinator try to keep the same state of that of the Primary, but they should not have the same activities. For examples, if both of the Primary Coordinator and Backup Coordinator have the same activities and both of them spawn distributed threads, it will double the number of the distributed threads that should be generated, and the result of the computation will generate wrong results. In order to avoid such kind of unwanted side effect, our system has been carefully designed to try to force the two coordinators to collaborate to run the com-

putation correctly. Our system guarantees that when the Primary Coordinator is alive, the Backup Coordinator will have no side effect to the running of the computation, only after the Primary Coordinator is found dead by the Backup Coordinator, the Backup Coordinator takes over and does the recovery from failure.

The two coordinators communicate via a Join object, which is initialized by the main thread firstly executed at Primary Coordinator. When the main thread runs to **Join.waitFor**() function, it automatically sends the main thread to the Backup Coordinator and runs there. Due to the event mechanism that will be discussed later on, both the primary main thread and the backup main thread can receive the same set of signals from distributed threads executed on client nodes so that both the backup main thread and primary main thread keep the same status of the computation.

After passing Join.waitFor()(that means one step of the computation has finished successfully), the main thread running on the Primary Coordinator proceeds as usual, while the backup main thread running on the Backup Coordinator is suspended. When the Backup Coordinator receives another main thread that the Primary Coordinator want to backup, it means that the suspended main thread is obsolete and should be discarded; When the Backup Coordinator finds that the Primary Coordinator is dead, the suspended main thread(if it has been) is reactivated and becomes the primary main thread. All we want to do is that we want to guarantee at least one copy of main thread is running

The following pseudo code shows how the Primary Coordinator and the Backup Coordinator collaborate via a **Join** object.

```
Join.waitFor(MainThread.this) {
    // if the current thread is the primary MainThread, it backup itself on the
    // BackupCoordinator
      if (current_node is PrimaryCoordinator)
            send MainThread to Backup Coordinator and run there;
      event.waitFor();
    // After passing the waitFor()
    // if the Primary is alive, this backup MainThread will be killed
    // otherwise, the backup mainThread will take over
      if (current_node is Backup Coordinator) {
            suspend();
}
```

In order to make a transaction to be persistent, we need checkpoint the global objects updated by this transaction and checkpoint the distributed thread which initializes this transaction so that after a failure happens, the whole computation can be recovered to a consistent state.

5 Global Objects and Memory Coherence Model.

As mentioned before, all the threads belonging to one computation communicate via global objects.

Each global object is represented by:

- unique id
- version, the version field is helpful for figuring out the newest global object during a recovery

In our model, global objects can be distributed over nodes since both the computation and global objects distributed over networks can help to avoid potential bottleneck.

A memory coherence model is best described as a contract between the system and the application program[NCG]. How to implement a memory coherence protocol is an important subject in DSM. A number of memory coherence protocol have been proposed to implement DSMs. Entry consistency memory conherence protocol was introduced by Midway[BZS]. In this kind of protocol, synchronization accesses are divided into acquire operation and release operation. An acquire is performed to gain permission of access to certain shared data, while a release operation relinquishes this right. Entry consistency requires that each shred data object be associated with a synchronization object. When a synchronization object is acquired, only the data associated with that synchronization object is made consistent.

In order to ensure consistent data sharing and at the same time maximize concurrence, we need to enforce concurrent read and exclusive write to global objects. In our entry consistency protocol, all write accesses must be included between *acquire_write_right* and *release_write_right* operation pair, and all write accesses must be included between *acquire_read_right* and *release_read_right* operation pair. All the acquire operations issued by multiple processes to the same global object are synchronized, i.e, a process is blocked until its acquire operation completes(need not repeated query). The updates only become visible in another process when that process acquires the right to access the global object, while some older version of the same global object resides in other nodes is not updated if those node don't have the access right or need not access it. That is to say, after a process get the right to access to certain global object, it always get the latest version of that global object, and at the same time all other potential unnecessary updates are not made. Entry consistency protocol minimizes the communication among processes by delaying the propagation of modifications to shared data. We implements entry consistency protocol because we want to minimize the global objects updating propagation, which is very expensive in software DSM.

6 Node Failure Detection

We assume that all nodes have fail-stop property: each node stops its execution as soon as a failure is detected, without interfering with the other nodes. This characteristic ensures that the state of failure-free nodes are not altered by the failure of a faulty node.

We also take care of the situation in which surviving node might be blocked waiting for messages from failed nodes that are lost due to failure.

The main goal of this project is to deal with single node failure case. It's very rare that two machines fail at the exact same time, so our assumption that there's only one workstation fails during a short interval is reasonable.

We assume that there exists a low-lever underline failure detection protocol. In our simulation, we let a node be unable to send out message to simulate the failure of that node. If a node A want to die, it broadcasts message "A is dead" to all other nodes to simulate the failure detection protocol, then it becomes really dead: it will not send out any message to other nodes.

7 Event and Event Model

Events are implemented to synchronize the action of threads that may be distributed over different nodes.

7.1 Aleph's Event Model

The Event model defined in the Aleph is below(figure 4):

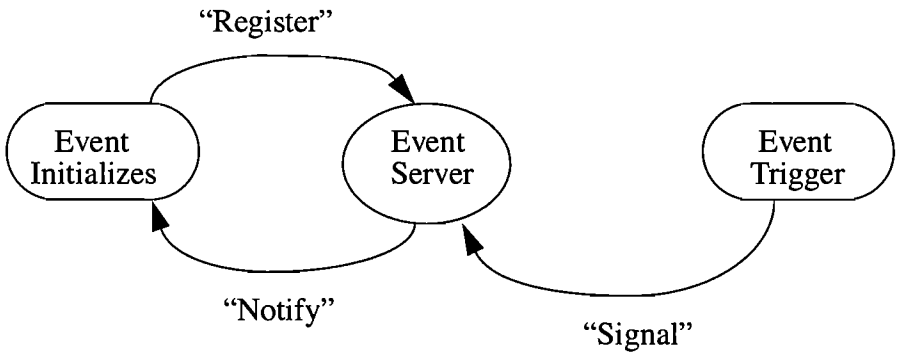


Figure 4 Aleph's Event Model

- An Event Initializer(a thread) intializes an event *e*,
- Event *e* is sent to the an Event Server to register that event;

- When the event happens on one node(we call it the Event Trigger, which may be the same as the Event Initializer, or may be not), the Event Generator informs *e*'s Event Server that event *e* has happened;
- The last step is that the Event Server informs the Event Initializer that event *e* has happened.

In Aleph, the default Event Server of event *e* is the Event Initializer of event *e* so that "register event" message need not be transmitted across network and thus it can decrease network traffic and save message transmission some time.

This model works perfect for failure-free environment but is not suitable for node failure cases. For example, event e_1's event server is node A, and threads running on node B_1, B_2, ..., B_n are listening to the happening of event e_1. At time t, a event trigger ET_{e1} sends a "signal" message to e_1's event server A informing the happening of e_1 and moves on. After receiving the "signal" message, event server A becomes dead, then all the listener on nodes B_1, B_2, ..., B_n will never receive "Notify" message from the server A and they might be blocked forever.

7.2 Event Classification

One goal of the implementation of Distributed Shared Memory is to get high performance, and it forces us to try all means to make our implementation as efficient as possible. In order to reach this goal, we carefully divide event into 2 classes: *one-sender-one-listener* event, and *multiple-sender-multiple-listener* event.

7.2.1 One-sender-one-listener Event Model

In the one-sender-one-listener event model, the event initiator itself is the event listener. The event initiator initializes an event and becomes blocked, waiting for the happening of that event. There's only one event trigger(event sender).

One-sender-one-listener event model is useful for synchronizing the activity of two threads. For instance, sometime we want to let thread A block until thread B has finished operation P. We can realize it in the following steps:

1. thread A initializes a event e and sends it to thread B,

2. thread A(the event listener) becomes blocked

3. after thread B(the event sender) finishes operation P, it triggers event *e*

4. an event agency(can be a message regarding event *e* that is sent by thread B) wakes thread A(informs A that event *e* has happened)

We notice that the event model introduced by Aleph needs three messages: "register" message, "signal" message and "notify" message. In order to decrease the number of messages and to make our implementation more efficient, we use just two message to realize *one-sender-one-listener* event mechanism without using the event model introduced by Aleph. The event initiator(also the event listener) sends a "WaitForEvent" message to the event sender and becomes blocked; After receiving the "WaitForEvent" message", the event sender does certain operation assigned by the event initiator, and sends back a "EventHappened" message to the event listener side, and then the event listener will be waken. Figure 5 shows how this scheme works.(In our experiment, we make use of active message to realize this model.)

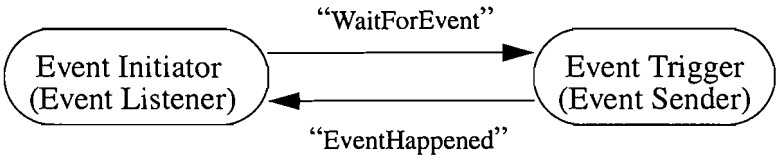


Figure 5 One-sender-one-listener event model

7.2.2 Multiple-sender-multiple-listener Event Model

In the multiple-sender-multiple-listener event model, there might be more than one event listener and more than one event sender(trigger).

Here's a specific example of the use of multiple-sender-multiple-listener event model. In our fault-tolerant model, in order to make the whole computation to be recoverable, we need to backup the main thread, and two copies of the mainthread are running on both the primary coordinator and the backup coordinator separately, these two copies need wait for a shared event *e*(a Join object) which synchronizes all the distributed threads during one step of a computation(the event *e* is initialized by the main thread running on the primary coordinator). If the primary coordinator is assigned as the event server of event *e*, the main thread running on the backup coordinator will not receive the message regarding event *e* from the event server in the case of the failure of the primary coordinator(the event server of event *e*), and the whole computation can't be continued in that situation. So Aleph's event model does not work in the situation in which nodes may fail. We need to extend the Aleph's event model so that it can recover from the failure of event server.

We want to implement a reliable multiple-sender-multiple-listener event model suitable for node failure cases on the basis of Aleph's event model. A event model is reliable means that each alive registered lis-

tener will be informed the happening of the event if the event sender has notified the event server about the happening of that event.

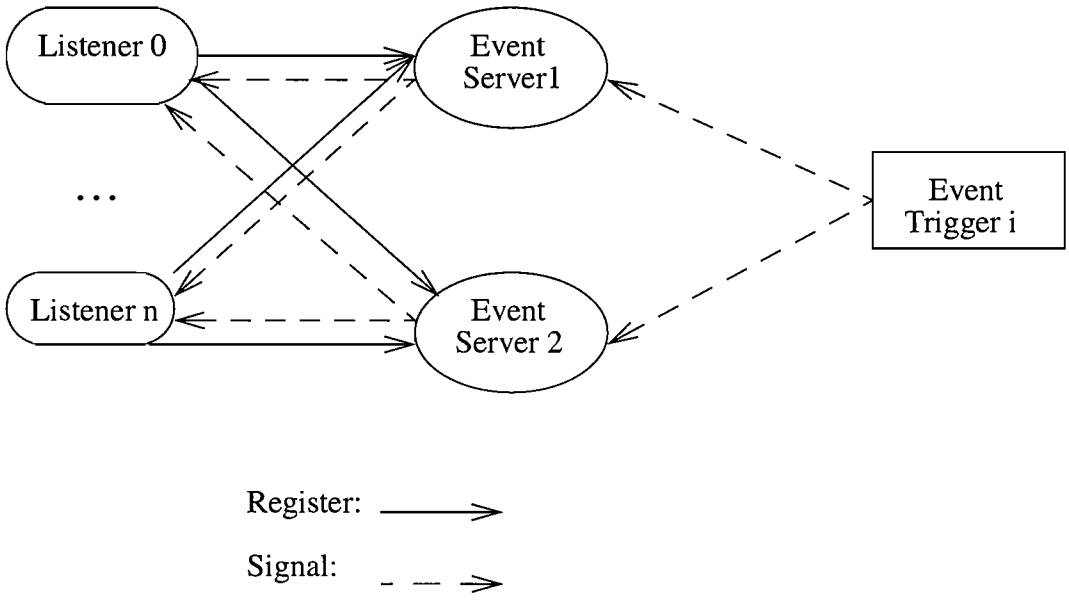


Figure 6 Reliable multiple-sender-multiple-listener event model

Multiple-listener events have two peer Event Server: Event Server1 and the Event Server2. When a thread(i.e., Event Trigger i) generate event *e*, it informs both of the two Event Servers by sending them "signal" message, then the two Event Servers inform all the listener. Since we assume that at most one node fails at any specific short interval, at least one event servers is alive, we guarantee that all the listener of this event *e* will eventually receive the message notifying the happening of event *e* generated by Event Trigger i.

Each event *e* is uniquely represented by:

- event initiator's id
- event trigger's id
- event triggering time(in case of there are multiple triggers of the same event *e* running on the same node)

Now since each event has two event servers at the same time, each listener may receive two copies of the same event(if both the two event servers are alive) and may make the system to be inconsistent. In order to remove this unneeded side effect, we let each event listener keep a list recording the events it has received, and it simply ignores the event that has been received.

In our experiment, event *e*'s initiator is chosen to be one of the event server, the other event server is selected from the initiator's neighbor. As for a Join event, It's very natural to select the primary coordinator and the backup coordinator as a Join event's event server.

8 Distributed Transaction and Checkpointing Scheme

Checkpointing is the act of saving the state of a running program. We notice that a consistent checkpointing made at certain time *t*, should make both the distributed thread itself and *all* the global objects updated by this distributed thread from last checkpoint to *t* to be *durable*. In addition, the checkpoint made should also has atomic property because if the above checkpointing procedure is not *atomic*, the computation may become inconsistent due to a failure.

In a DSM system, a distributed transaction accesses and updates a set of shared objects distributed over the network of workstations. Because *transaction* has the atomic-consistency-isolation-durability property, it's very nature to introduce distributed transaction concept to implement this procedure.

We force each distributed thread to implement the java.io.Serializable interface so that each distributed thread can be taken a snapshot(only variables, no run-time information) at any time.

As mentioned before, a distributed thread consists of a sequence of transactions. If we checkpoints a distributed thread at the end of every transaction, it's easy to see this checkpoint is consistent.

In order to commit a distributed transaction, all the site related to this transaction must agree on the final outcome. Among the simplest and most widely used commit protocols is the *two-phase commit protocol*(2PC). The disadvantage of the two-phase commit protocol is that it needs a lot of messages among nodes to ensure the *atomic* property of distributed transaction. For example, successfully committing a distributed transaction with n participating nodes needs 2*(n-1) messages.

In order to gain better performance, we checkpoints in the main memory instead of hard disk so that we can avoid the long latency of disk access. Because we mainly want to deal with single node failure, it's possible we can realize durability property only using main memory with active backup strategy.

Each node has its own local transaction manager. Local transaction managers cooperate to ensure the ACID properties of distributed transactions.

A checkpointing message contains all the updated global objects in the committing transaction and the current distributed thread.

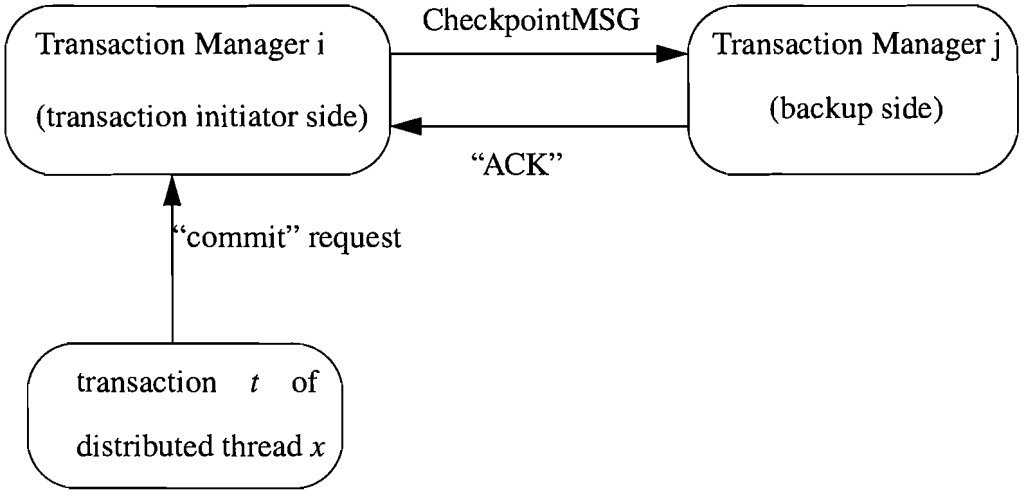


Figure 7 checkpointing scheme

The following lists the procedure to do a checkpointing:

1). The committing transaction sends a "commit" request to its local transaction manager(say i), and becomes blocked;

2) The local transaction manager i makes a corresponding checkpointing message and saves a copy of that message marked with "committing" in a checkpoint table(for fault-tolerant use), then it sends this message to transaction manager j on another node;

3) After receiving a checkpointing message from transaction manager i, transaction manager j saves the checkpoing message to a checkpointing table, and sends back a "ACK" message to inform the committing thread that the transaction has been successfully committed.;

4) Upon receiving the "ACK" from transaction manager j, transaction manager i changes the corresponding checkpointing message copy from "committing" to "committed" status.

5) Corresponding blocked thread is awaken by local event manager

As mentioned before, preserving the amotic property of transaction is difficult and needs many messages to synchronize various nodes participating the same transaction. In our approach, all the checkpointed information is integrated into one checkpointing message and is shipped to another node. Because we also assume that at most one can fail in an interval, we can guarantee that our scheme preserve the atomic and durability properties. We will prove that in the correctness section.

If we ignore the effect of message size, this scheme dramatically decreases the message complexity from 2*(n-1) to 2. Even we consider the effect of message size, we can imagine our approach is more efficient than the two-phase commit protocol since the cost of our approach is just related to the communication between two nodes while cost of the two-phase commit protocol is related to the communication between one node and other (n-1) nodes.

In order to avoid any potential bottleneck and make our checkpoint scheme more scalable and realize load balancing, we let each node take part in the checkpointing procedure with equally opportunity. We view the whole network of workstations participating computation as a directed ring(clockwise or anti-clockwise). Without loss of generality, we assume it's a ordered clockwise ring, which is just logically ordered. Assume there are n nodes, node i's right neighbor is (i+1)(mod n). And each node checkpoints on its right neighbor node. Figure 8 shows the case of 5 nodes: node 1 checkpoints on node 2, node 2 checkpoints on node 3, ..., and node 5 checkpoints on node 1.

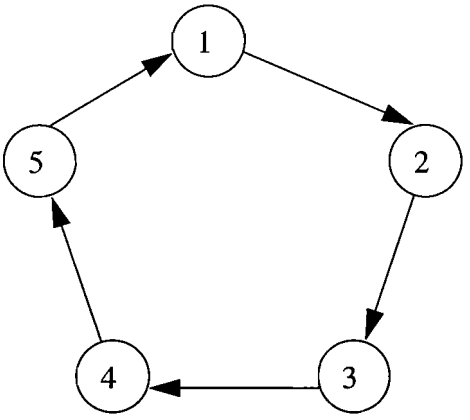


Figure 8 scalable checkpoint scheme

9 Failure Recovery

There are three kinds of nodes in our system model: primary coordinator, backup coordinator and client nodes. Accordingly, we need to deal with three kinds of node failure: primary coordinator failure, backup coordinator failure and client node failure.

Here's the basic strategy for recovering from a node failure. When a client node failure is detected, the Primary Coordinator is responsible for the recovery and the Backup Coordinator just skip it; When the Backup Coordinator fails, which is detected by the Primary Coordinator, the Primary Coordinator then finds another available node from client nodes and assigns it as the new Backup Coordinator, and recovery it; When the Primary Coordinator fails, which is detected by the Backup Coordinator, the Backup Coordi-

nator claims itself to be the new Primary Coordinator; then the new Primary Coordinator find another available node from client nodes and assign it as the new Backup Coordinator, and continue to do the recovery work. All algorithms will be discussed in detail below.

In order to recover from a failure, we need to:

- recover the global objects affected by the crashed node
- recover the distributed threads running on the crash node
- continue the main thread if the primary coordinator fails
- continue the backup work preparing for recovering from future failure

9.1 Client Node Failure

When a client node failure is detected by the Primary Coordinator, the Primary Coordinator is responsible for the recovery; When a client node failure is detected by the Backup Coordinator, and at the same time if the Primary Coordinator is alive, the Backup Coordinator just skips the client node's failure.

Algorithm Recovery from client node failure

1) recover the global objects owned/occupied by the failed client node

2) move the global objects owned by the failed client node to other available client nodes

3) inform all client nodes of the home change of the global objects owned by the failed client node

4) restart distributed threads running on the failed client node on other available client nodes

5) unblock those blocked threads waiting for messages from failed node

6) active duplication in case of future failure

Because our checkpointing scheme guarantees that there alway exists at least one copy of each global object and each distributed thread in different nodes after single node failure, we will prove our scheme can recover from any single client node failure.

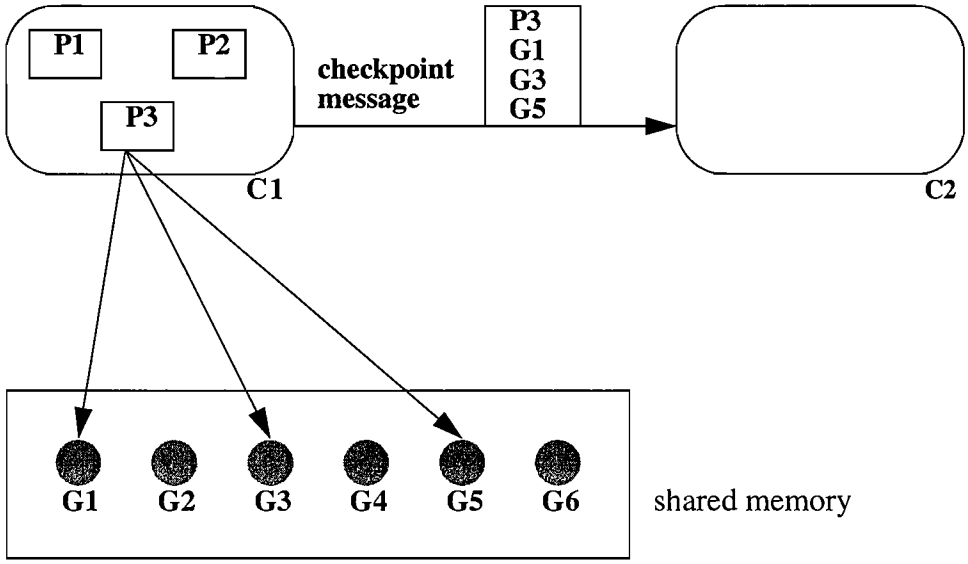


Figure 9: P3 checkpoints

We make use of Figure-- for the proof. Assume client node C1 crashes at some time *t*, C1's checkpointing neighbor is node C2. Assume distributed thread P3's current transaction is T, which accesses and update global objects G1, G3 and G5. T can be in any of the following status: *active*, *committing*, *committed*.

If T is in *active* state, since P3 has crashed due to the failure node C1, that means transaction T has been aborted automatically. Those Global objects occupied by T have not yet been updated by T because T has not reach *committed* state. The only problem is that other threads might be blocked, waiting for the global objects occupied by P3(shared read or exclusive write). We solve this problem by the following way.

For each global object G,

1) figure out G's version just before T get access to G;

2). if node C_i(i<>1) is the owner of G, let C_i remove C_1's access authority, and grants the access authority to other waiting threads;

3) if C1 is the owner of G, select another node C_i(i<>1) as the new owner of G, broadcasts this change to all other nodes.

If T is in *committing* state, there are two possibilities: C2 can receive the checkpoint message, the checkpoint message is lost due to that failure and C2 will not receive it. In the first case, both the distributed thread and the global objects updated by that distributed thread are saved in node C2, so the computa-

tion is consistent. In the later situation, neither the distributed thread nor the global objects updated by that distributed thread will be saved in node C2, so the computation is still consistent.

If T has been in *committed* status, which means C2 has received the checkpoint message from C1 since T has got acknowledgment from C2. At this time the computation is consistent. If T has not yet released all global objects during this transaction successfully, the access authorities will be removed by our recovery procedure mentioned before.

All the failed distributed threads running on failed node C1 will be restarted on other alive nodes. In order to realize load balancing, we dispatch the failed distributed threads equally to the surviving nodes.

Another problem is that there might be some blocked threads due to node failure. Generally, those blocked threads firstly send some messages to the failed node, and wait for the response message from that node. If the failed node crashes before sending response message, some threads waiting for the response will be blocked forever and it results in a deadlock. We solve this problem as follows: all messages that may block a thread are buffered in a table that stores all unacknowledged messages before being sent out, after a response is received, the corresponding message is removed from that table; After a failure happens, we re-direct all unacknowledged messages sent to the failed node before and send them to a a right node. Here we assume that all response messages sent from the failed node have been received before the current node knows of the failure. For example, a thread *t* sends out a message to a global object G's home H, asking for the access authority of G and is blocked; then H crashes and G's home is moved to H', we can switch that message to H' and sends it to H'. And eventually, thread can get access authority from H'.

9.2 Backup Coordinator Failure

The difference between a coordinator node and a client node is that a copy main thread runs on a coordinator node(a coordinator itself also a client node). So when a coordinator crashes, we need to do some extra work to recover the main thread except doing the recovery work for a client node.

After the Backup coordinator crashes(it is noticed by the Primary Coordinator), the Primary Coordinator choose a available client node as the new coordinator, and backups the necessary checkpoint table containing the checkpointed global objects and checkpointed distributed threads and runs the main thread from its last finished step on the new Backup Coordinator. Finally, the primary coordinator informs all other nodes about the change of backup coordinator.

The following is the algorithm for recovering from the failure of Backup Coordinator:

Algorithm Recovery from Backup Coordinator failure

1) find a light-loaded client node to be the new Backup Coordinator

2) run the backup mainThread on the new Backup Coordinator

3) inform the change of Backup Coordinator to all client nodes

4) active duplication in case of future failure

The reason that we choose a light-load client node as the coordinator candidate is that we want to realize load balancing by trying to make all the distributed equally distributed over the whole networks of work-stations.

9.3 Primary Coordinator Failure

After the Primary coordinator crashes(which is detected by the Backup Coordinator), then the Backup Coordinator claims itself to be the new Primary Coordinator, and chooses an available client node as the new Backup Coordinator, backups the checkpoint table containing the checkpointed global objects and checkpointed distributed threads and runs the main thread from its last finished step on the new Backup Coordinator. At the same time, all threads running on the client machines wait till the recovery of the Backup Coordinator.

The following is the algorithm for recovering from the failure of Primary Coordinator:

Algorithm Recovery from Backup Coordinator failure

1) claim itself to be the new Primary Coordinator

2) find a light client node as the new Backup Coordinator

3) run the backup mainThread on the new Backup Coordinator

4) inform all client nodes about the change of Primary coordinator and Backup Coordinator to all client nodes

5) active duplication in case of future failure

9.4 Multiple-Node Failure Case

Our scheme can recover from certain multiple-node failure case. As mentioned before, the topology of our system is a directed ring and threads on node *C* are checkpointed on C's right neighbor.

A computation interrupted by node failure is recoverable if and only if the system satisfies the following conditions:

- At least one copy of the main thread survives despite the failure;
- At least one copy of any most recently checkpointed distributed thread survive despite the failure;
- At least one copy of any most recently checkpointed global object survive despite the failure;

We let both the two coordinators keep track of the distribution of all running distributed threads so that we can easily implement an algorithm to detect whether our system satisfy the above conditions after multiple-node failure happen.

First we notice that if the two coordinators crash at the same time, the first condition can be satisfied, so the computation is not recoverable;

Secondly, in the case of any two neighbored nodes(say, node A and B) crash at the same time, if node B is node A's right neighbor(A checkpoints on B), if there exists a distributed threads running on A, neither the second nor the third condition could be met, the computation can't be recovered consistently.

Thirdly, for any active multi-sender-multi-listener event *e*, if *e*'s two event servers are dead at the same time, it's possible that some information regarding event *e* may get lost forever, so it's also unrecoverable in this case. For the sake of simplicity of implementation, we let the two coordinators to be the two event servers during our simulation.

Other multiple-node failure cases except the above three cases are recoverable.

10. Performance Evaluation

10.1 Checkpoint overhead

10.1.1 Message complexity

Compared with other recoverable schemes, our approach checkpoints independently, and there's no synchronization among distributed threads to take a snapshot. In addition, we checkpoints on main memory rather than hard disk so that we can avoid the long latency of disk access.

In our implementation of checkpoint scheme, each checkpointing message is explicitly acknowledged. It means we need 2 message to commit a transaction. Assume this computation includes n transaction, our checkpointing scheme need 2*n extra messages.

In addition, our scheme need to back up the mainThread on the backup coordinator for each step in the mainThread. This backup procedure also needs to be acknowledged. So assume the mainThread contains m step, 2*m message are needed.

Furthermore, we enhance Aleph's event model to make it to be fault-tolerant to at least one node failure by using backup event server. For each multi-sender-multi-listener event *e*, our enhanced event model doubles the number of messages that are needed by Aleph's event model.

10.1.2 Benchmark

In the networks of workstations environment, the failure is rare, and that's the main reason that we're interested in the failure-free performance comparison between scheme with checkpointing and scheme without checkpointing.

The platform we use is SunOS and JDK1.1.7B, and 2-16 workstations. The Coomunication mode is Aleph's reliable UDP. The two benchmarks are

- distributed 3-counter that contains 3 counters
- distributed algorithm that computes prime number from 2-n, each distributed thread compute the prime numbers ranging from [a,b], where [a,b] is a subset of [2,n]

Figure 10 shows the result of the distributed 3-counter benchmark, which has one distributed threads on one workstation, while each of these distributed threads contains 50 loops and in each loop, it increases(or decrease) a counter by 1. Our approach adds extra overhead ranging from 19% o 99% to the initial scheme without checkpointing. we can see when the number of workstations increases, the checkpoint overhead percentage of the overall runtime decreases.

Figure 11 shows the result of the distributed algorithm that computes prime number from 2 to 600. Our approach adds extra overhead ranging from 53% to 152% to the initial scheme without checkpointing.

The fisrt benchmark performs better than the later one because in the second benchmark, the overall messages and time with checkpointing consists of much more checkpoint messages and checkpointing time(by percentage) than that of the 3-counter benchmark..

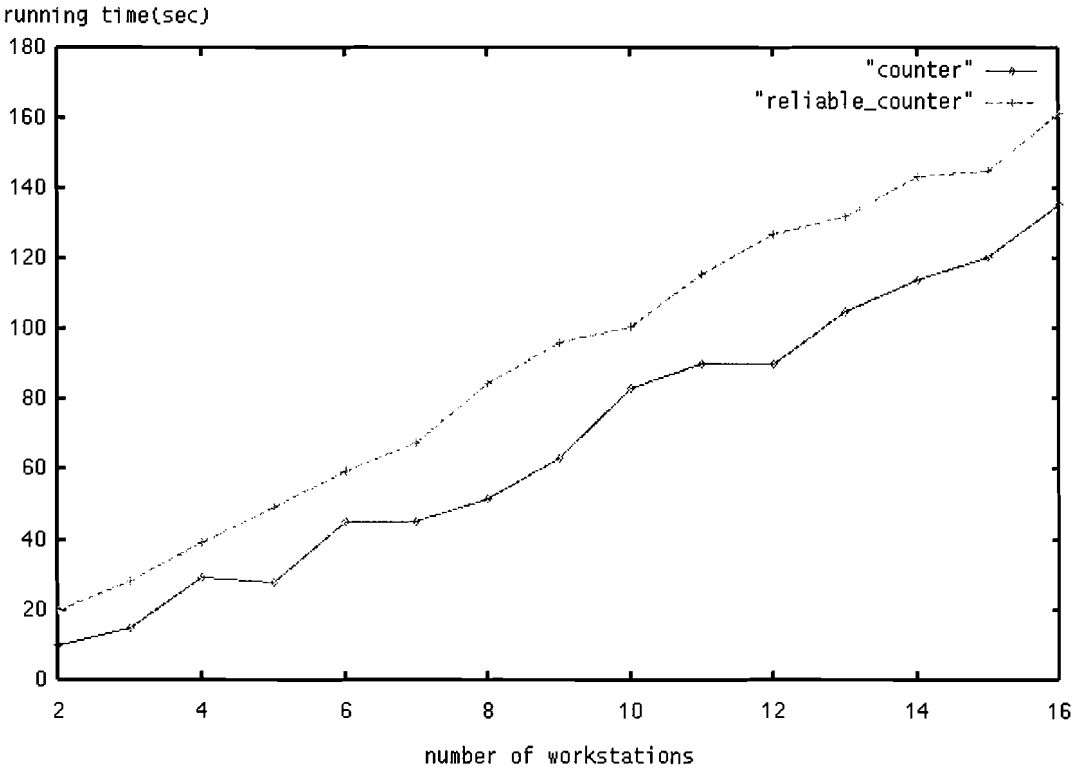


Figure 10: distributed 3-counter

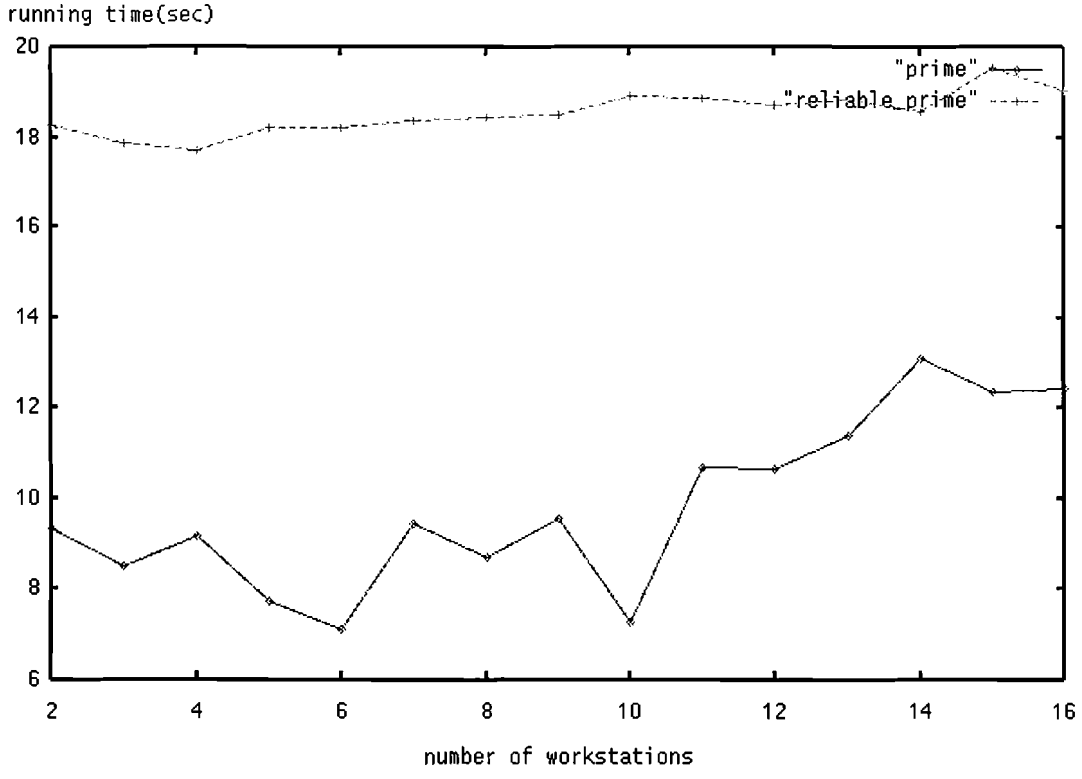


Figure 11: distributed algorithm computing primes from 2 to 600

10.2. Recovery overhead

First we identify those nodes that are forced to cooperated to do the necessayr recovery job. Assume node A crashes.

If A is a client node, for any global object g_i that was currently "write" held by certain thread running on node A, all node are synchronized to compute the most recent version of g_i before crash. Assume C is the primary coordinator. C collects a message regarding g_i from all other surviving node and then computes the most recent version of g_i. If there're m surving nodes, m message are needed to recover global object g_i.

If A is a coordinator, more job is needed: we need to backup the mainThread to a new backup coordinator, and broadcasts this change to all other nodes.

As mentioned before, we restart the failed distributed threads running on A from its backup node which stores the most recent version of the failed threads, so no message between different nodes is needed to do it.

In addition, if there're k event servers residing on A, we should backup them on other surviiving node, which needs another k messages.

Now we identify those distributed threads that will be affected by a failed node.

- If a thread t is waiting for a global object currently held by a thread running on a failed node, t will be blocked until the recovery
- If a thread t is waiting for the checkpointing acknowledgement from a failed node, t will be block until the recovery

We reactive those blocked threads by asking some agancy to resend a "want global object"(case 1) or another checkpoint message sent to a new backup node(case 2). Either case needs one extra message.

We notice that those unaffected threads will not be interrupted by this failure and run as before. It's obviously that our approach minimizes the overhead for recovery.

11. Conclusion

The job we've done shows our approach can recover from any single node failure, and can detect whether the whole computation can be restored to a consistent state in the event of multiple failures.

Our approach can be easily extended to implement k-resilient system if we

- assign $(k+1)$ event server to each multi-sender-multi-listener event e, and
- checkpoint each committing transaction on another k node

This is because in the case of k nodes crash at the same time, we can guarantees that there alway exists at least one copy of each global object and each distributed thread survive after single node failure so that the running computation can be recovered from any k-node failure.

11. References

[BZS] B.N. Bershad, et al, "The Midway Distributed Shared Memory System". In Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93), pp. 528-537, February,1993.

[BSS] A.Beguelin, E.Seligman, and P.Stephan. "Application level fault tolerance in heterogeneous networks of workstaitons", Journal of Parallel and Distributed Computing, Sep. 1997

[CA] C. Amza, A.L, et al, "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer

[CMP] Gilber Cabillic, Gilles Muller, and Isabelle Puaut, "The performance of Consistent Checkpointing in Distributed Shared Memory", 14th Symposium on Reliable Distributed Systems, Germany, September 1995

[HDJKK] Hector Garcia-Molina, et al, "Coordinating Multi-Transaction Activities"

[JSK] James S. Plank, "An Overview of Checkingpoint in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance", Technical Report UT-CS-97-372, University of Tennessee

[KCGMP] Anne-Marie Kermarrec, et al, "A Recoverable distributed shared memory integrating Coherence and Recoverability"

[KW] K.L. Wu and W.K. Fuchs, "Recoverable distributed shared memory: Memory coherence and storage structures", IEEE Transactions on Computers, 34(4):460-469, April 1990

[MPH] Maurice Herlihy, "The Aleph Toolkit: Platform-Independent Distributed Shared Memory(preliminary report)", Computer Science Department, Brown University

[NCG] Nuno Neves, Miguel Castro and Paulo Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory", 13th ACM Symposium on Principles of Distributed Computing, August 1994

[SSK] A. Silberschatz, H.F.Korth, and S.Sudarshan, "Database System Concepts", Third Edition, McGill-Hill Press

[SZ] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory", IEEE Computer,23(5), May 1990, pp. 54-64

[YC] Weimin Yu&Alan Cox, "Java/DSM: A platform for Heterogeneous Computing".